Efficient Program Execution Indexing

Bin Xin William N. Sumner Xiangyu Zhang

Department of Computer Science, Purdue University, West Lafayette, Indiana 47907
{xinb,wsumner,xyzhang}@cs.purdue.edu

Abstract

Execution indexing uniquely identifies a point in an execution. Desirable execution indices reveal correlations between points in an execution and establish correspondence between points across multiple executions. Therefore, execution indexing is essential for a wide variety of dynamic program analyses, for example, it can be used to organize program profiles; it can precisely identify the point in a re-execution that corresponds to a given point in an original execution and thus facilitate debugging or dynamic instrumentation. In this paper, we formally define the concept of execution index and propose an indexing scheme based on execution structure and program state. We present a highly optimized online implementation of the technique. We also perform a client study, which targets producing a failure inducing schedule for a data race by verifying the two alternative happens-before orderings of a racing pair. Indexing is used to precisely locate corresponding points across multiple executions in the presence of non-determinism so that no heavy-weight tracing/replay system is needed.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Diagnostics, Monitors; D.3.4 [*Programming Languages*]: Processors—Debuggers

General Terms Algorithms, Measurement, Reliability

Keywords Execution indexing, Execution alignment, Control dependence, Structural indexing, Semantic Augmentation, Data race

1. Introduction

During program execution, a static program statement could be executed multiple times, resulting in different execution points. A fundamental challenge in dynamic program analysis is to uniquely identify individual execution points so that the *correlation* between points in one execution can be inferred and the *correspondence* between execution points across multiple executions can be established. Solving this problem is significant for a wide range of applications.

Profiling. Program profiling collects information about program executions such as frequently executed paths, referenced addresses, produced values, and exercised dependences. Such information can be used in program optimizations, debugging, testing, parallelization, and so on. Currently, most program profiling techniques index profiles through static program points [7] and can effectively answer queries such as *finding the set of addresses referenced at program point* x. Such an indexing scheme merges the information of individual execution instances of x and thus is insufficient for some applications. For example, in order to study the available concurrency in program execution, it is essential to distinguish computation performed in different iterations of a loop. Moreover, we should only compare the iterations that have similar dynamic contexts because two iterations of a loop, although processing disjoint datasets, may be nested in completely different calling contexts so that parallelizing them requires significant code restructuring. In other words, applications like parallelization require more expressive indexing techniques to organize execution profiles so that the correlation between points can be unveiled.

Debugging. A debugging practice often entails setting breakpoints, re-running the program, and inspecting program state when the execution is trapped. In many situations, the need of setting breakpoints at a particular execution *instance* of a static program point arises. Although many debuggers support skipping a certain number of instances of a breakpoint, it is known to be insufficient as the ith instance of a statement s in the re-execution might not be the same ith instance in the original execution, due to nondeterminism or program state perturbations performed at earlier breakpoints. An indexing scheme that tolerates nondeterminism and execution perturbations is highly desirable.

Dynamic Instrumentation. The recent advances of program instrumentation techniques allow instrumentation to be arbitrarily turned on and off at runtime, which provides flexibility for many dynamic program analysis. For example, execution omission errors (EOE) result in failures through not executing certain statements. While any statements that are not executed cannot be traced, EOEs are difficult for most trace-based techniques. In [23], EOEs are tackled by identifying implicit dependence between a predicate execution instance and a memory reference point. This is achieved by forcing the predicate instance to take its opposite branch and then observing the value change at the memory reference point. Switching the predicate instance requires the instrumentation to precisely locate the instance. Furthermore, switching the predicate instance perturbs the execution and thus the instrumentation needs to find the corresponding memory reference point in the perturbed execution. Similarly, generating failure inducing scheduling for a datarace is often achieved by forcing the two alternative happens-before relations between the two racing execution points. Identifying the same two points in the two executions in the presence of nondeterminism demands accurate indexing techniques.

Execution Comparison. Execution comparison focuses on the similarity between executions. It has been used in debugging and testing. In [22], the minimal differences between the program states of two executions, one is passing and the other is failing, are composed as the failure inducing chain from the root cause to the failure symptom. The comparison entails a highly complex procedure of establishing matches between memory states in the two executions. An indexing that can establish correspondence between two differ-

PLDI'08 June 7–13, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-59593-860-2/08/06...$5.00

ent executions can significantly reduce the complexity. In testing, the similarity between executions can be used as a criterion to reduce redundancy and prioritize test cases.

Despite the great need for execution indexing, to the best of our knowledge, it has not been studied as a stand-alone problem. Some research has been carried out in related fields. Many existing program trace related techniques [24] use s_i, which denotes the ith instance of statement s or the instance s at time stamp i, to identify an execution point. While it uniquely identifies an execution point, it does not contain any information to represent the relations between points and lacks the power of establishing correspondence between executions. In [23, 9], offline algorithms are proposed to align multiple executions or regions in multiple executions for their own purposes. An ideal execution indexing technique should be online with low overhead such that indices can be queried any time during execution like querying calling contexts. With indices, execution alignment becomes trivial as two points in two different executions must align if they have the same index. In the context of aspect-oriented programming [18, 3] and security [13], event patterns are described by regular languages or automata to locate execution points where certain interesting states have been reached. These techniques are not general in the sense that they are only interested in some execution points and the user needs to define the set of events and their patterns.

In this paper, we propose a general execution indexing scheme that is based on execution structure and program state. The basic idea is to parse program executions according to their nesting structure, which is expressed as a set of grammar rules. The set of rules that have been used to parse a given execution point reveals its structure and thus constitutes its index. To improve the flexibility and expressiveness of our indexing scheme, the grammar rules can be augmented with semantic information such as values at particular execution points. Our technique features a cost-effective online implementation with a set of optimizations. Our contributions are highlighted as follows:

- We formally define the problem of execution indexing.
- We propose to describe program execution by a context free language. The set of rules are constructed based on the nesting structure of the program, which can be inferred from program control dependence. An algorithm is given to explicitly derive the set of rules.
- We present an efficient online algorithm to compute indices without requiring explicit construction of the grammar rules. A few optimizations further reduce the overhead to 42% on average.
- We propose semantic augmentation to the structure based execution indexing. With the augmentation, program state can be incorporated into the grammar rules and thus become part of an index. As a result, users' insights of program behavior can be leveraged in index construction.
- We perform a client study of the indexing technique by applying it to lightweight generation of a failure inducing scheduling for data races. Data races are benign if they can not lead to any failure. For a pair of racing program points, two executions are emitted to verify the two alternative orderings. The indices are used to precisely locate the corresponding points in these executions. Our experimentation shows that with indexing, failure inducing schedules can be easily generated without relying on an expensive tracing and replay system.

2. Execution Indexing

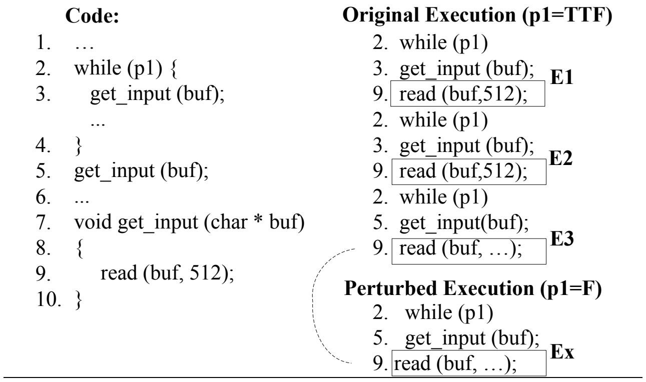
In this section, we first formally define the concept of *execution index*. We then describe the structural indexing scheme.

DEFINITION 1. **(Execution Index)** *Given a program $\mathcal{P}$, the index of an execution $\mathcal{P}(\vec{i})$, denoted as $EI^{\mathcal{P}(\vec{i})}$ with $\vec{i}$ being the input vector, is a function of execution points in $\mathcal{P}(\vec{i})$ that satisfies the following property:*
$\forall$ two execution points $x \neq y$, $EI^{\mathcal{P}(\vec{i})}(x) \neq EI^{\mathcal{P}(\vec{i})}(y)$.

From the definition, any function that uniquely identifies an execution point can serve as an index. A very important property of an index function is that it establishes a correspondence relation between points in multiple executions.

DEFINITION 2. **(Execution Correspondence)** *Two execution points, x in execution $\mathcal{E}$ and y in $\mathcal{E}'$, correspond to each other iff $EI^{\mathcal{E}}(x) \equiv EI^{\mathcal{E}'}(y)$.*

$\mathcal{E}$ and $\mathcal{E}'$ may correspond to different inputs, or even though they have the same input, $\mathcal{E}'$ is a perturbation of $\mathcal{E}$, caused by nondeterministic scheduling and so on.



```
Code:
1.   …
2.   while (p1) {
3.      get_input (buf);
        ...
4.   }
5.   get_input (buf);
6.   ...
7.   void get_input (char * buf)
8.   {
9.        read (buf, 512);
10.  }
```

```
Original Execution (p1=TTF)
2.  while (p1)
3.  get_input (buf);
9.  read (buf,512);       E1
2.  while (p1)
3.  get_input (buf);
9.  read (buf,512);       E2
2.  while (p1)
5.  get_input(buf);
9.  read (buf, …);        E3

Perturbed Execution (p1=F)
2.  while (p1)
5.  get_input (buf);
9.  read (buf, …);        Ex
```

Figure 1. Log Based Replay with Perturbation

2.1 Motivating Example

As described in the introduction, constructing execution correspondence is essential for a wide range of applications. Simple indexing schemes are not sufficient in providing meaningful correspondence. Consider the example in Figure 1. The program reads input from a file within a loop and then reads another piece of input outside the loop. Assume in the original execution, $p1$ takes the value sequence of *true*, *true*, and *false*. As a result, function `get_input()` is called three times, twice from inside the loop. The logged events are highlighted by boxes and labeled with **E1**, **E2**, and **E3**. Assume a failure happens and the programmer tries to identify the correlation between $p1$ and the failure by changing the branch outcome of the first instance of $p1$, i.e., from *true* to *false*, and observes if the failure disappears. Such a switching process is also used in handling execution omission errors [23] to automatically unveil implicit dependence. The programmer replays the execution using the event log and perturbs the replayed execution by switching $p1$ at its first evaluation. As the perturbed replay relies on the event log collected in the original run, the challenge lies in correctly supplying events during replay. In this case, it is to associate event **E3** with statement 9 in the perturbed run. Note that we assume **E3** is independent of **E1** and **E2**. This often occurs when the program parses **E1** and **E2** for one structure and then parses **E3** for another structure. The simplest indexing scheme that uses the time order fails because the first event in the original run is **E1**, whereas the first event expected in the perturbed execution is **E3**. A smarter indexing that represents an execution point as s_i, meaning the ith instance of statement s, although has been widely used in existing dynamic analyses [19, 24, 14], is not sufficient either. **E1** has the index of 9_1, which is the same as **Ex**'s. In other words, **E1** is the correspondence of **Ex** and thus **E1** is supplied to statement 9 during replay, which is wrong.

In this paper, we propose an indexing technique based on execution structure. In Figure 1, event **E1** is processed at statement 9, which is nested in the method call made at 3, which in turn is nested in the true branch of predicate $p1$ at 2. Other executed statements such as the second and third calls at 3 are not related to the nesting structure of **E1** and should not be part of the index. Therefore, **E1** has the index of $[2, 3, 9]$. **E2** has the index of $[2, 2, 3, 9]$. The two consecutive 2s in the index indicate that the event is nested in the second iteration of the loop. Both **E3** and **Ex** have the same index $[5, 9]$, meaning the structures of these two events occur within the calls made at 5 and are not related to other statements such as 2 or 3. Based on the structural indices, **E3** is provided as the expected event. Note that using call stacks does not work because **E1** and **E2** have the same call stack. In other words, call stacks are not a valid execution index function. This is due to call stacks only recording a partial image of the nesting structure. The proposed indexing scheme not only succeeds in establishing desired correspondence for points across different executions in many cases, but also facilitates highly efficient online computation.

2.2 Structural Indexing

In this subsection, we present our design in detail. Implementation will be discussed in the next section. The key observation of our technique is that *all possible executions of a program can be described by a language called execution description language (EDL) based on structure*. An execution is a string of the language.

Code	1 s_1;
	2 s_2;
	3 s_3;
	4 s_4;
		1 if (...)
		2 s_1;
		3 else
		4 s_2;
			1 while (...) {
			2 s_1;
			3 }
			4 s_2;
				1 void A() {
				2 B();
				3 }
				4 void B() {
				5 s_1;
				6 }

EDL	$S \longrightarrow \widetilde{1}\,\widetilde{2}\,\widetilde{3}\,\widetilde{4}$	$S \longrightarrow \widetilde{1}\ \mathit{R1}$ $\mathit{R1} \longrightarrow \widetilde{2} \mid \widetilde{4}$	$S \longrightarrow \widetilde{1}\ \mathit{R1}\ \widetilde{4}$ $\mathit{R1} \longrightarrow \widetilde{2}\,\widetilde{1}\ \mathit{R1} \mid \epsilon$	$S \longrightarrow \widetilde{2}\ \mathit{RB}$ $\mathit{RB} \longrightarrow \widetilde{5}$
Str.	1 2 3 4	1 2 1 4	1 2 1 4 1 2 1 2 1 4	2 5

Table 1. EDLs for Simple Constructs.

Table 1 presents the EDLs for a list of basic programming language constructs. The first column shows sequential code without nesting, whose execution is described by a grammar rule that lists all the statements. Note that a terminal symbol s is denoted as $\widetilde{s}$ in this paper. In the second column, the `if-else` construct introduces a level of nesting and thus the EDL has two rules, one expressing the top level structure that contains statement 1 and the intermediate symbol $R1$ representing the substructure led by 1. The two alternative rules of $R1$ denote the substructure of the construct. The self recursion in the grammar rule for the `while` loop in the third column expresses the indefinite iterations of the the loop. From these examples, we can see that *EDLs are different from programming languages*. The strings of EDLs are executions whereas the strings of programming languages are programs. The alphabet of an EDL contains all the statement ids in the program, whereas that of a programming language contains program constructs, variable identifiers, and so on. The second observation is that *program control dependence perfectly reflects execution structure*. A statement x control depends on another statement y, usually a predicate or a method call statement, if y *directly decides* the execution of x. The formal definition can be found in the seminal paper [8]. For example in the second column of Table 1, statement 2 is control dependent on statement 1. The statements that share the same control dependence are present on the right hand side of the same rule, representing the same level of nesting. Consider the rules for the `while` construct. Statements 1 and 4 have the same dependence and they are listed on the right hand side of the first rule; the body of rule $R1$ lists the statements that are dependent on statement 1. Note that statement 1 control depends on itself as the execution of a loop iteration is decided by its previous iteration.

We define the EDL of a program as follows.

DEFINITION 3. **(Execution Description Language)** *Given a program $\mathcal{P}$, its execution description language, denoted as $EDL(\mathcal{P})$, is the language described by the grammar rules generated by Algorithm 1.*

Algorithm 1 *Grammar Construction*

Input: a program $\mathcal{P}$.
Output: a set of grammar rules that describe the executions of $\mathcal{P}$.

```
1:  ConstructGrammar ($\mathcal{P}$)
2:  {
        $rules = \emptyset$;
3:      for each method $\mathcal{M}$ {
4:          /* CD denotes control dependence*/
5:          $T$ = statements in $\mathcal{M}$ in flow order
                    satisfying $CD = START_{\mathcal{M}}$;
6:          $rules \cup= R\mathcal{M} \rightarrow T$;
7:          for each statement $s$ in $\mathcal{M}$ {
8:              if ($s$ is a predicate) {
9:                  $T$ = statements in flow order s.t. $CD = s^{true}$;
10:                 $F$ = statements in flow order s.t. $CD = s^{false}$;
11:                 $rules \cup= Rs \rightarrow T \mid F$;
12:             }
13:         }
14:     }
15:     /*post processing to complete the rules */
16:     for each rule $r \rightarrow X$
17:         for each symbol $s \in X$ {
18:             if ($s$ is a predicate)
19:                 replace $s$ with "$s\ Rs$" in $X$;
20:             if ($s$ is a call to $\mathcal{M}$)
21:                 replace $s$ with "$s\ R\mathcal{M}$" in $X$;
22:         }
23: }
```

While there exist different grammar rules that describe the same language, we rely on the rules generated by Algorithm 1 as they lead to a clear and concise definition of execution index, which will be discussed later in this subsection. Algorithm 1 is based on program control dependence: a grammar rule is created for statements that share the same control dependence. It consists of two major steps. Lines 3-14 describe the first step, in which statements in individual methods are clustered based on their control dependences. Here we consider all statements in a method that have empty control dependence to be control dependent on the method. In the second step (lines 16-22), the rules generated in the first step are scanned and symbols that have control dependents are appended with the grammar rules that describe the substructures of their control dependents.

To demonstrate the algorithm, we use a more complete example in Figure 2. The code contains a recursive call (line 9) and non-structural control flow (line 5). The first rule in Figure 3 represents the top level structure of method `A`. Due to the `return` at line 5, as shown by the CFG, only statements 2 and 3 are control dependent on the start node of `A`, $START_{\mathcal{A}}$. The second step of

```
 1: A () {
 2:      s1;
 3:      if (C1) {
 4:            B();
 5:            return;
 6:      }
 7:      while (C2) {
 8:            if (C3)
 9:                  A();
10:            s2;
11:      }
12:      B();
13: }
14:
15: B () {
16:      s3
17: }
```

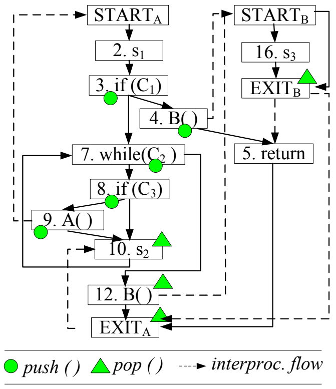


Figure 2. A Running Example

$$
\begin{array}{lcl}
RA & \longrightarrow & \widetilde{2}\ \widetilde{3}\ R3 \\
R3 & \longrightarrow & \widetilde{4}\ RB\ \widetilde{5} \mid \widetilde{7}\ R7\ \widetilde{12}\ RB \\
R7 & \longrightarrow & \widetilde{8}\ R8\ \widetilde{10}\ \widetilde{7}\ R7 \mid \epsilon \\
RB & \longrightarrow & \widetilde{16} \\
R8 & \longrightarrow & \widetilde{9}\ RA \mid \epsilon
\end{array}
$$

Figure 3. The Grammar Generated by Algorithm 1 for Figure 2.

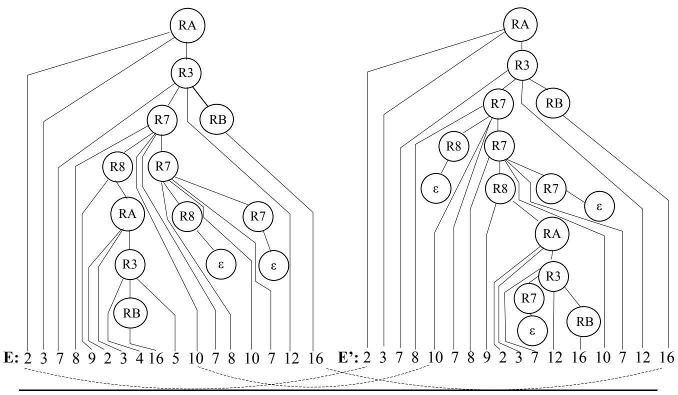

RA
R3
R7
RB
R8
ε
E: 2 3 7 8 9 2 3 4 16 5 10 7 8 10 7 12 16
E': 2 3 7 8 10 7 8 9 2 3 7 12 16 10 7 12 16

Figure 4. Indexing Two Executions of The Program In Figure 2.

the algorithm inserts *R3* right behind symbol 3 in the rule, denoting the lower level composition that are control dependent on 3. Note that the top level rule *RA* does not reflect the syntactic structure of method A as a rule derived from the syntactic structure, i.e., *RA* → $\widetilde{2}$ $\widetilde{3}$ *R3* $\widetilde{7}$ *R7* $\widetilde{12}$, fails to describe executions, e.g., "2 3 4 16 5". Statements 4 and 5 are control dependent on the true branch of 3 and statements 7 and 12 are dependent on the false branch. Adding the intermediate symbols denoting the substructures led by 4, 7, and 12 results in the second rule in Figure 3. The remaining rules are similarly derived.

Recall that our goal is to design an execution indexing technique. Based on EDLs, we are ready to introduce our indexing scheme. As illustrated earlier, any execution of a program $\mathcal{P}$ is a string of $EDL(\mathcal{P})$. The index of an execution point can be defined based on the derivation tree of the string.

DEFINITION 4. **(Structural Indexing)** *Given a program $\mathcal{P}$ and its $EDL(\mathcal{P})$, the structural index of a point x in execution $\mathcal{P}(\vec{i})$, denoted as $SEI^{\mathcal{P}(\vec{i})}(x)$, is the path from the root of the derivation tree to the leaf node representing x.*

According to the definition of EDL, each grammar rule captures the statements at the same nesting level. Therefore, the path in the derivation tree, which leads from the root to a terminal symbol and contains all the intermediate symbols, denotes the top-down nesting structure and serves as a perfect structural index. Figure 4 shows the indices for two executions of the code in Figure 2. Execution E recursively calls A() in the first iteration of the while loop, whereas the recursive call happens in the second iteration in E'. We can see $SEI^{E}(2_1) = SEI^{E'}(2_1) =$ [*RA*], in which 2_1 denotes the first instance of statement 2 in the traces. Thus, the first executed statement 2s correspond to each other in the two executions, as linked by the dotted line. $SEI^{E}(16_1) =$ [*RA*, *R3*, *R7*, *R8*, ...], which clearly expresses the nesting structure of the first executed 16. In contrast, the index of 16_1 in the second execution is $SEI^{E'}(16_1) =$ [*RA*, *R3*, *R7*, *R7*, ...], different from $SEI^{E}(16_1)$. The indices imply that the 16_1 in E' is nested in the second iteration of the while loop while the 16_1 in E is nested in the first iteration. Therefore, the two 16_1s do not structurally correspond. In some situations, the structural correspondence differs from the desired correspondence, we will discuss how we handle these cases in Section 4.

We have defined what is structural indexing. However, we are yet to show that structural indexing is a valid indexing scheme.

THEOREM 1. *The structural indexing function defined in Definition 4 is a valid execution indexing function.*

To prove this theorem, we need to show that no two different execution points in an execution have identical structural indices. The proof is omitted for brevity.

3. Implementation and Optimizations

A faithful implementation of structural indexing according to the definition is not practical because it requires collecting the whole execution trace and then parsing the trace. However, our goal can be interpreted as maintaining the current index for each execution point on the fly, just like maintaining the calling context. In this way, we avoid any form of logging. The user has the freedom to collect the indices for any interesting points such as breakpoints and perturbation points. This interpretation entails highly efficient implementation.

3.1 Indexing Stack

The basic idea is to use an indexing stack (IS) to keep track of the index, which is the set of rules used in parsing the current nesting structure. More specifically, an entry is pushed to IS once a predicate statement or a method call is executed, implying a new rule representing the lower level nesting structure is taken to parse the following execution. The entry is popped if the immediate postdominator of the predicate is executed or the method call is returned, indicating that parsing based on the current grammar rule is finished and the following execution should be parsed based on the parent rule, which is now found in the top entry of the current IS. The algorithm is presented in Algorithm 2.

According to the algorithm, only predicates and method calls and their immediate postdominators are instrumented. The circles and triangles in the CFG in Figure 2 illustrate the instrumentation of calls to `enter ()` and `exit ()`. For instance, statement 3 is instrumented with an `enter ()` call, meaning rule *R3* is employed to parse the following execution. At $EXIT_A$, the statement

Algorithm 2 Maintaining Indexing Stack.

s is an executed predicate or a executed method call;
b is the entry of a basic block;
$IPD(s)$ denotes the immediate postdominator of s;
Each stack entry is a pair, with the first element being the rule and the second element being the terminating IPD.

```
 1: Enter ($s$) {
 2:     if ($s$ is a predicate)
 3:         IS.push($< Rs,\ IPD(s) >$);
 4:     if ($s$ is a method call to $\mathcal{M}$)
 5:         IS.push($< R\mathcal{M},\ IPD(s) >$);
 6: }
 7: Exit ($b$) {
 8:     while (IS.top().second $\equiv b$)
 9:         IS.pop();
10: }
```

3's immediate postdominator, a call to `exit ()` is inserted. Note that a statement may serve as the immediate postdominator of multiple predicates or method calls. For example, statement 10 is the IPD of both 8 and 9. As a result, multiple entries in IS may have the same terminating IPD. The property of dynamic control dependence further dictates that multiple entries with the same terminating IPD must be consecutive in IS [20]. This explains why Algorithm 2 needs to push the terminating IPD to the stack and in lines 8-9 uses a loop to pop all entries with s being the terminating IPD.

trace	instrumentation	indexing stack
2. $s1$	-	$[RA^{XA}]$
3. $if\ (C_1)$	$\downarrow\ (R3^{XA})$	$[RA^{XA}\ \ R3^{XA}\ ]$
7. $while(C_2)$	$\downarrow\ (R7^{12})$	$[RA^{XA}\ \ R3^{XA}\ \ R7^{12}\ ]$
8. $if\ (C_3)$	$\downarrow\ (R8^{10})$	$[RA^{XA}\ \ R3^{XA}\ \ R7^{12}\ \ R8^{10}]$
...	...	...
10. s_2	$\uparrow\ (*^{10})$	$[RA^{XA}\ \ R3^{XA}\ \ R7^{12}]$
7. $while(C_2)$	$\downarrow\ (R7^{12})$	$[RA^{XA}\ \ R3^{XA}\ \ R7^{12}\ \ R7^{12}]$
...	...	...
7. $while(C_2)$	$\downarrow\ (R7^{12})$	$[RA^{XA}\ \ R3^{XA}\ \ R7^{12}\ \ R7^{12}\ \ R7^{12}]$
12. $B()$	$\uparrow\ (*^{12})$ $\downarrow\ (RB^{XB})$	$[RA^{XA}\ \ R3^{XA}\ \ RB^{XB}]$
...	...	...

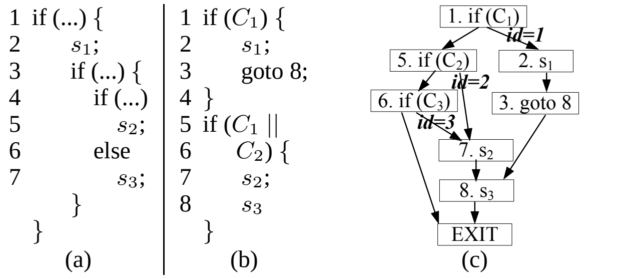
Figure 5. The Maintenance of IS for Execution E in Figure 4. A stack entry $(rule, ipd)$ is represented as $rule^{ipd}$ to save space. ↓ and ↑ stand for *push* and *pop*; XA stands for $EXIT_A$.

Table 5 shows the partial computation of IS for the execution E in Figure 4. At the first step, the IS inherits the entry of RA^{XA} from the preceding call site to `A()` beyond the trace, meaning the current parsing rule is *RA* and the its terminating IPD is $EXIT_A$. The statement executions of 3, 7 and 8 lead to push operations as they are predicates. The instrumentation at 10 pops the entry $R8^{10}$, indicating the current rule of $R8$ terminates. The next two steps of 7 push two $R7^{12}$ entries, representing two loop iterations. The execution of 12 terminates all entries that have 12 as their IPD and pushes a new entry. We want to point out that *the sequence of rules recorded in the IS is exactly the index of the current execution point.*

Algorithm 2 is extended to handle recursive functions and irregular control flow caused by `setjmp/longjmp`. The extension is similar to our prior work [20] and thus not the focus of this paper.

3.2 Optimizations

Algorithm 2 entails easy implementation. However, the algorithm incurs significant runtime overhead as it requires stack operations upon execution of predicates and their immediate postdominators. In this subsection, we discuss how to optimize the algorithm so that it becomes more affordable.



```
1  if (...) {
2        s1;
3        if (...) {
4            if (...)
5                s2;
6            else
7                s3;
         }
   }
```

(a)

```
1  if (C1) {
2        s1;
3        goto 8;
4  }
5  if (C1 ||
6      C2) {
7        s2;
8        s3
   }
```

(b)

Figure 6. Rule Inference.

Rule Inference. The key to reduce runtime overhead is to reduce the number of stack operations. The first optimization is rule inference, which removes stack operations for non-loop predicates and their postdominators. The first observation is that some of the predicate rules can be inferred from the current execution point such that it is not necessary to explicitly record them onto IS. Consider an example in Figure 6 (a). At the moment s_1 is executed, it must be the case that *R1* → $\widetilde{2}\ \widetilde{3}$ *R3* is the active grammar rule because it is the only rule to parse s_1. Similarly, when s_2 is the current execution point, it can be inferred that [*R1 R3 R4*] must be the top three entries on IS. The case for s_3 is similar. In other words, *a predicate and its postdominator are not instrumented if and only if any statements that ever appear in the body of the predicate's rules do not appear in rules of any other predicates*. It is equivalent to not instrumenting a predicate and its postdominator if and only if the predicate's control dependents have only one static controlling predicate.

The second observation is that even the above mentioned condition is not satisfied, the stack operations for non-loop predicates can still be replaced with simple counter operations. Consider the example in Figure 6 (b), whose CFG is presented in Figure 6 (c). Due to the *OR* operation at line 5 and the jump at line 3, statement 8 has three control dependence predecessors: predicates at line 1, 5, and 6. In other words, it appears in the following rules of these three different predicates:

$$\begin{array}{lcl} \mathit{R1} & \longrightarrow & \widetilde{2}\ \widetilde{3}\ \widetilde{8} \mid \widetilde{5}\ \mathit{R5} \\ \mathit{R5} & \longrightarrow & \widetilde{6}\ \mathit{R6} \mid \widetilde{7}\ \widetilde{8} \\ \mathit{R6} & \longrightarrow & \widetilde{7}\ \widetilde{8} \mid \epsilon \end{array}$$

In this case, the parsing rule cannot be inferred from the execution of 8 as it is not unique. To handle this situation, we enumerate the possible rules and use the number as the identification of a rule. Consider the CFG in Figure 6 (c). The instrumentation is highlighted on the flow edges. A variable *id* is used to enumerate the three possible rules. Different flow edges being taken implies different parsing rules, identified by different *id* values. Note that at runtime, only one out of these three edges can be taken.

The optimized instrumentation algorithm is presented in Algorithm 3. Assume $CD(s)$ is the ordered set of predicates on which s is control dependent. It implies that s is parsed by the rule of one of these predicates. Here the code order is used. In lines 4-7, the algorithm first identifies the set of non-singleton CDs that are maximum, namely, they are not subsets of any other CD sets. These maximum sets are stored in MAX. We use a counter id_j for set $MAX[j]$ to identify which predicate out of the set is exercised at runtime, In lines 8-11, the control flow edges that correspond to the predicates in $MAX[j]$ are instrumented by setting id_j to a constant that uniquely identifies the predicate. Here the constant is the order of the predicate in the set. It is worth noting that control dependence is indeed defined between one of the two branch outcome ($True/False$) of a predicate p and a statement s even though we often say the dependence is between p and s for brevity [8]. That explains the superscript at line 10.

Consider the example in Figure 6 (b). The control dependences for 7 and 8 are computed as $CD(7) = [5^T, 6^T]$, and $CD(8) = [1^T, 5^T, 6^T]$. The vector MAX has only one element, which is the maximum set $[1^T, 5^T, 6^T]$. According to the algorithm, a counter id is assigned to the set. All the predicates in the set are instrumented with assignments to id. The resulting instrumentations are exactly those presented in Figure 6 (c).

We can see that (a) counters are not assigned to singleton CDs at runtime and thus the rules can be inferred as discussed earlier; (b) counters are not assigned to CD sets that are subsets of some other sets. Assume x and y are two CD sets, and $x \subset y$ and y is maximum, the instrumentations caused by y are able to distinguish the exercised predicate in x at runtime. For instance, $CD(7) \subset CD(8)$ and thus it is not necessary to associate another counter to $CD(7)$ as the instrumentions of $id = 2$ and $id = 3$ on edges 5^T and 6^T are able to identify which rule should be used to parse an executed instance of 7.

THEOREM 2. *Algorithm 3 is correct, i.e., the parsing rule of a statement execution s can be always decided by the value of a counter.*

Theorem 2 asserts the correctness of Algorithm 3. We can easily derive the decoding algorithm that reconstruct the full execution index from the values of counters. The proof of Theorem 2 and the decoding algorithm are omitted due to the space limit.

Algorithm 3 Instrumentation For Non-Loop Predicates.

$\mathcal{M}$ is a method.
$CD(s)$ denotes the ordered set of non-loop predicates on which s is control dependent.
MAX is a vector storing the maximum CD sets.
$order(cd, p)$ returns the position of p in the ordered set cd.

```
1:  EncodePredicate ($\mathcal{M}$) {
2:      Compute control dependence for $\mathcal{M}$;
3:      $i = 0$;
4:      for each statement $s$ in $\mathcal{M}$
5:          if ($\nexists t.(t \in \mathcal{M} \wedge t! = s \wedge CD(s) \subseteq CD(t))$)
6:              if ($CD(s) \notin MAX$ && $|CD(s)| > 1$)
7:                  $MAX[i++] = CD(s)$;
8:      for each non-loop predicate $p$ in $\mathcal{M}$
9:          for $j$=0 to ($i$-1)
10:             if ($p^{b=True/False} \in MAX[j]$)
11:                 instrument the $b = True/False$ edge of $p$ with
                        "$id_j = order(MAX[j], p)$";
12: }
```

Loop Optimization. As many hot program paths reside in loops, optimizing the instrumentation for loop predicates is also essential to bringing down the cost. A loop predicate decides the execution of an iteration and thus a loop predicate instance at runtime is directly/indirectly control dependent on its previous instance. A loop predicate is different from a predicate inside a loop, which often does not decide the execution of an iteration. Therefore, the EDL grammar rule for a loop always contains a recursion, for example, the rule in the 3rd column in Table 1. Optimization opportunities arise if a loop has a unique predicate [1] as consecutive iterations are thus parsed by a sequence of the same rule that corresponds to the loop predicate. A counter can be used to compress the sequence on IS. Consider the unique loop predicate 7 in E of Figure 4. Symbol $R7$s of consecutive iterations are always consecutive along any paths from the root to a leaf. The optimization is to assign a unique counter to each loop that has one loop predicate. The counter is initialized to zero before entering the loop and then incremented whenever the back edge is taken. Therefore, push operations can be avoided for the loop predicate instances. Even in the presence of multiple loop predicates, we can still use a counter on IS to encode a sequence of identical predicates. Pushes are conducted upon encountering a different predicate.

Packing Multiple Counters. Let us consider counters for non-loop predicates first. Our experience shows that the sizes of CD sets tend to be small. The majority of CD sets are singleton and most of the non-singleton CD sets have the cardinality of 2. The reason is that non-singleton sets are mostly caused by Boolean OR operations or nonstructural control flow. In other words, the value range of a counter we need for non-loop predicates tends to be small because they only need to distinguish elements in small sets. Therefore, we further optimize our implementation by packing multiple counters into one word. We treat each bit of a word as a conceptual register. These bit-registers are allocated to a counter as needed. For instance, a counter with the range of [0-3] is allocated 2 bit-registers. These registers have a live range delimited by the predicates that are associated with the counter and their postdominators. Eventually, the counter packing problem is reduced to register allocation problem and we use standard algorithm to solve the problem.

In our current implementation, we do not analyze the ranges of loop iterators. We conservatively assign a word to each loop counter.

After the above optimizations, most remaining stack operations occur on function boundaries as we need to maintain the IS state across multiple functions. The information that is pushed to the stack usually contains only a few active loop counters and one or two words that contain multiple packed non-loop counters. Finally, handling multi-threading is straightforward, each spawned thread inherits the IS state from its parent.

```
6:   ...
7:   while ($C_2$) {
8:       $c$=getc();
9:       switch ($c$) {
10:          case 'a':
11:              $F(c)$; break;
12:          case 'b':
13:              $G(c)$; break;
14:          case 'c':
15:              $H(c)$; break;
16:      }
17:  }
18:  ...
```

$$\begin{array}{lll} R7 & \longrightarrow & \widetilde{8}\,\widetilde{9}\; R9\; \widetilde{7}\; R7 \mid \epsilon \\ R9 & \longrightarrow & \widetilde{11}\; RF \mid \widetilde{13}\; RG \mid \widetilde{15}\; RH \end{array}$$

is augmented to

$$\begin{array}{lll} R7 & \longrightarrow & \widetilde{8} \mid \epsilon \\ S8a & \longrightarrow & \widetilde{9}\; R9\; \widetilde{7}\; R7 \\ S8b & \longrightarrow & \widetilde{9}\; R9\; \widetilde{7}\; R7 \\ S8c & \longrightarrow & \widetilde{9}\; R9\; \widetilde{7}\; R7 \\ R9 & \longrightarrow & \widetilde{11}\; RF \mid \widetilde{13}\; RG \mid \widetilde{15}\; RH \end{array}$$

Figure 7. Indexing with Semantic Anchor Points.

4. Semantic-Augmented Indexing

One of the most important aims of execution indexing is to establish meaningful correspondence among points across multiple executions. However, as *correspondence* is essentially a semantic concept. It is machine undecidable to conclude if two execution points

[1] Note that a loop may have multiple loop predicates especially when `break` statements are used.

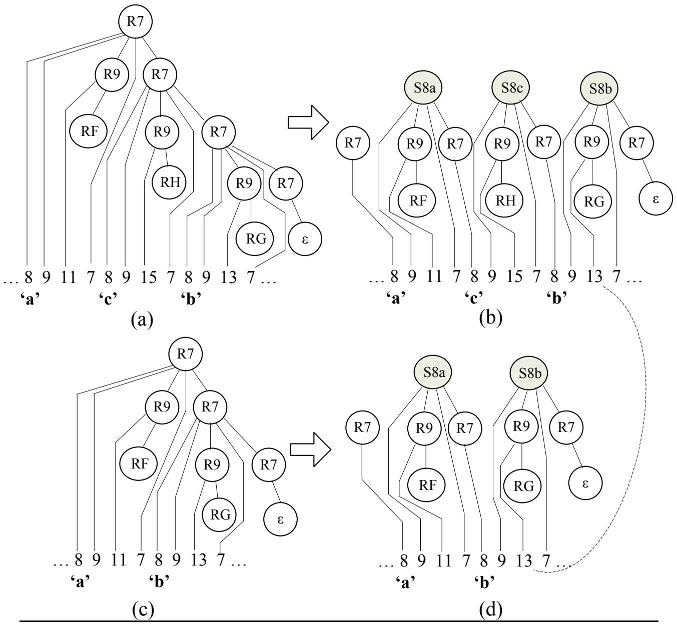


Figure 8. Indexing Executions with Input "acb" and "ab".

in two respective executions correspond to each other. In practice, programmers often decide the correspondence according to their understanding of the executions. For example, if the two executions have their inputs overlapped, the sub-executions driven by the overlapping input elements should correspond. The proposed technique so far constructs indices from execution structure. The advantage of structural indexing is the capability given by the execution description language, which is to confine perturbation in its nesting regions while sustaining correspondence in the remaining part of the execution. However, the technique parses an execution starting from a single point, namely, the beginning of the execution. We call the point an *anchor point*. In many applications, due to non-deterministic scheduling, mis-aligned inputs, execution perturbation, etc., the single *anchor point* is often insufficient to harness the whole execution and thus the technique fails to sustain meaningful correspondence.

An example is presented in Figure 7 and 8. The code is shown on the left hand side of Figure 7 and it is modified from our previous running example in Figure 2. Inside the while loop, the computation is replaced with a statement that gets input and a following `switch-case` statement that calls different functions according to the incoming input values. The EDL grammars are presented on the upper half of the right hand side. Given two executions: one with input "**acb**" and the other with "**ab**", their indices are shown in Figure 8 (a) and (c). The calls to function `G()` at 13 in the two executions do not correspond to each other as they have different indices although the programmer might intend to align them. Such misalignment will lead to no correspondence being identified inside the method body of the two respective `G()` calls. Further inspection shows that structure based indexing decides the second iterations in the two executions correspond to each other while one calls method `H()` and the other calls `G()`.

To overcome this problem, the technique has to acquire help from the programmer. We further extend our technique to incorporate the programmer's knowledge by introducing semantic-based anchor points to harness the whole execution. The idea is to incorporate the values at semantic anchor points into the EDL grammar rules so that different rules are selected to parse execution according to different anchor point values, resulting in semantic-based indices. Consider the previous example, the EDL grammar rules are enhanced as shown on the lower half of the right hand side of Figure 7. Rule *R7* is altered to only parse statement 8. Based on the input values at 8, three different root level rules are introduced. As a result, the derivation tree is transformed to a derivation forest as shown in Figure 8. With the semantic-augmented indexing, the calls to `G()` are successfully aligned.

The semantic-augmented indexing requires the programmer to annotate their intended anchor points by using our predefined C macroes. If aggregate data (e.g. arrays) are used to select a parsing rule, the programmer also need to provide a hash function to map the aggregate data to a single value (see the TSP client study in Section 6.2 for an example). In practice, the places that the programmer needs to annotate tend to be few. They are usually input points such as those reading values from a file or receiving external events.

The extension can be implemented by creating a new stack whenever an anchor point is executed and then pushing the value at the executed anchor point to the new stack. The new stack is maintained exactly as before. The old IS stack is restored once the execution goes beyond the semantic rules. Finally, it is worth mentioning that the same implementation provides the flexibility of selectively indexing execution. More precisely, indexing can be started at defined anchor points instead of from the beginning.

5. Discussion

The challenge of execution indexing has been lurking in a number of prior research projects [9, 23] on dynamic program analysis that require establishing correspondence among points across multiple executions. Although this problem is in general machine undecidable as the correct answer only resides in the programmer's mind, our technique shows the promise of providing a practical solution.

To the best of our knowledge, this is the first work that formulates and tackles the problem and thus it has its limitations. Our technique heavily relies on execution structure based on the observation that execution control structure is a strong indicator of program semantics. However, there exist applications whose control structure is decoupled from the semantics. An example we have encountered is LR parsers such as those generated by *yacc*. Our technique fails to construct meaningful mappings between two executions with highly similar inputs, even with semantic-augmented indexing. The main reason is that the execution of a LR parser is driven by a DFA whose semantics are revealed more by data flow than by control flow. This implies a different direction of execution indexing - data flow based indexing, which we leave to our future work.

6. Experiments

6.1 Overhead

The first experimentation is on the runtime overhead of our indexing technique. The implementation is based on Diablo/FIT [17], a generator that produces customized ATOM-like binary instrumentation tools. Diablo has its own toolchain based on GCC-3.2.2, which is used to compile the benchmarks, generate the desired post-dominance information, as well as the final instrumented programs. Instrumentation are inserted at the start and end of procedures, at the beginning of a basic block that has multiple static CDs, at and before loop head basic blocks, and at programmer defined semantic indexing points. Macros are provided for programmers to define semantic indexing points. Inlining is employed for instrumentation with a small number of instructions. We use a subset of SPEC benchmarks from SPEC95 and SPEC2000. Some of the SPEC benchmarks failed to get through Diablo/FIT infrastructure and thus are not used in our evaluation. All data are collected on a

Benchmarks	Statistics on BBL's CDs					Runtime				
	Total	0 CD	1 CD	2 CDs	(> 1 CDs)	Base	CDS	OPT	CDS Overhead	OPT Overhead
008.espresso	29059	3841	19588	4225	22%	0.9	3.2	1.5	256%	67%
124.m88ksim	28198	3173	19149	4288	23%	71.4	207.5	122.0	191%	71%
129.compress	21508	1877	14484	3774	26%	105.8	192.3	149.4	82%	41%
132.ijeg	26596	3370	17691	4112	24%	27.2	52.8	38.2	94%	40%
164.gzip	23179	2122	15713	3932	25%	3.3	9.2	4.6	179%	39%
175.vpr	28078	3257	19172	4194	23%	22.8	66.0	28.4	189%	25%
181.mcf	21743	1886	14689	3798	26%	54.5	76.4	64.6	40%	19%
197.parser	27636	3076	18849	4238	23%	13.3	30.4	21.7	129%	63%
256.bzip2	22740	2174	15283	3866	26%	23.5	45.6	31.3	94%	33%
300.twolf	31638	3120	22197	4640	22%	31.7	51.4	39.1	62%	23%
Average	-	-	-	-	24%	-	-	-	132%	42%

Table 2. Runtime Overhead and Statistics on Control Dependences.

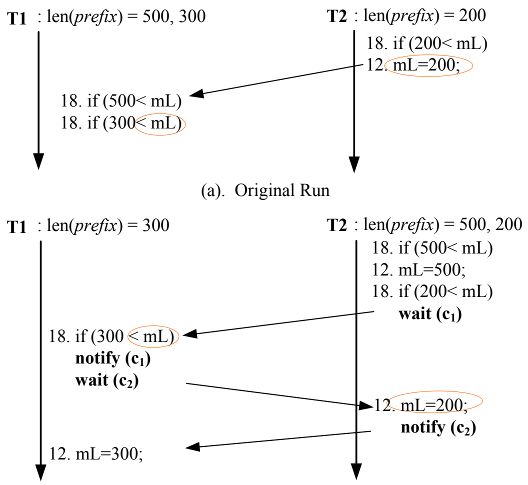
Pentium 4 (1.8GHz) platform with 500M of RAM, running Gentoo Linux (kernel 2.6.22).

We have described the *rule inference* optimization in Section 3.2. The premise for this optimization to work is that the number of basic blocks that have more than one static control dependence is small. The statistics of how many static control dependences a basic block has are shown in Table 2. The *Total* column represents the total number of basic blocks in each benchmark. The next three columns show that number of basic blocks that has 0, 1, and 2 static control dependences, respectively. We can clearly see that across all benchmarks, most of the basic blocks have either 0 or 1 static control dependence; for those basic blocks that have more then 1 static control dependence, majority of them have 2. On average, among the basic blocks that have at least 1 control dependence, only about 24% have more than 1 CD, as shown in the sixth column in the table.

Since FIT is an optimizing instrumentor, for comparison purposes, the *Base* version of the benchmarks are generated through a dummy instrumentor (No instrumentation code is added, but it benefits from all the optimizations available from FIT. Thus, they are actually faster than native runs, which are not shown here.). The *CDS* version corresponds to a faithful implementation of the stack-based algorithm (Algorithm 2) and the *OPT* version corresponds to the optimized implementation. The numbers are presented in Table 2. Both the *CDS Overhead* and *OPT Overhead* are relative to the *Base* runtimes. From the table, we can see that maintaining the execution indices for programs incurs, on average, a 42% runtime overhead. This cuts more than two thirds of the overhead of the *CDS* implementation.

```
1.  void run ( ) {                          15. void remaining (Node [] prefix)  {
2.     Node [ ] prefix;                     16.    …
3.     for (;;) {                           17.    if (prefix.size  == total) {
4.        lock (a);                         18.       if (len (prefix ) < minLen)
5.        prefix=next_prefix (minLen);      19.          set_best(len(prefix), …);
6.        unlock(a);                        20.    } else {
7.        remaining(prefix);                21.       … /*extend prefix*/
8.     }                                    22.       remaining(prefix);
9.  }                                       23        …
10. void set_best (int best, … ) {          24.    }
11.    lock (a);                                 }
12.    minLen=best;
13.    unlock (a);
14. }
```

Figure 9. The Abstract Code of TSP.



(a). Original Run

(b). A Second Run With The Happens-Before Switched

Figure 10. Benign and Harmful Race Discretion by Happens-before Switching.

6.2 Client Study - Lightweight Benign and Harmful Data Race Classification

To show the effectiveness of our indexing technique, we apply it to data race detection. One of the major challenges in data race detection is determining whether a data race is benign or harmful, as data race detection tools often produce many false positives, namely, benign races. In [15], Satish et al. propose classifying confirmed, real data races by switching the happens-before edge between the two dynamic accesses involved. Although the technique is highly effective, their system also relies on a heavyweight tracing and replay system because concurrent execution is nondeterministic.

We propose a lightweight race classification approach for *potential races* based on our indexing technique. The key idea is to first use a lockset algorithm [16], which is lightweight, to identify *potentially racing* accesses. To produce a failure inducing schedule for a pair of racing accesses, two more executions are performed nondeterministically with the two happens-before alternatives enforced and the output is observed. As in [15], a race is classified as harmful if different output is produced, otherwise it is benign, or, in the case of deadlock, the race itself could not be confirmed. We avoid tracing and replay by creating indices for the racing accesses, which are points in the original execution, and identifying them in the re-executions. As a lockset algorithm often produces a large number of racing pairs, we heuristically select accesses to achieve a reduced set of conflicting access pairs. In particular, we first prune conflicts by thread segments [10] and keep track of the indices for those conflicting accesses that occur furthest apart chronologically. The goal is to perturb the computation in between as much as possible when enforcing the two happens-before alternatives to increase the chance of a failure.

Let us consider the technique by observing its behavior on the TSP (travelling salesman problem) benchmark. As our implementation works on C programs, we port the java TSP benchmark to C and remove the double check in the the `setbest()` function to introduce a harmful race. Note that the original TSP does not have any harmful races. The abstract code is shown in Figure 9. As shown in method `run()`, the algorithm enumerates all prefixes with a fixed length and assigns them to individual threads. Each thread takes a prefix and finishes the traversal by calling `remaining()`. The `remaining()` method recursively checks if the current path covers all the nodes. If so, it compares the path length with the current minimum length and updates the minimum length by calling `setbest()` at lines 18-19. If the current prefix is not a full path, it continues traversing. Data races happen between execution instances of lines 12 and 18 as boxed in Figure 9.

The challenge is that *the same thread in different executions may pick up completely different sets of prefixes*, which causes simple indexing schemes to fail. As shown in Figure 10, in the original execution, thread one is assigned the prefix that leads to path lengths of 500 and 300 and thread two is assigned the prefix that leads to the length of 200. In a separate execution with the same input, thread one is assigned the 300 prefix and thread two is assigned the 500 and 200 prefixes. We define 5 as the semantic anchor point so that a new IS is started at line 5 and the current prefix (hash) is pushed to the stack as part of the index, As a result, we are able to locate the two conflicting accesses in the new run and insert synchronization as shown in fig. 10. The synchronization expresses both happens-before orderings of the racing accesses, and eventually produces the correct output of 200 in one execution and the faulty output of 300 in the other. Therefore, this is a harmful race. Note that due to nondeterminism, the conflicting accesses may not appear in the new run. It may be necessary to run the trial a number of times in order to have both accesses occur; however, because of the approach to selecting these conflicting pairs and the small number of scheduling decisions that determine whether or not they occur, their chances of omission are small, as seen by the frequency of successfully detecting harmful races in **TSP**^{2} of Table 3. Here we see that a large number of dynamic conflicts can be reduced to a small set of representative conflicts that are more likely to be harmful when flipped. For TSP, exactly one of these selected dynamic conflicts is harmful, and execution indexing enables its consistent classification.

By comparison, nondeterminism adversely affects simpler indexing schemes. Using the thread, instruction, and dynamic execution count as an identifying tuple is susceptible to *both* scheduling differences and execution perturbation, as previously seen. Thus, such an index may map one execution point to another in a different run or even to a nonexistent execution point. Such indices are frail and failure prone for this classification scheme, as seen by the low frequency of properly classifying races in **TSP**^{1} of Table 3. Here the reduced set is computed analogously by projecting accesses to static instructions to get static conflict pairs from the dynamic ones. In fact, the data race alone could not be detected in 56% of trials using these tuples, much less classified.

We further examine how indexing contrasts to a simple tuple identification scheme on real world races. In particular, we examine previously discovered races in the MySQL database server [2] and the Apache HTTP server [1] as also explored in [21, 12].

MySQL. The MySQL server contained a data race where issuing an `insert` command to the database could conflict with log maintenance such that its log files would omit database modifications [2]. Namely, the log maintenance operation briefly made server logs appear closed, preventing the insert command from recording its operation. This lost update bug can be detected in the trivial case where one client accesses the server on one thread to perform an update and another client accesses the server on a second thread to instigate log maintenance.

Note that, apart from the race in question, thread scheduling will not cause nondeterminism within the clients once they have connected to the server, but the order in which clients themselves connect to the server cannot strictly be guaranteed. Because of this, it is not knowable which of the server's threads must be instrumented using simple tuple identification; however, by semantically anchoring execution indices to the precise requests of the client threads, the threads pertaining to the database update and log maintenance can be distinguished and the appropriate accesses forced to exhibit the race. Under one happens-before direction, a lost update occurs, and under the other, the database operates normally. Thus the race is harmful. Testing this by performing both of the client requests in nondeterministic order confirms the results as reflected in **MySQL**^{2} of Table 3. Here the frequency again tells us that the race was consistently observed as being harmful. Again, simple tuple identification cannot consistently classify this race, yielding the lower frequency of successfully classifying the race in **MySQL**^{1} .

Apache HTTP Server. The Apache webserver contained a data race where threads handling different client requests could simultaneously try to write to a buffered operation log, causing it to become corrupted [1]. In the middle of one thread's update, the other can simultaneously write, and both threads can overwrite each other's information. This data corruption bug can be detected as a potential data race in a simple scenario where two clients submit different requests to the webserver.

In this example, thread nondeterminism influences the nature of the resulting corruption, causing either invalid writes or a lost update. Due to the nature of the signalling in our approach to forcing race expression, a lost update is significantly more likely. The particular update that is lost is determined by which of the two happens-before orders is enforced. In addition, because the update is dependent upon the actual request made by a client, the resulting output depends upon the happens-before ordering being imposed. Thus, as in the MySQL example, under nondeterminism, it becomes critical to identify the threads by the particular client requests to which they respond. Otherwise, the same happens-before ordering may be enforced twice, causing the same output to be produced. When both happens-before orderings are successfully imposed, the logs from the two executions differ, confirming that the race is harmful. In **Apache**^{2} and **Apache**^{1} of Table 3, we see that the frequencies of successful classification parallel those of MySQL, emphasizing the commonality of the difficulties they face under nondeterminism.

Thus, we see that execution indexing offers a tangible, real world benefit over traditional identification approaches using instructions and dynamic execution counts. For the purposes of consistently identifying execution points and identifying semantically appropriate execution points, execution indexing succeeds while traditional methods are prone to failure.

7. Related Work

Our work is related to [23], which tackles execution omission errors. They decide if an implicit dependence exists between an executed statement and a predicate by forcing the branch outcome of the predicate to be its opposite. An execution alignment algorithm was used to align the switched execution to the original so that in-

Test	Dynamic Conflicts	Reduced Conflicts	Harmful Conflicts	Success Frequency
TSP[1]	399614390	13	1	1/50
MySQL[1]	4	4	1	23/50
Apache[1]	7	6	6	27/50
TSP[2]	478861017	19	1	50/50
MySQL[2]	4	4	1	50/50
Apache[2]	7	6	6	50/50

Table 3. Potential and detected harmful races. 1) Using tuple based identification. 2) Using execution indices.

formation collected in the switched run can be migrated back to the original run. The alignment algorithm is based on matching execution regions. It is offline and requires constructing the dynamic program dependence graph. A similar offline algorithm was used by Liang et al. in [9] to identify similar execution traces for fault localization. Neither of these works realized that the challenges are essentially an execution indexing problem, which indeed is a grand challenge for dynamic program analysis and has the potential to impact a wide range of applications. This work, for the first time, formalizes the problem, devises efficient online algorithms, and proposes the concept of semantic anchor points to deliver flexibility.

Execution monitoring detects patterns of events at runtime. In the context of aspect oriented programming [18, 3], languages are proposed to describe event patterns and automata are constructed to parse these patterns at runtime. The goal is to address the programmer's design concerns such as checking if the process of enumeration, comprising a sequence of access events to the enumerator object, is intervened by a modification to the underlying collection. PQL [13] designs a query language to provide similar pattern matching at runtime for the purposes of debugging and security. PQL is able to recognize non-regular languages. In comparison, our work has a similar observation that describing execution by context free languages provides the capability of ignoring irrelevant part of an execution. However, our work has a different goal and does not require the programmer to pre-define pattern languages.

This work is tightly related to dynamic control dependence detection. The existing work [14, 20] disclose that dynamic control dependence has a stack-like structure at runtime so that a control dependence stack is proposed to detect dependence. Our technique is based on dynamic control dependence detection. However, as our goal is to index program execution, which has a different set of applications and higher requirement on runtime overhead, we focus more on defining the concepts of execution indexing and optimizing the algorithms.

This work is related to program trace representation, which records program execution at a fine-grained level for later inspection. Depending on applications, various information can be traced such as control flow [11, 24], values [5, 4], addresses [6], and so on. Existing tracing techniques focus on compressing traces. Our technique focuses on providing meaningful identification for execution points. It does not rely on traces. In particular, whole program paths [11] represent traces with grammar rules. However, the rules are derived for the optimal compression performance and cannot be used for the purpose of indexing.

8. Conclusion

We propose a novel dynamic analysis called execution indexing which provides a unique identification for an execution point so that points in one execution can be correlated and points across multiple executions can be aligned. Execution indexing can serve as a cornerstone in various applications such as profiling, debugging especially in the presence of nondeterminism, dynamic instrumentation and so on. We formally define the concepts of execution indexing, devise a highly optimized online algorithm, and conduct a client study on applying execution indexing to a light weight method to produce a failure inducing schedule for a data race warning without relying on a tracing and replay system.

Acknowledgments

This work is supported by grants from NSF grants CNS-0720516 and CNS-0708464 to Purdue University.

References

[1] Apache bug. http://issues.apache.org/bugzilla/show_bug.cgi?id=25520.

[2] MySQL bug. http://bugs.mysql.com/bug.php?id=791.

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 345–364, San Diego, CA, USA, 2005. ACM Press.

[4] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154–163, Ottawa, Canada, June 2006. ACM.

[5] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 159–169, New Orleans, Louisiana, 2003.

[6] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, Snowbird, UT, June 2001. ACM.

[7] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Research Triangle Park, NC, 1997.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[9] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *15th International Conference on Compiler Construction*, pages 80–95, Vienna, Austria, 2006.

[10] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, London, UK, 2000. Springer-Verlag.

[11] J. R. Larus. Whole program paths. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 259–269, Atlanta, Georgia, May 1999. ACM.

[12] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, San Jose, CA, 2006.

[13] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, San Diego, CA, 2005.

[14] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proceedings of the 15th International*

Symposium on Software Reliability Engineering (ISSRE'04), pages 198–209, Saint-Malo, Bretagne, France, 2004.

[15] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 22–31, San Diego, CA, June 2007.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[17] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, Athens, Greece, December 2005. IEEE.

[18] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 159–169, Newport Beach, CA, 2004.

[19] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *Proceedings of the 26th International Conference on Software Engineering*, pages 512–521, Edinburgh, United Kingdom, May 2004.

[20] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 185–195, London,UK, July 2007. ACM.

[21] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 1–14, Chicago, IL, 2005. ACM.

[22] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, Charleston, SC, November 2002. ACM.

[23] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 415–424, San Diego, California, USA, June 2007. ACM.

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