Hierarchical Dynamic Slicing

Tao Wang Abhik Roychoudhury
Department of Computer Science, National University of Singapore
{wangtao,abhik}@comp.nus.edu.sg

ABSTRACT

Dynamic slicing is a widely used technique for program analysis, debugging, and comprehension. However, the reported slice is often too large to be inspected by the programmer. In this work, we address this deficiency by hierarchically applying dynamic slicing at various levels of granularity. The basic observation is to divide a program execution trace into "phases", with data/control dependencies inside each phase being suppressed. Only the inter-phase dependencies are presented to the programmer. The programmer then zooms into one of these phases which is further divided into subphases and analyzed. We also discuss how our ideas can be used to augment debugging methods other then slicing (such as "fault localization", a recently proposed trace comparison method for software debugging).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Debuggers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools, Tracing*

General Terms

Experimentation, Measurement, Reliability

Keywords

Debugging, Dynamic Slicing, Phase Detection

1. INTRODUCTION

Dynamic slicing [2, 13] is a well-known technique for program analysis and comprehension. Given a program P, an input I and an "observable error", dynamic slicing can be used to find out statements of P executed under input I which can be responsible for the error (via control or data flow). Typically, the "observable error" is specified as a *slicing criterion* (l, v) — a variable v and a line number l. Thus, if the value of variable v in line number l is "unexpected" we perform slicing w.r.t. the criterion (l, v). The resultant slice can be inspected to explain the reason for the unexpected value. Dynamic slicing naturally corresponds to the notion of debugging — it tries to find out the reason for an unexpected variable value (like an unexpected output) for a given test input. However, for most real programs, the dynamic slices are too large for humans to inspect and comprehend. So, we either need to prune dynamic slices or we need tools to help a programmer understand a large dynamic slice.

In this paper, we take the second route. However, our method can be combined with techniques for pruning a dynamic slice (such as the recent work [29]). We build a dynamic slicing method where the human programmer is gradually exposed to a slice in a hierarchical fashion, rather than having to inspect a very large slice after it is computed. The key idea is simple — we systematically interleave the slice computation and comprehension steps. Conventional works on slicing have only concentrated on the computation of the slice, comprehension of the slice being left as a post-mortem activity. In our work, we integrate the two activities in a synergistic fashion as follows.

- Computation of the slice is guided (to a limited extent) by the human programmer so that very few control/data dependencies in a large slice need to be explored and inspected.
- The programmer's comprehension of the slice is greatly enhanced by the nature of our slice computation which proceeds hierarchically. Thus, for programs with long dependence chains, this allows the programmer to gradually zoom in to selected dynamic dependencies.

To understand the potential benefits one can gain from our method, let us examine the reasons which make the comprehension of dynamic slices difficult.

- Many programs have long dependence chains spanning across loops and function boundaries. These dependence chains are captured in the slice. However, the slice being a (flat) set of statements, much of the program structure (loops/functions) is lost. This makes the slice hard to comprehend.
- Programs often also have lot of inherent parallelism. So, a slice may capture many different dependence chains.

We now discuss how hierarchical computation/exploration of slices can help programmers to comprehend large slices containing these two features — (a) long dependence chains,

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

and (b) many different dependence chains. Figure 1(a) shows an example program with a long dependence chain. Consider an execution trace of the program ...3, 4, 5, 6 — where lines 3,4,5,6 of Figure 1(a) are executed. Slicing this execution trace w.r.t. the criterion $(line6, y)$ (*i.e.*, the value of y at line 6) yields a slice which contains lines $3, 4, 5, 6$ as well as lines *inside* the body of the functions $f1, f2, f3$. In other words, since the slice is a (flat) set of statements, the program structure is lost in the slice. This structure is explicitly manifested in Figure 1(b), where we show the dependence chain in a *hierarchical fashion*. In other words, the dependencies inside the functions $f1, f2, f3$ are not shown. Here, a hierarchical exploration of the dependence chains will clearly be less burdensome to the programmer. Thus, in Figure 1(b), by inspecting the dependencies hierarchically, the programmer may find it necessary to inspect the dependencies inside a specific function (say $f2$). As a result, we can avoid inspecting the dependence chain(s) inside the other functions (in this case $f1, f3$).

Now, let us consider programs with many different dependence chains. Figure 2(a) shows a schematic program with several dependence chains, and hence substantial inherent parallelism. If the slicing criterion involves the value of y in line 6 — we need to consider the dependencies between y and $x3$, y and $x2$, as well as, y and $x1$. These three dependencies are shown via broken arrows in Figure 2(b). Again, with the programmer's intervention, we can rule out some of these dependencies for exploration and inspection.

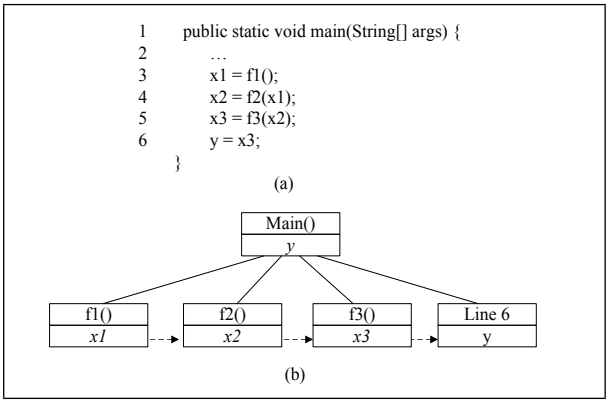

Figure 1: (a) A program with a long dynamic dependence chain (b) The corresponding phases. Dashed arrows represent dynamic dependencies that a programmer needs to follow for debugging. The reported variables for each phase appear in italics.

In summary, our method works as follows. Given an execution trace (corresponding to a program input) containing an observable behavior which is deemed as an "error" by the programmer, we divide the trace into **phases**. This division is typically done along loop/procedure/loop-iteration boundaries so that each phase corresponds to a logical unit of program behavior. Only the inter-phase data and control dependencies are presented to the programmer; the intra-phase dependencies are completely suppressed. The programmer then identifies a likely suspicious phase which is then subjected to further investigation in a similar manner (dividing the phase into sub-phases, computing dependencies across these sub-phases and so on). This process continues until the error is identified. Of course, an underlying assumption here is that the programmer will be able to identify the erroneous statement once this statement is pinpointed to him/her.[1]

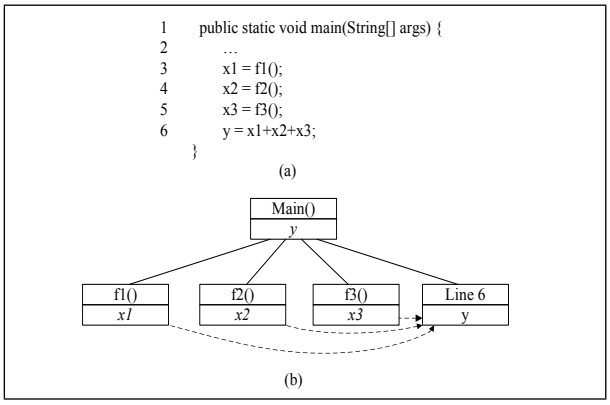

Figure 2: (a) A program with inherent parallelism (several dynamic dependence chains) (b) The corresponding phases. Dashed arrows represent dynamic dependencies that a programmer needs to follow for debugging. The reported variables for each phase appear in italics.

One may comment that such a hierarchical exploration of dynamic dependencies involves programmer's intervention, whereas conventional dynamic slicing is fully automatic. Here we should note that, the process of error detection by using/exploring a dynamic slice involves a *huge* manual effort; the manual effort in exploring/comprehending the slice simply happens *after* the computation of the slice. In our hierarchical method, we are interleaving the computation and comprehension of dynamic dependencies. As in dynamic slicing, the computation of the dynamic dependencies is automatic in our method; only the comprehension involves the programmer. Moreover, we are *gradually* exposing the programmer to the complex chain(s) of program dependencies, rather than all at once — thereby allowing better program comprehension.

Technical Contributions. The technical contributions of the paper can be summarized as follows.

- In this paper, we propose *hierarchical dynamic slicing* which systematically interleaves computation and comprehension of dynamic dependencies for program debugging. During this process, a program execution trace is divided into ***phases*** at various levels of granularity. For each phase, we collect and report dynamic data/control dependencies *across* phases. Detailed data and control dependence computation inside each phase is not exposed to the programmer, thereby reducing program understanding effort (as compared to inspecting the dynamic slice).

[1]This assumption is rather standard in existing works on debugging (*e.g.*, see the score computation by Renieris and Reiss [18], which forms the basis of experimentation in many fault localization techniques [6, 10, 18]).

- During hierarchical dynamic slicing, the generation of "***program phases***" is critical. In this paper, we propose an algorithm for *dividing an execution trace into phases in a hierarchical fashion*; this hierarchy corresponds to the levels of hierarchy we will explore for uncovering the dynamic dependencies gradually. Our phase division algorithm tries to divide a trace along control structure boundaries such as procedure calls and loop boundaries. We compare our notion of program phases with previous works on *phase detection* (*e.g.*, see [7]). These works have defined phases based on aggregate performance metrics (*e.g.*, which basic blocks are executed may define a phase).

- We have evaluated our approach on several subject programs, including programs from the Software artifact Infrastructure Repository (SIR) [8]. Our experimental results show that our approach could significantly reduce the number of statements/variables which the programmer needs to examine (as compared to conventional dynamic slicing). We have also evaluated the number of manual interventions that may be required for hierarchical dynamic slicing.

- Finally, we have applied our ideas of hierarchical dependence exploration to *fault localization*, a dynamic analysis method which compares a run showing an error with one that does not (thereby locating the cause of the error) [6, 10, 18].

Clearly, the notion of *phase* is central to our approach. In the next section, we present our notion of phases, which is then used in Section 3 to develop our slicing algorithm.

2. PHASES IN AN EXECUTION TRACE

Our definition of phase is based on the syntax structure of a program. The intuition is that programmers often use loops and methods to implement specific tasks within a program. Furthermore, programs are constructed hierarchically. Thus, a task which is implemented by a procedure may contain sub-tasks which are implemented by other procedures and/or loops. Our phase division algorithm is based on (and exploits) these observations regarding program development.

We now present our notion of phases using an example. The example program appears in Figure 3; it simulates a database system, where a user can perform various operations such as insertion, deletion and sorting. The example is similar to the `db` program in the SPEC JVM benchmark suite. The `main()` method of the program initializes a database (lines 3-5), and then presents the user with seven options (lines 7-35). Based on the user's choice, the database system invokes one of six methods, defined in the `Database` class. Finally, the `main()` method writes the database to a file before the system terminates (lines 36-37).

Consider any execution trace of the database program (given as a sequence of line numbers).

$$3, 4, 5, 7-13, 7-9, 14-17, 7-9, 30-32, 36, 37$$

Using our phase division algorithm, we compute three phases at the top-level: (1) lines 3-5 of Figure 3, (2) lines 7-35 of Figure 3, and (3) lines 36-37 of Figure 3. These phases exactly correspond to the tasks of database initialization, data processing, and finalization. These phases are detected by our method (see the three phases at the top level in Figure 4). Since our phase division is employed hierarchically, each of these three phases can be further divided into sub-phases. Let us consider the second phase – the execution of lines 7-35. The sub-phases of this phase will be the different iterations of the loop in lines 7-35. The three sub-phases perform three different operations on the database — read, insert and exit. These sub-phases are also (hierarchically) shown in Figure 4.

The preceding example captures the two main features of our approach. First of all, we use the sequence of statements visited in an execution trace and chop off the phases based on some distinguished statements (such as loop/procedure boundaries). Secondly, the phase division is done hierarchically, that is, the execution trace is divided into some phases at the top-level and each of these phases can be further sub divided and so on.

Our algorithm for dividing an execution trace H into phases appears in Figure 5. We can understand the mechanics of the algorithm as follows. Suppose that we visualize all loops as calls/return to dummy methods; thus, for each loop execution instance l contained in H suppose we insert a marker "*call to* m_l" ("*return from* m_l") at the beginning (end) of the loop. Here m_l is a dummy method name introduced by us (it does not appear in the program execution trace). Then, we first find the set of all method invocations appearing in H which are *not enclosed by any other method invocation*. We then use the entry and exit of such outermost method invocations to determine the phase boundaries. Clearly, these "outermost" method invocations may correspond to either a call to a procedure in the program or the entry to a loop in the program (recall that we converted the loop entries to dummy method invocations). If H contains outermost procedure calls as well as outermost loops, we give priority to the loops for defining the phase boundaries of H, and use the procedure calls for defining the sub-phases of these phases. We feel that programmers use loops as a higher-level structuring mechanism whereas sub-tasks appearing as initialization or activities within a loop are often written up as procedures.

If no procedure call or loop exists in H, we check whether H contains iterations of a loop and if so we set phase boundaries after a certain number of iterations (given by the constant Δ_{loop}). If H does not contain any loop iterations, we set phase boundaries after certain number of statement instances (given by the constant Δ_{stmt} in Figure 5). The reason for allowing phase division even in the absence of loops/procedures is to eventually focus at the level of statements, if the programmer chooses to do so for debugging. Java programs may have unstructured control flow by using break and continue statements. However, they are naturally loop iteration boundaries and do not require any special analysis.

The `dividePhase` method shown in Figure 5 itself is not hierarchical — given an execution trace it simply divides the trace into a finite number of phases. However, we will use it to achieve hierarchical division of a program execution trace H by invoking `dividePhase` on H, the phases of H returned by `dividePhase(`H`)`, and so on.

Differences with Conventional Phase Detection Methods. We note that phase detection for program optimization has been a rich area of research [7, 20]. Given an ex-

```
1    public class Main {
2        public static void main(String[] args) {
3        Database db = new Database();
4        db.read_db(args[1]);
5        boolean exit = false;
6        String s;

7        while (!exit) {
8            char command = readCommand();
9            switch (command) {
10           case 'r':    // read records from a file
11               s = readFromInput();
12               db.read_db(s);
13               break;
14           case 'i':    // insert a record
15               s = readFromInput();
16               db.insert(s);
17               break;
18           case 'd':    // delete the current record
19               db.delete();
20               break;
21           case 'n':    // points to the next record in the database
22               db.next();
23               break;
24           case 'p':    // points to the previous record in the database
25               db.prev();
26               break;
27           case 's':    // sort the database
28               db.sort();
29               break;
30           case 'e':    // exit the database
31               exit = true;
32               break;
33           default:
34               System.out.println("Command not support");
35               break;
             }
         }
36       db.write_db(args[2]);
37       return;
       }
       ……..
   }
```

```
38  public class Database {
39      private Vector entries;
40      private int current_record;

41      Database() {
42          entries = new Vector();
43          current_record = 0;
        }

44      public void read_db(String filename) {
45          BufferedReader dbReader = new BufferedReader(new FileReader(filename));
46          String s;
47          while ((s = dbReader.readLine()) != null) {
48              Record rec = new Record(s);
49              entries.addElement(rec);
            }
50          dbReader.close();
51          current_record = entries.size() -1;
        }

52      public void insert(String s) {
53          Record rec = new Record(s);
54          entries.add(current_record, rec);
55          current_record++;
        }

56      public void write_db(String filename) {
57          PrintWriter dbWriter = new PrintWriter(new BufferedWriter(new FileWriter(filename)));
58          for (int i = 0; i < entries.size() - 1; i++) {
59              Record rec = (Record) entries.elementAt(i);
60              dbWriter.println(rec.toString());
            }
61          dbWriter.close();
        }
        ……
    }
```

Figure 3: An example program which simulates a database system.

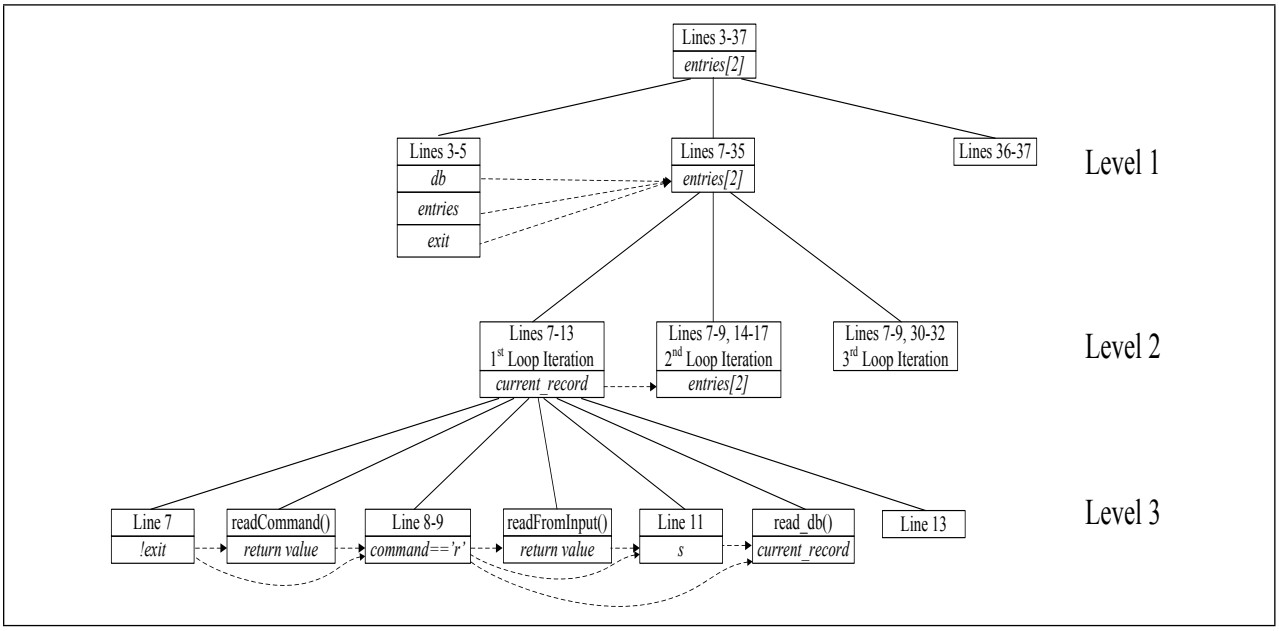

Figure 4: Phases for the running example in Figure 3. Rectangles represent phases. The reported variables for each phase appear in italics. Dashed arrows represent inter-phase dynamic dependencies that a programmer needs to follow for debugging.

```
1. dividePhase(H: an execution trace)
2. begin
3.   LOOPS = the set of loop entries which are not enclosed by any other loop or method in H;
4.   CALLS = the set of method calls which are not enclosed by any loop or method in H ;
5.   if (LOOPS != ∅)
6.       for ( each loop in LOOPS)
7.           mark entry of loop as phase boundary; mark exit of loop (if it exists in H) as phase boundary;
8.   if (LOOPS == ∅ && CALLS != ∅)
9.       for (each method invocation call in CALLS)
10.          mark the entry of call as phase boundary; mark return from call (if it exists in H) as phase boundary;
11.  if (LOOPS == ∅ && CALLS == ∅)
12.      if ( H consists of only iterations of one loop)
13.          for ( iter= every Δ_loop iterations of this loop)
14.              mark the beginning of iter as a phase boundary;
15.      else
16.          for (stmt= every Δ_stmt statement instances in H)
17.              mark the control location after stmt as a phase boundary;
18.  mark the beginning and end of H as phase boundaries;
19.  for (each marked phase boundary)
20.      ph[i]= the execution trace of H between the i-th and the (i + 1)-th phases boundaries;
21.  return ph;
22.  end
```

Figure 5: Divide an execution H into phases for debugging. Δ_{loop} (Δ_{stmt}) is a certain percentage of the number of loop iterations (statement instances).

ecution trace H, these techniques typically divide the trace H into *fixed-length intervals*. Program performance related information for each interval (such as *basic block vectors*, which, roughly speaking, capture the relative occurrences of various basic blocks in an interval) are collected. Finally, consecutive intervals with similar information are clustered into a single phase. The important issue here is that these methods (a) chop a trace into fixed-length intervals, (b) compute an aggregate metric (such as Basic Block Vectors or BBVs) for each interval, and (c) cluster adjacent intervals with similar metric into a phase.

We found the following major differences between our phase detection, and conventional phase detection methods for program optimization. These observations were obtained by comparing the phases computed by our method against the phases computed by Basic Block Vector based phase detection [20] on well-known SPECJVM benchmarks such as `jess` and `db`.

- From our experiments, we found that contrary to conventional phase detection methods, our algorithm can generate very short phases. In other words, our phase detection method is much more closely tied to the program behavior, rather than depending on artificial parameters (such as the minimum length of a phase).

- Furthermore, consecutive intervals which correspond to different logical operations and have very different execution traces will be identified to be in different phases by our method. Conventional methods can place these intervals in the same phase since the relative execution frequencies of basic blocks are similar in the two intervals (though the traces are different).

- Finally, consecutive intervals which are part of one single logical operation but have very different execution traces will be identified to be in the same phase by our method (as long as the code executed in these intervals have been modularly placed in a loop or procedure by the programmer). However, they will be placed in different phases by the conventional methods if the traces for the intervals produce substantially different metrics (such as Basic Block Vectors).

Overall, we find that our phases being based on the boundaries of control structures are more suitable for program understanding. This is because programmers often use loops or procedures to implement a specific functionality.

3. SLICING ALGORITHM

In this section, we present our slicing algorithm. Like dynamic slicing, hierarchical dynamic slicing explores dynamic data/control dependencies related to the observable error (also called the slicing criterion). The only difference lies in the manner in which these dependencies are explored and/or presented to the user. Here, we first summarize a standard dynamic slicing algorithm; we then discuss how to augment it for hierarchical dynamic slicing.

Traditionally, dynamic slicing is performed w.r.t. a slicing criterion (H, l, v), where H represents an execution trace of the program being debugged, l represents a control location in the program, and v is a program variable. A dynamic slice contains all statement instances (or statements) which have affected the value of variable v referenced at l in the trace H. A dynamic slicing algorithm can proceed by forward or backward exploration of an execution trace. Here we summarize a backwards slicing algorithm which is goal-directed (w.r.t. the slicing criterion), but requires efficient storage/traversal of the trace. During the trace traversal which starts from the statement in the slicing criterion, a dynamic slicing algorithm maintains the following quantities: (a) the dynamic slice φ, (b) a set of variables δ whose dynamic data dependencies need to be explained, and (c) a set of statement instances γ whose dynamic control dependencies need to be explained. Initially, we set the following

(a) $\varphi = \gamma =$ last instance[2] of location l in trace H, and (b) $\delta = \{v\}$.

For each statement instance *stmt* encountered during the backward traversal, the algorithm performs the following two checks. The algorithm terminates when we reach the beginning of the trace.

- *check dynamic data dependencies.* Let v_{def}^{stmt} be the variable defined by *stmt*. If $v_{def}^{stmt} \in \delta$, it means that we have found the definition of v_{def}^{stmt} which the slicing algorithm was looking for. So, v_{def}^{stmt} is removed from δ, and variables used by *stmt* are inserted into δ. In addition, *stmt* is inserted into φ and γ.

- *check dynamic control dependencies.* If any statement instance in γ is dynamically control dependent on *stmt*, all statement instances which are dynamically control dependent on *stmt* are removed from γ. Variables used by *stmt* are inserted into δ, and *stmt* is inserted into φ and γ.

When the dynamic slicing algorithm terminates, the resultant dynamic slice, (*i.e.*, the set φ) is reported back to the programmer for inspection. As discussed earlier, such a dynamic slice is often too big for human comprehension. Hierarchical dynamic slicing helps a human programmer explore and understand this large dynamic slice. Figure 6 shows our hierarchical dynamic slicing algorithm. The algorithm proceeds by employing a recursive procedure `hdslice()`. This procedure is invoked at the top level with the slicing criterion (H, l, v), where H represents an execution trace, l represents a control location in H, and v is a program variable. The `hdslice()` procedure first divides the trace H into phases (line 3 of Figure 6) by employing `dividePhase`, our phase division algorithm presented in the last section. Inter-phase dependencies (dynamic data and control dependencies across phases) are then detected and collected in the `ipd` set for each phase. Finally, these dependencies are reported to a programmer, who inspects them. Note that the programmer here is inspecting only dependencies *across phases*, not dependencies within a phase.

Each invocation of the `hdslice()` procedure detects inter-phase dependencies which are related to the observable error. This involves the following two steps.

1. determine which dynamic dependencies are (directly or indirectly) related to the observable error, and then

2. determine which of the dynamic dependencies identified in step 1 are inter-phase dependencies.

The first step is drawn from dynamic slicing, while the second step is novel to hierarchical dynamic slicing. Dynamic slicing algorithms maintain sets δ (γ) to capture the variables (statements) whose data (control) dependencies are yet to be explained. In hierarchical dynamic slicing, we maintain several δ and γ sets, one for each phase. The splitting of δ and γ sets is to ease the task of determining which dynamic dependencies are inter-phase dependencies. For every statement instance *stmt* encountered during the backward traversal for slicing, let $ph[s]$ be the phase which *stmt* belongs to. If *stmt* defines an variable v_{def} which is included in $ph[i].\delta$ for some value of i, this means that v_{def} is used by statement instance in the ith phase $ph[i]$, and *stmt* is involved in a dynamic data dependence which is related to the observable error. We can then easily determine whether this data dependence spans phase boundaries by determining whether $ph[s]$ and $ph[i]$ are the same phase. Similarly, we could use the γ sets of the individual phases to determine whether a dynamic control dependence spans phase boundaries. The data and control dependencies which are thus identified to be inter-phase dependencies are captured in the `ipd` sets (Line 33 of Figure 6).

The programmer can use the values of the variables involved in inter-phase dynamic dependencies[3] to identify the "first" (in terms of order of occurrence in the trace) suspicious inter-phase dependency. We would now like to employ slicing to explain this suspicious dependency. However, slicing requires the slicing criterion to be set as a triple

$$\langle trace,\ \ control\ location,\ \ variable \rangle$$

Thus, we need to extract these parameters from an inter-phase dependency if it is deemed "suspicious" by the programmer. For an inter-phase dependency from phase p to phase p', we set the execution trace for phase p as the trace to be explored for further slicing. Thus, the phase p is marked as the error phase *err_ph* and its trace is the execution trace H_{err} to be further explored (see Lines 37-40 of Figure 6). Also, given any suspicious inter-phase dependency, we can associate a variable v_{err} with it. For data dependencies, v_{err} is the variable which is defined/used; for control dependencies, we can consider an auxiliary boolean variable corresponding to the guard involved in the control dependency. Finally, the control location where v_{err} is defined in the error phase *err_ph* is marked as the suspected erroneous control location l_{err}. We now recursively invoke the hierarchical dynamic slicing procedure `hdslice` (see Line 41 of Figure 6) with slicing criterion $(H_{err}, l_{err}, v_{err})$.

Example. We use the program of Figure 3 as the running example. This program simulates a database system where the variable `current_record` should always point to the last database record operated on. We have introduced a bug in Line 51, which should be

```
current_record = entries.size()
```

instead of

```
current_record = entries.size() - 1
```

Let us consider the following execution trace — the program first reads one record into `entries` by executing lines 3-5, reads two additional records into `entries` by executing lines 7-13, and inserts one record into the database by executing line 7-9,14-17. Finally, the program exits the database by executing 7-9, 30-32, and writes the resultant database into a file by executing 36-37. Let us suppose the records read/inserted were "*Africa*", "*America*", "*Antarctica*" and "*Asia*" (in this order). Then, because of the faulty statement in line 51 of the program, the content of the database at the end of execution will be: "*Africa*", "*America*", "*Asia*", "*Antarctica*". In other words, the last and second last elements of the program array `entries` (*i.e.*, `entries[2]` and `entries[3]`) are reversed. This error can be observed from the file to which the database is written.

[2]We could also consider any other instance, or even all instances.

[3]The programmer may need to re-execute the program to obtain these values if the values are not captured in the execution trace.

```
1. hdslice(H: an execution trace, l: a location, v: a variable)
2. begin
3.   ph = dividePhase(H); /* See Figure 5 for dividePhase algorithm */
4.   for ( each ph[i] of ph )
5.       ph[i].δ = ∅;
6.       ph[i].γ = ∅;
7.       ph[i].ipd = ∅;
8.   stmt_e = the last statement instance of location l in trace H; /* the "suspicious" statement instance */
9.   let ph[e] = the phase which stmt_e belongs to;
10.  ph[e].δ = {v};
11.  ph[e].γ = {stmt_e};
12   stmt= stmt_e;
13.  while( stmt is defined )
14.      inInterPhaseDependence = false;
15.      let ph[s] = the phase which stmt belongs to;
16.      v_def = variable defined at stmt;
17.      V_use = the set of variables used at stmt;
18.      for ( each ph[i] of ph )
19.          if ( v_def ∈ ph[i].δ )
20.              ph[i].δ = ph[i].δ - {v_def};
21.              ph[s].δ = ph[s].δ ∪ V_use;
22.              ph[s].γ = ph[s].γ ∪ {stmt};
23.              if ( i != s ) /* ph[i] and ph[s] are not the same phase */
24.                  inInterPhaseDependence = true;
25.          if ( any statement instance in ph[i].γ is dynamically control dependent on stmt )
26.              CD_s = the set of statement instances which are dynamically control dependent on stmt;
27.              ph[i].γ = ph[i].γ - CD_s;
28.              ph[s].δ = ph[s].δ ∪ V_use;
29.              ph[s].γ = ph[s].γ ∪ {stmt};
30.              if ( i != s ) /* ph[i] and ph[s] are not the same phase */
31.                  inInterPhaseDependence = true;
32.      if ( inInterPhaseDependence )
33.          ph[s].ipd = ph[s].ipd ∪ {⟨stmt, v_def⟩};
34.      stmt = the statement instance before stmt;
35.  for ( each ph[i] of ph )
36.      report inter-phase dependencies ph[i].ipd to the programmer;
37.  ⟨stmt_err, v_err⟩= ProgrammerIntervention();
38.  l_err = the control location of stmt_err;
39.  err_ph = the phase which stmt_err belongs to, i.e. the suspicious phase;
40.  H_err = the execution trace for the suspicious phase err_ph;
41.  hdslice(H_err, l_err, v_err);
42. end
```

Figure 6: The Hierarchical Dynamic Slicing algorithm. After the algorithm reports all inter-phase dynamic dependencies (at line 36), the programmer needs to identify "suspicious" ones and return the first (in order of occurrence) suspicious inter-phase dependency (at line 37).

Figure 4 partially illustrates how the hierarchical dynamic slicing algorithm works to locate the faulty statement. Rectangles at the same horizontal level in Figure 4 are the phases generated in the same invocation of `hdslice()` procedure (our slicing algorithm). Dashed arrows in Figure 4 represent inter-phase dependencies. Variables involved in inter-phase dependencies for each phase appear in italics in Figure 4. We do not show the statement instances which define these variables, since they are clear from the program in Figure 3. As discussed earlier, the variable for an inter-phase control dependency may be captured by an auxiliary boolean variable representing the guard corresponding to the control dependency (see the guard *command == 'r'* for the phase representing lines 8-9 in Figure 4). Note that the variable(s) mentioned for each phase in Figure 4 effectively serve as the "**outputs**" of the phase which are passed to the succeeding phases as "**inputs**". This input-output relationship constitutes the inter-phase dependencies which are shown using dashed arrows in Figure 4. Thus, given an invocation of `hdslice` on an execution trace H , the programmer can inspect these "outputs" of the phases corresponding to given "inputs" and check whether this matches his/her expectation of the input-output relationship supposed to be captured by the corresponding phase. *The programmer can avoid thinking about the computation inside any phase.*

For the example program given in Figure 3, the `hdslice()` procedure is first invoked with the execution of lines 3-37 of Figure 3, and the "incorrect" variable `entries[2]` (deemed incorrect since it is involved in the observable error in this example). This execution is divided into three phases, as shown in Figure 4. The second phase (execution of lines 7-35) defines the variable `entries[2]`. Although the first phase (execution of lines 3-5) defines several variables (including `entries`/`exit`) which are involved in inter-phase dependencies, the programmer in this case deems the initialization code in the first phase as "correct". Typically, the programmer will do this by inspecting the "outputs", that is the values of variables produced by execution of first phase.

In this case, the programmer observes that at the end of the first phase `entries[2]` is not initialized (in fact, only `entries[0]` is initialized). So, the first phase is clearly unrelated to the error in `entries[2]`, and the programmer zooms into the second phase for further investigation. This results in a recursive invocation of the `hdslice` procedure on the second phase. The second phase is then further divided into three sub-phases. Again, the programmer observes from the inter-phase dependencies that the first sub-phase produces `current_record` as output which is fed as input to the second sub-phase (shown via dashed arrows in Figure 4). Furthermore, the value of the `current_record` variable is "unexpected"; this is based on the programmer's expectation that `current_record` should be an index to the current last record of the database. Consequently the programmer focuses on the value of `current_record` in the first sub-phase (lines 7-13 of Figure 3) via another invocation of `hdslice`.

4. EXPERIMENTS

In this section, we discuss the implementation and experiments for our hierarchical dependence exploration method.

4.1 Slicing Experiments

Subject	Description	Size
NanoXML	a XML parser for Java	7646 LOC 24 classes
JTopas	a Java library for parsing text	5400 LOC, 50 classes
Apache JMeter	a performance testing tool	43400 LOC, 389 classes

Table 1: Descriptions of subject programs used to evaluate the effectiveness of our hierarchical dynamic slicing approach for debugging.

We have implemented hierarchical dynamic slicing on top of the *Jslice* dynamic slicing tool [24] for Java programs. *Jslice* performs backwards dynamic slicing of sequential Java programs. Since backwards slicing requires storing of the execution trace, *Jslice* performs online compression during trace collection. The compressed trace representation is traversed without decompression during slicing. To understand the data compression methods used for trace representation in *Jslice*, the reader is referred to [25]. Our prototype implementation of hierarchical dynamic slicing also uses this compressed trace representation. In particular, the phase detection/representation/traversal in the execution trace are all done in compression domain.

We applied our prototype implementation to subject programs written in Java available from the Software-artifact Infrastructure Repository (SIR) [8]. Since our slicing technique is applicable to sequential programs, we chose the *NanoXML*, *JTopas* and *JMeter* subjects. Note that *JMeter* is actually a multi-threaded Java program, but some test cases from [8] run only one thread of *JMeter* thereby making our slicing technique applicable. Descriptions and sizes of these subjects are shown in Table 1.

Each SIR subject comes with a pool of test inputs. SIR [8] also provides several buggy versions of each subject program, where each buggy version has exactly one injected bug. Some of the buggy versions are such that none of the given test inputs (for the corresponding subject program) exposes the bug. We did not include them in our experiments, since the failing input (*i.e.*, the input corresponding to the execution trace to slice on) did not come with the subject programs. Furthermore, some other buggy versions are such that the faulty statement is not included in the dynamic slice[4]; we left out these buggy versions as well. Finally, we got three buggy versions for each of our three subject programs — resulting in a total of nine buggy programs.

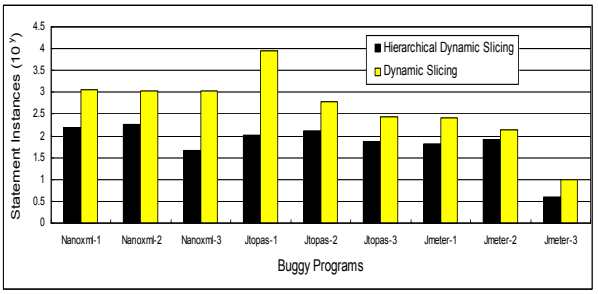

Figure 7: The number of statement instances that a programmer has to examine using the hierarchical dynamic slicing approach and the conventional dynamic slicing approach. *The figure is in log scale showing that our hierarchical approach is often orders of magnitude better.*

Statement Instances Examined. We first tried to evaluate the utility of hierarchical dynamic slicing for program debugging. Figure 7 compares (in a *log scale*) the number of statement instances which a programmer has to examine using the hierarchical dynamic slicing approach and the conventional dynamic slicing approach. For the hierarchical approach, a programmer has to examine only statement instances involved in inter-phase dependencies. We compare this against the size of the dynamic slice, which is what the programmer will typically examine in conventional dynamic slicing. As we can see from Figure 7, our approach can significantly reduce (often by *orders of magnitude*) the number of statement instances which the programmer needs to examine for debugging. The improvement comes from the usage of phases in slicing. By dividing an execution into phases and reporting inter-phase dependencies, a programmer can quickly identify suspicious dependence chains. The inter-phase dependencies effectively expose the "inputs" and "outputs" of each phase. This allows the programmer to think of each phase in terms of the expected input-output relationship rather than worrying about the computations within each phase. Consequently, the number of statement instances to be investigated is significantly reduced.

User Interaction. One of the key issues in hierarchical dynamic slicing is the interleaving of slice computation and comprehension steps. The aim is to aid program understanding by *gradually* exposing the programmer to compli-

[4]This is because dynamic slicing can only locate errors where the faulty statement is present in the program as well as in the execution trace (*e.g.*, it cannot capture "missing code" errors).

cated dependence chains. However, if the number of intervention steps required from the programmer is overwhelming, this can undermine the method's utility. For this reason, we experimentally evaluated the number of manual interventions required in the hierarchical dynamic slicing of our subject programs. The results appear in the column *# Interventions* of Table 2. In the experiments, we chose "simple" test cases which result in shorter length execution traces. We feel that this is natural, since programmers also favor a shorter execution trace demonstrating an error (over longer execution traces showing the same error) for debugging purposes. In practice, the programmer can generate such "simple" test cases (which produce shorter execution traces) based on his/her intuition about the program, or (s)he can use automatic methods for simplifying test inputs [16, 28].

From our slicing algorithm (Figure 6) it seems that the number of programmer interventions will be exactly equal to the number of hierarchy levels we explore (*i.e.*, the number of times we invoke the phase division algorithm). We were pleasantly surprised to find that *the number of manual interventions is often less than the number of hierarchies explored* (see the last two columns in Table 2). After dividing an execution trace into phases, we may find that dependence chains which are relevant to the observable error all lie in one phase, and dependence chains in other phases have no effect on the observable error. In other words, there is no inter-phase dependence which is relevant to the observable error. In such a situation, our approach could proceed to the phase which is relevant to observable error, without any user intervention.

Post-mortem pruning of slices. We also tried out the following variation of our experiments on slicing. We first compute the entire dynamic slice automatically, as in conventional slicing techniques. However, the slice is explored *post-mortem* along dependence chains (chains of length 1,2,...), starting from the slicing criterion until the error is found. Note that such an exploration is also not automatic since the programmer has to look through the dependence chains to check whether the cause of the error is found.

We compared the number of statement instances examined by such post-mortem exploration of the slice with our hierarchical dynamic slicing method (which performs exploration/comprehension as the slice is being computed). We found that the number of statement instances examined by this post-mortem guided exploration of the slice is still *substantially* higher than those examined by our hierarchical dynamic slicing method for most of the buggy programs. To be precise, hierarchical dynamic slicing required substantially less statements to examine (as compared to the pruned slices) in 6 out of the 9 buggy programs. This is presumably because exploiting user-guidance *during* the slice computation (rather then *after* the slice computation) makes the exploration/comprehension of the slice more goal-directed.

4.2 Application to Fault Localization

We can employ our idea of hierarchical dependence chain exploration to dynamic analysis methods other than slicing. In particular, we have done so for a recently proposed debugging method called fault localization [6, 10, 11, 18, 26] which proceeds by comparing the *failing program run* with a *successful run* (a run which does not demonstrate the error).

Subjects	# Interventions	# Hierarchy Levels
Nanoxml-1	17	29
Nanoxml-2	22	26
Nanoxml-3	2	7
Jtopas-1	4	5
Jtopas-2	10	10
Jtopas-3	7	8
Jmeter-1	3	3
Jmeter-2	8	10
Jmeter-3	2	2

Table 2: Number of Programmer Interventions & Hierarchy Levels in Hierarchical Dynamic Slicing.

We first recall our previous work on fault localization and then demonstrate via experiments how hierarchical dependence exploration may help in processing the bug report.

In our previous work [10], we have proposed a fault localization algorithm which compares control flow of two execution traces. Given a failing run H_f, and a successful run H_s, our comparison algorithm captures all branch instances with similar contexts but different outcomes in H_f and H_s (details appear in [10]). These branches are presented as a bug report R to the user. The algorithm in Figure 8 elaborates how to locate the cause of the error using such a bug report R (where the report R consists of branch instances br). Given the execution trace H_f of the failing run and a bug report R, the programmer starts from branches in R, and proceeds by examining statement instances $stmt$ in H_f which have dependence chains to/from branches in the bug report. We look for statements which have a dependence chain of length 0 w.r.t. the bug report (*i.e.*, statements inside the bug report !), followed by statements with dependence chain of length 1, and so on. This systematic exploration based on length of dependence chain is supported by the function `minDepLen`$(stmt, br)$ (see line 7 of Figure 8); it returns the minimum length among all the dependence chains between statement $stmt$ and branch br.

```
1. localize(H_f: failing execution run, R: a bug report)
2. begin
3.   φ = ∅;
4.   len = 0;
5.   while (φ does not include all statement instances of H_f)
6.       for (every statement instance stmt of H_f)
7.           if ( ∃br ∈ R s.t. minDepLen(stmt, br) == len)
8.               φ = φ ∪ {stmt};
9.       if (φ contains error cause, examined by programmer)
10.          return φ;
11.      len = len + 1;
12.      return ∅;
13. end
```

Figure 8: Locating the erroneous statement from a bug report.

We develop *hierarchical* exploration of the bug report produced by fault localization via the following two steps. The phases are computed w.r.t. the failing run.

1. determine which dynamic dependencies are related to branch instances in the bug report R, and

2. determine which of the dynamic dependencies identified in the first step cross phase boundaries.

Thus, we essentially capture inter-phase dynamic dependencies of the branch statement instances mentioned in the bug report (which was produced by comparing the successful and failing traces). As in conventional fault localization, we stop the exploration when the actual buggy statement is found. We have implemented a prototype of this approach. In our preliminary experiments, we have applied the prototype to the `schedule` and `print_tokens` subjects drawn from the Siemens suite [19]. `schedule` implements a priority scheduler, and `print_tokens` implements a lexical analyzer. Both the programs have about 400-500 lines of code. There are several versions of both programs in the Siemens suite [19], where each version has exactly one injected bug. We have conducted our experiments with all versions of `schedule` and `print_tokens`, a total of 16 buggy programs.

Figure 9 compares the number of statement instances explored during hierarchical exploration of the bug report's dependence chains (as opposed to normal exploration). In one of the 16 programs, the actual bug was unrelated to the branches contained in the bug report. In this case, we could not locate the erroneous statement using either methods. Among the remaining 15 programs, the hierarchical approach dramatically reduced number of statement instances to examine in 12 buggy programs. However, the conventional fault localization method outperformed the hierarchical approach for 3 buggy programs. This is not surprising, since in some programs the bug report may already pinpoint the error as a result of which conventional fault localization is very useful for debugging.

Thus, based on our experiments, we can suggest the following *hybrid* fault localization method. We first automatically generate the bug report by comparing the failing and successful runs as in conventional fault localization. After the programmer receives the bug report, (s)he first checks whether the actual bug is already contained in the bug report. If this is not the case, (s)he proceeds with hierarchical exploration of the dependence chains to/from statements in the bug report.

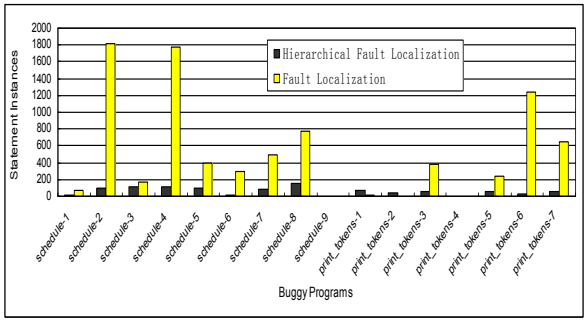

Figure 9: The number of statement instances that a programmer has to examine using the hierarchical fault localization method and the non-hierarchical one.

5. RELATED WORK

The idea of slicing came from the work of Weiser [27], who proposed it as a program understanding aid. Subsequently, it was observed that the slice corresponding to a specific input (called the dynamic slice) is much smaller than the static slice. Dynamic slicing was first proposed by Korel and Laski [13] and subsequently investigated in a rich body of work (*e.g.*, [2, 22, 29]).

It has been shown that dynamic slicing can be used in program debugging [3, 14]. In fact, it can significantly reduce the number of statements to be inspected for locating the cause of an error [23, 30]. However, the resultant dynamic slice is typically reported to a programmer without any post-processing. For real-life programs, the dynamic slice is often too large to be inspected/examined by a human programmer. Recently, Sridharan et al. proposed "thin slicing" — a mechanism to hierarchically explore the static slice according to data flow [21]. In this paper, we have proposed an algorithm to hierarchically construct *and* explore the dynamic slice according to the control constructs in the program. We belive our work can be fruitfully combined with a dynamic adaptation of thin slices.

The idea of using the phases of an execution trace for debugging has appeared in earlier works. Miller and Choi [15] propose to do so by presenting the dynamic dependence graph of each phase to the user. This is effectively exposing the dynamic dependence chains inside the phase completely to the programmer, thereby burdening him/her with lot of redundant information! Our approach is exactly the *reverse* — we seek to hide the dynamic dependence chains inside a phase. Instead we summarize a phase via its "input" and "output" variables, which is gleaned from the inputs/outputs of the program as well as those of the preceding and succeeding phases.

Balmas [4] proposed hierarchical exploration of *static program dependence graphs*. This approach was later extended for hierarchical visualization of dynamic data dependencies [5]. We note that [5] only discusses better visualization of dynamic dependencies, whereas we interleave the dependence computation and comprehension steps. Indeed this is the main thesis of our approach – we feel that *program comprehension cannot be left as a post-mortem activity, and should be used to guide dependence computation.* Additionally, we have proposed a phase division method which helps identify and structure the exact feedback needed from the programmer in general — the programmer needs to select one from among a given set of inter-phase dependencies. Last, but certainly not the least, we have conducted detailed experiments to show that our approach has the potential to make dynamic slicing more useful for software debugging.

As far as slicing tools are concerned, existing dynamic slicing tools [1, 17, 24, 14] highlight dynamic slices in the source code browser. Typically, it is the programmer's responsibility to analyze the statements / statement instances, and identify the suspicious statement instances for debugging.

The works of [9, 12] extend algorithmic debugging (where a user queries about behavior of procedures) by pruning some of the queries via dynamic slicing. These works bear some similarity to our work, since they also rely on summarizing the behaviors of execution trace fragments (or phases). However to summarize a phase (say corresponding to a procedure call), they rely on a static summary of the procedure itself. In particular, they summarize the variable definitions of a procedure, which in the context of Java programs will require static points-to analysis. In contrast, we only seek to identify the inter-phase *dynamic* dependencies which can proceed efficiently without any points-to analysis.

Recently, various software fault localization approaches have been proposed. They proceed by comparing two runs of a buggy program [6, 10, 11, 18, 26], where the two runs differ in whether any error is observed. Unlike dynamic slicing, these methods require that there exist some execution runs without the observable error, so that the comparison can proceed. We have employed our hierarchical exploration on these techniques, thereby augmenting them. Our initial experiments (see Section 4.2) show good potential in using such hierarchical fault localization to locate program errors.

Finally, we note that our approach is very different from the recently proposed Hierarchical Delta Debugging method [16]. This work seeks to simplify the program input that causes a program to fail. In this endeavor, it exploits the hierarchy present in the program input (*e.g.*, if the program input is an XML or HTML file). Our approach, on the other hand, seeks to hierarchically detect and explore the control/data dependence chains in a program.

6. DISCUSSION

In this paper, we have proposed hierarchical dynamic slicing to aid the comprehension of dynamic slices. The proposed application is in program debugging, where the programmer is gradually guided through complex program dependence chains. This is as opposed to the arduous task of understanding a full dynamic slice, where all of the comprehension is left to the programmer.

We have conducted detailed experiments on well-known subject programs written in Java drawn from the SIR repository [8] to evaluate the effectiveness of this approach. Our experiments show a substantial reduction in program understanding effort for our subject programs. In future, we plan to conduct more detailed experiments for hierarchical slicing as well as hierarchical fault localization.

Acknowledgments

This work was partially supported by a Public Sector Research Grant from A*STAR, Singapore.

7. REFERENCES

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *SPE*, 23(6), 1993.

[2] H. Agrawal and J. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[3] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, 1995.

[4] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal of Software Maintenance: Research and Practice*, 2004.

[5] F. Balmas, H. Wertz, and R. Chaabane. DDgraph: a tool to visualize dynamic dependences. In *Workshop on Program Comprehension through Dynamic Analysis*, 2005.

[6] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[7] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *MICRO*, 2003.

[8] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 2005. `http://www.cse.unl.edu/~galileo/sir`.

[9] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri. Generalized algorithmic debugging and testing. In *PLDI*, 1991.

[10] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *CC*, 2006.

[11] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[12] M. Kamkar. Application of program slicing in algorithmic debugging. *Information and Software Technology*, 1998.

[13] B. Korel and J. W. Laski. Dynamic program slicing. *IPL*, 29(3), 1988.

[14] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. In *AADEBUG*, 1997.

[15] B. P. Miller and J. D. Choi. A mechanism for efficient debugging of parallel programs. In *PLDI*, 1988.

[16] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *ICSE*, 2006.

[17] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: An efficient and economical way of reducing slice. In *ICSE*, 1999.

[18] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[19] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *TSE*, 24(6), 1998.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[21] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[22] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[23] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *TOPLAS*, 17(2), 1995.

[24] T. Wang and A. Roychoudhury. *Jslice*: A dynamic slicing tool for Java programs. National University of Singapore, `http://jslice.sourceforge.net`.

[25] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *ICSE*, 2004.

[26] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE*, 2005.

[27] M. Weiser. Program slicing. *TSE*, 10(4), 1984.

[28] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 28(2), 2002.

[29] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.

[30] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, 2005.