Software Debugging and Testing using the Abstract Diagnosis Theory

Samaneh Navabpour Borzoo Bonakdarpour Sebastian Fischmeister

Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada N2L 3G1
{snavabpo, borzoo, sfischme}@ece.uwaterloo.ca

Abstract

In this paper, we present a notion of *observability* and *controllability* in the context of software testing and debugging. Our view of observability is based on the ability of developers, testers, and debuggers to trace back a data dependency chain and observe the value of a variable by starting from a set of variables that are naturally observable (e.g., input/output variables). Likewise, our view of controllability enables one to modify and control the value of a variable through a data dependency chain by starting from a set of variables that can be modified (e.g., input variables). Consequently, the problem that we study in this paper is to identify the minimum number of variables that have to be made observable/controllable in order for a tester or debugger to observe/control the value of another set of variables of interest, given the source code. We show that our problem is an instance of the well-known *abstract diagnosis problem*, where the objective is to find the minimum number of faulty components in a digital circuit, given the system description and value of input/output variables. We show that our problem is NP-complete even if the length of data dependencies is at most 2. In order to cope with the inevitable exponential complexity, we propose a mapping from the general problem, where the length of data dependency chains is unknown a priori, to *integer linear programming*. Our method is fully implemented in a tool chain for MISRA-C compliant source codes. Our experiments with several real-world applications show that in average, a significant number of debugging points can be reduced using our methods. This result is our motivation to apply our approach in debugging and instrumentation of embedded software, where changes must be minimal as they can perturb the timing constraints and resource consumption. Another interesting application of our results is in data logging of non-terminating embedded systems, where axillary data storage devices are slow and have limited size.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging —Debugging aids, Dumps, Tracing

General Terms Algorithms, Performance, Theory

Keywords Software debugging, Testing, Diagnosis, Logging.

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.

1. Introduction

Software testing and debugging involves *observing* and *controlling* the software's logical behaviour and resource consumption. Logical behaviour is normally defined in terms of the value of variables and the control flow of the program. Resource consumption is defined by both the resources used at each point in the execution and the amount and type of resources used by each code block. Thus, *observability* and *controllability* of the state of variables of a software system are the two main requirements to make the software testable and debuggable.

Although there are different views towards observability [3, 8–10, 18–20, 25], in general, observability is the ability to test various features of a software and observe its outcome to check if it conforms to the software's specification. Different views stand for controllability as well [3, 8–10, 18–20, 25]. Roughly speaking, controllability is the ability to reproduce a certain execution behaviour of the software. The traditional methods for achieving observability and controllability incorporate techniques which tamper with the natural execution of the program. Examples include using break points, interactive debugging, and adding additional output statements. These methods are often unsuited (specially for embedded software), because they cause changes in the timing behaviour and resource consumption of the system. Hence, the observed outcome of the software is produced by a mutated program which can violate its correctness. The mutated program can cause problems for controllability as well. For instance, a previously seen execution behaviour may be hard or impossible to reproduce. Thus, in the context of software testing and debugging, it is highly desirable to achieve software observability and controllability with the least changes in the software's behaviour. In particular, in embedded software, this property is indeed crucial.

This goal becomes even more challenging as the systems grow in complexity, since the set of features to cover during the testing and debugging phase increases as well. This can result in requiring more points of observation and control (i.e., *instrumentation*) which increases the number of changes made to the program. For instance, testers will require more *"printf"* statements to extract data regarding a specific feature which causes more side effects in the timing behaviour.

With this motivation, we treat the aforementioned problem by first formalizing the notions of observability and controllability as follows. Our view of observability is based on the ability of developers, testers, and debuggers to trace back a sequence of data dependencies and observe the value of a variable by starting from a set of variables that are naturally observable (e.g., input/output variables) or made observable. Controllability is the other side of the coin. Our view of controllability enables one to modify and control the value of a variable through a sequence of data dependencies by starting from a set of variables that can be modified (e.g., input variables). Thus, the problem that we study in this paper is as follows:

> *Given the source code of a program, our objective is to identify the minimum number of variables that have to be made observable/controllable in order for a tester or debugger to observe/control the value of another set of variables of interest.*

We show that our problem is an instance of the well-known *abstract diagnosis problem* [17]. Roughly speaking, in this problem, given are the description of a digital circuit, the value of input/output lines, a set of components, and a predicate describing what components can potentially work abnormally. Now, if the given input/output relation does not conform with the system description, the goal of the problem is to find the minimum number of faulty components that cause the inconsistency. The general diagnosis problem is undecidable, as it is as hard as solving first-order formulae. We formulate our observability/controllability problem as a sub-problem of the diagnosis problem. In our formulation, our objective is not to find which components violate the conformance of input/output relation with the system description, but to find the minimum number of components that can be used to observe/control the execution of software for testing and debugging.

Following our instantiation of the abstract diagnosis problem, first, we show that our optimization problem is NP-complete even if we assume that the length of data dependency chains is at most 2. In order to cope with the inevitable exponential complexity, we propose a mapping from the general problem, where the length of data dependency chains in unknown a prior, to *integer linear programming* (ILP). Although ILP is itself an NP-complete problem, there exist numerous methods and tools that can solve integer programs with thousands of variables and constraints.

Our approach is fully implemented in a tool chain comprising the following three phases:

1. *Extracting data:* We first extract the data dependency chains of the variables of interest for diagnosing from the source code.
2. *Transformation to ILP:* Then, we transform the extracted data dependencies and the respective optimization problem into an integer linear program. This phase also involves translation to the input language of our ILP solver.
3. *Solving the optimization problem:* We solve the corresponding ILP problem of finding the minimum set of variables required to diagnose our variables of interest.

Using our tool chain, we report the result of experiments with several real-world applications. These applications range over graph theoretic problems, encryption algorithms, arithmetic calculations, and graphical format encoding. Our experiments target two settings: (1) diagnosing *all* variables involved in a slice of the source code, and (2) diagnosing a *hand-selected* set of variables typically used by a developer for debugging. Our experiments show that while for the former our method reduces the number of variables to be made directly diagnosable significantly, for the latter the percentage of reduction directly depends upon the structure of the source code and the choice of variables of interest. Since solving complex integer programs is time-consuming, our experimental observations motivate the idea of developing a simple metric in order to intelligently predict whether applying the optimization is likely to be worthwhile. To this end, we propose one such simple metric in this paper and discuss its correlation with our optimization problem. Another beneficial consequence of our optimization is in reducing the execution time of instrumented code. Indeed, we show that the overall execution time of the code instrumented optimally is significantly better than the corresponding time with the original instrumentation.

Organization. The rest of the paper is organized as follows. After discussing related work in Section 2, in Section 3, we present our notions of observability and controllability and discuss their relevance to software testing and debugging. Section 4 is dedicated to formally present our problem and its complexity analysis. Then, in Section 5, we present a transformation from our problem to ILP. Our implementation method and tool chain are described in Section 6. We analyze the results of our experiments in Section 7. Finally, we make concluding remarks and discuss future work in Section 8.

2. Related Work

As mentioned in the introduction our formulation of the problem is an instance of the abstract diagnosis theory [17]. The diagnosis theory has been extensively studied in many contexts. In [7], Fijany and Vatan propose two methods for solving the diagnosis problem. In particular, they present transformations from a sub-problem of the diagnosis problem to the integer linear programming and satisfiability problems. Our transformation to integer programming in this paper is more general, as we consider data dependency chains of arbitrary length. Moreover, in [7], the authors do not present experimental results and analysis. On the other hand, our method is fully implemented in a tool chain and we present a rigorous analysis of applying the theory on real-world applications.

On the same line of research, Abreu and van Gemund [1] propose an approximation algorithm to solve the minimum *hitting set problem*. This problem is in spirit very close to an instance of the diagnosis problem. The main difference between our work and [1] is that we consider data dependencies of arbitrary length. Thus, our problem is equivalent to a general *nested* hitting set problem. Moreover, we transform our problem to ILP whereas in [1], the authors directly solve the hitting set problem.

Ball and Larus [2] propose algorithms for monitoring code to *profile* and *trace* programs: profiling counts the number of times each basic block in a program executes. Instruction tracing records the sequence of basic blocks traversed in a program execution. Their algorithms take the control-flow graph of a given program as input and finds optimal instrumentation points for tracing and profiling. On the contrary, in our work, we directly deal with source code and actual variables. Ball and Larus also mention that vertex profiling (which is closer to our work) is a hard problem and their focus is on edge profiling.

Fujiwara [10] defines observability and controllability for hardware. He considers observability as the propagation of the value of all the signal lines to output lines. Respectively, controllability is enforcing a specific value onto a signal line. He notes that if observability and controllability are unsatisfied, additional outputs and inputs must be added to the circuit. This addition is tightly coupled with the position of the line and type of circuit. Moreover, this work does not address how to choose the best point to add pins. In addition, each time the pin counts change, one needs to re-evaluate the observability/controllability of the lines.

Freedman [8], Voas, and Miller [25] view observability and controllability from the perspective of black box testing. They consider a function to be observable, if all the internal state information affecting the output are accessible as input or output during debugging and testing. They present the following metric to evaluate observability: $DDR = \frac{\text{Cardinality of input}}{\text{Cardinality of output}}$, where DDR should be close to 1 to have observability. Their method affects the temporal behaviour of the software and may expose sensitive information.

In the context of distributed systems, the approach in [20, 21] defines a system behaviour to be observable, only if it can be uniquely defined by a set of parameters/conditions. The goal in this work is using deterministic instrumentation with minimal overhead to achieve observability. The shortcoming of this work is that the authors do not present a technique to find the instrumentation with minimal overhead. In addition, the instrumentation should remain in the deployed software to avoid probe effects. Thus, Thane considers a behaviour controllable with respect to a set of variables only when the set is controllable at all times. The author also proposes a real-time kernel to achieve offline controllability.

Schutz [18, 19] addresses observability and controllability for time triggered (TT) and event triggered (ET) systems. The author's method however, does not avoid instrumentation in the design phase and, hence, uses dedicated hardware to prevent probe effects. Schutz argues that TT systems offer alternative flexibility compared to ET systems when handling probe effects caused by enforced observability. Respectively, Schutz shows that unlike ET systems, TT systems have less probe effects in controllability since there is no missing information concerning their behaviour and, hence, an additional approach is needed to collect information.

The approach proposed in [6, 15, 16, 23, 24] is in spirit similar to our approach, but in a different setting. The author analyzes the data flow design of a system and define observability and controllability based on the amount of information lost from the input to the output of the system. Their calculations are based on bit-level information theory. This method estimates controllability of a flow based on the bits available at the inputs of the module from the inputs of the software via the flow. Respectively, they estimate observability as the bits available at the outputs of the software from the outputs of the module via the flow. We believe a bit-level information theoretic approach is unsuited for analysis of real-world large applications, because (1) the proposed technique ignores the type of operations constructing the data flow while it has an effect on observing and controlling data, (2) lost bits or corrupted propagated bits throughout a flow may lead us to inconsistent observations, and (3) although the amount of information propagated throughout a flow is of major importance, the bit count is an improper factor of measurement.

3. Observability and Controllability as a Diagnosis Problem

The *diagnosis problem* [17] was first introduced in the context of automatic identification of faulty components of logic circuits in a highly abstract fashion. Intuitively, the diagnosis problem is as follows: Given a system description and a reading of its input/output lines, which may conflict with the description, determine the minimum number of (faulty) components of the system that cause the conflict between the description and the reading of input/output. In other words, the input/output relation does not conform with the system description due to existence of faulty components. Formally, let SD be a system description and IO be the input/output reading of the system, both in terms of a finite set of first-order formulae. Let C be a set of components represented by a set of constants. Finally, let $\neg AB(c)$ denote the fact that component $c \in C$ is behaving correctly. A diagnosis for (SD, C, IO) is a minimal set $D \subseteq C$ such that:

$$SD \ \wedge \ IO \ \wedge \ \bigwedge_{c \in D} AB(c) \ \wedge \ \bigwedge_{c \in C-D} \neg AB(c)$$

is satisfiable. Obviously, this satisfiability question is undecidable, as it is as hard as determining satisfiability of a first-order formula.

Our instance of the diagnosis problem in this paper is inspired by Fujiwara's [10] definition to address *observability* and *controllability*[1]. Our view is specifically suitable for sequential embedded software where an external entity intends to diagnose the value of a variable directly or indirectly using the value of a set of other variables according to the system description SD. We instantiate the abstract diagnosis problem as follows. We interpret the set C of components as a set of variables. In our context, SD is a set of instructions in some programming language. In order to diagnose the value of a variable v using the value of another variable v', v must depend on v'. In other words, there must exist a function F that connects the value of v with the value of v'; i.e., $v = F(v')$. Thus, we consider the following types of data dependency:

1. *Direct Dependency*: We say that the value of v directly depends on the value of v' iff $v = F(v', V)$, where F is an arbitrary function and V is the remaining set of F's arguments.
2. *Indirect Dependency*: We say that v indirectly depends on v'' iff there exists a variable v', such that (1) v directly depends on v', and (2) v' directly depends on v''.

Data dependencies can be easily extracted from SD. For example, consider the following program as an instance of SD:

```
(l1) g := x + z + y;
(l2) if g > 100 then
(l3)     c := d / e;
(l4) else
(l5)     c := f * g;
(l6) m := x % y;
(l7) b := d - f;
(l8) a := b + c;
```

It is straightforward to see that the value of variable `a` directly depends on the value of `b` and `c` and indirectly on `d` and `g`. Observe that the notion of dependency is not necessarily interpreted by left to right assignments in a program. For example, in the above code, the value of variable `d` directly depends on the value of `b` and `f`. On the contrary, one cannot extract the value of `x` from `m` and `y`, as the inverse of the modulo operator is not a function.

In our framework, we interpret $\neg AB(c)$ as variable c is *immediately diagnosable* (i.e., c can be directly observed or controlled). For instance, in programming languages, the value of constants, literals, and input arguments are known a priori and, hence, are immediately diagnosable. Thus, $AB(c)$ means that variable c is not immediately diagnosable, but it may be diagnosed through other variables. To formalize this concept, we define what it means for a variable to be *diagnosable* based on the notion of data dependencies. Roughly speaking, a variable is diagnosable if its value can be traced back to a set of immediately diagnosable variables.

DEFINITION 1 (Diagnosable Variable). *Let V be a set of immediately diagnosable variables. We say that a variable v_1, such that $AB(v_1)$, is* diagnosable *iff there exists an acyclic sequence σ of data dependencies: $\sigma = \langle v_1, d_1, v_2\rangle, \langle v_2, d_2, v_3\rangle, \ldots, \langle v_{n-1}, d_{n-1}, v_n\rangle, \langle v_n, d_n, v_{n+1}\rangle$ such that $v_{n+1} \in V$.*

In the remainder of this work, we refer to any dependency sequence enabling variable diagnosis as a *diagnosis chain*. Finally, in our instance of the diagnosis problem, we assume that the predicate IO always holds.

As mentioned earlier, our instance of program diagnosis is customized to address variable *observability* and *controllability*. Intuitively, a variable is controllable if its value can be defined or

[1] Throughout the paper, when we refer to 'diagnosis', we mean 'observability/controllability'.

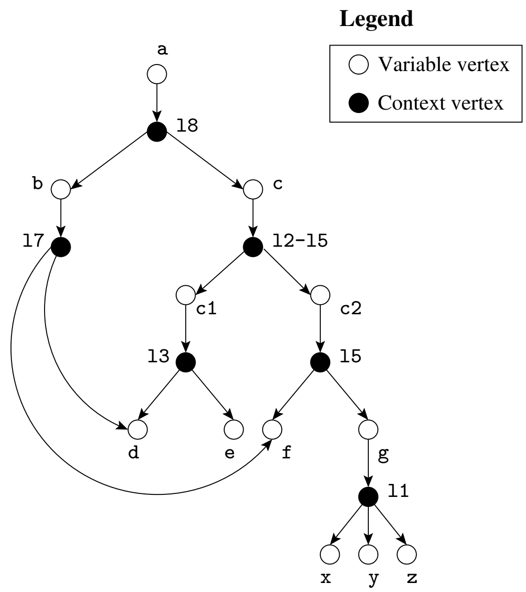

Figure 1. A simple diagnosis graph.

modified (i.e., controlled) by an external entity through a diagnosis chain. For instance, if the external entity modifies the value of an immediately diagnosable variable v, then it can potentially modify the value of variables that can be traced back to v. Such an external entity can be the developer, tester, debugger, or the environment in which the program is running. Likewise, a variable is observable if its value can be read through a diagnosis chain.

DEFINITION 2 (Controlability). *A variable v is* controllable *iff v is diagnosable and an external entity can modify the value of v by using an immediately controllable variable v' and a diagnosis chain that ends with v'.*

DEFINITION 3 (Observability). *A variable v is* observable *iff v is diagnosable and an external entity can read the value of v by using an immediately observable variable v' and a diagnosis chain that ends with v'.*

In order to analyze observability and controllability in a more systematic fashion, we introduce the notion of diagnosis graphs. A *diagnosis graph* is a data structure that encodes data dependencies for a given variable. For instance, with respect to variable `a` in the above program, all statements but `l6` are of interest. In other words, the *program slice* [4] of the above program with respect to variable `a` is the set $L_{\mathtt{a}} = \{$`l1`, `l2`, `l3`, `l4`, `l5`, `l7`, `l8`$\}$. A special case of constructing slices is for conditional statements (e.g., if-then-else and case analysis). For example, in the above program, since we cannot anticipate which branch will be executed at runtime, we have to consider both branches to compute dependencies. Thus, for variable `c`, we take both statements `l3` and `l5` into account; i.e., $L_{\mathtt{c}} = \{$`l2`, `l3`, `l4`, `l5`$\}$.

Formally, let v be a variable, L_v be a program slice with respect to v, and *Vars* be the set of all variables involved in L_v. We construct the diagnosis directed graph G as follows.

- *(Vertices)* $V = C \cup U$, where $C = \{c_l \mid l \in L_v\}$ and $U = \{u_v \mid v \in \mathit{Vars}\}$. We call the set C *context vertices* (i.e., one vertex for each instruction in L_v) and the set U *variable vertices* (one vertex for each variable involved in L_v).

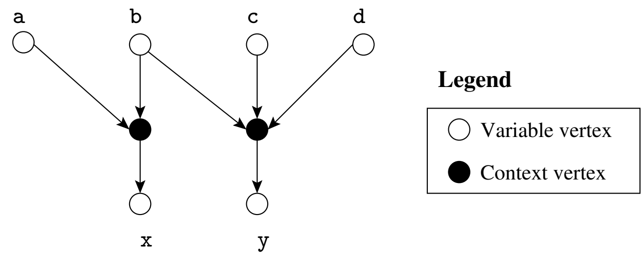

Figure 2. A diagnosis graph.

- *(Arcs)* $A = \{(u, c) \mid u \in U \ \wedge \ c \in C \wedge$ variable v is affected within context $c\} \ \cup \ \{(c, u) \mid u \in U \ \wedge \ c \in C \wedge$ variable v affects context $c\}$.

OBSERVATION 1. *Diagnosis graphs are acyclic and bipartite.*

The diagnosis graph of the above program with respect to variable `a` is shown in Figure 1. Although our construction of diagnosis graph is with respect to one variable, it is trivial to merge several diagnosis graphs.

4. Statement of Problem and Its Complexity Analysis

The notion of diagnosis graph presented in Section 3 can be used to determine several properties of programs in the context of testing and debugging. One such property is whether a debugger has enough information to efficiently observe the value of a set of variables of interest. Another application is in creating data logs. For instance, since it is inefficient to store the value of *all* variables of a non-terminating program (e.g., in embedded applications), it is desirable to store the value of a subset of variables (ideally minimum) and somehow determine the value of other variables using the logged variables. Such a property is highly valuable in embedded systems as they do not have the luxury of large external mas storage devices. Thus, the problem boils down to the following question:

> *Given a diagnosis graph and a set of immediately diagnosable variables, determine whether a variable is diagnosable.*

Another side of the coin is to identify a set of variables that can make another set of variables diagnosable. To better understand the problem, consider the diagnosis graph in Figure 1. One can observe the value of `a` by making, for instance, variables $\{$`c`, `d`, `f`$\}$ immediately observable.

A valid question in this context is that if a set of variables is to be diagnosed, why not make all of them immediately diagnosable. While this is certainly a solution, it may not be efficient or feasible. Consider again the graph in Figure 1. Let us imagine that we are restricted by the fact that variables $\{$`a`, `b`, `c`, `g`$\}$ cannot be made immediately observable (e.g., for security reasons). Now, in order to diagnose variable `a`, two candidates are $\{$`d`, `e`, `f`$\}$ and $\{$`d`, `f`, `x`, `y`, `z`$\}$. Obviously, for the purpose of, for instance, logging, the reasonable choice is the set $\{$`d`, `e`, `f`$\}$.

Figure 2 shows another example where choice of variables for diagnosis matters. For instance, in order to make `a`, `b`, and `c` diagnosable, it suffices to make `x` and `y` immediately diagnosable. Thus, it is desirable to identify the minimum number of immediately diagnosable variables to make a set of variables of interest diagnosable.

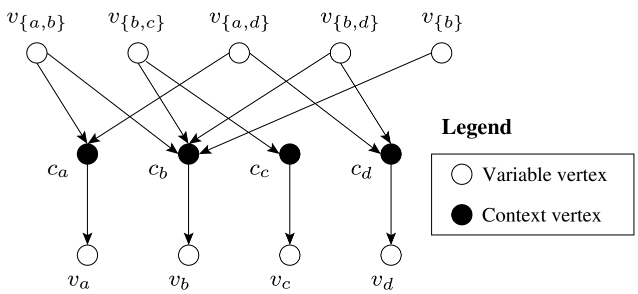

Figure 3. An example of mapping HSP to DGP.

It is well-known in the literature that the diagnosis problem is a typical covering problem [1, 7, 17]. In fact, many instances of the diagnosis problem is in spirit close to the *hitting set problem* [12] (a generalization of the vertex cover problem). We now show that this is indeed the case with respect to our formulation. We also show that a simplified version of our problem is also NP-complete.

Instance (DGP). A diagnosis graph G of diameter 2, where C and U are the set of context and variable vertices respectively and A is the set of arcs of G, a set $V \subseteq U$, and a positive integer $K \leq |U|$.

Question. Does there exist a set $V' \subseteq U$, such that $|V'| \leq K$ and all variables in V are diagnosable if all variables in V' are immediately diagnosable.

We now show that the above decision problem is NP-complete using a reduction from the *hitting set problem* (HSP) [12]. This problem is as follows: Given a collection $\mathcal{S}$ of subsets of a finite set S and positive integer $N \leq |S|$, is there a subset $S' \subseteq S$ with $|S'| \leq K$ such that S' contains at least one element from each subset in $\mathcal{S}$?

THEOREM 1. *DGP is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only show that the problem is NP-hard. We map the hitting set problem to our problem as follows. Let $\mathcal{S}$, S, and N be an instance of HSP. We map this instance to an instance of DGP as follows:

- *(Variable vertices)* For each set $s \in \mathcal{S}$, we add a variable vertex v_s to U. In addition, for each element $x \in S$, we add a variable vertex v_x to U.
- *(Context vertices)* For each element $x \in S$, we add a context vertex c_x to C.
- *(Arcs)* For each element $x \in S$ that is in set $s \in \mathcal{S}$, we add arc (v_s, c_x). And, for each $x \in S$, we add arc (c_x, v_x).
- We let $V = \{v_s \mid s \in \mathcal{S}\}$ (i.e., the set of variables to be diagnosed) and $K = N$.

Observe that the diameter of the graph constructed by our mapping is 2. To illustrate our mapping, consider the example, where $S = \{a, b, c, d\}$ and $\mathcal{S} = \{\{a, b\}, \{b, c\}, \{a, d\}, \{b, d\}, \{b\}\}$. The result of our mapping is shown in Figure 3.

We now show that the answer to HSP is affirmative if and only if the answer to DGP is also affirmative:

- ($\Rightarrow$) Let the answer to HSP be the set $S' \subseteq S$. Our claim is that in the instance of DGP, the set $V' = \{v_x \mid x \in S'\}$ is the answer to DGP (i.e., the set of immediately diagnosable variables). To this end, first, notice that $|V'| = |S'|$ and, hence, $|V'| \leq K$, as $K = N$ in our mapping and $|S'| \leq N$. Now, since every set $s \in \mathcal{S}$ is covered by at least one element in S', it implies that V' makes all variables in V diagnosable.
- ($\Leftarrow$) Let the answer to DGP be the set $V' \subseteq V$. Our claim is that in the instance of HSP, the set $S' = \{x \mid v_x \in V'\}$ is the answer to HSP (i.e., the hitting set). To this end, first, notice that $|S'| = |V'|$ and, hence, $|S'| \leq N$, as $N = K$ in our mapping and $|V'| \leq K$. Now, since every variable $v_s \in V$ is made diagnosable by at least one variable vertex in V', it implies elements of S' hit all sets in $\mathcal{S}$. $\square$

Note that our result in Theorem 1 is for diagnosis graphs of diameter 2. If the diameter of a diagnosis graph is unknown (i.e., the depth of diagnosis chains), we conjecture that another exponential blow-up is involved in solving the problem. In fact, our problem will be similar to a *nested* version of the hitting set problem and it appears that the problem is not in NP.

5. Mapping to Integer Linear Programming

In order to cope with the worst-case exponential complexity of our instance of the diagnosis problem, in this section, we map our problem to *Integer Linear Programming* (ILP). ILP is a well-studied optimization problem and there exist numerous efficient techniques for it. The problem is of the form:

$$\begin{cases} \text{Minimize} & c.\mathbf{x} \\ \text{Subject to} & A.\mathbf{x} \geq \mathbf{b} \end{cases}$$

where A (a rational $m \times n$ matrix), c (a rational n-vector), and $\mathbf{b}$ (a rational m-vector) are given, and, $\mathbf{x}$ is an n-vector of integers to be determined. In other words, we try to find the minimum of a linear function over a feasible set defined by a finite number of linear constraints. It can be shown that a problem with linear equalities or $<=$ linear inequalities can always be put in the above form, implying that this formulation is more general than it might look. ILP is known to be NP-complete, but as mentioned earlier there exist many efficient techniques and tools that solve ILP.

We now describe how we map our instance of the diagnosis problem described in Sections 3 and 4 to ILP. Although in our implementation, we start from the source code, here for the sake of clarity, we start our mapping from a given diagnosis graph. Let G be a diagnosis graph with the set C of context vertices, the set U of variable vertices, and the set A of arcs. Our goal is to make the set V of vertices diagnosable and we are looking for the minimum number of variable vertices that must become immediately diagnosable. In order to make variables of V diagnosable, one has to consider all diagnosis chains that start from the variables in V. Now, let v be a variable vertex in V whose outgoing arcs reach context vertices $\{c_1 \cdots c_n\}$. Obviously, v can be diagnosed through any of the context vertices, say c_i, where $1 \leq i \leq n$. Thus, in our mapping, we represent each context vertex c with a binary integer variable x_c; i.e., the value of x_c is 1 if and only if v can be diagnosed through c. Otherwise, x_c is equal to 0. Thus, for each variable $v \in V$ whose reachable context vertices are $\{c_1 \cdots c_n\}$, we add the following constraint to our integer program:

$$\sum_{i=1}^{n} x_{c_i} \geq 1 \qquad x_{c_i} \in \{0, 1\} \tag{1}$$

Intuitively, this constraint means that variable v is diagnosed by at least one of the context vertices that it can reach in the diagnosis

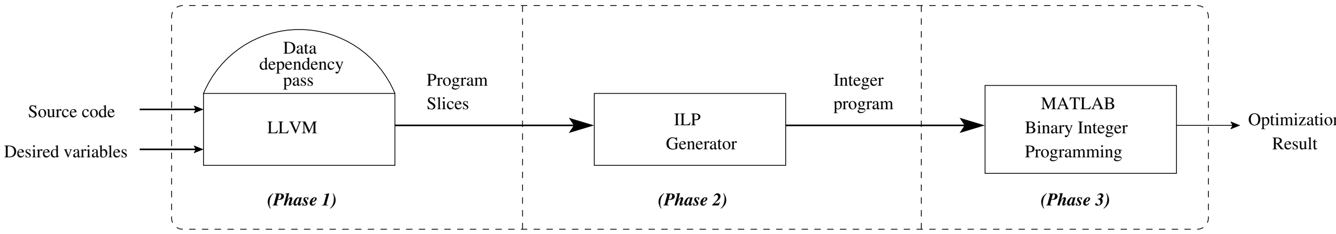

Figure 4. Data flow in the tool chain.

graph. For example, for Figure 3, we have the following constraints according to inequality 1:

$$\begin{array}{llllll} x_{c_a} + x_{c_b} & \geq 1, & x_{c_b} + x_{c_c} & \geq 1, & & \\ x_{c_a} + x_{c_d} & \geq 1, & x_{c_b} + x_{c_d} & \geq 1, & x_{c_b} & \geq 1 \end{array}$$

Now, if a variable vertex v is diagnosed through a context vertex c, then all variable vertices $\{v_1 \cdots v_m\}$ that are reachable from c by one step must be diagnosable as well (following our notion of diagnosis chains). In order to encode this in our integer program, for each context vertex c, in addition to integer variable x_c, we add another integer variable x'_c, whose value is the negation of x_c. Thus, for each context vertex, we add the following constraint to our integer program:

$$x_c + x'_c = 1 \tag{2}$$

We now encode the fact that if a context vertex c is chosen in the solution to the integer program (i.e., $x_c = 1$), then all its variable vertices must be diagnosable. To this end, if context vertex c is associated with variable vertices $\{v_1 \cdots v_m\}$, we add two integer variable x_{v_i} and x'_{v_i}, for all $1 \leq i \leq m$, to our integer program. Moreover, we add the following constraints:

$$x'_c + x_{v_i} \geq 1 \tag{3}$$

$$x'_c - x'_{v_i} \geq -1 \tag{4}$$

$$x_{v_i} + x'_{v_i} = 1 \tag{5}$$

$$x_{v_i} \in \{0, 1\}$$

Intuitively, Constraint 3 means that if a context vertex is not chosen by the solution, then its associated variable vertices may or may not appear in the solution. And, constraint 4 ensures that Constraint 3 is consistent with that fact that x'_{v_i} is the negation of x_{v_i}. For example, we have the following constraints for the graph in Figure 3 for context vertex c_a:

$$x'_{c_a} + x_{v_a} \; \geq \; 1, \quad x'_{c_a} - x'_{v_a} \; \geq \; -1$$

We keep repeating this procedure for each new reachable level of context vertices of the diagnosis graph and generate constraints 3-5 until we reach the vertices of the graph whose outdegree is 0. Such vertices do exist in a diagnosis graph according to Observation 1. Once we reach a vertex whose outdegree is 0, we generate no constraints. This is because such vertices must be considered as both a context vertex and a variable vertex (i.e., the variable can be made diagnosable only if it is made immediately diagnosable). Thus, the value of their corresponding integer variable is either 0 or 1 independent of other variables.

Finally, we specify our objective function. Let $V'' = \{v_1 \cdots v_k\}$ be the set of variable vertices that we allow to become immediately diagnosable. Obviously V'' can be equal to U, the set of all variable vertices. As mentioned in Section 4, our goal is to find the minimum number of variables that must become immediately diagnosable. Hence, our integer program optimization is the following:

$$\text{Minimize} \quad \sum_{i=1}^{k} x_{v_i} \tag{6}$$

For example, in our example in Section 3, the optimization criterion is to minimize $x_d + x_e + x_f + x_x + x_y + x_z$.

6. Implementation and Tool Chain

In this section, we present a detailed representation of our tool chain. Our tool chain consists of three main phases as depicted in Figure 4:

1. *Extracting data:* We first extract the data dependency chains of the variables of interest for diagnosis from the source code.
2. *Transformation to ILP:* Next, we transform the extracted data dependencies and the respective optimization problem into an integer linear program as described in Section 5. This phase also involves translation to the input language of our ILP solver.
3. *Solving the optimization problem:* We solve the corresponding ILP for finding the minimum set of variables required to become immediately diagnosable.

In the following, we describe each phase in detail.

6.1 Extracting data

As mentioned earlier, one input to our problem is the set of data dependency chains with respect to our variables of interest. Thus, we need to extract all the statements in the source code that affect the value of the variables of interest. This extraction is known as *program slicing* [4, 22]. We apply the technique in [5] to extract program slices from the single-static assignment (SSA) of the source code, as it requires less computation resources.

In terms of the corresponding tool, we leverage LLVM [13] to extract program slices. LLVM is a collection of modular and reusable compiler and tool chain technologies. We choose LLVM among other options such as GCC, because of its ability to convert programs into SSA. In addition, LLVM is well-established and well-documented with a rich API, which simplifies the process of implementing new analysis and transformation passes.

Consequently, we implement a new *data dependency pass* over the LLVM framework (see Phase 1 in Figure 4). First, the *data dependency pass* accepts the source code and the set of desired variables for diagnosis as input. We note that the source code is customized to be MISRA-C [14] compliant while only having analyzable use of pointers. Secondly, it converts all the memory variables into registers, since LLVM only considers register variables in its SSA conversion. Then, the *data dependency pass* incorporates the dominator tree passes and the *def-use* and *use-def* API's of LLVM to extract the data dependency chains. Finally, *data dependency pass* slices the program into its SSA form, so that only the SSA statements affecting the values of desired variables are extracted into an output file.

6.2 Transformation into Integer Linear Programming

Once we have the program slices as LLVM output, we perform another phase of processing to transform the slices into an integer program. To this end, we develop a transformer (see Phase 2 in Figure 4), called *ILP Generator*. This generator takes program slices as input and converts them to the input format of the ILP solver. The transformation follows our method described in Section 5. However, in our implementation, we do not start from a diagnosis graph. In fact, our transformation starts directly from program slices.

First, ILP Parser takes program slices and converts each SSA statement to a unique variable set. It considers the list of all sets as the vector of variables in the ILP model. Then, it analyzes the dependency of each set with the other sets extracted from program slices. In other words, for the variables of a set to be diagnosed, the parser finds all the sets whose variables must become diagnosable as well. This is similar to the role of context vertices in diagnosis graphs in Section 5. The completion of this phase results in identifying the set of all constraints and variables of the ILP model. Then, the ILP generator computes the objective function based on the variables used by the statements in program slices of the LLVM output (see also objective function 6 in Section 5). Finally, ILP generator puts the ILP model into an output file and feeds it into the ILP solver.

6.3 Solving the Optimization Problem

This is the last phase of our tool chain (see Phase 3 in Figure 4). Our choice of ILP solver is the binary integer programming toolbox of MATLAB. As described in Section 5, all variables in our integer program are binary and MATLAB's library suffices for our purpose. In our tool chain, the ILP input is in MATLAB's format. MATLAB's solution to the given ILP problem is the set of variables that need to be immediately diagnosable.

7. Experimental Results

In this section, we describe the results of our experiment on using our tool chain in order to evaluate our method based on the factor of *instrumentation point reduction*. By instrumentation point, we mean instructions that are added to the source code in order to observe or control the value of a set of desired variables. Examples of such instrumentation include logging instructions such as *printf* and *fprintf*.

In Subsection 7.1, we analyze the effectiveness of our method on a set of real-world applications by studying the reduction in instrumentation points. We present evidence in Subsection 7.2 that in order to handle our case studies, we have not reached the boundaries of capabilities of ILP solvers. Then, in Subsection 7.3, we look into the effect of optimizing instrumentation points on execution time of instrumented code. Finally, in Subsection 7.4, we present a metric for diagnosis graphs that can assist in predicting the effectiveness of our optimization method on a source code for a given set of desired variables to be diagnosed.

7.1 Effectiveness of Our Optimization Method

To conduct our experiments, we used MiBench [11] as the set of programs to run our method. MiBench is a free, commercially representative embedded benchmark suite. We choose MiBench because of its high diversity in programs and benchmarks. The programs differ in size, property, and behaviour and provide us with a wide range of source codes that we can use to study the effectiveness of our method. MiBench has six main categories

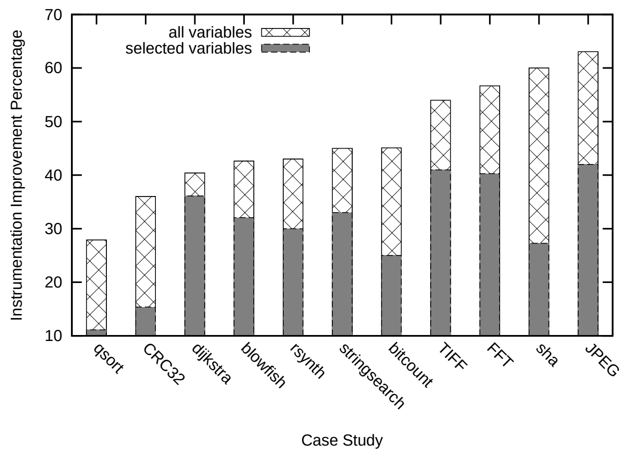

Figure 5. Results of experiments on effectiveness of optimization of instrumentation points.

of programs: automative, consumer, office, network, security, and telecomm. We randomly choose the following programs for our experiments:

- bitcount and qsort from automative,
- JPEG and TIFF from consumer,
- dijkstra from network,
- rsynth and stringsearch from office,
- blowfish and sha from security, and
- CRC32 and FFT from telecomm.

Figure 5 shows the percentage of reduction in the number of instrumentation points for each case study under two scenarios. Before we elaborate on the results, we recall that our goal in this paper is to optimize a given instrumentation scheme. In other words, our technique does not start with a null set of instrumentation points. It rather optimizes a user-specified mandatory instrumentation requirement. Thus, if the given instrumentation scheme is not efficient and already imposes a significant overhead, our method can reduce this overhead based on the program structure. In this context, absolute values of results in this section (e.g., execution time of the uninstrumented programs) are not relevant to the idea of the paper. What is indeed relevant is the relative optimization.

7.1.1 Scenario A: Optimization with Respect to a Subset of Variables

In the first scenario (marked by 'selected variables' in Figure 5), in each MiBench program, we use our intuition to set the instrumentation points (i.e., desired variables for diagnosis). Similar to any debugger, we pin down all the variables we would typically suspect as faulty when the output of the program is erroneous. Then, at each instruction point defining each of these suspicious variables, we insert an instrumentation instruction which logs the value of the variable. To clarify our procedure, consider the following code from the *qsort_large* program in MiBench:

```
struct my3DVertexStruct {
  int x, y, z;
  double distance;
};

int compare(const void *elem1, const void *elem2)
{
  double distance1, distance2;
```

```
9
10    distance1 =
11          (*((struct my3DVertexStruct *)elem1)).distance;
12    distance2 =
13          (*((struct my3DVertexStruct *)elem2)).distance;
14
15    return (distance1 > distance2) ? 1
16                : ((distance1 == distance2) ? 0 : -1);
17 }
18
19
20 int
21 main(int argc, char *argv[]) {
22    struct my3DVertexStruct array[MAXARRAY];
23    FILE *fp;
24    int i,count=0;
25    int x, y, z;
26
27    if (argc<2) {
28      fprintf(stderr,"Usage: qsort_large <file>\n");
29      exit(-1);
30    }
31    else {
32      fp = fopen(argv[1],"r");
33
34      while((fscanf(fp, "%d", &x) == 1) && (fscanf(fp, "%d"
35                , &y) == 1) && (fscanf(fp, "%d", &z) == 1)
36                              && (count < MAXARRAY)) {
37          array[count].x = x;
38          array[count].y = y;
39          array[count].z = z;
40          array[count].distance = sqrt(pow(x, 2) +
41                              pow(y, 2) + pow(z, 2));
42          count++;
43      }
44    }
45    qsort(array,count,sizeof(struct my3DVertexStruct)
46                                                ,compare);
47
48    for(i=0;i<count;i++)
49      printf("%d %d %d\n", array[i].x,
50                       array[i].y, array[i].z);
51    return 0;
52 }
```

Listing 1. qsort_large

In this code, the main variables taking part in the output are: *x*, *y*, *z*, *distance*, *distance1*, *distance2*, *fp*, *argc*, and *my3DVertexStruct*. Thus, we add instrumentation points, at lines 37, 38, 39, 40, 10, 12, 32, 27, and 45 to log each variable respectively. For instance, we do not instrument *count*, since it only controls access to the array elements and its value is not considered in the output of *qsort_large*. Likewise, we use the same procedure in instrumenting our other case studies. As can be seen in Figure 5, qsort shows an improvement of 11.11%. Our experiments show that the number of instrumentation points dropped from 9 to 8. The deleted instrumentation point was on line 40 for logging *distance*. This result is quite obvious, since *distance* has a data dependency with *x*, *y*, and *z*. Since *x*, *y*, and *z* are log they become immediately diagnosable, creating a resolvable data dependency with *distance*. This results in *distance* becoming observable as well, eliminating the need for it to be logged.

On the contrary, we see considerable improvement in the reduction of instrumentation points in JPEG, TIFF, and FFT. The case studies show reduction of 42%, 41%, and 40.27% in instrumentation points, respectively. For instance, in JPEG, our optimization reduces instrumentation points to an average of 10 in each C file and achieves a 42% reduction in total. Likewise, in FFT, the original 72 instrumentation points are dropped to 40, gaining 40.27% improvement.

The aforementioned results on percentage of reduction of instrumentation clearly shows the effectiveness of our method. For instance, if our method is applied to a reactive embedded system where there exists only a limited auxiliary storage for logging purposes, 40% reduction in the size of the data log can be significantly beneficial.

7.1.2 Scenario B: Optimization with Respect to all Variables

Our second scenario (marked by 'all variables' in Figure 5) is based on instrumenting all the variables in the program slice of the output variables. For instance, considering qsort, we logged *count* as well, since its value takes part in creating the output and, hence, it appears in the program slice. As can be seen, in this scenario, we achieve even more reduction. For instance, we achieve a 60% reduction in sha when logging all variables, while we achieved 27.27% reduction by choosing only a subset of variables based on our debugging intuition.

On average, our method achieves a 30.5% reduction in instrumentation points for the MiBench programs studied in this section. Hence, as argued earlier, if we can reduce instrumentation points of long-running programs close to 30%, it is worthwhile optimizing programs that create large data logs and require great amount of costly resources, such as the black box recorders in aircrafts. On the other hand, by reducing the instrumentation points in embedded systems by 30%, we reduce the debugger interference with the original source code to reduce debugging side-affects in the natural behaviour of the system.

As can be seen in Figure 5, the reduction in the instrumentation points fluctuates from one program to another. Thus, one needs to analyze the experimental results by relating each reduction to the program's structural and behavioural properties. To clarify our analysis process, consider our example of qsort. Based on our study, we conclude that the small number of dependencies among the variables in the source code is the reason behind small improvement in reduction of instrumentations. For instance, *x*, *y* and *z* are not data dependent and, hence, none can become diagnosable through the other. On the contrary, consider programs JPEG, TIFF, and FFT. All three show considerable instrumentation reduction. In general, the results from our studies show that there are three main factors causing the reduction:

1. Large number of common variables used and updated by different functions in different C files,
2. Large number of functions that their input arguments are developed by a set of other functions in different C files, and
3. The size of the input source code.

To generalize the results from our analysis, we draw the following set of factors as causes of fluctuations:

1. The data flow of the program which in turn defines the data dependencies among variables in the source code.
2. The control flow of the program.
3. The size (i.e., number of variables and instructions) and structure (i.e., functions and modules) of the program.
4. The set of variables of interest.

Although the proposed properties affect the effectiveness of our method, at this point, we do not claim that the presented set is closed. Extracting the complete set of factors and formalization of the relation between the reduction of instrumentation points and the above factors still need further research.

Based on the presented results, we are now able to explain the reason of the increase of instrumentation reduction when considering the logging of all the variables in the program slice. When we consider all the variables in a program slice, we are considering more variables in our analysis. Each variable introduces a new set of data dependencies which we can use to our benefit. These new data dependencies increase the possibility of diagnosing variables via other variables. In other words, the possibility that variables share common variables for their diagnosis increases. That is why we see an increase in the reduction. As for sha, the significant increase in the reduction shows that there is a large number of variables which take part in diagnosis of the output that are simply overseen by the debugger. This can be for various reasons, such as the layout of the source code. Hence, when all the variables are logged, all the variables affecting the output and their data dependencies are included in the analysis, which in turn introduces new data dependency chains, resulting in reducing the number of variables needed to become immediately diagnosable.

7.2 Complexity of the Integer Program

When it comes to dealing with problems with exponential complexity, a valid concern is the scalability issue. Our experiments show that in the context of our case studies, our method has not yet reached the boundaries of capabilities of current ILP solvers. For example, in case of JPEG, where the source code has 3105 lines of C code, the corresponding integer program has 220 binary variables and 65968 constraints. The approximate time to solve the integer program on a MacBook Pro with 2.26 GHz Intel Core 2 Duo and 2GM RAM is 5 minutes using MATLAB. Hence, scalability by itself is not an obstacle in optimizing instrumentation of real-world applications in the size of our case studies.

7.3 The Effect of Instrumentation Optimization in Execution Time

An important consequence of optimized instrumentation is better execution time of instrumented code. In order to validate this claim, we conduct three experiments to measure execution time of FFT, dijkstra, and JPEG in two settings:

1. Running the programs with instrumenting the hand-selected variables in the program slice,
2. Running the program with the reduced set of instrumentation points identified using our method.

Our instrumentation involves a fast I/O interaction: writing the value of variables on the hard-disk. The results from this experiment are presented in Figure 6. For FFT, the result shows a 69.25% decrease of execution time after our method reduces the number of instrumentations. As for the other programs, our method decreases the average execution time of JPEG by 31.5% and dijkstra by 34%. A counter intuitive observation in this context is in JPEG. Since JPEG has the best optimization, one may conjecture that it must show the highest decrease in execution time. However, this is not the case. Our analysis shows that the number of instrumentation points are not the only factor in the overhead caused by instrumentation. It also depends upon the location of the instrumentation and the type of variable being instrumented. For instance, deletion of an instrumentation inside a loop causes more reduction in the execution time compared to deletion of an instrumentation point which gets executed only once. Based on the above analysis, we conclude that the set of instrumentations deleted from FFT is smaller that of JPEG, but they execute more often. Hence, their deletion causes a larger decrease in the execution time of FFT as compared to JPEG.

As mentioned earlier, our instrumentation in this section involves writing the value of variables on a hard-disk. Given this assumption and the drastic decrease in execution time when optimized instrumented code is used, it follows that our method is expected to be highly beneficial in the context of embedded software where data logging involves interaction with typically slow I/O such as E2PROMs.

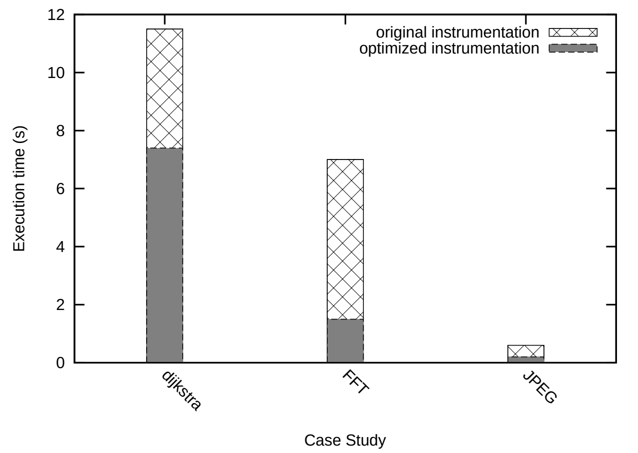

Figure 6. Comparison of execution time between instrumented and optimized instrumented code.

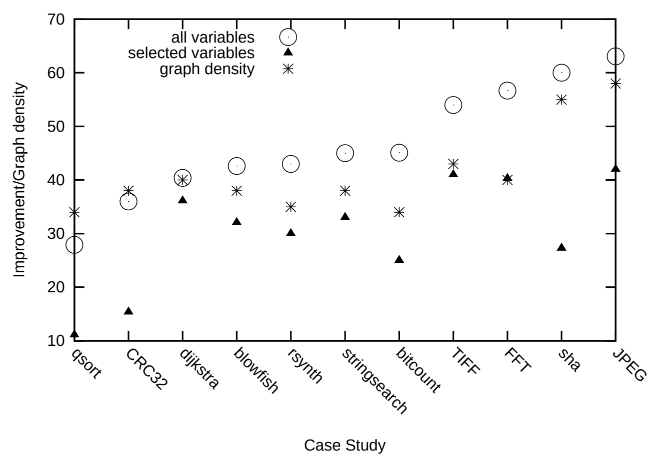

Figure 7. Results of experiments with respect to diagnosis graph density.

7.4 A Graph Theoretic Metric for Characterizing Effectiveness of Optimization

Although in Subsection 7.2, we argued that solving the ILP problem is not yet a stumbling block, it is highly desirable to have access to simple metrics that can characterize the chances that applying our method is effective before solving the ILP problem. One such metric can be characterized by density of diagnosis graph of source code. Let $|V|$ be the size of vertices and $|A|$ be the size of arcs of a diagnosis graph. Obviously, the more arcs are distributed among vertices in a diagnosis graph, the more data dependencies exist. Thus, our simple characterization metric, called *graph density* is the following:

$$\frac{|V|+|A|}{|V|}$$

As can be seen in Figure 7, in general with the increase in the *graph density*, our method achieves more reduction in instrumentation points. To understand the relation between the behaviour of the instrumentation reduction and graph density, we calculated the *correlation coefficient* of the instrumentation reduction of both *all variables* and *selected variables* against *graph density*. For the case of *all variables*, the correlation coefficient is 0.81 and for the case of *selected variables* it is 0.51. Although *graph density* is a simple metric, the obtained correlation coefficients is evidently high. In other words, when *graph density* is large in value, it shows that the number of arcs in the diagnosis graph of the corresponding program slice is considerably more than the variable vertices. This implies that variable vertices take part in multiple paths in the graph, originating from the variables of interest and, hence, diagnosis of such variables can possibly lead to the diagnoses of multiple variables.

We note that similar to any other predication metrics, our metric may suffer from outliers as well. Consider the case of *selected variables* for sha and *all variables* for FFT. In the case of FFT, graph density decreases while the reduction in the instrumentation points increases and as for sha, graph density increases while the reduction in instrumentation points decreases. Our analysis shows that this abnormality is due to whether a variable that appears in multiple diagnosis chains is in fact reachable from a variable of interest. In other words, when a variable takes part in multiple data dependency chains, it does not necessarily mean that this variable is diagnosable, making variables residing in the data chains become undiagnosable as well. Recall that the existence of a diagnosis chain between two variables is only one of the properties that needs to be satisfied to make one variable diagnosable via the other. Another vital condition is the diagnosability of the data chain dependency. Thus, if the data chain between the variables is not diagnosable, converting one of the variables into a diagnosable one does not serve the purpose. Consequently, the reduction in the instrumentation points depends not only on the number of variables taking part in multiple diagnosis chains, but also on the portion of the chains that are diagnosable. From these two conditions, FFT and sha only satisfy the former. That is why their graph density does not conform with their instrumentation improvement.

8. Conclusion and Future Work

In this paper, we focused on applying the *abstract diagnosis theory* to software debugging and testing. We formulated debugging and testing based on our interpretation of *observability* and *controllability*. Our view of observability/controllability is based on the ability of developers, testers, and debuggers to trace back a data dependency chain and observe/control the value of a variable by starting from a set of variables that are naturally observable/controllable (e.g., input/output variables). Consequently, the problem that we studied in this paper is to identify the minimum number of variables that have to be made directly observable/controllable in order for a tester or debugger to observe/control the value of another set of variables of interest, given the source code. We showed that our optimization problem is NP-complete even if the length of data dependencies is at most 2.

In order to cope with the inevitable exponential complexity, we proposed a mapping from the general problem, where the length of data dependency chains is unknown a priori, to *integer linear programming* (ILP). We implemented our method in a tool chain. The tool chain works in three phases: it (1) takes MISRA-C [14] compatible source code as input and generates program slices using LLVM [13], (2) takes the program slices and generates an integer program that encodes our optimization problem, and (3) uses MATLAB's binary integer programming toolbox to solve the integer program. Our experiments with several real-world applications show that we gain an average of 30% improvement in reducing the number of variables that need to be observed/controlled. This result is a strong motivation to apply our approach in debugging and instrumentation of embedded software, where changes must be minimal, as they can perturb the timing constraints and resource consumption. Another interesting motivation for incorporating our approach is in data logging of non-terminating embedded systems, where the size of axillary data storage for logging is limited and such devices are relatively slow.

Finally, we presented a metric for predicting whether our method for optimizing instrumentation of a program to achieve observability and controllability accomplishes reasonable reduction. We showed that this metric has a high correlation with two settings under which we conducted our experiments.

For future work, there are several open problems. One important problem is extending our formulation to the case where we can handle pointer data structures. We are currently working on developing prediction metrics better than the notion of graph density presented in Subsection 7.4. We believe that metrics that incorporate network flow or compute vertex disjoint paths are better suited, as these concepts are based on path properties rather than vertex properties. Having access to intelligent metrics is also desirable to develop transformations that can generate programs that can be instrumented more efficiently; i.e., generating programs that preserve all observational properties but have better slices for our optimization problem. Another interesting line of research is developing customized heuristics, approximation algorithms, and transformations to solve our optimization problem. One such transformation can be from our optimization problem to the satisfiability problem to employ SAT-solvers.

9. Acknowledgements

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, and ISOP IS09-06-037.

References

[1] R. Abreu and A. J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Abstraction, Reformulation, and Approximation (SARA)*, 2009.

[2] T. Ball amd and J. R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[3] Robert V. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9):87–101, 1994.

[4] David Binkley and Mark Harman. A Survey of Empirical Results on Program Slicing. In *Advances in Computers*, pages 105–178, 2003.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[6] Huy Vu Do, Chantal Robach, and Michel Delaunay. Automatic Testability Analysis for Data-Flow Designs of Reactive Systems. In *Proceedings of the First International Workshop on Testability Assessment*, pages 52–61, 2004.

[7] A. Fijany and F. Vatan. New high-performance algorithmic solution for diagnosis problem. In *IEEE Aerospace Conference (IEEEAC)*, 2005.

[8] Roy S. Freedman. Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.

[9] Hideo Fujiwara. *Logic Testing and Design for Testability*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[10] Hideo Fujiwara. Computational Complexity of Controllability/Observability Problems for Combinational Circuits. *IEEE Transactions on Computers*, 39(6):762–767, 1990.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on In Workload Characterization (WWC)*, pages 3–14, 2001.

[12] R. M. Karp. Reducibility among combinatorial problems. In *Symposium on Complexity of Computer Computations*, pages 85–103, 1972.

[13] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback directed and runtime Optimization*, page 75, 2004.

[14] Gavin McCall. *Misra-C: 2004*. MIRA Limited, Warwickshire, United Kingdom, 2004.

[15] Thanh Binh Nguyen, Michel Delaunay, and Chantal Robach. Testability Analysis Applied to Embedded Data-flow Software. In *Proceedings of the Third International Conference on Quality Software*, page 351, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Thanh Binh Nguyen, Michel Delaunay, and Chantal Robach. Testability Analysis of Data-Flow Software. *Electronic Notes in Theoretical Computer Science*, 116:213–225, 2005.

[17] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[18] Werner Schutz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[19] Werner Schutz. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems*, 7(2):129–157, 1994.

[20] Henrik Thane and Hans Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 360–369, Washington, DC, USA, 1999. IEEE Computer Society.

[21] Henrik Thane, Daniel Sundmark, Joel Huselius, and Anders Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 8, Washington, DC, USA, 2003. IEEE Computer Society.

[22] Frank Tip. A Survey of Program Slicing Techniques. Technical report, IBM, Amsterdam, The Netherlands, The Netherlands, 1994.

[23] Yves Le Traon, Farid Ouabdesselam, and Chantal Robach. Analyzing Testability on Data Flow Designs. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, page 162, Washington, DC, USA, 2000. IEEE Computer Society.

[24] Yves Le Traon and Chantal Robach. Testability Measurements for Data Flow Designs. In *Proceedings of the 4th International Symposium on Software Metrics*, page 91, Washington, DC, USA, 1997. IEEE Computer Society.

[25] Jeffrey M. Voas and Keith W. Miller. Semantic Metrics for Software Testability. *Journal of Systems and Software*, 20(3):207–216, 1993.