Unreachable Code Identification for Improved Line Coverage

Luke Pierce, Spyros Tragoudas
Department of Electrical and Computer Engineering
Southern Illinois University Carbondale
Email: { lukepier, spyros }@siu.edu

***Abstract*—It is shown that line coverage during verification is significantly enhanced when applying efficient methods to remove unreachable code so that time is not spent developing a set of test vectors to cover unexecutable code. Identification of unreachable code through examination of arithmetic operations and branch constraints has previously only been achieved through local examination of surrounding instructions. In this paper a method for unreachable code identification is presented. The method models software arithmetic operations and conditions as a set of Boolean constraints for which reachability is determined using a satisfiability solver, and unreachable code can identified by considering all relevant instructions preceding it.**

***Index Terms*—Dead Code; Unreachable Code; Static Code Analysis; Boolean Satisfiability; Line Coverage**

I. Introduction

The impact of unreachable code removal on line coverage is demonstrated with experimental results on the Mälardalen WCET benchmarks [10]. Removal of unreachable code is important for code coverage verification. Code coverage is typically performed by executing a set of test vectors on a given program and recording the lines which were executed [5]. Line coverage is the union of all lines of code executed for each test vector applied. The test vectors of a program are often generated manually by software developers with the intent of maximizing the line coverage. The task requires examination of the logic of the code and is time intensive to complete. Automated methods for maximizing code coverage have been explored but remain unreliable in practice [14].

Given the sophisticated circumstances to which code can become unreachable, developers may spend an extended amount of time trying to determine if there exists a test vector which can excite a segment of code. Without automated tools which can guarantee a code segment is unreachable, application of an exhaustive set of input vectors maybe required to prove the code is unreachable. Given the exponential set of possible inputs, exhaustive testing is time prohibitive. Furthermore, the presence of unreachable code can indicate improper implementation of code specifications, which can lead to errors during execution.

This material is based upon work supported by the National Science Foundation under Grant no 1361847. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Determination of unreachable code is challenging given there are exponential number of software paths to the number of control flow statements. To add to the difficulty, there is no known method to efficiently store the functionality of each software operation to later determine if a branch is satisfiable.

Thorough identification of unreachable code requires examination of all relevant operations leading up to a branch. The result of an arithmetic operation can indirectly trigger a branch condition to be unsatisfiable after the resultant is used in several other arithmetic operations for determination of variables that the condition is dependent on. From this it is possible that the sequence of instructions which causes the unreachable code maybe very long.

In this paper we present a novel method which models the functionality of the program as a set of boolean constraints. The set of constraints creates estimates and bounds for the functions, and does not require the storage of exact functions. As such, the method proposed allows for detection of path dependent non-executable branches, and can scale for modern software.

In [4], symbolic simulation has been combined with Boolean Satisfiabilty for detecting unreachable paths. Other techniques proposed in [13], [18] store boolean functions representing each operation in the program. However, these methods are constrained for variables consisting of few bits.

Integer linear programming satisfiability techniques have been implemented to rapidly determine if a set of constraints has a solution [17]. This can allow for non-executable code detection in programs without multiplication. More generalized solutions are Satisfiability Modulo Theories (SMT) Solvers such as Z3 [6] and Yices [8] which can be used to determine satisfiability of complex polynomial functions. However, computation performance for determining the satisfiability of non-linear integer arithmetic is considered undecidable. As such in practice finding an exact solution is not always feasible. Theorem solvers have also been applied in [12] to find unreachable code. However, in this technique the code required explicit annotation to aid in the determination of unreachable code.

Constraint based techniques have been used in static timing analysis in software to remove infeasible paths [11]. The con-

straints are formed when dependencies on data and conditions are found. The constraints are then processed to determine when a path is feasible. However, this method relies on variables being initially set to a constant value, which is often not the case.

In the area of compiler optimization, the problem of finding unreachable code has been explored through the use of range propagation [1], and constant propagation [19]. In range propagation [1], a lower and upper bound is computed for each variable. Using the bounds, detection of some unreachable code is possible if there exist a branch condition where the bound constraints and the branch constraint evaluate to a tautology. Constant propagation [19] is a method of determining whether the value of a variable can be determined at compile time. Originally used for identifying operations which can be computed during compile time instead of runtime, it has been extended to also identify unreachable branches. However, this method requires at least one of the operands to have a fixed value at compile time to find unreachable code, while the proposed algorithm has no such requirements. Furthermore, the proposed algorithm allows for multiple complex ranges for variable functions which occurs on path convergences.

Concolic testers such as KLEE have been used for generation of branch functions [3]. These use concrete values along with SMT solvers to generate functions for each operation and determine appropriate test vectors for each branch. While these tools are very good at generating test vectors to sensitize a path, they cannot guarantee code is unreachable when a test vector cannot be generated to sensitize a branch.

Details of the proposed method are given in Section II. The approach was tested under a variety of benchmarks shown in Section III. Finally, Section IV concludes the proposed method.

II. Unreachable Code Identification and Removal

It has been shown that the time required to determine whether a set of constraints can be satisfied is less when the constraints are abstracted as a boolean function [15]. As such, the proposed method transforms a program's arithmetic operations into a set of boolean constraints which model relationships between variables. For instance, it can often be inferred that one variable is greater or less than another under a set of conditions.

A branching operation applies additional constraints, which may conflict with constraints placed on a variable the condition is dependent upon. In this paper, a condition is modeled as a boolean function which is dependent on the variable operands, and the functionality of the operators. A boolean condition is said to be satisfiable if there exists an input for which the boolean function results in a true value. In the event there exists conflicting constraints and no input can satisfy the condition, the code contained within branch can be said to be unreachable. To achieve this, constraints are transformed into the Conjunctive Normal Form (CNF), and a boolean satisfiability solver is used to determine if the set of constraints can be satisfied.

A program is represented as a cyclic control flow graph where a vertex is a set of instructions for which the order of the execution always remains the same. Edges between vertices are generated from the presence of branching statements. A binary branching statement creates two edges to separate vertices depending on the true and false branches. Each program has a vertex representing the start of execution, v_s, and a vertex representing the exit of the program, v_e. The number of paths in a program grows exponentially to the number of branches since paths represent combinations of connected vertices which start at v_s and end at v_e. A path in a program is said to be sensitizable if there exists an input which causes the software path to be executable. A segment of code is considered unreachable if there exists one or many paths from the v_s to a particular vertex, v_i, but none are sensitizable.

The method is initially described using source code without feedbacks which can be represented as a control flow graph. The graph allows for a topological traversal to systematically generate boolean expressions and determine satisfiability.

The proposed method stores the relationships between variables as a set of relative linear constraints which offers a more scalar solution for determining satisfiability. Relationships can be modeled as a set of constraints shown in Table I. Constraints for a particular variable are created with respect to another variable, or a constant. Suppose there are a set of variables in a program, $\{x^1, x^2, ... x^n\}$, each variable can be represented as a m-bit vector such that $\{\{x^1 = X_1^1 X_2^1 ... X_m^1\}$, $\{x^2 = X_1^2 X_2^2 ... X_m^2\}$, ... $\{x^n = X_1^n X_2^n ... X_m^n\}\}$, where m is minimally the log of the number of relationships between a particular variable and another. A relationship between two variables can be described using a linear constraint such as $X^1 > X^2$ and stored using a comparator boolean logic function. Multiple relationships between variables can be modeled similarly as partially ordered sets, and can be stored using a conjunctive operator such as, $(X^1 > X^2) \wedge (X^1 < X^3)$. The conjunction of a partial ordered sets can create logical conflicts such as $(X^1 > X^2) \wedge (X^1 < X^3) \wedge (X^2 > X^3)$ which cannot be satisfied. Such logical conflicts often arise when executing within a series of nested branches, and are sufficient for identifying unreachable code.

When the constraints are stored as a set of boolean functions, a boolean SAT solver can be used to determine if a set of constraints is satisfiable, and whether the path can be executed. While the SAT solver will return back a vector of values for each variable, the values only indicate there exists a set of values which can satisfy the constraint, and the ordering of the variables, but not what the exact values are. This is due in part to the mapping function between large bit vector of a program variable, and its' smaller bit vector representation in this model. Furthermore, as will be shown, arithmetic functions are modeled using partially ordered sets.

For instance, suppose a source code has a branch condition $a < b$. The constraint can be stored as a $A < B$, and A, B can be a bit vector of length 1. A satisfying set of values would be $\{A_1 = 0, B_1 = 1\}$. Given that a and b are multi-bit datatypes of a program, there are a large range of satisfying solutions. The satisfying set of values $A_1 = 0$ and $B_1 = 1$ merely indicates the satisfying order to the partial set of a and b, i.e. that any value where $a < b$ is a satisfying value.

Removal of the requirement of having to express real decimal values for the SAT formulation has three major advantages. Requirements of each variable's vector length are reduced, different number representations such as floating point are abstracted, and modeling negative numbers requires no additional logic. The length of the bit vector is dependent on the number of dependencies with other variables. The length must allow for a unique binary encoding for each variable it is associated with. For instance, to determine if $a < b < c$, the bit vector for each variable must be at least two, otherwise it will not be possible for variable b to fall between two other variables. Even with the need to express all dependencies, the bit vector length will be less than the requirement of having to express all possible decimal numbers. Variables can also be constrained by a set of constants through representing the constant number as a variable.

Example 1: Suppose a set of branch conditions is the following, $a > -2$, $b < -4$, and $b > a$. The constant -2, and -4 shall be represented as ψ_1 and ψ_2 respectively. The constraint, $a > -2$ can be represented as $a > \psi_1$, and $b < -4$ as $b < \psi_2$. To model the constants as a variable, the relationship between the constants must be expressed. For this example, the relationship can be expressed as the constraint $\psi_1 > \psi_2$. For the final satisfiability formulation, the constraints can be expressed as Equation 1.

$$F = (B > A) \wedge (A > \psi_1) \wedge (B < \psi_2) \wedge (\psi_1 > \psi_2) \quad (1)$$

The set of constraints is unsatisfiable for Example 1. □

An arbitrary number of constants can be generated so long as the relationships between the constants are stored as a function. In practice this function, Ψ, grows linearly to the number of constants given the constraints are stored as $\psi_1 < \psi_2 < ... < \psi_n$ which can be expressed by Equation 2.

$$\Psi = \bigcap_{i=1}^{n-1} \psi_i < \psi_{i+1} \quad (2)$$

As a minimal, the method proposed requires use of three constants to evaluate conditions; zero, negative one, and one, to properly predict the outcomes of most arithmetic operations. Storing arithmetic operations represents a challenge since many operations require a prohibitive number of boolean terms to be represented. As such, the arithmetic operations are modeled as a set of inequalities which are dependent on relationship with a few constant terms. For instance, if the operation is addition, and the sign of the operands is known, a set of constraints can be inferred. For instance if one operand is positive, and the other is negative it can be inferred the result must fall between the two operands. The sign of a variable can be tracked by inserting a variable which represents the constant zero.

The implication constraints are shown in Table I. The arithmetic operation is shown in column 1. Columns 2 and 3 in the table represent the operand conditions for which the third column, the implication constraint, exists. It is notable that Table I does not contain a subtraction operation. Since the addition operation already shows all possible combinations of positive and negative signs. For instance, $A - B$, can be changed to $A + (-B)$.

Lets consider the condition for operands A and B to be α^A and α^B respectively and the associated constraint to be γ. For a particular operation there are several sets of conditions and inferred constraints represented as $\{\{\alpha_1^A, \alpha_1^B, \gamma_1\}, \{\alpha_2^A, \alpha_2^B, \gamma_2\}, ..., \{\alpha_n^A, \alpha_n^B, \gamma_n\}\}$. For a particular arithmetic operation, the union of the condition/constraint sets is then intersected with the operand functions to generate the constrained final function as shown in Equation 3.

$$F = F_A \wedge F_B \wedge \bigcup_{i=1}^{n} \left(\alpha_i^A \wedge \alpha_i^B \wedge \gamma_i\right) \quad (3)$$

For testing satisfiability of a branch, Equation 4 is applied, where F_A and F_B are the functions of A and B respectively. The function $\omega(A, B)$ represents the logical constraints of the branch condition and Ψ is the function representing constant ordering. The functions F_A and F_B are built using linear relationship between one or many variables. Relationships generated using Equation 3 and Table I represent estimations for arithmetic operations for which models all possible satisfiable variable orders. The function $\omega(A, B)$ represents the boolean relational operation of the branch condition and the exact functionality can be stored. Therefore, the conjunction of the partially ordered sets of F_A, F_B, $\omega(A, B)$ and Ψ represent sufficient conditions for detecting satisfiability of the branch.

$$F_{cond} = F_A \wedge F_B \wedge \omega(A, B) \wedge \Psi \quad (4)$$

Example 2: Consider the source code in Listing 1. It can be seen that the branching statement $if(b > c)$ is a tautology which will always evaluate to false. This tautology is guaranteed since c is the sum of a and b and the statement $if(b < a)$ ensures that a is greater than four in the body of the outer most conditional. This leads to the implication that c is the sum of b and a number greater than four, and thus cannot be less than b.

```
1 input(a);
2 b = 4;
3 c = a + b;
4 if (b < a)
```

```
5  {
6      d = 0;
7      if (b > c)
8      {
9          d = 1;
10     }
11 }
```

Listing 1: Source Code for Example 2

While some variables such as b remain constant over the lifetime of a program and therefore are known at compile time, many variables are dependent on runtime input such as a and can only be known at runtime. The result is that it is not possible to know the exact value of c, and that it must be represented as a function which is dependent on the variables, a and b.

In practice, as the execution of a program progresses the number of variables a function depends on can grow to be very large, especially since each dependent variable can itself be function.

The proposed method creates a bit vector of unique variables for each source code variable declared. On variable initialization, if the variable is set to a constant value then a constraint to the most relevant variable constant will be generated and stored as the function for that variable. In this case, the constant zero is represented as κ.

Consider the source code of Example 2. Variable b in Listing 1 is constrained to be greater than κ since it is greater than zero. This function is shown in Equation 6. Variable a is also initialized, however since the value is dependent on user input it is unconstrained and set to the universal set as shown in Equation 5.

$$F_a = U \tag{5}$$

$$F_b = B > \kappa \tag{6}$$

Variable c is initialized using an arithmetic operation with the operands being a, and b. To generate the function for Example 2, Equation 3 is used which produces Equation 7. It can be seen that F_c contains all sets of conditions and constraints for the arithmetic operator. In this case, since F_a is unconstrained and F_b is constrained to be greater than κ, two constraints are satisfiable; $(A > \kappa)(B > \kappa)(C > A)(C > B)$ and $(A < \kappa)(B > \kappa)(C < B)(C > A)$.

$$\begin{aligned} F_c = ((A > \kappa)(B > \kappa)(C > A)(C > B) \vee \\ (A > \kappa)(B < \kappa)(C > B)(C < A) \vee \\ (A < \kappa)(B > \kappa)(C < B)(C > A) \vee \\ (A < \kappa)(B < \kappa)(C < A)(C < B)) \wedge \\ F_a \wedge F_b \end{aligned} \tag{7}$$

Similarly, when evaluating a branch, the condition is expressed as a boolean function and is intersected with the function of its operands. Equation 8 shows the function for the branch on Line 4 in Listing 1.

$$B_4 = (B < A) \wedge F_a \wedge F_b \tag{8}$$

The branch function is then evaluated with a Boolean Satisfiability (SAT) Solver. For Example 2, Equation 8 can be satisfied.

For nested branches, as the one that exists on Line 7 in Listing 1 of Example 2, the functions of all predecessor branch functions in the hierarchy must also be considered. These branches add further constraints to the variable inside the body and therefore the intersection of the branch function with the predecessor branch functions is taken as shown in Equation 9.

$$B_7 = (B > C) \wedge F_b \wedge F_c \wedge B_4 \tag{9}$$

For B_7 there is no input vector which will satisfy the function, and all statements contained within the body of the conditional can subsequently be removed. □

Table I: Arithmetic Operation Relationships

Operation	A	B	Implication
$C = A + B$	A $\geq$ 0	B $\geq$ 0	C $\geq$ A, B
	A $\geq$ 0	B $<$ 0	C $\geq$ B, C $<$ A
	A $<$ 0	B $<$ 0	C $<$ A, B
$C = A * B$	A $\geq$ 1	B $\geq$ 1	C $\geq$ A, B
	A $\geq$ 1	B $\leq$ -1	C $\leq$ B, A
	A $\leq$ -1	B $\leq$ -1	C $\geq$ A, B
	A $\geq$ 1	1 $>$ B $\geq$ 0	A $>$ C, C $\geq$ B
	1 $>$ A $\geq$ 0	1 $>$ B $\geq$ 0	A, B$\geq$ C, C $\geq$ 0
	1 $>$ A $\geq$ 0	0 $\geq$ B $>$ -1	0 $\geq$ C, C $\geq$ B
	0 $\geq$ A $>$ -1	0 $\geq$ B $>$ -1	1 $\geq$ C $>$ 0
$C = A \setminus B$	A $\geq$ 1	B $\geq$ 1	C $\leq$ A, C $\geq$ 0
	A $\geq$ 1	B $\leq$ -1	C $\leq$ 0
	A $\leq$ -1	B $\leq$ -1	C $\geq$ 0
	A $\geq$ 1	1 $>$ B $>$ 0	C $>$ A
	1 $>$ A $>$ 0	1 $>$ B $>$ 0	C $>$ A, B
	1 $>$ A $>$ 0	0 $>$ B $>$ -1	C $<$ B
	0 $>$ A $>$ -1	0 $>$ B $>$ -1	C $>$ 0

For multiplication operations, further information is required to be known about the operands to develop a set of constraints. The sign of the operand must be known, and whether an operand is a fractional number between zero and one or negative one and zero. This determines whether the result of the multiplication operation is greater than the input operands, or in-between the two operands.

The constraints for division operations are also listed in Table I. The constraints are looser since the relationship between the result and the operands can typically not be known. For instance, if $A \geq 1$ and $B \leq -1$ it cannot be known if the result will be less than or greater than B. The systematic generation of each instruction's functionality is shown in Algorithm 1. The process starts with each node in

Algorithm 1 Find Unreachable Code

Input: G, v

$C \leftarrow \emptyset$
ProcessNode(G, $\emptyset$, v, $\emptyset$
for each node, n in G **do**
  if n is not marked reachable **then**
    $C \leftarrow C \cup n$
  end if
end for
return C

the control flow graph, G, being marked as unreachable. The algorithm starts with an initial source node, v, and processed recursively using Algorithm 2. When a node is visited it is marked as reached. At the end of the algorithm any node that has not been marked as reached is added to the set C representing the final set of unreachable code. This code is then removed from the source file. The recursive process of

Algorithm 2 Process Node

Input: G, F, v

if v is a branch with condition c_v and $SAT(c_v)$ **then**
  for each neighbor w of v **do**
    $< G, F >$ = ProcessNode(G, F, w)
  end for
else if v is not a branch **then**
  mark v as reachable
  f_v = $getFunc(v, F)$
  $F = F \cup f_v$
  for each neighbor w of v **do**
    $< G, F >$ = ProcessNode(G, F, w)
  end for
end if
return G, F

traversing the control flow graph and generating functions is shown in Algorithm 2. The algorithm requires the inputs G, F, and v which is the input program's control flow graph, the list of current variables and their functions, and the current node respectively. The process starts by determining if the current node is a branch which contains a condition. In the case that the node is a branch, a satisfiability solver is used to determine if the branch constraint can be satisfied. If the branch cannot be satisfied, it is not processed any further and all nodes within the branch block are subsequently not processed or marked as reachable. If the branch is reachable, then the blocks of the branch statement are processed recursively.

In the case that the node being processed is an instruction, the $getFunc()$ function is used to determine the boolean function of the operation. This utilizes Table I and the techniques previously discussed. The modeled instruction's function is then stored in the function list, F.

The proposed algorithm works well for storing relations to other variables. However, when only three variables are used to represent constants, -1, 0, and 1 some limitations occur. For instance, if variable A is set to the value 2, and variable B is set to the value 3, the proposed method would fail in identifying the condition $A > B$ as unsatisfiable. While the method described above only uses three constants, it can be extended to identify these types of conditions through the use of additional variables representing the constants.

Cyclic code found in looping structures is a challenge given that a single traversal of the loop structure cannot definitively determine if a piece of code will be executed. For example, an *if* statement maybe labeled as unreachable during the initial pass of analysis on the loop. However, a variable the branch was dependent on could be modified later in the loop's body altering the reachability of the branch when the feedback function is considered.

```
1  a = -2;
2  b = 0;
3  while(b < 5)
4  {
5      if(a > 0)
6      {
7          c = a + b;
8      }
9      a++;
10     b++;
11 }
```

Listing 2: Cyclic Code for Example 3

The requirement for an instruction to be further processed on a subsequent iteration of the loop is dependent on the operands of the instruction. For instance, say $Z = X + Y$ is an instruction inside a loop. If both variables X and Y are not altered in the loop, then the function of Z is guaranteed to remain the same for each analysis pass. However, if X or Y were to be altered in the loop, the function of Z will require an additional pass of analysis to account for the feedback of the loop.

Using a data dependency graph (DDG), the functionality of the variables can be computed in a systematic manner. Loop independent variables, i.e. variables whose value does not change between iterations are the nodes in the graph which have no incoming edges. All other nodes are considered loop dependent. instructions are processed iteratively based on the order they appear in the DDG. This means that if Q is dependent on P, then the functionality of P must be computed prior to the functionality of Q. In the event a node only points to itself, i.e. an instruction such as i++, the functionality can often be estimated. However, for more complex cyclic dependencies in the DDG graph, the nodes within the cycle are assumed to be unconstrained and are no longer processed.

III. Experimental Results

The infeasible path analysis algorithm was implemented in Python 3.3. The logic manipulation and translation to conjunctive normal form was performed by the pyEDA library

Benchmark	LOC	Pos. of all injected code	# of Inserted Unreachable Code
Bs	114	NA	NA
E_Bs	123	79, 83, 105	3
Expint	157	NA	NA
E_Expint	168	35, 104, 105, 140	3
Janne Complex	64	NA	NA
E_Janne Complex	72	31, 43	1
Nsichneu	4253	NA	NA
E_Nsichneu	4257	123, 4162	2
Prime	45	NA	NA
E_Prime	52	24, 25, 30, 31	2
Sqrt	77	NA	NA
E_Sqrt	89	59, 83	1
Ud	163	NA	NA
E_Ud	180	156	1

Table II: Insertion Points of Unreachable Code

version 0.22.0 [7]. The satisfiability was performed by PicoSAT version 959 [2]. The static path analysis was performed using Ubuntu 13.10 with an Intel Core 2 Duo running at 3GHz.

The unreachable code analysis algorithm was applied to a subset of the Mälardalen WCET benchmark suite to verify the correctness and scalability of the proposed algorithm [10]. This benchmark suite is written in the programming language of C, and are used to determine scalability of the WCET static analysis tools. Currently there are no benchmark suites intended for unreachable code analysis. The subset of chosen benchmarks were picked since recursion is not currently supported by the software tool, and the selected benchmarks contained the most amount of supported operations.

For verification purposes of the proposed algorithm, software code, some of which is unreachable, was injected into the benchmarks at various parts of the code, often inside of loops. Guaranteed identification of all unreachable code in the current benchmarks is an intractable problem. As such, injection of unreachable code allowed for the quantification of proposed method's identification of known unreachable code. The injected code also served to enhance the benchmarks to generate more challenging conditional functions.

Table II shows the positions for which the unreachable code was inserted. Column 1 lists the name of the Mälardalen benchmark. Benchmarks listed with a prefix of *E_* are the enhanced versions of the original Mälardalen benchmark, which have been made available [16]. Column 2 lists the number of lines of code in the benchmark. Column 3 shows the line number for all code inserted, which are marked as not available for the original benchmarks. Not all code inserted is unreachable. Some of the modifications insert new variables, and arithmetic operations required for generation of more complex variable functions. Column 4 lists the number inserted unreachable code blocks in the given benchmark. Again, the original benchmarks are marked as not available.

A set of test vectors was applied to each of the enhanced benchmarks, and the line coverage during execution was measured using the program *gcov* [9]. Each set of test vectors consisted of 100 unique combinations generated at random. After applying the tool which removed identified unreachable code, the line coverage was measured again using the same set of test vectors. The results are shown in Table III.

Benchmark	w/ Unreachable Code	Removed Unreachable Code
E_Expint	93.18%	100%
E_Janne Complex	87.50%	95.65%
E_Nsichneu	51.71%	52.11%
E_Sqrt	93.55%	100%
E_Ud	87.50%	100%

Table III: Line Coverage Results

Table III shows the lines of code covered by the set of test vectors with unreachable code and with removed unreachable code. In column 1, the name of the benchmark which was measured is listed. Columns 2 and 3 state the percentage of lines of code executed during the application of the test vectors with unreachable code and with removed unreachable code respectively. For all enhanced test benches, the line coverage improved with an average increase of 6.9%. Notable is the marginal increase in E_Nsichneu code coverage which was caused primarily due to the small amount of unreachable code inserted compared to the amount of reachable code as can be seen in Table II. Line coverage was not always absolute when the unreachable code was removed due to the large number of possible inputs, and limited amount of applied test vectors.

Table IV shows the performance characteristics of running the unreachable code detection algorithm. In column 1 the name of the Mälardalen benchmark is shown. Column 2, shows the lines of code (LOC) for each benchmark. In columns 3 and 4, the execution time of the algorithm is given in seconds and the maximum memory space required is given in megabytes, respectively.

While the overall trend reveals that increasing the number of lines of code will typically result in a longer execution time, there is significant variation in execution time between benchmarks of similar length. This variation can be attributed to two primary factors; the time required to convert the boolean expression to CNF, and the time required to perform boolean satisfiability. The largest benchmark, E_Nsichneu with 4256 lines of code, was solved in 163.25 seconds while utilizing 72.18 MB. This shows the proposed method is scalable for larger benchmarks.

IV. Conclusion

In this paper, a novel method for identifying unreachable code to increase line coverage during software verification

Benchmark	LOC	Execution Time (sec)	Memory (MB)
Expint	157	12.48	59.95
E_Expint	168	13.67	63.98
Janne Complex	64	1.27	57.46
E_Janne Complex	72	4.69	54.55
Nsichneu	4253	154.95	73.78
E_Nsichneu	4257	163.25	72.18
Sqrt	77	0.25	55.00
E_Sqrt	89	6.94	60.90
Ud	163	4.37	58.19
E_Ud	180	5.56	57.89

Table IV: Performance Metrics

was presented. The technique generates a series of boolean relationships with other variables to form a set of constraints for a particular variable. The satisfiability of branch conditions is evaluated using sets of boolean constraints. By not modeling the exact functionality of arithmetic operations the algorithm is shown to be scalable and able to cope with exponential number of paths. It has been experimentally verified that the presented approach identified non-executable code in the benchmarks and as a result line coverage increased.

REFERENCES

[1] Hansang Bae and Rudolf Eigenmann. Interprocedural symbolic range propagation for optimizing compilers. In Eduard Ayguad, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 413–424. Springer Berlin Heidelberg, 2006.

[2] A. Biere. Picosat. http://fmv.jku.at/picosat/, 2013.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[4] Hong-Zu Chou, Kai-Hui Chang, and Sy-Yen Kuo. Optimizing blocks in an soc using symbolic code-statement reachability analysis. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 787–792, 2010.

[5] Hong-Zu Chou, Kai-Hui Chang, and Sy-Yen Kuo. Facilitating unreachable code diagnosis and debugging. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 485–490, 2011.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] C. Drake. pyeda. https://pypi.python.org/pypi/pyeda, 2014.

[8] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, SRI International, 2006.

[9] GNU. gcov. https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html, 2014.

[10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[11] C. Healy and D. Whaley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*, pages 79–88, 1999.

[12] Mikoláš Janota, Radu Grigore, and Michal Moskal. Reachability analysis for annotated code. In *Proceedings of the 2007 Conference on Specification and Verification of Component-based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pages 23–30, New York, NY, USA, 2007. ACM.

[13] D. Jayaraman and S. Tragoudas. Performance validation through implicit removal of infeasible paths of the behavioral description. In *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pages 552–557, March 2013.

[14] K. Karhu, T. Repo, O. Taipale, and K. Smolander. Empirical observations on software testing automation. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 201–209, April 2009.

[15] Daniel Kroening, Jol Ouaknine, SanjitA. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of presburger arithmetic. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 308–320. Springer Berlin Heidelberg, 2004.

[16] Luke Pierce. Unreachable mälardalen benchmarks. http://github.com/lukepierc/Unreach_Malardalen, 2014.

[17] Hossein M. Sheini and Karem A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *In Proc. SAT05, volume 3569 of LNCS*, pages 241–256. Springer, 2005.

[18] Chunrong Song and S. Tragoudas. Identification of critical executable paths at the architectural level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(12):2291–2302, Dec 2008.

[19] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.