Using Compressed Bytecode Traces for Slicing Java Programs

Tao Wang Abhik Roychoudhury
School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543.
{wangtao,abhik}@comp.nus.edu.sg

Abstract

Dynamic slicing is a well-known program debugging technique. 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. Dynamic slicing algorithms often proceed by traversing the execution trace of P produced by input I (or a dependence graph which captures control / data flow in the execution trace). Consequently, it is important to develop space efficient representations of the execution trace.

*In this paper, we use results from data compression to compactly represent bytecode traces of sequential Java programs. The major space savings come from the optimized representation of data (instruction) addresses used by memory reference (branch) bytecodes as operands. We give detailed experimental results on the space efficiency and time overheads for our compact trace representation. We then show how dynamic slicing algorithms can directly traverse our compact traces without resorting to costly decompression. We also develop an extension of dynamic slicing which allows us to explain omission errors (*i.e. *why some events did not happen during program execution).*

1. Introduction

Program slicing 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. A survey of program slicing techniques appears in [20].

Slicing techniques are divided into two categories: static and dynamic. Static (Dynamic) slicing computes the fragment affecting V at l (some occurrences 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 dependences [7]. Dynamic slicing algorithms (w.r.t. an input I) on the other hand, often proceed by collecting the execution trace corresponding to I. The data and control dependences between the statement occurrences in the execution trace can be pre-computed or computed on demand (during slicing) [23].

Clearly, the representation of execution traces is important for dynamic slicing. In practice, traces tend to be huge; [23] reports experiences in dynamic slicing programs like `gcc` and `perl` where the execution trace runs into *several hundred million instructions*. Consequently, it is necessary 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. Furthermore, we want to traverse the execution trace (to retrieve control and data dependences for slicing) without decompressing the trace. In other words, the program trace should be collected, stored and analyzed (for slicing) – all in its compressed form.

In this paper, we describe a dynamic slicing technique for Java programs which operates on compact bytecode traces. First, the bytecode stream corresponding to an execution trace of a Java program is compactly represented. We then perform a backwards traversal of the compressed program trace to compute data/control dependences on-the-fly. The slice is updated as these dependences are encountered during trace traversal.

The compactness of our trace representation is owing to several factors. First, bytecodes which do not correspond to memory read / write (*i.e.* data transfer to and from the

heap) or control transfer are not stored in the trace. These bytecodes can be ignored for computing control and data dependences. Secondly, the sequence of addresses used by each memory reference / 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 [15] for this purpose.

Concretely, the contributions of this paper are as follows.

- We develop a space efficient representation of the bytecode stream for a single threaded Java program execution. We have used the Java Grande benchmarks to measure the compression ratio achieved by this representation; for most benchmarks we obtain 100-1000 times compression. We also show that the time overheads for constructing this representation on-the-fly during program execution is not high. Our experimental results show that the crucial factor leading to compression of program traces is the separation of address sequences used by conditional jump and memory reference instructions.
- Our dynamic slicing algorithm operates *directly* on the compressed program traces. This is an important feature of our work and contrasts with the approaches of [2, 10, 23]. We traverse the program traces in compression domain to retrieve dependences on demand (whose transitive closure is then computed).
- We also enhance our dynamic slicing algorithm to capture "omission errors". Traditional dynamic slicing algorithms explain the values of variables V at a control location l, by highlighting the executed program statements which affect the values of V at l. Our extended slicing algorithm also captures certain statements whose unintended omission affects V at l.

The rest of this paper is organized as follows. The next section describes our compressed representation of a Java bytecode stream. Section 3 reports the space efficiency and time overheads of our compressed trace representation. Section 4 presents our slicing algorithm which proceeds by traversing the compact bytecode traces. Extensions to handle omission errors are also discussed in this section. Section 5 presents related work on program slicing and compact trace representations. Section 6 concludes the paper with other applications of our compact traces.

2. Compact bytecode traces

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. Note that 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 bytecodes executed. Part of the reason for collecting traces at the level of bytecode (instead of program statements) is the added flexibility in tracing/not tracing certain bytecodes. For example, most trace based analysis techniques concentrate on control flow or memory access behavior. Hence computation bytecodes do not need to be traced. Unfortunately, representing a Java program trace as a bytecode sequence does not allow us to capture much of the repetitions in the trace. In particular, a linear representation of the program trace as a single string loses out structure in the following ways.

- The individual methods executed are not separated in the trace representation.
- Sequence of addresses accessed by individual memory load/store bytecodes are not separated out. These sequences capture data flow and exhibit high regularity (*e.g.* a read bytecode sweeping through an array).
- Similarly, sequence of target addresses accessed by control transfer bytecodes are not separated out. Again these sequences show fair amount of repetition (*e.g.* a loop branch repeats the same target many times).

In our representation, the compact trace of the whole program consists of trace tables, each of which is used for one method. The initial method executed is clearly marked. Subsequent method invocations are captured by tracing of bytecodes which invoke methods. The JVM may invoke some methods automatically when "special" events occur (*e.g* it may invoke the method to initialize static fields of a class automatically when a static field of the class is first accessed). We do not record these calls. Instead we can find out such invocations from the occurrences of the "special" events in the trace. Within the trace table for a method, each slot in the trace table maintains the run-time information for a specific bytecode. Monitoring and tracing every bytecode may incur too much time and space overhead. We trace only the following three kinds of bytecodes.

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 every *invokevirtual* and *invokeinterface* bytecode should record the classes which the invoked methods belong to.

Memory access bytecodes: The bytecodes to access local variables and class static fields are not traced since the addresses accessed by these bytecodes can be obtained from

```
class Demo
{
    public int foo(int i){
        if ( i % 2 == 1 )
            return 2;
        else
            return 5;
    }

    static public void main (String argvs[]){
        int a[], i;
        Demo obj= new Demo();

        a= new int[10];
        for (i=0; i < 10; i++)
            a[i]= obj.foo(i);
     }
}
```

```
Method void main(String[])
   0 new Class Demo
   3 dup
   4 invokespecial Demo()
   7 astore_3
   8 bipush 10
  10 newarray int
  12 astore_1
  13 iconst_0
  14 istore_2
  15 goto 29
  18 aload_1
  19 iload_2
  20 aload_3
  21 iload_2
  22 invokevirtual foo(int)
  25 iastore
  26 iinc 2 1
  29 iload_2
  30 bipush 10
  32 if_icmplt 18
  35 return
```

```
Method Demo()
   0 aload_0
   1 invokespecial Object()
   4 return

Method int foo(int)
   0 iload_1
   1 iconst_2
   2 irem
   3 iconst_1
   4 if_icmpne 9
   7 iconst_2
   8 ireturn
   9 iconst_5
  10 ireturn
```

Figure 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	Trace Sequences
22	(D, D, ..., D)
25	(X, X, ..., X) (0, 1, 2, ..., 9)
32	(18, 18, ..., 18, 35)

Table 1. Trace table for method *main(String[])*

Bytecode	Trace Sequences
4	(9, 7, ..., 9, 7)

Table 2. Trace table for method *foo(int)*

the class file. For bytecodes accessing object fields / array elements, we trace the addresses (or identifiers since an address may be used by different variables) corresponding to the bytecode operands.

Branch bytecodes: Each conditional branch bytecode should record which bytecode is executed immediately after the branch bytecode (*i.e.* the target address).

Example: The left part of Figure 1 presents a simple Java program, and the right part shows the corresponding bytecode stream. Table 1 and 2 show the trace table for method *main* and *foo*, respectively. Here D is the identifier of class $Demo$, and X represents the array object a. Note that for the array write bytecode `iastore` two sequences are stored. These correspond to the two operands of the bytecode, namely: the array object identifier and the array element index. Method *Demo* has no trace table, since no bytecode inside this method is traced. 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 outcomes of branch bytecodes within the method. These different traces are all stored implicitly via the sequence of target addresses accessed by the branch bytecodes. As an example, consider the trace table of method *foo* shown in Table 2. The different traces of *foo* result from the different outcomes of its only conditional branch (which is represented by bytecode 4 in Figure 1). Hence the different paths taken in different executions of *foo* are captured by the sequence of target addresses of this conditional branch (which is shown in the trace table, see Table 2).

2.2. Compression of operand sequences

So far, we have described how the bytecode operand sequences representing control flow (target addresses of a branch), data flow (memory accesses of a load/store), or dynamic call graph (class identifiers of invoked methods) 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 [15] which has been used to represent control flow information in program traces [13]. First we briefly describe SEQUITUR.

SEQUITUR The SEQUITUR algorithm represents a finite string σ as a context free grammar whose language is the singleton set $\{\sigma\}$. It reads symbols one-by-one from the input string 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 the string $abcabc$. After reading the first four symbols, the grammar consists of the single production rule $S \rightarrow abca$ (where S is the start symbol). On reading the fifth symbol, it becomes $S \rightarrow abcab$. Since the adjacent symbols ab appear twice in this rule (violating the first invariant), SEQUITUR introduces a non-terminal to get $\{S \rightarrow AcA, A \rightarrow ab\}$. Note that here the rule defining non-terminal A is used twice. Finally, on reading the last symbol of the string $abcabc$ the above grammar becomes $\{S \rightarrow AcAc, A \rightarrow ab\}$. This grammar needs to be restructured since the symbols Ac appear twice. SEQUITUR introduces another non-terminal to solve the problem. We get the rules $\{S \rightarrow BB, B \rightarrow Ac, A \rightarrow ab\}$. However, now the rule defining non-terminal A is used only once. So, this rule is eliminated to produce the final result. $\{S \rightarrow BB, B \rightarrow abc\}$. Note that the grammar accepts only the string $abcabc$.

Our compaction scheme To represent the address sequences accessed by branch and memory read/write bytecodes, we employ an improved version of the SEQUITUR algorithm. Note that SEQUITUR cannot efficiently represent contiguous repeated symbols, including both terminal and non-terminal symbols. For example, to represent the string $abababab$, SEQUITUR will produce the following rules: $\{S \rightarrow BB, B \rightarrow AA, A \rightarrow ab\}$. The string $(ab)^k$ will be represented by $O(lgk)$ rules.

To solve this problem, we have proposed Run-Length Encoding SEQUITUR (RLESe); RLESe constructs a context free grammar to represent a sequence *on the fly* (this contrasts with the work of [19] which modifies the SEQUITUR grammar post-mortem). The right hand side of each rule is a sequence of "**nodes**"; Each node is a pair $\langle sym : run \rangle$, representing run times contiguous occurrences of sym. For space efficiency, the run can be omitted when it equals 1. Thus, the string $(ab)^k$ is represented using only two rules in RLESe: $\{S \rightarrow A : k, A \rightarrow ab\}$. Note that patterns like $(ab)^k$ (where RLESe is significantly less space expensive than SEQUITUR) are not uncommon in program executions. Consider an `if-then-else` statement within a loop which alternates its outcome with every iteration of the loop. The target addresses of the conditional branch for the `if-then-else` statement will contain the pattern $(ab)^k$ where a and b are the beginning addresses of `then`, `else` parts of the statement.

The RLESe algorithm reads from the input sequence symbol by symbol. On reading a symbol sym, a node $\langle sym : 1 \rangle$ is created and appended to the start rule. When a run of a symbol has finished, the grammar rules are restructured by preserving the following properties. The first two properties are taken (and modified) from SEQUITUR. The third property is unique to RLESe, resulting from its maintenance of runs of terminals / non-terminals.

1. **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 symbols *e.g.* $\langle a : 2, b : 2 \rangle$ is similar to $\langle a : 3, b : 4 \rangle$, but $\langle a : 3, b : 2 \rangle$ is not similar to $\langle b : 3, a : 2 \rangle$.
2. **Rule utility property.** This rule states that every rule (except the start rule S) is referenced more than once. So when a rule is referenced by only one node and the run in that node equals 1, the reference will be replaced with the right side of this rule.
3. **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.

To maintain the digram uniqueness property in RLESe, we might need to split nodes during grammar construction. Consider a string represented by $S \rightarrow a : 8, b : 1, a : 6$. Now, if the next symbol[1] is b, S is restructured to $S \rightarrow a : 8, b : 1, a : 6, b : 1$. To ensure digram uniqueness we will now split the node $a : 8$ to $a : 6, a : 2$. This is to find and remove duplicate occurrences of similar digrams as:

$$S \rightarrow a : 2, A : 2 \qquad A \rightarrow a : 6, b : 1$$

In addition to the run-length encoding performed in RLESe, we also need to modify the terminal symbols fed into SEQUITUR or RLESe algorithm. In particular, we need to employ difference representations in memory reference sequences. For example, the trace sequence (0, 1, ... 9) in Table 1 cannot be compressed; this sequence represents the indices of the array elements written by the `iastore` bytecode in Figure 1. By converting it into its difference representation as (0, 1, 1, ..., 1), RLESe can compactly represent it with one rule $S \rightarrow 0, 1 : 9$. On the other hand, SEQUITUR requires three rules for representing this sequence. This inefficiency stems from its inability to capture runs of symbols, as shown in its representation of $(ab)^k$. RLESe compression is linear in space and time, assuming that we take constant time to find similar digrams (as in SEQUITUR [15]).

3. Efficiency of compaction

In this section, we quantitatively evaluate the time and space overheads of our compaction scheme, and compare it against existing compression techniques.

1 We assume here that the rules are restructured after we encounter b. Note that this is not the case if b is followed by other bs, *i.e.* we wait for the run of bs to end before restructuring.

Subject	Description	LOC	Input Size
Crypt	IDEA encryption and decryption	968	200,000 bytes
FFT	1-D fast Fourier transform	706	2^{15} complex numbers
HeapSort	Integer sorting	649	10000 integers
LUFact	LU factorisation	1076	200×200 matrix
Series	Fourier coefficient analysis	705	200 Fourier coefficients
SOR	Successive over-relaxation on a grid	559	100×100 grid

Table 3. Description of subject programs.

Program	Orig.	Trace Table	Trace Sequences	All	All/Orig. (%)	gzipped	gzipped / Orig. (%)
Crypt	69.48M	6.56K	3.99K	10.55K	0.02	3.99M	5.74
FFT	89.51M	3.26K	121.79K	125.04K	0.14	15.25M	17.04
HeapSort	23.29M	2.93K	2.44M	2.44M	10.48	2.03M	8.72
LUFact	139.30M	4.29K	190.02K	194.31K	0.14	15.48M	11.11
Series	38.22M	2.97K	582.43K	585.40K	1.53	10.29M	26.92
SOR	75.54M	2.74K	3.74K	6.48K	0.01	14.35M	19

Table 4. Compression efficiency of our bytecode traces. All sizes are in bytes.

3.1. Subject programs and trace collection

We have used six sequential Java programs to study the performance of our compact bytecode trace. These programs are taken from the Java Grande Forum benchmark suite [9]. Descriptions and sizes of these programs and the inputs to generate the traces are shown in Table 3. To collect execution traces, we have modified Kaffe virtual machine to monitor interpreting bytecodes. In our traces, we use object identifiers instead of addresses to represent objects. This is because the same address may be used by different objects during a program execution. Note that creation of structures such as multi-dimensional arrays, constant strings etc. may implicitly create objects. We trace and allocate identifiers to these objects as well.

In practice, programmers often use methods in libraries provided by other vendors. The behaviors of these methods are typically not interesting to programmers. Hence it is not necessary to trace all the details during execution. We can use method specifications to describe which variables and bytecodes in library methods are needed for slicing, and instrument them. For example, assume that the program has invoked *obj.chatAt(index)*, a method of class *java.lang.String* to retrieve the character at the specified index of a string. The specification tells that the return variable is data dependent on the $index$th character of string obj, so we only trace the identity of object obj and the value of $index$.

Post-mortem analysis may require forward or backward traversal of execution traces. In the first situation, the conditional branch bytecodes at the end of a basic block with multiple outgoing control flow edges need to be monitored. Indeed our compact traces as described in the last section are suitable for forward traversal. For efficient backwards traversal, we need to monitor/trace bytecodes at the beginning of basic blocks with multiple incoming edges. Since our slicing algorithm performs backwards trace traversal, we have implemented this representation of traces.

3.2. Efficiency of compaction

Table 4 shows the compression efficiency of our compact trace representation. The column *Orig.* represents the cost of storing uncompressed execution traces. Next two columns *Trace Table* and *Trace Sequences* show the space overhead to maintain the trace tables and to store the compressed operand sequences of bytecodes. The column *All* represents the overall space costs of our compact bytecode traces. The first % column indicates *All* as a percentage of *Orig.*. For all our subject programs, this percentage is less than 11% (*i.e.* approximately 10 times compression). Indeed for most of the programs we get two orders of magnitude (*i.e* 100 times) compression.

We also compare the space efficiency of our compact bytecode trace representation with a general purpose data compression scheme such as the *gzip* utility [26]. This is shown in the column *gzipped* of Table 4. The second % column of Table 4 indicates *gzipped* as a percentage of *Orig.*, showing the compression ratio achieved by *gzip*. For most programs, the space efficiency achieved by gzip is more than 11%. Our trace representation is significantly more efficient than gzip. The only exception is the *HeapSort* program which employs random data access patterns (and hence is not suitable for SEQUITUR based compression algorithms).

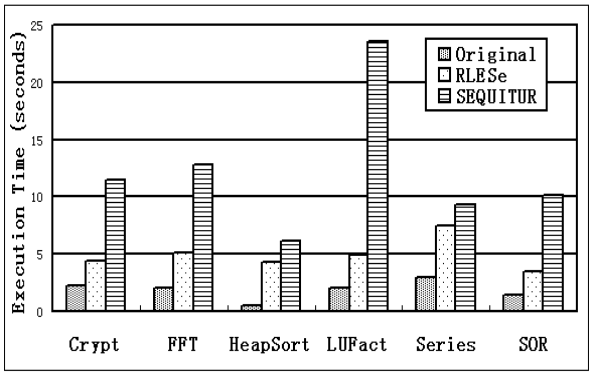

Figure 2. Time overhead of RLESe and SEQUITUR. The time unit is *second*.

Figure 2 presents the running time of selected programs without instrumentation, tracing with RLESe and tracing with SEQUITUR. The experiments were performed on a Pentium 4 1.6 GHz machine with 512MB memory. It shows that collecting bytecode traces using RLESe is time-efficient for most programs except *HeapSort*. Random memory access patterns in this program make some long operand sequences contain too many digrams. Thus it becomes costly to look for similar digrams during checking the digram uniqueness property of RLESe.

For each program studied, we have only presented performance results with one input. Traces of these programs have many contiguous repetitions which can be exploited by RLESe. Most of our programs are such that an increase in input size leads to linear increase in uncompressed trace size and hence in the time overhead for trace collection/compression. However, with the increase in input and trace size, our compression ratio does not decrease; in fact it remains roughly the same or improves slightly.

Comparing RLESe with SEQUITUR We now compare the time and space efficiency of RLESe against SEQUITUR. Both algorithms were performed on operand sequences of bytecodes. Table 5 compares the space costs of both algorithms by presenting their compression ratio (in percentage). Figure 2 compares their time overheads. We exploited the same hash function to search for similar diagrams in the implementation of both algorithms. Clearly RLESe outperforms SEQUITUR in both space and time on studied programs. Since RLESe employs run-length encoding of terminal and non-terminal symbols over and above the SEQUITUR algorithm, the final compressed sequence cannot be less space efficient than the output of the SEQUITUR algorithm. The time overheads of both algorithms are mainly caused by restructuring grammar rules. RLESe restructures

Program	RLESe %	SEQUITUR %
Crypt	0.02	0.05
FFT	0.14	0.34
HeapSort	10.48	10.52
LUFact	0.14	0.4
Series	1.53	2.56
SOR	0.01	0.02

Table 5. Comparing compression ratio of RLESe and SEQUITUR.

the rules after a run of the same symbol has finished, whereas SEQUITUR does this on reading every symbol. In other words, RLESe restructures the grammar rules less frequently than SEQUITUR. RLESe also produces less symbols, so that similar digrams can be found more efficiently. The studied program traces have many contiguous repetitions of patterns. This matches the mechanism of RLESe well. When there are only few contiguous repeated symbols, performance of RLESe will be similar with that of SEQUITUR, and the intermediate representation used by RLESe may consume more memory, because each node used by RLESe has to use one more field to represent the run length.

4. Dynamic slicing on compact traces

In this section, we describe how our compact bytecode traces can be used for dynamic slicing. This involves (a) backwards traversal of the program's bytecode stream and the compressed trace simultaneously, and (b) computation of control/data dependences.

4.1. Slicing algorithm

Dynamic slicing is performed w.r.t. a slicing criterion (I, α, V), where I is an input, α represents some bytecodes the programmer is interested in, and V is a set of variables referenced at these bytecodes. Often the user-defined criterion is of the form (I, l, V) where l is a control location; in this case α represents the bytecodes corresponding to the statements immediately preceding l. Dynamic slice contains all bytecodes which have affected the values of variables in V referenced at last occurrences of α during execution.

During dynamic slicing, we maintain δ, a list of variables whose values need to be explained, γ, a set of bytecodes to check control dependences, and φ, the dynamic slice. Initially we set δ, γ and φ to $\emptyset$. We now traverse the compressed program trace and program's bytecode stream backwards starting from the last bytecode occurrence recorded in the trace. At every bytecode occurrence β encountered during this backwards traversal, let b_β be the bytecode executed at β. We perform the following steps.

- If β is the last occurrence of b_β in trace and b_β belongs to α (the slicing criterion), then insert the variables used at β into δ, and the bytecode b_β into γ and φ.
- Check whether any bytecode currently included in γ is control dependent on b_β using the static program dependence graph. If so, b_β is included in the slice φ; bytecodes which are control dependent on b_β are removed from γ and b_β is inserted into γ; variables used at β are inserted into δ.
- Check whether the variable defined at β appears in the set δ. If so, b_β is also included in γ and φ; the set δ is updated by (a) removing the variable defined at β, and (b) inserting the variables used at β.

Thus, our algorithm essentially proceeds by traversal and is similar in flavor to the precise dynamic slicing algorithms of [23]. The crucial operation in constructing slices is to detect the control/data dependences between bytecodes. The data dependence analysis is complicated by Java's stack based architecture as explained in the following.

Data Dependences Java exploits stacks to exchange data between program variables, and perform arithmetic calculations during execution. This leads to additional complications for data dependence analysis, because dynamic data dependence exists implicitly or explicitly, depending on whether or not the stack is involved. In particular, data dependence exists (a) explicitly when the same variable is defined and used by bytecodes or (b) implicitly when the stack is involved. Taking bytecodes of the method *main()* in Figure 1 as an example, bytecode 29 is explicitly data dependent on bytecode 26, since 29 uses the variable (which is the local variable i in this case) which is defined by 26. Bytecode 32 is implicitly data dependent on bytecode 29, since 29 pushes the value of a variable which is later used by 32 into the stack. In other words, implicit data dependence exists when a bytecode uses values which were pushed into the stack by its predecessor bytecode(s). Explicit data dependence can be detected by comparing the addresses (or identities) of accessed variables. However, detection of implicit data dependence needs to simulate the stack. Simulating the stack involves two issues:

- The stack maintained during slicing will not contain the actual values. We maintain only the exact number of push and pop operations performed by each bytecode, (*i.e.* the content of each stack element is ignored).
- Note that our slicing algorithm is based on backwards traversal of the program trace, whereas the actual stack is constructed via forward program execution. Thus, during slicing we reverse the roles of pushing and popping the stack. In other words, a push (pop) operation by a bytecode b during program execution amounts to a pop (push) operation when b is encountered during slicing.

Traversal without Decompression The bytecode traces collected during execution do not need to be decompressed during slicing. The maximum compaction in our scheme comes from the representation of bytecode operand sequences via RLESe. To be able to traverse our program traces without decompression, we need to traverse the RLESe representation. For example, consider a sequence of target addresses for a branch bytecode $\langle 9, 7, 8, 7, 9, 7\rangle$. During backwards traversal of the program trace, we need to mark the portion of the sequence which has been seen so far (say $\langle 9, 7, 8, \mid 7, 9, 7\rangle$). We use this marking to find the previous unvisited occurrence of the branch bytecode (here the corresponding target address is 8). Such markings can be efficiently maintained and updated on the RLESe representation of the sequences as follows.

Note that the RLESe grammar of a string σ can simply be stored as a directed acyclic graph (DAG). This DAG is constructed from the rules of the grammar starting from the start symbol. Each node of this graph is of the form $\langle sym : run\rangle$ where sym is a terminal or non-terminal symbol and *run* denotes a run-length. Thus, the symbol in σ that was last visited in the backwards traversal can be marked by a root-to-leaf path π in the DAG representation. Furthermore each node $X = \langle sym : run\rangle$ in path π is annotated with an integer $n \leq run$ denoting the number of occurrences of sym which has been visited so far. During backwards traversal of the program trace, we can use the annotations of π to find the previous unvisited terminal symbol of the RLESe grammar as shown in the algorithm *GetPrevious* (refer Figure 3). In Figure 3, G is the DAG representation of the RLESe grammar and the integer annotation of any node X in path π is denoted as $annot_X$.

Implementation We employed our backwards traversal algorithm on the compact bytecode traces of the six Java Grande benchmarks of Table 3. We also implemented in Java some example programs given in Agrawal's thesis [1], including euclid.c[2], inter-proc.c[3]. We then used our slicing algorithm to debug them. In all the cases, the time overhead for a single slicing request was minimal, typically comparable to the time overhead for trace collection.

Improvement Our dynamic slicing algorithm is performed directly on the execution trace. This is suitable for single slicing request, while not perfect for multiple slicing requests. The latter requires traversing the entire trace multi-

2 It implements the Extended Euclidean algorithm to compute the greatest common divisor (GCD) of two input integers a and b, as well as integers x and y such that ax + by = gcd(a,b).

3 It finds the sum of areas of given triangles.

```
GetPrevious(G: Grammar, π: path in G)
begin  X := leaf node of π;
       while root of π not reached do
           Let X = ⟨_sym : run⟩;
           if annot_X < run then
               break;
           else search for immediate left sibling of X in G;
               if such a sibling Sib_X exists then
                   X := Sib_X; break
               else X := parent of X in path π;
               endif;
           endif;
       endwhile;
       G1 := DAG rooted at node X within G;
       return symbol in rightmost leaf node of G1;
end
```

Figure 3. One step in the backwards traversal of a RLESe sequence (represented as DAG) without decompressing the sequence.

ple times, and the time overheads can grow. Building the dynamic dependence graph can reduce the traversal cost. Unfortunately, Zhang et al. have reported that full dynamic dependence graphs of real applications tend to be too huge to fit in the main memory [23]. They suggested some preprocessing of dependences before slicing.

Our slicing algorithm can be improved as follows: during the first backward traversal of the trace, we store a summary of addresses (or identities) of all defined variables of each *block instance*, where a *block instance* can be either a method invocation or an execution instance of some contiguous basic blocks. During later traversals, before entering the trace for a block instance, we check whether any variables appearing in δ (the list of variables whose values need to be explained) are present in the summary. If no such variable is found, checking data dependences can be avoided. Using the summary information, we can reduce the number of address comparisons performed to detect data dependences during slicing [23]. Thus the performance of our dynamic slicing algorithm for multiple slicing requests can be improved.

4.2. Detecting omission errors

Conventional dynamic slices only explain why and how a statement is executed; they do not consider the fact that execution of a particular statement may be wrongly omitted. Thus, the statement which is responsible for the program failure may be excluded from the slice. Consider the "buggy" program fragment presented in Figure 4, where the statement `b=1` at line 1 should be `b=2` in a correct program.

With input `a=2`, x at line 6 is 1 unexpectedly. The execution trace is ⟨1,2,3,4,6⟩. If we want to explain the value of x after line 6, the dynamic slice (computed by conventional

```
1    b = 1;
2    x = 1;
3    if (a > 1) {
4        if (b > 1){
5            x = 2
         }
     }
6    ... = x
```

Figure 4. A "buggy" program fragment

slicing algorithms like [23]) contains only lines 2 and 6. Unfortunately, line 1, the source of the bug, is excluded. In the above example, the execution of the statement `x=2` at line 5 is wrongly omitted due to the wrong value being set to b in line 1. Dynamic slicing concludes that lines $1, 4$ have no actual effect on x at line 6 for this particular execution, and excludes these lines from the slice.

We propose an *extended dynamic slicing* algorithm to capture omissions of bytecodes' execution. The *extended dynamic slice* is defined w.r.t. an *Extended Dynamic Dependence Graph* (EDDG); the slice contains those EDDG nodes from which node(s) for the slicing criterion can be reached. The EDDG is an extension of the Dynamic Dependence Graph (DDG) of [2]. Recall that each node of the DDG can represent one particular occurrence of a statement (or bytecode in our case) in the program execution; edges, of course, represent dynamic data and control dependences. We construct the EDDG from DDG by adding (a) a dummy node for each occurrence of a branch bytecode, and (b) edges for dynamic data and *potential dependences* w.r.t. these dummy nodes. The *potential dependence* captures possible execution omissions; a bytecode occurrence β' has *potential dependence* on an earlier branch bytecode occurrence β if and only if a variable v used at β' is defined before β, and v may be re-defined before reaching β' if β is evaluated differently. Figure 5 shows the EDDG for the above example, and the resulting extended dynamic slice w.r.t. x at line 6 consists of lines ⟨1,2,4,6⟩. Note that potential dependences have been studied in the context of *relevant slices* [3, 6] which also capture omission errors. We compare our approach to these works in the next section.

We can compute the extended dynamic slice by traversing the trace exactly once, without constructing the EDDG. Before traversing, we will construct the Extended Static Program Dependence Graph (ESPDG), capturing control, data and conditional dependences [6]. A bytecode $s1$ has *conditional dependence* on a branch bytecode $s2$ if a reaching definition of a variable v used at $s1$ is (transitively) control dependent on $s2$. Such a conditional dependence is denoted by $s1 \rightarrow^{v}_{cond} s2$, that is, variable v is kept track of.

On each occurrence β of a bytecode b_β encountered during backwards traversal of the trace, we first check it as per

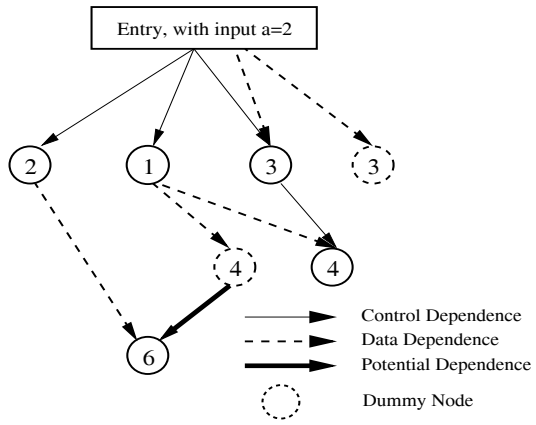

Figure 5. The EDDG of a simple program

the slicing algorithm in Section 4.1. If any of the conditions of this algorithm is satisfied (*i.e.* β has effect on the slicing criterion via control/data dependences), we insert b_β into the slice φ, and maintain the sets δ and γ as in Section 4.1. Otherwise, we check for *potential dependence*, that is, check whether b_β is a branch bytecode and any variable v in δ satisfies: (1) there exists a bytecode b such that $b \rightarrow^{v}_{cond} b_\beta$, and (2) v may be re-defined if β is evaluated differently. If so, we insert b_β into the slice, and update δ with variables used at β. Details are omitted due to space consideration. We are currently implementing this approach to examine its impact on slice sizes and time overheads.

5. Related work

Weiser originally introduced the concept of program slicing [21]. In 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 [3, 11, 14, 20]. The first dynamic slicing algorithm was introduced by Korel and Laski [10]. In particular, they exploited *dynamic flow concepts* to capture the dependences 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 [1, 2]. Static and dynamic slicing of object-oriented programs based on dependence graphs have been studied in [12] and [22] respectively. Computation of control and data dependences between Java bytecodes has been discussed in [25]. Ohata et al. [16] 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. [23] studied the performance issues in computing the control/data dependences from the execution traces for slicing purposes. To the best of our knowledge, none of the existing slicing algorithms operates on *compressed* representation of the program's run-time information as in our case. Thus, our slicing algorithms achieves immense space savings at the cost of tolerable additional time overheads.

Literature [3, 6] have studied *relevant slices* to detect omission errors. However, the approach in [3] relies on the huge dynamic dependence graph. Furthermore, if b is a branch statement on which statements in the slice are potential dependent, [3] only computes the closure of data and potential dependences of b. In other words, control dependences are ignored w.r.t. statements on which b is data dependent. The forward relevant slicing algorithm in [6] avoids using the huge dynamic dependence graph. However, such an algorithm will compute many redundant dependences since it is not goal directed. Some additional unrelated statements will also be included in the slice. This is because while computing the dependences of a later occurrence of certain branch statements (those which appear in the slice due to potential dependences), the algorithm also includes statements which affect an early occurrence of the same branch statement.

Various compact trace representation schemes have been developed in [5, 13, 17, 24] to reduce the high space overhead. Pleszkun presented a two-pass trace scheme, which recorded basic block's successors and data reference patterns [17]. The organization of his trace is similar to that of ours. However, Pleszkun's technique does not allow traces to be collected on the fly, and the trace is still large because no data compression technique is exploited. Recently, Larus proposed a compact and analyzable representation of a program's dynamic control flow via the on-line compression algorithm SEQUITUR [13]. The entire flow 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 to break the traces into per-method traces [24]. However, it is not clear how to efficiently represent data flow in their traces. The idea of separating out the data accesses of load/store instructions into a separate sequence (which is then compressed) was explored in [5] in the context of parallel program executions. However, this work uses the SEQUITUR algorithm (which is not suitable for representing runs of 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 [19], since it is an on-line compression algorithm, whereas Reiss and Renieris suggested modifying SEQUITUR grammar rules in a post processing step. On-the-fly generation of the grammar, in fact, leads

to significant space savings (as can be seen from the relative sizes of RLESe and SEQUITUR presented in Table 5).

6. 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 performed post-mortem. We also extend our dynamic slicing algorithm to explain ceratin classes of omission errors (errors arising due to omission of a statement's execution).

Besides dynamic slicing, our compact bytecode traces are also useful for many other applications in code optimization and program visualization. First, the trace contains the sequence of target addresses for each conditional branch bytecode. We can obtain the most likely taken target addresses from these sequences to merge basic blocks into a superblock [8]. Secondly, the operand sequences of bytecodes to invoke virtual/interface methods describe which methods are most likely to be invoked; this information is helpful in inlining methods for optimization [4]. Finally, note that by recording addresses of objects that each bytecode creates, our trace provides information about memory allocations. This can be used to understand the memory behavior via visualization, as discussed in [18].

References

[1] H. Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1991.

[2] H. Agrawal and J. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[3] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *International Conference on Software Maintenance (ICSM)*, pages 348–357, 1993.

[4] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–26, 2000.

[5] A. Goel, A. Roychoudhury, and T. Mitra. Compactly representing parallel program executions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 191–202, 2003.

[6] T. Gyimóthy, A. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 303–321, 1999.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[8] W. W. Hwu et al. The superblock: An effective structure for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1), 1993.

[9] JGF. The Java Grande Forum Benchmark Suite. web cite: `http://www.epcc.ed.ac.uk/javagrande/seq/contents.html`.

[10] B. Korel and J. W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[11] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. In *International Workshop on Automatic Debugging*, 1997.

[12] L. Larsen and M. Harrold. Slicing object-oriented software. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 495–505, 2001.

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

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

[15] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference (DCC)*, pages 3–11, 1997.

[16] F. Ohata, K. Hirose, M. Fujii, and K. Inoue. A slicing method for object-oriented programs using lightweight dynamic information. In *Asia-Pacific Software Engineering Conference*, 2001.

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

[18] S. P. Reiss and M. Renieris. Generating Java trace data. In *ACM Java Grande Conference*, 2000.

[19] S. P. Reiss and M. Renieris. Encoding program executions. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 221–230, 2001.

[20] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[22] B. Xu, Z. Chen, and H. Yang. Dynamic slicing object-oriented programs for debugging. In *IEEE International Workshop on Source Code Analysis and Manipulation*, 2002.

[23] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 319–329, 2003.

[24] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 180–190, 2001.

[25] J. Zhao. Dependence analysis of Java bytecode. In *IEEE Annual International Computer Software and Applications Conference*, pages 486–491, 2000.

[26] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–349, 1977.