A Framework for Cost-Effective Dependence-Based Dynamic Impact Analysis

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

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

Abstract—Dynamic impact analysis can greatly assist developers with managing software changes by focusing their attention on the effects of potential changes relative to concrete program executions. While dependence-based dynamic impact analysis (DDIA) provides finer-grained results than traceability-based approaches, traditional DDIA techniques often produce imprecise results, incurring excessive costs thus hindering their adoption in many practical situations.

In this paper, we present the design and evaluation of a DDIA framework and its three new instances that offer not only much more precise impact sets but also flexible cost-effectiveness options to meet diverse application needs such as different budgets and levels of detail of results. By exploiting both static dependencies and various dynamic information including method-execution traces, statement coverage, and dynamic points-to data, our techniques achieve that goal at reasonable costs according to our experiment results. Our study also suggests that statement coverage has generally stronger effects on the precision and cost-effectiveness of DDIA than dynamic points-to data.

Keywords—Dependence analysis, dynamic impact analysis, statement coverage, dynamic points-to, cost-effectiveness

I. INTRODUCTION

Analyzing the impacts of constant changes is crucial for successful software evolution [1], even more so as modern software is increasingly complex. Impact-analysis techniques proposed to this date can be classified into two categories: dependence-based and traceability-based [2]. In comparison, dependence-based analysis produces finer-grained impact sets (potentially affected entities), which are generally more useful for understanding code-level changes [3], while for such tasks traceability-based analyses are usually insufficient [4].

Furthermore, among the dependency-based techniques, static approaches compute impact sets for all possible executions that are often highly imprecise due to their conservative nature [5], [6]. In contrast, dynamic impact analysis produces more focused results by using runtime information [7]–[9] that represents specific subsets of all executions. For developers looking for concrete program behavior or understanding the effects of potential code changes relative to those specific subsets, impact sets given by dynamic impact analysis are preferable. Therefore, we target in this paper dependence-based dynamic impact analysis (DDIA).

For clients of impact analysis such as regression testing [10], program comprehension [11], and change understanding [3], imprecise impact sets are clearly undesirable because developers using such results could waste time and other resources on examining entities that are falsely reported as impacted. Moreover, the imprecision can adversely affect the effectiveness of application tasks. For example, when used in program understanding, false-positive impacts may suggest spurious complications that make the program harder to understand than it should be. For another example, applying changes based on imprecise impact sets could lead to severe consequences such as system failure.

Unfortunately, developing a DDIA technique of practical cost and effectiveness, even on the method level, remains a challenging problem. For instance, COVERAGE IMPACT [10] is highly efficient but also very imprecise [9]. In contrast, PATH IMPACT [12] is more precise yet less efficient than COVERAGE IMPACT [9]. Since these two techniques, much research focused on performance optimizations (e.g., [13]) while only a few invested on improving the precision of PATH IMPACT, yet their improvements were marginal only and at much greater costs (e.g., [7]). To the best of our knowledge, the most cost-effective method-level DDIA prior to our work remains to be PATH IMPACT combined with its optimization execute-after sequences (EAS) [13], which we call PI/EAS. The slightly more precise technique is INFLUENCE DYNAMIC [7] but it is much less efficient than PI/EAS.

Nonetheless, existing techniques can still suffer from great imprecision. Our recent study [14] revealed that the mean precision of PI/EAS was about 50% only. Although some other techniques may provide better precision (e.g., [15]), they are descriptive impact analyses [2] applicable only after actual changes are made. However, for many software evolution tasks, impact analysis needs be performed before making those actual changes [1]. In software industry, it has been reported that delayed impact analysis is among the highest-priority issues that both organization and developers encountered [4]. Therefore, we focus on predictive impact analysis [2], which predicts potential impacts of candidate changes.

In this paper, we present three cost-effective DDIA techniques within a unified framework that are much more precise than PI/EAS, called Trace Only (TO), Trace plus statement Coverage (TC), and Full Combination (FC). While all use the dependence graph of the target program [16], they differ in the amount of dynamic data used for the analysis. By leveraging different combinations of static and dynamic program information, including statement coverage and dynamic alias data, these techniques also offer multiple levels of cost-effectiveness tradeoffs of DDIA, and thus provide developers with flexible technical options, which are in growing demand in practice nowadays [3], [4], [17].

We implemented this framework and its three instances above for Java, and evaluated them using seven subjects of up to 220K lines of code in size. Our results show that these new analyses are generally all cost-effective, with continuous and significant precision gains over the baseline approach PI/EAS at reasonable costs. In addition, among the three instances, TC attained the best cost-effectiveness in most cases as regards the overall analysis time overhead, which implies

1The download is publicly available at http://nd.edu/~hcai/diver.
that statement coverage can play a strong role in general DDIA design. The study also suggests that more precise points-to data may not translate to significant gains in the precision or cost-effectiveness of DDIA, akin to previous such findings in the context of program slicing [18]. In comparison, statement coverage mostly leads to greater precision improvements at costs that are better paid off than do dynamic points-to data.

Beyond these three instances, more others can be instantiated from our framework as well. To differentiate those from the three we study here, and also for brevity, we hereafter refer to as IAPRO any of $TO$, $TC$, and $FC$, or them together. Note that IAPRO only prunes dependencies (and consequent impacts) that it ascertains are not exercised by the employed dynamic data, with $TO$, $TC$, and $FC$ incrementally doing so in a conservative manner. Thus, we regard IAPRO as safe with respect to actual impact sets, for which actual changes must be known thus two program versions are needed, it is safe with respect to the execution set utilized. Under this assumption about recall, we express the precision as the impact-set size ratio of IAPRO to the baseline, which is appropriate at least for evaluating relative precision gains as we focus on [9], [13].

The main contributions of this paper include:

- A unified framework that incorporates multiple types of program information to support multiple levels of cost-effectiveness tradeoffs of DDIA (Section IV-A).
- A suite of three new DDIA techniques that instantiate the framework and provide flexible options to meet various cost-effectiveness needs for impact analysis (Section IV).
- An extensive empirical study that demonstrated the cost and effectiveness of the proposed techniques and their superiority over existing options (Section V).
- An analysis of the effects of various dynamic data on the cost-effectiveness of DDIA that can inform future design of more advanced dynamic impact analysis (Section V-C).

II. MOTIVATION

One important application of DDIA is regression testing, for which developers use the results of impact analysis to guide regression test selection and prioritization [10], [19]. For test selection, only test cases that cover at least one impacted entity need be executed; for test prioritization, test cases that cover more impacted entities can be given higher priority. However, imprecise (potentially large) impact sets can lead to unnecessary test cases selected or prioritized, apparently reducing the effectiveness of these techniques in practice.

Since DDIA analyzes the relationships among program components, it is naturally suitable to assist developers with such tasks as program comprehension [11] and change-risk estimation [20]. For example, a new development team member can anchor one particular method and use the impact sets of the method to get a quick picture of the interactions between that method and impacted ones. Also, potential impacts of a set of methods can provide a project manager with necessary information for estimating the risks of changing those methods. However, producing very imprecise and large impact sets can greatly hinder the adoption of impact analysis due to the large cost of careful impact-set inspection, which is required indeed in these usage scenarios.

When studying the accuracy of PI/EAS [14], one of our insights was that the main cause for the imprecision of PI/EAS is its using the execution order of methods only for impact computation, whereas in general not all methods executed after a method $m$ are affected by $m$. We believe that static dependence analysis can help guide using the execution trace to produce more precise results, and that propagating impacts across and through methods is necessary for excluding false positives. INFLUENCEDYNAMIC did utilize such static information yet ignored impact propagation through the insides of methods. We also believe, as previous studies showed, that developers need flexible cost-effectiveness options with impact analysis, for which other types of dynamic information beyond method execution traces could be exploited.

Although our previous study also revealed recall issues with existing DDIA [14], we chose to focus on the precision problem first at our current step. For one thing, while recall is of no less importance than precision in general, for impact analysis highly imprecise impact sets may not be worthy of inspection even if they are safe (of 100% recall). For another, although DDIA may suffer from imperfect recall with respect to actual impact sets, for which actual changes must be known thus two program versions are needed, it is safe with respect to the execution sets utilized for tasks involving a single program version only, such as program understanding and debugging.

III. BACKGROUND

This section presents necessary background with an example program used for illustration purposes. In Figure 1, program $E$ inputs two integers $a$ and $b$ in its entry method $M_0$, manipulates them via $M_1$ and $M_4$ and prints their return values concatenated. $M_2$ updates the static variable $g$ used by $M_4$ later on. $M_3$ and $M_5$, invoked by $M_1$ and $M_2$, include field accesses, conditionals, and arithmetics. In this paper, we addresses predictive impact analysis [12] which assumes no knowledge about actual changes to programs. Such 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 [12], 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 1 shows an example trace of $E$ for PATHIMPACT (bottom left), where $x$ is a method-return event and $x$ the program-exit event. The remaining marks are the entry events of methods. Suppose $M=\{M_2\}$, PATHIMPACT first finds $\{M_5, M_3, M_4\}$ as impacted because they were entered after $M_2$ was entered and then finds $\{M_0, M_1\}$ because these methods returned after $M_2$ was entered. Thus, the resulting impact set is $\{M_0, M_1, M_2, M_3, M_4, M_5\}$ for this trace. For multiple traces, PATHIMPACT takes the union of all per-trace impact sets.

The execute-after sequences (EAS) optimization [13] came later to reduce the space costs of PATHIMPACT without losing precision. This approach exploits the observation that only the first and last occurrence of each method in a trace are needed. The resulting technique, PI/EAS, keeps track at runtime of those two events per method without tracing all method occurrences as PATHIMPACT did.

INFLUENCEDYNAMIC [7] tries to improve the precision of EAS by considering method dependencies in addition to method traces. It models interface-level data dependencies via parameter passing and returns between methods, using an influence graph on which impacts are computed with impact propagation through intraprocedural dependencies ignored. For an example, given the same query set $\{M_2\}$, example program
E, and execution trace as used above, the resulting impact set will be the entire program as PI/EAS produced.

However, INFLUENCE DYNAMIC is marginally (3–4%) more precise only, yet much (10x) more expensive, than PI/EAS [7]. Thus, regardless of its age, PI/EAS remains the most cost-effective technique to compare with ours. Note that while dynamic slicing is even more precise, it would be overly heavyweight for impact analysis at method level.

IV. Technique

To prune false-positive impacts given by PI/EAS or INFLUENCE DYNAMIC, we need more precise information than both the execute-after sequences of PI/EAS and the incomplete dependence analysis of INFLUENCE DYNAMIC. To that end, we propose to utilize the (whole-program) static dependencies, together with the execute-after relation between methods, and, optionally, statement coverage and dynamic alias information, to develop DDIA techniques that offer better precision with various cost-benefit tradeoffs.

A. Overview

We first introduce the design of a unified framework for DDIA, from which both the two existing and three proposed DDIA approaches can be instantiated. This framework, as shown in Figure 2, helps illuminate the rationale underneath various levels of DDIA precision and cost, as well as the relationship among them.

The framework works in three phases: static analysis, runtime, and post-processing, as shown at the top of the diagram which sketches the overall common workflow steps of a typical DDIA. The inputs for the entire process are a program P, an input set (e.g., test suite) T of P, and a set M of queries (i.e., the set of methods for which impact sets are to be queried). In most cases, a mandatory step is to create the dependence graph of P used by all the DDIA techniques we proposed. There are five different workflow paths (numbered with 1 through 5) that correspond to five alternative approaches to computing the impact set of M. The output of the process (framework) is consistently the resulting impact set of M.

The first path directly leads to the impact-computation step without using any execution information, constituting a static impact analysis. Although we do not include it in the study here, it still provides a viable option to developers that we plan to explore in the future. Path 5 effectively leads to PI/EAS, which we use as the baseline approach for aforementioned reasons. The other three paths correspond to the three proposed DDIA techniques: path 2 to TO using method traces only, path 3 to TC with statement coverage added to TO, and path 4 to FC further including dynamic points-to data on top of TC.

When building the dependence graph, the framework first runs a profiler which executes T on P to make decisions on the inclusion of exceptional control dependencies. Since such decisions can greatly affect the graph size hence overall performance of the DDIA, this step is an integral part of the framework. The details about the exception profiler and how the decisions are made can be found in [16].

After profiling, the static-analysis phase computes data dependencies (DD) and control dependencies (CD) to build the dependence graph [16] of P. Next, runtime monitors for collecting constituent dynamic information (method trace, statement coverage, and dynamic points-to data) are inserted through byte-code instrumentation, configured based on the needs of an instance. The outputs of this phase are the dependence graph and instrumentation version P' of P.

During the runtime, the framework executes T on P' to produce the dynamic information configured during static analysis. For example, all the three forms of dynamic data will be produced in FC, while in TO only the method traces will be generated and collected. To reduce the storage at small time overheads well paid off by overall cost savings, dynamically generated data are all compressed on the fly. For programs of relatively long-lasting runs, such extra data processing can be particularly instrumental and often necessary.

The last phase, impact computation is essentially the post-processing of all the static and dynamic information collected during the previous phases. The general idea is to prune executed methods that have no dependencies on the query exercised by the dynamic data utilized. Intuitively, with more dynamic data employed for such pruning, more precise impact set will be obtained and, at the same time, larger costs of the entire analysis will be incurred. When multiple methods are queried, the process computes the impact set for one method at a time and then takes the union of all. Note that for any number of queries with respect to the same execution set, the previous phases are performed only once, with their outputs shared by the impact computation for all queries.

B. Trace Only (TO)

The method execution trace is the single type of dynamic data used in TO (path 2 in Figure 2). Corresponding to

```java
1. public class A {
2.   static int A = 1;
3.   String M1 (int i, int z) {
4.     int x = i + z, y = 2, h = 1;
5.     if (x > y) {
6.       M2 (x, y);
7.     }
8.   }
9.   return s;
10.  }
11.  void M2 (int m, int n) {
12.     A a2 = trans (this);
13.     CM5 (a2);
14.     int w = m - d;
15.    if (w < 0) {
16.      g = m / w;
17.    }
18.    }
19.  }
20.  }
21.  { 22.    }
23.  }
24.  }
25.  }
26.  }
27.  }
28.  }
29.  }
30.  }
31.  public class B {
32.    static short t;
33.    int M3 (int a, int b) {
34.      int j = 0;
35.      t = -4;
36.    }
37.    }
38.    return j;
39.    }
40.  }
41.  }
42.  }
43.  System.out.print(u);
44.  }
45.  }
46.  }
47.  public class C {
48.    static boolean M5 (A q) {
49.      long y = q.d;
50.      boolean b = B.t > y;
51.      q.d = -2;
52.      return b;
53.    }
54.    }
55.  }
56.  }
57.  }
58.  public static void
59.  M0 (String[] args) {
60.    int a = 0, b = 3;
61.    A o = new A();
62.    String s = o.M1(a, b);
63.    double d = B.M4();
64.    String u = s + d;
65.    System.out.print(u);
66.  }
67.  }
68.  }
69.  }
70.  }
71.  Fig. 1: The example program E and its execution traces used by PATHIMPACT and IAPro for illustration.
```
DIVER [16], TO finds methods from the trace directly or transitively dependent on the query using the dependence graph. Details about this technique are presented in [16], and here we recap only the key ideas as follows.

TO consists of two major steps. The first step conceptually corresponds to the entire PI/EAS analysis: using the method trace, filtering methods never executed after the query thus cannot be impacted in that trace. However, an impact set obtained as such is only a rough approximation of methods dynamically dependent on the query. The execution order in the trace implicitly exercised runtime control flows but not control dependencies of the program, with data dependencies entirely ignored. Thus, TO takes the second step to further prune methods that are neither data nor control dependent on the query using the static dependencies, which is crucial for its achieving a better precision than PI/EAS.

When tracking impact propagation along the method trace, TO differentiates three types of data dependencies, parameter, return and heap, so as to apply different propagation rules for precise false-impacts pruning. To be exercised, the first two types need an immediate posterior relation, whereas the last type just a posterior one, between the events of a method already affected and the method being considered. Further, interprocedural dependencies through which impacts may propagate into and out of a method are referred to as incoming and outgoing dependencies of that method, respectively.

C. Trace plus Coverage (TC)

On top of the hybrid approach TO, the TC technique goes further to add statement coverage to the DDIA (path 3 in Figure 2). While the method execution trace informs the analysis with method coverage and execution order, the statement coverage offers a form of finer-grained execution data to enable a finer-grained pruning for the DDIA. With TO, it is implicitly assumed that statements bridging incoming to outgoing dependencies within a method are always executed. In essence, impact propagation through inside each method is based on static, rather than dynamic, dependencies.

However, this conservative assumption can lead to falsely identified impact propagation hence false impacts. TC thus exploits statement coverage to prune those false impacts. There are two alternative ways of such pruning: (1) pre-pruning, which prunes edges on which at least one statement is not covered, from the dependence graph before it is applied to post-processing, and (2) post-pruning, which incorporates the statement coverage data into the impact computation algorithm [16] along with the dependence graph, without pre-processing the graph itself. We expect that both strategies contribute equivalently to the effectiveness (precision), but differently to the overhead, of DDIA.

D. Full Combination (FC)

FC (path 4 in Figure 2) combines all three types of execution data performing the finest-grained analysis out of the five DDIA instances we studied. By employing statement coverage on top of method execution traces, TC supposedly addresses false positives due to spurious transitive dependencies, but only does that partially. Another source of imprecision can be resulted from spurious data dependencies due to imprecise static pointer analysis, which is a well-known problem as true points-to sets are often conservatively approximated only during static analysis. Yet, precise points-to data can be obtained at runtime.

FC thus attempts to further improve the precision of DDIA by exploiting dynamic points-to data. This technique collects the full set of allocation sites that each pointer points to during program execution, by monitoring memory addresses of pointer targets. Then, during the post-processing phase, spurious aliasing-induced data dependencies can be identified by checking the intersection of relevant points-to sets. Given a data dependence \( s_1 \rightarrow s_2 \) in the dependence graph, for example, FC will regard it as a spurious dynamic dependence if there does not exist a variable \( v_1 \) defined on \( s_1 \) and a variable \( v_2 \) used on \( s_2 \) such that the points-to set of \( v_1 \) and that of \( v_2 \) have a non-empty intersection.

In collaboration with the method trace, which contains all method execution instances, dynamic points-to sets can be collected and employed at two granularity levels; method level and method-instance level. The method level strategy maintains a single points-to set of each heap variable (exercised by the runtime input) in a method \( m \) that contains allocation sites pointed to by that variable for all instances of \( m \). In contrast, the method-instance level data includes such points-to sets for each instance of \( m \) separately.
Once spurious dependencies are identified, they are used for pruning false impacts from the method execution trace. The method level data can be applied either before or during impact computation, similar to the pre-pruning and post-pruning strategies described for TC above. However, only post-pruning can be adopted with the method-instance level data as the dependence graph is static and does not incorporate information about method execution instances.

In comparison, the method-level data tends to be conservative thus only approximately precise points-to data that the method-instance level captures, yet with potentially lower time and space overheads than the other. In this paper, we study two variants of FC, FC/ml and FC/ml, which use the method and method-instance level points-to data, respectively.

E. Other Instantiations

Beyond the three techniques proposed above, more instances can also be spawned from this DDIA framework. For example, as marked by path 5, PI/EAS and EAS are both the instance of the framework that does not use the static dependence model (dependence graph) but purely relies on the method-level execution trace to predict impacts of the input queries. INFLUENCE/DYNAMIC can be instantiated from this framework too, which would be a variant of TO that ignores intraprocedural data dependencies and all control dependencies (although it does utilize control flows), and uses a different impact computation algorithm based on the method trace and a partial dependence graph called influence graph [7].

The framework could also be instantiated such that a DDIA technique would use the dynamic points-to sets as the only type of dynamic data, or even without using the dependence graph (i.e., adding dynamic alias analysis to PI/EAS). While even more instances potentially exist, we leave them for future exploration for space reasons.

F. Illustration

We illustrate IAPRO using the example program and traces of Figure 1. In the example trace used by IAPRO (bottom right of the figure), e denotes method-entry events while i denotes method-returned-events. Suppose the query set is \{M5\}, and function _trans is a library call that clones the input object and returns the clone after transforming it. Then, the ground-truth impact set is \{M5\} in this example case.

PI/EAS again finds the entire program (all methods) of E impacted as every method executes after the first entry event of M5. INFLUENCE/DYNAMIC does not prune any method from that imprecise impact set because the influence [7] of M1 propagates to each of other methods according to its influence graph for this case. The impact set of IAPRO starts with \{M5\} upon the occurrence of event M5_e, growing as more methods are found dependent on M5 during the traversal of the method trace with reference to, if available, statement coverage and/or dynamic points-to sets.

Next, control returns into M2. With TO, which assumes that line 15 is covered and object g at line 35 points to the same allocation site (of line 40) as the base object of field d at line 13, the (heap) DD of M2 on M5 (via instance field d) is exercised, so M2 is regarded as impacted. Later in the trace, when M4 is entered, another heap DD, of M4 on M2 (via class field g), is exercised and M4 is thus added to the impact set: The impact reached to (line 13 of) M2 from (line 35 of) M5 continues to propagate (via line 15) to (line 25 of) M4. The first and second heap DD here are the only outgoing dependence of M5 and M2, respectively. As a result, the impact set computed by TO is \{M2, M4, M5\}.

However, with TC, which checks statement coverage and finds that statement 15 is not covered \(w\) is 5 at line 14), the second heap DD above is not exercised. As a result, TC reports a more precise impact set \{M2, M5\}. Finally, on top of TC, FC further checks dynamic points-to data and thus finds that the two base objects, of field d at line 35 and that at line 13, are not aliased at runtime, so the first heap DD is skipped too. Consequently, FC reports M5 as the only impacted method, thus gives the most precise (also accurate) impact set. (Coincidentally, in this example, FC would find the accurate impact set even without using the statement coverage, because by checking dynamic aliasing data the impact would not even propagate into M2 hence M4 would also be pruned.)

V. Evaluation

This section presents our empirical evaluation of the DDIA framework we proposed, by studying the IAPRO techniques as its representative instances: TO, TC, and FC (both variants). Our main goal with this empirical study was twofold. First, we wanted to assess the precision of the new techniques and its practicality in terms of time and space costs against the existing most cost-effective DDIA PI/EAS as the baseline approach. Second, we intended to analyze the effects of different types of dynamic data on the cost-effectiveness of DDIA in order to inform future design of better DDIA techniques.

A. Experiment Setup

The DDIA framework was implemented in Java based on the Soot byte-code analysis framework [21] and our data-flow analysis and instrumentation toolkit DUA-FORENSICS [22]. For the dependence graph construction including the exceptional control dependence analysis, and PI/EAS facilities including the method execution trace instrumentation and monitoring, we reused relevant modules of DIVER [16]. In the implementation of IAPRO and PI/EAS, we included the exception-handling enhancement [14] to ensure the safety of analysis results relative to the execution data utilized.

For statement coverage monitoring capable of exceptional control flows, we computed whole-program reverse dominance frontiers and branch-induced control dependencies [23], whereby statement coverage was monitored indirectly through branch coverage monitoring. For dynamic alias analysis, we piggybacked the collection of object memory addresses on the method-event monitors. Monitoring method and method-instance level points-to sets shares the same instrumentation with differences in runtime monitors.

For impact computation, TO implements the same as DIVER [16]; TC, for which we adopted the post-pruning strategy for a uniform implementation for all IAPRO techniques (recall that FC/ml can not adopt the pre-pruning algorithm), skips any edges of the dependence graph having at least one node whose underlying statement is not covered during the method trace traversal; FC, for both variants, prunes the method trace similarly to TC but checks heap edges only.

We chose seven Java programs of various types and sizes, as summarized in Table I, for our experimental study. The size of each subject is measured as the number of non-comment non-blank lines of code (#LOC) in Java. For each subject, the last two columns give the number of total methods (#Methods)

\footnote{The TO technique was described before but only initially evaluated [16].}
TABLE I: STATISTICS OF EXPERIMENTAL SUBJECTS

<table>
<thead>
<tr>
<th>Subject</th>
<th>#LOC</th>
<th>#Tests</th>
<th>#Methods</th>
<th>#Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule1</td>
<td>290</td>
<td>2,650</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>NanoXML</td>
<td>3,521</td>
<td>214</td>
<td>282</td>
<td>172</td>
</tr>
<tr>
<td>Ant</td>
<td>18,830</td>
<td>112</td>
<td>1,863</td>
<td>607</td>
</tr>
<tr>
<td>XML-security</td>
<td>22,361</td>
<td>92</td>
<td>1,928</td>
<td>632</td>
</tr>
<tr>
<td>JMeter</td>
<td>35,547</td>
<td>79</td>
<td>3,054</td>
<td>712</td>
</tr>
<tr>
<td>Jaba</td>
<td>37,919</td>
<td>70</td>
<td>3,332</td>
<td>1,229</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>102,400</td>
<td>211</td>
<td>8,856</td>
<td>1,098</td>
</tr>
</tbody>
</table>

and that of the subset (#Queries) covered by at least one of the tests (#Tests) used by our dynamic analyses.

Schedule1 is a priority scheduler and NanoXML an XML parser. XML-security is an Apache library for encryption, and JMeter an Apache performance-testing application. Ant is a cross-platform build tool. We took these subjects from the SIR repository [24] and picked the first available version of each. Jaba is a Java-bytecode analyzer provided by its developers. ArgoUML is a UML modeling tool for which we used a stable version r3121 from its SVN repository.

B. Experimental Methodology

For our experiments, we applied TO, TC, FC\textsubscript{ml}, FC\textsubscript{mil}, and PI/EAS separately to each subject on a Linux workstation of an Intel i5-2400 3.10GHz processor and 8GB DDR2 RAM. To obtain the method traces and other dynamic data, we used the entire test suite provided with each subject except for Jaba, for which we use the shortest-running 70 tests only on which the current implementation did not run out of memory. Per technique and subject, we computed the impact sets for all methods, each as an individual query. Expectedly, some methods were not executed by any test and thus had empty impact sets. We excluded them from our results.

To assess the cost-effectiveness of IAPRO, we compared three major metrics against the baseline technique PI/EAS. The first measure is precision. Without actual changes available hence the lack of ground-truth impact sets, this precision is measured relatively through the impact-set size ratios of IAPRO to PI/EAS. We report the mean and five quartiles of the precision for all queries per subject. It is crucial to note that IAPRO attempts to improve the precision yet without penalizing the recall of PI/EAS, because the additional dependence analyses it employs for pruning false impacts of PI/EAS are still all conservative in nature.

The second metric is the time and space costs. For each technique and subject, we collected the time cost (per-thread CPU time) separately per DDIA phase; for static-analysis and runtime phases that are needed once for all queries, we did one measurement of time; for post-processing costs, we calculated the mean and standard deviation of query time for all queries; for space costs, we gauged the sizes of disk data used.

The third metric is the average cost-effectiveness of IAPRO. To see how many gains in effectiveness (precision) different forms of dynamic data contribute with respect to the extra costs incurred by using them, we compared the ratio of precision gain to cost increase among the three IAPRO instances relative to the baseline, for the one-time cost of the first two DDIA phases and per-query post-processing cost separately. This data analysis helps reveal the effects of different dynamic data on the cost-effectiveness of DDIA.

C. Results and Analysis

1) Precision: The main precision results are shown in Figure 3, where the per-subject data points are summarized by separate plots. In each plot, the impact-set size ratios (y axis) to the baseline are grouped by the IAPRO instances we studied, with each group characterized by a single boxplot that consists of the maximum (upper whisker), 75% quartile (top of middle box), 25% quartile (bottom of middle box) and the minimum (lower whisker). The central dot within each middle box indicates the median, surrounded by a pair of triangular marks that represent the comparison interval of that median. These comparison intervals, together within each plot, give a quick indicator of the statistical significance of the differences in medians among the four groups in that plot: For any two groups, their medians are significantly different at the 5% significance level if their intervals do not overlap.

The entire figure suggests that neither statement coverage nor dynamic points-to data contributed to the precision gain of IAPRO over the baseline so much as the static dependence information did. On the other hand, the comparison intervals show no significant differences in the medians of precision among all IAPRO instances, except for those between TO and the other three in the only case of NanoXML.

The complementary results on precision shown in the left five columns of Table II further demonstrated the effects of the additional dynamic data beyond method traces, where the means of impact-set size ratios reveal that both TC and FC gained only little in precision on top of TO. Between the two forms of dynamic information, however, statement coverage appears to be more effective than dynamic points-to data. Consistent with the boxplots, the overall mean ratios, as listed in the bottom row of the table, show that relative advantage of statement coverage. Note that, as in other tables too, these overall numbers are not simple averages over the per-subject means listed in the table but weighted averages by the number of queries per subject as shown in the last column of Table I.
Finally, method-instance level dynamic alias data led to slightly larger precision gain than method-level data, which is most noticeable with Schedule1, NanoXML and Jaba, possibly because of relatively heavier use of pointers in these subjects. Overall, IAPRO reduces the baseline impact sets by 61%–66%, because of relatively heavier use of pointers in these subjects. Naturally, higher costs are associated with subjects of denser dependencies (e.g., Jaba and ArgoUML), longer traces (e.g., Jaba), and/or larger test input (e.g., Schedule1). The lowest efficiency of IAPRO was seen with Jaba, which has a much larger dependence graph and longer traces than other subjects. In addition, the generally large standard deviations suggest the post-processing time fluctuates greatly across different queries in most subjects. The overall numbers show that the query cost of IAPRO is about half a minute per query. In fact, the average cost for the other six subjects than Jaba would be 10–20s. In all, the query cost of IAPRO looks reasonable. Further, multiple queries can be easily parallelized since their computation is independent of each other.

Comparing across the IAPRO instances reveals that the method-instance level alias data is much more expensive than the method level data and statement coverage, which mostly brought small overheads only with respect to TO. This is not surprising because the instance level data can be substantially larger than the other two. The right five columns of Table III compare the sizes of such execution data used by PI/EAS and IAPRO. As can be seen, the largest jump among the IAPRO instances occurred when the method-instance level dynamic points-to data was added to the analysis. On the other hand, as expected also, the space cost constantly grows from TO to FCmil as expected also, the space cost constantly grows from TO to FCmil. This is not surprising because the instance level data can be substantially larger than the other two. The right five columns of Table III compare the sizes of such execution data used by PI/EAS and IAPRO. As can be seen, the largest jump among the IAPRO instances occurred when the method-instance level dynamic points-to data was added to the analysis. On the other hand, as expected also, the space cost constantly grows from TO to FCmil because of the continuous addition of dynamic data. Yet, such costs are all quite small—the largest by FCmil with Jaba is about 30M only. Another source of space cost comes from the dependence graph (not shown in the table), which is no more than 41M (the largest, with ArgoUML), though.

The rest of Table III shows the time costs of the first two phases of DDIA techniques studied here, including the uncaught-exception profiling costs incurred by all IAPRO

---

**Fig. 3: Precision of the IAPRO techniques expressed as impact-set size ratios against PI/EAS (the lower the ratio, the better).**

**TABLE II: AVERAGE PRECISION AND QUERYING COST OF THE IAPRO TECHNIQUES VERSUS PI/EAS**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Mean IS ratio to PI/EAS (%)</th>
<th>Query time in seconds: mean (stdev)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule1</td>
<td>71.3</td>
<td>14.6 (6.0)</td>
</tr>
<tr>
<td>NanoXML</td>
<td>51.7</td>
<td>6.2 (8.8)</td>
</tr>
<tr>
<td>Ant</td>
<td>25.7</td>
<td>3.2 (7.6)</td>
</tr>
<tr>
<td>XML-security</td>
<td>28.8</td>
<td>7.4 (9.0)</td>
</tr>
<tr>
<td>JMeter</td>
<td>18.8</td>
<td>23.3 (7.9)</td>
</tr>
<tr>
<td>Jaba</td>
<td>66.9</td>
<td>9.1 (10.5)</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>31.5</td>
<td>11.1 (14.8)</td>
</tr>
<tr>
<td>Overall</td>
<td>38.3</td>
<td>6.2 (6.0)</td>
</tr>
</tbody>
</table>

**TABLE III: SPACE AND OTHER TIME COSTS OF THE IAPRO TECHNIQUES VERSUS PI/EAS**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Static analysis cost in seconds</th>
<th>Runtime cost in seconds</th>
<th>Execution data size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule1</td>
<td>12</td>
<td>14</td>
<td>116</td>
</tr>
<tr>
<td>NanoXML</td>
<td>12</td>
<td>14</td>
<td>66</td>
</tr>
<tr>
<td>Ant</td>
<td>29</td>
<td>12</td>
<td>56</td>
</tr>
<tr>
<td>XML-security</td>
<td>51</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>JMeter</td>
<td>62</td>
<td>11</td>
<td>51</td>
</tr>
<tr>
<td>Jaba</td>
<td>190</td>
<td>10</td>
<td>38</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>82</td>
<td>10</td>
<td>35</td>
</tr>
<tr>
<td>Overall</td>
<td>82</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>
techniques (Prof.) and the execution time of inputs on the original (uninstrumented) programs (Normal). Mostly, both
the profiling cost and static-analysis time tend to increase
with the program size, with the peak number seen by the
largest subject ArgoUML. Similar to previous observations,
the method-instance level dynamic alias data again led to the
greatest cost growth, in both the static-analysis and runtime
phases. Nonetheless, the profiling is generally cheap, as is
the generation of addition dynamic data at runtime. Static
analysis is mostly efficient too, except for the largest subject
ArgoUML. However, as the runtime phase, the static analysis
incurs one-time costs for all queries for the same program
version, and it can be incorporated in nightly builds in practice.

In sum, the three IAPRO instances come with time and
space costs higher than the baseline which, however, are still
reasonable. Note that such extra time and spaces are, by design,
natural additional costs for the sake of better effectiveness.
Also, incorporating more dynamic information into the DDIA
generally causes increasing cost of both types as expected, with
the method-instance level dynamic alias analysis incurring the
largest overhead among the IAPRO instances.

3) Effects of Dynamic Data: Our statistical testing results
for all the studied DDIA techniques are shown in Table IV.
For each subject, the table presents the p-values from seven
paired two-sided Wilcoxon tests each comparing one pair of
DDIA instances, for which the null hypothesis was constantly
the means of the impact-set sizes for the two compared
techniques being equal. The hypothesis testing covered all
combinations of the IAPRO instances, but only considered
TO when compared to the baseline. We stopped continuing
comparing other IAPRO instances to the baseline as the
significance was found constantly quite strong with TO, and
we saw that those others kept improving in precision over TO.

The numbers indicate that IAPRO is strongly significantly
more precise than PI/EAS, and in the majority of cases
statement coverage contributes significantly to the precision of
DDIA while dynamic points-to data generally does not. And
nevertheless the dynamic alias analysis has stronger effect on
the precision when applied at method-instance level than done
at method level. The bottom row lists the combined p-values
for all subjects per test, which confirmed the same contrasts. In
all, these observations resound with those from the precision
results expressed by impact-set size ratios discussed earlier.

Finally, Figure 4 puts the precision and costs together
showing the cost-effectiveness of the IAPRO instances. In
both plots, the y-axis indicates the precision gains (for a size
ratio of \( r \), the precision gain is \( (1 - r)/r \) \) divided by the
factor of cost increases, of each of these instances, shown
on the x-axis, relative to the baseline results. The per-query
post-processing cost and one-time cost of the first two phases
combined are separately considered, shown in the left and right
plot, respectively. We do not consider the space costs in this
regard as those costs are all marginal, almost negligible relative
to today’s storage resources, and they are all one-time costs.

When considering the query cost only, \( FC_{mil} \) appears to be
the most cost-effective IAPRO instance for any subject. When
considering the static-analysis and runtime costs, however,
\( TO \) has the best cost-effectiveness for all subjects but Ant
and JMeter, for which \( TC \) is the best. \( TC \) is always more
cost-effective than the two variants of \( FC \), though. Taken
together, these results suggest that the most cost-effective
option may vary as certain parts of the overall costs are
weighed more than others, which implies that IAPRO allows
users to choose different best options for varying needs.

For example, if developers readily afford the one-time cost
but are less tolerant for possibly long query time, they would
not bother applying additional dynamic data such as statement
coverage and dynamic points-to sets here. On the other hand,
developers concerned about large static-analysis cost with very
large subjects but willing to wait for impact computation may
find that the effectiveness gain given by statement coverage
and method-level dynamic alias analysis pays off the extra
cost utilizing those additional dynamic data will incur.

Nevertheless, it is also possible that developers would
choose a DDIA technique preferably based on its precision
gain even if it may not be the most cost-effective option: They
opt for better precision anyway no matter whether the added
costs are best paid off. For example, they may choose the most
expensive technique \( FC_{mil} \) for its highest precision among the
five DDIA instances we studied, if the relatively largest cost
is still affordable to them. For those developers, IAPRO does
provide more technical options than existing alternatives.

As we mentioned earlier, the impact sets give by IAPRO are
all safe relative to the execution set utilized by the analysis.
Further, since \( TO \), \( TC \), and \( FC \) prune false-positive impacts
continuously with increasing amount of dynamic data, the
impact set of a query produced by \( TC \) is a subset of that of
the same query by \( TO \), and similar inclusion relations hold for
\( FC \) compared to \( TO \) and \( FC \) to \( TC \). At the same time, such
incremental precision gains come with growing overheads in
general. Accordingly, developers are suggested to adopt the
IAPRO instance that best fits their effectiveness need, time and
storage budget, and availability of program information.

D. Threats to Validity

One internal threat to the validity of our results is the
possibility of implementation errors in the five instances of
our framework and our experimentation scripts. However, all
the DDIA instances were based on Soot and DUA-FORENSICS
that have both matured over many years, and we verified the
scripts manually for each experimental step. We paid special
attention to verifying the modules of our framework used
for monitoring and applying the additional dynamic data. For
threats common to DIVER, we adopted similar solutions [16].

The main external threat is that our study subjects may not
be representative of all real-world software. And an additional
such threat lies in the limited coverage of the inputs available
for the chosen subjects that we used in our dynamic analyses.
To reduce such limitations, we purposely chose as many and
much diverse subjects as possible that come with reasonably
large and complete set of inputs, which were also often used
by other researchers before.

The main construct threat concerns our use of relative
impact-set size comparisons for precision contrasts, and the
assumption about the safety of impact sets given by our
techniques. With respect to actual impact sets for concrete
changes, the recall may not be perfect especially when
those changes modify control flows of programs at runtime.
Nevertheless, our analyses are safe relative to the execution
data utilized for the single program version available to them.

Finally, a conclusion threat is the appropriateness of
our statistical analyses. To reduce this threat, we used a
non-parametric hypothesis test which makes no assumptions
about the distribution of the underlying data (e.g., normality).
TABLE IV: WILCOXON P-VALUES FOR NULL HYPOTHESES THAT THERE ARE NO DIFFERENCES IN MEANS WITH RESPECT TO VARIOUS PAIRED CONTRASTS AMONG PI/EAS AND IAPRO TECHNIQUES (CONFIDENCE LEVEL = 0.95)

<table>
<thead>
<tr>
<th>Subject</th>
<th>PI/EAS vs TO</th>
<th>TO vs TC</th>
<th>TO vs FCmil</th>
<th>FCmil vs FCmil</th>
<th>FCmil vs TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule1</td>
<td>1.73E-05</td>
<td>1</td>
<td>0.4854</td>
<td>0.2261</td>
<td>0.2261</td>
</tr>
<tr>
<td>NanoXML</td>
<td>9.41E-24</td>
<td>0.0053</td>
<td>0.0052</td>
<td>0.0001</td>
<td>0.1298</td>
</tr>
<tr>
<td>Ant</td>
<td>8.06E-130</td>
<td>0.0033</td>
<td>0.0005</td>
<td>0.0014</td>
<td>0.9126</td>
</tr>
<tr>
<td>XML-security</td>
<td>1.22E-104</td>
<td>0.1029</td>
<td>0.0307</td>
<td>0.0196</td>
<td>0.733</td>
</tr>
<tr>
<td>JMeter</td>
<td>2.71E-181</td>
<td>0.6369</td>
<td>0.5411</td>
<td>0.1745</td>
<td>0.8892</td>
</tr>
<tr>
<td>Japa</td>
<td>6.43E-42</td>
<td>0.0384</td>
<td>0.0125</td>
<td>0.0001</td>
<td>0.5083</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>1.77E-196</td>
<td>0.1575</td>
<td>0.0995</td>
<td>0.0006</td>
<td>0.4086</td>
</tr>
<tr>
<td>Fisher Overall</td>
<td>0</td>
<td>0.0006</td>
<td>4.99E-05</td>
<td>2.79E-09</td>
<td>0.9935</td>
</tr>
</tbody>
</table>

Fig. 4: Cost-effectiveness of the IAPRO techniques expressed as the ratios of their precision gain to the increase in the average query cost (left) and total cost of the first two phases (right), both against PI/EAS.

Another conclusion threat concerns the data points analyzed: We applied the statistical analyses only to methods for which impact sets could be queried (i.e., methods executed at least once). To minimize this threat, we adopted this strategy consistently for all experiments and compared the techniques of interest with respect to those methods only.

VI. RELATED WORK

In [16], we presented the technique of DIVER, which corresponds to the TO instance in this paper. Also, our DDIA framework reused many parts of DIVER to unify all the five DDIA instances we studied here. In contrast, this paper focuses on the unified DDIA framework by which we provide DDIA techniques of multiple level of cost-effectiveness tradeoffs, and studies the effects of two additional types of dynamic data on that tradeoff. In addition, our empirical study includes a much extended evaluation of DIVER and PI/EAS plus three others.

The baseline PI/EAS comes from the method-level DDIA introduced by Law and Rothermel [12] and its performance optimization EAS [13] by Apiwattanapong and colleagues. Our IAPRO techniques all utilize the method execution events as by PI/EAS as the common form of dynamic data to prune interprocedural dependencies that are not exercised by runtime inputs. Unlike PI/EAS that relied solely on the method execution order, however, IAPRO leverages static program dependencies in addition to that data and other types of dynamic information to significantly improve precision against PI/EAS. Impact analysis based on static-execute-after relations [27] also exploits method execution order but that for a static approach (considering all possible program inputs).

DDIA approaches employing both static and dynamic information (i.e., hybrid) have also been explored before, such as INFLUENCE/DYNAMIC [7] and its extension in [28]. While these techniques improve PI/EAS in terms of analysis precision, they all model partial dependencies yet exploit a single type of dynamic data (the method execution trace) only. Previous studies have shown that none of them achieved significant precision gain over PI/EAS [7], [28]. SD-IMPALA [29] also explored hybrid DDIA yet it focuses on improving the recall of DDIA and does that at the cost of penalizing precision. It uses call graphs as the static dependence model and is shown even less precise than its predecessor Impala [30] and PI/EAS. In contrast, IAPRO is built on a complete (albeit conservative) dependence model to accomplish significantly better precision not only than PI/EAS but even further beyond DIVER we recently developed by using diverse dynamic information. It is difficult to include INFLUENCE/DYNAMIC in our study as its design is constrained to procedural languages such as C while our current IAPRO implementation targets object-oriented software, and also its environment is different from that of IAPRO.

Statement coverage has been used in regression testing [31], test generation [32], and in general as a test metric [33], but not yet directly for predictive DDIA or even impact analysis generally, to the best of our knowledge. For DDIA, Orso and colleagues used coverage data but at method level to guide impact computation [10], which is much less precise than PI/EAS [9], though. We used statement coverage in TC and FC to improve the precision of predictive DDIA and showed that using statement coverage can greatly contribute to the precision and cost-effectiveness of DDIA.

Mock and colleagues used dynamic points-to data to improve the precision of program slicing and intensively studied the effects of that data on slice sizes [18]. In their study, they examined flow-sensitive and flow-insensitive dynamic points-to sets for both variables and function pointers, and found that dynamic pointer analysis does not significantly lead to better slicing precision in general. With the two FC instances of IAPRO, we exploited dynamic points-to data too but for DDIA. We examined two types of such data also but differentiated them by method instance instead of by pointer dereference site, based on our different application contexts and needs. While our finding that dynamic points-to data generally may not translate to significantly higher precision for DDIA is akin to theirs, our study suggests higher overhead of using dynamic points-to data relative to the total cost of DDIA, at the method-instance level in particular, than what they found in the context of program slicing.

Forward dynamic slicing [34] could work as the finest-grained DDIA, yet in theory it can be too expensive.
for a method-level impact analysis [13], [35] since it would have to be applied to most, if not all, statements inside the queried method. On the other hand, dynamic slicing can be an extended instance of our DDIA framework as it implicitly exploited statement coverage and (statement-instance level) dynamic points-to data too, and beyond. While we assume that dynamic slicing would incur much higher cost than IAPRO, it may still be worth comparing them empirically, especially on their cost-effectiveness, for DDIA. In that regard, variants of dynamic slicing such as relevant slicing [36] and quasi slicing [37] will be of interest as well as they have different levels of efficiency, precision, and/or recall.

VII. CONCLUSION

We have presented in this paper a DDIA framework and its three representative instances together called IAPRO, which exploit both static program dependencies and various dynamic data, including method execution trace, statement coverage, and dynamic points-to data, to make dynamic impact analysis more useful. Beyond separately evaluating the precision and costs of each IAPRO instance against the baseline technique PI/EAS, we also studied the effects of these different types of dynamic information on the cost-effectiveness of DDIA.

Our study demonstrated that IAPRO is constantly much (160%–200%) more precise than the baseline approach with strong statistical significance, and that leveraging statement coverage and dynamic points-to data can further improve the precision of DDIA, all at reasonable costs. In addition, we showed that since different IAPRO instances give the best cost-effectiveness in different situations or application contexts, our DDIA framework and techniques can provide flexible options to meet various user needs for impact analysis.

We are currently expanding the study of DDIA to include both larger number and diversity of subjects and more types of changes (e.g., multiple-method changes and real changes from software repositories). We are also planning to expand the result analysis to further examine individual effects of various static and dynamic data on DDIA in general, and the interactions among them. Our another next step is to focus on the recall of predictive DDIA with respect to true impact sets.

VIII. ACKNOWLEDGMENTS

This work was partially supported by ONR Award N0001414140037 to the University of Notre Dame.

REFERENCES