DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning

Haipeng Cai
University of Notre Dame
Indiana, USA
hcai@nd.edu

Raul Santelices
University of Notre Dame
Indiana, USA
rsanteli@nd.edu

ABSTRACT

Impact analysis determines the effects that the behavior of program entities, or changes to them, can have on the rest of the system. Dynamic impact analysis is one practical form that computes smaller *impact sets* than static alternatives for concrete sets of executions. However, existing dynamic approaches can still produce impact sets that are too large to be useful. To address this problem, we present a novel *dynamic* impact analysis called DIVER that exploits *static* dependencies to identify *runtime* impacts much more precisely without reducing safety and at acceptable costs. Our preliminary empirical evaluation shows that DIVER can significantly increase the precision of dynamic impact analysis.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Dynamic impact analysis; trace pruning; dependence analysis

1. INTRODUCTION

Modern software is increasingly complex and changes rapidly, posing serious risks to its quality and reliability. Thus, it is crucial to analyze and evolve these systems efficiently and effectively. A key activity in this process is *impact analysis* (e.g., [9, 20]), which identifies the effects that the behavior of program entities or changes to those entities can have on the rest of the software.

Different approaches to impact analysis provide different tradeoffs in accuracy, costs, and other qualities for computing *impact sets* (i.e., potentially affected entities). Static analysis can produce safe but overly-conservative impact sets [10]. Dynamic impact analysis, in contrast, uses runtime information such as coverage [17] or execution traces [14] to produce smaller and more focused impact sets than static analysis at the expense of some safety [5, 15, 18]. Yet, users looking for the *actual* behavior of the software, as represented by a set of executions, may afford unsafe results [14], making dynamic impact analysis an attractive option.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642950.

Different dynamic impact analyses also provide different cost-effectiveness tradeoffs. For example, COVERAGEIMPACT [17] is based on runtime coverage and ignores execution order, which makes it very efficient but also very imprecise [18]. Another technique, PATHIMPACT [14], is more precise by using execution order but is less efficient because it requires tracing [18]. For intermediate tradeoffs, optimizations of PATHIMPACT have been proposed [3, 4], including an incremental version [13]. The most efficient one that preserves the precision of PATHIMPACT is based on *execute-after sequences* (EAS) [3], which we call PI/EAS.

Unfortunately, approaches such as PI/EAS can still have too many false positives. Our studies [6] show that they can have a low *change-impact-prediction* precision of 50% on average. An alternative is forward dynamic slicing [1], but it works at the statement level which makes it expensive and would have to be applied to all statements in a method. At the method level, hybrid techniques [16] combining static and dynamic analysis have been proposed to improve precision, such as INFLUENCEDYNAMIC [5]. However, these techniques improve precision over PI/EAS only marginally and at a much greater cost [5]. Another hybrid technique combines runtime coverage with text analysis [9] but it remains unclear how it performs with more precise dynamic data.

To address this imprecision, we present a new dynamic impact analysis called DIVER that incurs the *same runtime costs* as PATHIMPACT but can be much more precise. PI/EAS uses the execution order of methods to find all potential runtime impacts of a method m, which can be quite imprecise because, in general, not all methods executed after m are affected by m. DIVER, in contrast, uses a one-time static dependence analysis to *safely* reduce the impact set to only those methods that could have depended on m at runtime. Although applying that information increases the cost of querying for impact sets, the cost per query is still acceptable and multiple queries can be processed in parallel.

We implemented DIVER for a preliminary study on four Java subjects and we compared the results with those of PI/EAS—the most precise of the efficient techniques in the literature.[1] (Our implementation of DIVER and PI/EAS is available to the public for download.[2]) Our results confirm the imprecision of PI/EAS, whose impact sets often contain *hundreds* of methods. DIVER, in contrast, computed impact sets containing in most cases a few dozen methods. Our findings for these subjects are dramatic: the impact sets of DIVER are 30.8% the size of the impact sets of PI/EAS for the *same level of safety*. This means that DIVER improved the precision of a representative existing technique by a factor of 3.33 (i.e., by 200%). We also found that its costs are acceptable at less than half a minute per query on average.

[1] INFLUENCEDYNAMIC [5] is marginally better but costlier.
[2] The DIVER website is http://nd.edu/~hcai/diver

```
1 public class A {
2    static int g; public int d;
3    String M1(int f, int z) {
4       int x = f + z, y = 2, h = 1;
5       if (x > y)
6          M2(x, y);
7       int r = new B().M3(h, g);
8       String s = "M3val: " + r;
9       return s;}
10   void M2(int m, int n) {
11      int w = m - d;
12      if (w > 0)
13         n = g / w;
14      boolean b = C.M5(this);
15      System.out.print(b);}}
```

```
16 public class B {
17    static short t;
18    int M3(int a, int b) {
19       int j = 0;
20       t = -4;
21       if ( a < b )
22          j = b - a;
23       return j;}
24    static double M4() {
25       int x = A.g, i = 5;
26       try {
27          A.g = x / (i + t);
28          new A().M1(i, t);
29       } catch(Exception e) { }
30       return x;}}
```

```
31 public class C {
32    static boolean M5(A q) {
33       long y = q.d;
34       boolean b = B.t > y;
35       q.d = -2;
36       return b;}
37    public static void
38      M0(String[] args) {
39       int a = 0, b = 3;
40       A o = new A();
41       String s = o.M1(a, b);
42       double d = B.M4();
43       String u = s + d;
44       System.out.print(u);}
45    }
```

Figure 1: The example program E used for illustration throughout this paper.

PATHIMPACT: M0 M1 M2 M5 r r M3 r r M4 r r x
DIVER: $M0_e$ $M1_e$ $M2_e$ $M5_e$ $M2_i$ $M1_i$ $M3_e$ $M1_i$ $M0_i$ $M4_e$ $M4_i$ $M0_i$ x

Figure 2: Execution traces of E used by PATHIMPACT and DIVER.

2. MOTIVATION AND BACKGROUND

This section presents the motivation and necessary background of our technique. Figure 1 shows the example program E we use for illustration throughout the paper.

Motivation. In previous work, we found that existing dynamic impact analyses such as PI/EAS can be too imprecise in practice [6]—on average, only about half of the methods reported as impacted are really impacted. In particular, for methods at the core of a software system, PI/EAS can include in the impact sets for such methods most or all methods in the system. For example, if querying the entry method of the Java application JMeter, the developer using PI/EAS will end up inspecting all 732 methods executed by the test suite, making impact analysis virtually impossible to use in practice. Therefore, because analyzing potential impacts is critical before applying changes, much smaller and precise impact sets are desirable for developers, as long as *safety* is preserved (i.e., no real dynamic impacts are missed).

Program Dependencies. Program dependencies are classified as control or data dependencies [19]. A statement s_1 is *control dependent* [8] on a statement s_2 if a branching decision taken at statement s_2 determines whether statement s_1 is necessarily executed. A statement s_1 is *data dependent* [2] on a statement s_2 if a variable v defined (written) at s_2 is used (read) at s_1 and there is a *definition-clear path* in the program for v (i.e., a path that does not re-define v) from s_2 to s_1. A *dependence graph* [8] is a *static* program representation where nodes represent statements and edges represent both control and data dependencies among those statements.

Dynamic Impact Analysis. In this paper, we focus on analyses that identify dynamic impacts that occur on individual program versions, without changes being necessarily involved [14]. Such a dynamic impact analysis takes a program P, a test suite T, and a set of methods M and outputs an *impact set* containing the methods in P potentially impacted by M when running T. One example technique is PATHIMPACT [14], which collects runtime traces of executed methods. For each method m in M that is queried for its impacts, PATHIMPACT uses the method execution order found in the runtime traces of P for T. The analysis identifies as impacted m and all methods executed in any trace after m.

Figure 2 shows an example trace for PATHIMPACT, where `r` is a method-return event and `x` the program-exit event. The remaining marks are the entry events of methods. For query `M2`, in addition to `M2` itself, PATHIMPACT first finds `M5`, `M3`, and `M4` as impacted because they were entered after `M2` was entered and then finds `M0` and `M1` because they returned after `M2` was entered (i.e., parts of them executed after parts or all of `M2`). Thus, the resulting dynamic impact set of `M2` is {`M0`, `M1`, `M2`, `M3`, `M4`, `M5`} for this trace. When more than one trace exists, PATHIMPACT returns the union of the impact sets for all traces.

EAS [3] optimizes PATHIMPACT by tracking only the first and last events per method. Thus, we consider PI/EAS the most *cost-effective* and *representative* dynamic impact analysis comparable to DIVER. One other technique is more precise than PI/EAS, but for a very small margin and a greater cost [5].

3. TECHNIQUE

For our new impact analysis DIVER to be *safe* with respect to an execution set and also precise and practical, we need something much better than the *execute-after* relation of PI/EAS, which is too conservative. The problem is that reaching a method m' after a method m at runtime is necessary for m to impact m', but not all such methods m' necessarily depend on m. To fix this problem, we propose to build first the (whole-program, static) dependence graph and then use it to find which of those methods really depend on m.

3.1 Overview

The process of computing the dependence graph and using it for dynamic impact analysis is shown in Figure 3. It works in three phases: static analysis, runtime, and post-processing. The inputs for the entire process are a program P, a test suite T, and impact-set queries M. To optimize the static-analysis phase, the process first runs a profiler which executes the test suite on the program to quickly find whether any exceptions are raised by a method but are not caught there or not caught at all. This lets the static-analysis phase decide whether it can *safely skip* computing some control dependencies caused by unhandled exceptions.

After profiling, the static-analysis phase computes data dependencies (DD) and control dependencies (CD). For the CDs, the process first computes *regular* CDs caused by branches, polymorphic calls, and intraprocedural exception control-flows. For the remaining exception control-flows [22] (*ExInterCDs* in Figure 3), the process computes the CDs for the exception types that the profiler detected as not handled by the originating methods. After all dependencies are found, the dependence graph is built and passed to the *post-processing* phase (i.e., not needed at runtime).

For the runtime phase, the static analysis creates the instrumented version P' of P using only probes for monitoring *method-entry*

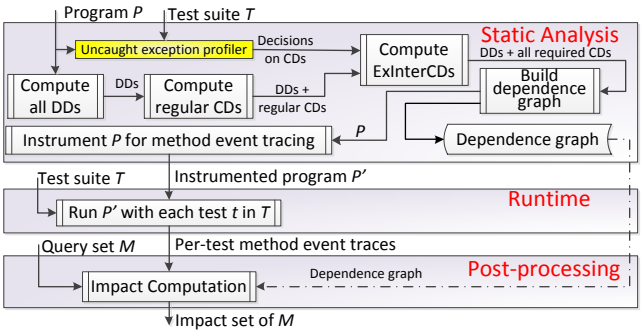

Figure 3: Process for dynamic impact analysis using DIVER.

and *returned-into* events (Section 2). The instrumented program P' is similar to that of PI/EAS except that, instead of tracking two values per method, it traces the *whole* sequence of method events—as PATHIMPACT does—because DIVER needs entire traces to determine *transitive* dependencies at post-processing. Nevertheless, DIVER compresses these traces on the fly at reasonable costs, as PATHIMPACT does, to make space costs acceptable. In all, the runtime phase executes P' with test suite T to produce one compressed method-level trace per test case.

The post-processing phase lets the user query the impact set of a method set M. Any number of queries can be made at this point *without re-running the first two phases*. For each query, DIVER uses the dependence graph from the static phase and the traces from the runtime phase to identify all methods that depended directly or transitively on any method in M on any of those traces. M is included in the result because every method impacts itself.

To illustrate, consider the trace for input $\langle a{=}0,\ b{=}{-}3\rangle$ in Figure 2 where subscripts e and i denote the entry and returned-into events, respectively. For a query $M = \{\texttt{M2}\}$, DIVER traverses the trace to find which dependencies (from the dependence graph) are exercised due to methods executed after `M2` and, via those dependencies, which methods depended directly or transitively on any occurrence of `M2`. When DIVER finds `M2`, the impact set starts as $\{\texttt{M2}\}$. Then, the only outgoing dependence from `M2` in the graph is exercised because its target `M5` occurs next, so `M5` is impacted. Thus, DIVER finds the impact set $\{\texttt{M2}, \texttt{M5}\}$, in contrast with PATHIMPACT which reports $\{\texttt{M0}, \texttt{M1}, \texttt{M2}, \texttt{M3}, \texttt{M4}, \texttt{M5}\}$.

3.2 Dependence Graph and Propagation

The static dependence graph of the entire program is a key ingredient of our technique. Unlike the *system dependence graph* [10], however, the dependence graph built in the static phase of our technique does not include summary edges because DIVER is dynamic after all and, thus, does not require context-sensitive analysis. DIVER uses this graph only to prune runtime traces.

Interprocedural (i.e., across methods) DDs in the dependence graph are classified into three types: *parameter* DDs from actual to formal parameters in method calls, *return* DDs from return statements to caller sites, and *heap* DDs from definitions to uses of *heap variables* (i.e., dynamically-allocated variables not passed or returned explicitly by methods). *Parameter* and *return* DDs are exercised at runtime only if the target method executes *immediately* after the source. Thus, the type of a DD lets DIVER at post-processing decide whether the dependence was exercised and the target method of that dependence was impacted by the source.

To facilitate our presentation of DIVER, we refer to the specific target statements of incoming interprocedural dependence edges to, and the source statements of outgoing interprocedural edges from, a method as *incoming ports* (*IP*s) and *outgoing ports* (*OP*s) of that method, respectively. An impact propagating to a method via an incoming edge e will enter the method through the *IP* for e. If an impact propagates beyond this method through outgoing edges, it will exit through all *OP*s that are *reachable via intraprocedural (i.e., within the method) edges* from the *IP* for e. An impact that starts in a method will propagate through *all OP*s of that method.

3.3 Impact Computation

The post-processing phase of DIVER answers queries for impact sets using the dependence graph from the static phase and the traces from the runtime phase. Algorithm 1 formalizes this process.

Algorithm 1 : COMPIS(Dependence graph G, trace L, method c)

```
1:  ImpOPs := ∅   // map of edge type to set of impacted OPs
2:  ImpactSet := {c}  // impact set of c
3:  start := false, pre_m := null  // preceding method occurrence
4:  for each method event e∈L do
5:      if ¬ start then {start := m(e) = c; if ¬start then continue}
6:      if e is a method-entry event then
7:          if m(e)=c then
8:              for each outgoing edge oe from n(m(e)) in G do
9:                  ImpOPs[type(oe)] ∪= {src(oe)}
10:             pre_m := m(e) // method occurrence; continue
11:         for each incoming edge ie to n(m(e)) in G do
12:             if type(ie)=return∨src(ie)∉ImpOPs[type(ie)] then
13:                 continue
14:             ImpactSet ∪= {m(e)}
15:             for each outgoing edge oe from n(m(e)) in G do
16:                 if src(oe) is reachable from tgt(ie) in G then
17:                     ImpOPs[type(oe)] ∪= {src(oe)}
18:     else  // e is a method-returned-into event
19:         for each incoming edge ie to n(m(e)) in G do
20:             if type(ie)=parameter∨src(ie)∉ImpOPs[type(ie)] then
21:                 continue
22:             ImpactSet ∪= {m(e)}
23:             for each outgoing edge oe from n(m(e)) in G do
24:                 if src(oe) is reachable from tgt(ie) in G then
25:                     ImpOPs[type(oe)] ∪= {src(oe)}
26:     if pre_m=m(e) then {continue}
27:     for each edge type t ∈ {parameter, return} do
28:         ImpOPs[t] \= {z | z ∈ ImpOPs[t] ∧ m(z) = pre_m}
29:     pre_m := m(e) // preceding method occurrence
30: return ImpactSet
```

The algorithm inputs a dependence graph G, an execution trace L and a queried method c, and outputs the impact set of c. *m(e)* gives the method associated with a method event e; *n(m)* is the set of dependence-graph nodes for all statements in method m; *m(z)* is the method to which port z belongs; *src(d)*, *tgt(d)*, and *type(d)* are the source node, target node, and type of edge d, respectively.

To maximize precision, an interprocedural edge d exercised for the i^{th} time in the trace propagates an impact to its target (IP port) only if the source (OP port) of d for that i^{th} occurrence has also been impacted. To that end, an impacted *OP* set per edge type, which starts empty at line 1, is maintained at lines 9, 17, and 25. These sets track impact propagations on ports to ensure that only the methods transitively reachable from c through impacted ports are reported as impacted. The impact set starts with the queried method c (line 2) and grows as the trace is traversed (lines 4–29).

Methods executed before the first occurrence of c cannot be impacted, so their events are skipped using a flag *start* (lines 3 and 5). Methods executed after c are checked to determine if they are impacted, for which two key decisions are made. First, the algorithm decides that the impact of c *propagates into* a method

m(e) if there is an impacted port in *ImpOPs* that is the source of an interprocedural edge of the same type to *m(e)* (lines 11–14 for method-entry events and lines 19–22 for returned-into events).

The second key decision is to determine whether an impact *propagates out* of *m(e)* by finding the *OP* ports of *m(e)* that are reachable, via *intraprocedural* edges inside the method, from the *impacted IP* ports of that method (i.e., the target ports of impact-propagating edges according to the first decision). Those impacted *OP*s are added to *ImpOPs* to continue looking for impacts in the rest of the trace (lines 15–17 for method-entry events and lines 23–25 for returned-into events). As for the queried method c, all of its *OP*s are added to *ImpOPs* when c executes (lines 7–10).

To determine impact propagations through interprocedural edges on the dependence graph, those edges can be classified into two categories, described next. All edges in each category share the same propagation rules.

Adjacent edge. DD edges of types *parameter* and *return* are classified as *adjacent* edges. An adjacent edge from method m to method m' models an interprocedural DD between the two methods. Through these edges, an impact can propagate from m to m' only if m' executes immediately after m. To realize this rule, an *OP* z that is the source of an adjacent edge is added to the impacted *OP* set *ImpOPs*, as other impacted *OP*s, when found to propagate the impact beyond *m(z)* in the trace. The port is then removed from that set (lines 26–28) after matching it to an *IP* in the immediate caller or callee because the corresponding parameter or return value should not be matched to *IP*s of methods that occur later in the trace. The method occurrence is tracked by *pre_m*, initialized at line 3 and updated at lines 10 and 29.

Execute-anytime-after edge. All other interprocedural edges are *execute-anytime-after* edges. Such an edge from m to m' models an interprocedural CD or *heap* DD between these two methods such that an impact in m propagates to m' if and only if m' executes *anytime* after m in the trace. Such edges propagate impacts to their targets if their sources (*OP*s) are *impacted* when the targets are reached later. Thus, the sources of these edges are never removed from the impacted *OP* set *ImpOPs* once added to that set.

It is worth noting that the way in which propagations rules are applied depends on the type of method event being processed in the trace. For instance, no *return* edges are considered for impact propagation at method-entry events (lines 12–13) and no *parameter* edges are considered at returned-into events (lines 20–21) because of the semantics of those event types (see Section 2). Also, all *OP*s of the queried method c are marked as impacted at each entry event found for c. Thus, it is not necessary to do the same for the returned-into events of c because the *OP*s of c are already marked as propagated at the entry of c in the trace.

In sum, according to the propagation rules of DIVER, for each event in the trace, the method associated with that event is added to the impact set if it is determined that at least one of its *IP*s is directly or transitively impacted by the queried method. After all events of the trace are processed in order, the algorithm returns as its output the resulting impact set for that trace (line 30). If multiple traces are available, one run of the algorithm per trace is required and the result is the union of the individual impact sets. Also, for the impact of multiple methods, the algorithm can be run once per method or can be easily adjusted to treat c as a set for efficiency.

4. EVALUATION

This section presents our preliminary empirical evaluation of DIVER. Our goal was to assess the precision of this new technique and its practicality in terms of time and space costs.

4.1 Implementation

Exception-handling Fix for PI/EAS. The original description of PI/EAS [3] deals with exceptions handled in the raising method or its caller. However, if neither method handles the exception at runtime, the *returned-into* events for all methods in the call stack that do not handle the exception will not be logged and those methods can be mistakenly missed in the resulting impact set. To address this problem, we implemented a *corrected* version of PI/EAS, which we call PI/EAS$_C$. PI/EAS$_C$ captures all returned-into events by wrapping the entire body of each method in a try-catch block to identify uncaught exceptions. The added catch block, when reached by such an exception, adds the corresponding returned-into event (which would be missed otherwise) and then *re-throws* the exception to continue the execution, thus preserving the original semantics of the program.

DIVER. To build the dependence graph, we used our dependence-analysis system DUA-FORENSICS [21]. For exceptional control dependencies, our implementation takes the *exceptional control flow graph* (ExCFG) provided by Soot [12] and applies both the classical algorithm for control-dependence computation [8] and the extended algorithm for interprocedural control dependencies [22]. When computing interprocedural-exception CDs, DIVER includes in the *throwable* set of each ExCFG node all exceptions, both *checked* (declared) and *unchecked* (undeclared) for that method, thrown by that node due to a throw instruction in it or a method that it calls that can throw unhandled exceptions.

4.2 Experiment Setup

Subjects. We chose four Java programs of different types and sizes, as summarized on the first three columns of Table 1, for this preliminary study. The size of each subject is measured as the number of non-comment non-blank lines of code (LOC) in Java. Schedule1 is a priority scheduler. NanoXML is a lean and efficient XML parser. Ant is a cross-platform build tool. XML-security is an Apache library for signatures and encryption. We took these subjects and their test suites from the SIR repository [7] and picked the first available version of each one in that repository.

Methodology. For our experiments, we applied PI/EAS$_C$ and DIVER separately to each subject on a Linux workstation with a Quad-core Intel Core i5-2400 3.10GHz processor and 8GB DDR2 RAM. To obtain the method traces, we used the entire test suites provided with the subjects. To compare the analysis *precision*, we calculated for each query the impact set size for DIVER and PI/EAS$_C$ and the *size ratio* of the first one to the second one. To measure and compare the *efficiency* of the techniques, we first computed for each subject the time and space costs of their respective static-analysis and runtime phases. We did this only once per subject and technique because all queries performed later *reuse* the results of the first two phases. Then, for the post-processing phase, we collected the time costs *per query*.

4.3 Results and Analysis

In this section, we report and discuss the relative precisions of DIVER and PI/EAS$_C$ and the costs that both techniques incur.

4.3.1 Precision

Table 1 presents the precision results for DIVER and PI/EAS$_C$, with two statistics per subject and overall for all queries (last row) for the corresponding data points: the *mean* and the standard deviation (*stdev*) of the impact set sizes and ratios. The *#Queries* column lists the number of single-method queries versus the method total per subject (in parenthesis), which is equal to the respective method-level *test coverage*.

Table 1: Precision in terms of impact set sizes and their ratios for DIVER to PI/EAS$_C$.

Subject	#LOC	#Tests	#Queries (#all methods)	PI/EAS$_C$ IS Size mean	PI/EAS$_C$ IS Size stdev	DIVER IS Size mean	DIVER IS Size stdev	IS Size Ratio mean	IS Size Ratio stdev	Wilcoxon p-value
Schedule1	290	2650	20 (24)	18.0	1.6	12.8	4.7	71.3%	24.5%	6.65E-05
NanoXML	3,521	214	172 (282)	82.6	48.1	37.1	28.9	51.7%	33.1%	2.40E-30
Ant	18,830	112	607 (1863)	159.5	173.4	17.9	34.3	25.7%	33.6%	2.94E-100
XML-security	22,361	92	632 (1928)	199.8	168.4	45.1	68.1	28.8%	30.3%	4.79E-102
Average				166.2	164.9	32.2	53.1	**30.8**%	**33.3**%	**9.29E-07**

Table 2: Time and space costs of DIVER and PI/EAS$_C$, including the overheads of profiling uncaught exceptions for DIVER.

Subject	Prof.	Static analysis phase		Runtime phase			Post-processing phase				Execution data size	
							PI/EAS$_C$		DIVER			
		PI/EAS$_C$	DIVER	Normal	PI/EAS$_C$	DIVER	mean	stdev	mean	stdev	PI/EAS$_C$	DIVER
Schedule1	12.7s	4.8s	5.6s	4.0s	10.1s	15.7s	0.7s	0.1s	14.6s	6.0s	1.0M	8.2M
NanoXML	12.1s	11.3s	14.4s	0.4s	1.0s	5.4s	0.1s	0.1s	6.2s	8.8s	0.4M	2.4M
Ant	29.2s	27.3s	142.4s	1.2s	1.5s	2.0s	0.1s	0.1s	3.2s	7.6s	1.0M	2.0M
XML-security	37.1s	33.4s	157.7s	4.3s	4.8s	14.8s	0.0s	0.0s	7.4s	9.6s	0.5M	3.8M
Average	**30.4s**	**27.8s**	**131.9s**	**2.5s**	**3.0s**	**8.3s**	**0.1s**	**0.1s**	**5.6s**	**8.6s**	**0.7M**	**2.9M**

The results in the table show that, on average, DIVER impact sets were much smaller than for PI/EAS$_C$, especially for the two largest subjects. Large numbers of false positives for PI/EAS$_C$ were identified as such and pruned by DIVER. For example, PI/EAS$_C$ identified 160 methods on average in its impact sets for Ant, whereas DIVER reported only 18 for a mean ratio of 25.7%. (These values are means of ratios—not ratios of means.) Also, the large standard deviations indicate that the impact-set sizes fluctuate greatly across queries for every subject except Schedule1. The results suggest that DIVER is even stronger with respect to PI/EAS$_C$ for larger subjects, which are more representative of modern software. For the smaller subjects Schedule1 and NanoXML, DIVER provides smaller gains possibly due to the proximity and interdependence of the few methods they contain.

We applied the Wilcoxon signed-rank one-tailed test [25] for all queries in each subject and also for the set of all queries in all subjects. This is a *non-parametric* test that makes no assumptions on the distribution of the data. The last column in Table 1 shows the resulting p-values. For $\alpha = .05$, the null hypothesis is that DIVER is not more precise than PI/EAS$_C$. The p-values show strongly that the null hypothesis is rejected and, thus, the superiority of DIVER is statistically significant for these subjects and test suites.

In all, DIVER can *safely* prune 70% of the impact sets computed by PI/EAS$_C$, which amounts to an increase in precision by a factor of 3.33 (i.e., by 200%) over the almost-best existing technique.

4.3.2 *Efficiency*

Table 2 reports the time costs of each phase per technique, including the uncaught exception profiling (*Prof.*), static analysis, execution for the non-instrumented program (*Normal*) and for both techniques, and post-processing. For the last phase, we show per subject and overall the means and standard deviations of query costs. The last row shows averages weighted by *#Queries*.

The profiling numbers suggest that automatically finding the static-analysis settings is cheap—half a minute or less. As expected, for static analysis, DIVER incurred higher costs than PI/EAS$_C$. For both techniques, these costs increase with the size of the program, with DIVER growing faster. However, on average the DIVER static analysis finished within 2.2 minutes, which seems reasonable because this is done *only once* per program for all possible queries. For the runtime phase, both techniques had small overheads. For the post-processing phase, due to the traversal of longer traces, DIVER needed more time than PI/EAS$_C$. Yet, the average cost of 5.6 seconds per query still seems quite practical.

The space costs for the runtime data for the two techniques are shown on the rightmost two columns of Table 2. As expected, the DIVER traces use more space than the PI/EAS$_C$ registers. One expected correlation is that longer traces lead to greater post-processing costs. In addition, DIVER incurs on average only 3MB cost for storing the dependence graph during static analysis.

In all, DIVER achieved significantly greater precisions for these subjects at acceptable time and space costs.

4.4 Threats to Validity

The main *internal* threat is the possibility of implementation errors in DIVER and our study scripts. However, DIVER is based on Soot and DUA-FORENSICS, both of which have matured over the years, and we verified the scripts manually for each experiment phase. Another *internal* threat is the risk of missing static dependencies due to Java language features such as reflection and multi-threading. However, we confirmed that, for our study subjects running on their test suites, there was no use of such features except for Ant where reflection is used. Thus, we refactored Ant's code to obtain a reflection-free yet *semantically-equivalent* version (at least for the test suite).

The main *external* threat is that the subjects we used do not necessarily represent all types of programs from a dynamic impact-analysis perspective. Another *external* threat is inherent to dynamic analysis: the test suites we used cannot exercise all behaviors of the respective subjects. Thus, our results must be interpreted in light of the extent of the ability of those test suites to exercise their subjects. To address these issues, we chose subjects of diverse sizes and functionality types for which reasonable test suites are provided.

The main *construct* threat is that we used impact set sizes as inverse indicators of precision assuming that recall is not affected. This is safe for analyzing individual program versions, but for predicting the impacts that *actual* changes will have, recall might be less than perfect [6] if those changes modify the control flow of the program to execute methods not reported by DIVER.

A *conclusion* threat is that we statistically analyzed only methods for which we could obtain impact sets (i.e., executed at least once), but this is safe for our comparison with other dynamic techniques.

5. RELATED WORK

Law and Rothermel introduced PATHIMPACT [14] to compute dynamic impacts based on the execution order of methods and Apiwattanapong and colleagues proposed EAS [3] to safely reduce

the size of PATHIMPACT traces. This technique improves the efficiency of PATHIMPACT but not its precision. DIVER also uses the whole method-level execution traces, as PATHIMPACT does, but it does so while *improving* its precision significantly.

Impact-analysis techniques based on dependence analyses other than static slicing have been explored as well. Sun and colleagues proposed OOCMDG [23] and LoCMD [24] to model dependencies among classes, methods, and class fields. Their techniques were also extended for impact analysis with hierarchical slicing and for multiple levels of granularity. These models, however, include only structural dependencies (e.g., call edges) based on object-oriented features whereas our dependence graph models all interprocedural data and control dependencies for all types of software.

INFLUENCEDYNAMIC [5] combines dependence analysis and dynamic information for impact analysis. However, it considers only a subset of the method dependencies that DIVER models and its precision improvements over PATHIMPACT are only marginal. Huang and Song [11] extended INFLUENCEDYNAMIC for objected-oriented programs by adding dependencies between fields. Unlike DIVER, however, these approaches model partial data dependencies only and none of them achieve a noticeably-better precision than PI/EAS, as DIVER remarkably does.

6. CONCLUSION

We presented a novel dynamic impact analysis called DIVER. By tracking impacts via method-level execution traces and applying static-dependence knowledge, DIVER attains at acceptable costs a much better precision than existing dynamic impact analyses (e.g., PI/EAS). Our preliminary study on Java software shows that DIVER can prevent almost 70% of false positives from the impact sets produced by PI/EAS, with strong statistical significance.

Acknowledgments

This work was partially supported by ONR Award N0001414100-37 to the University of Notre Dame.

7. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools (2nd Ed.)*. Sept. 2006.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Int'l Conf. on Softw. Engg.*, pages 432–441, 2005.

[4] B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. In *Euro. Conf. on Softw. Maint. and Reengg.*, pages 143–152, 2005.

[5] B. Breech, M. Tegtmeyer, and L. Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Int'l Conf. on Softw. Maint.*, pages 55–65, 2006.

[6] H. Cai, R. Santelices, and T. Xu. Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis. In *Int'l Conf. on Softw. Security and Reliability*, pages 48–57, 2014.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *EMSE*, 10(4):405–435, 2005.

[8] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Systems*, 9(3):319-349, 1987.

[9] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Int'l Conf. on Softw. Engg.*, pages 430–440, 2012.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Prog. Lang. and Systems*, 12(1):26-60, 1990.

[11] L. Huang and Y.-T. Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *Int'l Conf. on Software Engineering Research, Management & Applications*, pages 374–384, 2007.

[12] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. Soot - a Java Bytecode Optimization Framework. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[13] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Int'l Symp. on Software Reliability Engineering*, pages 430–441, 2003.

[14] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Int'l Conf. on Softw. Engg.*, pages 308–318, 2003.

[15] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[16] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero. The hybrid technique for object-oriented software change impact analysis. In *Euro. Conf. on Software Maintenance and Reengineering*, pages 252–255, 2010.

[17] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *European Softw. Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Softw. Eng.*, pages 128–137, 2003.

[18] A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Int'l Conf. on Softw. Eng.*, pages 491–500, 2004.

[19] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. on Softw. Eng.*, 16(9):965–979, 1990.

[20] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Conf. on Obj. Oriented Prog. Syst., Lang., and Appl.*, pages 432–448, 2004.

[21] R. Santelices, Y. Zhang, H. Cai, and S. Jiang. DUA-Forensics: a fine-grained dependence analysis and instrumentation framework based on Soot. In *ACM SIGPLAN Workshop on State Of the Art in Java Program Analysis*, pages 13–18, 2013.

[22] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Trans. Softw. Eng. Methodol.*, 10(2):209–254, 2001.

[23] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang. Change impact analysis based on a taxonomy of change types. In *IEEE Computer Software and Applications Conference*, pages 373–382, 2010.

[24] X. Sun, B. Li, S. Zhang, C. Tao, X. Chen, and W. Wen. Using lattice of class and method dependence for change impact analysis of object oriented programs. In *ACM Symposium on Applied Computing*, pages 1439–1444, 2011.

[25] R. E. Walpole, R. H. Myers, S. L. Myers, and K. E. Ye. *Probability and Statistics for Engineers and Scientists*. Prentice Hall, 2011.