M.Tech. Dissertation

Debugging of Optimized Code

Submitted in partial fulfillment of requirements
for the degree of
Master of Technology

By
Zoheb A. Vacheri
Roll No : 00305011

Under the guidance of

Prof. D.M. Dhamdhere

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai, 400 076
January 2002

Dissertation Approval Sheet

This is to certify that the dissertation titled

Debugging of Optimized Code

By

Zoheb A. Vacheri

(00305011)

is approved for the degree of **Master of Technology**.

Prof. D.M.Dhamdhere

(Guide)

Internal Examiner

External Examiner

Chairperson

Date :

Acknowledgment

I take this opportunity to express my sincere thanks to my guide **Prof. D.M. Dhamdhere** for his support,encouragement and advice. His constant guidance was invaluable in the realization of this project.

Zoheb Vacheri
IIT Bombay
February 13, 2002

Abstract

We look at various techniques for debugging of optimized code by value recovery of variables through the use of execution trajectories. Execution trajectories are traditionally unbounded in size. A technique for value recovery using construction of a bounded size execution trajectory is then studied in detail and algorithms to perform code instrumentation are specified. The design of a debugger that uses execution trajectories to provide for value traceback and debugging of optimized code is also discussed.

Contents

Chapter 1

Introduction

Compilers perform optimizations when translating a source program into an object program. These optimizations introduce different types of changes in the object code when compared to the source program. Variables are removed, introduced or reused. Code is inserted, deleted, reordered etc. This disturbs the mapping between a source program and object code, which makes debugger functionality difficult in many ways.

We will be concerned with the design of a debugger for optimized programs. The debugger has to handle the problem of currency of variables in the program. The currency problem is defined as follows : An expected value of a variable at some point in a program is a value the user expects the variable to have at that point. The actual value of a variable is the value which the location assigned to the variable contains at that point during execution. A variable is current at a point if its expected and actual values are same at that point during every execution of the program. A variable is noncurrent if its actual and expected values are different at that point for every execution of the program, and is suspect if its expected and actual values are same for some executions and different for other executions of the program. The problem of finding whether a variable is current, noncurrent or suspect is called currency determination. The currency determination problem arises when assignments to program variables are moved during optimization and when partially dead assignments are eliminated.

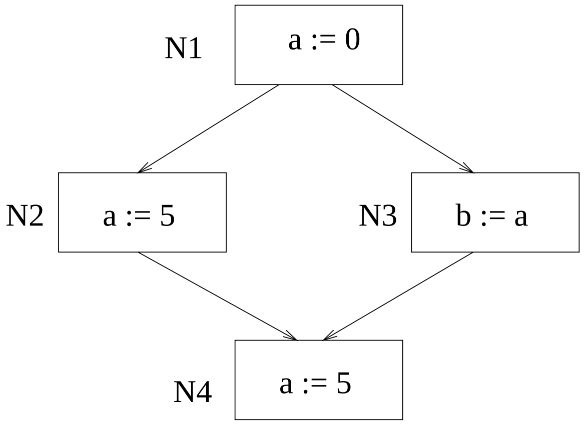

Figure 1.1: Insertion and deletion of computations during optimization.

A debugger is expected to report the expected value of a variable to the user. Hoisted assignments can cause the debugger to report values that mislead the programmer. We will also look at the techniques we have used to recover the expected value of noncurrent variables.

The scheme can be extended to allow optimization of programs in the presence of exception handling constructs.

1.1 The Currency determination problem

Static analysis of currency of variables can report three status values of the variable to the user while debugging. Current, noncurrent and suspect. However one cannot statically determine at what point a suspect variable contains a noncurrent value (or a current value). This problem can be solved only through the use of dynamic currency analysis.

Currency determination is vitally important in dynamic debugging of a program because it would be incorrect for a debugger to report the actual value of a noncurrent or suspect variable to the user. Recovery is the procedure for obtaining the expected value of a suspect or noncurrent variable at a breakpoint.

Figure 1.1 shows the program flow graph of a program before optimization. The program is then optimized by moving the assignment $a := 5$ from node 4 to node 3 to suppress its redundancy along path 1-2-4 in the program. Consider a breakpoint set at the start of node 4. The actual value of a at the breakpoint is the value found in its memory location. This value is suspect. The expected value would depend on the path followed during execution; it would be 5 if execution followed the path 1-2-4, and 0 if execution followed the path 1-3-4. Thus, the value of x is suspect at the start of node 4. Note that x is current at the end of node 4.

Currency determination involves analysis of all possible paths in the original and optimized programs to determine whether the expected and actual values of a variable would be identical at a breakpoint. This analysis can be performed using standard methods of data flow analysis. Hence this approach is called static currency determination. In Figure 1.1 static currency determination would deduce that a is suspect due to the reasons mentioned earlier. Static currency determination is safe in that it considers all possible paths in a program and declares a variable to be current only if its expected and actual values are the same along all paths. However, the information provided by static currency determination is not precise enough for the purpose of dynamic debugging. For example, in Figure 1.1 the value of a can be reported at the breakpoint if execution had followed path 1-2-4. However, static currency determination forbids this by declaring a to be suspect. We use the notion of dynamic currency of a variable to indicate whether the actual and expected values of a variable are the same at a breakpoint in a specific execution of a program. This provides a precise answer to the currency determination problem. As is obvious, run-time information is used to determine the dynamic currency of a variable. We use the term dynamic currency determination for such an approach. Dynamic currency can be determined by timestamping of basic blocks to find out the precise execution trajectory. This is followed by dataflow analysis to determine the currency of the variable.

1.2 Problems due to noncurrency of variables

Noncurrent variables can confuse the programmer while debugging optimized code. Noncurrent variables can also cause problems in programs that are optimized in the presence of exception handling constructs if control flow due to exceptions are not considered.

Traditionally optimizations in the presence of exceptions takes control flow due to exceptions also into consideration. However if one takes control flow due to exceptions into account, smaller basic blocks result. Also optimizations are carried out only if the resulting optimization is computationally optimal even when control flow is along edges due to exceptions thrown. One generally would not like to disable optimizations to preserve computational optimality in the case when exceptions are thrown, since exceptions are very infrequent.

Let us then consider a model where control flow due to exceptions are ignored. This gives rise to the following kinds of problems

- Variables that are live in the exception handler may be noncurrent.
- The wrong exception may be thrown by the optimized code(The issues and work done in this area will be discussed in chapter 2.)

We discuss methods to overcome these problems in this report.

1.3 Recovery of the value of variables

We report the current(expected) value of a variable to the user by performing value recovery i.e. we recover the previous value of the variable by re-executing the previous assignment to the variable. If any of the operands in the RHS of the previous assignment has been overwritten, its value is also recovered similarly. In essence, we perform a traceback of the program that resulted in the assignment of the expected value to the variable.

This traceback facility can be further enhanced to provide the programmer with an additional debugging tool, a retrospect facility that allows the programmer to trace back all assignments in a program that resulted in the value reported by a variable. This facility can be extremely useful to the programmer when debugging and developing programs.

1.4 Overview

This report describes various techniques for debugging of optimized code through value recovery, as well as a traceback facility for programmers so that they can find out how a variable acquired its value at the break point.

Chapter 2 describes various techniques used in the debugging of optimized code in current literature. These include dynamic currency determination, use of shadow variables etc. The issue of throwing the correct exceptions in an optimized program is also considered.

In Chapter 3 we discuss the use of execution trajectories, CET (Complete Execution Trajectory), SET (Selective Execution Trajectory) and MET (Minimum Execution Trajectory), to debug programs in the presence of any kind of optimization.

In Chapter 4 we discuss our work related to the design of a retrospective debugger used for debugging optimized code by using execution trajectories and value recovery. The scheme allows for additional functionality to be incorporated in a debugger, namely retrospect (traceback). The scheme can also be extended to allow for optimizations in the presence of exceptions.

In Chapter 5 the design of the system is discussed.

In Chapter 6 we summarize the report and look at further possibilities.

Chapter 2

Literature Survey

Various methods have been discussed in literature for debugging of optimized code and optimization in the presence of exceptions. We discuss some of them here.

2.1 Using Recovery of variables

[4] uses timestamp-based dynamic currency analysis[3] and recovery scheme to enable optimization in the presence of exception handling and debugging.

This scheme will be discussed in detail in chapter 3. Only an overview is being offered in this section.

They propose a model for exception handling and debugging of optimized code. In this model an exception handler first recovers the values of noncurrent variables and stores them in special variables called shadow variables. It then goes about its business of performing exception handling actions desired by the programmer using the values in the shadow variables. The compiler helps the exception handler by making appropriate provisions for currency analysis and recovery. A debugger similarly recovers the values of noncurrent variables before performing debugging actions.

This scheme has bounded storage requirements. Every basic block has a time stamp associated with it which represents the most recent execution of the basic block. It is proved that dataflow analysis on such a modified PFG, where the nodes are ordered by timestamps is adequate to determine the currency of any variable at run time.

However all optimizations are not enabled. An irrecoverability condition is defined for this purpose. The compiler provides a facility for currency determination and recovery of noncurrent variables. An exception handler (debugger) recovers values of noncurrent variables before performing exception handling (debugging) actions.

A recoverable optimization is one in which any variable made noncurrent or suspect by it can be recovered in all situations that may arise during the execution of the program. This property ensures that exception handling can be performed without any interference from optimization.

It is uniformly applicable to both exception handling and debugging. Also, its execution overheads are marginal when the refinements described in [4] are implemented. This makes it very promising for exception handling and debugging of optimized programs. They define a condition for irrecoverability and present an algorithm to determine recoverability of an optimization. It shows that the value of a noncurrent variable can be recovered if only recoverable optimizations are performed.

2.2 Copying of overwritten values

[5] describes a method for debugging of optimized code by using save points. In this approach, a breakpoint in the source code is mapped to several invisible breakpoints in the optimized object code.

They consider 3 cases.

1. **Code Hoisting**
   If the value of a variable at a breakpoint is noncurrent (since it was overwritten early), reaching definition analysis is performed to determine the program points where the expected value has been overwritten by the hoisted expression. Invisible breakpoints are inserted at all such sites. On reaching the invisible breakpoints the debugger copies the expected value into a

shadow variable and allows execution to continue.

On hitting the actual breakpoint the debugger reports the value in the shadow variable to the user.

2. **Code sinking**
   If the value of a variable at a breakpoint is noncurrent (since a definition of the variable has been sunk), data flow analysis is performed to determine the program points where the assignment expression has been sunk. Invisible breakpoints are inserted at all such sites. On hitting the actual breakpoint the debugger flags the variable as noncurrent and execution of the program is allowed to continue. On reaching the invisible breakpoints the debugger reports the value in the variable to the user.

3. **Dead Code Elimination**
   In this case, breakpoints are inserted at points where it is known that a certain definition wont be executed. When the program hits this breakpoint the user is informed that the value is not computed by the program.

2.2.1 Data Flow equations for analysis

Definition : The predicate $SimplePath(x, y)$ is true if $\exists$ path P from program point x to program point y in the optimized program and P does not include backedges of loops enclosing both x and y.

Let $ARHead(D_v)$ denote the actual (optimized) position of Dv (defn of variable v). Let $ORHead(D_v)$ denote the original (unoptimized) position of D_v (defn of variable v).

If $SimplePath(ARHead(D_v), ORHead(D_v))$ is true and the location of v at the program point before $ARHead(D_v)$ is the same location that is used to hold the value of D_v, then v is overwritten early at D_v in the optimized code.

To determine where values of variables are written late in the optimized program, they compare, the original positions of the definitions and their actual positions in the optimized program. That is, for a definition D_v, the existence of a path P from $ORHead(D_v)$ to $ARHead(D_v)$ such that P does not include backedges enclosing both points is determined. Thus, if $SimplePath(ORHead(D_v), ARHead(D_v))$

is true, then definition D_v is written late in the optimized code.

Computing SaveDiscardPoints[] and EarlyAtBkpts[].

If a value of x is overwritten early at D_v in the optimized code, then a save point is associated at the position of D_v in the optimized code, and discard points are associated at the corresponding positions of original definitions of x that are reachable from D_v in the optimized code (since the variable is now current) . Data flow analysis is used to determine reachable original definitions, which is similar to the reachable definitions problem. After the save and discard points of D_v are computed, the breakpoints where reportability is affected by D_v is calculated. $D_v \in EarlyAtBkpts[b]$ if b lies along paths from save to corresponding discard points of D_v. $EarlyAtBkpts[]$ is easily computed by solving the following data flow equation :

$$EarlyAt(B) = \bigcup_{N \in pred(B)} Gen(N) \cup (EarlyAt(N) - Kill(N))$$

where

$Gen(B) = \{D_v : D_v$ overwrites early and a save point of D_v is at B$\}$ and

$Kill(B) = \{D_v : D_v$ overwrites early and a discard point of D_v is at B$\}$.

Computing StopPoints[] and LateAtBkpts[].

For a definition D_v that is written late, $StopPoints[Dv]$ are the earliest points at which execution can stop because either (1) the late value is computed or (2) a point is reached such that it is known the value will not be computed in the execution. A stop point of D_v is associated at the $ARHead(D_v)$. Stop points are also associated with the earliest points along paths from $ORHead(D_v)$ where the appropriate instance of D_v does not execute. That is,

$$\begin{gathered} p \in StopPoint(D_v) if \\ p = ARHead(D_v) \vee ...(1) \\ (\mathrm{D}_v \notin ReachableLate(p) \wedge ...(2) \\ \nexists p0(SimplePath(p0, p) \wedge p0 \in StopPoint(D_v)))...(3) \end{gathered}$$

Condition 1 ensures a stop point is placed at D_v. Condition 2 ensures the rest of the stop points are not placed at program points where the appropriate instance of the late statement would execute. Condition 3 ensures stop points are placed at the earliest points. $ReachableLate(p)$ is the set of statements written late that are reachable at p. $ReachableLate()$ is easily computed by solving the following data flow equation :

$$ReachableLate(B) = \bigcap_{N \in succ(B)} Gen(N) \cup (ReachableLate(N) - Kill(N))$$

where

$Gen(B) = \{D_v : ARHead(D_v) = B\}$

$Kill(B) = \{D_v : ORHead(D_v) = B\}$.

2.3 Correct ordering of exceptions

[6] discusses a model in which control flow due to exceptions are ignored during the construction of basic blocks. This results in larger basic blocks called extended basic blocks that enable a higher number of optimizations.

However Optimized programs may throw the wrong exceptions due to reordering of code. Consider the following program segment

```
a := c / b ;
f := object.field ;
```

After optimization the code could be reordered as follows

```
f := object.field ;
.
.
.
a := c / b ;
```

Consider the case where the value of variable b = 0 and the object is a null reference. The programmer expects the divide by zero exception to be thrown. However due to code hoisting a null reference exception is thrown.

[6] discusses a method to avoid this by introducing artificial data dependencies between two exception throwing instructions to ensure that the correct exception is thrown. In this case a data dependence is introduced between the two instructions by modelling every PEI (potentially excepting instruction) as involving a write to a common exception variable.

The sample program in Figure 2.1c and its corresponding dependence graph as constructed by [6] is shown in Figure 2.1 . Note the dependence edge between every PEI.

[7] does away with this dependency by allowing an incorrect exception to be thrown. This exception is caught by a handler which then executes compensation code to throw the correct exception. This model enables far more optimizations in an environment like Java where a high percentage of instructions are potentially exception throwing.

The basic idea is that instead of throwing exceptions the instructions set a flag associated with it. Execution of the program continues as though no exception was thrown. No instructions in the program that are dependent on the instructions that set the flag are executed. After execution of all instructions that are not dependent on the exception throwing instruction, all the flags that have been set are scanned. Amongst the instructions that have set flags, the instruction that occurs earliest in the original source code (as against the optimized code) is selected and the exception corresponding to the flag set by that particular instruction is thrown. Thus the correct exception is thrown as expected by the user.

Every PEI has a flag associated with it. All the flags in a function are ordered according to their occurence in the unoptimized program. This ordering is used to determine later which exception is to be thrown if more than one flag is set.

Since this scheme involves setting of flags checking of flags before executing any instruction, it incurs runtime costs. To avoid this the compiler generates 2 sets of code. One where flags are not set but exceptions are thrown directly. The other one as described above is the compensation code in which flags are set and the correct exception is thrown after scanning all flags. Initially the code that throws exceptions is executed. However once an exception is thrown control is transferred to the code where the flags are used. Execution then proceeds as described earlier. In effect, occurence of the exception is compensated and the appropriate exception is thrown. Thus the overhead of scanning flags to check if the instruction is to be 2 is incurred only if the function throws an exception.

All variables are current at all times. Optimizations that make a variable suspect are disabled through live variable analysis.

Consider the following Java program.

```
public class EXCEPT {
        int f, g;
        static void foo(int a[], int b[], int i, EXCEPT p) {
                a[i] = p.f + p.g ;
                b[i] = p.g;
        }
}
```

Its corresponding IR form is

```
Byteoffset    Operator     Operands                              Graph Node
----------    ---------   ----------                            -------------

label B0
3          PEI null.check   c1 = p                                  P1
3              int.load t0 = @-p, -16"", !EXCEPT.f?, [c1]          n2 // t0 = p.f
               int.load t1 = @-p, -20"", !EXCEPT.g?, [c1]          n3 // t1 = p.g
10             int.add t2 = t0, t1                                 n4 // t2 = t0 + t1
11         PEI null.check c2 = a                                   P5
               int.load t5 = @-a, -4"", [c2]                       n6 //
11         PEI trap.if c3 = t5, i, !=U, [c2]                       P7 // bounds.check
               int.shl t6 = i, 2, [c2, c3]                         n8
```

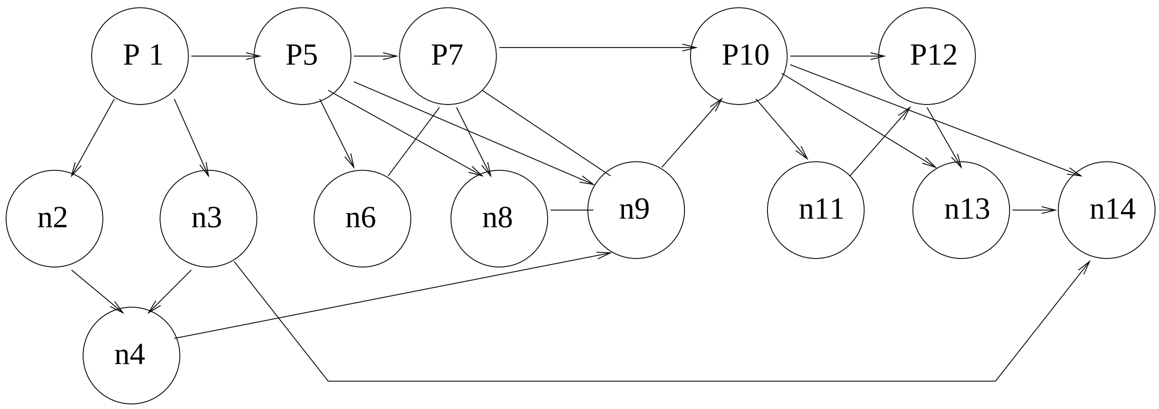

Figure 2.1: Dependence graph using extended basic blocks.

```
11              int.store t2, @-a, t6"", [c2, c3]              n9 // a[i] = t2
18          PEI null.check c4 = b                              P10
                int.load t7 = @-b, -4"", [c4]                  n11 //
18          PEI trap.if c5 = t7, i, !=U, [c4]                  P12 // bounds.check
                int.shl t8 = i, 2, [c4, c5]                    n13
18              int.store t1, @-b, t8"", [c4, c5]              n14 // b[i] = t1
19              return                                         n15

end.block B0
```

Figure 2.1c : Sample Java Program.

The data dependence graph for the program is as shown. In this program the second statement cannot be moved ahead of the first because of the dependency between the PEIs.

If the code compensation technique is used, dependencies involving the Pi nodes are completely eliminated. The modified dependence graph is as shown in Figure

The generated code for a method foo with PEIs would be as follows

```
foo() {
```

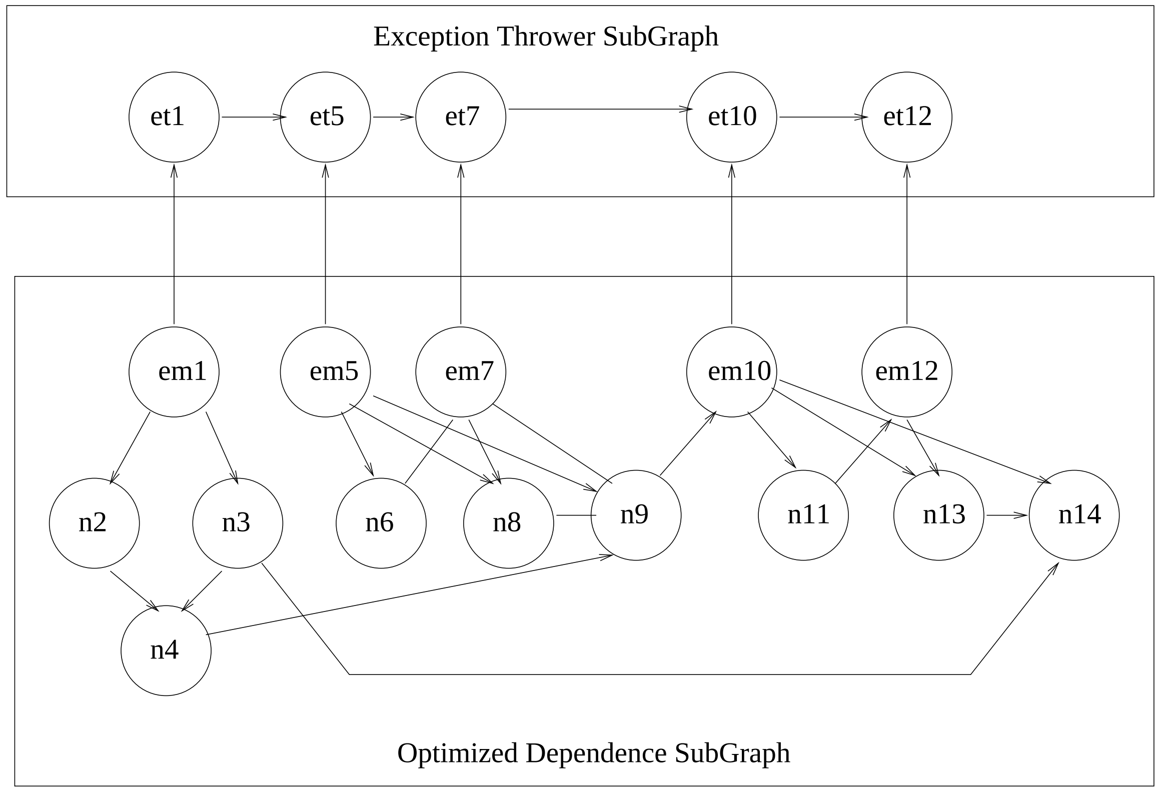

Figure 2.2: Modified dependence graph using code compensation

```
   try {
          // optimized code of foo from the optimized graph
          // p1 occurs before p5 in the original source program
          // p5 has been hoisted
          // p1,p5,p7 can throw exceptions.
          p5 ;
          p1 ;
          p7 ;
   }
   catch (... e)
   {
     // handler prologue all flags = false;
     if (e was thrown by PEI1)
         { flag.p1 = true; branch to LABEL.1;}
     if (e was thrown by PEI5)
         { flag.p5 = true; branch to LABEL.5;}
     if (e was thrown by PEI7)
         { flag.p7 = true; branch to LABEL.7; ""}
   }

// Note that code for PEI5 has been code-motioned before PEI1. . . .
  LABEL.5:
      // code immediately after PEI5 monitor in optimized code . . .
  LABEL.1:
      // code immediately after PEI1 monitor in optimized code . . .
  LABEL.7:
      // code immediately after PEI7 monitor in optimized code . . .
  if (flag.p1) throw exception(PEI1);
  if (flag.p5) throw exception(PEI5);
  if (flag.p5) throw exception(PEI5);

}
```

This scheme results in no runtime overhead during the normal execution of the program. However optimizations of variables that are live in the exception handler are disabled.

Chapter 3

Recovery using Execution Trajectory

[1], [2] and [3] discuss the use of execution trajectories for the debugging of optimized code and optimization in the presence of exceptions.

This model where the noncurrent variables are recovered at a breakpoint is extendable to program optimizations in the presence of exceptions only if one type of exception can be thrown by the function body. It does not handle the problems that can occur if a function can throw different types of exceptions. However, this scheme can be merged with that in [7] to enable optimizations even when the exceptions thrown are of different types.

Our retrospective debugger is based on the ideas in [1] and [2]. Hence the execution trajectories used in [1] and [2] will be studied in detail in this chapter.

We will then continue by specifying the data flow equations required for currency analysis. Then two different kinds of execution trajectories will be discussed from the point of view of currency determination and value recovery of noncurrent variables.

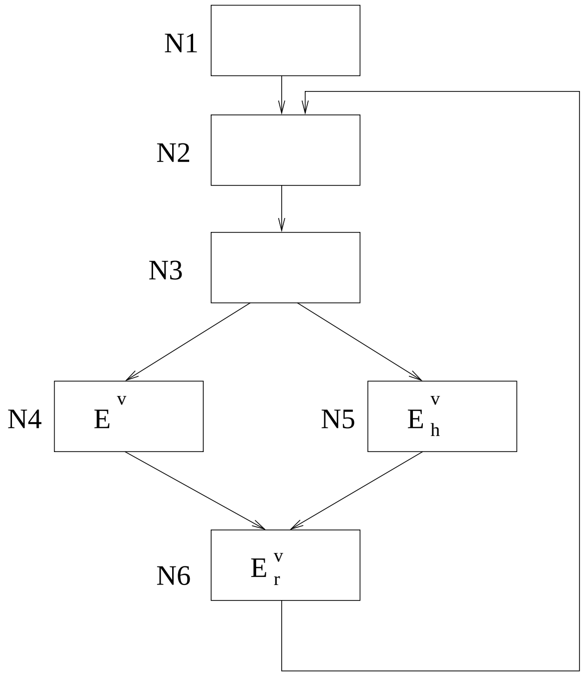

Figure 3.1: A sample program

3.1 Currency Analysis

Currency can be determined by static data flow analysis on the PFG. If the execution history of the program is available, dynamic currency can be determined by data flow analysis over the execution path. Dynamic currency is more precise than static currency determination.

Consider the program in figure 3.1. E_r^v is an assignment of an expression E to the variable v in the source code, that has been removed in the object code. E_h^v is an assignment of an expression E to the variable v in the object code, that is not present in the source code. The compiler has hoisted the expression E_r^v to E_h^v.

A redundant assignment expression E^v hoist-reaches via a path $P = (start, ..., O)$, if E_h^v reaches O along P, and E_r^v does not occur after the last occurrence of E_h^v along P.

- V is noncurrent at O if E_v hoist-reaches along all paths $(start, ..., O)$
- V is suspect at O if E_v hoist-reaches along at least one path, $(start, ..., O)$,

(but not along all paths)

- V is current at O if E_v does not hoist reach along any path.

A hoisted assignment site makes the variable noncurrent. ($Hoist\_gen_i^V = true$). A redundant assignment site makes the variable current again. ($Hoist\_kill_i^V = true$) If a hoisted assignment reaches a program point along all paths, the variable is noncurrent there. If a hoisted assignment reaches along some, but not all paths, the variable is suspect.

The above currency determination problem can be solved through data flow analysis on the PFG. The local properties used are

- $Hoist\_gen_i^V$: It is true if the basic block contains a hoisted assignment statement to variable v not followed by a redundant assignment site (an assignment to variable v).
- $Hoist\_kill_i^V$: It is true if the basic block contains a redundant expression site (an assignment to variable v) not followed by a hoisted assignment to variable v.

Data Flow Equations

$Hoist\_reach\_in_i^V = \Sigma_p Hoist\_reach\_out_p^V$

$Hoist\_reach\_out_i^V = Hoist\_reach\_in_i^V . Hoist\_kill_i^V + Hoist\_gen_i^V$

3.2 Execution Trajectories

We shall discuss techniques for currency determination of statically suspect variables and the recovery of value of noncurrent variables using the execution trajectory of the program.

Three execution trajectories will be considered

- Complete Execution Trajectory (CET)
- Minimal Execution Trajectory (MET)

- Selective Execution Trajectory (SET)

An execution trajectory is like a trace of the program. It is obtained by instrumenting the program to record information regarding the execution path followed during execution. An execution trajectory has the form $...J^{q-1}, I^q, K^{q+1}...$ where I, J, K are statements in the program and $q-1, q, q+1$ are their positions in the sequence.

The execution trajectory of a program is typically obtained by timestamping each Basic Block on execution and storing the timestamps in an array. Different kinds of time stamping schemes can be used. We will begin by considering CET (Complete Execution Trajectory).

The execution trajectory of a program is used to determine currency of variables by solving the *hoist_reach* data flow. It is also used to perform the value recovery of noncurrent variables.

3.3 Complete Execution Trajectory

[1] discusses the Complete Execution Trajectory. CET records the complete execution history of the program. The *Complete Execution Trajectory* (CET) is constructed using the following procedure: Time-stamps are associated with each node of a program flow graph (PFG) G = *(N, E,* n_0*)*. Every time the node is visited during execution of the program the timestamp is recorded. This is implemented through code instrumentation. A time-stamp is the value in an integer counter which is incremented by 1 whenever a new node is reached during execution. A block is defined as a pair (N_i, t_j), where N_i is a node in a program flow graph and t_j is the time stamp when N_i was executed. CET is a sequence of blocks such that

- $\forall$ blocks $(N_i, t_j), N_i \in N$ and$t_j > 0$
- $\forall$ pairs of adjacent blocks $((N_i, t_j), (N_k, t_l))$ in CET,
  $t_l = t_j + 1$.

The memory requirement of CET is unbounded, since the array size is unbounded.

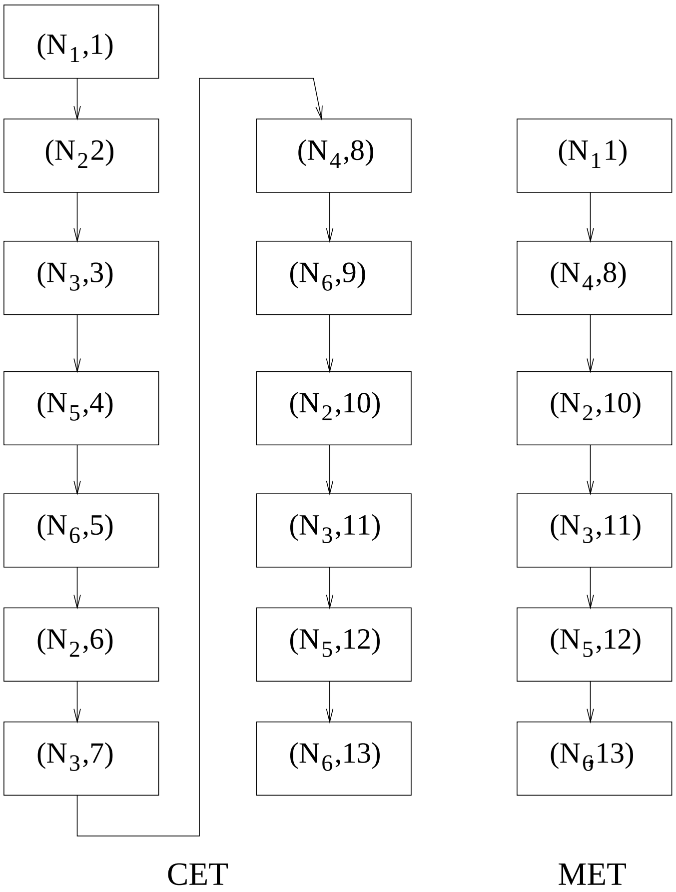

Figure 3.2: CET and MET

3.3.1 CET Example

The storage requirement of CET is unbounded. Figure 3.2 shows CET for an execution of the program of figure 3.1 in which the branches $(N_3, N_5), (N_3, N_4)$ and $(N_3, N_5)$are taken in the first three iterations before execution reaches the breakpoint at the start of node N_6.

3.3.2 Currency analysis

To dynamically determine the currency of suspect variables in the program, one solves the hoist reach dataflow problem over the entire CET. Since the size of CET is unbounded, this dataflow analysis problem can take unbounded time to terminate.

$Hoist\_gen^v$ is true in block N_5. $Hoist\_kill^v$ is true in N_6. Forward Dataflow analysis is started from the first block $(N_1, 1)$ 2 the breakpoint. If the breakpoint is after $(N_6, 13)$ the variable v is current. If it is between $(N_5, 12)$ and $(N_6, 13)$ it is noncurrent.

3.3.3 Value Recovery

If the a variable is found to be noncurrent at a breakpoint its value can be recovered using CET as follows. The previous definition that reaches the current definition is evaluated again and displayed to the user. If any of the operands on the RHS of the definition are found to be noncurrent then the value of the operands is recursively evaluated. The previous definition of the variable is determined using CET.

Procedure $Recover\_value(v, r, Bkpt)$

Let $q := Previous\_defn(v, r)$;

Let q contain an assignment $v := fv(...)$

such that RHS_v is the set of variables in the RHS of the assignment.

$\forall x \in RHS_v$

if $Same\_value(x, Bkpt, q) = false$ **then**

$Recover\_value(x, q, Bkpt)$;

Execute the assignment $v := fv(...)$;

end

Figure 3.1c : Procedure Recover value

3.4 Minimum Execution Trajectory

The storage requirements of CET is unbounded. We begin by considering the scheme [4], that involves dynamic currency determination by associating only a single timestamp per basic block. The storage requirements for this scheme is O(N).

In this scheme, in every execution of the basic block, the old timestamp is overwritten by the new value. The execution trajectory formed by arranging the basic blocks in the sorted order of timestamps is called a minimum unrolled graph. We shall refer to it as MET (Minimal Execution Trajectory) in this report.

At the end of execution, a block is formed for each node. MET is the sequence of blocks in ascending order by timestamps.

3.4.1 An Example

Figure 3.2 shows the MET for the program flow graph in figure 3.1 The size of the MET is bounded by the number of nodes in the PFG.

3.4.2 Currency Analysis

It has been shown in [3] that MET is adequate for dynamic data flow analysis i.e. forward bit-vector data flow analysis on CET and MET produce the same results at the last node in the execution trajectory. Memory requirement is bounded by the number of nodes in the PFG and so is the time complexity of the dataflow analysis.

Consider the MET in figure 3.2. If the breakpoint is after $(N_6, 13)$ the variable v is current. If it is between $(N_5, 12)$ and $(N_6, 13)$ it is noncurrent. The result is the same as that obtained with the CET.

3.4.3 Recovery of Value

An execution history is adequate if it supports recovery of a dynamically non-current variable under all conditions. CET is adequate however MET, which is a sub-sequence of CET, has been shown to be inadequate for recovery [2,4] due to the following observation: Consider two statements s1 and s2 located in nodes N_i and N_k, respectively, of program P. Let blocks (N_i, t_j) and (N_k, t_l) be consecutive blocks in MET and let $t_l > t_j + 1$. It is clear that any nodes executed between

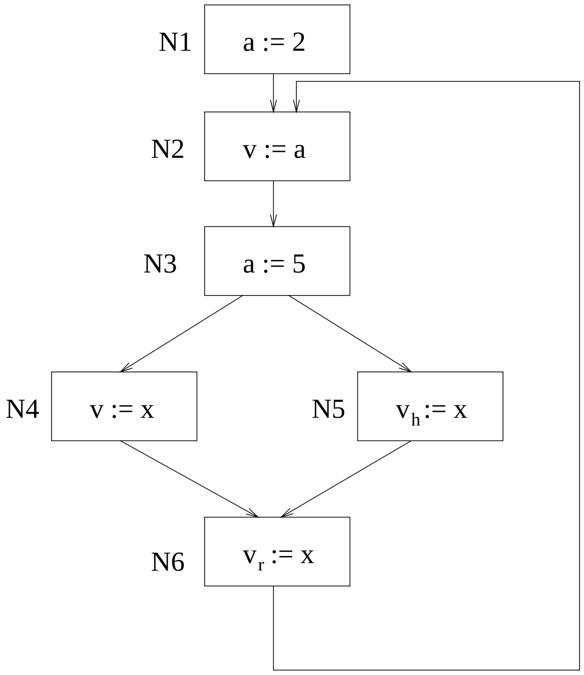

Figure 3.3: Computations in the program

these instances of execution of N_i and N_k must have been executed again. However, since MET contains information only about the last execution of a node, it cannot indicate which nodes executed after N_k had been executed between the executions of N_i and N_k represented by blocks (N_i, t_j) and (N_k, t_l). This makes recovery using MET impossible in certain cases.

An Example of Inadequacy

Consider Figure 3.3 which shows computations in the program flow graph of figure 3.1. MET is same as given in Fig. 3.2. Let p_{ji} designate the program point at which statement i of node N_j is situated. At the breakpoint, using dynamic currency analysis we determine that variable v is noncurrent and make the call $Recover\_value(v, p_{51}, Bkpt)$. Using the MET of Fig. 3.2, the call $Previous\_defn(v, p_{51})$ returns p_{21} , which is the site of the assignment $v := a$. The assignment $a := 5$ at point p_{32} causes $Same\_value(a, p_{21}; Bkpt)$ to be false. The call $Previous\_defn(a, p_{21})$ returns p_{11}, which is the site of the assignment $a := 2$. Now, recovery will be completed by first executing $a := 2$ and then executing $v := a$ using shadow variables. However, this recovery is incorrect since the expected value is $v = 5$ due to execution of node N_3 in the previous iteration of the loop. This is evident from the CET of Fig. 3.2

MET is insufficient because the statement $a := 5$ is executed once after its previous execution reaches $v := a$ but before the execution of the breakpoint in node N_6. This causes the timestamp required for value recovery (in $Previous\_defn$) to be overwritten on its second definition.

[2] identifies such nodes as critical nodes. The basic property of such nodes is that there exists a path from node N_2 which is a recovery definition for the variable v to the breakpoint which includes the node itself (N_3). N_3 itself should be a recovery definition for N_2.

Thus MET is sometimes inadequate for value recovery. Thus there are two options. To abort recovery of the value of a variable in such a case. The other possibility is to inhibit optimizations in such cases.

3.4.4 The Irrecoverability condition

[4] define an Irrecoverability condition that is applied after any code hoisting transformation to test for recoverability of non current variables. If the test fails the optimization is not performed.

Terminology and Notation

The terminology used for the definition of the irrecoverability condition will be explained with respect to Figure 3.3.

A point in the program where an expression E_v is located is called an expression site ES_v (e.g. N_4) A point at which a hoisted expression E_h^v (e.g. $v_h := x$)is located is called an insertion site IS_v (e.g. N_5), and a point at which a redundant expression E_r^v is located is called a redundant expression site RE_v(e.g. N_6). One or more definitions of variable v would reach an insertion site IS_v. Such definitions are called recovery definitions $RD_v^i, i = 1, 2, ...$ (e.g. $v := a$ in N_2). The point in the program where recovery definition RD_v^i is located is called a recovery site RS_v^i (e.g. N_2). $node(s_i)$ gives the node containing statement s_i. Each variable occurring in the right hand side of a recovery definition RD_v^i (e.g.

a) is called a recovery variable. The set of such recovery variables is denoted by RV_v^i (e.g. a). DRV_v^i (e.g. $a := 5$ in N_3) denotes the set of definitions of variables in RV_v^i which occur in a path starting from a recovery site RS_v^i and terminating at the breakpoint or exception-raise point. Each definition in DRV_v^i causes the predicate $Same\_value(x, RS_v^i; Bkpt)$ to be false for some x $\in RV_v^i$, e.g. the assignment $a := 5$ at point p_{31} causes $Same\_value(a, p_{21}, Bkpt)$ to be false.

Condition IC1 : Irrecoverable variable.

A variable V is irrecoverable at an expression site ES_v if $\exists S_k \in DRV_v^i$ such that S_k reaches RS_v^i.

Condition IC2 : Recursively Irrecoverable variable.

A variable V is recursively irrecoverable at an expression site ES_v if V is irrecoverable at the site or $\exists x \in RV_v^i$ such that x is recursively irrecoverable at RS_v^i.

In Figure 3.3, p_6 is an expression site E_v. In the optimized program this definition has been hoisted to node N_5. This makes the variable noncurrent at p_6. So at the breakpoint the corresponding Recovery Definition RD_v^1 for variable v is p_2 (v := a). Now the Recovery variable for this defn is RV^1 is a. Now the definition of a reaching p_2, DRV_v^1 is p_3. Thus p_3 forms an s_k. But this definition reaches the breakpoint. Hence the optimization is irrecoverable.

3.5 Selective Execution Trajectory

In this section we shall discuss the use of SET (Selective Execution Trajectory) to perform recovery. We will also specify the code instrumentation technique to be used to perform the value recovery.

CET contains redundant timestamps that are not required for value recovery and currency determination and has unbounded storage requirements. MET has bounded storage requirement but is insufficient for value recovery. [2] propose the use of Selective Execution Trajectory (SET) as a via media between CET and MET.

SET is constructed by identifying critical nodes in the program that require more than one timestamp. During execution a trajectory containing all executions of critical nodes is constructed. This is known as the Critical Nodes Execution Trajectory (CNET). SET is constructed at the breakpoint by combining MET and CNET. Note that the size of the CNET is unbounded, but CNET will be typically smaller than CET.

3.5.1 Identification of Critical Nodes

A critical node is a node of the PFG which must appear more than once in an execution trajectory to guarantee correct recovery of a noncurrent variable under all conditions.

The identification of critical nodes follows directly from the irrecoverability condition presented in the previous section.

Definition

The definition of critical node uses terminology from subsection 3.4.4.

Given an expression site ES_v

(a) A node N_i *in PFG is a critical node if* $\exists s_k \in DRV_v^i$ *such that* s_k *is situated in node* N_i *and* s_k *reaches* RS_v^i.

(b) A node containing a definition $d_k : x := ...$ *is a critical node if* d_k *reaches a critical statement which is enclosed in the same loop as* d_k *and uses* x.

Part (a) of the definition follows from the fact that since s_k occurs on a path from RS_v^i to the breakpoint or exception point, its last execution may follow RS_v^i in the execution history. Hence to find the definition of a recovery variable which reaches RS_v^i, it may be necessary to consider the second-last (or third-last, etc.) execution of s_k. Since MET records only the last execution of a basic block, it will not suffice for value recovery in such cases. In figure. 3.4, let p_{ij} designate the program point where statement j of node N_i is located. p_{51} is an insertion site for the assignment $v := x$. Let us designate this as IS_v^1. The definition $v := a$ located at point p_{21} reaches p_{51} along the path N_2 - N_3 - N_5. Hence $v := a$ is a recovery definition, p_{21} is a recovery site and variable a is a recovery variable. The definition of variable a located at point p_{31} is in DRV_v^1. This definition thus

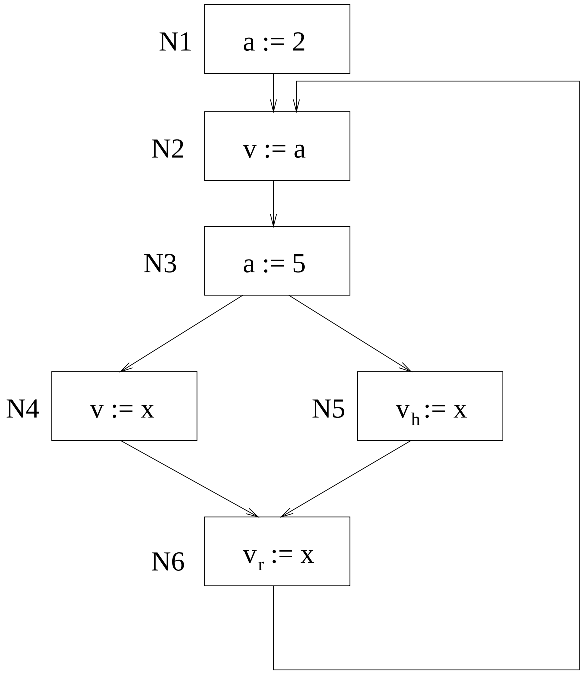

Figure 3.4: A sample program

is an s_k (hence $Same\_value(a, p_{21}, Bkpt)$ is false). Now because of the back edge (N_6, N_2) this definition reaches point p_{21} which is the corresponding recovery site. Hence node N_3 is a critical node

While recovering with the use of a previous execution of s_k, it would be necessary to use corresponding definitions of its recovery variables. Part (b) of the definition therefore identifies nodes containing such definitions also as critical nodes.If these definitions are not in the same loop as s_k then the last execution of these definitions reach s_k. Hence MET suffices for these nodes. Consider the program in figure 3.6. Part(b) would identify the node N_5 as critical.

3.5.2 Currency determination

As the SET is a superset of MET, it is adequate for currency analysis. The SET for the program in figure 3.4, after three iterations, is shown in figure 3.5. Node N_4 is executed in the 2nd iteration, while N_5 is executed in the 1st and 3rd iteration.

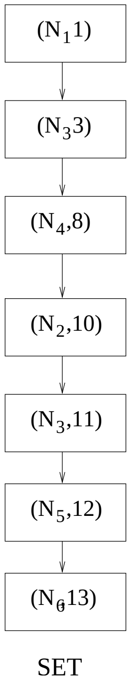

Figure 3.5: SET

3.5.3 Recovery of Value

SET is constructed by merging the CNET and MET. The CNET is unbounded in size. However it is possible to reduce overheads by collecting bounded size information under certain circumstances [1,2]. The critical nodes have more than one, but a bounded number of timestamps associated with them. For example In the preceding example, the timestamps of the last two executions of the node N_3 are stored.

Criticality Number

Criticality number of a critical node N_i, represented as C_{N_i} , is the smallest number of times N_i must appear in an execution history to ensure correct recovery under all conditions. In figure 3.4 the criticality number of node N_3 is 2 because recovery is implemented using the block $(N_3, 3)$, which is its second-last execution.

Consider the program in figure 3.6. As explained earlier, the critical nodes are N_3 and N_5. Here the criticality of node N_3 is again 2. However the criticality

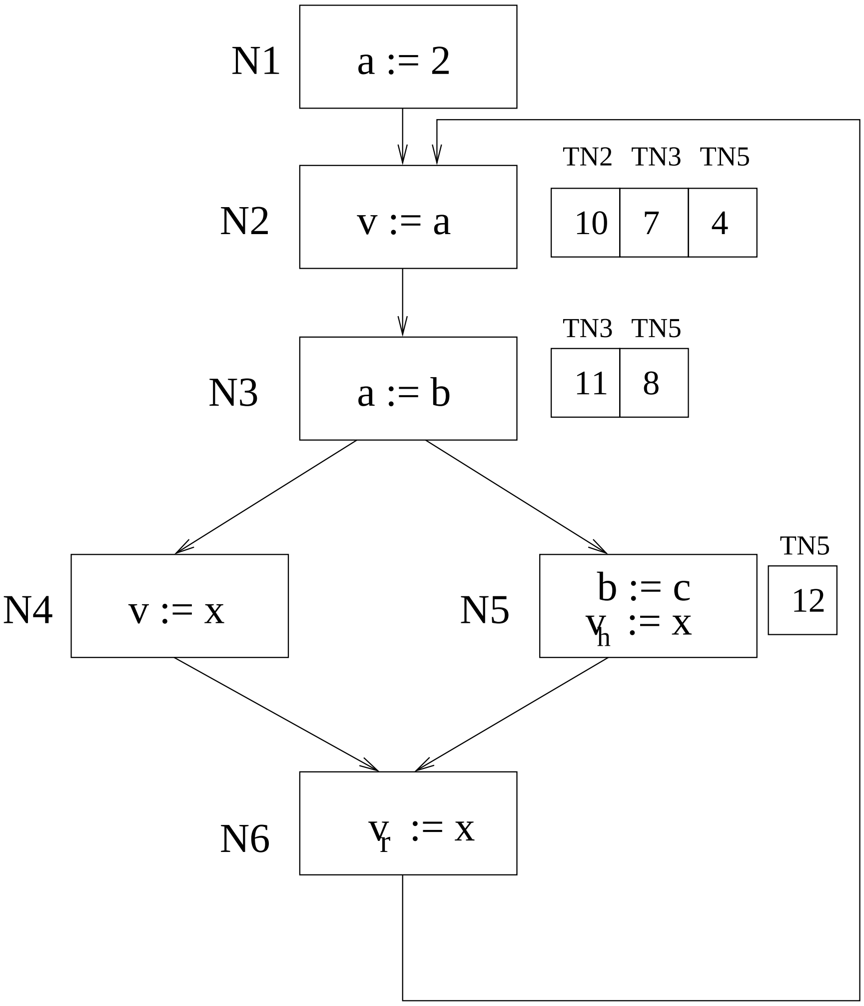

Figure 3.6: Time stamps for a sample program

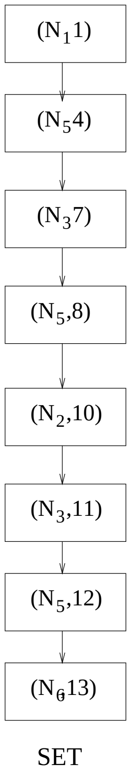

Figure 3.7: SET for sample program of figure 3.6

of N_5 is 3. This can be explained as follows: Recovery of v according to the definition $v := a$ requires recovery of a according to the definition $a := b$ of the previous, i.e. second-last execution. This requires recovery of b according to the definition $b := ...$ in the previous, i.e. the third-last, execution of b. Hence information concerning last 3 executions of b should be preserved to enable recovery. The SET for the program is shown in figure 3.7.

Criticality numbers are small integers in most cases, except in cases like

while (condition)
v := v+1;

where the criticality number is unbounded since the definition $v := v + 1$ is both RS_v^i and s_k in the definition. Such nodes are identified as infinitely critical nodes. Optimizations are disabled in such cases.

[1] also propose an algorithm for instrumentation of code using the idea of criticality numbers to restrict the size of the CET to small values.

Chapter 4

Design of a debugger with retrospection abilities

In this chapter we discuss the design of a debugger with retrospection abilities. The debugger reports the expected value of a variable at breakpoints in the presence of optimizations. We shall also discuss the modifications required to additionally allows for a trace back of the value of any variable at any breakpoint.

4.1 Modification of SET

The SET construction technique is based on the idea of associating an array of time stamps with each basic block, to record their execution history. [1] propose an algorithm for recording the SET. However, the space overheads involved are higher.

Instead we propose a scheme that does not make explicit use of time stamps. The execution trajectory thus constructed does not contain as much information as the CET (exact sequence of execution of basic blocks is not known, only a partial order is obtained). However the trajectory suffices for value recovery and the overheads involved are lesser. Here a history structure is associated with each variable, instead of each basic block and node numbers (or statement numbers) are used instead of timestamps to record the execution trajectory.

4.2 Data Structures for recording the execution trajectory

We associate a history structure with each variable. Consider a variable v. The history structure of v consists of two fields.

- Latest statement number
  This field stores the statement number of the last assignment to the variable v.

- The recovery info
  It stores the history structures of all the variables used in the last assignment to the variable v.

Consider the sample program in figure 4.1. The breakpoint is at the beginning of N_6. The variable v is noncurrent. Statically two definitions of a reach N_2. The recovery definition of v is in node N_2 ($v := a$). The recovery definition for a is in N_3 ($a := b$), and in N_1 ($a := 5$).

The assignment to a in node N_3 uses b. Thus a history structure is required for variable b. The recovery definition for b is in N_4 ($b := 6$) and N_5.

The figure shows the data structure, called the history structure, associated with each variable. The history structure of b consists of a single field to record the last assignment to b. This is because the RHS of all assignments to b contain only constants. The history structure of a consists of two fields. The first field records the last assignment to a. The second field is used to copy the history structure of b. Thus the history structure of a is of size 2. The history structure of v consists of two fields. The first field records the last assignment to v. The second field is used to copy the history structure of a. Thus the history structure of v is of size 3.

Consider three iterations of the loop through the path $N_2 - N_3 - N_5 - N_6$. Thus the sequence of execution is $N_1 - N_2 - N_3 - N_5 - N_6 - N_2 - N_3 - N_5 - N_6 - N_2 - N_3 - N_5 - N_6$.

All the fields of the history structure are initialized to zero.

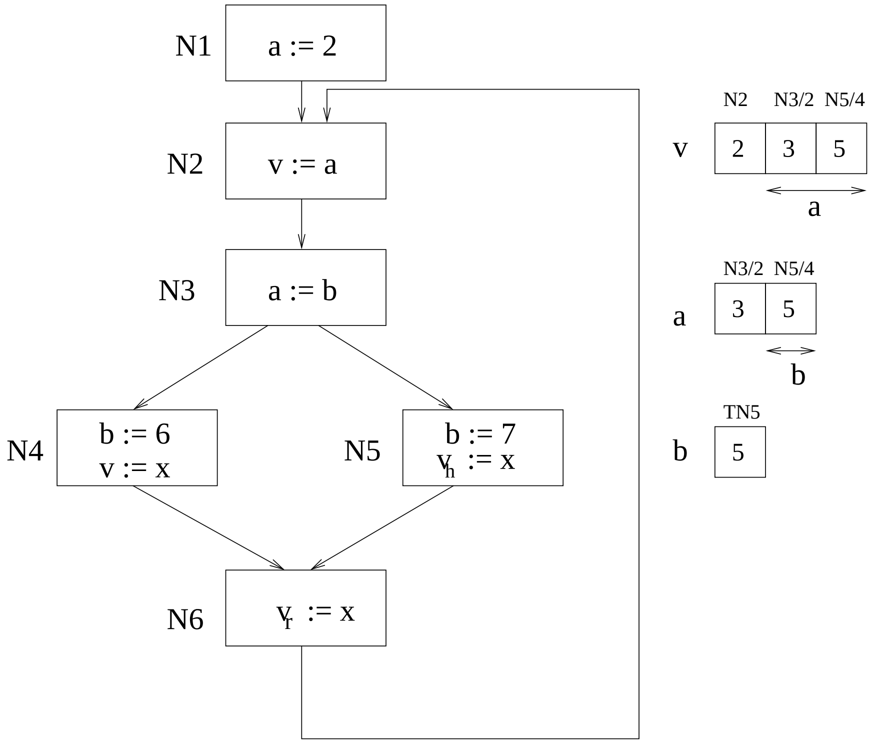

Figure 4.1: History structures for a sample program

Iteration 1 The latest statement executed for variable a is N_1. So the first field in the history structure of variable a is stamped 1 (N_1). Instrumentation code is generated following every assignment to a to stamp the history structure of variable a to indicate the last statement that was executed. Thus after execution of node N_1 the first field of the history structure of a is marked 1. The second field is not stamped and left blank (-1). The contents of the history structure of a now reads (1,-1).

On execution of node N_2 the first field of variable v is stamped with the value 2. Since a is on the RHS of the assignment the entire history structure of variable a is copied into the recovery field of v. The contents of the history structure of v now reads (2,1,-1)

On execution of N_3 the first field of the history structure of a gets updated to 3. The second field is now stamped i.e. the history structure of b (of size 1) is copied into it. Since no assignment to b has executed the contents of the recovery field is -1. The contents of the history structure of a now reads (3,-1).

On execution of N_5 the history structure of b gets updated to 5.

The status of the history structures after the first iteration is
v = (2,1,-1),
a = (3,-1),
b = (5)

Iteration 2 On execution of N_2 the history structure of v gets updated to (2,3,-1). The recovery field is updated fro the history structure of a, (3,-1). On execution of N_3 the history structure of a gets updated to (3,5). The recovery field is updated fro the history structure of b, (5). On execution of N_5 the history structure of b gets updated to (5).

Iteration 3 On execution of N_2 the history structure of v gets updated to (2,3,5). The recovery field is updated fro the history structure of a, (3,5). On execution of N_3 the history structure of a gets updated to (3,5). The recovery field is updated fro the history structure of b, (5). On execution of N_5 the history structure of b gets updated to (5).

Since we use statement numbers rather than timestamps for recording the trajectory, ordinarily a byte should suffice for recording an assignment to a variable (assuming a maximum of 255 unique assignments to a particular variable in a function), whereas a timestamp may have to be at least 4 bytes long. Thus a history structure containing 2 4 or 8 entries can be copied using a single machine instruction.

4.2.1 Infinitely Critical Nodes

To allow for optimizations in the case of infinitely critical nodes, we introduce savepoints that save the value of variables at infinitely critical statements. Thus the recovery algorithm terminates at an infinitely critical node and accesses the value of the variable from the savepoint. The savepoint also forms a part of the history structure.

Consider the program in figure 4.2. Here p_6 is the original source assignment that has been hoisted to p_5. Thus the variable v is suspect at p_6. The recovery definition for the variable v when executing along path $(..., N_2, N_3, N_5, N_6)$ is in p_2 i.e. $(v := a)$ is the RD_v^1. Here a is a recovery variable. The set of definitions

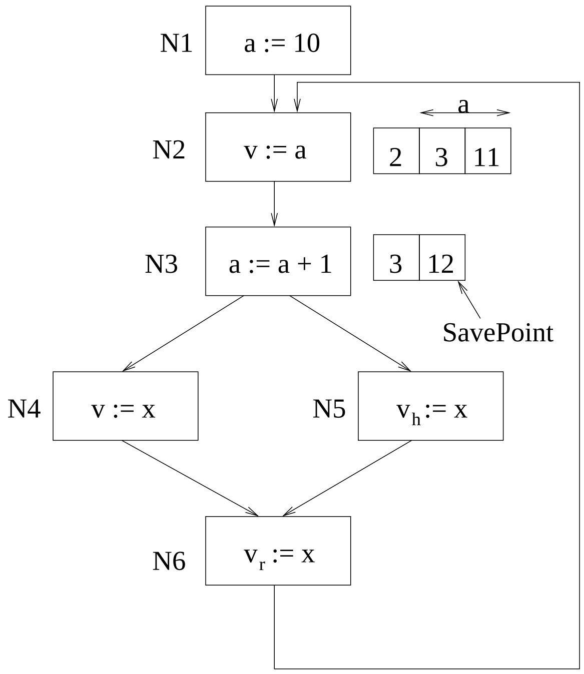

Figure 4.2: Infinitely critical nodes

of a reaching p_2 statically are p_1 and p_3. The statement $a := a + 1$ in p_3 uses the variable a again. So now we look for the definitions of variable a that statically reach p_3. The definitions reaching are again p_1 and p_3.

Thus while performing recursive recovery for statement p_3 during static analysis p_3 is encountered again. If one continues to recursively evaluate the definitions reaching p_3 again, we shall get stuck in a nonterminating loop. This implies that the history structure requirement for infinitely critical nodes is unbounded in size.

We restrict the structure size to a bounded number by introducing a savepoint after the assignment to a in p_3. This savepoint forms part of the history structure (the last field) of a.

The history structures of variable v and variable a are shown in figure 4.2. The second field in the history structure of a is a savepoint for a. Consider the following execution sequence of the program $N_1 - N_2 - N_3 - N_5 - N_6 - N_2 - N_3 - N_5 - N_6$.

Iteration 1 On execution of N_1 the history structure of a gets updated to

(1,0). On execution of N_2 the history structure of v gets updated to (2,1,0). The recovery field is updated fro the history structure of a, (1,0). a is assigned the value 11. On execution of N_3 the history structure of a gets updated to (3,11). The value 11 is the savepoint.

Iteration 2 On execution of N_2 the history structure of v gets updated to (2,3,11). The recovery field is updated fro the history structure of a, (3,11). On execution of N_3 the history structure of a gets updated to (3,12).

4.2.2 Code Instrumentation

We present an algorithm that instruments the code to provide a trace back facility in a retrospective debugger. This can be adapted to provide for debugging of optimized code with minor changes, indicated later.

A history structure is associated with every variable v in the program. The first entry in the history structure is the statement no. of the last definition of the variable v that executed. The rest of the history structure, called the recovery part, contains the history of all variables used in the last assignment to v.

The instrumentation algorithm considers all possible breakpoints in the program.

MET suffices for the timestamping information required in non-critical nodes. For critical nodes, one begins the code instrumentation with the first nodes where the definitions in critical nodes reach i.e. we begin by considering RD_v.

Given a definition D_v of variable v in node N_0, whose value has to be recovered, we consider the variables on the right hand side of the assignment. The value of the variable v can be re-evaluated using this expression. The value of the operands also have to be recursively recovered.

The algorithm inserts instrumentation code that copies The history structure of RHS operands into the recovery field.

The algorithm first identifies all assignment statement sites in the program and generates instrumentation code for each site by invoking $Instrument\_if\_critical$. The function takes three arguments. The first argument is the site of the breakpoint. The next argument is RD_v. The last argument is DRV. The program then invokes $Iscritical$ function, which checks part a) of the definition of critical node and generates instrumentation code. If DRV is not critical the procedure recursively checks if the definitions reaching DRV are critical.

The $Iscritical$ function performs a depth first search starting from DRV and returns true if it finds a path from DRV to the $Bkpt$ that does not include RD.

$Instrument\_Node$ function recursively generates code for the statements that satisfy Part b) of the definition of the critical node.

The following data structures are used in the algorithm.

The *hascriticalpred* data structure is used by the debugger at run time while recovering the value of a variable. The *hascritical* data structure is essentially a table where each element has three statement number fields. The first field is the breakpoint, the second field is the RD_v and the third field is the critical s_k. Thus an entry $Bkpt, RD_v, s_k$ is inserted in the *hascriticalpred* table, if s_k is critical This structure is built at compile time by the compiler for use by the debugger. The structure is used by the debugger to determine whether the MET suffices for recovery. If not, the debugger consults the history structure of the variable for recovery.

The *nodehistorysize* array has an entry per statement in the function. This array stores the size of the history structure used for recovering that particular assignment. The size of the history structure of a variable is equal to the largest nodehistory size for an assignment to the said variable.

The *visited* array is used to check if a node is infinitely critical. Every time a the procedure instruments a particular statement, the *visited* flag is set. When the procedure recursively reaches the statement again the statement is identified as infinitely critical.

The algorithm does not terminate all tracebacks on encountering an infinitely critical node. If the RHS of an infinitely critical assignment statement contains more than one variable, traceback is terminated only for the variable in the RHS that causes the statement to be infinitely critical. This variable is kept track of in the *curruse* array.

The *isonesameloop* variable is used to keep track of whether any of the definitions of the variables in the RHS reaching the statement is in the sameloop. If this is the case then the code generation is continued recursively.

Program Instrumentation

$\forall s_i : v := ... \in function$

  $nodehistorysize[s_j] := 0$

  $\forall x \in use(s_i)$

    $infcritical(s_j,x) := false$

$\forall s_i : v := ... \in function$

  $\forall s_j : v :=$ ... reaching s_i

    $\forall x \in use(s_j)$

      hascriticalpred(s_i,s_j,x) := false

      $\forall s_k \in s_k : x :=$ ... reaching s_j

        if*Instrument_if_critical*(s_i, s_j, s_k) **then**

          hascriticalpred(s_i,s_j,x) := true

Function Instrument_if_critical(s_i, s_j, s_k)

if $nodehistorysize[s_j] > 0$ **then return**

$\forall N_i \in BasicBlocks$

  $flag(N_i) := false$

if *Is_critical*(s_i, s_j, s_k) **then** *Instrument_Node*(s_j)

else

  $\forall v \in use(s_k)$

    hascriticalpred(s_i,s_j,v) := false

    $\forall$ *nextdef* : $v :=$ ... reaching s_k

      if*Instrument_if_critical* (s_i, s_k, *nextdef)* **then**

        hascriticalpred(s_i,s_j,v) := true

Function Is_critical(s_i, s_j, s_k)

Let $N_k = node(s_k)$

if *flag* $(N_k) = true$ **then return** *false*

$flag(N_k) := true$

if $N_k = node(s_i)$ **then return** *true*

if $s_j = node(s_k)$ **return** *false*

$\forall n \in succ\ (N_k)$

if *Is_critical* (s_i, s_j, n) **then return** *true*

return *false*

Function Instrument_Node($s_j : v := ...$**)**

if *nodehistorysize*$[s_j] > 0$ **then return** *nodehistorysize*$[s_j]$

visited(s_j) := *true*

histsize := 1

$\forall x \in use(s_j)$

curruse$[s_j]$:= x

defsize := 0

isonesameloop := false

$\forall$ *def* : $x := ... \in$ reaching s_j

if *visited(def) = false* **then**

if *sameloop*(s_j,*def*) **then**

isonesameloop := true

size := Instrument_Node(def)

defsize := max(defsize,size)

else

infcritical(*def*,*curruse*$[s_j]$) := *true*

if *isonesameloop* **and not** *infcritical*(s_j,*curruse*$[s_j]$) **then**

InsertCode (Node(s_j)*,COPY history*$[v][x]$*.recovery, history*$[x]$

, *defsize*)

$histsize = histsize + defsize$

if *isonesameloop* **and** *infcritical*(s_j,*curruse*$[s_j]$) **then**

InsertCode(Node(s_j)*,COPY history[v][x].savept , value[x])*

$histsize = histsize + 1$

$InsertCode(Node(s_j), COPY history[v][x].current, statementno(s_j))$

nodehistorysize$[s_j]$ = *histsize*

visited(s_j) := *false*

return *histsize*

Recover(v,ts,histcomp,bkpt)

$(si, sits) = getfromMET(v, ts)$

$\forall$ *def* $\in$ *use*(si)

if *hascriticalpred*(*bkpt*,*si*,*def*) **AND** *histcomp*[*def*][0] = *criticaldef* **then**

$Recover\_Value(def)$

else

recover(*def*, *sits*, *histcomp*[*def*])

Recover_Value($s_i : v := ...$,histcomp)

$flag(s_i) := true$

$\forall x \in use(s_i)$

$def := histcomp[x][0]$

if not $flag(s_i)$

$val(x) := Recover\_Value(def, histcomp[x])$

else

$val(x) := histcomp[x].savept$

$flag(s_i) := false$

return $evaluate(val)$

To modify the above algorithm to allow debugging of optimized code, only those statements that cause variables to be noncurrent or suspect need to be considered in the main program. Also complete backward trace is not required. One only needs to recover values for which the $Same\_Value$ predicate is false.

4.2.3 Optimizations during time-stamping

The following optimizations can be applied to the timestamping scheme just described.

1. Consider a sequence of blocks N_i, N_j, N_k such that a definition of a variable v_i in node N_i reaches and is used in the definition of variable v_j in Node

N_j. Similarly definition of variable v_j reaches and is used in the definition of variable v_k in node N_k.

As per the algorithm, given previously, the timestamps in the history structure of N_i is first copied into N_j and then again into N_k. This multiple copying can be avoided if it is known that there exists no path between N_j and N_k that include an execution of N_i. In this case we can directly copy the history structure into the recovery field of N_k on execution of N_i. This is because, it is known that the timestamps that are copied from N_i to N_k will again be copied into N_j as there is no intervening execution of N_i. This idea can be extended to a sequence of n nodes.

2. If there is only a single definition reaching a node, then it is unnecessary to timestamp that particular node.

3. Nodes in which neither hoist_kill nor hoist_reach is true need not be time stamped for inclusion in the MET.

4. In general, we need not time-stamp nodes which are the only predecessors of their successors. A further refinement is possible if edge (p, x) is a critical edge, i.e. if node x has many predecessors and a predecessor node p is a branch node. If p is the only predecessor of another successor y then time-stamping of p is unnecessary as far as control flow along edge (p, y) is concerned. In such cases it is beneficial to split the edge (p, x) and insert the time-stamp in a synthetic node along the edge rather than in node p . The algorithm in Fig. 3.5c implements these ideas.

$\forall$ *nodes x in G*

if x has > 1 predecessors then

time-stamp each predecessor p of x; (split edge (p; x) if it is a critical edge)

Figure 5.2c: Algorithm_Implied_ts

4.3 InterProcedural Recovery

While performing the recovery of value of a variable if there is an assignment of the form

$x := f(x, y);$

one needs to perform value recovery for the statements in the function. For this one needs to instrument the code in the function body.

The analysis of functions for recovery is performed as follows. The call to the function is replaced by deemed definitions. If a function passes any parameters by reference, then deemed definitions for all such parameters are also inserted. A deemed definition is a fictitious function that summarizes the variables that are used in the final assignment made to a referenced parameter.

A conservative estimate is made for the global variables that are suspect at the end of execution of the function. All local variables are assumed to be non-current at the exit of the program. Analysis is started from the last statement(s) of the function.

Consider a procedure call where x and y are passed by reference

$p(x, y)$

For analysis this code is transformed into the following.

$xin := x;$

$yin := y;$

$x := \alpha(xin, yin);$

$y := \beta(xin, yin);$

where α and β are deemed definitions of x and y respectively in terms of xin and yin. Note that x and y need not necessarily be a function of both xin and yin. It could also be $y := \beta(xin)$. The dependency of the parameters x and y on the global variables and parameters passed can be found by recursively performing reaching definitions analysis starting from the exit nodes of the procedure p. This analysis needs to be performed only once for each function and procedure.

Consider the following program

```
while(condition)
{
        y := 3 ;
        x := 4 ;
        f(x,y) ;
        v := x + 1 ; // Critical Node
}

f(x, y)
{
   x := y ;
   y := 5 ;
}
```

Assume the assignment to v is critical. This code is replaced for analysis by

```
while(condition)
{
        y := 3 ;          // Marked as critical
        x := 4 ;
        xin := x ;
        yin := y ;        // Marked as critical
        x := alpha(yin) ; // Critical Node
        y := beta() ;
        v := x + 1 ; // Critical Node
}

f(xin, yin)
{
   xin := yin ;
   yin := 5 ;
```

```
}
```

Here *alpha* and *beta* are deemed definitions.

First criticality analysis is performed for the function f. This is done by considering the last assignment statements to xin and yin in f. When performing recursive reaching definitions analysis from the last assignments, it is found that xin is data dependent only on yin and yin is data dependent on none of the parameters. Thus *alpha* takes only yin as parameter and *beta* has no parameters at all.

As the assignment $v := x+1$; is critical $x := alpha(xin, yin)$ is also critical.

Now when performing criticality analysis for the calling function, one applies the transformation explained above and proceeds as usual. Thus, for example, if x is found to be noncurrent, then the operands on the RHS of $x := \alpha(xin, yin)$; are examined. Thus for the example shown above the statement $x := alpha(yin)$ is critical. Now the last definition of yin is recovered recursively and thus the assignment to y in the caller site is also marked as critical.

For performing interprocedural recovery, one associates a set of timestamps with each call site of the function. These set of timestamps are passed as an extra parameter to the function.Thus a separate set of timestamps are maintained for each call site in the program.

Thus the function f accepts an extra parameter. A pointer to the set of timestamps used by f. The history structure of the statement containing the assignment to v now copies the entire set of timestamps of the previous call into its history structure.

4.4 Recovery of subscripted variables

Subscripted variables cannot be handled efficiently by extending the basic scheme for scalars if any such variables are involved in an infinitely critical statement. This is because a save point for the array will involve copying the entire array into the history structure. This can slow down the program being debugged unacceptably.

However, if this is not the case, the basic scheme can be extended in the following manner to accommodate subscripted variables. Consider an assignment $a[i] := ...x...b[j]...$. To recover the value of $a[i]$ according to this assignment, procedure *Recover_value* should proceed as if the assignment is of the form $a := f(i, ...x, ..., j, b[j])$. This approach ensures that i, j and $b[j]$ will have appropriate values when the previous value of $a[i]$ is computed. Compared to the recovery of scalar variables, two new issues arise in such recovery. First, predicate $Same\_value(b[j], p0, Bkpt)$ will need to decide whether the value of $b[j]$ has changed between points $p0$ and $Bkpt$. This will require examination of all assignments of the form $b[k] := ...$ for some k to determine whether $b[k]$ and $b[j]$ refer to the same element of b. To be conservative, *Same_value* should return false if the debugger is unable to determine whether the value of $b[j]$ has changed between points $p0$ and $Bkpt$. Second, while recovering the value of $a[i]$, $Previous\_defn$ should examine every assignment of the form $a[k]$ and decide whether it is an assignment of $a[i]$. Alias information should be used during this analysis.

Criticality analysis should similarly consider a statement $a[i] := ...x...b[j]...$ to be of the form $a := f(i, ...x, ..., j, b[j])$. Thus apart from nodes containing assignments to $b[j]$, nodes containing assignments to i and j may also become critical nodes. Nodes containing assignments of the form $b[k] := ...$ for some k should also be considered unless it can be inferred during instrumentation time (using static analysis) that k cannot have the same value as the value of j when the assignment to $a[i]$ is executed.

4.5 Performance

The following figures were obtained on benchmark programs.

Program	w/o Instru	with Instru	Overhead	Size
Dhrystone	0.85	0.93	09	17
Hanoi	14	19	35	14
mcf	165	190	15	318

The first column is the name of the program. The second column gives the execution time of the program without instrumentation. The third column gives the execution time of the program with instrumentation. The fourth column gives the calculated percentage overhead in terms of execution time. The fifth column gives the total size of history structures used in the program in words(4 bytes).

Chapter 5

Design

The implementation makes use of the lcc compiler and GENSAT data flow analysis framework. An IR (Intermediate Representation) and the program flow graph (PFG) is generated by GENSAT for each function in the program as an intermediate output. These form the input to our system.

5.1 High Level Design

The C program is passed to lcc. The output of lcc is fed to GENSAT which generates the IR (Intermediate Representation) file, the PFG file and the symbol table file.

The IR of each procedure is loaded into memory in the form of an AST (Abstract Syntax Tree). Some code is inserted at the head of each node (for analysis of reference parameters). The function calls in each procedure are replaced by deemed definitions. These modifications to the IR are performed by the Transform class.

The algorithm given in the previous chapter is then applied to each procedure individually and the history structures sizes and instrumentation code to be inserted is computed.

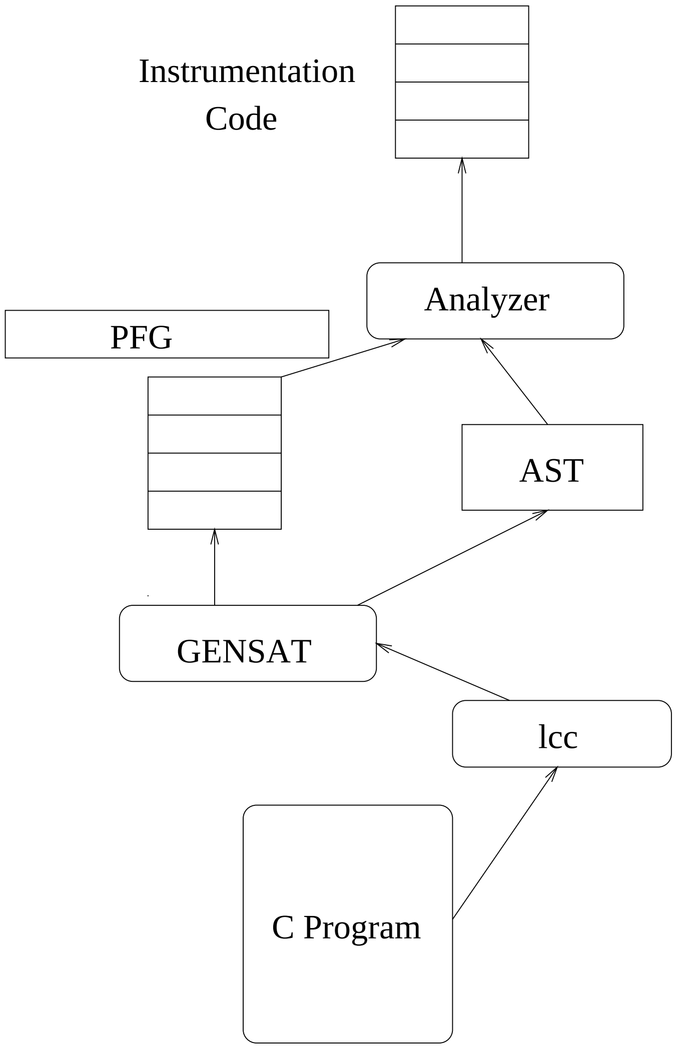

Figure 5.1: Input and Output of the System

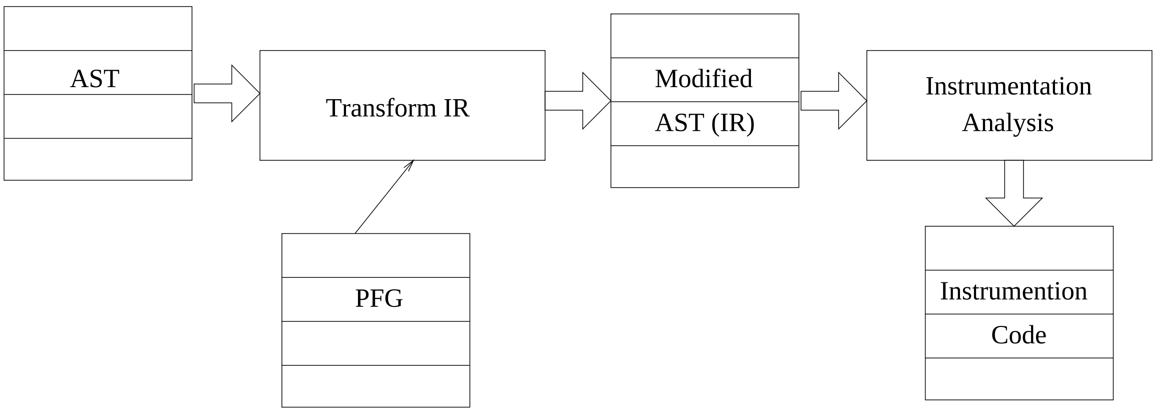

Figure 5.2: High Level Design

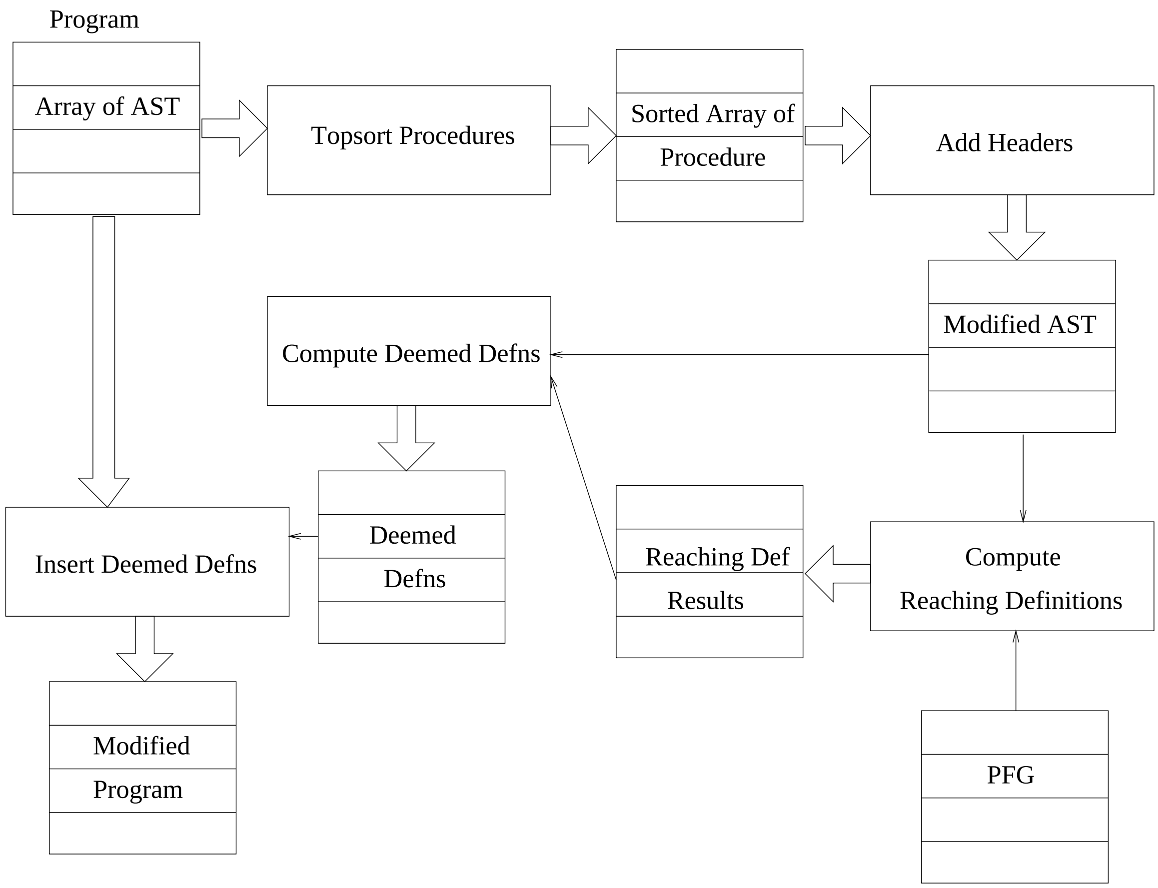

Figure 5.3: InterProcedural Analysis and Transformation

5.2 InterProcedural Analysis and Transformation

For every function some code is inserted into the start of the function that copies actual parameter variables into formal local variables. This is required so that the actual parameter variables are not falsely analyzed as modified, during reaching definitions analysis.

The entire program is topologically sorted, so that the callee functions are analyzed before the caller functions. The modified AST is fed to a *reachDefn* object that performs reaching definition analysis on the AST. This reaching definition info is used to compute deemed definitions.

After deemed definitions of all functions have been computed, calls to the function in a function body are replaced by the corresponding deemed definitions of the parameters that are passed by reference.

5.3 Classes used

The following classes are used.

- AST (Abstract Syntax Tree)

  This is the representation of the program in IR form. The AST is loaded from the IR file generated by GENSAT. It is organized as an array of statements. Each statement is an expression. Assignments statements are also considered to be expressions. The expression tree is composed of ASTNode objects. The AST also assigns unique node numbers to each ASTNode object in the AST.

- ASTNode

  ASTNode represents one node in an expression tree. It has three attributes.

  1. instr
     The instr attribute stores the opcode of the instruction.
  2. syms
     A symbols array (of size 2) stores the operands if any. The operands are typically entries from the symbol table, typically variable names and labels etc.
  3. kids
     A kids array (of size 2) thats stores the node number of the child ASTNode object.

- PFG

  This is a representation of the program flow graph. It contains an array of *Basic Block*.

- BasicBlock

  It has the following attributes

   1. The list of predecessors of this Basic block. Each entry in the list is an index into the array of *BasicBlock* in the *PFG*
   2. The list of successors of this Basic block. Each entry in the list is an index into the array of *BasicBlock* in the *PFG*
   3. The node numbers of all the *ASTNode* objects contained in that particular *Basic Block*

- Procedure

  This class encapsulates all the information regarding a particular procedure. This class has the following attributes.

   1. name

      The name of the procedure.
   2. theAST

      The AST of the corresponding procedure.
   3. thePFG

      The Program Flow Graph of the given procedure.
   4. params

      An array of strings that are the formal parameters that are passed to the procedure.
   5. sym

      A reference to the global symbol table.
   6. readfile()

      This method loads a procedure from the filename passed.

- WholeProg

  An object of this class encapsulates the whole program. It has the following attributes.

   1. proclist

      This is an array of *Procedure* objects.

2. symtable

   This is a flat symbol table for the whole program. All *Procedure* objects will contain a reference to it.

3. readproclist()

   This method is passed a filename that maps function names to the corresponding IR and PFG files. The method then proceeds to load all procedures in the program into memory.

4. readvar()

   This method is passed a filename that contains the global symbol table. The symbol table is read into memory.

- Data Flow Functions

  1. computelocalalias

     This function computes the local aliasing information in a basic block.

  2. DataFlow

     This function uses the local alias information and computes global aliasing properties using the algorithm in [8].

  3. locpropbuild

     This function builds the local reaching definition properties, taking into consideration pointer aliasing. It uses the previous two functions for alias analysis.

  4. forwardDataFlow

     This function performs forward bitvector data flow analysis on a pfg, given the local properties.

- reachDefn

  This class contains a reference to the *AST* and and *PFG*. It is passed as a parameter to an object of this class on construction.

  The class computes reaching definitions, along with pointer alias analysis. It uses the functions *locpropbuild* and *forwardDataFlow* for this purpose. The reachingdefinition object provides two methods.

1. def(int stmtno,const string s)

   This method returns a vector of statement nos that is a definition of variable s, that reach the statement stmtno statically.

2. indirdef(int stmtno,const string s)

   This method returns a vector of statement nos that is a definition of variable pointed to by pointer s, that reach the statement stmtno statically.

- DeeemedDefn

  An object of this class is passed a reference to a *Procedure* object on construction. It uses this to construct a *reachDefn* object that makes available the reaching definition information required to compute the deemed definition.

- Transform

  The *Transform* class adds parameter header code to the start of the AST of each procedure.

- SameLoop

  An object of this class takes *PFG* as a parameter on construction. It uses the algorithm in [9] to compute same loop information in almost linear time.

- Instrument

  This class is used during the code instrumentation phase. An object of this class takes a *Procedure* object as parameter on construction. It constructs a *reachDefn* object that provides it with reaching definition information and a *SameLoop* object that provides same loop information. This class implements the algorithm in chapter 4 to perform criticality analysis and code instrumentation.

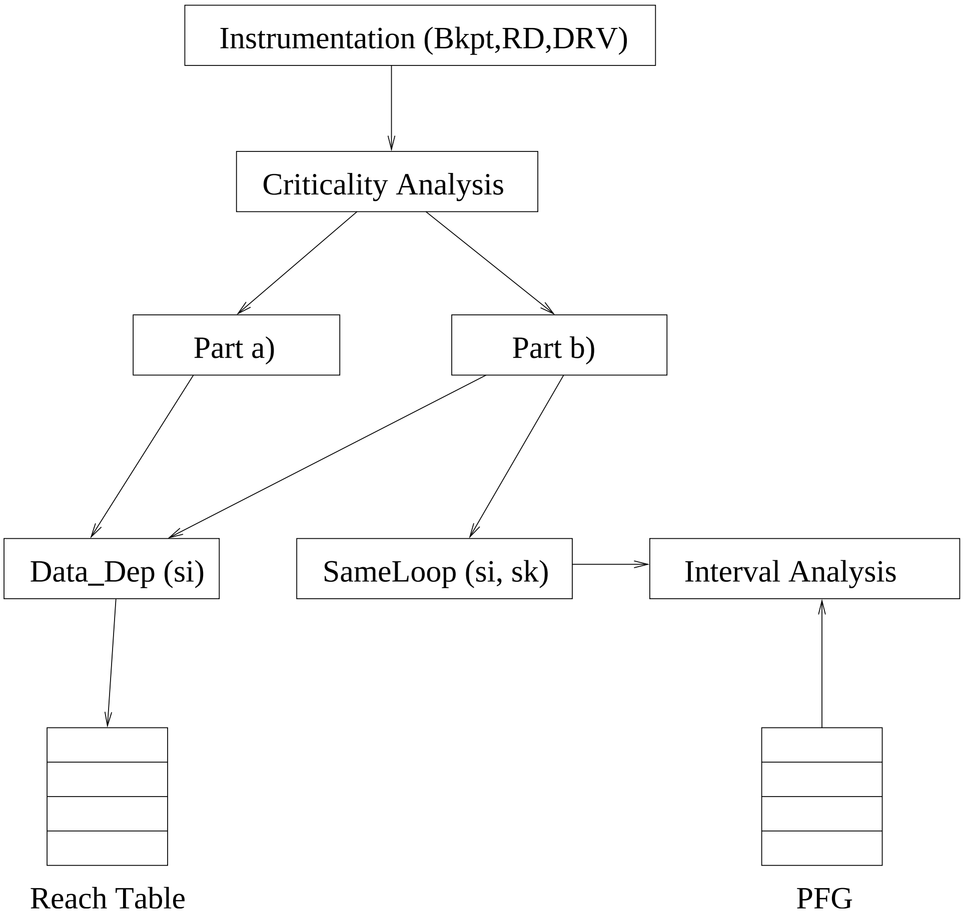

Figure 5.4: Criticality Analysis and Instrumentation

Chapter 6

Conclusion

In this project, we explored the design of a debugger for optimized code that reports the expected value of a variable to the user through value recovery and the use of execution trajectories. The scheme was extended to provide an additional facility of retrospect to the debugger. Both intraprocedural and interprocedural issues were studied. A system was developed using lcc and GENSAT that computed the history size requirements and instrumentation code required for an input program to provide such a facility. Benchmark programs were also tested.

For future work, as part of the project, the following areas can be studied in further detail. The possibility of reduction of execution overhead by optimizing the recording of the execution trajectory can be explored. From the implementation point of view, a complete debugger that performs value recovery could be developed.

Appendix A

LCC, ASDL and GENSAT

A.1 LCC

LCC is a retargetable compiler for ANSI C [10]. The target-independent front end and target independent back ends are packaged as a single program tightly coupled by a compact and efficient interface. Lcc is small, at least in comparison with other compilers, because it omits some components, most notably a global optimizer.

As shown in Figure A.1, lcc is –by design– a *monolithic compiler*: the front end and the back ends are combined into a single address space, so the front and back ends communicate by procedure calls that exchange pointers to shared data structures read and written by both parties, as Fig. A.1 illustrates. Also, back ends can make upcalls to functions provided by the front end, and the front end reads data structures written by the back ends.

ASDL facilitates an easy spliting of lcc into separate programs with no change to the code-generation interface. So, the existing back ends can be used unmodified.

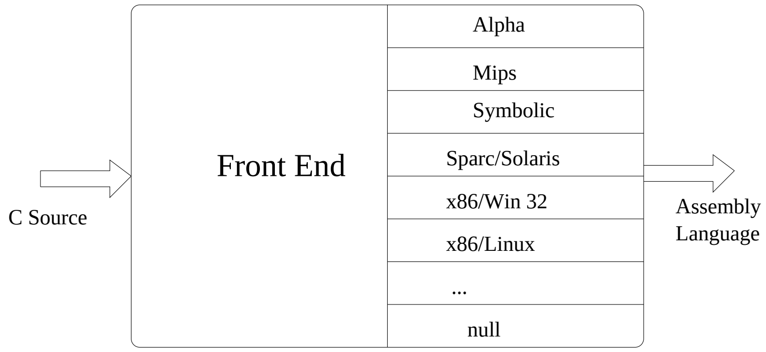

Figure A.1: Monolithic design of lcc

A.2 Abstract Syntax Description Language

ASDL is a small, domain-specific language for describing tree data structures. ASDL specifications are concise and independent of any particular programming language. The ASDL generator, asdlGen, accepts an ASDL specification and emits code that defines a concrete representation for the data structures described in the specification, along with code that constructs , reads, and writes instances of those data structures. Currently, asdlGen can emit data-structure implementations in C, C++, Java, ML, and Haskell.

Figure A.2 depicts a revised design. The front end, rcc, emits a pickle that encodes all the data structures and the function calls made when compiling a C source file. Essentially, rcc converts the lcc IR to an IR defined by the ASDL grammar. The new program, dubbed 'pass2,' reads a pickle, recreates the internal data structures, and makes the function calls encoded in the pickle. That is, pass2 converts the ASDL-defined IR back to the lcc IR. The generated assembly language is often byte-for-byte identical to the code emitted by the monolithic compiler. Differences occur only when the back end calls the label generator, which causes the revised design to number labels differently.

A.3 GENSAT

Static analysis of programs is a fundamental technique used in a variety of language processing tasks. A program transformation tool (PT tool) typically uses

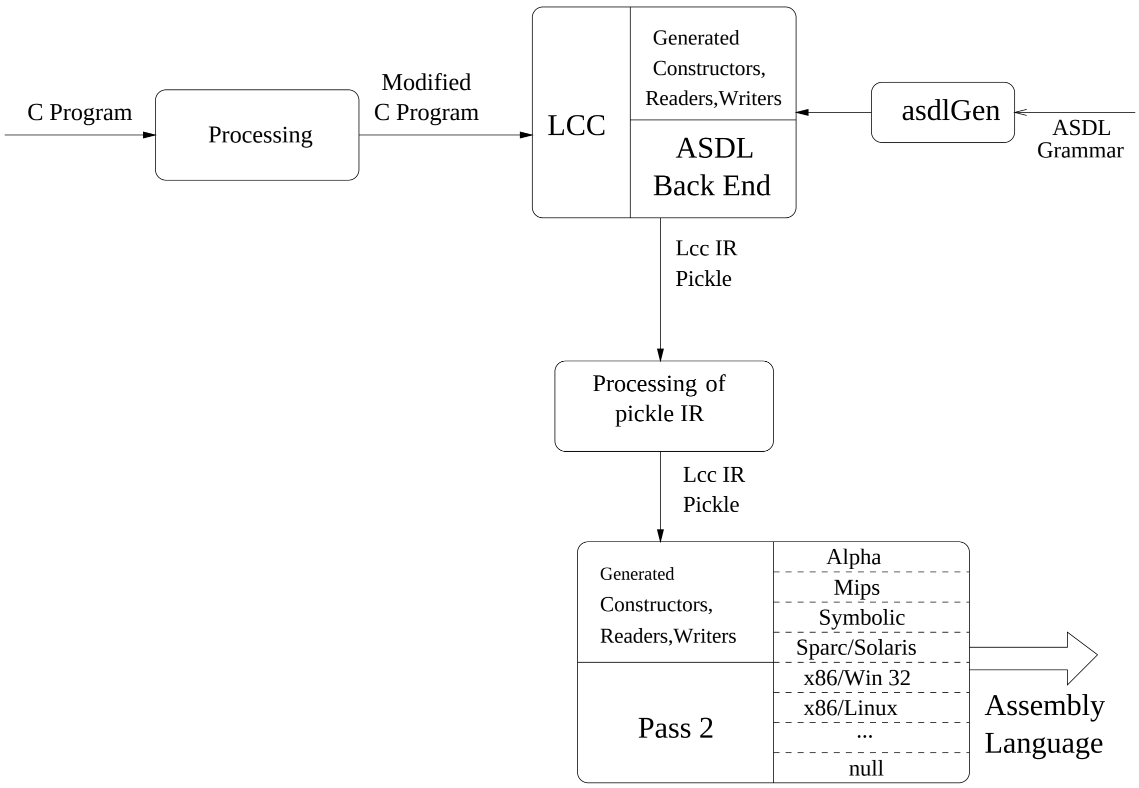

Figure A.2: Revised design of lcc using ASDL different kinds of static analyses to support its activities.

GENSAT, a generator for Static Analysis and Transformation (SAT) programs facilitates in selecting a set of static analyses to support the language processing activities. GENSAT visualizes that the SAT programs generated by it would be used in conventional Front end - Transformer - Back end model. The front end of the PT tool constructs the abstract syntax tree (AST) of the input program and produces an intermediate representation (IR) of it. This is a *pre-order* listing of AST. The SAT programs read in the IR and construct

- the corresponding AST, and
- the program flow graph (PFG) for the input program.

Static analysis is performed on the program flow graph to collect information of interest. Fig. A.3 contains a schematic of the GENSAT environment.

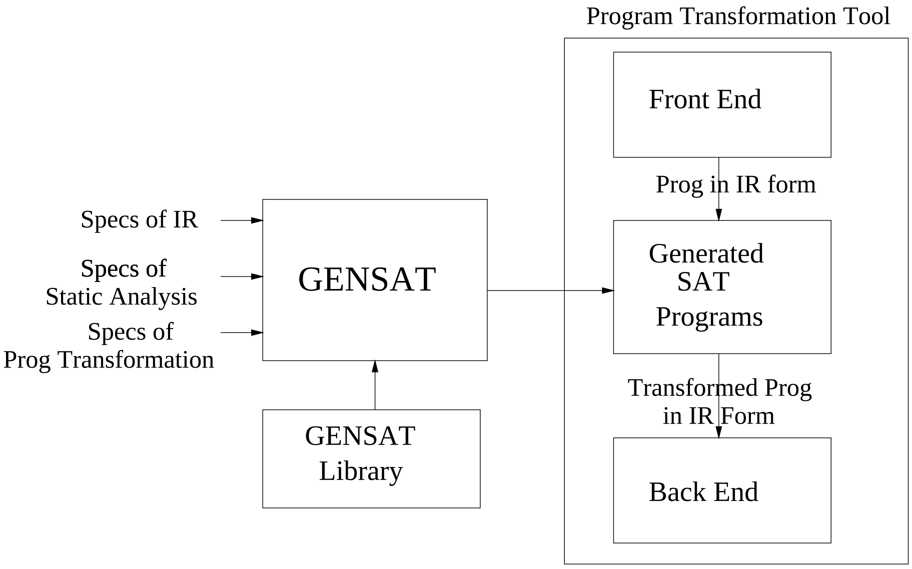

Figure A.3: The GENSAT Environment

GENSAT provides a specification language called GENPLUS to specify the details of the IR, the static analysis and the program transformations. A sample IR specifications are described in subsection A.4. By analyzing these specifications, GENSAT generates a suite of C++ programs, collectively called SAT programs to process the IR and perform program transformation. The SAT programs generated by GENSAT consists of:

- Program to read in the IR of a program,
- Program(s) to perform data flow analysis and transformation, and
- Program to write out the transformed program in the IR form.

In our implementation, we use GENSAT for obtaining the PFG of the input program. GENSAT also provides us with reaching definition analysis information.

A.3.1 A Sample IR Specification

GENSAT provides a generic format for describing the IR. Each node in the tree has an operator which identifies the type of the node, some attributes and pointers to its children. Fig. A.4 shows some sample IR specifications. The exact relationship between the basic blocks of the PFG and the subnodes of a specific IR node is determined by the following control flow semantics of the IR node:

- A basic block in the PFG consists of a sequence of nodes and sub-nodes in the IR.
- The sub-nodes of an IR node may belong to the same or different basic block of the PFG.

The block description part of the IR node description specifies whether the sub-nodes of an IR node belong to the 'current' basic block being constructed, or whether a new basic block should be started. The keyword 'CONTINUE' indicates that the next IR node is to be added to the current basic block, while 'NEW' indicates that the next IR node should belong to a new block. The control description also specifies the control flow of a particular IR construct with respect to its textual continuation 'C'. The operator '.' in the control-flow description indicates sequential flow and '| |' indicates alternatives in the execution.

```
%operator
ASGNI SEQ_OP JUMP
```

```
%start Prog

%%
Prog : Seq_op_rec Stmt Stmt
        {
        BLOCK_DESC :
                $1 : (CONTINUE,CONTINUE)
                $2 : (CONTINUE,CONTINUE)
                $3 : (CONTINUE,CONTINUE);
        CONTROL($$,C) = $1.$2.$3.C
        }

Seq_op_rec : SEQ_OP [KEY]

Stmt : Asgn_stmt
     | Jump_stmt
     | Conditional_jump_stmt
     | Label_stmt
     | Return_stmt ;


Asgn_stmt : Asgni
        | Asgnf
        | Asgnp
        | Asgnu
        | Asgnb ;
```

```
Asgni : Asgni_op_rec Reg Reg
        {
        BLOCK_DESC :
                $1 : (CONTINUE,CONTINUE)
                $2 : (CONTINUE,CONTINUE)
                $3 : (CONTINUE,NEW) ;
        CONTROL($$,C) = $1.$2.$3.C
        }

Asgni_op_rec : ASGNI [KEY]

Conditional_jump_stmt : Jump_eq
        | Jump_ge
        | Jump_gt
        | Jump_le
        | Jump_lt
        | Jump_ne ;

Jump_eq : Jump_eq_rec Reg Reg Jump_stmt
        {
        BLOCK_DESC :
                $1 : (CONTINUE,CONTINUE)
                $2 : (CONTINUE,CONTINUE)
                $3 : (CONTINUE,NEW)
                $4 : (NEW,NEW);
        CONTROL($$,C) = $1.$2.$3.($4 || C)
        }

Jump_eq_rec : EQI [KEY]
```

Figure A.4: Sample IR Specification

Appendix B

Sample Results

The following is a sample input program to the system. The function *critic* is similar to the standard examples considered in the report. The *main* function calls the functions *fun* and *fun1*. The system replaces these function calls by deemed definitions and performs the analysis.

```
void critic()
{
        int a,b,x,v ;

        a = 5 ;
        while(1)
        {
                b = 7 ;
                v = a ;
                if(v > 3) a = b ;
                v = x ;
        } ;
}

void fun1(int x,int *p,int *r,int *s)
{
```

```
        int *q ;
        x = x*x ;
        *s = *s + 2 ;
        while (x > 100)
        {
                x -= 10 ;
                *r = x ;
        }
        r = p ;
        if (x) q = p ;
        q = p ;
        *q = x ;
        if (x) *p = x - 10 ;
        *r = *s ;
}

void fun(int x,int *p)
{
        int *q ;
        x = srand() ;
        while (x > 100)
        {
                x -= 10 ;
        }
        q = p ;
        *q = x ;
        *p = x - 10 ;
}

int main()
{
        int a,b,c,d,e ;
        while (b < 9)
        {
```

```
                b = b + a ;
                fun1(b,&c,&d,&e) ;
                if (e > 5)
                {
                        b = d + 1 ;
                }
                fun(c,&d) ;
        }
        return 0 ;
}
```

The results list the history sizes, infinitely critical nodes, and the instrumentation code in that order for each function.

```
RESULTS critic

History size : v = 3
History size : a = 2
History size : b = 1


Node : 0 - Size : 1
COPY(a,0:0)
Node : 1 - Size : 0
Node : 2 - Size : 0
Node : 3 - Size : 1
COPY(b,0:3)
Node : 4 - Size : 3
COPY2(v,1,a)
COPY(v,0:4)
Node : 5 - Size : 0
```

```
Node : 6 - Size : 2
COPY1(a,1,b)
COPY(a,0:6)
Node : 7 - Size : 0
Node : 8 - Size : 0
Node : 9 - Size : 0
Node : 10 - Size : 0
Node : 11 - Size : 0



RESULTS fun1

History size : x = 3

InfCritical : 10 , x

Node : 0 - Size : 0
Node : 1 - Size : 0
Node : 2 - Size : 0
Node : 3 - Size : 0
Node : 4 - Size : 0
Node : 5 - Size : 0
Node : 6 - Size : 1
COPY0(x,1,x)
COPY(x,0:6)
Node : 7 - Size : 3
COPY1(x,1,x)
COPY1(x,2,x)
COPY(x,0:7)
Node : 8 - Size : 0
Node : 9 - Size : 0
Node : 10 - Size : 2
COPYval(x,1,x)
```

```
COPY(x,0:10)
Node : 11 - Size : 0
Node : 12 - Size : 0
Node : 13 - Size : 0
Node : 14 - Size : 0
Node : 15 - Size : 0
Node : 16 - Size : 0
Node : 17 - Size : 0
Node : 18 - Size : 0
Node : 19 - Size : 0
Node : 20 - Size : 0
Node : 21 - Size : 0
Node : 22 - Size : 0
Node : 23 - Size : 0
Node : 24 - Size : 0



RESULTS fun

History size : x = 2

InfCritical : 6 , x

Node : 0 - Size : 0
Node : 1 - Size : 0
Node : 2 - Size : 0
Node : 3 - Size : 1
COPY0(x,1,srand)
COPY(x,0:3)
Node : 4 - Size : 0
Node : 5 - Size : 0
Node : 6 - Size : 2
COPYval(x,1,x)
```

```
COPY(x,0:6)
Node : 7 - Size : 0
Node : 8 - Size : 0
Node : 9 - Size : 0
Node : 10 - Size : 0
Node : 11 - Size : 0
Node : 12 - Size : 0



RESULTS main

History size : b = 3
History size : d = 2
History size : c = 3
History size : e = 2

InfCritical : 2 , b
InfCritical : 7 , e

Node : 0 - Size : 0
Node : 1 - Size : 0
Node : 2 - Size : 2
COPYval(b,1,b)
COPY0(b,2,a)
COPY(b,0:2)
Node : 3 - Size : 0
Node : 4 - Size : 0
Node : 5 - Size : 0
Node : 6 - Size : 0
Node : 7 - Size : 2
COPYval(e,1,e)
COPY(e,0:7)
Node : 8 - Size : 2
```

```
COPY1(d,1,d)
COPY0(d,2,b)
COPY(d,0:8)
Node : 9 - Size : 3
COPY2(c,1,e)
COPY(c,0:9)
Node : 10 - Size : 0
Node : 11 - Size : 0
Node : 12 - Size : 3
COPY2(b,1,d)
COPY(b,0:12)
Node : 13 - Size : 0
Node : 14 - Size : 0
Node : 15 - Size : 0
Node : 16 - Size : 1
COPY(d,0:16)
Node : 17 - Size : 0
Node : 18 - Size : 0
Node : 19 - Size : 0
Node : 20 - Size : 0
Node : 21 - Size : 0
```

Bibliography

[1] D.M. Dhamdhere, Sachindra Joshi. Debugging and Exception Handling of Optimized Code Using a Selective Execution Trajectory. Computer Science and Engineering IIT Bombay,2000.

[2] D. M. Dhamdhere. Effective execution histories for debugging and slicing. Computer Science Engineering, IIT Bombay, 2000.

[3] D. M. Dhamdhere and K. V. Sankaranarayanan. Dynamic currency determination in optimized programs. ACM Transactions on Programming Languages and Systems, 20(6):1111 - 1130, 1998.

[4] D.M. Dhamdhere, Sachindra Joshi. Optimization in the presence of exception handling and debugging. Computer Science and Engineering IIT Bombay,2000.

[5] C. Jaramillo, R. Gupta, and M.L. Soffa. FULLDOC: A Full Reporting Debugger for Optimized Code, 7th International Static Analysis Symposium, LNCS 1824 , Springer Verlag, pages 240-259, Santa Barbara, CA, June/July 2000.

[6] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, Harini Srinivasan. Dependence Analysis for Java. LCPC 1999: 35-52.

[7] M. Gupta, J.-D. Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00), 2000.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers-Principles, Techniques and Tools, Addision-Wesley, 1986.

[9] G. Ramalingam. Identifying Loops In Almost Linear Time. ACM Transactions on Programming Languages and Systems 21(2) : 175 - 188, 1999.

[10] Christopher W. Fraser, David R. Hanson, A Retargetable C Compiler: Design and Implementation, The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1995.