THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRB3/07

Dynamic Slicing on Java Bytecode Traces

Tao, WANG and Abhik ROYCHOUDHURY

March 2007

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Dynamic Slicing on Java Bytecode Traces

TAO WANG and ABHIK ROYCHOUDHURY
National University of Singapore, Singapore

Dynamic slicing is a well-known technique for program analysis, debugging and understanding. Given a program P and input I, it finds all program statements which directly/indirectly affect the values of some variables' occurrences when P is executed with I. In this paper, we develop a dynamic slicing method for sequential Java programs. Our technique proceeds by backwards traversal of the bytecode trace produced by an input I in a given program P. Since such traces can be huge, we use results from data compression to compactly represent bytecode traces. The major space savings in our method come from the optimized representation of (a) data addresses used as operands by memory reference bytecodes, and (b) instruction addresses used as operands by control transfer bytecodes. We show how dynamic slicing algorithms can directly traverse our compact bytecode traces without resorting to costly decompression. We also extend our dynamic slicing algorithm to perform "relevant slicing"; the resultant slices can be used to explain omission errors that is, why some events did not happen during program execution. Detailed experimental results on space/time overheads of tracing and slicing are reported in the paper. The slices computed at the bytecode level are translated back by our tool to the source code level with the help of information available in Java class files. Our *JSlice* dynamic slicing tool has been integrated with the Eclipse platform and is available for usage in research and development.

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: Algorithms, Experimentation, Measurement

Additional Key Words and Phrases: Program slicing, Tracing, Debugging

1. INTRODUCTION

Program slicing [Weiser 1984] is a well-known technique for program debugging and understanding. Roughly speaking, program slicing works as follows. Given a program P, the programmer provides a slicing criterion of the form (l, V), where l is a control location in the program and V is a set of program variables referenced at l. The purpose of slicing is to find out the statements in P which can affect the values of V at l via control and/or data flow. So, if during program execution the values of V at l were "unexpected", the corresponding slice can be inspected to explain the reason for the unexpected values.

Slicing techniques are divided into two categories: static and dynamic. Static (Dynamic) slicing computes the fragment affecting V at l (some occurrence of l) when the input program is executed with any (a specific) input. Thus, for the same slicing criterion, dynamic slice for a given input of a program P is often smaller than the static slice of P. Static slicing techniques typically operate on a program dependence graph (PDG); the nodes of the PDG are simple statements / conditions and the edges correspond to data / control dependencies [Horwitz et al. 1990]. Dynamic slicing w.r.t. an input I on the other hand, often

A preliminary version of this paper appeared in the International Conference on Software Engineering (ICSE) 2004, see [Wang and Roychoudhury 2004]. *Authors' addresses*: Tao Wang and Abhik Roychoudhury, School of Computing, National University of Singapore, S16 Level 5, 3 Science Drive 2, Singapore 117543. E-mail: {wangtao,abhik}@comp.nus.edu.sg proceeds by collecting the execution trace corresponding to I. The data and control dependencies between the statement occurrences in the execution trace can be pre-computed or computed on demand during slicing [Zhang et al. 2005].

Due to the presence of objects and pointers in programs, static data dependence computations are often conservative, leading to very large static slices. On the other hand, dynamic slices capture the closure of *dynamic* data and control dependencies, hence they are much more precise, and more helpful for narrowing the attention of the programmer. Furthermore, since dynamic slices denote the program fragment affecting the slicing criterion for a *particular* input, they naturally support the task of debugging via running of selected test inputs.

Though dynamic slicing was originally proposed for debugging [Korel and Laski 1988; Agrawal and Horgan 1990], they have subsequently also been used for program comprehension in many other innovative ways. In particular, dynamic slices (or their variants which also involve computing the closure of dependencies by trace traversal) have been used for studying causes of program performance degradation [Zilles and Sohi 2000], identifying isomorphic instructions in terms of their run-time behaviors [Sazeides 2003] and analyzing spurious counter-example traces produced by software model checking [Majumdar and Jhala 2005]. Even in the context of debugging, dynamic slices have been used in unconventional ways e.g. [Akgul et al. 2004] studies reverse execution along a dynamic slice. Thus, dynamic slicing forms the core of many tasks in program development and it is useful to develop efficient methods for computing dynamic slices.

In this paper, we present an infrastructure for dynamic slicing of Java programs. Our method operates on bytecode traces; we work at the bytecode level since slice computation may involve looking inside library methods and the source code of libraries may not always be available. First, the bytecode stream corresponding to an execution trace of a Java program for a given input is collected. The trace collection is done by modifying a virtual machine; we have used the Kaffe Virtual Machine in our experiments. We then perform a backward traversal of the bytecode trace to compute dynamic data and control dependencies on-the-fly. The slice is updated as these dependencies are encountered during trace traversal. Computing the dynamic data dependencies on bytecode traces is complicated due to Java's stack based architecture. The main problem is that partial results of a computation are often stored in the Java Virtual Machine's operand stack. This results in implicit data dependencies between bytecodes (involving data transfer via the operand stack). For this reason, our backwards dynamic slicing performs a "reverse" stack simulation while traversing the bytecode trace from the end.

Dynamic slicing methods typically involve traversal of the execution trace. This traversal may be used to pre-compute a dynamic dependence graph or the dynamic dependencies can be computed on demand during trace traversal. Thus, the representation of execution traces is important for dynamic slicing. This is particularly the case for backwards dynamic slicing where the trace is traversed from the end (and hence needs to be stored). In practice, traces tend to be huge; [Zhang et al. 2005] reports experiences in dynamic slicing programs like `gcc` and `perl` where the execution trace runs into *several hundred million instructions*. It might be inefficient to perform post-mortem analysis over such huge traces. Consequently, it is useful to develop a compact representation for execution traces which capture both control flow and memory reference information. This compact trace should be generated *on-the-fly* during program execution.

Our method proceeds by on-the-fly construction of a compact bytecode trace during program execution. The compactness of our trace representation is owing to several factors. First, bytecodes which do not correspond to memory access (*i.e.* data transfer to and from the heap) or control transfer are not stored in the trace. Operands used by these bytecodes are fixed and can be discovered from Java class files. Secondly, the sequence of addresses used by each memory reference bytecode or control transfer bytecode is stored separately. Since these sequences typically have high repetition of patterns, we exploit such repetition to save space. We modify a well-known lossless data compression algorithm called SEQUITUR [Nevill-Manning and Witten 1997] for this purpose. This algorithm identifies repeated patterns in the sequence on-the-fly and stores them hierarchically.

Generating compact bytecode traces during program execution constitutes the first phase of our dynamic slicer. Furthermore, we want to traverse the compact execution trace to retrieve control and data dependencies for slicing; this traversal should be done without decompressing the trace. In other words, the program trace should be collected, stored and analyzed for slicing – all in its compressed form. This is achieved in our dynamic slicer which traverses the compact bytecode trace and computes the data/control dependencies in compression domain. Since we store the sequence of addresses used by each memory-reference/control-transfer bytecode in compressed format, this involves marking the “visited” part of such an address sequence without decompressing its representation.

Finally, we extend our dynamic slicing method to support ”relevant slicing”. Traditional dynamic slicing algorithms explain the values of variables V at some occurrences of a control location l, by highlighting the executed program statements which affect the values of V at l. They do not report the statements which affect the value of V at l, by *not being executed*. To fill this caveat, the concept of relevant slicing was introduced in [Agrawal et al. 1993; Gyimóthy et al. 1999]. We design and implement a relevant slicing algorithm working on our compact bytecode traces.

Summary of Contributions. In summary, the main contribution of this paper is to report methods for dynamic slicing of *single threaded* Java programs. Our slicer proceeds by traversing a compact representation of a bytecode trace and constructs the slice as a set of bytecodes; this slice is then transformed to the source code level with the help of Java class files. Our slicing method is complicated by Java’s stack based architecture which requires us to simulate a stack during trace traversal. We have made our *Jslice* slicing tool available from `http://jslice.sourceforge.net/` To the best of our knowledge, ours is the first dynamic slicing tool for Java programs.

Since the execution traces are often huge, we develop a space efficient representation of the bytecode stream for a single threaded Java program execution. This compressed trace is constructed on-the-fly during program execution. Our dynamic slicer performs backward traversal of this compressed trace *directly* to retrieve data/control dependencies, that is, slicing does not involve costly trace decompression. Our compressed trace representation is interesting in general as a program trace representation. For our subject programs (drawn from standard suites such as the Java Grande benchmark suite or the SPECjvm suite) we obtain compression in varying amounts ranging from $5 - 5000$ times. We show that the time overheads for constructing this representation on-the-fly during program execution is tolerable.

We also enhance our dynamic slicing algorithm to capture “omission errors”. The extended dynamic slicing method is called as “*relevant slicing*” [Agrawal et al. 1993; Gyimóthy et al. 1999]. The relevant slicing algorithm also operates *directly* on the compressed bytecode traces, as our dynamic slicing algorithm. Our experimental results indicate that the additional capability of relevant slices (*i.e.* capturing omission errors) comes at the cost of modest additional overheads in terms of computation time or slice sizes.

The main changes w.r.t. the conference version [Wang and Roychoudhury 2004] of this paper are as follows. We explain in full details the dynamic slicing and relevant slicing algorithms (*e.g.* issues like (a) computing slices without trace decompression, and (b) computing data dependencies via stack simulation, are elaborated in this version). In addition, more extensive experiments have been conducted for understanding the efficiency of our compact trace representation and the sizes of dynamic/relevant slices. Last but certainly not the least, we have made our Java dynamic slicing tool available for use.

Section Organization. The rest of this paper is organized as follows. Section 2 describes our compressed representation of Java bytecode traces. Section 3 and 4 discuss our dynamic slicing and relevant slicing algorithms. Section 5 presents experimental evaluation, while Section 6 describes our *JSlice* dynamic slicing tool for single-threaded Java programs. Section 8 summarizes related work. We conclude the paper in section 9.

2. COMPRESSED BYTECODE TRACE

In this section, we will discuss how to collect compact bytecode traces of Java programs *on the fly*. This involves a discussion of the compaction scheme as well as the necessary instrumentation. The compaction scheme used by us is exact, lossless and on-the-fly.

2.1 Overall representation

The simplest way to define a program trace is to treat it as a sequence of “instructions”. For Java programs, we view the trace as the sequence of executed bytecodes, instead of program statements. This is because only bytecodes are available for Java libraries, which are used by Java programs. Furthermore, collecting traces at the level of bytecode has the flexibility in tracing/not tracing certain bytecodes. For example, the `getstatic` bytecode loads the value of a static field. This bytecode does not need tracing, because which static field to access is decided at compile-time, and can be discovered from class files during post-mortem analysis.

However, representing a Java program trace as a bytecode sequence has its own share of problems. In particular, it does not allow us to capture many of the repetitions in the trace. Representation of the program trace as a single string loses out structure in several ways.

— The individual methods executed are not separated in the trace representation.

— Sequences of target addresses accessed by individual control transfer bytecodes are not separated out. These sequences capture control flow and exhibit high regularity (*e.g.* a loop branch repeats the same target many times).

— Similarly, sequences of addresses accessed by individual memory load/store bytecodes are not separated out. Again these sequences show fair amount of repetition (*e.g.* a read bytecode sweeping through an array).

In our representation, the compact trace of the whole program consists of trace tables, each of which is used for one method. Method invocations are captured by tracing bytecodes which invoke methods. The last executed bytecode w.r.t. the entire execution is clearly marked. Within the trace table for a method, each row maintains traces of a specific bytecode or of the exit of the method. Monitoring and tracing every bytecode may incur too much time and space overheads. We monitor only the following five kinds of bytecodes to collect the trace, where the first two are necessary to capture data flow of the execution, and the last three are necessary to capture control flow of the execution.

— *Memory allocation bytecodes.* Memory allocation bytecodes record the identities of created objects.

— *Memory access bytecodes.* The bytecodes to access local variables and static fields are not traced since the addresses accessed by these bytecodes can be obtained from the class file. For bytecodes accessing object fields / array elements, we trace the addresses (or identities since an address may be used by different variables in the lifetime of a program execution) corresponding to the bytecode operands.

— *Method invocation bytecodes.* Java programs use four kinds of bytecodes to invoke methods. Two of them, `invokevirtual` and `invokeinterface`, may invoke different methods on different execution instances. These invoked methods have the same method name and parameter descriptor (which can be discovered in class files), but they belong to different classes. So, for every `invokevirtual` and `invokeinterface` bytecode, we record the classes which the invoked methods belong to.

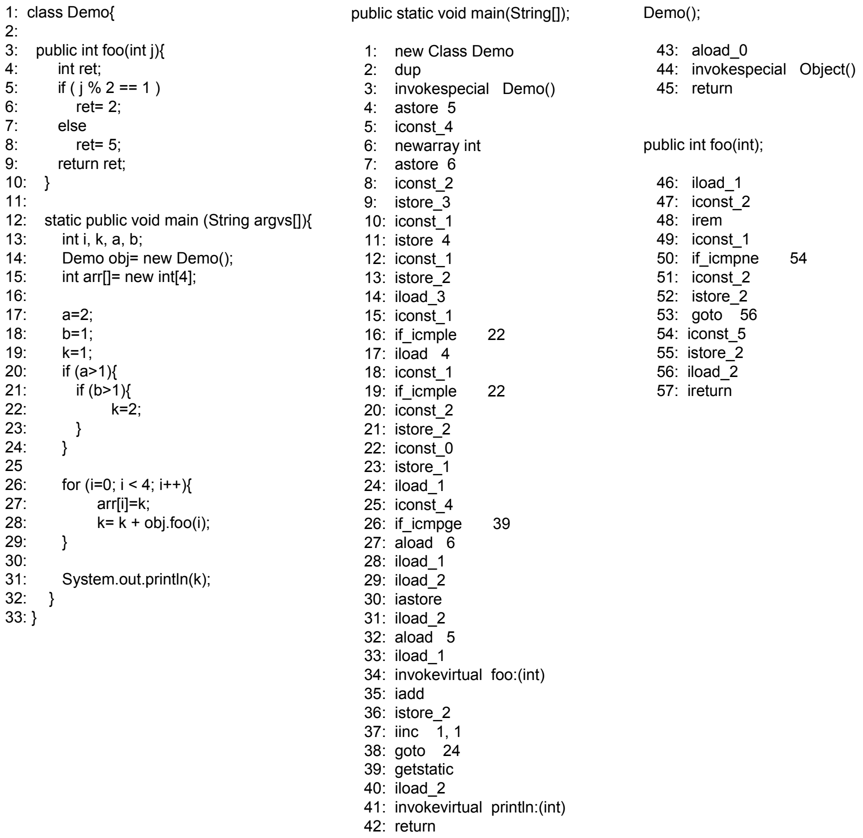
— *Bytecodes with multiple predecessors.* Some bytecodes have multiple predecessors in the control flow graph. For such a bytecode, we record which bytecodes are executed immediately before itself.

— *Method return bytecodes.* If a method has multiple `return` bytecodes, the trace of the method-exit records which `return` bytecodes are executed.

Monitoring the last two kinds of bytecodes (bytecodes with multiple predecessors and method return bytecodes) and marking the last executed bytecode are required due to backward traversal of the trace during post-mortem analysis. On the other hand, if slicing proceeds by forward traversal of the trace, it is not necessary to monitor bytecodes with multiple predecessors and method return bytecodes. Instead, for each conditional branch bytecode we can record which bytecodes are executed immediately after the branch bytecode (*i.e.*, the target addresses).

As mentioned earlier, our trace representation captures each method's execution in the trace as a trace table. Each row of the trace table for a method m represents the execution of one of the bytecodes of m (in fact it has to be a bytecode which we trace). A row of a trace table thus captures all execution instances of a specific bytecode. The row corresponding to a bytecode b in method m stores the sequence of values taken by each operand of b during execution; if b has multiple predecessors, we also maintain a sequence of the predecessor bytecode of b. Thus, in each row of a trace table we store several sequences in general; *these sequences are stored in a compressed format.* Separating the sequence of values for each bytecode operand allows a compression algorithm to capture and exploit regularity and repetition in the values taken by an operand. This can be due to regularity of control or data flow (*e.g.*, a read bytecode sweeping through an array or a loop iterating many times). Before presenting how to compress trace sequences, let us look at an example to understand the trace table representation. Note that sequences are not compressed in this example for ease of understanding.

Example. The left part of Figure 1 presents a simple Java program, and the right part shows the corresponding bytecode stream. Table I shows the trace tables for methods `main` and `foo`, respectively. The constructor method `Demo` has no trace table, because no



```
1:  class Demo{
2:
3:    public int foo(int j){
4:         int ret;
5:         if ( j % 2 == 1 )
6:             ret= 2;
7:         else
8:             ret= 5;
9:         return ret;
10:   }
11:
12:   static public void main (String argvs[]){
13:        int i, k, a, b;
14:        Demo obj= new Demo();
15:        int arr[]= new int[4];
16:
17:        a=2;
18:        b=1;
19:        k=1;
20:        if (a>1){
21:            if (b>1){
22:                    k=2;
23:            }
24:        }
25
26:        for (i=0; i < 4; i++){
27:                arr[i]=k;
28:                k= k + obj.foo(i);
29:        }
30:
31:        System.out.println(k);
32:    }
33: }
```

```
public static void main(String[]);

   1:   new Class Demo
   2:   dup
   3:   invokespecial   Demo()
   4:   astore  5
   5:   iconst_4
   6:   newarray int
   7:   astore  6
   8:   iconst_2
   9:   istore_3
   10:  iconst_1
   11:  istore  4
   12:  iconst_1
   13:  istore_2
   14:  iload_3
   15:  iconst_1
   16:  if_icmple      22
   17:  iload   4
   18:  iconst_1
   19:  if_icmple      22
   20:  iconst_2
   21:  istore_2
   22:  iconst_0
   23:  istore_1
   24:  iload_1
   25:  iconst_4
   26:  if_icmpge      39
   27:  aload   6
   28:  iload_1
   29:  iload_2
   30:  iastore
   31:  iload_2
   32:  aload   5
   33:  iload_1
   34:  invokevirtual  foo:(int)
   35:  iadd
   36:  istore_2
   37:  iinc    1, 1
   38:  goto    24
   39:  getstatic
   40:  iload_2
   41:  invokevirtual  println:(int)
   42:  return
```

```
Demo();

   43:  aload_0
   44:  invokespecial   Object()
   45:  return

public int foo(int);

   46:  iload_1
   47:  iconst_2
   48:  irem
   49:  iconst_1
   50:  if_icmpne      54
   51:  iconst_2
   52:  istore_2
   53:  goto    56
   54: iconst_5
   55: istore_2
   56: iload_2
   57: ireturn
```

Fig. 1. The left part is a simple Java program, and the right part shows corresponding bytecodes. Method `Demo()` is generated automatically as the class constructor.

bytecode of this method is traced. Each row in the trace table consists of: (a) the id/address for a bytecode (in the $Bytecode$ column), and (b) collected traces for that bytecode (in the $Sequences$ column).

For our example Java program, there are 57 bytecodes altogether, and only 8 of them are traced, as shown in Table I. Bytecodes 1 and 6 (*i.e.* two `new` statements at lines 14 and 15 of the source program) allocate memory for objects, and their traces include o_1 and o_2, which represent identities of the objects allocated by these bytecodes. Bytecode 30 defines an element of an array (*i.e.* define `arr[i]` at line 27 of the source program). Note that for this `iastore` bytecode, two sequences are stored. These sequences correspond to the two operands of the bytecode, namely: identities of accessed array objects (*i.e.* $\langle o_2, o_2, o_2, o_2 \rangle$) and indices of accessed array element (*i.e.* $\langle 0, 1, 2, 3 \rangle$). Both sequences consist of four elements, because bytecode 30 is executed four times and accesses $o_2[0], o_2[1], o_2[2], o_2[3]$ respectively; each element in a sequence records one operand for one execution of bytecode 30. Bytecodes 34 and 41 invoke virtual methods; the operand sequences record classes which invoked methods belong to, where C_{Demo} represents class `Demo` and C_{out} represents the standard output stream class. Bytecodes 22, 24 and 56 have

Bytecode	Sequences
1	$\langle o_1 \rangle$
6	$\langle o_2 \rangle$
22	$\langle 19 \rangle$
24	$\langle 23, 38, 38, 38, 38 \rangle$
30	$\langle o_2, o_2, o_2, o_2 \rangle$ $\langle 0, 1, 2, 3 \rangle$
34	$\langle C_{Demo}, C_{Demo}, C_{Demo}, C_{Demo} \rangle$
41	$\langle C_{out} \rangle$

(a)

Bytecode	Sequences
56	$\langle 55, 53, 55, 53 \rangle$

(b)

Table I. Trace table for (a) method `main(String[])` and (b) method `foo(int)` of Figure 1

multiple predecessors in the control flow graph. For example, bytecode 56 (*i.e.* `return ret` at line 9 of the source program) has two predecessors: bytecode 53 (*i.e.* after `ret=2` at line 6 of the source program) and bytecode 55 (*i.e.* `ret=5` at line 8 of the source program). The sequence recorded for bytecode 56 (see Table I(b)) captures bytecodes executed immediately before bytecode 56, which consists of bytecodes 55 and 53 in this example. Note that every method in our example program has only one `return` bytecode, so no `return` bytecode is monitored and no trace of the method-exit is collected.

Clearly, different invocations of a method within a program execution can result in different traces. The difference in two executions of a method results from different operands of bytecodes within the method. These different traces are all stored implicitly via the sequence of operands used by the traced bytecodes. As an example, consider the trace table of method `foo` shown in Table I(b). The different traces of `foo` result from the different outcomes of its only conditional branch, which is captured by the trace sequence for predecessors of bytecode 56 in Figure 1, as shown in Table I(b).

2.2 Overview of SEQUITUR

So far, we have described how the bytecode trace sequences representing control flow, data flow, or dynamic call graph are separated in an execution trace. We now employ a lossless compression scheme to exploit the regularity and repetition of these sequences. Our technique is an extension of the SEQUITUR, a lossless data compression algorithm [Nevill-Manning and Witten 1997] which has been used to represent control flow information in program traces [Larus 1999]. First we briefly describe SEQUITUR.

The SEQUITUR algorithm represents a finite sequence σ as a context free grammar whose language is the singleton set $\{\sigma\}$. It reads symbols one-by-one from the input sequence and restructures the rules of the grammar to maintain the following invariants: (A) no pair of adjacent symbols appear more than once in the grammar, and (B) every rule (except the rule defining the start symbol) is used more than once. To intuitively understand the algorithm, we briefly describe how it works on a sequence 123123. As usual, we use capital letters to denote non-terminal symbols.

After reading the first four symbols of the sequence 123123, the grammar consists of the single production rule

$$S \rightarrow 1, 2, 3, 1$$

where S is the start symbol. On reading the fifth symbol, it becomes

$$S \rightarrow 1, 2, 3, 1, 2$$

Since the adjacent symbols $1, 2$ appear twice in this rule (violating the first invariant), SEQUITUR introduces a non-terminal A to get

$$S \rightarrow A, 3, A \qquad A \rightarrow 1, 2$$

Note that here the rule defining non-terminal A is used twice. Finally, on reading the last symbol of the sequence 123123 the above grammar becomes

$$S \rightarrow A, 3, A, 3 \qquad A \rightarrow 1, 2$$

This grammar needs to be restructured since the symbols $A, 3$ appear twice. SEQUITUR introduces another non-terminal to solve the problem. We get the rules

$$S \rightarrow B, B \qquad B \rightarrow A, 3 \qquad A \rightarrow 1, 2$$

However, now the rule defining non-terminal A is used only once. So, this rule is eliminated to produce the final result.

$$S \rightarrow B, B \qquad B \rightarrow 1, 2, 3$$

Note that the above grammar accepts only the sequence 123123.

2.3 Capturing Contiguous Repeated Symbols in SEQUITUR

One drawback of SEQUITUR is that it cannot efficiently represent contiguous repeated symbols, including both terminal and non-terminal symbols. However, contiguous repeated symbols are not uncommon in program traces. Consider the example in Figure 1. Bytecode 24 (*i.e.* `i<4` at line 26 of the source program in Figure 1) has two predecessors: bytecode 23 (*i.e.* `i=0` at line 26 of the source program in Figure 1) and bytecode 38 (after `i++` at line 26 of the source program in Figure 1). The `for` loop is iterated four times, so the predecessor sequence for bytecode 24 is: ⟨ 23, 38, 38, 38, 38 ⟩ as shown in Table I(a). To represent this sequence, SEQUITUR will produce the following rules:

$$S \rightarrow 23, A, A \qquad A \rightarrow 38, 38$$

In general, if the `for` loop is iterated k times, SEQUITUR needs $O(lgk)$ rules in this fashion. To exploit such contiguous occurrences in the sequence representation, we propose the Run-Length Encoded SEQUITUR (RLESe).

RLESe constructs a context free grammar to represent a sequence *on the fly*; this contrasts with the work of [Reiss and Renieris 2001] which modifies the SEQUITUR grammar post-mortem. The right side of each rule is a sequence of "**nodes**". Each node $\langle sym : n \rangle$ consists of a symbol sym and a counter n (*i.e.* run length), representing n contiguous occurrences of sym. RLESe can exploit contiguous repeated symbols, and represent the above trace sequence ⟨ 23, 38, 38, 38, 38 ⟩ of bytecode 24 using the following one rule:

$$S \rightarrow 23 : 1, 38 : 4$$

The RLESe algorithm constructs a context free grammar by reading from the input sequence symbol by symbol. On reading a symbol sym, a node $\langle sym : 1\rangle$ is appended to the end of the start rule, and grammar rules are re-structured by preserving following three properties. The first property is unique to RLESe, resulting from its maintenance of contiguous occurrences of grammar nodes. The last two properties are taken (and modified) from SEQUITUR.

(1) *No contiguous repeated symbols property.* This property states that each pair of adjacent nodes contains different symbols. Continuous repeated symbols will be encoded within the run-length.

(2) *Digram uniqueness property.* This property means that no *similar* digrams appear in resulting grammar rules. Here a digram refers to two consecutive nodes on the right side of a grammar rule. Two digrams are *similar* if their nodes contain the same pair of symbols *e.g.* $\langle a : 2, X : 2\rangle$ is similar to $\langle a : 3, X : 4\rangle$, but $\langle a : 3, X : 2\rangle$ is not similar to $\langle X : 2, a : 3\rangle$.

(3) *Rule utility property.* This rule states that every rule (except the start rule S) is referenced more than once. When a rule is referenced by only one node and the run length n of that node equals 1, the reference will be replaced with the right hand side of this rule.

To maintain the digram uniqueness property in RLESe, we might need to split nodes during grammar construction. This split operation allows the algorithm to obtain duplicated *identical* digrams, and represent them by one grammar rule for potential space saving. Two digrams are *identical* if they have the same pairs of symbols and counters. For example, digram $\langle a : 2, X : 2\rangle$ is identical to $\langle a : 2, X : 2\rangle$, but digram $\langle a : 2, X : 2\rangle$ is not identical to $\langle a : 3, X : 4\rangle$.

Given two similar digrams $\langle sym_1 : n_1, sym_2 : n_2\rangle$, and $\langle sym_1 : n'_1, sym_2 : n'_2\rangle$, we can split at most two nodes to obtain identical digrams as $\langle sym_1 : min(n_1, n'_1), sym_2 : min(n_2, n'_2)\rangle$, where $min(n_1, n'_1)$ denotes the minimum of n_1 and n'_1. Consider bytecode 24 of Figure 1, which corresponds to the termination condition `i<4` of loop at line 26 of the source program. Assume that in some execution, such a loop is executed twice, one time with 6 iterations, and another time with 8 iteration. Recall that bytecode 24 has two predecessors: bytecode 23 (corresponding to `i=0` of the source program in Figure 1) and bytecode 38 (after `i++` of the source program in Figure 1). The predecessor sequence for bytecode 24 is:

$$S \rightarrow 23 : 1, 38 : 6, 23 : 1, 38 : 8$$

To ensure digram uniqueness property, we will split the node $\langle 38 : 8\rangle$ to a digram $\langle 38 : 6, 38 : 2\rangle$. This is to remove duplicate occurrences of similar digrams as:

$$S \rightarrow A : 2, 38 : 2 \qquad A \rightarrow 23 : 1, 38 : 6$$

The split operation introduces more nodes (at most two) into the grammar, but may save space when the identical digram appears frequently in the sequence being compressed.

In addition to the run-length encoding performed in RLESe, we also need to modify the terminal symbols fed into RLESe algorithm. In particular, we need to employ difference representations in memory reference sequences. For example, the sequence ⟨0, 1, 2, 3⟩ in Table I(a), which represents the indices of the array elements defined by bytecode 30 in Figure 1, cannot be compressed. By converting it into its difference representation as ⟨0, 1, 1, 1⟩, RLESe can compactly represent the trace sequence with one rule, as

$$S \rightarrow 0 : 1, 1 : 3$$

As with SEQUITUR [Nevill-Manning and Witten 1997], the RLESe compression algorithm is linear in both space and time, assuming that it takes constant time to find similar digrams. *Detailed space/time complexity analysis of the RLESe compression scheme is* *presented in Appendix A.* Experiments comparing RLESe with SEQUITUR (refer Section 5) show that RLESe can often achieve competitive compression ratio in less time. This is because RLESe can postpone re-constructing the grammar so the grammar rules are re-constructed less frequently. That is, on reading a symbol sym from input, instead of appending node $\langle sym : 1\rangle$ and re-constructing the grammar immediately, the RLESe algorithm first compares sym against the last node $\langle sym' : n\rangle$ of the start rule. If sym is the same as sym', the node $\langle sym' : n\rangle$ is updated to $\langle sym' : n + 1\rangle$ and the grammar is not further re-constructed. If not, node $\langle sym : 1\rangle$ is appended to the end of the start rule, and the grammar is re-structured so as to preserve the three properties of RLESe.

3. TECHNIQUES FOR DYNAMIC SLICING

In this section, we focus on how to perform dynamic slicing of Java programs. Our dynamic slicing algorithm operates on the compact bytecode traces described in the last section. Dynamic slicing is performed w.r.t. a slicing criterion (H, α, V), where H is an execution trace, α represents some bytecodes the programmer is interested in, and V is a set of variables referenced at these bytecodes. The dynamic slice contains all bytecodes which have affected values of variables in V referenced at last occurrences of α in the execution trace H.

Often, the user understands a Java program at the statement level. Thus, the user-defined criterion is often of the form (I, l, V), where I is an input, and l is a line number of the source program; the user is interested in statements (instead of bytecodes) which have affected values of variables in V referenced at last occurrences of statements at l during the execution with input I. This form is a little different from our bytecode based slicing criterion (H, α, V) and dynamic slice. In this case, program execution with input I produces the trace H, and α represents the bytecodes corresponding to statements at l. The user is interested in statements corresponding to bytecodes included in the dynamic slice. *In order to map bytecodes to a line number of the source file and vice versa, we use the* LineNumberTable *attribute in a Java's class file [Lindholm and Yellin 1999] which describes such a map.*

The dynamic slice includes the closure of dynamic control and data dependencies from the slicing criterion. Formally, a dynamic slice can be defined over *Dynamic Dependence Graph* (DDG) [Agrawal and Horgan 1990]. The DDG captures dynamic control and data dependencies between bytecode occurrences during program execution. Each node of the DDG represents one particular occurrence of a bytecode; edges represent dynamic data and control dependencies. The dynamic slice is then defined as follows.

Definition 3.0.1. **Dynamic slice** for a slicing criterion consists of all bytecodes whose occurrence nodes can be reached from the node(s) representing the slicing criterion in the DDG.

We can construct the DDG as well as the dynamic slice during a backwards traversal of the execution trace.

3.1 Core Algorithm

Figure 2 presents an inter-procedural dynamic slicing algorithm, which returns dynamic slice defined in Definition 3.0.1. Before slicing, we pre-compute the *control flow graph* for the program. In addition, we pre-compute the *control dependence graph* [Ferrante et al. 1987], where each node in the graph represents one bytecode, and an edge from node v

to v' represents that bytecode of v' decides whether bytecode of v will be executed. This static control dependence graph is used at lines 22 and 23 of the algorithm in Figure 2 to detect dynamic control dependencies.

Lines 1-5 of Figure 2 introduce five global variables for the slicing algorithm, including the slicing criterion. During dynamic slicing, we maintain δ, a list of variables whose values need to be explained, φ, the set of bytecode occurrences which have affected the slicing criterion, op_stack, a operand stack for simulation (see Section 3.3), and $fram$, a stack of frames for method invocations. The dynamic slice includes all bytecodes whose occurrences appear in φ at the end of the algorithm. For every method invocation during trace collection, we create a frame for this invocation during slicing. Each frame contains the method name and a γ set; the γ set includes bytecode occurrences β, where (a) β belongs to this method invocation, and (2) the dynamic control dependencies w.r.t. β need to be explained.

Initially we will set δ, φ, op_stack, and $fram$ to empty. However, if the program had aborted in the middle of an execution, the call stack $fram$ is initialized differently. In this case, our tracing will record the call stack at the point of abort, call it stk_{abort}. Our dynamic slicing algorithm then initializes $fram$ to stk_{abort} and proceeds by backward traversal of the execution trace.

Our slicing algorithm traverses the program's execution trace backwards, starting from the last executed bytecode recorded in the trace H. For each occurrence β of bytecode b_β (*i.e.* β represents one execution of the bytecode b_β) encountered during the backward traversal for slicing, a frame is created and pushed to $fram$ whenever b_β is a return bytecode (lines 9-12 of Figure 2). The γ set of the new frame is initialized to empty (line 11 of Figure 2), since no bytecode occurrence for this method invocation has been traversed. If b_β is a method invocation bytecode, a frame is popped from $fram$ (lines 13-16 of Figure 2). The dynamic slicing algorithm then checks whether the encountered bytecode occurrence β has affected the slicing criterion during trace collection, at lines 19-31 of Figure 2. In particular, line 19 of Figure 2 checks if β is the slicing criterion. Line 22 of Figure 2 checks dynamic control dependencies when b_β is a control transfer bytecode. The method `computeControlDependence`(b_β, $curr\_fram$, $last\_fram$) returns true iff. any bytecode occurrence included in the dynamic slicing is dynamically control dependent on β, where $curr\_fram$ is the top of the stack $fram$, and $last\_fram$ is the frame popped from $fram$ when b_β is a method invocation bytecode. More specifically, the `computeControlDependence` method returns true iff.

—b_β is a conditional branch bytecode, and some bytecode whose occurrence appears in $curr\_fram.\gamma$ is statically control dependent on b_β (intra-procedural control dependence check), or

—b_β is method invocation bytecode, and the $last\_fram.\gamma$ set is not empty, that is some of the bytecode occurrences in the method body are included in the slice (inter-procedural control dependence check).

Bytecode occurrences which are dynamically control dependent on β are then removed from $curr\_fram.\gamma$ at line 24 of Figure 2, because their dynamic control dependencies have just been explained. Line 27 of Figure 2 checks dynamic data dependencies. When the algorithm finds that the bytecode occurrence β has affected the slicing criterion (*i.e.* any of the three checks in lines 19, 22 and 27 of the algorithm in Figure 2 succeeds), β is included into $curr\_fram.\gamma$ and used variables are included into δ (at lines 32-34

```
1  $(H, \alpha, V)$= the slicing criterion
2  $\delta$= $\emptyset$, a set of variables whose values need to be explained
3  $\varphi$= $\emptyset$, the set of bytecode occurrences which have affected the slicing criterion
4  $op\_stack$= empty, the operand stack for simulation
5  $fram$= empty, the frames of the program execution

6  **dynamicSlicing**()
7      $b_\beta$ = get last executed bytecode from $H$;
8      **while** ($b_\beta$ is defined)
9          **if** ($b_\beta$ is a return bytecode)
10             $new\_fram$= $createFrame()$;
11             $new\_fram.\gamma$= $\emptyset$;
12             $push(fram, new\_fram)$;
13         **if** ($b_\beta$ is a method invocation bytecode)
14             $last\_fram = pop(fram)$;
15         **else**
16             $last\_fram = null$;
17         $\beta$ = current occurrence of bytecode $b_\beta$;
18         $curr\_fram$ = the top of $fram$;
19         **if** ($\beta$ is the last occurrence of $b_\beta$ and $b_\beta \in \alpha$)
20             $use\_vars$ = $V \cap$ variables used at $\beta$;
21             $\varphi = \varphi \cup \{\beta\}$;
22         **if** ($computeControlDependence(b_\beta, curr\_fram, last\_fram)$ )
23             $BC$= all bytecode occurrences in $curr\_fram.\gamma$ which are dynamically control dependent on $\beta$;
24             $curr\_fram.\gamma = curr\_fram.\gamma - BC$ ;
25             $use\_vars$ = variables used at $\beta$;
26             $\varphi = \varphi \cup \{\beta\}$;
27         **if** ($computeDataDependence(\beta, b_\beta)$)
28             $def\_vars$ = variables defined at $\beta$;
29             $\delta = \delta - def\_vars$;
30             $use\_vars$ = variables used at $\beta$;
31             $\varphi = \varphi \cup \{\beta\}$;
32         **if** ($\beta \in \varphi$)
33             $curr\_fram.\gamma = curr\_fram.\gamma \cup \{\beta\}$;
34             $\delta = \delta \cup use\_vars$;
35         $updateOpStack(\beta, b_\beta)$;
36         $b_\beta$ = $getPrevBytecode(\beta, b_\beta)$;
37     **return** bytecodes whose occurrences appear in $\varphi$;
```

Fig. 2. The dynamic slicing algorithm

of Figure 2), in order to find bytecode occurrences which have affected β and hence the slicing criterion. The simulation operand stack op_stack is also properly updated at line 35 for further check of data dependencies (this is explained in Section 3.3). The dynamic control and data dependencies checks (lines 22 and 27 of Figure 2) can be reordered, since they are independent of each other.

In the rest of this section, we elaborate on the underlying subtle issues in using the slicing framework of Figure 2. Section 3.2 presents how to traverse the execution backwards without decompressing the compact bytecode trace. Section 3.3 explains the intricacies of our dynamic data dependence computation in presence of Java's stack based execution. Section 3.4 illustrates the dynamic slicing algorithm with an example and Section 3.5 shows the correctness and cost of our dynamic slicing algorithm.

```
1  getPrevBytecode ($\beta$: bytecode occurrence, $b_\beta$: bytecode)
2      if ($b_\beta$ has exactly one predecessor in the control flow graph)
3          $b_{last}$= the predecessor bytecode;
4      else
5          $G$= compressed control flow operand sequence for $b_\beta$ in the compact bytecode trace $H$
6          $\pi$= a root-to-leaf path for $G$;
7          $b_{last}$= $getLast(G, \pi)$;
8      if ($b_{last}$ is a method invocation bytecode)
9          $meth$= the method invoked by $\beta$;
10         if ($meth$ has exactly one return bytecode)
11             return the return bytecode;
12         else
13             $G'$= compressed operand sequence for exit of $meth$ in trace $H$
14             $\pi'$= a root-to-leaf path for $G'$;
15             return $getLast(G', \pi')$;
16     if ($b_{last}$ represents the start of a method)
17         return the bytecode which invokes current method;
18     return $b_{last}$;
```

Fig. 3. Get the previous executed bytecode during backward traversal of the execution trace.

3.2 Backward Traversal of Trace without decompression

The dynamic slicing algorithm in Figure 2 traverses the program execution backwards, starting from the last executed bytecode recorded in the trace H (line 7 of Figure 2). The algorithm proceeds by iteratively invoking the `getPrevBytecode` method to obtain the bytecode executed prior to current occurrence β of bytecode b_β during trace collection. Figure 3 presents the `getPrevBytecode` method. The method first retrieves the last executed bytecode within the same method invocation of b_β to b_{last}, and returns b_{last} if b_{last} does not cross method boundaries (lines 2-7 of Figure 3). If b_{last} invokes a method $meth$, the last executed return bytecode of method $meth$ is returned (lines 8-15 of Figure 3). If b_{last} represents the start of a method, the bytecode which invokes current method is returned (lines 16-17 of Figure 3).

The `getPrevBytecode` method has to retrieve last executed bytecode from the compact bytecode trace H. For this purpose, it needs to traverse the predecessor sequence of a bytecode with multiple predecessors. Since such sequences are compactly stored as RLESe grammars, we need to efficiently traverse RLESe grammars; this is accomplished by the method `getLast`. The `getLast` method gets last executed predecessor from a RLESe grammar G without decompression, using a root-to-leaf path π in G. The slicing algorithm maintains such a path π for each compressed RLESe sequence G to clearly mark which portion of G has been already visited. We now explain in details the mechanics of efficient traversal over RLESe representation.

In our trace compression scheme, all operand sequences are compressed using RLESe. The dynamic slicing algorithm traverses these sequences from the end to extract predecessors for computation of control flow, and to extract identifies of accessed variables for computation of data flow. For example, consider the operand sequence of array indices for bytecode 30 in Table I(a), which is $\langle 0, 1, 2, 3\rangle$. During dynamic slicing on the program of Figure 1, we traverse this sequence backwards, that is, from the end. At any point during the traversal, we mark the last visited operand (say $\langle 0, 1, \overline{2}, 3\rangle$) during slicing. The sequence beginning with the marked operand (*i.e.* $\langle\overline{2}, 3\rangle$) has been visited. When the slicing algorithm tries to extract next operand from the operand sequence, we use this mark to find last unvisited element (*i.e.* value 1 in this example). We now describe how such markers can be maintained and updated in the RLESe grammar representation.

The RLESe grammar of a sequence σ can simply be represented as a directed acyclic graph (DAG). The RLESe grammar consists of run-length annotated symbols of the form $\langle sym : n\rangle$ where sym is a terminal or non-terminal symbol and n denotes a run-length, representing n contiguous occurrences of symbol sym. Let us consider an example where the operand sequence is $\langle abbabbcabb\rangle$. The RLESe compression algorithm will produce the following rules to represent this operand sequence:

$$S \rightarrow A : 2, c : 1, A : 1 \qquad A \rightarrow a : 1, b : 2$$

Small letters denote terminal symbols in the operand sequence, and capital letters denote non-terminal symbols. Figure 4(a) shows corresponding DAG representation, where dashed circles represent non-terminal symbols, circles represent terminal symbols, and edges are annotated by run-lengths. For example, the run-length annotated symbol $\langle A : 1\rangle$ at the end of the start rule is captured by the node A and the incoming edge $S \xrightarrow{1} A$ for node A. We then use a root-to-leaf path π over the DAG representation to mark the symbol in σ that was last visited during backward traversal. For every edge of the path π, we maintain both the run length n of corresponding grammar node $X = \langle sym : n\rangle$, and an visitation counter $k \leq n$, where k denotes the number of times that node X has been visited so far. For example, in the edge $A \overset{2,1}{\rightarrow} b$ in Figure 4(c), 2 represents the run length of grammar node $\langle b : 2\rangle$, and 1 represents that this node has been visited once.

The dynamic slicing algorithm maintains one root-to-leaf path π for every compressed operand sequence. The path π is initialized from the root to the rightmost leaf node in the DAG, and the visitation counter annotated for the last edge in π is set to 0, since no symbol has been visited. The slicing algorithm then uses the path π to find the last unvisited terminal symbol of the RLESe grammar by invoking the `getLast` method of Figure 5. In the `getLast` method, G is the DAG representation of the RLESe grammar for a sequence σ and π is the root-to-leaf path for the symbol in σ that was last visited. The `getLast` method returns the last unvisited symbol and updates the path π. Note that the “immediate left sibling” of an edge $e = x \rightarrow y$ is the edge $e' = x \rightarrow z$ where node z is the immediate left sibling of node y in the graph; this notion is used in lines 10 and 11 of Figure 5.

Figure 4(b) shows the initialized root-to-leaf path π for the RLESe grammar. Figure 4(c-g) present the resultant path π by calling the `getLast` method of Figure 5 each time, and the symbol of the leaf node pointed by the path π is returned as the last unvisited symbol. For example, Figure 4(c) shows the path π after the first calling the `getLast` method. The path π includes two edges (*i.e.* $S \overset{1,1}{\rightarrow} A$ and $A \overset{2,1}{\rightarrow} b$), representing both edges have been visited once. The leaf node b is referenced by π. Thus, b is returned by the `getLast` method, which represents the last symbol b in the original sequence $\langle abbabbcabb\rangle$.

With the `getLast` algorithm in Figure 5, we can extract an operand sequence efficiently. Given the grammar for a sequence with length N, the `getLast` method will be invoked N times to extract the entire sequence. The overall space overhead is $O(N)$, and the overall time overhead is $O(N)$. The space overhead is caused by maintaining the root-to-leaf path π, which is used by all invocations of the `getLast` method to extract a sequence. The length of path π is linear in the number of grammar rules (the grammar has

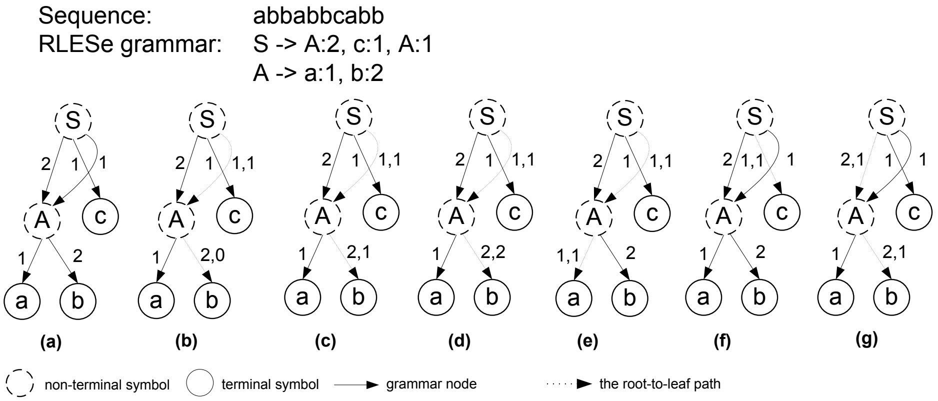


Fig. 4. An example shows how to extract operand sequence over RLESe representation without decompression

```
getLast($G$: Grammar, $\pi$: path in $G$)
    $e$ = the last edge of $\pi$;
    while ($e$ is defined)
        let $e$ = $sym_1 \overset{n_1,k_1}{\longrightarrow} sym'_1$;
        if ( $k_1 < n_1$)
            break;
        else
            remove edge $e$ from $\pi$;
            change the annotation of edge $e$ from $(n_1, k_1)$ to $(n_1)$;
            $Sib_e$= immediate left sibling of $e$ in $G$;
            if ( such a sibling $Sib_e$ exists)
                $e = Sib_e$;
                break;
            else
                $e$ = last edge of $\pi$;
    let $e$ = $sym_2 \overset{n_2,k_2}{\longrightarrow} sym'_2$;
    change the annotation of edge $e$ from $(n_2, k_2)$ to $(n_2, k_2 + 1)$;
    $G_X$ = DAG rooted at node $sym'_2$ within $G$;
    for (each edge $e'$ from node $sym'_2$ to rightmost leaf node of $G_X$)
        insert edge $e'$ into $\pi$;
        let $e'$ = $sym_3 \overset{n_3}{\rightarrow} sym'_3$;
        change the annotation of edge $e'$ from $(n_3)$ to $(n_3, 1)$;
    return symbol in rightmost leaf node of $G_X$;
```

Fig. 5. One step in the backward traversal of a RLESe sequence (represented as DAG) without decompressing the sequence.

no recursive rules). There are fewer grammar rules than grammar nodes, and the number of grammar nodes is bound by the length of the original sequence. Thus, the space overhead to extract the entire sequence is $O(N)$.

The time overhead to extract the entire sequence comes from the time to access edges and nodes in the DAG. Whenever a node with terminal symbol is accessed, the `getLast` method immediately returns the terminal symbol. So, the total number of times to access node with terminal symbol is $O(N)$ in order to extract a sequence with length N. The total number of accesses to non-terminal nodes is $O(\sum_i \frac{N}{2^i})$= $O(N)$. Consequently, the time overhead to extract the entire sequence from a RLESe grammar (by repeatedly invoking `getLast` to get the last symbol which has not been visited) is $O(N)$.

3.3 Computing Data Dependencies

The typical way to detect dynamic data dependencies is to compare addresses of variables defined/used by bytecode occurrences (line 27 of Figure 2). However, this is complicated by Java's stack based architecture. During execution, the Java virtual machine uses an operand stack to hold partial results of execution. Thus, dynamic data dependence exists between bytecode occurrences β and β', when a value is pushed into the operand stack by β and is popped by β'. Consider the program in Figure 1 as an example. Assume that statement 27 of the source program is executed, and the corresponding trace at the level of bytecode is $\langle 27^1$, 28^2, 29^3, $30^4\rangle$ where 27^1 means that the first element of the trace is bytecode 27 and so on. Bytecode occurrence 30^4 (which defines the array element `arr[i]` at line 27 of the source program) is dynamically data dependent on bytecode occurrences 27^1, 28^2 and 29^3 (which load local variables `k` and `i`, array object reference `arr` at line 27 of the source program, respectively). The three bytecode occurrences push three values into the operand stack, all of which are popped and used by bytecode occurrence 30^4.

Clearly, dynamic data dependencies w.r.t. local variables and fields can be easily detected by comparing the addresses (or identities) of accessed variables. However, detecting data dependencies w.r.t. the operand stack requires *reverse* simulating the operand stack (since we traverse the trace backwards). Figure 6 presents how to maintain the stack op_stack for simulation, which is used by the dynamic slicing algorithm at line 35 of Figure 2. We pop the simulation stack for defined operands, and push used operands into the simulation stack. The function $def\_op(b_\beta)$ ($use\_op(b_\beta)$) at line 2 (4) of Figure 6 returns the number of operands defined (used) by bytecode b_β. Note that the stack simulated during slicing does not contain actual values of computation. Instead, each entry of the stack stores the bytecode occurrence which pushed the entry into the stack.

Figure 7 shows the method to determine whether a bytecode occurrence has affected the slicing criterion via dynamic data dependencies. This method is used by the dynamic slicing algorithm at line 27 of Figure 2. If a bytecode occurrence β defines a variable which needs explanation (lines 2-10 of Figure 7), or β defines a partial result which needs explanation (lines 11-13 of Figure 7), the method returns *true* to indicate that β has affected the slicing criterion. A partial result needs explanation if the bytecode occurrence which pushes corresponding entry into the stack has already been included in φ. The function $def\_op(b_\beta)$ at line 11 of Figure 7 returns the number of operands defined by bytecode b_β. Our dynamic slicing algorithm needs the addresses of variables accessed by each bytecode occurrence for detecting data dependencies (lines 20, 25, 28 and 30 of Figure 2 and lines 3-8 of Figure 7). If a local variable or a static field is accessed, the address can be found from the class files. If an object field or an array element is accessed, the address can be

```
**updateOpStack** ($\beta$: bytecode occurrence, $b_\beta$: bytecode)
    **for** ($i = 0; i < def\_op(b_\beta); i = i + 1$)
        $pop(op\_stack)$;
    **for** ($i = 0; i < use\_op(b_\beta); i = i + 1$)
        $push(op\_stack, \beta)$;
```

Fig. 6. Maintain the simulation stack op_stack.

```
**computeDataDependence** ($\beta$: bytecode occurrence, $b_\beta$: bytecode)
    **if** ($\beta$ defines a variable)
        **if** ($\beta$ defines a static field or local variable)
            $def\_loc$= get address of the defined static field or local variable from class files;
        **if** ($\beta$ defines an object field or an array element)
            $G$= compressed operand sequence for $b_\beta$ in the compact bytecode trace $H$
            $\pi$= a root-to-leaf path for $G$;
            $def\_loc$= $getLast(G, \pi)$;
        **if** ( $def\_loc \in \delta$ )
            **return** true;
    $\omega$= the set of bytecode occurrences in top $def\_op(b_\beta)$ entries of $op\_stack$;
    **if** ( $\omega \cap \varphi \neq \emptyset$)
        **return** true;
    **return** false;
```

Fig. 7. Detect dynamic data dependencies for dynamic slicing

found from operand sequences of corresponding bytecode in the compact bytecode trace.

When the algorithm detects data dependencies via reverse stack simulation, it requires analyzing some bytecodes whose operands are not traced. Consider the program in Figure 1 as an example. The statement `if (j%2==1)` at line 5 of the source program corresponds to bytecode sequence $\langle 46^1, 47^2, 48^3, 49^4, 50^5\rangle$. Figure 8 shows the op_stack after processing each bytecode occurrence during backward traversal. Note that the operands of bytecode 48 (*i.e.* bytecode `irem` which stands for the mathematical computation "%") are not traced. When the bytecode occurrence 48^3 is encountered, the `computeDataDependence` method in Figure 7 can detect that 50^5 is dynamically data dependent on 48^3. In addition, the bytecode 48 will also update the op_stack, as shown in Figure 8. As we can see from the example, in order to detect implicit data dependencies involving data transfer via the operand stack, it is important to know which bytecode occurrence pushes/pops an entry from the op_stack. The actual values computed by the bytecode execution are not important. This highlights difference between our method and the work on Abstract Execution [Larus 1990]. In Abstract Execution, a small set of events are recorded and these are used as guide to execute a modified program. In our work, we record some of the executed bytecodes in our compressed trace representation. However, the untraced bytecodes are not re-executed during the analysis of the compressed trace.

3.4 Example

Consider the example program in Figure 1, and corresponding compressed trace in Table I. Assume that the programmer wants to find which bytecodes have affected the value of

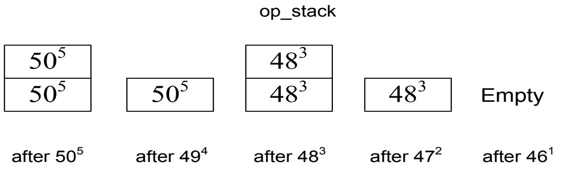


Fig. 8. An example shows the op_stack after each bytecode occurrence encountered during backward traversal

k at line 31 of the source program. Table II shows each stage using the dynamic slicing algorithm in Figure 2 w.r.t. the k. For simplicity, we do not illustrate slicing over the entire execution, but over last executed eight statements – $\langle 26,\ 27,\ 28,\ 5,\ 6,\ 9,\ 26,\ 31\rangle$. The corresponding bytecode sequence is a sequence of thirty-one bytecode occurrences shown in the first column of Table II. For each bytecode occurrence, the position number $1, \ldots, 31$ of the bytecode occurrence in the sequence is marked as superscript for the sake of clarity.

For each bytecode occurrence β encountered during backward traversal, one row of Table II shows resultant δ, $fram$, op_stack and φ after analyzing β by our dynamic slicing algorithm. The $\star$ in the last column indicates that the corresponding bytecode occurrence has affected the slicing criterion and is included into φ.

When bytecode occurrence 40^{30} is encountered, it is found to be the slicing criterion. The used variable k is inserted to δ to find which bytecode occurrence defines k; the bytecode occurrence 40^{30} is inserted to γ for control dependency check; we pop 41^{31} from the operand stack op_stack because 40^{30} loads one value to the operand stack during trace collection. It should be noted that in this simple example, we refer to a variable with its name (*e.g.* k), since both methods are invoked once, and every variable has a distinct name in this example. This is for simplicity of illustration. In the implementation, we use identifiers to distinguish between variables from same/different method invocation.

After 40^{30}, bytecode occurrence 39^{29} is encountered during backward traversal and so on. We omit the details of the entire traversal but highlight some representative bytecode occurrences.

—After analyzing bytecode occurrence 56^{20}, the slicing algorithm finds that bytecode 56 has two predecessors, and retrieves last unvisited value from operand sequence of bytecode 56 in Table I(b). Therefore, bytecode occurrence 53^{19} is next analyzed.

—When bytecode occurrence 30^7 is encountered, o_2 and 3 are retrieved from operand sequences of bytecode 30 in Table I(a), representing an assignment to array element $o_2[3]$. However, $o_2[3]$ is irrelevant to the slicing criterion, so neither δ nor γ is updated.

3.5 Proof of Correctness and Complexity Analysis

In this section we first prove the correctness of our dynamic slicing algorithm; details of the lemmas used in the proof can be found in Appendix B. Complexity analysis of the algorithm is then presented.

THEOREM 3.5.1. *Given a slicing criterion, the dynamic slicing algorithm in Figure 2 returns dynamic slice defined in Definition 3.0.1.*

β	δ	$fram$		op_stack	$\in \varphi$
		method	γ		
41^{31}	$\{\}$	$main$	$\{\}$	$\langle 41^{31}, 41^{31} \rangle$	
40^{30}	$\{k\}$	$main$	$\{40^{30}\}$	$\langle 41^{31} \rangle$	$\star$
39^{29}	$\{k\}$	$main$	$\{40^{30}\}$	$\langle\rangle$	
26^{28}	$\{k\}$	$main$	$\{40^{30}\}$	$\langle 26^{28}, 26^{28} \rangle$	
25^{27}	$\{k\}$	$main$	$\{40^{30}\}$	$\langle 26^{28} \rangle$	
24^{26}	$\{k\}$	$main$	$\{40^{30}\}$	$\langle\rangle$	
38^{25}	$\{k\}$	$main$	$\{40^{30}\}$	$\langle\rangle$	
37^{24}	$\{k\}$	$main$	$\{40^{30}\}$	$\langle\rangle$	
36^{23}	$\{\}$	$main$	$\{40^{30}, 36^{23}\}$	$\langle 36^{23} \rangle$	$\star$
35^{22}	$\{\}$	$main$	$\{40^{30}, 36^{23}\}$	$\langle 35^{22}, 35^{22} \rangle$	$\star$
57^{21}	$\{\}$	foo	$\{57^{21}\}$	$\langle 35^{22}, 57^{21} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
56^{20}	$\{ret\}$	foo	$\{57^{21}, 56^{20}\}$	$\langle 35^{22} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
53^{19}	$\{ret\}$	foo	$\{57^{21}, 56^{20}\}$	$\langle 35^{22} \rangle$	
		$main$	$\{40^{30}, 36^{23}\}$		
52^{18}	$\{\}$	foo	$\{57^{21}, 56^{20}, 52^{18}\}$	$\langle 35^{22}, 52^{18} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
51^{17}	$\{\}$	foo	$\{57^{21}, 56^{20}, 52^{18}, 51^{17}\}$	$\langle 35^{22} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
50^{16}	$\{\}$	foo	$\{57^{21}, 56^{20}, 50^{16}\}$	$\langle 35^{22}, 50^{16}, 50^{16} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
49^{15}	$\{\}$	foo	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}\}$	$\langle 35^{22}, 50^{16} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
48^{14}	$\{\}$	foo	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}, 48^{14}\}$	$\langle 35^{22}, 48^{14}, 48^{14} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
47^{13}	$\{\}$	foo	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}, 48^{14}, 47^{13}\}$	$\langle 35^{22}, 48^{14} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
46^{12}	$\{j\}$	foo	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}, 48^{14}, 47^{13}, 46^{12}\}$	$\langle 35^{22} \rangle$	$\star$
		$main$	$\{40^{30}, 36^{23}\}$		
34^{11}	$\{\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}\}$	$\langle 35^{22}, 34^{11}, 34^{11} \rangle$	$\star$
33^{10}	$\{i\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}\}$	$\langle 35^{22}, 34^{11} \rangle$	$\star$
32^{9}	$\{i, obj\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}\}$	$\langle 35^{22} \rangle$	$\star$
31^{8}	$\{i, obj, k\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle\rangle$	$\star$
30^{7}	$\{i, obj, k\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle 30^{7}, 30^{7}, 30^{7} \rangle$	
29^{6}	$\{i, obj, k\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle 30^{7}, 30^{7} \rangle$	
28^{5}	$\{i, obj, k\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle 30^{7} \rangle$	
27^{4}	$\{i, obj, k\}$	$main$	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle\rangle$	
26^{3}	$\{i, obj, k\}$	$main$	$\{40^{30}, 26^{3}\}$	$\langle 26^{3}, 26^{3} \rangle$	$\star$
25^{2}	$\{i, obj, k\}$	$main$	$\{40^{30}, 26^{3}, 25^{2}\}$	$\langle 26^{3} \rangle$	$\star$
24^{1}	$\{i, obj, k\}$	$main$	$\{40^{30}, 26^{3}, 25^{2}, 24^{1}\}$	$\langle\rangle$	$\star$

Table II. An example to show each stage of the dynamic slicing algorithm in Figure 2. The column β shows bytecode occurrences in the trace being analyzed.

PROOF. Let φ_i be the φ set after i loop iterations of the dynamic slicing algorithm in Figure 2, φ_* be the resultant φ set when the algorithm finishes, and β be the bytecode occurrence encountered at the ith loop iteration.

We first prove the soundness of the algorithm by induction on loop iterations of the slicing algorithm, *i.e.* for any $\beta' \in \varphi_*$ we show that β' is reachable from the slicing criterion in the dynamic dependence graph (DDG).

Base: Initially, $\varphi_0 = \emptyset$, so the result trivially holds.

Induction: By induction hypothesis, $\forall \beta'' \in \varphi_{i-1}$, β'' is reachable from the slicing criterion in the dynamic dependence graph (DDG). Note that $\varphi_i = \varphi_{i-1}$, or $\varphi_i = \varphi_{i-1} \cup \{\beta\}$. Then, $\forall \beta' \in \varphi_i$, (1) if $\beta' \in \varphi_{i-1}$, then β' is reachable from the slicing criterion in the DDG according to the induction hypothesis. (2) if $\beta' = \beta$, then $\beta \in \varphi_i - \varphi_{i-1}$. Either

β belongs to the slicing criterion, or $\exists\overline{\beta} \in \varphi_{i-1}$, where $\overline{\beta}$ is dynamically control/data dependent on β, according to Lemma B.0.5. Because $\overline{\beta}$ is reachable from the slicing criterion in the DDG according to the induction hypothesis, β is reachable from the slicing criterion in the DDG.

Next, we prove the completeness of the slicing algorithm, *i.e.* $\forall\beta'$ reachable from slicing criterion in the DDG $\Rightarrow \beta' \in \varphi_*$. Note that there is no cycle in the DDG, so we prove the completeness by induction on structure of the DDG.

Base: Consider a bytecode occurrence β' where β' belongs to the slicing criterion. Let β' be encountered at the ith loop iteration of the slicing algorithm. Thus $\beta' \in \varphi_i \subseteq \varphi_*$ (refer Lemma B.0.5 and B.0.1 in Appendix B for details).

Induction: Assume that a bytecode occurrence β' which is reachable from the slicing criterion in the DDG is included into φ_*. For every β'' where there is an edge from β' to β'' in the DDG, β' is dynamically control/data dependent on β''. Let β' be encountered at the ith loop iteration of the algorithm, so $\beta' \in \varphi_i$. Let β'' be encountered at the jth loop iteration of the algorithm. Because β'' appears earlier than β' during trace collection, backward travel of the trace will encounter β'' after β', *i.e.* $i < j$. Thus, $\beta' \in \varphi_i \subseteq \varphi_{j-1}$ according to and then $\beta'' \in \varphi_j - \varphi_{j-1}$ (refer Lemma B.0.5 and B.0.1 in Appendix B for details).

Consequently, $\beta'' \in \varphi_*$, since $\varphi_j - \varphi_{j-1} \subseteq \varphi_j \subseteq \varphi_*$.

□

Now we analyze the cost of the dynamic slicing algorithm in Figure 2. Given the compressed trace for an execution which executes N bytecodes, the space overhead of the slicing algorithm is $O(N)$, and the time overhead is $O(N^2)$.

The space overhead of the algorithm is caused by maintaining δ, φ, op_stack, $fram$, compressed operand sequences and root-to-leaf paths for every compressed sequence. The sizes of δ, φ, op_stack and $fram$ are all $O(N)$. For δ, this is because one execution of a bytecode can use a constant number of variables; for op_stack, this is because the number of operands popped from and pushed to the operand stack by one bytecode is bound by a constant; for $fram$, this is because the γ set of each method invocation in $fram$ only contains bytecode occurrences for this invocation, and does not overlap with each other. Assume that each bytecode b_i has executed $\eta(b_i)$ times, so $\sum_{b_i} \eta(b_i)= N$. The size of all compressed operand sequences is $\sum_{b_i} O(\eta(b_i))= O(N)$, because every bytecode has a fixed number of operand sequences, and the size of the compact representation is linear in the length of original operand sequence (proof is presented in Appendix A). The size of each root-to-leaf path is bound by the size of corresponding compressed operand sequence. Consequently, the overall space cost of the slicing algorithm is $O(N)$.

During dynamic slicing, the algorithm performs the following four actions for each occurrence β of bytecode b_β encountered during backward traversal of the execution trace.

(1) extract operand sequences of bytecode b_β from the compressed trace for backward traversal,

(2) perform slicing criterion, dynamic control/data dependency checks,

(3) update δ, φ, op_stack, and $fram$,

(4) get previous executed bytecode.

```
1   $b = 1;$
2   $k = 1;$
3   if ($a > 1$) {
4       if ($b > 1$){
5           $k = 2$
        }
    }
6   $\ldots = k$
```

Fig. 9. A "buggy" program fragment

According to the complexity analysis of the `getLast` method which is presented in Section 3.2, it needs $O(\eta(b_i))$ time to extract an operand sequence of bytecode b_i which is executed $\eta(b_i)$ times during trace collection. Note that $\sum_i \eta(b_i)$= N. Consequently, the overall time overhead to perform action (1) to extract all operand sequences is $\sum_i O(\eta(b_i))$= $O(N)$ (*i.e.*, linear in the length of the original execution. After extracting operand sequences, the overall time to perform action (2) and (3) is clearly $O(N^2)$, since they may update/enquire sets δ, φ, op_stack and $fram$ whose sizes are bound by N. This complexity can be improved further by using efficient data structures for set representation.

To get previous executed bytecode (action (4) in the preceding), the `getPrevBytecode` method of Figure 3 may perform two actions: (a) get the predecessor bytecode from class files, or (b) extract last executed bytecode from compressed operand sequences. The overall time to perform get predecessor bytecode from class files is $O(N)$, since it needs constant time to get the predecessor bytecode from Java class files every time. The overall time to extract operand sequences is $O(N)$ as discussed earlier.

Consequently, the overall time cost of the dynamic slicing algorithm is $O(N^2)$.

4. RELEVANT SLICING

In the previous section, we presented dynamic slicing which can be used for focusing the programmer's attention to a part of the program. However, there are certain difficulties in using dynamic slices for program debugging/understanding. In particular, dynamic slicing does not consider that the execution of certain statements is wrongly omitted. Consequently, all statements responsible for the error may not be included into the dynamic slice, and the slice may mislead the programmer.

As an example, consider the program fragment presented in Figure 9, which is a simplified version of the program in Figure 1. Let us assume that the statement `b=1` at line 1 should be `b=2` in a correct program. With input `a=2`, the variable `k` at line 6 is 1 unexpectedly. The execution trace is $\langle$ 1^1, 2^2, 3^3, 4^4, 6^5 $\rangle$, where 1^1 means statement 1 is executed as the *first* statement and so on. If we use dynamic slicing to correct the bug w.r.t. the incorrect value of `k` at 6^5, the dynamic slice computed by the slicing algorithm of Section 3 contains only lines 2 and 6. Changing line 2 may fix the bug, since such a change will change the value of `k` at line 6. Unfortunately, line 1, the actual bug, is excluded from the dynamic slice. In this example the error arises from the execution of line 5 being wrongly omitted, which is caused by the incorrect assignment at line 1. In fact, changing line 1 may cause the predicate at line 4 to be evaluated differently; then line 5 will be executed and the value of `k` at line 6 will be different. In other words, dynamic slicing does not consider the effect of the unexecuted statement at line 5.

The notion of *relevant slicing*, an extension of dynamic slicing, fills this caveat. Relevant slicing was introduced in [Agrawal et al. 1993; Gyimóthy et al. 1999]. Besides dynamic control and data dependencies, relevant slicing considers *potential dependencies* which capture the potential effects of unexecuted paths of branch and method invocation statements. The relevant slice include more statements which, if changed, may change the "wrong" behaviors w.r.t. the slicing criterion. In the example of Figure 9 with input a=2, 6^5 is *potentially dependent* on execution of the branch at line 4 (4^4), because if the predicate at 4^4 is evaluated differently, the variable k is re-defined and then used by 6^5. Considering that 4^4 is dynamically data dependent on 1^1, 1^1 has potential effects over 6^5. Thus, line 1 is included into the resultant relevant slice.

In general, *Dynamic Slice* $\subseteq$ *Relevant Slice* $\subseteq$ *Static Slice*. In our experiments (refer Section 5), we show that the sizes of the relevant slices are close to the sizes of the corresponding dynamic slices. Like dynamic slices, the relevant slices are also computed w.r.t. a particular program execution (*i.e.* it only includes executed statements).

The rest of this section is organized as follows. In Section 4.1, we recall *past work* on potential dependencies, a notion which is crucial for computing the relevant slice. We then present our definition of relevant slice. Since the existing works on relevant slicing present a slice as a set of program statements, we first define a relevant slice as a set of statements. However, like our dynamic slicing algorithm, our relevant slicing algorithm (refer Figure 11 in Section 4.2) operates at the level of bytecodes. This is discussed in Section 4.2.

4.1 Background

In this subsection, we recapitulate the definition of potential dependence. This portion is *not new* which was mostly studied in [Agrawal et al. 1993; Gyimóthy et al. 1999].

Relevant slicing extends dynamic slicing by considering statements which may affect the slicing criterion. These statements are executed and do not affect the criterion. However, if these statements are changed, branches may be evaluated differently, or alternative methods may be invoked. We consider the following two situations for the execution of alternative code fragment due to program changes.

(1) *Branch*: If the value of a variable used by the predicate of a branch statement is changed, the predicate may be evaluated differently.

(2) *Method invocation*: If the programmer changes the assignment w.r.t. the object which invokes a method, or the declaration of parameters, an alternative method may be called.

In relevant slicing, we use the notion of *potential dependence* to capture the effects of these branch and method invocation statements. If these statements are evaluated differently, variables may be re-defined and affect the slicing criterion. We define potential dependence as follows; for any statement occurrence β, the corresponding statement is written as S_β.

Definition 4.1.1. **Potential Dependence** Given an execution trace H of a program P, for any statement instances β and β' appearing in H, β is *potentially dependent* on an earlier branch or method invocation statement instance β' if and only if all of the following conditions hold.

(1) β is not (transitively) dynamically control/data dependent on β' in H.

(2) there exists a variable v used by β s.t. v is not defined between β' and β in H and v may be defined along an outgoing edge l of statement $S_{\beta'}$ where $S_{\beta'}$ is the statement at statement occurrence β'. That is, there exists a statement X satisfying the following.
 - (a) X is (transitively) control dependent on $S_{\beta'}$ along the edge l, and
 - (b) X is an assignment statement which assigns to variable v or a variable u which may be aliased to v.

(3) the edge l is not taken at β' in H.

The purpose of potential dependence is to find those statements which will be missed by dynamic slicing. If a branch or method invocation statement has affected the slicing criterion, we do not consider that it has potential influence, since the dynamic slice will always include such a statement. We use condition (1) to exclude dynamic control and data dependencies from potential dependencies. Conditions (2) and (3) ensure that β' has potential influence on β. The statement X in condition 2(b) appears in program P, but its execution is prohibited by the evaluation of branch/method invocation in β'. However, if it is executed (possibly due to a change in the statement $S_{\beta'}$), the value of variable v used at β will be affected. We cannot guarantee that v must be re-defined if β' is evaluated to take edge l. This is because even if β' is evaluated differently, execution of the assignment to v may be guarded by other (nested) conditional control transfer statements. Furthermore, condition (2) requires computation of static data dependence. In the presence of arrays and dynamic memory allocations, we can only obtain conservative static data dependencies in general. We now proceed to define a relevant slice, and introduce our relevant slicing algorithm to compute such a slice.

4.2 Relevant Slice and the Relevant Slicing Algorithm

To define relevant slices, first we define the notion of an *Extended Dynamic Dependence Graph* (EDDG). The EDDG captures dynamic control dependencies, dynamic data dependencies and potential dependencies w.r.t. a program execution. It is an extension of the *Dynamic Dependence Graph* (DDG) described in [Agrawal and Horgan 1990]. Each node of the DDG represents one particular occurrence of a statement in the program execution; edges represent dynamic data and control dependencies. The EDDG extends the DDG with potential dependencies, by introducing a dummy node for each branch statement occurrence or a method invocation statement occurrence. For each statement occurrence β in the execution trace, a non-dummy node $nn(\beta)$ appears in the EDDG to represent this occurrence. In addition, if β is a branch statement or a method invocation statement, a dummy node $dn(\beta)$ also appears in the EDDG. As far as the edges are concerned, the following are the incoming edges of any arbitrary node $nn(\beta)$ or $dn(\beta)$ appearing in the EDDG.

- dynamic control dependence edges from non-dummy node $nn(\beta')$ to $nn(\beta)$, iff. β' is dynamically control dependent on β.
- dynamic data dependence edges from both non-dummy node $nn(\beta')$ and dummy node $dn(\beta')$ (if there is a dummy node for occurrence β') to $nn(\beta)$, iff. β' is dynamically data dependent on β.
- potential dependence edges from non-dummy node $nn(\beta')$ and dummy node $dn(\beta')$ (if there is a dummy node for β') to $dn(\beta)$, iff. β' is potentially dependent on β.

These dependencies can be detected during backwards traversal of the execution trace.

What is a Relevant Slice. The *relevant slice* is then defined based on the extended dynamic dependence graph (EDDG) as follows.

Definition 4.2.1. **Relevant slice** for a slicing criterion consists of all statements whose occurrence nodes can be reached from the node(s) for the slicing criterion in the Extended Dynamic Dependence Graph (EDDG).

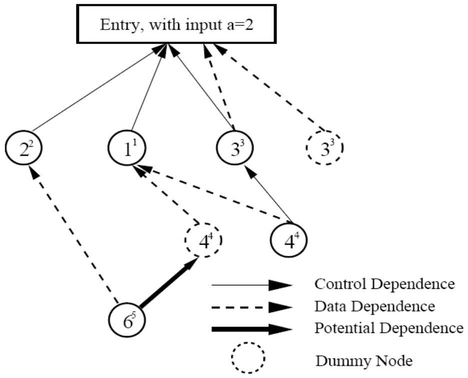


Fig. 10. The EDDG for the program in Figure 9 with input a=2.

In order to accurately capture the effects w.r.t. potential dependencies, we have introduced a dummy node into the EDDG for each occurrence of a branch or method invocation statement. The EDDG can then represent dynamic control dependence edges and potential dependence edges separately. This representation allows the reachability analysis for relevant slicing not to consider dynamic control dependence edges which are immediately after potential dependence edges, because such dynamic control dependencies cannot affect behaviors w.r.t. the slicing criterion. Figure 10 shows the EDDG for the example program in Figure 9 for input `a = 2`. The execution trace is $\langle 1^1, 2^2, 3^3, 4^4, 6^5\rangle$. The resultant relevant slice w.r.t. variable `k` at line 6 consists of lines $\{1, 2, 4, 6\}$, because nodes $1^1, 2^2, 4^4$ and 6^5 can be reached from the criterion 6^5. Note that statement occurrence 6^5 is potentially dependent on 4^4, and 4^4 is dynamically control dependent on statement occurrence 3^3. However, changes related to line 3 will not affect the value of `k` at 6^5, nor will it decide whether line 6 will be executed. Therefore, line 3 should not be included into the slice and reported. By using the dummy node to separate dynamic control dependencies and potential dependencies w.r.t. statement occurrence 4^4, we can easily exclude line 3 of Figure 9 from our relevant slice.

We now discuss how our dynamic slicing algorithm described in Section 3 (operating on compact Java bytecode traces) can be augmented to compute relevant slices. Thus, like the dynamic slicing algorithm, our relevant slicing algorithm also operates on the compact bytecode traces described in Section 2.

As in dynamic slicing, relevant slicing is performed w.r.t. a slicing criterion (H, α, V), where H is the execution history for a certain program input, α represents some bytecodes the programmer is interested in, and V is a set of variables referenced at these bytecodes. Again, the user-defined criterion is often of the form (I, l, V) where I is the input, and l

is a line number of the source program. In this case, H represents execution history of the program P with input I, and α represents the bytecodes corresponding to statements at l.

Figure 11 presents a relevant slicing algorithm, which returns the relevant slice as defined in Definition 4.2.1. This algorithm is based on backward traversal of the compressed execution trace H (described in Section 2). The trace H contains execution flow and identities of accessed variables, so that we can detect various dependencies during the traversal. Although the slice can also be computed after constructing the whole EDDG, this approach is impractical because the entire EDDG may be too huge in practice. Before slicing, we compute the *control flow graph*, which is used to detect potential dependencies and to get last executed bytecode. We also pre-compute the *control dependence graph* [Ferrante et al. 1987] which will be used at lines 22, 23 and 47 of Figure 11, and at line 9 of the `computePotentialDependence` method in Figure 12. Also, prior to running our relevant slicing algorithm we run static points-to analysis [Andersen 1994; Steensgaard 1996]; the results of this analysis are used for determining potential dependencies.

In the relevant slicing algorithm, we introduce a global variable θ, to keep track of variables used by bytecode occurrences β, iff. β is included into the slice because of potential dependencies. In particular, each element in θ is of the form $\langle\beta, prop\rangle$, where β is an bytecode occurrence, and $prop$ is a set of variables. Every variable in $prop$ is used by a bytecode occurrence β', where

— β' is included into φ (the relevant slice) because of potential dependencies, and

— $\beta' = \beta$, or β' is (transitively) dynamically control dependent on β.

The purpose of the δ set (set of unexplained variables) is the same as in the dynamic slicing algorithm. That is, the δ set includes variables used by bytecode occurrence β for explanation, where β is included into the slice φ because β belongs to the slicing criterion, or there is any bytecode occurrence in φ which is dynamically control/data dependent on β.

For each bytecode occurrence β of bytecode b_β encountered during the backward traversal for slicing, we first check if β has affected the slicing criterion via dynamic control/data dependencies as the in dynamic slicing algorithm of Figure 2. In particular, line 19 checks whether β belongs to the slicing criterion. Line 22 checks dynamic control dependencies if b_β is a conditional control transfer bytecode. Line 27 checks dynamic data dependencies. With the introduction of θ, variables in both δ and θ need explanation. Consequently, the `computeDataDependence` method has been slightly changed, as re-defined in Figure 13. If all the three checks fail, the algorithm proceeds to check the potential dependencies at line 38, by invoking the `computePotentialDependence` method in Figure 12.

For the computation of potential dependencies, we need to pre-compute the effects of various outcomes of a control transfer bytecode. Each such outcome triggers a different code fragment whose effect can be summarized by all possible assignment bytecodes executed. This summarization is used by the `computePotentialDependence` method in Figure 12. Note that δ is used to check dynamic data dependencies (line 27 of Figure 11) as well as potential dependencies (line 6 in Figure 12).

The `intersect(`MDS`, `δ`)` method used by the `computePotentialDependence` method in Figure 12 checks whether the execution of the alternative path of a bytecode b_β may define some variables which are used by bytecode occurrences in the slice. Here MDS includes variables which may be defined if b_β is evaluated differently, and δ includes variables which have affected the slicing criterion and need explanation. This check

```
$(H, \alpha, V)$= the slicing criterion
$\delta$= $\emptyset$, a set of variables whose values need to be explained
$\varphi$= $\emptyset$, the set of bytecode occurrences which have affected the slicing criterion
$op\_stack$= empty, the operand stack for simulation
$fram$= empty, the frames of the program execution
$\theta$= empty, bytecode occurrences and their used variables included into the slice because of potential dependencies

relevantSlicing()
    $b_\beta$ = get last executed bytecode from $H$;
    while ($b_\beta$ is defined)
        if ($b_\beta$ is a return bytecode)
            $new\_fram$= $createFrame()$;
            $new\_fram.\gamma$=$\emptyset$;
            $push(fram, new\_fram)$;
        $last\_fram$=$null$;
        if ($b_\beta$ is a method invocation bytecode)
            $last\_fram$=$pop(fram)$;
        $\beta$ = current occurrence of bytecode $b_\beta$;
        $curr\_fram$ = the top of $fram$;
        if ($\beta$ is the last occurrence of $b_\beta$ and $b_\beta \in \alpha$)
            $use\_vars$ = $V \cap$ variables used at $\beta$;
            $\varphi = \varphi \cup \{\beta\}$;
        if ($computeControlDependence(b_\beta, curr\_fram, last\_fram)$ )
            $BC$= all bytecodes occurrences in $curr\_fram.\gamma$ which are statically control dependent on $b_\beta$;
            $curr\_fram.\gamma = curr\_fram.\gamma - BC$ ;
            $use\_vars$ = variables used at $\beta$;
            $\varphi = \varphi \cup \{\beta\}$;
        if ($computeDataDependence(\beta, b_\beta)$)
            $def\_vars$ = variables defined at $\beta$;
            $\delta = \delta - def\_vars$;
            $use\_vars$ = variables used at $\beta$;
            $\varphi = \varphi \cup \{\beta\}$;
            for (each $\langle\beta', prop'\rangle$ in $\theta$)
                $prop' = prop' - def\_vars$;
        if ($\beta \in \varphi$)
            $curr\_fram.\gamma = curr\_fram.\gamma \cup \{\beta\}$;
            $\delta = \delta \cup use\_vars$;
        else
            if ($computePotentialDependence(\beta, b_\beta)$)
                $\varphi = \varphi \cup \{\beta\}$;
                if ( $b_\beta$ is a branch bytecode)
                    $use\_vars$ = variables used at $\beta$; $\theta = \theta \cup \{\langle\beta, use\_vars\rangle\}$;
                if ( $b_\beta$ is a method invocation bytecode)
                    $o$ = the variable to invoke a method; $\theta = \theta \cup \{\langle\beta, \{o\}\rangle\}$;
        if ($b_\beta$ is a branch bytecode or method invocation bytecode)
            $prop$=$\emptyset$;
            for (each $\langle\beta', prop'\rangle$ in $\theta$)
                if ($\beta' = \beta$ or $\beta'$ is control dependent on $\beta$)
                    $prop = prop \cup prop'$; $\theta = \theta - \{\langle\beta', prop'\rangle\}$;
            $\theta = \theta \cup \{\langle\beta, prop\rangle\}$;
        $updateOpStack(\beta, b_\beta)$;
        $\beta$ = $getPrevBytecode(\beta, b_\beta)$;
    return bytecodes whose occurrences appear in $\varphi$;
```

Fig. 11. The relevant slicing algorithm.

```
computePotentialDependence ($\beta$: bytecode occurrence, $b_\beta$: bytecode)
    if ( $b_\beta$ is a branch or method invocation bytecode )
        for (each possible outcome $x$ of $b_\beta$)
            if (outcome $x$ of $b_\beta$ did not occur at $\beta$ )
                $MDS$ = the set of variables which may be defined
                        when outcome $x$ occurs;
                if $(intersect(MDS, \delta))$
                    return true;
                for (each $\langle \beta', prop' \rangle$ in $\theta$)
                    if ($\beta'$ is not control dependent on $\beta$ and $intersect(MDS, prop')$ )
                        return true;
    return false;
```

Fig. 12. Detect potential dependencies for relevant slicing.

```
computeDataDependence ($\beta$: bytecode occurrence, $b_\beta$: bytecode)
    if ($\beta$ defines a variable)
        if ($\beta$ defines a static field or local variable)
            $def\_loc$= get address of the defined static field or local variable from class files;
        if ($\beta$ defines an object field or an array element)
            $G$= compressed operand sequence for $b_\beta$ in the compact bytecode trace $H$
            $\pi$= a root-to-leaf path for $G$;
            $def\_loc$= $getLast(G, \pi)$;
        if ( $def\_loc \in \delta$ )
            return true;
        for (each $\langle \beta', prop' \rangle$ in $\theta$)
            if ($def\_loc \in prop'$)
                return true;
    $\omega$= the set of bytecode occurrences in top $def\_op(b_\beta)$ entries of $op\_stack$;
    if ( $\omega \cap \varphi \neq \emptyset$)
        return true;
    return false;
```

Fig. 13. Detect dynamic data dependencies for relevant slicing.

is non-trivial because MDS contains static information, while δ contains dynamic information. Let $meth$ be the method that the bytecode b_β belongs to, and $curr\_invo$ be the current invocation of $meth$. Note that both MDS and δ include local variables and fields. Every local variable in MDS is also a local variable of method $meth$, and is represented by its identity $\langle var \rangle$. Every local variable in δ for explanation is represented as $\langle invo, var \rangle$, where $invo$ refers to the method invocation which uses the local variable var. Many points-to analysis algorithms [Andersen 1994; Steensgaard 1996] represent abstract memory locations of objects using their possible allocation sites. Thus, we represent an object field in MDS as $\langle site, name \rangle$, where $site$ refers to a possible allocation site of the object, and $name$ refers to the name of the field; we also represent an object field in δ as $\langle site, \_timestamp, name \rangle$, where $site$ refers to the allocation site of the object, $\_timestamp$ distinguishes between objects created at the same allocation site, and $name$ refers to the name of the field. Note that $\_timestamp$ is only important for detecting dynamic data dependencies. The `intersect(`MDS`,` δ`)` method returns true iff:

—there is a local variable where $\langle var\rangle \in MDS$ and $\langle curr\_invo, var\rangle \in \delta$, or

—there is a field where $\langle site, name\rangle \in MDS$ and $\langle site, \_timestamp, name\rangle \in \delta$.

We do not consider partial results (*i.e.* operands in the operand stack) for potential dependencies because partial results will not be transferred between bytecodes of different statements.

The proof of correctness of the relevant slicing algorithm in Figure 11 is similar to that of the dynamic slicing algorithm in Figure 2, which is described in Section 3.5. *Details of the proof of correctness can be found in Appendix C.*

Now we analyze the cost of the relevant slicing algorithm in Figure 11. The space overheads of the slicing algorithm are $O(N^2+m^3)$, and the time overheads are $O(m^2 \cdot N^3)$, where N is the length of the execution, and m is the number of bytecodes of the program. Since the relevant slicing algorithm is similar with the dynamic slicing algorithm in Figure 2, the cost analysis is also similar except costs w.r.t. the θ and MDS.

The θ set contains at most N elements of the form $\langle\beta, prop\rangle$, because every bytecode occurrence β has at most one element $\langle\beta, prop\rangle$ in θ. The size of each $prop$ is $O(N)$, because at most N bytecode occurrences are (transitively) dynamically control dependent on β and every bytecode occurrence uses a constant number of variables. Consequently, the space overheads of θ are $O(N^2)$.

Each MDS includes the set of variables which may be defined when a specific outcome of a branch bytecode occurs. If m is the number of bytecodes in the program, clearly there are at most m MDSs. How do we bound the size of each MDS? Each MDS may have at most m assignments and each of these assignments may affect at most m locations (provided we distinguish locations based on allocation sites as is common in points-to analysis methods). Thus, the size of each MDS is $O(m^2)$. Since there are at most m MDSs, the space overheads of maintaining the MDSs are $O(m^3)$. The other portions of the relevant slicing algorithm are taken from dynamic slicing; the space overheads of these portions of the relevant slicing algorithm are $O(N)$, as explained in Section 3. So, the overall space overheads of our relevant slicing algorithm are $O(N^2 + m^3)$.

We now calculate the time overheads of relevant slicing. First we estimate the time overheads for maintaining the θ set in the relevant slicing algorithm. Note that the θ set contains $O(N)$ elements of the form $\langle\beta, prop\rangle$. The set difference operation at line 33 of Figure 11 is executed $O(N^2)$ times. Since each $prop$ set contains $O(N)$ variables, the overall time overheads to perform the set difference operation at line 33 of Figure 11 are $O(N^3)$. The total time overheads to perform the set union operation at lines 41, 43 and 49 of Figure 11 are $O(N^2)$, because lines 41, 43 and 49 are executed $O(N)$ times and the size of the θ set is $O(N)$. Given a bytecode occurrence β and the θ set, there are a constant number of elements $\langle\beta', prop'\rangle \in \theta$, where β' is (directly) dynamically control dependent on β. This is because there is no explicit `goto` statement in Java programs. Different occurrences of the same bytecode are dynamically control dependent on different bytecode occurrences. Note that there are a constant number of bytecodes which are statically control dependent on the bytecode b_β of occurrence β. Thus, there are a constant number of bytecode occurrences which are *directly* dynamically control dependent on β. In addition, every bytecode occurrence β' has at most one $\langle\beta', prop'\rangle$ in θ. Consequently, line 48 is executed $O(N)$ times, and the overall time overheads to execute line 48 are $O(N^3)$. The total time overheads to perform check at line 12 of Figure 13 are $O(N^3)$, which is similar to perform set difference operation at line 33 of Figure 11.

Now, we analyze the overall time overheads to perform the `intersect` operation by the `computePotentialDependence` method in Figure 12. The `intersect` operation at line 6 of Figure 12 can be executed at most N times. In each execution of the intersect operation we compare the contents of MDS and δ. Since the size of MDS is $O(m^2)$ and the size of the set δ is $O(N)$, therefore the time overheads of a single `intersect` operation are $O(N \cdot m^2)$. Thus, the total time overheads to execute all the `intersect` operations at line 6 of Figure 12 are $O(N \cdot N \cdot m^2)$. Similarly, the total time overheads to execute all the `intersect` operations at line 9 of Figure 12 are $O(N^3 \cdot m^2)$, since this `intersect` operation is executed $O(N^2)$ times.

The other portions of the relevant slicing algorithm are taken from dynamic slicing; the time complexity of these portions of the relevant slicing algorithm is $O(N^2)$, as discussed in Section 3. This leads to a time complexity of $O(N^3 \cdot m^2)$ for our relevant slicing algorithm.

5. EXPERIMENTAL EVALUATION

We have implemented a prototype slicing tool and applied it to several subject programs. The experiments report time and space efficiency of our trace collection technique. We compress traces using both SEQUITUR and RLESe, and compare the time overheads and effectiveness of the improved compression algorithm against the previous one, in order to investigate the cost effectiveness of the proposed compression algorithm. In addition, the experiments report time overheads to compute dynamic and relevant slices, and the sizes of these slices.

To collect execution traces, we have modified the Kaffe virtual machine [Kaffe] to monitor interpreted bytecodes. In our traces, we use object identities instead of addresses to represent objects. Creation of structures such as multi-dimensional arrays, constant strings etc. may implicitly create objects. We trace and allocate identities to these objects as well. The virtual machine may invoke some methods automatically when “special” events occur (*e.g.* it may invoke the static initializer of a class automatically when a static field of the class is first accessed). These event are also stored, and used for backward traversal of the execution trace.

Most Java programs use libraries. Dynamic dependencies introduced by the execution of library methods are often necessary to compute correct slices. Our implementation does not distinguish between library methods and non-library methods during tracing and slicing. However, after the slicing finishes, we will filter statements inside the libraries from the slice. This is because libraries are often provided by other vendors, and programmers will not look into them.

5.1 Subject Programs

The subjects used in our experiments include six subjects from the Java Grande Forum benchmark suite [JGF], three subjects from the SPECjvm suite [SPECjvm98], and one medium sized Java utility program [Berk and Ananian]. Descriptions and inputs of these subjects are shown in Table III. We ran and collected execution traces of each program once, with inputs shown in the third column in Table III. Corresponding execution characteristics are shown in Table IV. The second column in Table IV shows the number of bytecodes of these subjects. The column *Executed bytecodes* in Table IV presents the number of distinct bytecodes executed during one execution, and the fourth column in Table IV shows corresponding total number of bytecode occurrences executed. The last two

Subject	Description	Input
Crypt	IDEA encryption and decryption	200,000 bytes
SOR	Successive over-relaxation on a grid	100 × 100 grid
FFT	1-D fast Fourier transform	2^{15} complex numbers
HeapSort	Integer sorting	10000 integers
LUFact	LU factorisation	200 × 200 matrix
Series	Fourier coefficient analysis	200 Fourier coefficients
_201_compress	Modified Lempel-Ziv method (LZW)	228.tar in the SPECjvm suite
_202_jess	Java Expert Shell System	fullmab.clp in the SPECjvm suite
_209_db	Performs multiple database functions	db2 & scr2 in the SPECjvm suite
JLex	A Lexical Analyzer Generator for Java	sample.lex from the tool's web site

Table III. Descriptions and input sizes of subject programs.

Subject	Total # of Bytecodes	Executed Bytecodes	# Bytecode Instances	# Branch Instances	# Method invocations
Crypt	2,939	1,828	103,708,780	1,700,544	48
SOR	1,656	740	59,283,663	1,990,324	26
FFT	2,327	1,216	72,602,818	2,097,204	37
HeapSort	1,682	679	11,627,522	743,677	15,025
LUFact	2,885	1,520	98,273,627	6,146,024	41,236
Series	1,800	795	16,367,637	1,196,656	399,425
_201_compress	8,797	5,764	166,537,472	7,474,589	1,999,317
_202_jess	34,019	14,845	5,162,548	375,128	171,251
_209_db	8,794	4,948	38,122,955	3,624,673	19,432
Jlex	22,077	14,737	13,083,864	1,343,372	180,317

Table IV. Execution characteristics of subject programs.

Subject	Orig. Trace	Trace Table	RLESe Sequences	All	All/Orig. (%)
Crypt	64.0M	8.9k	8.8k	17.8k	0.03
SOR	73.4M	7.6k	10.8k	18.5k	0.02
FFT	75.2M	8.2k	87.3k	95.5k	0.12
HeapSort	23.6M	7.7k	1.7M	1.7M	7.20
LUFact	113.1M	9.1k	179.7k	188.9k	0.16
Series	24.4M	7.7k	444.4k	452.2k	1.81
_201_compress	288.6M	23.7k	8.8M	8.8M	3.05
_202_jess	25.7M	79.2k	3.4M	3.4M	13.23
_209_db	194.5M	29.0k	39.2M	39.2M	20.15
Jlex	49.9M	62.6k	1.5M	1.5M	3.01

Table V. Compression efficiency of our bytecode traces. All sizes are in bytes.

columns present the number of branch bytecode occurrences and method invocations, respectively. Bytecodes from Java libraries are not counted in the table; however, bytecodes of user methods called from library methods are included in the counts shown.

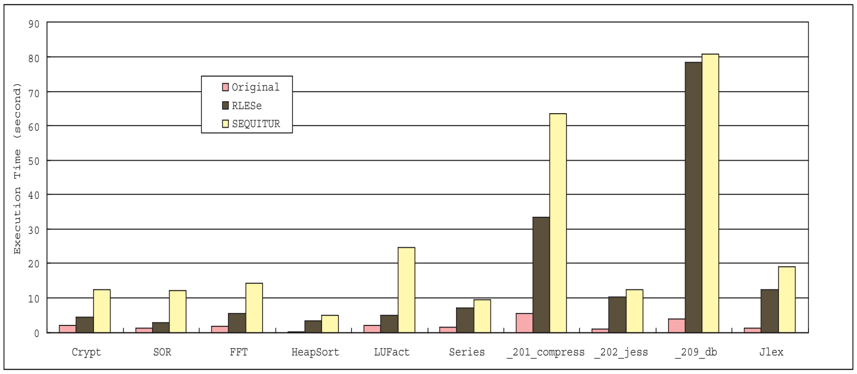


Fig. 14. Time overheads of RLESe and SEQUITUR. The time unit is *second*.

5.2 Time and Space Efficiency of Trace Collection

We first study the efficiency of our compact trace representation. Table V shows the compression efficiency of our compact trace representation. The column *Orig. Trace* represents the space overheads of storing uncompressed execution traces on disk. To accurately measure the compression achieved by our RLESe grammars, we leave out the untraced bytecodes from the *Orig. Trace* as well as the compressed trace. Next two columns *Trace Table* and *RLESe Sequences* show the space overheads to maintain the trace tables and to store the compressed operand sequences of bytecodes on disk. The column *All* represents the overall space costs of our compact bytecode traces, *i.e.* sum of space overheads of *Trace Table* and *RLESe Sequences*. The % column indicates *All* as a percentage of *Orig. Trace.* For all our subject programs, we can achieve at least 5 times compression. Indeed for some programs we get more than two orders of magnitude (*i.e* 100 times) compression.

Figure 14 presents the absolute running time of the subject programs without instrumentation, tracing with RLESe and tracing with SEQUITUR. All experiments were performed on a Pentium 4 3.0 GHz machine with 1 GB of memory. From this figure we can see that the slowdown for collecting traces using RLESe compared to the original program execution is $2 - 10$ times for most subject programs, and the slowdown is near 20 times for $\_209\_db$ which randomly accesses a database. This shows that our method is suitable for program executions with many repetitions, but the time overhead may be not scalable to program executions with too many random accesses. Note that these additional time overheads are caused by compression using RLESe/SEQUITUR. The results reflect one limitation of RLESe (as well as SEQUITUR) — the time efficiency of the compression algorithm heavily depends on the efficiency of checking the digram uniqueness property. To speed up this check, a sophisticated index of digrams should be used. However, the choice of the index has to trade-off between time and space efficiency, since the index will introduce additional space overheads during trace collection. In our implementation, we use the B+ tree (instead of sparse hash) to index digrams during compressing, and the index is no longer needed after compression is completed. However, the index may lead to high time overheads in some situations.

We now compare the time and space efficiency of RLESe against SEQUITUR. Both algorithms were performed on operand sequences of bytecodes. For a fair comparison,

Subject	RLESe %	SEQUITUR %
Crypt	0.03	0.07
SOR	0.02	0.04
FFT	0.12	0.34
HeapSort	7.20	7.20
LUFact	0.16	0.40
Series	1.81	2.50
_201_compress	3.05	3.12
_202_jess	13.23	13.62
_209_db	20.15	18.35
Jlex	3.01	4.01

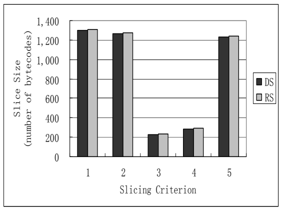
Table VI. Comparing compression ratio of RLESe and SEQUITUR.

Subject	RLESe	SEQUITUR
Crypt	800,605	20,499,488
SOR	393,202	25,563,918
FFT	2,020,976	25,722,054
HeapSort	2,262,916	7,066,784
LUFact	561,233	42,586,136
Series	2,501,001	9,507,982
_201_compress	10,240,652	80,923,387
_202_jess	2,090,153	6,666,460
_209_db	24,386,746	54,033,314
Jlex	4,555,338	15,497,211

Table VII. The number of times to check digram uniqueness property by RLESe and SEQUITUR.

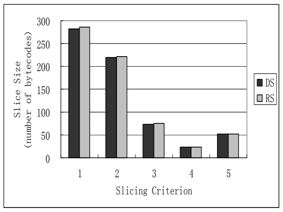
we use the same index to search for identical/similar digrams in the implementation of both algorithms. Table VI compares the space costs of both algorithms by presenting their compression ratio (in percentage). From the tables, we can see that the space consumption of compressed traces produced by both algorithms are somewhat comparable. RLESe outperforms SEQUITUR for eight subject programs, and SEQUITUR outperforms for one (where the _209_db program randomly accesses a database and does not have many contiguous repeated symbols in operand traces). Since RLESe employs run-length encoding of terminal and non-terminal symbols over and above the SEQUITUR algorithm, nodes in grammars produced by RLESe are usually less than those produced by SEQUITUR.

Figure 14 compares the time overheads of RLESe and SEQUITUR. Clearly RLESe outperforms SEQUITUR in time on studied programs. The time overheads of both algorithms are mainly caused by checking digram uniqueness property. RLESe usually produces less nodes in grammars, so that similar digrams can be found more efficiently. In addition, RLESe checks this property after contiguous repeated symbols have finished, whereas SEQUITUR does this on reading every symbol. Table VII shows this by representing the frequency of checking the digram uniqueness property. When there are many contiguous repeated symbols in the execution traces (*e.g.* the LUFact, SOR subjects), RLESe checks the property much less frequently than SEQUITUR; consequently the tracing overheads of RLESe are also much less than those of SEQUITUR.



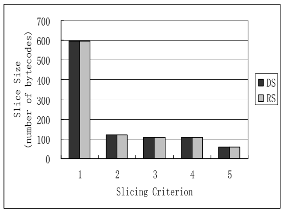
Pgm.	DS	RS	%
2,939	862	868	0.70

Crypt



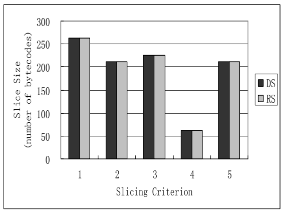
Pgm.	DS	RS	%
1,656	130	131	0.77

SOR



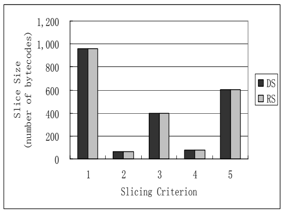
Pgm.	DS	RS	%
2,327	199	199	0.00

FFT



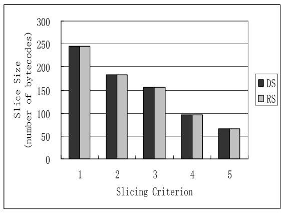
Pgm.	DS	RS	%
1,682	195	195	0.00

HeapSort



Pgm.	DS	RS	%
2,885	420	420	0.00

LUFact



Pgm.	DS	RS	%
1,800	149	149	0.00

Series

Fig. 15. Compare sizes of relevant slices with dynamic slices. *Pgm.* represents size of the program. *DS* and *RS* represent average sizes of dynamic slices and relevant slices, respectively. All sizes are reported as the number of bytecodes. The last % column represents the increased sizes of relevant slices (*i.e.* $\frac{RS-DS}{DS}$) in percentage.

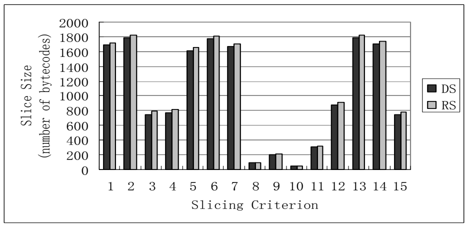
5.3 Overheads for Computing Dynamic Slice and Relevant Slice

We now measure the sizes of dynamic and relevant slices and the time overheads to compute these slices. Thus, apart from studying the absolute time/space overheads of dynamic slicing, these experiments also serve as comparison between dynamic and relevant slicing. For each subject from the Java Grande Forum benchmark suite, we randomly chose five distinct slicing criteria because of their relatively simple program structures; for other bigger subjects, we randomly chose fifteen distinct slicing criteria.

Dynamic slice and relevant slice. Figure 15 and 16 present the slice sizes. As we can see, most dynamic slices and relevant slices were relatively small, which were less than 30% of corresponding source programs on average in our experiments. This is because programs often consist of several parts which work (almost) independently. The irrelevant portions are excluded from both dynamic slices and relevant slices.

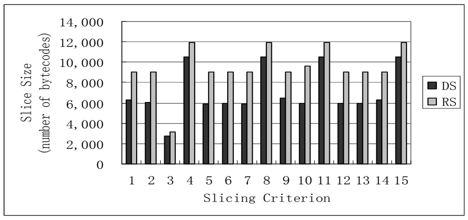
In our experiments, control and data structures of subject programs from Java Grande Forum benchmark suite are quite simple. Furthermore, class hierarchies of these programs are simple, with only limited use of inheritance. The relevant slices are almost the same as corresponding dynamic slices for these programs, as shown in Figure 15.

On the other hand, other subject programs (*i.e.* _201_compress, _202_jess, _209_db, and JLex) are more complex. These programs have more sophisticated control structures. In addition, these programs use inheritance and method overloading and overriding, so that



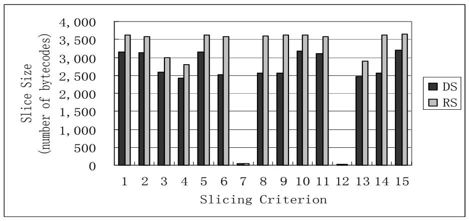
Pgm.	DS	RS	%
8,797	1,054	1,082	2.66

_201_compress



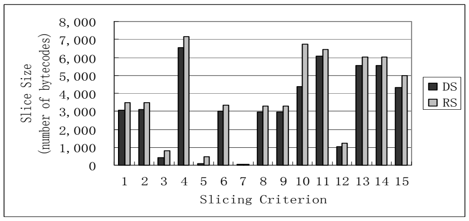
Pgm.	DS	RS	%
34,019	7,023	9,440	34.42

_202_jess



Pgm.	DS	RS	%
8,794	2,448	2,988	22.06

_209_db



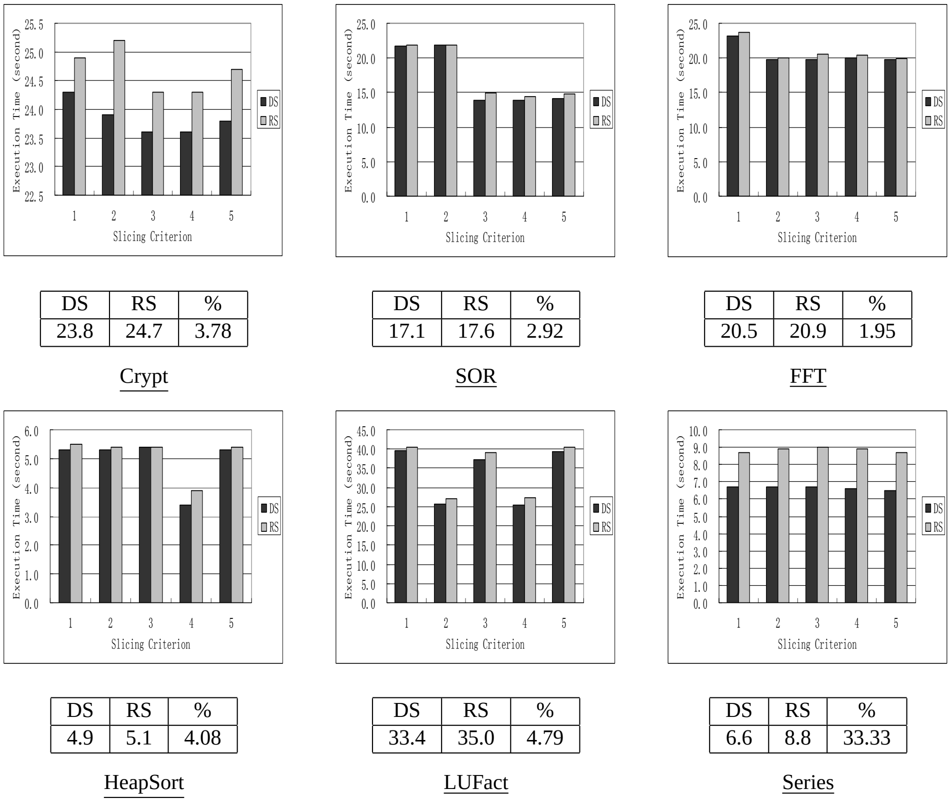
Pgm.	DS	RS	%
22,077	3,283	3,784	15.26

JLex

Fig. 16. Compare sizes of relevant slices with dynamic slices. *Pgm.* represents size of the program. *DS* and *RS* represent average sizes of dynamic slices and relevant slices, respectively. All sizes are reported as the number of bytecodes. The last % column represents the increased sizes of relevant slices (*i.e.* $\frac{RS-DS}{DS}$) in percentage.

different methods may be called by the same method invocation bytecode. Both of the factors lead to potential dependencies between bytecode occurrences. More importantly, these subject programs involve substantial use of Java libraries, such as collection classes and I/O classes. These library classes contribute a lot to relevant slices, because of their complex control structures and usage of object-oriented features like method overloading (which lead to potential dependencies). In fact, if we do not consider potential dependencies inside such library classes, the average sizes of relevant slices for _201_compress, _202_jess, _209_db, and JLex were 1072, 8393, 2523, and 3728, which were 1.71%, 19.51%, 3.06%, 13.55% bigger than corresponding dynamic slices, respectively.

Time overheads. Figure 17 and 18 show the time overheads to compute dynamic slices and relevant slices. Clearly, the time overheads to compute dynamic slices and relevant slices are sensitive to choice of programs and slicing criteria. According to our slicing algorithms in Figure 2 and 11, the time overheads are mainly caused by two tasks: (1) extract operand sequences of bytecodes and backwards traverse, and (2) update and compare various sets to detect dependencies. For a particular execution, the first part is common for slicing w.r.t. every slicing criterion, and the second part is sensitive to the sizes of the slices. For bigger slices, their sets to detect dependencies are also bigger during slicing, resulting in longer time to maintain and compare.



DS	RS	%
23.8	24.7	3.78

Crypt

DS	RS	%
17.1	17.6	2.92

SOR

DS	RS	%
20.5	20.9	1.95

FFT

DS	RS	%
4.9	5.1	4.08

HeapSort

DS	RS	%
33.4	35.0	4.79

LUFact

DS	RS	%
6.6	8.8	33.33

Series

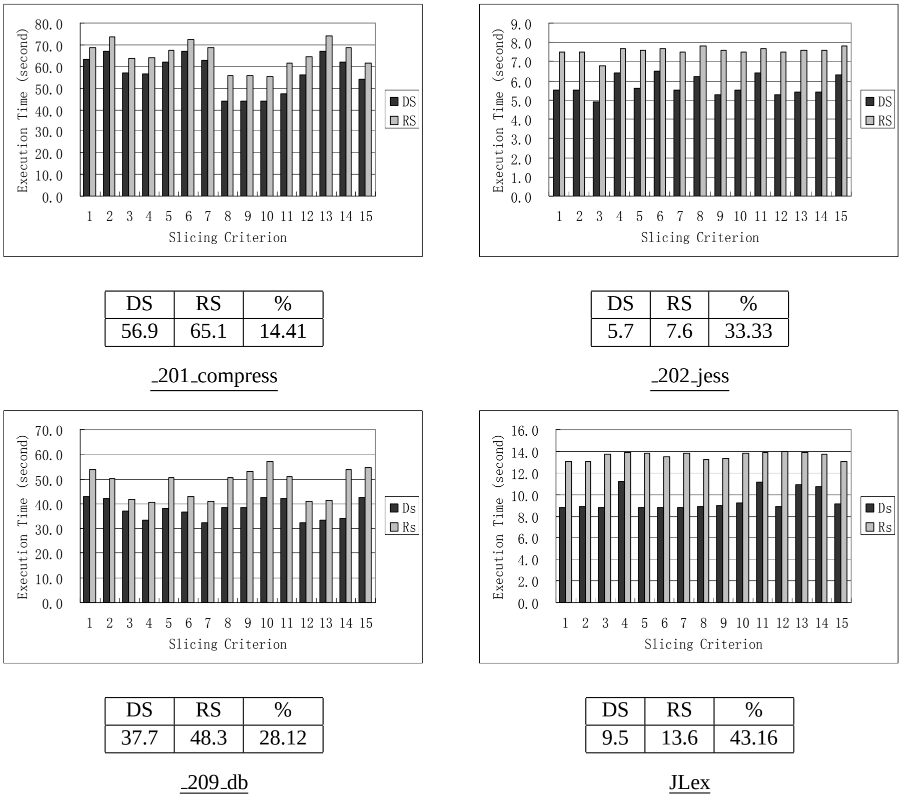
Fig. 17. Compare time overheads of relevant slicing with dynamic slicing. *DS* and *RS* represent average time to perform dynamic slicing and relevant slicing, respectively. All time are reported in second. The last % column represents the increased time for relevant slicing (*i.e.* $\frac{RS-DS}{DS}$) in percentage.

5.4 Summary and Threats to Validity

In summary, the RLESe compaction scheme achieves comparable or better compression ration than SEQUITUR. The time overheads for the online compaction in RLESe is found to be less than the compaction time in SEQUITUR.

Our compact trace representation can be used to perform dynamic slicing and relevant slicing efficiently, both in time and space. The resultant dynamic slices are relatively small, compared against the entire program. They can guide further analysis (*e.g.* program debugging) working on a small but important portion of the program. Relevant slices are often bigger than dynamic slices (since they consider *potential dependencies*), but relevant slices are still relatively small compared against the entire program. Of course, relevant slicing for a given criterion takes more time than dynamic slicing for the same criterion. However, the increased time overheads, which depend on the choice of slicing criterion, the control structures and data flow structures of the program, were tolerable in our experiments.

We note that there are various threats to validity of the conclusion from our experiments. Our conclusions on less time overheads of RLESe (as compared to SEQUITUR) can be invalidated if the execution trace has very few contiguous repeated symbols. In this case,



DS	RS	%
56.9	65.1	14.41

_201_compress

DS	RS	%
5.7	7.6	33.33

_202_jess

DS	RS	%
37.7	48.3	28.12

_209_db

DS	RS	%
9.5	13.6	43.16

JLex

Fig. 18. Compare time overheads of relevant slicing with dynamic slicing. *DS* and *RS* represent average time to perform dynamic slicing and relevant slicing, respectively. All time are reported in second. The last % column represents the increased time for relevant slicing (*i.e.* $\frac{RS-DS}{DS}$) in percentage.

RLESe checks the diagram uniqueness property almost as frequently as SEQUITUR. This can happen in a program with random data accesses (*e.g.*, the _209_db subject program used in our experiments). Our conclusions on the size of dynamic and relevant slices can be invalidated for programs with little inherent parallelism resulting in long chains of control/data dependencies. For such programs, the dynamic and the relevant slices may not be substantially smaller than the program size. Furthermore, as the slice sizes increase so do the time to compute these slices, since additional statements in the slice will result in more unexplained variables in the δ set. It is also possible to encounter a situation where the dynamic slice is quite small compared to the program size, but the relevant slice is much bigger than the dynamic slice. Since relevant slice computation involves detecting potential dependencies and potential dependencies involve computing static data dependencies, a lot depends on the accuracy of the points-to analysis used to detect static data dependencies. If the points-to analysis is very conservative, it may lead to a large relevant slice.

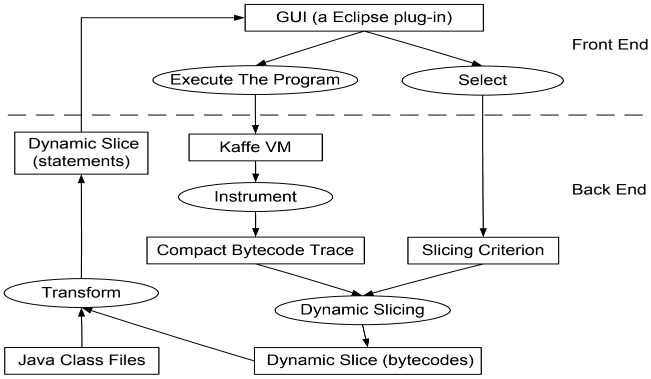


Fig. 19. The architecture of the JSlice tool.

6. JSLICE: A DYNAMIC SLICING TOOL FOR JAVA PROGRAMS

We have made our Java dynamic slicing tool available for use by researchers and developers. The tool is integrated with the Eclipse platform[1]. Programmers can specify the dynamic slicing criterion, perform dynamic slicing, and inspect the dynamic slice in the Integrated Development Environment (IDE) of Eclipse. Note that programmers also develop and debug Java programs in this IDE. They can use this dynamic slicing tool easily during their development. We will briefly introduce the dynamic slicing tool in the following. More details are available in the tool's web site `http://jslice.sourceforge.net/`

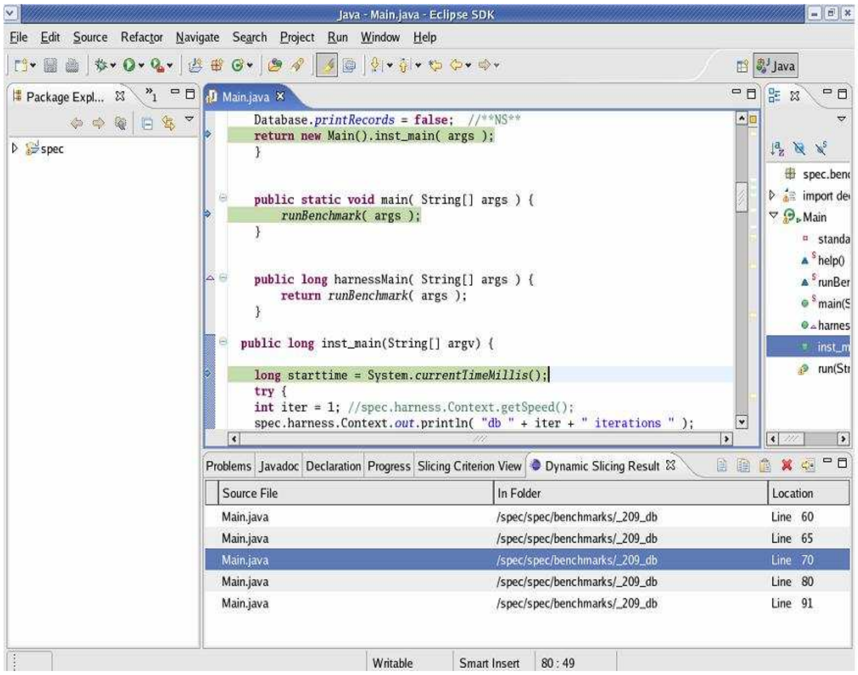
Figure 19 presents the architecture of our *JSlice* dynamic slicing tool. The tool consists of two parts: (a) a front end, which is the user interface, and (b) a back end, which collects traces and performs dynamic slicing.

The front end is implemented as a Eclipse plug-in. It helps programmers to specify the dynamic slicing criterion, and read the dynamic slice. For example, programmers can specify the slicing criterion by selecting a line in the source code editor. All variables referenced at the selected line will be included into the slicing criterion. Figure 20 demonstrators how statements included in the dynamic slice are highlighted in the source code editor, so that programmers can easily find and inspect them.

The back end is integrated with the open-source Kaffe Virtual Machine. When Kaffe is executing a Java program, we instrument the interpreter of the virtual machine, and collect the compact trace at the level of bytecode. After the Java program terminates, we perform dynamic slicing on the compact trace without decompression, w.r.t. the slicing criterion specified via the front end. The resultant slice is then transformed to the source code level with the help of information available in Java class files.

The dynamic slicing tool supports various ways to specify the slicing criterion. When a programmer selects a line in the source code editor as the slicing criterion in the tool, he/she can specify whether the last occurrence or all occurrences of this statement should be considered as the criterion.When a programmer is interested in variables referenced at a specific occurrence of a statement in the middle of the execution, he/she can specify the

[1] Details about Eclipse can be found at `http://www.eclipse.org/`.



```
    Database.printRecords = false;  //**NS**
    return new Main().inst_main( args );
    }


    public static void main( String[] args ) {
        runBenchmark( args );
    }


    public long harnessMain( String[] args ) {
        return runBenchmark( args );
    }

  public long inst_main(String[] argv) {

    long starttime = System.currentTimeMillis();
    try {
    int iter = 1; //spec.harness.Context.getSpeed();
    spec.harness.Context.out.println( "db " + iter + " iterations " );
```

Source File	In Folder	Location
Main.java	/spec/spec/benchmarks/_209_db	Line 60
Main.java	/spec/spec/benchmarks/_209_db	Line 65
Main.java	/spec/spec/benchmarks/_209_db	Line 70
Main.java	/spec/spec/benchmarks/_209_db	Line 80
Main.java	/spec/spec/benchmarks/_209_db	Line 91

Fig. 20. Highlighted statements capture the dynamic slice computed by the JSlice tool.

statement as the slicing criterion, and terminate the program's execution as soon as the "interesting" statement occurrence appears. The tool will then return the slice w.r.t. variables referenced at the "interesting" statement occurrence. The next example shows how our dynamic slicing tool supports this feature. Suppose we have the following program fragment,

1. **for** $(i = 0; i < 10; i++)\{$
2. $\quad j = k + f(i);$
3. $\}$

and we are interested in how the value of variable j in the 6th loop iteration is computed. We can specify statement 2 as the dynamic slicing criterion, and add two lines (*i.e.* line 3 and 4) into the program as shown in the following.

1. **for** $(i = 0; i < 10; i++)\{$
2. $\quad j = k + f(i);$
3. $\quad$ **if** $(i >= 5)$
4. $\qquad$ java.lang.System.exit(1);
5. $\}$

Our tool will then return the dynamic slice w.r.t. variables referenced at the 6th occurrence of statement 2.

7. ONGOING EXTENSIONS OF OUR SLICING TOOL

The dynamic slicing tool JSlice supports most features of the Java programming languages, such as object, field, inheritance, polymorphism and etc. In future, we plan to enhance JSlice to support more features of the Java programming language. In particular, exceptions, reflection and multi-threading are widely used features of the Java programming language. We plan to extend our dynamic slicing tool to handle these features in the following manner.

7.1 Exceptions

When a program violates any semantic constraint of the Java programming language, the Java virtual machine throws an exception to signal this error [Joy et al. 2000]. This exception will cause a non-local transfer of control from the point where the exception occurred to the exception handler which can be specified by the programmer. It is necessary to store this non-local transfer of control during trace collection, so that we can reconstruct such a control transfer during the backward traversal for dynamic slicing.

JSlice maintains traces of each bytecode separately in the trace tables for the program, as discussed in Section 2. Thus, the non-local transfer of control should be stored in the traces of the first bytecode of the exception handler. Note that this control transfer will cause the Java virtual machine to change the call stack, in the process of looking for an appropriate exception handler. In particular, the virtual machine will pop method invocations from the call stack up to the method invocation $invo\_excep$ (which the exception handler belongs to), and then execute the exception handler. For each invocation $invo$ which is popped from or revised in the call stack, we need to record the following (assume that $invo$ is an invocation of method $meth$).

—the class name of $meth$, and

—the method name of $meth$, and

—the signature of $meth$, and

—the id/address of last executed bytecode of $invo$, and

—the size of the operand stack of $invo$ before $invo$ is popped or revised.

When the first bytecode b of an exception handler is encountered during the backward traversal for slicing, the dynamic slicing algorithm should retrieve information from the traces of b, and reconstruct the call stack, so that the backward traversal can continue.

Exception handling also introduces extra dependencies into the program: the dynamic control dependence between (a) the bytecode occurrence which throws the exception and (b) the exception handler which catches the exception [Sinha and Harrold 2000]. This means that, when any bytecode instance in the exception handler is included into the dynamic slice, the bytecode occurrence which throws the exception should also be included into the slice.

Finally block. For Java programs, exception handlers often come with a `finally` block, where the Java Virtual Machine ensures that the finally block is always executed even if an unexpected exception occurs. However, the usage of the finally block complicates the construction of the control flow graph of a Java method, as discussed in the following. During the execution of a Java method, a finally block is always entered by executing a `JSR` bytecode. The semantics of the `JSR` bytecode is very similar to that of the `goto` bytecode. However, when a `JSR` bytecode b is executed, the address of the bytecode b' which immediately follows b is stored into the operand stack. When the finally block finishes execution, the saved address of the bytecode b' is retrieved and the execution continues from the bytecode b'. In other words, the bytecode b', which is executed after the

```
findNext ($meth$: a Java method)
    initialize each element of the array $next$ and $succ$ to $\emptyset$;
    initialize $entryStack$ to null;
    for (each bytecode $b$ of the method $meth$)
        if (the bytecode $b$ is a JSR bytecode)
            $entry$ = the operand bytecode of the JSR bytecode $b$;
            mark the bytecode $entry$ as an entry of a finally block;
            $b'$ = the bytecode which immediately follows the JSR bytecode $b$;
            $next[entry] = next[entry] \cup \{b'\}$;
    for (each bytecode $b$ of the method $meth$)
        if (the bytecode $b$ is an entry of a finally block)
            $push(entryStack, b)$;
        if (the bytecode $b$ is the last bytecode of a finally block)
            $entry = pop(entryStack)$;
            $exit = b$;
            $succ[exit] = next[entry]$;
    return $succ$;
```

Fig. 21. The algorithm to find the bytecodes which may be executed after each `finally` block.

finally block, is not represented as an operand in the last bytecode of the finally block. As a result, it is not clear which bytecodes may be executed after a finally block. In order to discover this kind of information, the algorithm in Figure 21 is used. Given a method $meth$, the algorithm in Figure 21 returns an array $succ$, where for every bytecode $exit$ which is the last bytecode of a finally block, $succ[exit]$ represents the set of bytecodes which may be executed after the bytecode $exit$. The algorithm proceeds by traversing the bytecode sequence of the method $meth$ twice. During the first traversal, we mark the entry bytecode $entry$ of each finally block, and maintain $next[entry]$, the set of bytecodes which may be executed after the finally block. During the second traversal, for every $entry$ bytecode of a finally block, we detect the corresponding $exit$ bytecode which exits the finally block. Additional, we set $succ[exit]$ to $next[entry]$, so that we can get the control flow information w.r.t. these $exit$ bytecodes and construct the control flow graph as usual. The stack $entryStack$ is required here because of nested finally blocks.

7.2 Reflection

Reflection gives the Java code access to internal information of classes in the Java Virtual Machine, and allows the code to work with classes selected during execution, not in the source code. The main difficulty to support reflection for slicing lies in the fact that many reflection methods are implemented as native methods, and JSlice cannot trace details of native methods. Of all the native reflection methods, the following two kinds of methods are particularly important for dynamic slicing.

—*Native methods which invoke a Java method*, such as the java.lang.reflect.Method.invoke method. Clearly, there exists a control transfer from the native method to the callee Java method. This control transfer is important for dynamic slicing, since we need to traverse the callee Java method for dynamic data dependence analysis. Here, we have to explicitly record the control transfer. The class name, method name, and signature of the callee method should be recorded.

—*Native methods which read/write fields or arrays*, where we can deduce which variables are accessed according to the parameters and the invoking objects. For example, field access methods in the java.lang.reflect.Field class fall into this category. These native methods are also essential for dynamic data dependence analysis. Note that these methods behave similarly with field/array access bytecodes, we can trace and analyze these methods in a similar way as corresponding bytecodes. That is, we trace the address (or identity) corresponding to the object/array, and the field name or the index of the array element. During dynamic slicing, such information is retrieved to detect dynamic data dependencies.

7.3 Multi-threading

We plan to extend JSlice to support multi-threaded Java programs. The trace representation for a multi-threaded Java program could be similar to that for a single-threaded Java program. That is, each method has one trace table, and each row of the table maintains the control and data flow traces of a specific bytecode (see Section 2 for the trace representation). However, Java threads often communicate with each other through inter-thread events, such as shared variable access events, wait/notify events. The order of these events is required for dynamic slicing, because such an order is essential to reason/detect inter-thread dynamic dependencies. Levrouw et al. have proposed an efficient mechanism which can be used to trace the order of these inter-thread events [Levrouw et al. 1994]. We now briefly describe this approach in the following.

Levrouw's approach is based on the Lamport Clocks [Lamport 1997]. During the execution, each thread t_i has a scalar clock c_t^i, and each object o also maintains a clock c_o. These clocks are initialized to 0. Whenever there is an inter-thread event e where the thread t_i accesses the object o, this event is recorded with a time stamp $c_e = max(c_t^i, c_o) + 1$. The function max returns the maximum value of the two inputs. Additionally, c_t^i and c_o are updated to c_e. These recorded time stamps actually impose an partial order on all inter-thread events. Levrouw et al. show that we can replay the original execution and re-construct the dynamic dependencies, by enforcing inter-thread events following the partial order.

There is one practical problem in employing the above scheme for tracing multi-threaded Java programs — all objects can be accessed by different threads, and it is often non-trivial to know which objects are shared before the execution. As a result, every access to an object should be considered as an inter-thread event. However, if we trace the time stamp for every object access, the trace size may explode. Fortunately, Levrouw et al. show that it is not necessary to trace all time stamps to record the partial order. In particular, for an inter-thread event e where the thread t_i accesses the object o, let c_t^i be the time stamp of t_i, and c_o of be the time stamp of o. We only need to trace the increment of c_t^i before and after the event e, if $c_t^i < c_o$. The reader is refereed to [Ronsse and Bosschere 1999; Levrouw et al. 1994] for details. Note that the tracing scheme given here will work for multi-threaded Java programs running on multi-processor platforms as well.

The dynamic slicing algorithm for multi-threaded programs is similar to that for single-threaded programs (see Section 3). However, the algorithm should now maintain several operand stacks and call stacks, each of which corresponds to one thread. At any specific time, only one operand stack and one call stack are active. When we encounter an inter-thread event during the backward traversal, we pause the traversal along this thread until we have traversed all inter-thread events with a bigger time stamp. In addition, besides dynamic control and data dependencies, the slicing algorithm should also consider inter-thread dependencies, such as the dependencies introduced by wait-notify operations.

8. RELATED WORK

In this section, we survey related literature on compressed program trace representations, dynamic slicing and extensions of dynamic slicing for detecting omission errors.

8.1 Work on Dynamic Slicing

Weiser originally introduced the concept of program slicing [Weiser 1984]. In the last two decades, program slicing, in particular dynamic slicing, has been widely used in many software engineering activities, such as program understanding, debugging and testing [Agrawal et al. 1993; Korel and Rilling 1997; Lucia 2001; Tip 1995]. The first dynamic slicing algorithm is introduced by Korel and Laski [Korel and Laski 1988]. In particular, they exploited *dynamic flow concepts* to capture the dependencies between occurrences of statements in the execution trace, and generated executable dynamic slices. Later, Agrawal and Horgan used the *dynamic dependence graph* to compute non-executable but precise dynamic slices, by representing each occurrence of a statement as a distinct node in the graph Agrawal and Horgan 1990; Agrawal 1991]. A survey of program slicing techniques developed in the eighties and early nineties appears in [Tip 1995].

Static and dynamic slicing of object-oriented programs based on dependence graphs have been studied in [Larsen and Harrold 2001] and [Xu et al. 2002] respectively. Computation of control and data dependencies between Java bytecodes has been discussed in [Zhao 2000]. The work of [Ohata et al. 2001] combined dynamic and static slicing to avoid the space overheads of processing traces, at the cost of precision of computed slices. Recently, Zhang et al. [Zhang et al. 2005] studied the performance issues in computing the control/data dependencies from the execution traces for slicing purposes.

In [Dhamdhere et al. 2003], Dhamdhere et al. present an approach for dynamic slicing on compact execution traces. However, they do not employ any data compression algorithm on the execution trace. In fact, they classify execution instances of statements as critical or non-critical, and store only the latest execution instances for non-critical statements. The classification of statements as critical/non-critical is sensitive to the slicing criterion.

Our compression scheme is not related to the slicing criterion and exploits regularity/repetition of control/data flow in the trace. Our slicing algorithm operates directly on this compressed trace achieving substantial space savings at tolerable time overheads.

Extensions of Dynamic Slicing for Detecting Omission Errors. In the past, *relevant slicing* has been studied as an extension of dynamic slicing for the purpose of detecting omission errors in a program [Agrawal et al. 1993; Gyimóthy et al. 1999]. The approach in [Agrawal et al. 1993] relies on the huge dynamic dependence graph. Furthermore, if b is a branch statement with which statements in the slice have potential dependencies, [Agrawal et al. 1993] only computes the closure of data and potential dependencies of b. In other words, control dependencies are ignored w.r.t. statements on which b is data dependent. Gyimóthy proposed an forward relevant slicing algorithm in [Gyimóthy et al. 1999], and it avoided using the huge dynamic dependence graph. However, such an algorithm will compute many redundant dependencies since it is not goal directed. In addition, while computing the dependencies of a later occurrence of certain branch statements (those which appear in the slice due to potential dependencies), the algorithm also includes statements which affect an early occurrence of the same branch statement. In this paper, we define a relevant slice over the *Extended Dynamic Dependence Graph* (EDDG), and it is more accurate than previous ones. We have also presented a space-efficient relevant slicing algorithm which operates *directly* on the compressed bytecode traces.

8.2 Work on Compact Trace Representations

Various compact trace representation schemes have been developed in [Goel et al. 2003; Larus 1999; Pleszkun 1994; Zhang and Gupta 2001; 2004] to reduce the high space overheads of storing and analyzing traces. Pleszkun presented a two-pass trace scheme, which recorded basic block's successors and data reference patterns [Pleszkun 1994]. The organization of his trace is similar to our trace table. However, Pleszkun's technique does not allow traces to be collected on the fly, and the trace is still large because the techniques for exploiting repetitions in the trace are limited. Recently, Larus proposed a compact and analyzable representation of a program's dynamic control flow via the on-line compression algorithm SEQUITUR [Larus 1999]. The entire trace is treated as a single string during compression, but it becomes costly to access the trace of a specific method. Zhang and Gupta suggested breaking the traces into per-method traces [Zhang and Gupta 2001]. However, it is not clear how to efficiently represent data flow in their traces. In a later work [Zhang and Gupta 2004], Zhang and Gupta presented a unified representation of different types of program traces, including control flow, value, address, and dependence.

The idea of separating out the data accesses of load/store instructions into a separate sequence (which is then compressed) is explored in [Goel et al. 2003] in the context of parallel program executions. However, this work uses the SEQUITUR algorithm which is not suitable for representing contiguous repeated patterns. In our work, we have developed RLESe to improve SEQUITUR's space and time efficiency, by capturing contiguous repeated symbols and encoding them with their run-length. RLESe is different from the algorithm proposed by Reiss and Renieris [Reiss and Renieris 2001], since it is an on-line compression algorithm, whereas Reiss and Renieris suggested modifying SEQUITUR grammar rules in a post processing step.

Another approach for efficient tracing is not to trace all bytecodes/instructions during trace collection. This approach is explored by the *abstract execution* technique [Larus 1990], which differs from ours in how to handle untraced instructions. In particular, abstract execution executes a program P to record a small number of significant events, thereby deriving a modified program P'. The program P' then executes with the significant events as the guide; this amounts to re-executing parts of P for discovering information about instructions in P which were not traced. On the other hand, our method records certain bytecodes in an execution as a compressed representation. Post-mortem analysis of this compressed representation does not involve re-execution of the untraced bytecodes. To retrieve information about untraced bytecodes we detect dynamic dependencies via a lightweight flow analysis.

9. DISCUSSION

In this paper, we have developed a space efficient scheme for compactly representing bytecode traces of sequential Java programs. The time overheads and compression efficiency of our representation are studied empirically. We use our compact traces for efficient dynamic slicing. We also extend our dynamic slicing algorithm to explain omission errors (errors arising from omission of a statement's execution due to incorrect evaluation of branches/method invocations). Our method has been implemented on top of the open source Kaffe Virtual Machine. The traces are collected by monitoring Java bytecodes, they are compressed online and then slicing is performed post-mortem by traversing the compressed trace without decompression. We have released our *JSlice* dynamic slicing tool as open-source software for usage in research and development.

APPENDIX

A. COMPLEXITY ANALYSIS OF THE RLESE ALGORITHM

In this appendix, we prove that the RLESe compression algorithm described in Section 2.3 is linear in both space and time.

A.1 Properties preserved by RLESe algorithm

Recall that RLESe constructs a context free grammar to represent a sequence, by preserving three properties w.r.t. the grammar: (1) no contiguous repeated symbols property, (2) digram uniqueness property, and (3) rule utility property (details can be found in Section 2.3). Figure 22 presents the algorithm. The RLESe algorithm proceeds by iteratively reading a symbol from the input sequence (line 2 of Figure 22), appending a node $\langle sym : 1\rangle$ to the end of start rule (line 3 of Figure 22), and re-structuring the grammar by preserving the above three properties (line 4-18 of Figure 22). When the algorithm checks whether any property is violated (lines 5, 7, and 15 of Figure 22), it is sufficient to examine changed nodes or digrams, instead of going through the entire grammar. For example, when the algorithm looks for continuous repeated nodes (line 5 of Figure 22), only nodes which have just been inserted into the grammar are necessary to check. This is particularly important for the efficiency of the algorithm. We explain how to re-construct the grammar when any one of the three properties of RLESe are violated.

The first property (*i.e.* no contiguous repeated symbols property) is preserved by line 5 and 6 of Figure 22. If two nodes $\langle sym : n\rangle$ and $\langle sym : n'\rangle$ are adjacent in the grammar, line 6 merges the two nodes. That is we delete node $\langle sym : n'\rangle$, and change node $\langle sym : n\rangle$ to $\langle sym : n+n'\rangle$. Clearly, this merge operation can save one node in the grammar size.

Lines 7-14 of Figure 22 preserve the second property of RLESe (*i.e.,* the digram uniqueness property). Recall that two digrams are *similar* if their nodes contain the same pair of symbols. Two digrams are *identical* if they have the same pairs of symbols and counters. Line 7 checks whether there are similar digrams in the grammar. If so, the algorithm can obtain two identical digrams, and replace both identical digrams with a nonterminal node for a rule (possibly already in existence) that has the identical digram as its right side. Note that, when there are two similar digrams $\langle sym_1 : n_1, sym_2 : n_2\rangle$, and $\langle sym_1 : n_1', sym_2 : n_2'\rangle$, it may require splitting nodes (line 9 of Figure 22) to obtain identical digrams as $\langle sym_1 : min(n_1, n_1'), sym_2 : min(n_2, n_2')\rangle$, where $min(n_1, n_1')$ is the minimum of n_1 and n_1'. Splitting one node will introduce one more node into the grammar. Line 9 of Figure 22 checks whether one of the two identical digrams is exactly the right side of an existing rule. If so, the algorithm replaces another identical digram with a non-terminal node for the rule (line 11 of Figure 22). We change the grammar rules

$$A \rightarrow \cdots sym_1 : n_1, sym_2 : n_2, \cdots \qquad B \rightarrow sym_1 : n_1, sym_2 : n_2$$

to

$$A \rightarrow \cdots B : 1, \cdots \qquad B \rightarrow sym_1 : n_1, sym_2 : n_2$$

This can save one node in the resultant grammar. If not, a new rule is introduced, and the algorithm replaces both identical digrams with a non-terminal node for the new rule (line 13-14 of Figure 22). That is, we change the grammar rule

$$A \rightarrow \cdots sym_1 : n_1, sym_2 : n_2, \cdots sym_1 : n_1, sym_2 : n_2, \cdots$$

to

$$A \rightarrow \cdots B : 1, \cdots B : 1, \cdots \qquad B \rightarrow sym_1 : n_1, sym_2 : n_2$$

This operation does not introduce more nodes nor save any node, but introduces one more rule in the resultant grammar.

Lines 15-17 of Figure 22 preserve the third property — the rule utility property. RLESe eliminates a rule referenced only once by replacing the reference with the right side of the rule. That is, we change the grammar rules

$$A \rightarrow \cdots B : 1, \cdots \qquad B \rightarrow sym_1 : n_1, sym_2 : n_2$$

to

$$A \rightarrow \cdots sym_1 : n_1, sym_2 : n_2, \cdots$$

This operation can save one node in the resultant grammar.

A.2 Operations in the RLESe algorithm

We proceed to prove that the complexity of the RLESe algorithm in Figure 22 is linear in both space and time w.r.t. the length of input sequence. The proof is similar to the proof for SEQUITUR in [Nevill-Manning and Witten 1997], where we do not put a bound on each operation to re-construct the grammar. Instead we calculate the amortized costs, that is, we obtain a bound on the *total* amount of work done re-constructing the grammar. Our analysis of RLESe also uses an assumption made in SEQUITUR's analysis in [Nevill-Manning and Witten 1997] — given a digram, the average time to look for its similar digram is bounded by a constant. This can be achieved by indexing digrams with a hash table [Knuth 1973].

Table VIII shows variables which will be used later for complexity analysis, as well as descriptions of these variables. The third column shows line number of the RLESe algorithm in Figure 22 in which each variable is used; the last column shows corresponding operations for each line. Clearly, the time to perform each operation in Table VIII is constant. For example, it needs constant time to check whether a specific node has the same symbol with its neighboring nodes (line 5 of Figure 22). Thus, the total time cost to execute line 5 is proportional to m_2, the number of times to check the first property. Some lines of the compression algorithm are always executed together (*e.g.* lines 13 and 14 of Figure 22). They are considered as one operation during complexity analysis, so they are put in the same entry in this table. The size of the RLESe grammar (*i.e.* m) denotes the nodes in the right-hand side of grammar rules, because nodes in the left-hand side of grammar rules can be recreated according to the order in which these rules appear.

A.3 Space Complexity

We derive an equation describing the size of the final grammar. Operations 3,7,10 in Table VIII refer to merging nodes, using an existing rule and removing a rule; these operations

```
1   while (input sequence is not empty)
2       $sym$= read next symbol from input;
3       append node $\langle sym : 1\rangle$ to the end of start rule $s$
4       do
5           if (there are continuous repeated nodes)
6               merge the two nodes;
7           if (there are two similar digrams)
8               if (the two digrams are not identical)
9                   split nodes to get two identical digrams;
10              if (one of the identical digrams is a complete rule)
11                  replace another digram with a non-terminal node for the rule
12              else
13                  create a new rule, where the right side of the rule is the identical digram;
14                  replace both digrams with a non-terminal node for the new rule;
15          if (rule R is referenced only once)
16              replace the use of R with the right side of R;
17              remove the rule R;
18      while (any of the three properties is violated)
```

Fig. 22. The RLESe compression algorithm

Var.	Description	Line in Fig. 22	Operation
n	the size of the input sequence		
m	the number of nodes in the final grammar		
r	the number of rules in the final grammar		
m_1	the number of times to read a symbol from input	2, 3	1: read
m_2	the number of times to check the first property	5	2: check the 1st property
m_3	the number of times to merge two nodes	6	3: merge
m_4	the number of times to check the second property	7	4: check the 2nd property
m_5	the number of times two similar digrams are found	8, 10	5: check digrams
m_6	the number of split nodes	9	6: split
m_7	the number of times an existing rule is used	11	7: use an existing rule
m_8	the number of times a new rule is introduced	13, 14	8: introduce a new rule
m_9	the number of times to check the third property	15	9: check the 3rd property
m_{10}	the number of times a rule is removed	16, 17	10: remove a rule

Table VIII. Operations in the RLESe algorithm

save the number of nodes in the grammar. On the other hand, operation 6 (splitting a grammar node), increases the grammar size. Thus, we get the following equation relating the grammar size (m) and the length of the input sequence (n).

$$n - m = m_3 + m_7 + m_{10} - m_6 \tag{1}$$

In addition, we can only split nodes which have been merged. Note that if a node was not produced by any merging, its run-length must be 1, so the question of splitting does not arise. We get:

$$m_6 \leq m_3 \tag{2}$$

From the two formulas, we can conclude that

$$n - m \geq m_7 + m_{10} > 0 \tag{3}$$

This shows that the size of the final grammar (*i.e.* the variable m) is bound by the length of the input sequence (*i.e.* the variable n), and the algorithm is linear in space.

A.4 Time Complexity

Next, we study the time complexity of the RLESe algorithm. When the algorithm checks the ”no contiguous repeated symbols property”, the first property of RLESe (operation 2), it is sufficient to look at the nodes inserted by operations 1, 7, 8, and 10 (refer Table VIII). That is,

$$m_2 = m_1 + m_7 + 2m_8 + 2m_{10} \tag{4}$$

The coefficient 2 is used because both operations 8 and 10 insert two nodes.

Operations 1, 7, 8, and 10 introduce new digrams which necessitate check of the digram uniqueness property, the second property of RLESe. We have

$$m_4 = m_1 + 2m_7 + 4m_8 + 2m_{10} \tag{5}$$

When a new rule is introduced and two identical diagrams are removed (operation 8), the number of references to a rule may be reduced, and the third property of RLESe (the rule utility property) is checked. Therefore,

$$m_9 = m_8 \tag{6}$$

In addition, the following formulas hold according to the structure of the algorithm,

$$m_3 \le m_2 \quad \text{and} \quad m_5 \le m_4$$

Now, let us look at the total time overhead for the compression algorithm in Figure 22, which is sum of time cost for each operation. The expression for the total time overhead can be simplified as:

$$\begin{aligned} & m_1 + m_2 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8 + m_9 + m_{10} \\ \le\ & m_1 + 3m_2 + 2m_4 + m_7 + 2m_8 + m_{10} \\ \le\ & m_1 + 3(m_1 + m_7 + 2m_8 + 2m_{10}) + 2(m_1 + 2m_7 + 4m_8 + 2m_{10}) + \\ & m_7 + 2m_8 + m_{10} \\ =\ & 6n + 8m_7 + 16m_8 + 11m_{10} \end{aligned} \tag{7}$$

For the number of rules r in the grammar, we have $r = m_8 - m_{10}$ since operation 8 introduces new rules, and operation 10 removes rules. Also, note the right side of each rule has at least two nodes. Thus $r < m$.

The expression for the total time overhead in Formula 7 can then be simplified as

$$\begin{aligned} & m_1 + m_2 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8 + m_9 + m_{10} \\ =\ & 6n + 8m_7 + 16(r + m_{10}) + 11m_{10} \\ <\ & 6n + 16m + 27(m_7 + m_{10}) \end{aligned} \tag{8}$$

Recall from Formula 3

$$0 < m_7 + m_{10} \le n - m < n \tag{9}$$

So the time complexity for the RLESe algorithm in Figure 22 is $O(n)$.

B. ANALYSIS OF THE DYNAMIC SLICING ALGORITHM

In this appendix, we prove the lemmas used in the proof of Theorem 3.5.1, which proves the correctness of the dynamic slicing algorithm in Figure 2.

LEMMA B.0.1. *Let φ_i be the φ set after i loop iterations of the dynamic slicing algorithm in Figure 2. Then $\forall i, j$, $0 < i < j \Rightarrow \varphi_i \subseteq \varphi_j$.*

PROOF. Let β be the bytecode occurrence encountered at the ith loop iteration. According to the algorithm, φ_i=φ_{i-1} or φ_i=$\varphi_{i-1} \cup \{\beta\}$. Thus, for all i we have $\varphi_{i-1} \subseteq \varphi_i$, and the lemma holds. □

LEMMA B.0.2. *Let φ_i be the φ set, and $fram_i$ be the $fram$ after i loop iterations of the dynamic slicing algorithm in Figure 2. Let $fram_i^j$ represents a method invocation in $fram_i$. Then $\forall\beta'$, $\exists fram_i^j \in fram_i$, $\beta' \in fram_i^j.\gamma$, iff. $\beta' \in \varphi_i$ and the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations.*

PROOF. Let $\Gamma_i = \cup_j fram_i^j.\gamma$, *i.e.* the union of γ sets of all method invocations in $fram_i$, after i loop iterations of the dynamic slicing algorithm in Figure 2.

To prove this lemma, it is equivalent to prove: $\beta' \in \Gamma_i$, iff. $\beta' \in \varphi_i$ and the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations. Next we prove this by induction on loop iterations of the slicing algorithm.

$Base$: Initially, φ_0 and Γ_0 are both empty, so the lemma holds.

$Induction$: Assume $\forall\beta'' \in \Gamma_{i-1}$ iff. $\beta'' \in \varphi_{i-1}$ and the algorithm has not found the bytecode occurrence which β'' is dynamically control dependent on after $i-1$ loop iterations. Let β be the bytecode occurrence encountered at the ith loop iteration. According to the algorithm in Figure 2, $\Gamma_i = (\Gamma_{i-1} - \mathcal{C}) \cup \mathcal{O}$, where,

— $\mathcal{C}$ is the set of bytecode occurrences in Γ_{i-1} which are dynamically control dependent on β. Note that if β is a method invocation bytecode occurrence, $\mathcal{C}$= $last\_fram.\gamma$ (line 14 in Figure 2). If β is a branch bytecode occurrence, $\mathcal{C}$= BC (line 23 in Figure 2).

— $\mathcal{O} = \{\beta\}$ iff. $\beta \in \varphi_i$, and $\mathcal{O} = \emptyset$ iff. $\beta \notin \varphi_i$ (lines 32 and 33 in Figure 2).

We first prove the only if part of the lemma. For any $\beta' \in fram_i$,

(1) if $\beta' \in \Gamma_{i-1} - \mathcal{C} \subseteq \Gamma_{i-1}$, $\beta' \in \varphi_{i-1}$ and the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after $i-1$ loop iterations according to the assumption. Lemma B.0.1 shows $\varphi_{i-1} \subseteq \varphi_i$, so $\beta' \in \varphi_i$. Since $\beta' \notin \mathcal{C}$, β' is not dynamically control dependent on β. This means that the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations.

(2) if $\beta' \in \mathcal{O}$ and $\mathcal{O} \neq \emptyset$, then $\beta' = \beta \in \varphi_i$. Clearly, the slicing algorithm has not found the bytecode occurrence $\overline{\beta}$ which β is dynamically control dependent on, because backward traversal has not encountered $\overline{\beta}$, which appears earlier than β during trace collection.

Next, we prove the if part of the lemma. Note that $\varphi_i = \varphi_{i-1}$ or $\varphi_i = \varphi_{i-1} \cup \{\beta\}$ according to the slicing algorithm. For any $\beta' \in \varphi_i$ s.t. the slicing algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations, we need to show that $\beta' \in \Gamma_i$. The following are the two possibilities.

(1) if $\beta' \in \varphi_{i-1}$, then $\beta' \in \Gamma_{i-1}$ according to assumption. Since β' is not dynamically control dependent on β, $\beta' \notin \mathcal{C}$ and $\beta' \in \Gamma_i$.

(2) if $\beta' = \beta$, then $\beta \in \varphi_i$ and $\mathcal{O} = \{\beta\}$. So $\beta' \in \Gamma_i$.

This completes the proof. □

LEMMA B.0.3. *Let φ_i be the φ set, and δ_i be the δ set after i loop iterations of the dynamic slicing algorithm in Figure 2. Then $\forall v$, $v \in \delta_i$ iff. variable v is used by a bytecode occurrence in φ_i and the slicing algorithm has not found any assignment to v after i loop iterations.*

PROOF. We prove the lemma by induction on loop iterations of the slicing algorithm.

$Base$: Initially, φ_0 and δ_0 are both empty, so the lemma holds.

$Induction$: Assume that $\forall v'$, $v' \in \delta_{i-1}$ iff. variable v' is used by a bytecode occurrence in φ_{i-1} and the algorithm has not found any assignment to v' after $i-1$ loop iterations. Let β be the bytecode occurrence encountered at the ith loop iteration. According to the algorithm, $\delta_i = (\delta_{i-1} - def\_vars) \cup use\_vars$, where

— def_vars is the set of variables assigned by β (lines 28 and 29 in Figure 2).

— use_vars is the set of variables used by β iff. $\beta \in \varphi_i$, and use_vars=$\emptyset$ iff. $\beta \notin \varphi_i$. (lines 20, 25, 30, 32 and 34 in Figure 2)

We first prove the only if part of the lemma. For any $v \in \delta_i$,

(1) if $v \in \delta_{i-1} - def\_vars \subseteq \delta_{i-1}$, v is used by a bytecode occurrence in φ_{i-1} and the algorithm has not found any assignment to v after $i-1$ loop iterations according to the assumption. Lemma B.0.1 shows $\varphi_{i-1} \subseteq \varphi_i$. So, v is used by a bytecode occurrence in φ_i. Since $v \notin def\_vars$, v is not defined by β. We can infer that the algorithm has not found any assignment to v after i loop iterations.

(2) if $v \in use\_vars$ and $use\_vars \neq \emptyset$, then v is used by bytecode occurrence β and $\beta \in \varphi_i$. Clearly, the slicing algorithm has not found any assignment to the variable v after i loop iterations, because backward traversal has not encountered these assignments, which appear earlier than β during trace collection.

Next, we prove the if part of the lemma. Note that $\varphi_i = \varphi_{i-1}$ or $\varphi_i = \varphi_{i-1} \cup \{\beta\}$ according to the slicing algorithm. Consider a variable v which is used by a bytecode occurrence in φ_i, and the slicing algorithm has not found any assignment to v after i loop iterations. For such a variable, we have the following two cases.

(1) if v is used by a bytecode occurrence in φ_{i-1}, then $v \in \delta_{i-1}$ according to assumption. Since v is not defined by β, then $v \notin def\_vars$ and $v \in \delta_i$.

(2) if v is used by bytecode occurrence β and $\beta \in \varphi_i$, then $v \in use\_vars$ and $use\_vars \subseteq \delta_i$. Thus, $v \in \delta_i$.

In both cases, we show that $v \in \delta_i$. This completes the proof. □

LEMMA B.0.4. *During dynamic slicing according to the algorithm in Figure 2, a bytecode occurrence β pops an entry from op_stack, which is pushed to op_stack by bytecode occurrence β', iff. β' uses an operand in the operand stack defined by β during trace collection.*

PROOF. The op_stack for slicing is a reverse simulation of the operand stack for computation during trace collection. That is, for every bytecode occurrence β'' encountered during slicing, the slicing algorithm pops entries from (pushes entries to) the op_stack iff. β'' pushes operands to (pops operands from) the operand stack during trace collection — as shown in the `updateOpStack` method in Figure 6. Consequently, a bytecode occurrence β pops an entry from op_stack, and this entry is pushed to op_stack by bytecode occurrence β' during slicing, iff. β defines an operand in the operand stack, and β' uses the operand during trace collection. □

LEMMA B.0.5. *Let φ_i be the φ set after i loop iterations of the dynamic slicing algorithm in Figure 2, and β be the bytecode occurrence encountered at the ith loop iteration. Then $\beta \in \varphi_i - \varphi_{i-1}$ iff. (1) β belongs to the slicing criterion, or, (2) $\exists \beta' \in \varphi_{i-1}$, β' is dynamically control or data dependent on β.*

PROOF. Note that $\beta \notin \varphi_{i-1}$. According to the slicing algorithm, $\beta \in \varphi_i - \varphi_{i-1}$ iff. any of lines 19, 22 and 27 in Figure 2 is evaluated true so that any of lines 21, 26, and 31 in Figure 2 is executed. We next prove that any of lines 19, 22 and 27 in Figure 2 is evaluated true iff. (1) β belongs to the slicing criterion, or, (2) $\exists \beta' \in \varphi_{i-1}$, β' is dynamically control or data dependent on β.

First, line 19 in Figure 2 is evaluated to true iff. β belongs to the slicing criterion.

Next, we prove that line 22 in Figure 2 is evaluated to true iff. $\exists \beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β. According to the slicing algorithm, the check *computeControlDependence*(b_β, $curr\_fram$, $last\_fram$) in line 22 of the dynamic slicing algorithm (see Figure 2) returns true iff:

— β is a branch bytecode occurrence, and $\exists \beta' \in curr\_fram.\gamma$, $curr\_fram \in fram_{i-1}$ β' is dynamically control on β, or

— β is a method invocation bytecode occurrence, and $\exists \beta' \in last\_fram.\gamma$, $last\_fram \in fram_{i-1}$, β' is dynamically control on β,

According to Lemma B.0.2, $\forall \beta'$, $\exists fram^j_{i-1} \in fram_{i-1}$, $\beta' \in fram^j_{i-1}.\gamma$ only if $\beta' \in \varphi_{i-1}$. So, line 22 returns true only if $\exists \beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β.

On the other hand, if $\exists \beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β, then the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after $i-1$ loop iterations, because every bytecode occurrence is dynamically control dependent on exactly one bytecode occurrence. So, $\exists fram^j_{i-1} \in fram_{i-1}$ $\beta' \in fram^j_{i-1}.\gamma$, according to Lemma B.0.2. If β is a branch bytecode occurrence, then $\beta' \in curr\_fram.\gamma$, $curr\_fram \in fram_{i-1}$, since β and β' should belong to the same method invocation. If β is a method invocation bytecode occurrence, then $\beta' \in last\_fram.\gamma$, $last\_fram \in fram_{i-1}$, since β' should belong to last method invocation, which is called by β. So line 22 in Figure 2 returns true if $\exists \beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β.

Finally, we prove that line 27 in Figure 2 is evaluated to true iff $\exists \beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β. Note that line 27 invokes the *computeDataDependence* method defined in Figure 7 to check dynamic data dependence. The check *computeDataDependence*(β, b_β) returns true iff either of the following conditions holds:

— if β defines a variable in δ_{i-1} (line 9 of Figure 7), where δ_{i-1} represents the δ set after $i-1$ loop iterations.

— if one of the top $def\_op(b_\beta)$ entries of the op_stack is pushed by a bytecode occurrence $\beta' \in \varphi_{i-1}$ (line 12 of Figure 7), where $def\_op(b_\beta)$ is the number of operands defined by bytecode b_β of occurrence β during trace collection.

When the `computeDataDependence` method returns true: (a) if β defines a variable $v \in \delta_{i-1}$, then $\exists \beta' \in \varphi_{i-1}$, v is used by β' and the algorithm has not found any assignment to v after $i-1$ loop iterations according to Lemma B.0.3. So β' is dynamically data dependent on β. (b) if one of the top $def\_op(b_\beta)$ entries of the op_stack is pushed by a bytecode occurrence $\beta' \in \varphi_{i-1}$. Because all the top $def\_op(b_\beta)$ entries of the op_stack will be popped by β (lines 2 and 3 of method `updateOpStack` in Figure 6), β' uses an operand in the operand stack defined by β during trace collection according to Lemma B.0.4. Consequently, β' is dynamically data dependent on β. This proves that line 27 in Figure 2 is evaluated to true, only if $\exists \beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β.

On the other hand, if $\exists \beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β, then either (a) $\exists v, \beta'$, $\beta' \in \varphi_{i-1}$, v is used by β' and v is defined by β. According to Lemma B.0.3, $v \in \delta_{i-1}$; so the `computeDataDependence` method returns true and line 27 in Figure 2 is evaluated to true. (b) $\exists \beta'$, $\beta' \in \varphi_{i-1}$, β' uses an operand in the operand stack defined by β during trace collection. According to Lemma B.0.4, β should pop an entry from op_stack, which is pushed into op_stack by β'. Since β pops top $def\_op(b_\beta)$ entries from the op_stack, line 12 in Figure 7 is evaluated to true, and the `computeDataDependence` method returns true. This proves that line 27 in Figure 2 is evaluated to true, if $\exists \beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β. □

C. ANALYSIS OF THE RELEVANT SLICING ALGORITHM

In this appendix, we prove the correctness of the relevant slicing algorithm in Figure 11.

LEMMA C.0.6. *Let φ_i be the φ set after i loop iterations of the relevant slicing algorithm in Figure 11. Then $\forall i, j$, $0 < i < j \Rightarrow \varphi_i \subseteq \varphi_j$.*

PROOF. Proof of this lemma is the same as the proof of Lemma B.0.1 in Appendix B, for the dynamic slicing algorithm. □

LEMMA C.0.7. *Let φ_i be the φ set, and $fram_i$ be the $fram$ set after i loop iterations of the relevant slicing algorithm in Figure 11. Let $fram_i^j$ represents a method invocation in $fram_i$. Then $\forall \beta$, $\exists fram_i^j \in fram_i$, $\beta' \in fram_i^j.\gamma$ iff. (1) $\beta \in \varphi_i$, and (2) β belongs to slicing criterion or $\exists \beta' \in \varphi_i$ s.t. β' is dynamically control/data dependent on β, and (3) the algorithm has not found the bytecode occurrence which β is dynamically control dependent on after i loop iterations.*

PROOF. Proof of this lemma is the similar to the proof of Lemma B.0.2 in Appendix B, for the dynamic slicing algorithm. □

LEMMA C.0.8. *Let φ_i be the φ set, and δ_i be the δ set after i loop iterations of the relevant slicing algorithm in Figure 11. Then $\forall v$, $v \in \delta_i$ iff. (1) variable v is used by a bytecode occurrence $\beta \in \varphi_i$ s.t. (a) β belongs to slicing criterion, or (b) $\exists \beta' \in \varphi_i$ s.t. β' is dynamically control/data dependent on β, and (2) the algorithm has not found any assignment to v after i loop iterations.*

PROOF. Proof of this lemma is the similar to the proof of Lemma B.0.3 in Appendix B, for the dynamic slicing algorithm. □

LEMMA C.0.9. *Let φ_i be the φ set, and θ_i be the θ set after i loop iterations of the relevant slicing algorithm in Figure 11. Then $\forall v$, $\exists prop, v \in prop$, and $\langle \_\beta', prop\rangle \in \theta_i$ iff. (1) variable v is used by a bytecode occurrence $\beta \in \varphi_i$, where (a) β does not belong to slicing criterion, and (b) there is no $\beta' \in \varphi_i$ s.t. β' is dynamically control/data dependent on β, and (2) the algorithm has not found any assignment to v after i loop iterations.*

PROOF. The proof of this lemma is similar to the proof of Lemma C.0.8. □

Indeed, the δ set (in Lemma C.0.8) includes variables used by bytecode occurrences β s.t. β is added into φ when (1) β belongs to the slicing criterion, or (2) there is any bytecode occurrence in φ which is *dynamically control/data dependent* on β. On the other hand, the $prop$ sets of θ (in Lemma C.0.9) includes variables used by bytecode occurrences β s.t. β is added into φ when (1) there is any bytecode occurrence in φ which is *potentially dependent* on β, and (2) β does not belong to slicing criterion, and no bytecode occurrence in φ is dynamically control/data dependent on β.

LEMMA C.0.10. *During relevant slicing according to the algorithm in Figure 11, a bytecode occurrence β pops an entry from op_stack, which is pushed to op_stack by bytecode occurrence β, iff. β uses an operand in the operand stack defined by β during trace collection.*

PROOF. Proof of this lemma is the same as the proof of Lemma B.0.4 in Appendix B, for the dynamic slicing algorithm. □

LEMMA C.0.11. *Let φ_i be the φ set after i loop iterations of the relevant slicing algorithm in Figure 11, and β be the bytecode occurrence encountered at the ith loop iteration. Then $\beta \in \varphi_i - \varphi_{i-1}$ iff.*

(*1*) *β belongs to the slicing criterion, or,*

(*2*) *$\exists\beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β, and β' was not introduced into the relevant slice φ because of potential dependencies.*[2]

(*3*) *$\exists\beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β, or*

(*4*) *none of above three conditions is satisfied, and $\exists\beta' \in \varphi_{i-1}$, β' is potentially dependent on β.*

PROOF. Note that $\beta \notin \varphi_{i-1}$. According to the slicing algorithm, $\beta \in \varphi_i - \varphi_{i-1}$ iff. any of lines 21, 26, 31 and 39 in Figure 11 is executed. Further,

I. line 21 in Figure 11 is executed iff. condition (1) in this lemma holds, which checks the slicing criterion.

II. line 26 in Figure 11 is executed iff. condition (2) in this lemma holds, which checks dynamic control dependencies.

III. line 31 in Figure 11 is executed iff. condition (3) in this lemma is satisfied, which checks dynamic data dependencies.

IV. line 39 in Figure 11 is executed iff. condition (4) in this lemma is satisfied, which checks potential dependencies.

[2] In other words, either there exists a bytecode β" $\in \varphi_{i-1}$ which is dynamically data/control dependent on β', or β' belongs to the slicing criterion.

Proofs of I, II and III are similar to proof of Lemma B.0.5 in Appendix B, for the dynamic slicing algorithm.

Next, we prove IV., that is line 39 in Figure 11 is executed iff. condition (4) in this lemma is satisfied. According to the slicing algorithm, line 39 in Figure 11 is executed iff. line 34 in Figure 11 is evaluated to false and line 38 in Figure 11 is evaluated to true. Note that line 34 in Figure 11 is evaluated to false iff. lines 19, 22, and 27 are all evaluated to false, which are equivalent to that none of conditions (1) (2) and (3) of this lemma holds. Note that line 38 invokes the *computePotentialDependence* method defined in Figure 12 to check potential dependencies. The check *computePotentialDependence*(β, b_β) returns true iff. either of following conditions holds:

(1) line 6 in Figure 12 is evaluated to true, or

(2) line 9 in Figure 12 is evaluated to true.

We first prove that the *computePotentialDependence* method returns true only if $\exists \beta' \in \varphi_{i-1}$, β' is potentially dependent on β, assuming that line 34 in Figure 11 is evaluated to false. We have the following two cases:

(1) there exists $v \in \delta_{i-1}$ which may be defined by evaluating the branch bytecode occurrence β differently. The β refers to the bytecode occurrence encountered at the ith loop iteration of the relevant slicing algorithm. According to Lemma C.0.8, $\exists \beta' \in \varphi_{i-1}$, v is used by β'. So, β' is potentially dependent on β.

(2) there exists $v, prop''$, $v \in prop''$, $\exists \langle \_\beta'', prop'' \rangle \in \theta_{i-1}$, and v may be defined by evaluating the branch bytecode occurrence β differently. According to Lemma C.0.9, $\exists \beta' \in \varphi_{i-1}$, v is used by β'. So, β' is potentially dependent on β.

In both cases, there exists one bytecode occurrence in φ_{i-1} which is potentially dependent on β.

Now we prove that the *computePotentialDependence* method returns true if $\exists \beta' \in \varphi_{i-1}$, β' is potentially dependent on β, assuming that line 34 in Figure 11 is evaluated to false. The following are two possibilities:

(1) there exists v used by a bytecode occurrence $\beta' \in \varphi_{i-1}$, where β' was not introduced into the relevant slice φ because of potential dependencies. According to Lemma C.0.8, $v \in \delta_{i-1}$. So line 7 of Figure 12 is executed and the *computePotentialDependence* method returns true.

(2) there exists v used by a bytecode occurrence $\beta' \in \varphi_{i-1}$, where β' was introduced into the relevant slice φ because of potential dependencies. According to Lemma C.0.9, $\exists prop''$, $v \in prop''$, and $\exists \langle \beta'', prop'' \rangle \in \theta_{i-1}$. According to the algorithm, β' is (transitively) dynamically control dependent on β'', so β'' is not dynamically control dependent on β. Thus, line 10 of Figure 12 is executed and the *computePotentialDependence* method returns true.

The completes our proof that the *computePotentialDependence* method returns true if $\exists \beta' \in \varphi_{i-1}$, β' is potentially dependent on β, assuming that line 34 in Figure 11 is evaluated to false. Consequently, line 39 in Figure 11 is executed iff. condition (4) in this lemma is satisfied.

In all cases, we have shown that any of lines 21, 26, 31 and 39 in Figure 11 is executed iff. any of the four conditions in the lemma is satisfied. Consequently, the lemma holds. □

Finally, we prove the correctness of the relevant slicing in Figure 11. Note that the relevant slice defined in Definition 4.2.1 is based on the *Extended Dynamic Dependence Graph* (EDDG). In the EDDG, two nodes in the graph may refer to the same bytecode occurrence. In the following, we use $nn(\beta)$ to represent the *non-dummy node* for bytecode occurrence β in the EDDG, and $dn(\beta)$ to represent corresponding *dummy node* for bytecode occurrence β. Two nodes of the same bytecode occurrence do not contribute to relevant slice together. This is because in the EDDG, non-dummy nodes only have incoming edges representing dynamic control/data dependencies, and dummy nodes only have incoming edges representing potential dependencies. Further, the relevant slicing algorithm includes a bytecode occurrence β into the slice φ when $\exists\beta' \in \varphi$ s.t. β' is dependent on β for *any* of dynamic control, dynamic data and potential dependencies.

THEOREM C.0.1. *Given a slicing criterion, the relevant slicing algorithm in Figure 11 returns relevant slice defined in Definition 4.2.1.*

PROOF. Let φ_i be the φ set after i loop iterations of the relevant slicing algorithm in Figure 11, φ_* be the resultant φ set when the algorithm finishes, and β be the bytecode occurrence encountered at the ith loop iteration. As mentioned in the above, there may be two nodes $nn(\beta')$ and $dn(\beta')$ for a bytecode occurrence β' in the EDDG. So, we will prove this lemma by showing: φ_*={$\beta'|nn(\beta')$ or $dn(\beta')$ is reachable from the slicing criterion in the EDDG}.

We first prove the soundness of the algorithm, *i.e.* for any $\beta', \beta' \in \varphi_*$, only if either $nn(\beta')$ or $dn(\beta')$ is reachable from the slicing criterion in the EDDG. In particular, we prove that: $\forall\beta' \in \varphi_*$, (a) if β' is added into φ_* because of slicing criterion or dynamic control/data dependencies, then $nn(\beta')$ is reachable from the slicing criterion in the EDDG, and (b) if β' is added into φ_* because of potential dependencies, then $dn(\beta')$ is reachable from the slicing criterion in the EDDG. We prove this by induction on loop iterations of the slicing algorithm. Initially, $\varphi_0 = \emptyset$, so the base case holds.

$Induction$: Assume that for any $\beta'' \in \varphi_{i-1}$, (a) if β'' is added into φ_{i-1} because of slicing criterion or dynamic control/data dependencies, then $nn(\beta'')$ is reachable from the slicing criterion in the EDDG, and (b) if β'' is added into φ_{i-1} because of potential dependencies, then $dn(\beta'')$ is reachable from the slicing criterion in the EDDG.

Note that $\varphi_i = \varphi_{i-1}$, or $\varphi_i = \varphi_{i-1} \cup \{\beta\}$. Then, $\forall\beta' \in \varphi_i$, we have two cases:

(1) if $\beta' \in \varphi_{i-1}$, the induction hypothesis still hosts, since $\varphi_{i-1} \subseteq \varphi_i$ according to Lemma C.0.6.

(2) if $\beta' = \beta$, where β is the bytecode occurrence encountered at the ith loop iteration of the slicing algorithm, then $\beta \in \varphi_i - \varphi_{i-1}$. According to Lemma C.0.11, we have following four possibilities to add β into φ_i:

   I. if β belongs to the slicing criterion,then clearly $nn(\beta)$ belongs to slicing criterion,

   II. if $\exists\beta'' \in \varphi_{i-1}$, β'' is dynamically control dependent on β, and β'' was not added into the relevant slice because of potential dependencies, then $nn(\beta'')$ is reachable from the slicing criterion in the EDDG according to the induction hypothesis. In addition, there is an dynamic control dependence edge from $nn(\beta'')$ to $nn(\beta)$ in the EDDG. Thus, $nn(\beta)$ can be reached from the slicing criterion.

   III. $\exists\beta'' \in \varphi_{i-1}$, β'' is dynamically data dependent on β, then either $nn(\beta'')$ or $dn(\beta'')$ is reachable from the slicing criterion according to the induction hypothesis. In the EDDG, there are dynamic data dependence edges from $nn(\beta'')$ to

$nn(\beta)$, and from $dn(\beta'')$ to $nn(\beta)$. Thus, $nn(\beta)$ can be reached from the slicing criterion.

IV. $\exists \beta'' \in \varphi_{i-1}$, β'' is potentially dependent on β, then either $nn(\beta'')$ or $dn(\beta'')$ can be reached from the slicing criterion according to the induction hypothesis. In the EDDG, there are potential dependence edges from $nn(\beta'')$ to $dn(\beta)$, and from $dn(\beta'')$ to $dn(\beta)$. Thus, $dn(\beta)$ can be reached from the slicing criterion.

In all four cases, we show that (a) if β is added into φ_i because of slicing criterion or dynamic control/data dependencies, then $nn(\beta)$ is reachable from the slicing criterion in the EDDG, and (b) if β is added into φ_i because of potential dependencies, then $dn(\beta)$ is reachable from the slicing criterion in the EDDG.

Next, we prove the completeness of the slicing algorithm, *i.e.* for any $\beta', \beta' \in \varphi_*$, if either $nn(\beta')$ or $dn(\beta')$ is reachable from the slicing criterion in the EDDG. Note that there is no cycle in the EDDG, so we prove the completeness by induction on structure of the EDDG.

$Base$: Consider a bytecode occurrence β' where β' belongs to the slicing criterion. Clearly, $nn(\beta')$ is reachable from the slicing criterion in the EDDG. Let β' be encountered at the ith loop iteration of the slicing algorithm. By Lemma C.0.11 & C.0.6, $\beta' \in \varphi_i \subseteq \varphi_*$.

$Induction$: Assume that a set of bytecode occurrences $\beta'' \in \varphi_*$, which satisfy (1) if $nn(\beta'')$ is reachable from the slicing criterion in the EDDG, β'' is added into the relevant slice φ_* because of slicing criterion or dynamic control/data dependencies, and (2) if $nn(\beta'')$ is not reachable and $dn(\beta'')$ is reachable from the slicing criterion, then β'' is added into the relevant slice φ_* because of potential dependencies.

Consider a bytecode occurrence β, which can be reached from the slicing criterion by traversing only nodes of bytecode occurrences in φ_*. Clearly, $\exists \beta' \in \varphi_*$, β is dynamically control, or dynamically data, or potentially dependent on β'. Let β be encountered at the ith loop iteration of the algorithm, and β' be encountered at the jth loop iteration of the algorithm. Because β appears earlier than β' during trace collection, backward traversal of the trace will encounter β after β', *i.e.* $j < i$. Thus, $\beta' \in \varphi_j \subseteq \varphi_{i-1}$ according to Lemma C.0.6. We now show that $\beta \in \varphi_i$ according to the relevant slicing algorithm. In particular, (1) if $nn(\beta)$ is reachable from the slicing criterion in the EDDG, then β is added into the slice because of slicing criterion, or dynamic control/data dependencies, and (2) if $nn(\beta)$ is not reachable and $dn(\beta)$ is reachable from the slicing criterion, then β is added into the slice because of potential dependencies. Note that the relevant slicing algorithm check dynamic control/data, and potential dependencies in order. The following are three possibilities:

I. if (1) there is a dynamic control dependence edge from $nn(\beta')$ to $nn(\beta)$, and (b) $nn(\beta')$ is reachable from the slicing criterion, then β' is added into the relevant slice φ_* because of slicing criterion or dynamic control/data dependencies, according to the induction hypothesis. Thus, $\beta \in \varphi_i$ and β is added into the relevant slice φ_* because of dynamic control dependencies, since condition (2) of Lemma C.0.11 is satisfied.

II. if (a) condition of case I does not hold, and (b) there is a dynamic data dependence edge from either $nn(\beta')$ ($dn(\beta')$) to $nn(\beta)$, and (c) $nn(\beta')$ ($dn(\beta')$) is reachable from the slicing criterion. Note that β' is in the slice. So $\beta \in \varphi_i$ and β is added into the relevant slice φ_* because of dynamic data dependencies, since condition (3) of Lemma C.0.11 is satisfied.

III. if (a) conditions of cases I-II do not hold, and (2) there is a potential dependence edge from $nn(\beta')$ ($dn(\beta')$) to $dn(\beta)$, and (c) $nn(\beta')$ ($dn(\beta')$) is reachable from the slicing criterion. Note that β' is in the slice, so:
- —$nn(\beta)$ is not reachable (due to the conditions for cases I-II not being true) and $dn(\beta)$ is reachable from slicing criterion
- —β is added into the relevant slice φ_* because of potential dependencies, and $\beta \in \varphi_i$ since condition (4) of Lemma C.0.11 is satisfied.

In all possible cases, (1) if $nn(\beta)$ is reachable from the slicing criterion in the EDDG, then β is added into the slice because of slicing criterion, or dynamic control/data dependencies, and (2) if $nn(\beta)$ is not reachable and $dn(\beta)$ is reachable from the slicing criterion, then β is added into the slice because of potential dependencies. Consequently, $\beta \in \varphi_i \subseteq \varphi_*$. This completes the proof. □

ACKNOWLEDGMENTS

This work was partially supported by a Public Sector research grant from the Agency of Science, Technology and Research (A*STAR) Singapore.

REFERENCES

AGRAWAL, H. 1991. Towards automatic debugging of computer programs. Ph.D. thesis, Purdue University.

AGRAWAL, H. AND HORGAN, J. 1990. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

AGRAWAL, H., HORGAN, J., KRAUSER, E., AND LONDON, S. 1993. Incremental regression testing. In *International Conference on Software Maintenance (ICSM)*. 348–357.

AKGUL, T., MOONEY, V., AND PANDE, S. 2004. A fast assembly level reverse execution method via dynamic slicing. In *International Conference on Software Engineering (ICSE)*.

ANDERSEN, L. O. 1994. Program analysis and specialization for the c progranzming language. Ph.D. thesis, University of Copenhagen.

BERK, E. J. AND ANANIAN, C. S. A lexical analyzer generator for Java. website: `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

DHAMDHERE, D., GURURAJA, K., AND GANU, P. 2003. A compact execution history for dynamic slicing. *Information Processing Letters 85*, 145–152.

FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9,* 3, 319–349.

GOEL, A., ROYCHOUDHURY, A., AND MITRA, T. 2003. Compactly representing parallel program executions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 191–202.

GYIMÓTHY, T., BESZÉDES, A., AND FORGÁCS, I. 1999. An efficient relevant slicing method for debugging. In *7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 303–321.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12,* 1, 26–60.

JGF. The Java Grande Forum Benchmark Suite. website: `http://www.epcc.ed.ac.uk/javagrande/seq/contents.html`.

JOY, B., STEELE, G., GOSLING, J., AND BRACHA, G. 2000. *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Pub Co.

KAFFE. The kaffe virtual machine. website: `http://www.kaffe.org`.

KNUTH, D. E. 1973. *The Art of Comuter Programming 4: searching and sorting*. Addison-Wesley Pub Co.

KOREL, B. AND LASKI, J. W. 1988. Dynamic program slicing. *Information Processing Letters 29,* 3, 155–163.

KOREL, B. AND RILLING, J. 1997. Application of dynamic slicing in program debugging. In *International Workshop on Automatic Debugging*.

LAMPORT, L. 1997. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 558–565.

LARSEN, L. AND HARROLD, M. 2001. Slicing object-oriented software. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*. 495–505.

LARUS, J. 1990. Abstract execution: A technique for efficiently tracing programs. *Software - Practice and Experience (SPE) 20,* 1241–1258.

LARUS, J. R. 1999. Whole program paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 259–269.

LEVROUW, L. J., AUDENAERT, K. M. R., AND CAMPENHOUT, J. M. 1994. A new trace and replay system for shared memory programs based on lamport clocks. In *The second Euromicro Workshop on Parallel and Distributed Processing*.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley Pub Co.

LUCIA, A. D. 2001. Program slicing: Methods and applications. In *IEEE International Workshop on Source Code Analysis and Manipulation*. 142–149.

MAJUMDAR, R. AND JHALA, R. 2005. Path slicing. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*.

NEVILL-MANNING, C. G. AND WITTEN, I. H. 1997. Linear-time, incremental hierarchy inference for compression. In *Data Commpression Conference (DCC)*. 3–11.

OHATA, F., HIROSE, K., FUJII, M., AND INOUE, K. 2001. A slicing method for object-oriented programs using lightweight dynamic information. In *Asia-Pacific Software Engineering Conference*.

PLESZKUN, A. R. 1994. Techniques for compressing programm address traces. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 32–39.

REISS, S. P. AND RENIERIS, M. 2001. Encoding program executions. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 221–230.

RONSSE, M. AND BOSSCHERE, K. D. 1999. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS) 17,* 133–152.

SAZEIDES, Y. 2003. Instruction isomorphism in program execution. *Journal of Instruction-Level Parallelism 5*.

SINHA, S. AND HARROLD, M. J. 2000. Analysis and testing of programs with exceptionhandling constructs. *IEEE Transactions on Software Engineering 26,* 9, 849–871.

SPECJVM98. Spec JVM98 benchmarks. website: `http://www.specbench.org/osg/jvm98/`.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.

TIP, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 3,* 3, 121–189.

WANG, T. AND ROYCHOUDHURY, A. 2004. Using compressed bytecode traces for slicing Java programs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 512–521. Available from `http://www.comp.nus.edu.sg/~abhik/pdf/icse04.pdf`.

WEISER, M. 1984. Program slicing. *IEEE Transactions on Software Engineering 10,* 4, 352–357.

XU, B., CHEN, Z., AND YANG, H. 2002. Dynamic slicing object-oriented programs for debugging. In *IEEE International Workshop on Source Code Analysis and Manipulation*.

ZHANG, X. AND GUPTA, R. 2004. Whole execution traces. In *IEEE/ACM International Symposium on Microarchitecture (Micro)*. 105–116.

ZHANG, X., GUPTA, R., AND ZHANG, Y. 2005. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

ZHANG, Y. AND GUPTA, R. 2001. Timestamped whole program path representation and its applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 180–190.

ZHAO, J. 2000. Dependence analysis of Java bytecode. In *IEEE Annual International Computer Software and Applications Conference*. 486–491.

ZILLES, C. B. AND SOHI, G. 2000. Understanding the backward slices of performance degrading instructions. In *International Symposium on Computer Architecture (ISCA)*.