DIAPRO: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness

HAIPENG CAI, University of Notre Dame
RAUL SANTELICES, Dell SecureWorks
DOUGLAS THAIN, University of Notre Dame

Impact analysis not only assists developers with change planning and management, but also serves a range of other client analyses such as testing and debugging. In particular, for developers working in the context of specific program executions, *dynamic* impact analysis is usually more desirable than static approaches as it produces more manageable and relevant results with respect to those concrete executions. However, existing techniques for this analysis mostly lie on two extremes, either fast but too imprecise or more precise yet overly expensive, while in practice both more cost-effective techniques and variable cost-effectiveness tradeoffs are in demand to fit a variety of usage scenarios and budgets of impact analysis.

This paper aims to fill the gap between these two extremes with an array of cost-effective analyses and, more broadly, to explore the cost and effectiveness dimensions in the *design space* of impact analysis. We present the development and evaluation of DIAPRO, a framework that unifies a series of impact analyses including *three new hybrid techniques* which combine static and dynamic analyses. Harnessing both static dependencies and multiple forms of dynamic data including method-execution events, statement coverage, and dynamic points-to sets, DIAPRO prunes false-positive impacts with varying strength for variant effectiveness and overheads. The framework also facilitates an in-depth examination of the effects of various program information on the cost-effectiveness of impact analysis.

We applied DIAPRO to ten Java applications in diverse scales and domains, and evaluated it thoroughly on both arbitrary and repository-based queries from those applications. We show that the three new analyses are all significantly more effective than existing alternatives while remaining efficient, and the DIAPRO framework as a whole provides flexible cost-effectiveness choices for impact analysis with the best options for variable needs and budgets. Our study results also suggest that hybrid techniques tend to be much more cost-effective than purely dynamic approaches in general, and that statement coverage has mostly stronger effects than dynamic points-to sets on the cost-effectiveness of dynamic impact analysis while static dependencies have even stronger effects than both forms of dynamic data.

CCS Concepts: •**Software and its engineering** → **Software reverse engineering; Software evolution;**

Additional Key Words and Phrases: Impact analysis, dependence analysis, coverage, points-to, cost effectiveness

ACM Reference Format:
Haipeng Cai, Raul Santelices, and Douglas Thain, 2015. DIAPRO: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness. *ACM Trans. Softw. Eng. Methodol.* V, N, Article 0000 (2015), 45 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Constant code changes drive modern software systems to evolve so as to accommodate volatile requirements, yet can risk their quality and reliability as well [Rajlich 2014]. To reduce and prevent such risks, it is crucial to perform impact analysis [Bohner and Arnold 1996; Li et al. 2013b] as an integral step of software evolution through which the developers assess potential change effects so that right decisions can be made regarding whether and where changes ought to be applied. Impact-analysis techniques developed to this date can be broadly classified into two categories: dependence-based and traceability-based [Bohner and Arnold 1996]. In comparison, dependence-based analysis produces code-based *impact sets* (the set of potentially impacted

This work is supported by the Office of Naval Research under grant N000141410037.
Authors' addresses: Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556; emails: hcai@nd.edu, rasantel@gmail.com, dthain@nd.edu.

DOI: http://dx.doi.org/10.1145/0000000.0000000

entities) which are generally more useful for understanding code changes [Tao et al. 2012], while for such tasks traceability-based analyses are usually insufficient [Rovegard et al. 2008].

For the dependency-based impact analysis, static approaches compute impact sets for all possible executions thus are often highly imprecise due to their overly conservative nature [Horwitz et al. 1990; Li and Offutt 1996]. For example, such analysis has to assume that any path of the program could be taken at runtime, leading to the excessively large results; other source of imprecision include over-approximations during call-graph construction and points-to analysis [Aho et al. 2006; Horwitz et al. 1990; Ryder 2003]. In contrast, dynamic impact analysis produces more focused results by using runtime information [Breech et al. 2006; Li et al. 2013b; Orso et al. 2004] that represents specific subsets of all executions. For developers looking for concrete program behavior or potential effects of candidate code changes relative to particular executions, impact sets given by dynamic impact analysis are apparently more preferable. Therefore, we aim at *dependence-based dynamic impact analysis* (DDIA). Further, as dynamic impact analysis is commonly adopted at method level [Orso et al. 2003; Orso et al. 2004; Apiwattanapong et al. 2005; Breech et al. 2006; Cai and Santelices 2014], this paper too addresses method-level DDIA.

As with many other techniques, impact analysis needs to be cost-effective in order to be practically useful. More specifically, a DDIA is regarded as cost-effective if it produces effective impact sets in relation to the cost incurred by performing the analysis itself and inspecting its results. In this work, we measure the effectiveness of DDIA through precision assuming perfect recall for a dynamic analysis. Further, we gauge cost-effectiveness, the ratio of effectiveness to cost, as a relative measure—for example, if an analysis A generally has a higher such ratio than a baseline B, it is cost-effective relative to B. Despite a large body of previous research, however, developing a DDIA technique of practical cost and effectiveness remains a challenging problem. For instance, COVERAGEIMPACT [Orso et al. 2003] is highly efficient but also very imprecise [Orso et al. 2004]. In comparison, PATHIMPACT [Law and Rothermel 2003b] is more precise but significantly less efficient [Orso et al. 2004]. Following these two techniques, many efforts aimed to optimize the efficiency (e.g.,[Apiwattanapong et al. 2005]); only a few focused on improving the precision, yet the improvements were marginal only and achieved at much greater costs (e.g., [Breech et al. 2006]). To the best of our knowledge, the most cost-effective DDIA prior to our work remains to be PATHIMPACT combined with its optimization *execute-after sequences* (EAS) [Apiwattanapong et al. 2005], which we call PI/EAS. Yet, even PI/EAS attains an average-case precision of 50% only according to our extensive studies [Cai et al. 2014; Cai and Santelices 2015b].

Although some other techniques may provide better precision (e.g., [Ren et al. 2004]), they are *descriptive* impact analyses [Bohner and Arnold 1996] applicable only *after* actual changes are made. However, for many software evolution tasks, impact analysis has to be performed *before* making those changes [Rajlich 2014]. In software industry, it has been reported that delayed impact analysis is among the highest-priority issues that both organizations and individual developers encountered [Rovegard et al. 2008]. Also, when working in the prediction setting, DDIA addresses in general the *influence* of a program entity, independent of any specific changes to be made there, on the rest of the program. This *predictive* nature enables a wider range of applications of DDIA than change planning and management: In essence, the prediction computes or reasons about the interaction (via dependencies) among different program entities. Thus, this paper targets predictive impact analysis (or *impact prediction* for short) in the scope of DDIA.

To address the great imprecision of predictive DDIA at acceptable costs, we recently developed DIVER [Cai and Santelices 2014], a technique that provides significantly higher precision and cost-effectiveness than PI/EAS with a hybrid approach which combines static and dynamic analyses. In its dynamic analysis, though, DIVER only utilizes one type of, coarse (method-level) dynamic information: method-execution events. As a result, the technique has to assume that any impacts that reached the entry of a method propagate beyond the method once it is executed as long as there is a path inside the procedure dependence graph (PDG) [Ferrante et al. 1987] of the method for that propagation. Apparently, this assumption implies excessive conservativeness, signaling considerable room for further improvement in effectiveness. Moreover, several industrial

studies have suggested that developers would prefer multiple choices of techniques in terms of cost-effectiveness tradeoffs to best fit their needs for different task scenarios and budgets of impact analysis [de Souza and Redmiles 2008; Rovegard et al. 2008; Tao et al. 2012]. Our experience with DIVER shows that combining static and dynamic program information can help develop more cost-effective DDIA techniques over those purely based on dynamic information. Yet, a natural another direction has not been probed—harnessing diverse forms of dynamic data—by which we may offer not only further improved precision and cost-effectiveness beyond DIVER but also, more importantly, flexible cost-effectiveness options to developers to meet their aforementioned needs.

In this context, we develop a unified framework DIAPRO of cost-effectiveness DDIA techniques to explore such potentials. Through DIAPRO, we aim at an integrated and versatile scheme for dynamic impact analysis by utilizing both static program dependencies and three different forms of dynamic data: method-execution events (*trace*), statement coverage (*coverage*), and dynamic points-to sets (*aliasing*). By combining the static and one or more forms of these dynamic program information, DIAPRO unifies PI/EAS, DIVER, and *three new* DDIA techniques, named by the dynamic data included as the static dependencies are always utilized: *TC* using trace and coverage, *TD* using trace and aliasing, and *TCD* using all these three types of dynamic data. Intuitively, by using different amount of program information, these DIAPRO instances offer different levels of effectiveness (precision) at various amount of overheads thus collectively provide developers with flexible technical options for impact analysis. Additionally, this framework facilitates an in-depth examination on the effects of various program information on the cost and effectiveness of DDIA, understanding which is instrumental to designing more advanced future DDIA techniques.

We have implemented DIAPRO with the three new instances for Java and successfully applied them to ten Java applications in various scales and domains.[1] We gauge the performance (cost, effectiveness, and average cost-effectiveness) of DIAPRO via that of its instances on both arbitrary impact-set queries that cover the entire program of each subject and queries based on real changes developers committed to active code repositories. We also compare among all the studied DIAPRO instances to investigate the effects of static and dynamic data on the overall performance of DDIA. Our results show that the new DDIA techniques are all practically cost-effective, achieving continuous and significant precision gains over PI/EAS and DIVER at mostly increasing yet constantly reasonable costs. In addition, among the three new DIAPRO instances, *TC* attained the best cost-effectiveness in most cases with respect to the overall analysis time overhead, which implies that statement coverage can play an important 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 [Mock et al. 2005]. In comparison, statement coverage mostly leads to greater effectiveness improvements at costs that are better paid off than do dynamic points-to sets, although the effects of neither dynamic data appeared stronger than static dependencies, reaffirming the merits of hybrid techniques for DDIA relative to purely dynamic approaches like PI/EAS.

The main contributions of this work include:

— *A unified framework* DIAPRO that incorporates multiple types of program information to offer variable levels of cost-effectiveness tradeoffs for DDIA to meet various needs and budgets for dynamic impact prediction.
— *Three new DDIA techniques* that instantiate the framework to provide better precision and cost-effectiveness than existing alternatives at practically acceptable costs.
— *An open-source unified implementation* of DIAPRO, including two representative existing DDIA techniques and the three new instances, that has been successfully applied to ten Java programs in different sizes and application domains.
— *An extensive empirical evaluation* that demonstrates the superior effectiveness of the proposed techniques over existing options and the ability of DIAPRO to provide variable cost-effectiveness.

[1] The download of source code, executable, and usage demo is publicly available at https://chapering.github.io/diver.

— *An in-depth examination* of the effects of static dependencies and various dynamic data on the cost-effectiveness of DDIA that informs about future design considerations for more advanced dynamic impact prediction and other dependence-based analyses.

2. MOTIVATION

During software evolution and maintenance, change planning and management has been a common application of DDIA [Li et al. 2013b]. However, applying changes based on imprecise impact sets could lead to severe consequences such as system failure. For clients of DDIA which use its results indirectly, imprecise impact sets are clearly undesirable too because such results would not only result in waste of time and other resources (for inspecting false impacts) but can also be misleading to developers. For example, since it analyzes the relationships among program components, DDIA is naturally suitable to assist developers with such tasks as program comprehension [Buckner et al. 2005] and change-risk estimation [Tao et al. 2012; Ajrnal Chaumun et al. 1999]: 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. Another important application of DDIA is regression testing, for which developers use the results of impact analysis to guide regression test selection and prioritization [Orso et al. 2003; Schrettner et al. 2013]. 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 client techniques in practice. In all, the imprecision of DDIA would not only impede the task of impact analysis itself, but also further adversely affect the effectiveness of its client analyses.

From studying the accuracy of PI/EAS [Cai et al. 2014; Cai and Santelices 2015b], one of our insights was that the main cause for the imprecision of PI/EAS consists in its overly-conservative impact inference based on the method execution order only, while in general execution ordering does not necessarily imply dependence relations. On the other hand, our experience with DIVER [Cai and Santelices 2014] showed that static dependencies can guide the execution-order-based impact inference to produce more precise results, and that propagating impacts both *across* (i.e., via interprocedural dependencies) and *through* (i.e., via intraprocedural dependencies) methods is necessary for that improvement. In fact, such hybrid approaches have been attempted earlier [Breech et al. 2006; Huang and Song 2007; Maia et al. 2010], yet none of them achieved significant precision or cost-effectiveness improvement over PI/EAS, for which two possible reasons are their considering partial dependencies only and ineffective utilization of both (static and dynamic) information together. For instance, INFLUENCEDYNAMIC [Breech et al. 2006] did utilize static dependencies yet ignored impact propagation through intraprocedural dependencies.

Unfortunately, focusing on better effectiveness only would not solve all the main challenges to impact analysis in its practical application: as previous studies showed, developers also need flexible options in terms of variant cost-effectiveness tradeoffs when performing the analysis. For example, for developers to understand the effects of code changes, impact analysis needs to play a variety of roles, from developing new features to fixing bugs [Tao et al. 2012]. Also, developers have to perform different kinds of impact analysis to deal with various types of change requests [Rovegard et al. 2008] and need different techniques to fit different usage scenarios of impact analysis [de Souza and Redmiles 2008]. Moreover, developers are usually constrained by their resources (e.g., task schedule and computation time) allocated for impact analysis [Rovegard et al. 2008; Acharya and Robinson 2011], in large part due to the increasingly shortening software evolution cycle [Rajlich 2014]. For example, if aiming to get a quick high-level picture on the interactions among constituent members of a module in order to understand the new architecture of that module, developers would prefer smaller cost over higher precision when choosing an

impact-analysis technique to use; when facing a highly complex software system where a change is being proposed in a few critical methods on which a large set of other methods are dependent, developers would focus much more on the precision if their resources are not tightly restricted [Cai and Santelices 2015b; Tao et al. 2012].

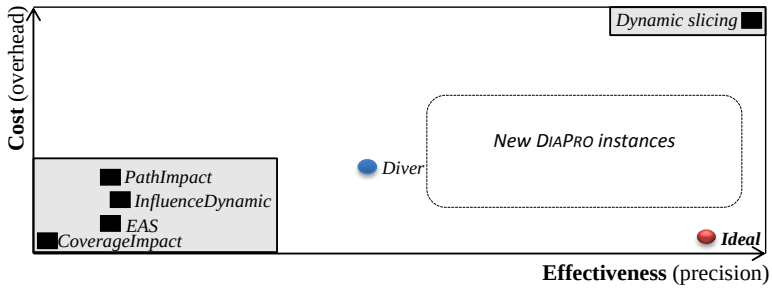

Fig. 1: A schematic illustration of tradeoffs between the cost and effectiveness dimensions in the DDIA design space, where the cost indicates all overheads a DDIA technique incurs and the effectiveness is mainly measured by precision assuming that all techniques are sound modulo the executions utilized. Sample techniques are placed in the two-dimension space only to approximate relative contrasts among these samples for illustration purpose rather than to exactly situate them based on their actual cost-effectiveness metrics. In this diagram, the closer to the bottom-right (*Ideal*) the better (more cost-effective). As is shown, existing DDIA techniques mostly lie at the two extremes for low cost-effectiveness: either overly low effectiveness (bottom left) or prohibitively high cost (top right). This work aims at the untapped band (*New* DIAPRO *instances*).

For exploring techniques that meet the multiple cost-effectiveness needs, consider a two-dimensional design space (i.e., combination of multiple variables each characterizing one of the dimensions) of DDIA, where cost and effectiveness are the two dimensions. We depict this design space in Figure 1 and roughly place sample techniques for an approximate illustration: the coordinate of each DDIA technique does not exactly reflect its exact measure of cost and effectiveness but simply show its relative comparison to other instances. At one extreme (bottom left), techniques like PATHIMPACT, EAS, COVERAGEIMPACT, and INFLUENCEDYNAMIC provide rapid but very rough (imprecise) results; at the other extreme (top right), finest-grained solutions like forward dynamic slicing are able to give the highest precision at the cost of scalability [Law and Rothermel 2003a]. While these two extremes may indeed offer useful choices to developers, we believe that more and better options of cost-effectiveness tradeoffs would be more attractive and viable, and thus should be further probed. The blank region (outside the two extremes) illustrates this untapped band, where DIVER has been the first successful attempt. Yet, it would be rewarding to exploit more program information so as both to fill the remaining gap and to draw closer to the ideal cost-effectiveness (bottom right). Towards that step, we set out to investigate two additional forms of dynamic data beyond method-execution events: statement coverage and dynamic aliasing data for three new instances of DDIA (the region with dotted boundary). Next, we present the DIAPRO framework to show how this motivation can be fulfilled to meet the developers' needs for multiple levels of DDIA cost-effectiveness tradeoffs, in addition to more cost-effective dynamic impact prediction techniques, beyond the two extremes and DIVER.

3. BACKGROUND

This section presents necessary background with an example program used for illustration purposes. In Figure 2, program E inputs two integers `a` and `b` in its entry method `M0`, manipulates them via `M1` and `M4` and prints their return values concatenated. `M2` updates the static variable `g` used by `M4` later on. `M3` and `M5`, invoked by `M1` and `M2`, include field accesses, conditionals, and arithmetics. In this paper, we addresses *predictive* DDIA [Law and Rothermel 2003b] which assumes no knowledge

```
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) M2(x, y);
6        int r = new B().M3(h, g);
7        return '' + r;
8     }
9     void M2(int m, int n) {
10       A a2 = transfrom(this);
11       C.M5(a2);
12       int w = m - d;
13       if (w < 0)
14          g = m / w;
15    }
16 }
17 public class B {
18    static short t;
19    int M3(int a, int b) {
20       int j = 0; t = - 4;
21       if ( a < b )
22          j = b - a;
23       return j;
24    }
25 }
```

```
26 public class C {
27    static double M4() {
28       int x = A.g, i = 5;
29       try {
30          i = x / (i + t);
31          new A().M1(i, t);
32       } catch(Exception e) {
33          e.printstack();
34       }
35       return x;
36    }
37    static boolean M5(A q) {
38       long y = q.d;
39       boolean b = B.t > y;
40       q.d = -2;
41       return b;
42    }
43    public static void M0(String[] args) {
44       int a = 0, b = 3;
45       A o = new A();
46       String s = o.M1(a, b);
47       double d = C.M4();
48       System.out.print ( s + d );
49    }
50 }
```

PATHIMPACT: M0 M1 M2 M5 r r M3 r r M4 r r x

DIAPRO: $M0_e$ $M1_e$ $M2_e$ $M5_e$ $M2_i$ $M1_i$ $M3_e$ $M1_i$ $M0_i$ $M4_e$ $M4_i$ $M0_i$ x

Fig. 2: The example program E and its execution traces used by PATHIMPACT and DIAPRO.

about actual changes to programs. Such analysis takes a program P, an input set (e.g., test suite) T, and a set M of *queries* (i.e., methods for which impacts are queried) and outputs an impact set containing the methods in P potentially impacted by M when running T.[2]

3.1. PI/EAS

One representative DDIA technique is PATHIMPACT [Law and Rothermel 2003b], which collects runtime traces of executed methods. For each method q in M that is queried for its impacts, the analysis uses the method execution order found in the runtime traces of P for T to compute the impact set: It identifies as impacted the query q itself and any other methods executed after q in any of the traces. Figure 2 shows an example trace of E for PATHIMPACT (at the bottom), where `r` is a method-return event and `x` the program-exit event. The remaining marks are the entry events of methods. Suppose M=$\{$`M2`$\}$, PATHIMPACT first finds $\{$`M5`, `M3`, `M4`$\}$ as impacted because they were entered after `M2` was entered and then finds $\{$`M0`, `M1`$\}$ because these methods returned after `M2` was entered (the method associated with `r` is figured out by backwardly matching the first unmatched entry event). Thus, the resulting impact set is $\{$`M0`, `M1`, `M2`, `M3`, `M4`, `M5`$\}$ for this trace. For multiple traces, PATHIMPACT takes the union of all per-trace impact sets. Later, the *execute-after sequences* (EAS) optimization [Apiwattanapong et al. 2005] reduces the time and space costs of PATHIMPACT without losing precision. This approach exploits the observation that only the first and last occurrences 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. It has been shown that, for discovering the execute-after relations only, the per-method first and last events give the same information as the full tracing would do.

INFLUENCEDYNAMIC [Breech et al. 2006] tries to improve the precision of PI/EAS by considering static dependencies at method level in addition to method-execution event traces. It

[2]The impact set initially includes the queries themselves, and we use *query (set)* as identical terms to *initial impact (set)*.

models interface-level data dependencies via the passing of parameters and return values among methods, using a representation called *influence graph* on which impacts are computed with impact propagation through intraprocedural dependencies ignored—any impacts that reached the entry of an executed method propagate out of the method even if there is no any path inside the PDG of the method to enable that propagation. As an example, given the same query set M={M2}, example program E, and execution trace as used above, the resulting impact set will be the entire program as PI/EAS produced. Overall, INFLUENCEDYNAMIC is marginally (3–4%) more precise only, yet much (10x) more expensive, than PI/EAS [Breech et al. 2006]. In consequence, PI/EAS remained the most cost-effective technique prior to DIVER. Note that while dynamic slicing is even more precise, it would be overly heavyweight for impact analysis at method level. An extensive discussion on the advantages of a method-level DDIA such as PATHIMPACT over static and dynamic program slicing can be found in [Law and Rothermel 2003b].

3.2. DIVER

The latest technique for predictive DDIA in the literature, DIVER [Cai and Santelices 2014] achieves significantly higher precision and cost-effectiveness than the most cost-effective alternative PI/EAS prior to it. To avoid unnecessary static-analysis overhead, the technique first finds out which methods throw what types of exceptions that are not handled with respect to the input set T by running an exception-profiler on P for T. Next, it instruments P for method event monitoring and creates a whole-program static dependence graph of P that approximates the system dependence graph (SDG) [Horwitz et al. 1990] with control dependencies due to unhandled exceptions skipped, by discarding context-sensitivity during the computation of transitive dependencies. It then traces all occurrences of entry and returned-into events of each method [Apiwattanapong et al. 2005] by running the instrumented version of P on T. Finally, given a query m, DIVER utilizes the approximate dependence graph to prune methods executed after m in the method-event traces but not dynamically dependent on m according to the static dependence information. At the core of DIVER is its dependence-based trace pruning for impact computation, for which the detailed algorithm is presented and explained in [Cai and Santelices 2014]. Here we introduce key concepts and summarize the main steps of the algorithm that we reuse for building the DIAPRO framework.

In the DIVER dependence graph, intraprocedural dependencies are those directly from per-method PDGs; interprocedural dependencies connect among all PDGs of P, and are classified into two major classes—we use the terms *edge* and *dependence* interchangeably in the context of a dependence graph consisting of data dependencies (DD) and control dependencies (CD):

— *Adjacent edge*, including two types of interprocedural DDs: *parameter* DDs each connecting a call site to its corresponding callee for an actual-formal parameter link, and *return* DDs each connecting a return site to its corresponding caller site for the passing of a return value. Such an edge propagates impacts from its source to its target node only if the enclosing method of the source (e.g., a caller) is *adjacent* to that of the target (e.g., a callee of that caller) in the trace.
— *Posterior edge*, including all interprocedural CDs and the third type interprocedural DDs, *heap* DDs each connecting a definition-use pair of a dynamically-allocated (heap) variable. Such an edge propagates impacts from its source to its target as long as the enclosing method of the target (e.g., a method that reads from a class field) appears *after* (no matter how far apart from) that of the source (e.g., a method that writes to that class field) in the trace.

Then, starting from the query itself as the first method marked as impacted, DIVER determines that the original impact propagates from an already-impacted method a to a candidate method b, then marks b as impacted as well, if one of the following two conditions is satisfied:

— b *immediately* follows a in any trace and the dependence graph has an adjacent edge from a to b.
— b occurred after a in any trace and the dependence graph has a posterior edge from a to b.

As such, DIVER finds all impacts iteratively when traversing a trace (from each input in T), and then takes the union of per-trace impact sets for all traces as the final impact set for T. We give an illustration of DIVER as part of that for the entire DIAPRO framework in Section 4.1.2.

4. APPROACH

It has been shown that more precise information than the method execution order of PI/EAS and the incomplete dependence analysis of INFLUENCEDYNAMIC can significantly raise the precision and cost-effectiveness of DDIA over these previous techniques [Cai and Santelices 2014]. We thus propose to utilize an even larger variety of program information, combining the whole-program static dependencies, method-execution traces, and, optionally, statement coverage and dynamic points-to sets, to develop new DDIA techniques that further improve the precision and cost-effectiveness. The employment of increasing amounts of data is also expected to bring growing costs. Nevertheless, the new techniques will ultimately provide more, variable cost-effective options for DDIA should the added overheads be well paid off by the effectiveness gains in return.

4.1. Overview

We first introduce the design of our unified framework for DDIA, DIAPRO, from which both the representative existing (PI/EAS and DIVER) and three proposed DDIA approaches can be instantiated. This framework, as shown in Figure 3, helps illuminate the rationale underneath various levels of DDIA precision and cost, as well as the relationship among them. As marked in the figure, there are six different process paths (numbered with 1 through 6) that correspond to in total six impact-analysis techniques that have been unified in the framework thus far: the two existing approaches and the three new DIAPRO instances, plus the static approach as a special case here (which is not a DDIA technique). Of the five DDIA techniques, four are hybrid approaches where a *mandatory* step is to create the dependence graph of P that contains the static dependencies utilized by these analyses. The inputs for the entire framework are a program P, an input set (e.g., test suite) T of P, and a set M of queries, and the output is consistently the impact set of M with respect to T, just as those of a common predictive DDIA technique (see Section 3). As such, despite widely using static analysis and benefiting immensely from static dependencies as shown by our empirical studies later on, our framework serves *dynamic* impact analysis which addresses potential impacts relative to *specific program executions*, although it unifies both hybrid and purely-dynamic approaches to that analysis.

Path 1 directly leads DIAPRO to the impact-computation step without using any execution information, constituting a static impact analysis. Although we do not include it in the study of this work, it still provides a viable option to developers that can be employed in special scenarios, such as when concrete program inputs (executions) are not available. The other five paths represent the five DDIA techniques we address in this paper, each annotated with the types of program information (abbreviated per the acronym index on the right) utilized by the corresponding instance and the instance name (in the parentheses). Path 2 indicates the process flow for the DIVER instance, and path 3 effectively leads to PI/EAS, which skips the first step for the dependence graph construction common to all other DIAPRO instances for DDIA here. Among the remaining three paths, path 4 corresponds to *TC* using method traces and statement coverage, path 5 to *TD* using method traces and dynamic points-to sets, and path 6 to *TCD* combining all the three forms of dynamic data.

4.1.1. Process Flow. The DIAPRO framework works in three phases: static analysis, runtime, and post-processing, as shown at the top of Figure 3, which outlines the typical overall process flow of predictive dynamic impact prediction shared by the five DDIA-instance paths described above.

— *Phase I: static analysis.* For the three new instances and DIVER, the first step of static analysis is to compute the whole-program static dependencies. This step is realized by building the approximate dependence graph, after running the exception profiler for making optimal decisions on the inclusion of control dependencies due to exceptional control flows, just as in DIVER (see Section 3.2). Since such decisions can greatly affect the dependence-graph size hence overall

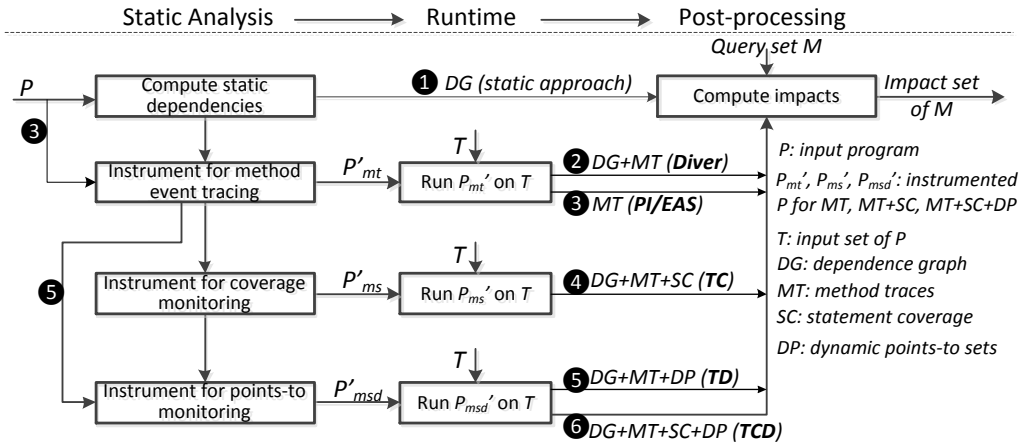

Fig. 3: A unified framework for DDIA that incorporates static dependence information and various forms of dynamic data to achieve multiple levels of cost-effectiveness tradeoffs of dynamic impact prediction: The marked paths (with circled 1 to 6) illustrate the five DDIA techniques we studied and the static impact analysis as a special instantiation (marked by path 1).

performance of the DDIA, this step is an integral part (pre-processing) of the framework. Next, probes for runtime monitors that collect required 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 of the framework. The outputs of this phase are the dependence graph and instrumented version P' of P (P' is further differentiated for different DDIA instances with subscripts indicating the dynamic data that the instrumentation intends to monitor).

— *Phase II: runtime.* During the runtime, the framework executes T on P' to generate the dynamic information configured during static analysis. For example, all the three forms of dynamic data will be produced in *TCD*, while for DIVER 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 as per our experience.

— *Phase III: post-processing.* The last phase, impact computation is essentially the post-processing of all the static and dynamic information collected during the previous two 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 such impact sets as the final result. Note that for any number of queries with respect to the same execution set, the previous two phases are performed once only, with their outputs shared by the impact computation for all queries.

4.1.2. Illustration. We illustrate DIAPRO through the five DDIA techniques instantiated from it using the example program and traces of Figure 2. In the example trace used by DIAPRO (at the bottom of the figure), *e* denotes method-entry events while *i* denotes method-returned-into events. Suppose the query set is {`M5`}, and `transform` is a library function that clones the input object and returns the clone after transforming it. In this example case, the ground-truth dynamic impact set (i.e., actual impact set) is {`M5`}. PI/EAS again finds the entire program (all methods) of E impacted as every method executes after the first entry event of `M5`; INFLUENCEDYNAMIC does not prune any method from that imprecise impact set because the influence [Breech et al. 2006] of `M1` propagates to each of other methods according to its influence graph for this case.

In contrast, the impact set of DIAPRO starts with $\{\texttt{M5}\}$ upon the occurrence of event $\texttt{M5}_e$, growing as more methods are found dynamically 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 DIVER, which assumes that line 14 is covered and object q at line 40 points to the same allocation site (of line 45) as the base object of field d at line 12, 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 12 of) M2 from (line 40 of) M5 continues to propagate (via line 14) to (line 28 of) M4. The first and second *heap* DD here is the only outgoing dependence of M5 and M2, respectively. As a result, the impact set computed by DIVER is $\{\texttt{M2}, \texttt{M4}, \texttt{M5}\}$.

However, with *TC*, which checks statement coverage and finds that statement 14 is not covered (w is 5 at line 13), the second *heap* DD above is not exercised. As a result, *TC* reports a more precise impact set $\{\texttt{M2}, \texttt{M5}\}$. Further, on top of *TC*, *TCD* checks dynamic points-to data in addition to statement coverage and thus finds that the two base objects, of field d at line 40 and that at line 12, are not aliased at runtime, so the first *heap* DD is skipped too. Consequently, *TCD* reports M5 as the only impacted method, thus gives the most precise (and accurate) impact set. Coincidentally, in this example, *TD* would find the same accurate impact set as *TCD* because by checking dynamic aliasing data the impact would not even propagate into M2 hence $\{\texttt{M2}, \texttt{M4}\}$ would also be pruned.

4.1.3. Application Context. Dynamic impact analysis enables developers to zero in on the particular operational profiles of the program under analysis, thus it is most useful when those specific, rather than all possible, program executions are the focus of the developer's ongoing task. Furthermore, DDIA assists with such tasks by exploiting the program code itself and its execution data without relying on additional software artifacts such as design documents and commit logs. Therefore, our DDIA framework is particularly suitable for dynamic impact analysis where those artifacts are not available. Additionally, as we illustrate above, DIAPRO provides a versatile kit for such analyses with which a variety of cost-effectiveness tradeoffs are required due to the diversity in task nature, change request, resource constraints, and time budget, which becomes increasingly prevalent in practice as we discussed in Section 2. Accordingly, for client analyses of DDIA in similar circumstances, such as regression testing and program understanding, our framework also offers a more attractive solution than existing ones.

On the other hand, as with any dynamic analysis, concrete program executions must be available in order to apply DIAPRO; also, the effectiveness of our framework is subject to the quality (e.g., coverage and representativeness) of the corresponding program inputs. Intuitively, inputs that exercise larger portion and more typical functionalities of the code would lead to more effective impact prediction and client-analysis results with DIAPRO.

Currently, DIAPRO focuses on the precision of resulting impact sets as the main metric of effectiveness, assuming perfect recall of the analysis. This assumption holds for the single program version and the particular execution data sets that our framework analyzes. Yet, the recall with respect to actual impacts for concrete changes, which DIAPRO has no any priori knowledge about as a predictive analysis, is very likely to be imperfect just as PI/EAS [Cai and Santelices 2015b]. While recall is no less critical than precision in general, we focus on precision in this work because (1) any dynamic impact prediction inherently suffers from imperfect recall, and (2) a highly imprecise impact set might not be worthy of inspection even if it is of perfect recall. We further discuss the recall metric of DDIA, including possible approaches to computing it, in Section 6.2.

4.2. DIAPRO Instances

4.2.1. TC: Trace plus Coverage. The method execution trace is the only form of dynamic data used in DIVER, which finds methods from the trace that are directly or transitively dependent on the query using the dependence graph. This hybrid approach consists of two major steps. The first step conceptually corresponds to the entire PI/EAS analysis: using the method trace, filter methods that are 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, DIVER 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 accomplishment of the higher precision over PI/EAS.

On top of DIVER (path 2 in Figure 3), the *TC* technique goes further to add statement coverage to the DDIA (path 4). While the method execution trace essentially informs the analysis of method-level coverage and execution order, the statement coverage offers a form of finer-grained execution data to help the DDIA exert a more aggressive false-positive pruning. With DIVER, 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 examined with respect to compile-time (*static*) dependencies rather than (*dynamic*) dependencies exercised by the inputs (from T). However, this conservative assumption can lead to falsely applied impact propagation hence false-positive impacts.

TC exploits statement coverage to prune those false positives—methods that are not dynamically impacted by (dependent on) the given query but falsely included in the resulting impact set of an analysis. And there are two alternative ways to realize 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 the DIAPRO post-processing, and (2) *post-pruning*, which incorporates the statement coverage data into the impact computation algorithm of DIAPRO along with the static dependencies, without pre-processing the dependence graph itself. Intuitively, both strategies contribute equivalently to the effectiveness (precision), but differently to the overhead, of this DDIA approach.

4.2.2. TD: Trace plus Dynamic Points-to Sets. *TD* (path 5 in Figure 3) seeks a different way from *TC* to prune false dynamic dependencies at method level by checking dynamic aliasing [Mock et al. 2005]. In short, *TC* addresses the imprecision of DIVER that comes from ignoring all *intraprocedural* control decisions (i.e., runtime evaluation of predicates inside individual methods). Yet, another source of the imprecision may exist too, which is resulted from spurious data dependencies due to imprecise static pointer analysis. Although the imprecision of static pointer analysis is a well-known problem as the points-to sets can only be conservatively approximated during static analysis, it is relatively simple to determine alias relations precisely at runtime (with respect to the specific inputs).

TD improves the precision of DDIA over DIVER by exploiting dynamic points-to sets. This technique collects the full set of allocation sites that each pointer (i.e., reference to heap objects) 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 related points-to sets. Given a data dependence $s1 \rightarrow s2$ in the dependence graph, for example, *TD* will regard it as a spurious dynamic dependence if there does not exist a variable $v1$ defined on $s1$ and a variable $v2$ used on $s2$ such that the points-to set of $v1$ and that of $v2$ have a non-empty intersection. In light of the method-execution event 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.

Similar to the pre- and post-pruning strategies available for *TC*, the method-level points-to sets can be applied either before or during impact computation. 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. We have developed both variants of *TD* (method- and method-instance-level), referred to as TD_{ml} and TD_{mil}, respectively. In comparison, TD_{ml} tends to be more conservative thus only *approximates* the precise points-to sets that TD_{mil} captures, but incurs potentially lower time and space overheads than the other.

4.2.3. TCD: Combine All Together. *TCD* (path 6 in Figure 3) combines all the three types of execution data performing the finest-grained analysis among the five DDIA techniques 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. *TD* also prunes such false positives partially, although it does so in another, and complementary, approach relative to *TC*. Yet, *TC* suffers from the imprecision similar to DIVER in regards of aliasing: False aliases may still exist among statements that are all covered by the program inputs. In contrast, *TD* suffers from the imprecision similar to DIVER in regards of statement coverage: The dynamic-aliasing examination assumes that every aliasing-relevant statement in a method is covered, since the points-to data of each heap variable is collected at the granularity of method rather than statement.

Therefore, to further improve the precision of DDIA, *TCD* exploits both statement coverage and dynamic points-to sets to find spurious dependencies that would be either filtered by *TC* or *TD* alone: A dynamic dependence identified by *TCD* must be one that has both the dependent and dependee statements covered, and the aliasing check also passed if it is a heap DD. Once spurious dependencies are identified, they are used for pruning associated false impacts from the method execution trace. As in *TD*, since the points-to sets can be gathered and utilized at either method level or method-instance level, *TCD* has two variants too, referred to as TCD_{ml} and TCD_{mil}, using the method- and method-instance-level dynamic aliasing data, respectively.

4.2.4. Other Instantiation. Beyond DIVER and the three new DIAPRO instances described above, more other impact-prediction techniques can be instantiated from this framework. For example, as we mentioned earlier, PATHIMPACT and EAS are both the instance of the framework, marked by path 3, that does not use static dependencies (i.e., the dependence graph) but purely relies on the method-level execution trace to predict impacts of the input queries; and path 1 indicates a static approach which only utilizes the static dependencies to predict the impacts. In addition, INFLUENCEDYNAMIC can be instantiated from this framework as well, which would be a variant of DIVER that ignores intraprocedural data dependencies and all control dependencies (although control-flow information would implicitly remains in use), and predicts the impacts based on the method trace and a partial dependence graph (i.e., the *influence graph*) [Breech et al. 2006].

The framework could also be instantiated such that a DDIA technique would use the dynamic points-to sets or statement coverage as the only type of dynamic data, or even without using the dependence graph. While these and even more other instances potentially exist, we leave them out from this work for future explorations. To distinguish the three new DDIA techniques from all other (including DIVER and PI/EAS) instances, and also for brevity, we hereafter refer to as DIAPRO any of *TC*, *TD*, and *TCD*, or them together, as we name the framework.

4.3. Data Collection

As far as the five DIAPRO instances for DDIA are concerned, our framework utilizes one form of static information and three forms of dynamic information. The static information, represented via the approximate dependence graph, is collected during the static-analysis phase and serialized as a holistic data (graph) structure afterwards. The serialized graph is later passed to the post-processing phase but not used during runtime. All of the dynamic information are collected at runtime.

Specifically, the method event trace per test input consists two lists of pairs, one list for method-entry events and the other for returned-into events. Each pair contains two entries: the timestamp of the event occurrence and the method associated with the event. Statement coverage is simply collected as a list of unique statements executed per test input. Dynamic points-to sets are collected separately for the two levels of granularity. The method-level points-to sets are collected by recording, for each heap variable, a list of memory locations (i.e., object addresses) pointed to by the variable. To represent the method-instance-level aliasing data, a mapping per heap variable is used, from a timestamp to the list of memory locations the variable points to. This timestamp equals to the one that is associated with the method event during which the statement defining or using the

heap variable executes. The consistency between these two timestamps is necessary for checking dynamic aliasing at method-instance level during the post-processing phase.

4.4. Analysis Algorithm

The advantageous cost-effectiveness gained by DIVER over its prior alternatives is not just about using more program information (i.e., static dependencies); what is also crucial is a careful design of the hybrid analysis algorithm. This section presents the details of the DIAPRO impact-computation algorithm, as shown in Algorithm 1 in pseudo code. The algorithm demonstrates how different types of data are utilized in synergy according to the configurations of respective instances. It unifies the post-processing algorithms of all the five DDIA techniques (PI/EAS, DIVER, and the three new DIAPRO instances). It also subsumes thus supersedes the impact-set querying algorithm of DIVER [Cai and Santelices 2014] which only deals with static dependencies and method-execution events. Nonetheless, DIAPRO reuses two elements from the DIVER querying algorithm: the dependence classification and the impact-propagation rules as per the two major classes of interprocedural dependencies (see Section 3.2).

The algorithm inputs the approximate dependence graph G of the program under analysis, a query q consisting of a single method (referred to as *single-method query*) from the program, and three forms of dynamic information generated by running the program on a given input set: method event trace L, statement coverage set *SC*, and dynamic aliasing data *DP*; and the output is the impact set *ImpactSet* of q. For a query consisting of multiple methods (referred to as *multiple-method query*), the impact set will be obtained by first running this algorithm multiple times (easily in parallel if needed since each run is independent of all others) each on a single method as one query and then taking the union of impact sets of all the single-method queries. Of the four inputs, only the event trace is required while the others are optional. The trace includes the full sequence of method events stored in ascending order of the time of event occurrences (i.e., the event timestamps).

The algorithm performs the impact-set querying for an instance of DIAPRO that is able to utilize (and needs) the maximal set of inputs available in the argument list with *null* indicating unavailability, procedurally in the following four steps (as also annotated in the pseudo code). We used two notations to facilitate the presentation: *m(e)* retrieves the method associated with an event e; *nodes(m)* indicates the set of nodes in the dependence graph representing the statements in method m. In addition, that "a node v dynamically depends on a node u with respect to dynamic points-to sets *DP*" means that the heap variable a defined at u points to the same memory location (allocation site) pointed to by the heap variable b used at v at runtime as per *DP*, where a and b are regarded as aliased to each other during static analysis; the dynamic dependence is checked by examining the dynamic points-to sets of a and b to see whether these two sets intersect. Accordingly, that "a path exists from u to v with respect to *DP*" means that the target node dynamically depends on the source of each edge on the path, meaning that each aliasing-induced static dependence on that path is exercised according to all relevant dynamic points-to sets in *DP*.

(1) The algorithm initializes the impact set to initially include the query itself which trivially impacts itself, and two other accessory data structures. To keep track of impact propagation in the trace via each edge in the dependence graph, we maintain a mapping (*ImpactedSrcs*) from each of the four (interprocedural) edge types (*heap*, *parameter*, *return*, and *control dependence*) to impacted sources of dependencies (edges) of that type, because whether the impact propagates along the edge to its target depends on both the type of that edge and the impact status of its source (i.e., whether the source has been impacted or not). Also, to help update this mapping, we keep tab on the method (*prevm*) preceding the one associated with the event being processed. After these initializations, the algorithm repeats the following steps on each event while traversing the trace. Since methods only executed before the query cannot be impacted, method events in the trace are skipped until the first occurrence of an event of the query is encountered (line 5).

ALGORITHM 1: The unified impact-set querying algorithm of DIAPRO

Input: approximate dependence graph G, method-execution event trace L, covered statement set *SC*, dynamic points-to sets *DP*, query q
Output: impact set *ImpactSet* of q for L

```
/* 1. initialize the impact set and accessory data structures */
ImpactedSrcs := ∅; // mapping from edge type to set of impacted sources of edges of the type
ImpactSet := {q}; // impact set of query q, which itself is trivially impacted first
prevm := null; // preceding method occurrence
foreach method event e∈L do
    skip e until the first event of q is encountered;
    /* 2. start impact propagation from within the query method */
    if G==null then
        add m(e) to ImpactSet;
        continue;
    end
    if m(e)==q and e is a method-entry event then
        foreach interprocedural edge (u,v) of type t in G such that u∈nodes(m(e)) do
            if SC==null or u∈SC then
                add u to ImpactedSrcs[t];
            end
        end
        prevm := m(e);
        continue;
    end
    /* 3. propagate impacts transitively */
    foreach interprocedural edge (u1,v1) of type t1 in G such that v1∈nodes(m(e)) do
        if (e is a method-entry event and (u1,v1) is a return edge) or
        (e is a method-returned-into event and (u1,v1) is a parameter edge) or
        u1∉ImpactedSrcs[t] then
            continue;
        end
        if SC≠null and v1∉SC then
            continue;
        end
        if (u1,v1) is not a heap edge or
        (DP==null or v1 dynamically depends on u1 with respect to DP) then
            add m(e) to ImpactSet;
        end
        foreach interprocedural edge (u2,v2) of type t2 in G such that u2∈nodes(m(e)) do
            if (SC==null or u2∈SC) and
            ∃ a path v1⇝u2 in G with respect to DP if DP≠null and also to SC if SC≠null then
                add u2 to ImpactedSrcs[t2];
            end
        end
    end
    /* 4. cease impact-propagation along adjacent edges */
    if prevm≠m(e) then
        foreach edge type t ∈ {parameter, return} do
            remove nodes(prevm) from ImpactedSrcs[t];
        end
        prevm := m(e);
    end
end
return ImpactSet
```

(2) If the static dependence information is not provided, DIAPRO computes impact sets purely based on the method event trace: Any method executed after the query is added to the impact set (lines 6–9), effectively corresponding to the PI/EAS instance. Otherwise, the dependence graph is always utilized. Every time an entry event of the query is encountered, sources of outgoing edges from any nodes of the query are marked as impacted (line 13). When statement coverage is available, an additional check precedes (line 12): sources not covered are skipped. Then, the process jumps to next event (line 17). Since we are only concerned about dynamic dependencies at method level for the method-level dynamic impact prediction, only *interprocedural* edges are examined as in the next step.

(3) When an event e of method m other than the query is encountered, every interprocedural incoming edge d to nodes of m is examined. An incoming *parameter* edge will not propagate impacts from its source to its target on a method-returned-into event, and an incoming *return* edge will not do so on a method-entry event, according to the semantics of these edge and event types. Also, the source of d must be impacted already when checking if the edge can propagate the impact further along. If any of these three conditions are not satisfied, the process jumps to the next edge (lines 20–24). Otherwise, the target of d is checked against statement coverage if the coverage set is provided (lines 25). When dynamic aliasing data is available, the edge d is checked against that data if d is a heap edge. If these screenings are all passed, DIAPRO regards that d indeed propagates the impacts, and then adds m to the impact set (line 30). Next, the algorithm matches the source of each interprocedural outgoing edge d' of m against the target of d to see if the target is reachable from that source via a path in the dependence graph G. Such source of d' is screened by the statement coverage if available, and the reachability is computed using both the coverage and aliasing data if provided. Note that only *intraprocedural* dependencies within (the PDG of) m are considered when finding such a path. If any of such paths exists, that source of d' is marked as impacted (by adding it to *ImpactedSrcs*) (line 35).

(4) For *adjacent* edges, since impacts reached their sources can propagate only one (different) method away in the trace, the "impacted" status of the sources *expires* after the targets have attempted gaining the impacts in the previous step. Thus, when the trace traversal moved that far already, nodes of the most recent last method (*prevm*) are removed from *ImpactedSrcs* (line 41). Since *posterior* edges are able to propagate impacts any far forward along the trace, sources of such edges are not removed once added to the mapping.

During this process, checking a node in the dependence graph against statement coverage is simply done by examining the membership of the statement corresponding to the node in *SC*, which is collected during runtime in relevant DIAPRO instances (*TC* and *TCD*). In contrast, how the dynamic aliasing is checked depends on how fine-grained the points-to sets are collected at runtime. For method-level data, *DP* has a list of memory locations for each heap variable, thus the checking against an edge (u,v) is realized by examining the intersection between the two such lists of u and v concerning the relevant heap variable, regardless of which method event is being processed. For method-instance-level data, *DP* has a mapping from a timestamp to such a list per heap variable, thus the checking is realized by first retrieving the two lists whose timestamps match the timestamp of the method event being processed and then examining the intersection. Once all events in trace L are processed, the impact set is obtained for query q with respect to the trace (line 46).

The initialization step takes constant time $\mathcal{O}(1)$, and the computation time of each iteration of the outermost loop comprises that of the other three steps. Let E be the largest edge set, and V the largest node set, among all per-method PDGs, the cost of both step 2 and 4 is bounded by $\mathcal{O}(|V|)$, and step 3 takes $\mathcal{O}(|V|^2|E|)$ time where the reachability solution costs $\mathcal{O}(|V|+|E|)$ to traverse the PDG of m with generally $|E| \gg |V|$. With appropriate data structures (e.g., hashing), traversing the interprocedural edges at lines 19 and 32 is reduced to visiting ports on the PDG, and the set operations (e.g., membership check) only need constant time. Thus, the time complexity of Algorithm 1 is $\mathcal{O}(|L||V|^2|E|)$ as a loose upper bound, where $|L|$ is the number of events in the trace.

In comparison, the time complexity of DIVER [Cai and Santelices 2014] is the same as DIAPRO while that of EAS [Apiwattanapong et al. 2005] is linear of the number of unique methods in L.

5. IMPLEMENTATION

We implemented the DIAPRO framework for Java based on the Soot byte-code analysis framework [Lam et al. 2011] and our data-flow analysis and instrumentation toolkit DUA-FORENSICS [Santelices et al. 2013]. For the dependence graph construction including the exception-driven control flow analysis, and PI/EAS facilities including the method-execution event instrumentation and monitoring, we reused relevant modules of DIVER [Cai and Santelices 2014]. Particularly in the implementation of method-event tracing, we adopted the exception-handling enhancement [Cai et al. 2014; Cai and Santelices 2015b] to ensure the safety of analysis results relative to the execution data utilized. In addition, we developed separate facilities for monitoring statement coverage and dynamic points-to sets, and implemented the unified querying algorithm as presented in Algorithm 1 which incorporates these two forms of dynamic information on top of the dependence graph and method-event traces previously used in DIVER.

For statement coverage monitoring capable of dealing with exceptional control flows, we computed whole-program reverse dominance frontiers (RDFs) and branch-induced control dependencies [Aho et al. 2006], whereby statement coverage was monitored indirectly through branch coverage monitoring [Santelices et al. 2013]. This indirect treatment bypasses an otherwise straightforward but much heavier instrumentation scheme of inserting one probe after every single statement, and thus helps avoid related issues that may occur: In our experience, for some of the experimental subjects we used, the straightforward instrumentation ended up with unexecutable code because the size (i.e., the number of bytecode instructions) of a few instrumented methods exceeds the JVM limit on that size. By monitoring branch coverage, we only insert one such probe for each branch. From covered branches and control dependencies in the dependence graph, covered statements are easily derived: If a branch is covered, all statements directly control-dependent on that branch are covered too; otherwise, none of them are covered.

For collecting the dynamic aliasing data, we added the monitoring of heap-object memory addresses to the method-event monitors. To that end, a single probe is inserted for all heap variables defined or used at a statement if any. Monitoring dynamic points-to sets at method level and that at method-instance level share the same instrumentation with differences only consisting in the runtime monitors devoted to collecting these data.

For better efficiency, the current implementation parallelizes the impact computation of a multiple-method query by simply running the algorithm on each member method of the query in a separate thread: In case of queries having very-large numbers (e.g., over 100) of methods, the parallel processing can be adapted to assign multiple single-method queries to one thread. The parallelization scheme is straightforward as each single-method querying thread is independent of all other concurrent querying threads, as we mentioned earlier. Also, the inputs to the querying algorithm is loaded only once and then shared by all the threads.

6. EMPIRICAL STUDY

We designed and conducted an empirical study of the DIAPRO framework, specifically through the three new instances (*TC*, *TD*, and *TCD*) against the two representative previous peer approaches (PI/EAS and DIVER). Our main goal with this study was twofold. First, we wanted to assess the effectiveness of the new techniques and their practicality in terms of time and storage overheads, so as to gauge their cost-effectiveness, relative to the prior alternatives. Second, we intended to analyze the effects of static and different types of dynamic data on the cost-effectiveness of DDIA in order to inform about future design of more advanced DDIA techniques.

Accordingly, we formulated four research questions as follows in this study:

— **RQ1** How effective are the new DIAPRO instances in comparison to existing representative DDIA techniques?

Table I: STATISTICS OF SUBJECTS OF STUDY ON ARBITRARY QUERIES

Subject	Description	#LoC	#Classes	#Methods	#Tests	#Queries
Schedule1	priority scheduler	290	1	24	2,650	20
NanoXML-v1	XML parser	3,521	92	282	214	172
Ant-v0	project builder	18,830	214	1,863	112	607
XML-security-v1	encryption library	22,361	222	1,928	92	632
BCEL 5.3	binary analyzer	34,839	455	3,834	75	993
JMeter-v2	performance gauge	35,547	352	3,054	79	732
JABA	program-analysis tool	37,919	419	3,332	70	1,129
OpenNLP 1.5	NLP utilities	43,405	734	4,141	344	1,657
PDFBox 1.1	PDF processor	59,576	596	5,401	29	588
ArgoUML 0.20	UML modeling kit	102,400	1,202	8,856	211	1,098

— **RQ2** Are the time and storage costs of all the DIAPRO instances, hence potentially the framework in general, acceptable for them to be practically useful and adoptable?
— **RQ3** What are the effects of the three forms of dynamic information, namely the method event trace, statement coverage, and dynamic points-to sets, on the overall performance of DDIA?
— **RQ4** Does the DIAPRO approach provide multiple cost-effective options for dynamic impact prediction and fill the gap in the present cost-effectiveness design space of DDIA?

6.1. Study Setup

We use the implementation of DIAPRO as described in Section 5 for our empirical study. Since the five DDIA techniques are implemented for Java, we utilize Java subject programs only but those of a variety of scales and application domains. Table I lists the ten subjects of our study, including for each subject a brief description, the source scale measured by the number of non-comment non-blank lines of code (*#LoC*) in Java, the total number of classes (*#Classes*) and that of methods (*#Methods*) defined in the subject, the size of test input set (*#Tests*), and the number of methods covered by (at least one test in) the input set that are used as impact-set queries (*#Queries*).

Half of the ten subjects, Schedule1, NanoXML, Ant, XML-security and JMeter, are obtained from the Software-artifact Infrastructure Repository (SIR) [Do et al. 2005]. For these subjects, when multiple versions are available, we picked the first version of each for which reasonable sizes of test inputs are provided along with the programs. Among the other five subjects, BCEL[3], OpenNLP[4], PDFBox[5], and ArgoUML[6] are retrieved from their respective online source repositories, where we randomly picked a recent stable release for each. Finally, JABA[7] is obtained directly from its developers for a stable version (the version number is not available). Of these programs, Schedule1 and OpenNLP produce the largest total traces, BCEL, JABA, OpenNLP, PDFBox, and ArgoUML have relatively large and/or dense dependence graphs, and others are in between, together constituting a diverse set of subjects with respect to evaluating the performance of DDIA.

For each of the ten subjects, we exhaustively take every single method as a query against each of the five studied DDIA techniques to involve all possible single-method queries in our study; we also randomly group various numbers of methods as multiple-method queries such that every single method is included in at least one such query. The motivation for this exhaustive strategy is to obtain a reasonable bound of expected performance of DIAPRO by assessing the performance against *arbitrary queries*. However, during realistic evolution of a software system, it may not always be the case that every method will be queried for its impacts given the mostly varying roles and importance of different methods in the system. Instead, the developer is likely to focus more on some subsets of methods than others as queries when performing predictive impact analysis.

[3]http://commons.apache.org/proper/commons-bcel/.
[4]http://opennlp.apache.org/.
[5]http://pdfbox.apache.org/.
[6]http://argouml.tigris.org/.
[7]http://gamma.cc.gatech.edu/jaba.html.

Table II: STATISTICS OF SUBJECTS OF STUDY ON REPOSITORY QUERIES

Repository	Range Retrieved	#Valid	#LoC	#Methods	#Changed	#Tests
XML-security 1.0	350550–350854	33	27,018–28,021	1,712–1,924	25.5 (84.4)	92
Ant 1.4	269454–269755	44	41,487–42,678	4,144–4,399	10.4 (13.0)	183
PDFBox 1.1	925362–1038216	66	63,720–65,163	5,334–5,951	7.4 (14.3)	32

Therefore, to complement the measurement on arbitrary queries, we intended to consider as queries those methods that are actually changed between source-code revisions in real-world software evolution scenarios. These methods may be more representative of *practical* queries than the arbitrary queries because it is expected that the developer should indeed first assess the potential impacts of changing the methods before actually applying the changes. We refer to such queries based on real repository changes as *repository queries* to distinguish them from arbitrary ones. To that end, we chose open-source repositories for three subjects from the ten-subject pool of Table I: XML-security, Ant, and PDFBox (these three all happen to use the SVN version control system). This selection is mainly driven by (1) our additional intent to obtain an estimation of the lower-bound performance of our techniques, (2) our desire to use, for more representative experimental results, repositories that are still in active and frequent evolution and that are of subjects of different sizes with reasonable length of revision history, and (3) the attempt to cover both single- and multiple-method queries. Concerning these criteria, we found that both DIVER and our new DIAPRO instances had the worst-case cost-effectiveness for arbitrary queries on the three chosen subjects and JABA among the ten; however, JABA does not meet the second criterion since it is not in active development now and has a few archived revisions only. As a secondary benefit, picking these repositories from the ten-subject pool enables the reuse of experimentation utilities (e.g., environment settings and experiment scripts) as well.

Specifically, for each of the chosen three, we randomly chose a revision prior to the latest stable release of the subject to start with. Then, we iteratively checked out the next revisions until we found no less than 30 revisions that each had at least one line of source code changed from its previous revision, dismissing non-code changes such as those in comments or other artifacts (i.e., build files, release notes, or software documentation). In addition, newly added methods are not counted as changed ones (thus will not be used as queries) since our techniques are predictive and work on the program version where changes have not been applied yet: It is impossible to predict impacts of methods not even present in the program under analysis. Therefore, revisions that had no code changes other than the addition of new methods are also skipped during the procedure. We refer to revisions eventually singled out as such for use in our study as *valid revisions* as opposed to the whole range of revisions we checked out (before reaching at least 30 valid ones). Table II shows the outcome of this procedure, including for each repository (subject), the version of release before which we chose the starting revision (in the first column), the range of revisions retrieved (*Range Retrieved*), the number of valid revisions (*#Valid*), the range of sizes of valid revisions in terms of LoC in Java (*#Loc*), the range of total numbers of methods in single revisions among the valid ones (*#Methods*), the mean (and standard deviation in parentheses) of numbers of methods changed between every two consecutive valid revisions (*#Changed*), and the number of test inputs provided for the starting revision (*#Tests*).

6.2. Experimental Methodology

For our experiments, we applied DIVER, *TC*, *TD* (including its two variants TD_{ml} and TD_{mil}), *TCD* (including its two variants TCD_{ml} and TCD_{mil}), and PI/EAS separately to each subject on a RedHat Linux workstation with an Intel i5-2400 3.10GHz processor and 4GB DDR2 RAM. To obtain the method traces and other dynamic data for the dynamic analysis in DIAPRO, we used the entire test suite provided with each subject. Particularly for each of the three subjects in the study on repository queries, we consistently employed the test inputs provided with the starting revision on all the valid revisions because there was no changes found in any of those revisions for test inputs.

Importantly, since no method that only executes before the query can be impacted (by the query or any changes to be made in it), PI/EAS would not miss dynamic impacts for the set of executions it analyzed; on the basis of PI/EAS impact sets, DIAPRO only prunes those methods that it ascertains are not dependent on the query, according to the static program dependencies and other forms of the dynamic information used for the impact computation, with DIVER, *TC* (or *TD*), and *TCD* incrementally doing so in a conservative manner. Note that this incremental and conservative nature is revealed in Algorithm 1 indeed: given that the static analysis it uses is sound, DIVER prunes a method in an impact set given by PI/EAS *only* if it finds out that the method is not even statically dependent on the query; then *TC* (or *TD*) prunes a method that passes the check of DIVER as impacted *only* if that static dependence, albeit exists in the dependence graph, is not exercised by any executions—at least one of the two nodes on the dependence is not covered, as *TC* finds out (or that dependence is statically established because of aliasing but dynamic points-to data reveal that the aliasing does not occur, as *TD* finds out) in the executions. Finally, *TCD* cuts off false-positive impacts given by DIVER *only* if the static dependence is not exercised due to either case (the order of cutting both types of static dependencies that are not exercised does not affect the ultimate impact sets *TCD* computes as we also verified empirically). Thus, we regard DIAPRO as *safe* with respect to the execution data it utilizes. Under this assumption about recall (in fact, a dynamic analysis is defined as sound if all its results hold modulo the program executions used by the analysis [Jackson and Rinard 2000]), we express the precision in terms of the impact-set size ratio of DIAPRO to PI/EAS and use precision as the metric of effectiveness in impact prediction, which is appropriate at least for evaluating relative precision *gains* as we focus on in our studies as also adopted previously [Orso et al. 2004; Apiwattanapong et al. 2005]. As we identified earlier, PI/EAS is still representative of the most cost-effective and best-known prior DDIA techniques that are safe (as dynamic analysis). Thus, we continue to use it as the baseline in our evaluation.

Through these relative effectiveness (precision) improvements versus relative overhead increases, we are able to *compare* the cost-effectiveness tradeoffs of various DDIA techniques. All these measurements together suffice for answering the research questions we aim to address through this study. For an impact-analysis technique, the precision and recall in absolute terms can be computed with respect to actual (ground-truth) impacts of concrete changes which, however, are not known to predictive analyses like ours in DIAPRO. Such precision and recall metrics can be *estimated*, though, by computing the ground truth for a large number and variety of concrete sample changes as in our predictive accuracy study [Cai and Santelices 2015b]. The study also targeted PI/EAS, thus can help associate the relative precision measures in this work with absolute ones.

Alternatively, correlated changes across different revisions can be mined, and co-change patterns be inferred, from software repositories [Hassan and Holt 2006; Petrenko and Rajlich 2009]; then, actual impact sets can be estimated through evolutionary couplings and correlated changes [Zimmermann et al. 2005; Kagdi et al. 2010]. It is noteworthy, however, that the credibility of such derived actual impact sets would be subject to the availability and quality of appropriate change commits in the repositories used for the co-change mining and coupling measure computation. Nevertheless, complementary to the approach in our accuracy study, using actual impact sets estimated with the coupling measures to further evaluate predictive DDIA such as ours would be of interest for future work (e.g., taking those actual impact sets as the ground truth and computing recall and precision of predicted impact sets in absolute terms [Petrenko and Rajlich 2009; Gethers et al. 2012]).

6.2.1. Procedure. In the study on arbitrary queries, for each of the five DDIA techniques compared and the ten subjects studied, we computed the impact sets for all single-method queries—in each subject every method is taken as a query. For multiple-method queries, we consider nine query sizes (i.e., the number of methods in one such query) from 2 to 10; and for each query size qs, we created the first query by randomly selecting qs unique methods from the complete set P_M of methods in the subject P and then iteratively selected the next qs methods to create the next queries, without replacement, until the set P_M is exhausted (the last query may has less than qs

methods). As such, every method of the subject is included in exactly one multiple-method query. We then computed the impact sets of all these multiple-method queries, using the parallel querying approach implemented in DIAPRO as we described earlier. Further, to reduce potential biases in the random method grouping, we repeated the entire experiment on multiple-method queries 1,000 times with a distinct randomization seed applied in each repetition.

In the study on repository queries, the number of queries for each of the three subjects was the number of valid revisions we studied for that subject, with all methods changed in each revision together taken as one query, whether it be a single- or multiple-method query. In the cases of multiple-method queries, the parallel querying was used as in the cases of multiple-method arbitrary queries. We computed the impact set of each repository query per subject using each of the five DDIA techniques separately. As in the arbitrary-query study, the impact set of some queries is empty because none of the member methods were covered by any test input. We excluded the impact sets of such queries from our evaluation as no dynamic impact prediction technique would be applied to those queries (to obtain meaningful results).

6.2.2. Metrics and Measures. The central metric of our study is the *performance* of the DIAPRO framework, which is concretely demonstrated through that of its instances and characterized through three supporting metrics: effectiveness, efficiency (cost), and average cost-effectiveness. Yet, we did not compute the accuracy of each impact set relative to ground-truth impacts to measure the effectiveness. Instead, we used PI/EAS as a common baseline for DIVER and the new DIAPRO instances and computed the ratios of impact-set sizes of the latter four DDIA techniques over PI/EAS as the effectiveness metric in light of the soundness of our framework as explained above. Also, it is crucial to note that both DIVER and DIAPRO are expected to improve the precision of PI/EAS without affecting its recall, because the additional dependence analyses they employ for pruning false impacts from PI/EAS remain conservative in nature: As is shown in Algorithm 1, a method executed after the query is dismissed from the impact set only if the method is definitely not to be dynamically dependent on the query with respect to the program information utilized.

The first supporting metric is the effectiveness (relative precision) expressed by impact-set size ratios with respect to the baseline analysis PI/EAS. For each query, we calculated such ratios, each for one of the four DDIA techniques over the baseline. As a result, we report the mean and five quartiles of these ratios for all queries per subject. The second supporting metric is the time and space costs. For each technique and subject, we collected the time cost per DDIA phase separately; for static-analysis and runtime phases that are needed only once for all queries, we did one measurement of the time; for post-processing costs, we calculated the mean and standard deviation of query time for all queries with one measurement per query; for space costs, we gauged the disk storage consumed during the entire DDIA process flow of DIAPRO. We attempted for consistent system environment for all runs of each of these experiment steps and took the total CPU time elapsed during each run as the time-cost measure.

With both the effectiveness and cost metrics, the average cost-effectiveness of DIAPRO was calculated as a combination of them as the last supporting metric. 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 DIVER and the three new DIAPRO instances relative to the baseline, for the one-time cost of the first two DDIA phases and per-query post-processing cost separately. Such comparisons shed light on the effects of different dynamic data on the cost-effectiveness of DDIA. Also, by examining the contrast between DIVER and PI/EAS versus that between each of the three new DIAPRO instances and DIVER, the impact of the static dependencies can be discovered as well. Note that we calculated effectiveness-to-cost ratios instead of those of cost to effectiveness to facilitate the understanding and interpretation of this metric: the higher the metric and the better (more cost-effective).

To further investigate those effects, we also assess the statistical significance of the differences in effectiveness among the DIAPRO instances. To that end, we performed a set of paired Wilcoxon signed-rank tests [Walpole et al. 2011] where the two groups were the impact-set sizes given

by each pair of techniques being contrasted. We adopted this non-parametric test to lift the assumption about the distribution of underlying data points. Furthermore, for each subject, we applied the Holm-Bonferroni correction [Holm 1979] for all the p-values from multiple comparisons between DDIA techniques and report adjusted values. In addition to p-values computed separately for each subject at the 0.95 confidence level, we also report combined p-values for each pair of techniques using the Fisher method [Mosteller and Fisher 1948]. Finally, to complement the hypothesis testing, we compute the Cliff's delta [Cliff 1996] (in a paired setting with α=.05), a non-parametric effect-size metric, to gauge the magnitude of differences in impact-set sizes between two compared techniques. We omit combined effect sizes though as we are not aware of an appropriate procedure for aggregating multiple Cliff's deltas while per-subject effect sizes suffice already for our evaluation. We also omit statistical analyses for efficiency and cost-effectiveness measurements because the space costs, and the time costs of two of the three DIAPRO phases, are measured once per subject and technique, so is in the case of cost-effectiveness results, resulting in numbers of data points that are too small (only one pair of sample data for each pair of techniques contrasted on each subject) to be sufficient for such analyses. In our study, there were four *independent variables*, namely the four types of program information having effects on the performance of DIAPRO: static dependencies, method event trace, statement coverage, and dynamic points-to sets. The *dependent variables* are the three supporting metrics of DIAPRO performance.

6.3. Threats to Validity

One *internal* threat to our study is possibility of having implementation errors in the DIAPRO framework with the five DDIA techniques on top it and our experimentation scripts. Fortunately, all these DDIA techniques were based on Soot and DUA-FORENSICS that have both matured over many years. Part of the framework, including the dependence graph construction and method event tracing, was adapted from DIVER which has already been tested and tuned for two years. We paid special attention to verify the modules of our framework that are used for monitoring and applying the additional dynamic data relative to DIVER, including the statement coverage and dynamic aliasing data at both method and method-instance levels. We checked each phase of DIAPRO and the holistic DDIA process flow, for each individual instance separately, against the example program E, two small subjects Schedule1 and NanoXML, and sample cases in larger subjects, and ensured that they all function correctly before applying them to our experimental study. Similar strategy and test data were also used for our manual verification of the study setup and data-analysis scripts.

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. For the three repositories chosen for the study on repository queries, the ranges of revisions we examined do not necessarily represent the entire evolution cycle of respective subjects. To mitigate this risk, we picked repositories that cover various sizes and application domains and have reasonably length of revision histories. Also, the three subjects are all under active development and evolution, and we have retrieved and studied considerable numbers of revisions. An additional *external* threat lies in the representativeness of the multiple-method queries studied for all such queries in the respective subject. To reduce this threat, we considered a total of 10 different query sizes and repeated the random query construction covering the entire program for 1,000 times per query size and subject. Also, the range of query sizes we studied closely covered the average numbers of methods changed between revisions (i.e., real-world query sizes) for the three chosen repositories. During our time measurements, changes in system loads and OS activities can affect the validity of efficiency results, which we contained by maintaining the same experiment environments (e.g., constant number of running processes) for each static analysis phase, subject execution, and query processing.

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

Table III: MEAN EFFECTIVENESS ON ARBITRARY SINGLE-METHOD QUERIES

Subject	Mean PI/EAS Impact-Set Size	Mean Impact-Set Size Ratios to PI/EAS (%)					
		DIVER	*TC*	TD_{ml}	TD_{mil}	TCD_{ml}	TCD_{mil}
Schedule1	18.0 (1.6)	71.31	71.31	71.31	69.75	66.60	65.04
NanoXML	82.6 (48.1)	51.68	45.57	51.59	50.36	45.51	43.48
Ant	159.5 (173.4)	25.71	17.22	25.63	25.38	17.17	16.89
XML-security	199.8 (168.4)	28.76	24.75	28.67	28.37	24.59	24.12
BCEL	446.0 (280.1)	17.40	11.73	14.92	10.16	11.68	7.61
JMeter	149.6 (172.6)	18.80	18.17	18.69	18.23	18.09	17.55
JABA	677.0 (301.2)	66.91	63.80	66.48	65.50	63.32	61.51
OpenNLP	146.9 (202.4)	26.75	25.80	26.74	22.69	25.79	20.92
PDFBox	258.2 (172.6)	57.76	56.02	56.10	51.65	54.38	47.92
ArgoUML	151.0 (261.2)	31.50	29.36	31.35	31.29	29.19	29.15
Overall	**278.1 (151.0)**	**34.55**	**31.50**	**33.99**	**31.87**	**31.23**	**28.71**

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 the DDIA techniques we studied. One more *construct* threat is that we picked the worst-case subjects out of the ten-subject pool used for the arbitrary-query study to estimate the lower-bound performance of DIAPRO (the new instances in particular). Yet, the three chosen subjects may not represent all of their kind. Moreover, it is noticeable that, in despite of the same project names, these three subjects have considerably evolved beyond their versions as used in that arbitrary study, according to the growth in source sizes and in some cases in test input set sizes also. In consequence, the lower bound estimated might not exactly reflect the actual bound with respect to the entire pool of ten subjects. However, our code review confirmed that the main features and core functionalities of these subjects remain steady during the evolution period we examined, even for Ant which had the most drastic size increases in both source code and test inputs among the three. Nonetheless, we do not claim that the lower bound we report generalizes for all practical use cases of DIAPRO.

Finally, a *conclusion* threat is the appropriateness of our statistical analyses. To reduce this threat, we used a non-parametric hypothesis testing and a non-parametric effect-size metric which both make no assumptions about the distribution of the underlying data (e.g., normality). 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.

7. RESULTS AND ANALYSIS

In this section, we report and discuss the empirical results, focusing on the effectiveness (RQ1), efficiency (RQ2), and cost-effectiveness (RQ3) of the DIAPRO techniques with respect to the baseline approach. Then, based on these results, we examine the effects of various program information used in the studied DDIA techniques on their performance (RQ3), and draw conclusions on the variety of cost-effective options our framework offers (RQ4). We followed the study methodology with the experimental setup described above to obtain all the following results. Note that the statistical tests on differences in effectiveness presented for RQ3 serves RQ1 as well, although we put them once and collectively in Section 7.3 for a compact presentation.

7.1. RQ1: Effectiveness

7.1.1. Arbitrary Single-Method Queries. The main effectiveness results for arbitrary single-method queries are shown in Figure 4, 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 DIVER and the three DIAPRO instances (including the finer-grained variants), with each group characterized by a single boxplot that consists of the maximum (upper whisker), 75% quartile

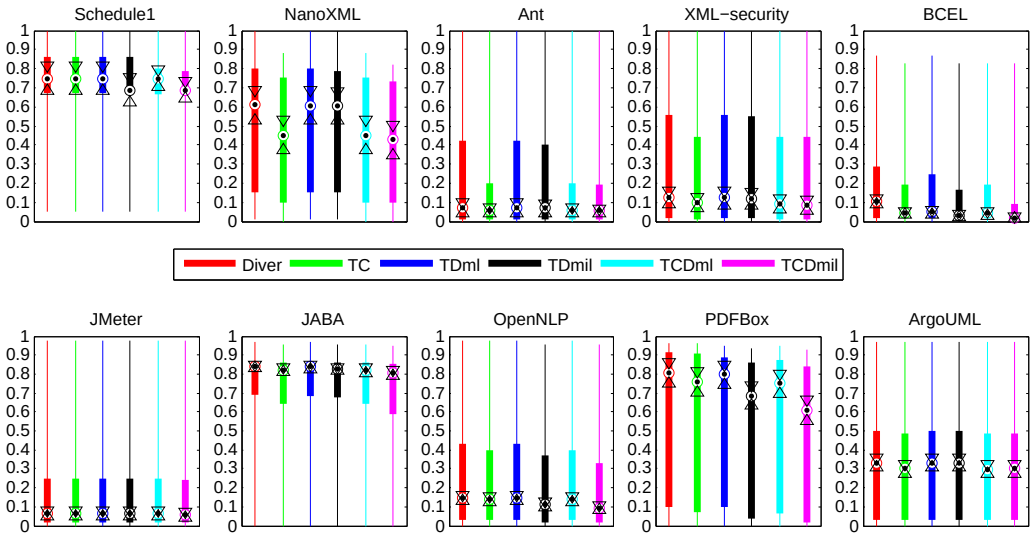

Fig. 4: Effectiveness of DIVER and DIAPRO expressed as impact-set size ratios over PI/EAS on arbitrary single-method queries. The lower the ratio, the better (more effective).

(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. Within each plot, these comparison intervals collectively give a quick indicator of the statistical significance of the differences in *medians* among the six groups in that plot: For any two groups, their medians are significantly different at the 5% significance level if their intervals do not overlap.

Overall, the four hybrid DDIA techniques all greatly reduced the sizes of the purely execution-order-based baseline impact sets, with DIVER, *TC* (*or* TD_{ml}, TD_{mil}), TCD_{ml}, and TCD_{mil} in order continuously improving the relative precision. Although DIAPRO achieved noticeably smaller gains over DIVER in contrast to the improvements DIVER attained over the baseline, increasing addition of dynamic data to DDIA appeared to bring in increasingly enhanced effectiveness of the analysis. Also, except for Schedule1, there always existed cases in which the hybrid DDIA techniques cut off the entire impact set reported by the baseline analysis—they reported empty impact sets for queries with empty method body—leading to the minimal size ratios of 0%. And except for the four smallest subjects, which had a few worst cases where the baseline impact sets were not reduced at all, every single instance of DIAPRO was always able to prune some false impacts: The maximal ratios were constantly below 100% for other subjects. In addition, for NanoXML, employment of statement coverage helped overcome such worst cases.

The largest gains in effectiveness were seen by Ant, XML-security, BCEL, JMeter, and OpenNLP, for which DIAPRO drastically pruned the baseline impacts by over 85% for more than half of the queries. Especially for BCEL and JMeter, 75% of the queries received impact sets from DIAPRO less than 30% large as those from the baseline. ArgoUML also had a substantial impact-set reduction from DIAPRO, which reported as potentially impacted no more than 35% of the methods produced by the baseline. DIAPRO improved relatively less for Schedule1 and JABA, partly due to their tight inter-function couplings as we manually found out. For PDFBox, DIAPRO cut only 15% or less of baseline impacts on worst-case queries, and for 50% of the total queries the reduction was merely around 30%. Nevertheless, quite a few (over 25%) queries received a large effectiveness improvement (about 90% impact-set reduction).

Comparing DIAPRO to DIVER suggests that neither statement coverage nor dynamic points-to sets contributed to the effectiveness gain of the hybrid DDIA techniques over the baseline so much

Table IV: MEAN EFFECTIVENESS ON ARBITRARY MULTIPLE-METHOD QUERIES

Technique	Query size									
	1	2	3	4	5	6	7	8	9	10
DIVER	0.35(0.28)	0.35(0.28)	0.35(0.27)	0.35(0.25)	0.35(0.24)	0.35(0.23)	0.35(0.23)	0.35(0.22)	0.35(0.21)	0.35(0.20)
TC	0.31(0.27)	0.31(0.26)	0.31(0.25)	0.31(0.24)	0.31(0.23)	0.32(0.22)	0.32(0.21)	0.32(0.20)	0.32(0.20)	0.32(0.19)
$\boldsymbol{TD}_{ml}$	0.34(0.29)	0.34(0.28)	0.34(0.27)	0.34(0.25)	0.34(0.24)	0.34(0.23)	0.34(0.23)	0.34(0.22)	0.34(0.21)	0.35(0.20)
$\boldsymbol{TD}_{mil}$	0.32(0.28)	0.32(0.26)	0.32(0.25)	0.32(0.24)	0.32(0.23)	0.32(0.22)	0.32(0.21)	0.32(0.21)	0.32(0.20)	0.32(0.19)
$\boldsymbol{TCD}_{ml}$	0.31(0.27)	0.31(0.26)	0.31(0.25)	0.31(0.24)	0.31(0.23)	0.31(0.22)	0.31(0.21)	0.32(0.20)	0.32(0.20)	0.32(0.19)
$\boldsymbol{TCD}_{mil}$	0.29(0.26)	0.29(0.25)	0.29(0.24)	0.29(0.23)	0.29(0.22)	0.29(0.21)	0.29(0.20)	0.29(0.19)	0.29(0.19)	0.29(0.18)

as the static dependence information did. In the meanwhile, for most of these ten subjects, the comparison intervals show no significant differences in the medians of the impact-set size ratios among DIVER and all DIAPRO instances. However, there are two findings worth noting. First, relative to DIVER, DIAPRO either produced even smaller impact sets for the majority of all queries, according to the lower median ratios; or brought down the impact sets for more queries, according to the lower 25% and 75% quartiles, for almost all subjects. The only exception is JMeter, for which the improvements of any DIAPRO instance over DIVER were barely noticeable. This exception may suggest relatively strong internal logic couplings within the individual methods of JMeter, thus incoming dependencies have generally strong connectivity to outgoing dependencies in the PDGs of this program. Also, since DIVER has already pruned most of the false positives from the baseline impact sets, according to the prominently low ratio it attained, intuitively it is harder to bring the impact-set sizes down further. Second, in the cases of NanoXML, BCEL, OpenNLP, and PDFBox, DIAPRO, at least with certain instances, accomplished significantly smaller median impact-set sizes than DIVER, which is mainly ascribed to the employment of statement coverage for the first two and to the use of dynamic aliasing data (at method-instance level in particular) for the last two.

Table III summarizes the means of the effectiveness data points depicted in Figure 4 for corresponding subjects, and in addition the mean effectiveness over all the ten subjects (in the bottom row). These means complement to the effectiveness distribution shown by the figure. In all, the exact numbers on the impact-set size ratios of DIAPRO over DIVER further demonstrated the positive contributions of additional dynamic data beyond method traces to DDIA effectiveness in average cases: Using the extra data consistently led to further improvements in the effectiveness. While reasonably appreciable for NanoXML, Ant, BCEL, and PDFBox, these improvements seem to be moderate in absolute terms for other subjects. However, the baseline impact-set sizes listed in the first column suggest that even seemingly small impact-set reductions may truly matter, implying considerable savings of inspection efforts by developers when using DIAPRO versus DIVER. In fact, results of our Wilcoxon tests on, and the effect sizes of, the differences in impact-set sizes, reported in (the top portion of) Table XI, also show that DIAPRO produced *significantly* smaller impact sets by average than DIVER, so did DIVER over PI/EAS, for all subjects but Schedule1.

7.1.2. Arbitrary Multiple-Method Queries. Figure 5 summarizes the effectiveness results of DIAPRO versus DIVER on arbitrary multiple-method queries, including a separate plot for each individual subject. Due to large numbers of data points, we show the summary results, instead of the distribution of effectiveness metrics for all queries per subject. In each plot, the x axis lists the ten query sizes from 1 to 10 and the y axis indicates the mean effectiveness, with standard deviations shown by the error bars; the data points are all impact-set size ratios of the six DDIA instances (shown by the six curves) over the baseline PI/EAS in the 1,000 repetitions. Including the means for query size of 1 that correspond to the numbers in Table III to facilitate comparisons, these plots mainly highlight the variation of DIAPRO performance in average cases in contrast to DIVER when the query size varies. The closeness of the effectiveness means and variances among the six techniques results in visual occlusions among the error bars even overlapping among the curves: The data sets are plotted in the same order as in which the techniques are listed in the legend.

While the effectiveness fluctuates indeed with changing query sizes, either growing or downsizing but both monotonously in most cases (with an only exception with Schedule1 at query size of 7),

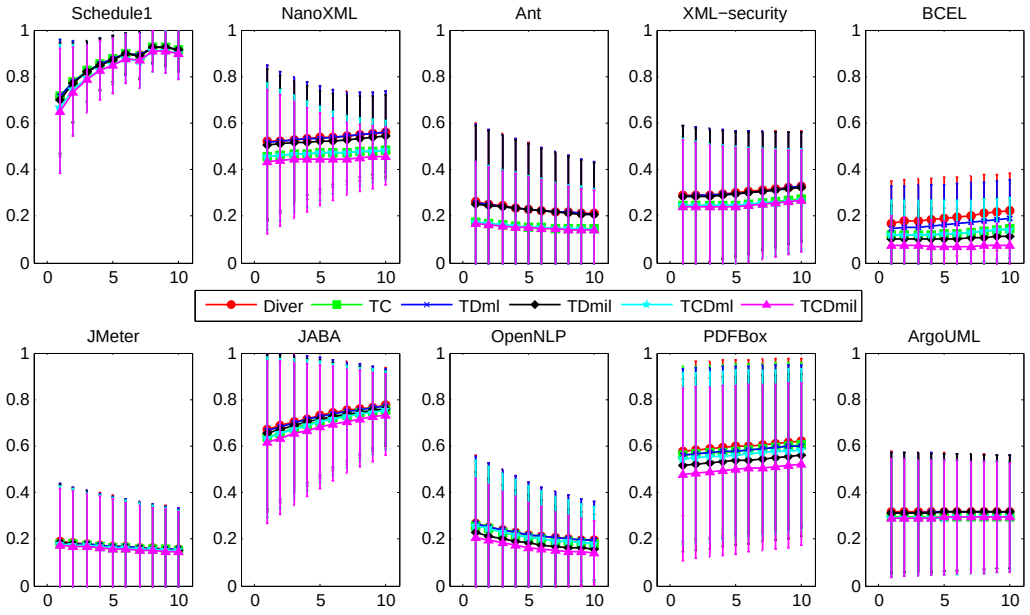

Fig. 5: Effectiveness of DIVER and DIAPRO expressed as impact-set size ratios on arbitrary multiple-method queries for query sizes from 1 to 10. The lower the ratio, the better (more effective).

the fluctuations are generally small for the majority of these subjects: Both the effectiveness means and standard deviations are stable as the query size increases. One of the key observations is that the variations of effectiveness of all the DIAPRO instances are highly consistent with those of DIVER, in terms of both the changes in the means with query sizes and the variances of the means. In addition, the distances among the six curves, although quite small for some subjects such as JMeter, reaffirm the constant effectiveness advantages of DIAPRO over DIVER as well as of instances with more dynamic data utilized over those with less, regardless of the differences in query sizes. Yet another finding is that DIAPRO, the best-performing instance TCD_{mil} in particular, has even more steady effectiveness than DIVER: It appears that the higher the effectiveness a DDIA technique achieves, the more stable the effectiveness of that technique is with query-size variations. A similar trend is found among the DIAPRO instances as well.

The general consistency in the effectiveness of the six DDIA instances is further highlighted when putting the data points of all subjects together, as summarized in Table IV. These exact numbers of the over-all-subject statistics consolidate the earlier observation that DIAPRO produces impact sets for multiple-method queries as effectively as for single-method ones, in despite of different query sizes, for the ten sizes and subjects studied at least. Moreover, if excluding the exceptional case, the smallest subject Schedule1 which exhibits a quite different pattern in the correlation between effectiveness and query size as noted and explained above, the overall variations would have been even more negligible. Thus, we omit the statistical analyses on multiple-method queries.

7.1.3. Repository Queries. The distribution of all effectiveness metrics of DIVER and DIAPRO on repository queries is plotted in Figure 6 in the same format as Figure 4. The type of repository noted in the plot title (in the parenthesis) is added to distinguish these subjects from those of the same names used in the arbitrary-query experiments. Note that the repository queries for which the data points were computed are either single- or multiple-method queries depending on how many methods developers changed in one commit, although multiple-method changes predominate in all revisions of any of the three repositories studied. Overall, the incremental benefits of using more

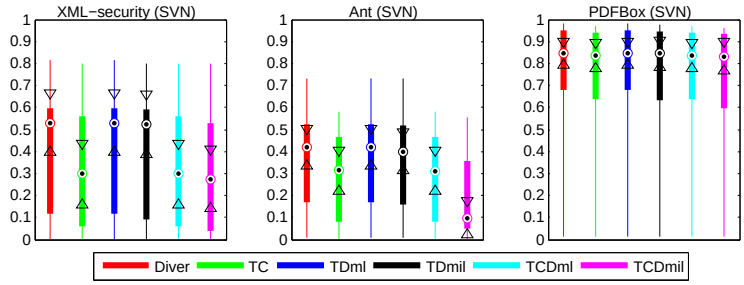

Fig. 6: Effectiveness of DIVER and DIAPRO expressed as impact-set size ratios on repository queries. The lower the ratio, the better (more effective).

Table V: MEAN EFFECTIVENESS ON REPOSITORY QUERIES

Subject	Technique (hybrid DDIA techniques)					
	DIVER	***TC***	TD_{ml}	TD_{mil}	TCD_{ml}	TCD_{mil}
XML-security (SVN)	0.43(0.27)	0.34(0.27)	0.43(0.27)	0.41(0.28)	0.34(0.27)	0.32(0.27)
Ant (SVN)	0.36(0.22)	0.29(0.20)	0.36(0.22)	0.34(0.22)	0.29(0.20)	0.19(0.19)
PDFBox (SVN)	0.75(0.26)	0.73(0.28)	0.74(0.26)	0.73(0.27)	0.72(0.28)	0.71(0.28)
Overall	**0.56(0.31)**	**0.51(0.33)**	**0.56(0.31)**	**0.54(0.31)**	**0.50(0.33)**	**0.46(0.34)**

dynamic data are demonstrated again and in a way very similar to they were shown in the cases of arbitrary queries, according to the effectiveness improvements DIAPRO gained over DIVER. While the dynamic information that contributed the most to the improvements varied (e.g., statement coverage for XML-security and dynamic points-to sets for Ant), DIAPRO largely enhanced the DIVER effectiveness for two of the three repositories. The exception with PDFBox is akin to that seen in the experiments on arbitrary queries, suggesting that DIAPRO may not be as effective for some particular programs as in most cases. Our manual examination revealed that PDFBox, with another worst-case subject JABA, has much stronger interprocedural logical couplings than other subjects (e.g., the `viewer`, `writer`, and `parser` modules of PDFBox densely interfaced via its `pdmodel` module, and three tightly connected components, `instruction`, `du`, and `graph`, in JABA), potentially causing most of the methods executed after a query were indeed dependent on the query at runtime hence the relatively high precision of the baseline impact sets.

On the other hand, the effectiveness DIAPRO and DIVER attained is mostly lower on repository queries than on arbitrary queries in absolute terms, for each of the three subjects, especially XML-security and Ant, across the two respective experiments. Two possible reasons may explain this difference. First, the source code of these subjects has considerably evolved (all grew in size) between the version we utilized for the experiment on arbitrary queries and the versions we retrieved for the experiment on repository queries. For instance, the versions of Ant from the SVN repository have grown to be more than twice as large as the version from the SIR repository. Second, compared to that of repository queries which are based on practical changes made by developers, the full set of arbitrary queries may contain much more trivial cases—queries that have very few transitive interprocedural dependencies. Impacts originated in such queries usually propagate quite shortly, typically ending up with having themselves only in their DIAPRO impact sets. In addition, the total number of queries (i.e., that of valid revisions) is much smaller in the study on repository queries than that on arbitrary queries, which could also have contributed to the difference in question. Yet, it is also noteworthy that (1) the individual differences with these three subjects are much smaller than the overall differences, and (2) both the individual and overall differences between DIAPRO and DIVER are very close, across the two experiments.

Table V shows the mean effectiveness of DIAPRO versus DIVER for each subject and over all the three subjects, with standard deviations of corresponding means (in parentheses).

Table VI: MEANS (STANDARD DEVIATIONS) OF POST-PROCESSING COMPUTATION COSTS IN SECONDS OF THE DIAPRO FRAMEWORK FOR ARBITRARY SINGLE-METHOD QUERIES

Subject	PI/EAS	DIVER	*TC*	TD_{ml}	TD_{mil}	TCD_{ml}	TCD_{mil}
Schedule1	0.7 (0.1)	14.6 (6.0)	15.7 (5.9)	18.9 (5.5)	42.4 (6.3)	19.2 (5.8)	44.3 (6.7)
NanoXML	0.1 (0.1)	6.2 (8.8)	6.4 (8.9)	5.7 (7.9)	7.7 (10.0)	5.6 (7.7)	7.9 (10.4)
Ant	0.1 (0.1)	3.2 (7.6)	3.4 (7.9)	3.1 (6.8)	4.9 (9.1)	3.3 (7.2)	5.2 (9.7)
XML-security	0.1 (0.1)	7.4 (9.6)	8.1 (10.4)	7.7 (9.9)	16.4 (20.4)	8.2 (10.5)	16.9 (20.9)
BCEL	0.1 (0.1)	116.1 (105.6)	107.9 (98.1)	113.6 (103.3)	154.9 (140.8)	118.7 (107.8)	153.5 (139.3)
JMeter	0.1 (0.1)	2.3 (7.8)	2.3 (7.9)	1.8 (5.6)	2.1 (5.9)	1.8 (5.6)	2.2 (6.2)
JABA	0.3 (0.2)	78.3 (82.5)	99.7 (102.5)	70.2 (71.1)	80.9 (77.1)	82.6 (83.0)	105.2 (99.7)
OpenNLP	0.1 (0.1)	75.2 (228.5)	74.8 (227.8)	72.7 (222.2)	95.6 (254.7)	74.3 (224.6)	85.3 (234.7)
PDFBox	0.1 (0.1)	113.2 (149.7)	114.6 (151.4)	111.6 (147.1)	145.7 (172.3)	104.4 (138.1)	139.6 (162.9)
ArgoUML	0.1 (0.1)	15.9 (58.2)	15.9 (57.8)	12.4 (42.6)	15.3 (48.6)	12.6 (42.8)	15.8 (49.9)
Overall	**0.1 (0.1)**	**55.5 (1.6)**	**57.5 (135.3)**	**52.6 (128.9)**	**68.6 (157.8)**	**54.9 (130.9)**	**69.4 (146.6)**

Individually, DIAPRO improves the effectiveness against DIVER largely for XML-security and Ant but moderately for PDFBox, consistent with what was observed from the distribution of all effectiveness measurements in Figure 6. As shown, the best-performing instance of DIAPRO can reduce the impact-set sizes of DIVER by 18% overall, implying an increase in precision by 22%. It is noteworthy that the overall means have been largely skewed (downplayed) by the worst-case subject PDFBox which received much smaller effectiveness gains while having much more queries hence much larger weights in the overall statistics than the other two subjects. Also, recall that these three subjects were chosen to represent the worst-case subjects in the ten-subject pool used in the study on arbitrary queries for an attempt to estimate the lower-bound performance of DIAPRO: They are both among the subjects for which DIVER achieved the least effectiveness improvements over the baseline PI/EAS and among those for which DIAPRO improved the least further beyond DIVER, for queries of larger sizes in particular (see Figure 5); and only subjects with accessible and reasonably long revision history were considered in this selection process, as we mentioned earlier in the study design. Nonetheless, results of statistical analyses in Table XII indicate that DIAPRO, with most of its instances (all but TD_{ml}), reduced the average size of impact sets significantly in contrast to DIVER and PI/EAS for the three studied repositories both individually and collectively.

7.2. RQ2: Efficiency

7.2.1. Arbitrary Queries. Table VI summarizes the means and standard deviations (in the parentheses) of post-processing (mainly trace deserialization and impact computation) costs incurred by the baseline PI/EAS, DIVER, and the DIAPRO instances studied, for all single-method queries per subject and over all the ten subjects (in the bottom row). First of all, the cost of PI/EAS was much smaller than that of any of the six hybrid DDIA approaches. This was expected because of not only the larger size of method-execution traces (full method event sequences by our framework versus two integers per method by PI/EAS), but also the additional static (dependence graph) and dynamic information (coverage and aliasing data) analyzed by the hybrid techniques. Second, in most cases the average post-processing overhead of DDIA increased with the addition of more program information to the analysis. Yet, starting from DIVER beyond, the increments appeared to be slight, much smaller than the gap between PI/EAS and the other techniques. For instance, even the largest of such increments among the six hybrid techniques—the one between the most expensive (heaviest) instance TCD_{mil} of DIAPRO and DIVER—is about 20% only.

On the other hand, compared to DIVER, at least one instance of DIAPRO incurred even lower costs for all the ten subjects but Schedule1. This observation is not surprising, as the employment of additional dynamic information can not only prune more false positives from the baseline results but also iteratively reduce the cost of traversing the static dependence graph during the entire process of impact computation: The static dependencies to be traversed can also be gradually pruned by using the dynamic data. For example, since DIAPRO uses method-level statement coverage, once it is found that a dependence (at least one of its two end nodes) was not covered, the dependence

Table VII: STORAGE COSTS OF DIAPRO FOR ARBITRARY SINGLE-METHOD QUERIES

Subject	D.G. Size in MB	Execution Data Size in MB						
		PI/EAS	DIVER	*TC*	TD_{ml}	TD_{mil}	TCD_{ml}	TCD_{mil}
Schedule1	0.05	1.54	5.36	7.08	9.62	15.08	11.34	16.8
NanoXML	0.9	0.36	1.56	2.09	2.06	3.41	2.6	3.95
Ant	5.49	0.35	1.69	2.14	3	5.09	3.45	5.54
XML-security	5.38	0.37	1.8	2.2	3.46	9.21	3.85	9.6
BCEL	11.6	0.29	23.04	23.7	33.82	57.49	34.48	58.14
JMeter	9.03	0.2	0.7	0.94	1.22	1.96	1.46	2.19
JABA	20.42	0.9	9.79	11.6	15.99	24.07	17.8	25.88
OpenNLP	20.97	0.34	83.87	84.44	88.14	145.05	88.71	145.62
PDFBox	17.1	0.09	14.78	14.88	15.75	32.46	15.86	32.56
ArgoUML	40.8	0.66	3.91	4.72	7.15	13.57	7.95	14.38
Average	**13.174**	**0.510**	**14.650**	**15.379**	**18.021**	**30.739**	**18.750**	**31.466**

may be skipped in all the following traversals of the dependence graph. When the total of such savings *overweighs* the total extra overheads brought by the additional dynamic data, the ultimate post-processing costs of the DDIA using those additional data will be even lower than otherwise.

In fact, two of the subjects (JMeter and ArgoUML) have seen a consistent overweighing as such: every instance of DIAPRO has incurred less post-processing overhead than DIVER. Moreover, from comparing TCD_{ml} with *TC*, it can be seen that adding method-level dynamic aliasing data even reduced the ultimate cost for seven out of the ten subjects. A possible cause is that spurious alias-induced data dependencies account for a relatively large portion of all dependencies in these subjects. In consequence, for the reason similar to that for the previous observation (overweighing), pruning those spurious data dependencies using dynamic points-to sets sped up searches on the reduced dependence graph during the process of impact computation. For the exceptional case Schedule1, however, the small program size led to small dependence graph—thus reducing the graph has minor effect—while the largest number of inputs used for it implies generally larger overheads for generating additional dynamic data, in contrast to other 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 set (e.g., Schedule1). The lowest efficiency of DIAPRO was seen by BCEL, JABA, OpenNLP, and PDFBox, which have much larger dependence graphs and/or longer traces than other subjects (see Table VII). In addition, the generally large standard deviations (relative to corresponding means) suggest that the post-processing time greatly fluctuates across different queries in most subjects. The *overall* numbers show that the querying cost of DIAPRO is about one minute per query. In fact, the average cost for the other six subjects is 10–20s. In all, the post-processing time of DIAPRO looks reasonable for its practical use. Note that two instances of DIAPRO, TD_{ml} and TCD_{ml} answered impact-set queries faster, while the other DIAPRO instances were only slightly slower, than DIVER for the consistently higher effectiveness of any DIAPRO instance over DIVER in return, implying eventual advantage of DIAPRO in cost-effectiveness—concerning the post-processing cost at least.

Comparing within the DIAPRO instances reveals that the method-instance-level aliasing data is much more expensive than the method-level data and statement coverage which mostly brought small overheads only in comparison to DIVER. This was anticipated because the finer-grained data can be substantially larger than the other two at coarser level. Table VII (columns 2–8) compares the sizes of such execution data used by DIAPRO versus PI/EAS and DIVER. As can be seen, the largest jump in such sizes among the DIAPRO instances occurred when the method-instance-level dynamic points-to data was added to the analysis. On the other hand, as expected also, the storage cost steadily grew with continuous addition of dynamic data. Yet, such costs were all quite small in terms of their absolute metric values—the largest by TCD_{mil} with OpenNLP was merely about 145MB. Another source of storage cost lies in the static dependence graph used by all the hybrid DDIA techniques. The table (in the second column) shows that these costs were even more negligible: 40MB in the worst case of ArgoUML. Intuitively, larger programs have larger dependence graphs,

Table VIII: STATIC-ANALYSIS TIME AND RUNTIME OVERHEAD OF THE DIAPRO FRAMEWORK FOR ARBITRARY SINGLE-METHOD QUERIES

Subject	Prof.	Static analysis cost in seconds					Runtime cost in seconds							
		PI/EAS	DIVER	*TC*	*TD*	*TCD*	Org.	PI/EAS	DIVER	*TC*	TD_{ml}	TD_{mil}	TCD_{ml}	TCD_{mil}
Schedule1	13	5	6	11	12	17	4	10	16	19	33	71	36	74
NanoXML	12	11	14	25	28	39	1	1	5	7	12	17	14	18
Ant	29	27	142	170	243	311	1	2	2	4	7	18	9	20
XML-security	37	33	158	190	248	280	4	5	15	20	25	55	30	60
BCEL	48	33	1,717	1,758	1,979	2,019	5	7	21	45	51	148	75	172
JMeter	51	38	372	408	728	764	12	13	15	28	29	34	42	47
JABA	62	55	289	326	563	600	11	12	14	25	38	78	51	89
OpenNLP	54	51	735	787	1,475	1,527	52	57	59	64	102	390	117	395
PDFBox	70	59	690	748	1,049	1,107	7	10	12	14	22	115	24	117
ArgoUML	190	172	7,465	7,542	11,822	11,998	8	10	11	23	26	60	38	72
Average	**57**	**49**	**1,159**	**1,196**	**1,815**	**1,866**	**11**	**13**	**17**	**25**	**35**	**99**	**44**	**106**

which seems to be testified as per this table. Yet, the existence of few exceptions (BCEL and PDFBox) suggests that there are other factors involved (e.g., the density of the dependence graph).

Table VIII shows the time cost of the first two DDIA phases of our framework, including the uncaught-exception profiling time incurred by all the hybrid DDIA techniques we studied (*Prof.*) and the execution time of the original (uninstrumented) programs (*Org.*). Generally, both the profiling and static-analysis time costs tend to increase with the program size, with the peak numbers of both seen by the largest subject ArgoUML. Yet, an obvious exception is found with BCEL, for which our manual examination of source code revealed that this subject contains quite a few methods of very-large sizes (over 3,000 instructions in a single method body) due to very-long hard-coded instantiations of arrays, which resulted in heavy PDGs hence extraordinary time for building the static dependence graph. By contrast, the runtime overhead did not exhibit consistent correlation with the subject size, as the size of input set utilized in the executions is another major factor. Indeed, the largest runtime overhead was encountered with OpenNLP which has both one of the largest source sizes and one of the largest input-set sizes. In addition, as previously led to the largest jump in storage costs, the method-instance-level dynamic aliasing data again led to the greatest cost growth here, in both the static-analysis and runtime phases.

The exception profiling was generally cheap, as was the generation of various forms of dynamic data at runtime, for which the cost was about 3 and 5 minutes at most, respectively. Static analysis was mostly efficient too, except for the largest subject ArgoUML and the special case of BCEL. Yet, for a program at the scale as ArgoUML (over 100KLOC), the two to three hours of static analysis may still be acceptable given the effectiveness gains in return that well pay off the cost: For example, even a 10% impact-set reduction could mean the saving of much more hours (than two or three) that would be spent on inspecting a hundred false-positive methods in such a large and complex system. Also, as the runtime phase, the static analysis incurs just *one-time* cost in that the static analysis needs to be performed only once—for any query with respect to the single program version under analysis, the outputs of these two phases can be reused. Furthermore, the static-analysis phase may be incorporated in nightly builds by practitioners. Note that for other subjects, the static analysis took only about 30 minutes mostly, as was the overall average shown in the bottom row.

In sum, DIAPRO comes with time and storage costs higher than the baseline, and in some cases higher than DIVER too. Yet, in terms of the absolute numbers, these costs are still reasonably practical. Also, such extra time and storage 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 DIAPRO instances. Finally, for arbitrary queries, we only reported the efficiency results on single-method ones, for two main reasons. First, multiple-method queries are dealt with in the post-processing phase through parallel processing of single-method queries each for one member method of the query. As a result, for the maximal query

size of ten at least, the impact-computation costs for multiple-method queries are very close (1.1x on average where the extra costs came from the overhead of parallelization) to those for single-method queries reported above. Second, with respect to our framework, query sizes do not affect the first two phases of DDIA and other steps in the post-processing phase than impact computation.

Table IX: MEANS (STANDARD DEVIATIONS) OF POST-PROCESSING COMPUTATION COSTS IN SECONDS OF THE DIAPRO FRAMEWORK FOR REPOSITORY QUERIES

Subject	**PI/EAS**	**DIVER**	***TC***	TD_{ml}	TD_{mil}	TCD_{ml}	TCD_{mil}
XML-security (SVN)	0.8 (2.1)	24.5 (23.3)	25.5 (23.6)	26.0 (23.9)	59.9 (41.7)	26.5 (23.9)	61.4 (42.5)
Ant (SVN)	1.3 (1.8)	8.1 (11.3)	9.3 (12.3)	10.7 (13.7)	22.6 (23.5)	11.3 (14.1)	23.8 (24.6)
PDFBox (SVN)	0.1 (0.1)	195.0 (282.7)	194.9 (279.2)	198.4 (285.9)	288.1 (347.1)	197.8 (283.2)	290.4 (349.7)
Overall	**0.6 (1.5)**	**98.1 (211.8)**	**98.7 (209.5)**	**100.8 (214.1)**	**153.7 (267.3)**	**100.9 (212.2)**	**155.5 (269.2)**

7.2.2. Repository Queries. The post-processing time cost of DIAPRO versus DIVER and PI/EAS is presented in Table IX, which summarizes the means and corresponding standard deviations (in the parentheses) of such costs per subject and over all subjects for all the valid revisions studied. The numbers show again that using the finer-grained aliasing data can substantially add to the per-query impact-computation time. For any of the DDIA techniques compared, the average costs here were also noticeably higher than those on single-method queries as shown in Table VI, with Ant and XML-security in particular. The main reason, as also suggested by the large standard deviations, is that a small portion of the repository queries had quite large query sizes (e.g., 55 with Ant and 473 with XML-security) hence particularly high querying costs: even with parallel processing, the larger the query size, the higher the total cost of computing the impact set due to the increasing overhead of parallelization (e.g., multlthreading overheads). Nevertheless, the median costs (not shown in the table) were much smaller than the means (e.g., 18.8s with Ant and 146.4s with PDFBox), suggesting a skewed distribution of the samples and that the waiting time for impact sets would be much shorter in most (over 50%) cases. An additional reason is the larger size of each of these three subjects than that of the corresponding one used for the study on single-method queries (e.g., over two times larger in the case of Ant). For Ant, which has seen the largest increase in the post-processing time among the three subjects, the growth by over 60% in the size of input set utilized for repository queries is another factor. Yet one more factor is the existence of much more trivial queries in the latter study as we discussed earlier—the extremely small time cost for those queries collectively brought down the means. Nevertheless, the extra overheads incurred by DIAPRO beyond DIVER were relatively small, and the waiting time of a few minutes in absolute terms, and a few more seconds relative to DIVER, should be practically worth investing for the greatly reduced impact-set sizes.

Table X lists the efficiency results for the first two DDIA phases of the DIAPRO framework, including the means and corresponding standard deviations of time costs incurred during those two phases for all valid revisions per subject and over all the three subjects. Compared to the numbers from the study on single-method queries, the profiling and static-analysis costs were very close for XML-security and PDFBox which have relatively small growth in size relative to the corresponding subjects used in that study. A much larger gap in these costs was seen by Ant because of the much larger growth in size with this subject: For these costs, and those of the runtime phase, the main factor in the efficiency difference here is the difference in source size whereas the sizes of query and input set are not relevant. Yet, in terms of the absolute numbers, the highest of the one-time static-analysis cost is half an hour at most for all the revisions of these three subjects we studied, which is reasonable and can be accommodated with solutions similar to those discussed before. The runtime overheads were higher, yet mostly just slightly, than those in Table VIII between corresponding subjects mainly due to the growth (in size) during the evolution period we examined, but all still remained reasonably acceptable: two to three minutes for the most time-consuming instance TCD_{mil}. Once again, method-instance-level dynamic aliasing data has been shown much more expensive than statement coverage, in the case of PDFBox in particular. We omitted reporting

Table X: STATIC-ANALYSIS AND RUNTIME COMPUTATION COSTS OF THE DIAPRO FRAMEWORK FOR REPOSITORY QUERIES

Subject	Prof.	Static analysis cost in seconds				
		PI/EAS	DIVER	*TC*	*TD*	*TCD*
XML-security (SVN)	34.7 (0.8)	27.6 (0.7)	107.4 (5.1)	135.1 (5.2)	197.8 (8.6)	225.5 (8.6)
Ant (SVN)	79.3 (3.2)	44.9 (1.0)	932.0 (32.6)	975.9 (33.1)	1,814.9 (61.9)	1,858.8 (62.6)
PDFBox (SVN)	71.5 (1.7)	57.3 (1.5)	718.3 (46.6)	772.7 (48.0)	1,108.6 (85.8)	1,163.0 (87.2)
Overall Average	**67.4 (15.7)**	**48.4 (11.2)**	**672.8 (281.8)**	**719.4 (288.7)**	**1,155.2 (546.2)**	**1,201.8 (550.9)**

Subject	Runtime cost in seconds							
	Org.	PI/EAS	DIVER	*TC*	TD_{ml}	TD_{mil}	TCD_{ml}	TCD_{mil}
XML-security (SVN)	4.7 (0.2)	5.7 (0.2)	8.3 (0.3)	13.8 (0.4)	20.6 (0.6)	89.6 (6.1)	26.1 (0.8)	95.1 (6.1)
Ant (SVN)	27.6 (0.5)	28.8 (2.1)	30.9 (0.7)	57.5 (0.9)	68.7 (1.7)	110.4 (2.3)	95.2 (1.7)	136.9 (2.5)
PDFBox (SVN)	5.9 (1.1)	9.1 (5.6)	29.7 (12.9)	30.2 (12.5)	56.5 (25.4)	294.9 (105.3)	57.0 (25.0)	295.5 (104.8)
Overall Average	**12.6 (10.8)**	**14.4 (9.5)**	**26.3 (12.5)**	**35.6 (18.0)**	**53.8 (24.6)**	**203.2 (122.8)**	**63.1 (29.9)**	**212.4 (116.2)**

the results on storage costs for repository queries as those costs were still negligible (less than 200MB in the worst case) for today's storage configurations.

7.3. RQ3: Effects of Dynamic Data

7.3.1. Arbitrary Queries. While focusing on the first two research questions, the foregoing comparative results on the effectiveness and efficiency of DIAPRO versus DIVER have already shed light on the effects of the additional dynamic data beyond method traces on the cost and effectiveness of DDIA. On the effectiveness, as shown in Figures 4, 5, and 6, statement coverage appeared to have much stronger impacts than dynamic aliasing data, according to the improvements of *TC* over DIVER versus *TD* over DIVER, despite the granularity of the aliasing data (method- or method-instance level); and in the most effective instance TCD_{mil} of DIAPRO, statement coverage also contributed more to its effectiveness gains over DIVER with almost every single subject and aggregately for all subjects, in despite of query sizes (single- or multiple-method) and categories (arbitrary or repository). Yet, there did exist cases, albeit few, in which dynamic points-to sets have stronger effects than statement coverage: for example, the cases of arbitrary queries in Schedule1 and PDFBox. Also found were cases in which both forms of dynamic information together have much stronger effects on either alone: for example, the cases of arbitrary queries in Schedule1, NanoXML, and PDFBox, and the cases of repository queries in Ant (SVN versions). Moreover, method-instance-level dynamic points-to sets always led to, albeit just slightly in most cases, larger effectiveness gains of DIAPRO over DIVER than method-level ones, which was most noticeable with Schedule1, BCEL, OpenNLP, and PDFBox, possibly because of relatively heavier use of pointers (object references in Java) in these subjects than others. On the cost of DDIA, however, the finer-grained aliasing data had consistently by far the strongest impacts than any other forms of either static or dynamic information. And even at the coarser (method) level, dynamic alias analysis was also always more expensive than collecting and using statement coverage.

To look further into the effects of static dependencies and different forms of dynamic data on the cost-effectiveness of DDIA, we performed an exhaustive set of statistical analyses examining the statistical significance and effect sizes of differences in impact-set sizes between DIVER and PI/EAS, between DIAPRO and DIVER, and among different DIAPRO instances. Our statistical-analysis results for all the studied DDIA techniques with respect to arbitrary single-method queries are shown in Table XI. For each subject, the table presents the p-values from sixteen paired two-sided Wilcoxon signed rank tests each comparing one pair of DIAPRO instances, for which the null hypothesis was constantly the means of the impact-set sizes from the two compared techniques being equal; and the effect size listed in the parentheses indicates the magnitude of impact-set size differences for that pair—using the Cliff's delta, an effect size of d means $100|d|\%$ of values in the first (left of the pair) group are larger (smaller if d is negative) than values in the second (right) one. The statistics covered all possible pairings among

Table XI: WILCOXON p-VALUES AND CLIFF'S DELTAS EFFECT SIZES (IN PARENTHESES) WITH RESPECT TO IMPACT-SET SIZES BETWEEN ALL PAIRS OF THE STUDIED DDIA TECHNIQUES (α=.05): ON ARBITRARY SINGLE-METHOD QUERIES

Subject	**PI/EAS:DIVER**	**DIVER:*TC***	**DIVER:*TD*$_{ml}$**	**DIVER:*TD*$_{mil}$**	**DIVER:*TCD*$_{ml}$**	**DIVER:*TCD*$_{mil}$**
Schedule1	2.13E-03 (.95)	1.00E+00 (0)	1.00E+00 (0)	4.43E-01 (.25)	4.43E-01 (.25)	5.32E-02 (.5)
NanoXML	7.66E-29 (.99)	1.06E-21 (.72)	3.53E-02 (.04)	7.87E-20 (.61)	6.50E-22 (.73)	9.86E-22 (.73)
Ant	9.41E-99 (.98)	2.54E-26 (.25)	2.44E-03 (.02)	2.01E-12 (.11)	1.79E-27 (.27)	1.64E-31 (.31)
XML-security	1.53E-100 (0.96)	7.61E-37 (.35)	3.78E-07 (.05)	6.65E-21 (.18)	1.99E-40 (.38)	4.63E-45 (.43)
BCEL	7.31E-163 (1)	5.15E-72 (.44)	1.86E-26 (.14)	2.69E-103 (.63)	1.09E-72 (.44)	2.13E-108 (.66)
JMeter	2.75E-120 (1)	5.76E-11 (.08)	1.78E-04 (.03)	3.22E-20 (.16)	3.11E-13 (.1)	3.48E-28 (.23)
JABA	4.83E-185 (1)	5.64E-150 (.81)	2.18E-51 (.39)	8.19E-107 (.69)	3.36E-125 (.76)	1.73E-129 (.77)
OpenNLP	3.92E-271 (1)	4.24E-41 (.14)	2.09E-03 (.01)	5.05E-116 (.42)	5.90E-43 (.15)	6.29E-131 (.48)
PDFBox	7.62E-97 (1)	1.30E-54 (.54)	4.30E-59 (.52)	9.37E-60 (.61)	7.35E-55 (.55)	1.92E-61 (.63)
ArgoUML	5.31E-180 (1)	3.09E-53 (.29)	1.08E-06 (.03)	3.78E-06 (.05)	9.39E-58 (.31)	3.29E-59 (.32)
Fisher Overall	**0.00E+00**	**0.00E+00**	**7.06E-141**	**0.00E+00**	**0.00E+00**	**0.00E+00**

Subject	***TC*:*TD*$_{ml}$**	***TC*:*TD*$_{mil}$**	***TC*:*TCD*$_{ml}$**	***TC*:*TCD*$_{mil}$**	***TD*$_{ml}$:*TD*$_{mil}$**
Schedule1	1.00E+00 (0)	4.43E-01 (.25)	4.43E-01 (.25)	5.32E-02 (.5)	4.43E-01 (.25)
NanoXML	1.05E-21 (-.7)	1.19E-18 (-.6)	1.74E-01 (.02)	1.61E-18 (.59)	6.86E-19 (.58)
Ant	3.76E-25 (-.23)	1.01E-15 (-.15)	2.44E-03 (.02)	2.53E-12 (.11)	1.61E-10 (.09)
XML-security	1.72E-35 (-.31)	2.29E-29 (-.25)	4.14E-09 (.07)	1.83E-22 (.2)	9.92E-16 (.13)
BCEL	3.77E-50 (-.23)	3.34E-36 (.42)	7.74E-29 (.15)	1.35E-101 (.62)	3.10E-103 (.63)
JMeter	6.06E-05 (-.06)	5.00E-07 (-.01)	2.29E-04 (.02)	6.83E-22 (.17)	6.12E-18 (.14)
JABA	4.96E-100 (-.65)	5.25E-01 (0)	7.16E-55 (.4)	5.45E-106 (.68)	1.46E-129 (.69)
OpenNLP	4.74E-39 (-.14)	1.04E-64 (.33)	1.40E-04 (.01)	9.52E-121 (.44)	9.63E-116 (.42)
PDFBox	3.12E-38 (-.04)	9.00E-50 (.55)	6.94E-58 (.51)	3.84E-60 (.61)	9.14E-60 (.61)
ArgoUML	1.18E-49 (-.26)	1.85E-46 (-.25)	1.27E-05 (.03)	2.08E-08 (.05)	3.28E-02 (.02)
Fisher Overall	**0.00E+00**	**5.60E-244**	**5.35E-147**	**0.00E+00**	**0.00E+00**

Subject	***TD*$_{ml}$:*TCD*$_{ml}$**	***TD*$_{ml}$:*TCD*$_{mil}$**	***TD*$_{mil}$:*TCD*$_{ml}$**	***TD*$_{mil}$:*TCD*$_{mil}$**	***TCD*$_{ml}$:*TCD*$_{mil}$**
Schedule1	4.43E-01 (.25)	5.32E-02 (.5)	1.00E+00 (0)	4.43E-01 (.25)	4.43E-01 (.25)
NanoXML	1.22E-21 (.72)	1.92E-21 (.72)	1.19E-18 (.62)	1.22E-21 (.73)	4.08E-18 (.58)
Ant	5.00E-26 (.25)	5.10E-30 (.29)	3.67E-17 (.17)	1.48E-27 (.27)	1.87E-10 (.09)
XML-security	1.01E-36 (.35)	2.30E-41 (.39)	8.40E-32 (.29)	3.79E-38 (.36)	1.65E-15 (.13)
BCEL	1.23E-72 (.44)	2.13E-108 (.66)	1.85E-35 (-.42)	5.20E-78 (.47)	1.57E-98 (.6)
JMeter	1.07E-10 (.08)	3.30E-26 (.21)	2.19E-05 (-.08)	3.86E-13 (.1)	5.97E-20 (.16)
JABA	1.64E-149 (.8)	2.29E-149 (.81)	2.12E-07 (.21)	7.16E-149 (.8)	8.47E-126 (.68)
OpenNLP	4.75E-42 (.15)	1.25E-130 (.48)	5.12E-64 (-.33)	7.53E-65 (.23)	3.97E-120 (.44)
PDFBox	2.26E-54 (.54)	1.92E-61 (.63)	3.14E-23 (-.15)	2.84E-54 (.54)	3.52E-60 (.61)
ArgoUML	1.48E-53 (.29)	7.62E-55 (.3)	2.62E-51 (.28)	5.96E-53 (.29)	6.19E-03 (.02)
Fisher Overall	**0.00E+00**	**0.00E+00**	**2.40E-229**	**0.00E+00**	**0.00E+00**

DIVER and the DIAPRO instances, plus the pairing of DIVER with the baseline. We dismissed comparing the DIAPRO instances to the baseline as the significance was found constantly quite strong with DIVER already, and we have confirmed before that any of the DIAPRO instance kept improving in effectiveness over DIVER. We also omitted such statistical tests with respect to arbitrary multiple-method queries because foregoing experiments have shown that query sizes had no substantial influence on the effectiveness of any of the studied DDIA techniques, for the ten different sizes we considered at least. In this table, the smaller the p value as a result of the signed rank test for the two techniques compared (through the comparison between the sample groups of impact-set sizes of the techniques), the stronger the statistical significance in the impact-set differences, hence the stronger the effect of the program information (mostly the dynamic data) utilized by one technique but not by the other on the effectiveness of DDIA, as indicated by larger effect sizes. For example, the lower statistical significance (and smaller effect sizes) from the test on DIVER versus *TC* (column DIVER:*TC*) than that on DIVER versus *TD*$_{ml}$ for PDFBox indicates that method-level dynamic points-to sets had stronger effect on the DDIA effectiveness (i.e., it contributed more to the effectiveness gains of DIAPRO over DIVER) than statement coverage for

Table XII: WILCOXON p-VALUES AND CLIFF'S DELTAS EFFECT SIZES (IN PARENTHESES) WITH RESPECT TO MEAN IMPACT-SET SIZES BETWEEN ALL PAIRS OF THE STUDIED DDIA TECHNIQUES (α=.05): ON REPOSITORY QUERIES

Instance Pair	XML-security (SVN)	Ant (SVN)	PDFBox (SVN)	Fisher Overall
PI/EAS:DIVER	9.02E-06 (1)	1.26E-07 (1)	2.64E-11 (1)	*4.18E-20*
DIVER:*TC*	6.27E-04 (.67)	1.45E-05 (.7)	1.87E-10 (.91)	*1.50E-15*
DIVER:*TD*$_{ml}$	1.00E+00 (0)	1.00E+00 (0)	8.15E-06 (.36)	*6.63E-04*
DIVER:*TD*$_{mil}$	9.80E-03 (.48)	2.06E-05 (.66)	9.81E-10 (.82)	*1.37E-13*
DIVER:*TCD*$_{ml}$	6.27E-04 (.67)	1.45E-05 (.7)	1.87E-10 (.91)	*1.50E-15*
DIVER:*TCD*$_{mil}$	6.27E-04 (.67)	3.87E-06 (.8)	1.61E-10 (.92)	*3.67E-16*
TC:*TD*$_{ml}$	8.77E-04 (-.61)	1.45E-05 (-.7)	6.33E-05 (-.45)	*3.36E-10*
TC:*TD*$_{mil}$	2.09E-03 (-.61)	2.11E-03 (-.48)	2.99E-02 (-.08)	*1.88E-05*
TC:*TCD*$_{ml}$	1.00E+00 (.03)	3.17E-03 (.25)	8.15E-06 (.36)	*4.42E-06*
TC:*TCD*$_{mil}$	1.06E-03 (.58)	3.53E-05 (.61)	9.81E-10 (.82)	*2.78E-14*
TD$_{ml}$:*TD*$_{mil}$	9.80E-03 (.49)	2.06E-05 (.66)	9.81E-10 (.82)	*1.37E-13*
TD$_{ml}$:*TCD*$_{ml}$	8.77E-04 (.61)	1.45E-05 (.7)	1.87E-10 (.91)	*2.06E-15*
TD$_{ml}$:*TCD*$_{mil}$	6.27E-04 (.61)	3.87E-06 (.8)	1.61E-10 (.92)	*3.67E-16*
TD$_{mil}$:*TCD*$_{ml}$	2.09E-03 (.61)	2.11E-03 (.48)	8.07E-03 (.44)	*5.87E-06*
TD$_{mil}$:*TCD*$_{mil}$	4.27E-04 (.67)	4.95E-06 (.77)	1.87E-10 (.91)	*5.37E-16*
TCD$_{ml}$:*TCD*$_{mil}$	1.06E-03 (.51)	3.53E-05 (.61)	9.81E-10 (.82)	*2.78E-14*

this subject. For another example, the p value of 1 (and effect size of 0) for DIVER versus *TC* with Schedule1 indicates that statement coverage had no significant effect on effectiveness in this case.

The second column of the top sheet confirmed again about the tremendous effectiveness advantage of hybrid approach over purely dynamic one (i.e., the baseline), mainly ascribed to the utilization of static program dependencies as previously studied [Cai and Santelices 2014]. The other columns of this sheet show that, beyond DIVER, the new DIAPRO instances also gained strongly significant improvements in the effectiveness by using more dynamic program information, although for different subjects the strongest significance was seen by the addition of different forms of dynamic information to the framework. With JABA, for instance, statement coverage helped DIAPRO prune the false positives of DIVER more significantly than dynamic points-to sets; yet with Schedule1 neither statement coverage nor dynamic points-to data at method level had significant effects, but a finer-grained alias analysis (at the method-instance level) enabled significant impact-set reductions relative to DIVER. Nevertheless, the overall results indicate that, for the majority of these subjects, statement coverage contributed far more greatly to the effectiveness of DDIA than dynamic points-to sets. Consistently, the dynamic alias analysis had stronger effects when applied at method-instance level than done at method level, and in fact the latter contributed the least to the DDIA effectiveness in any case, according to the generally much smaller effect sizes from the statistical tests where the two techniques differ only in the use of method-level dynamic aliasing data, in comparison to other tests.

The bottom row of Table XI lists the combined p-values for all subjects per test, which demonstrated the same contrasts among various forms of dynamic data as recognized above. In all, any of the new DDIA techniques instantiated from our DIAPRO framework was significantly more effective than DIVER and PI/EAS, and the addition of any of these dynamic data to the framework led to further reductions in the impact-set sizes of DIAPRO significantly. Moreover, the effects of statement coverage were significantly stronger than method-instance-level dynamic points-to sets, which had constantly stronger effects than such sets at method level also significantly. On the other hand, comparing the hybrid DDIA techniques as a whole to the baseline analysis revealed that the static dependence information had always stronger effects than any form of the additional dynamic data beyond method-event traces, with either a single form applied or multiple forms combined.

7.3.2. Repository Queries. Table XII shows the results from an exhaustive set of statistic analyses on repository queries with respect to the three SVN repositories, with each test and effect-size computation dealing with one of the same 16 pairs (comparisons) of DDIA techniques as those

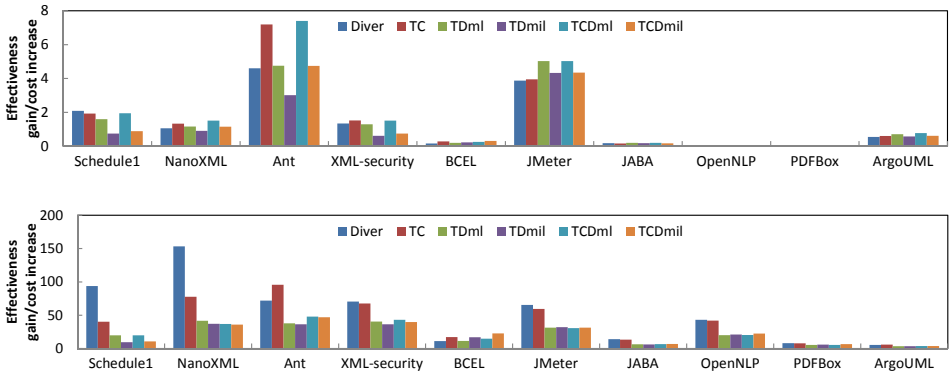

Fig. 7: Cost-effectiveness of DIVER and the new DIAPRO instances expressed as the ratios of their mean effectiveness gain to the factor of increase in the average post-processing cost (top) and total cost of the first two phases (bottom), both against PI/EAS on arbitrary single-method queries. The higher the ratio, the better (more cost-effective).

on the arbitrary queries above. Each row of the table shows the statistical significance out of one test for all subjects individually and combined, where each column gives the results of all tests for one individual subject and over all subjects; effect sizes are listed next to the p-values from the same pair of groups. In comparison to Table XI, the p values are noticeably smaller, which is consistent with the magnitude of effectiveness improvements being smaller on repository queries than on arbitrary queries as shown before. Nevertheless, the effect sizes and overall significance values indicate that impact sets of the studied repository queries between the two techniques in any of the 16 pairs were significantly different in statistical sense, signaling the significant effects on DDIA effectiveness of every single form of dynamic data working alone and of multiple forms of dynamic data collaborating together (for any of the combinations we considered).

As previously found on arbitrary queries, the largest significance values (and greatest effect sizes also) for differences between PI/EAS and DIVER indicate that static dependencies had again the strongest effects among all types of program information considered in our framework for any individual subject, as can be seen also from the combined metrics over all the three subjects. On the other hand, comparing the effect sizes among these hybrid DDIA techniques manifested general contrasts similar to those observed from the tests on arbitrary queries: Statement coverage had stronger effects than dynamic aliasing data, and dynamic alias analysis had apparently stronger effects when applied at method-instance level than at method level, for both any subject individually and over all subjects. The effect sizes further consolidate these observations and support the same conclusions. Note that the absolute values of effect sizes on these repository queries are generally larger than those on arbitrary ones, mainly as a result of the much smaller sample sizes in the former (which is the number of repository revisions) for each pair of compared groups.

7.4. RQ4: Variety of Cost-Effectiveness of DIAPRO

7.4.1. Arbitrary Queries. Figure 7 puts the separate metrics of effectiveness and cost together to show the cost-effectiveness as one single metric of the new DIAPRO instances versus DIVER. In both plots, the y-axis indicates the percentage of effectiveness gains (for an impact-set size ratio of r, the potential precision gain is $(1-r)/r$) divided by the factor of cost increases, of each of the six DDIA 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 top and bottom plot, respectively. We do not consider the storage costs in this regard as those costs are all marginal, almost negligible relative to today's storage resources, and they are all one-time cost.

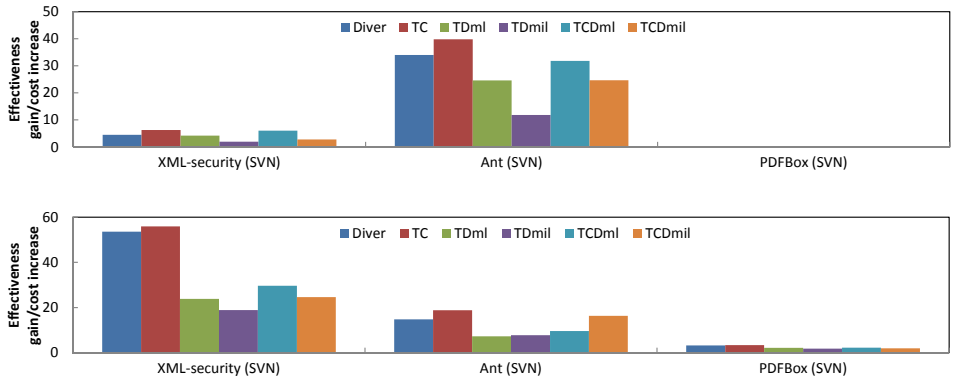

Fig. 8: Cost-effectiveness of DIVER and the new DIAPRO instances expressed as the ratios of their mean effectiveness gain to the factor of increase in the average post-processing cost (top) and total cost of the first two phases (bottom), both against PI/EAS on repository queries. The higher the ratio, the better (more cost-effective).

Also, we report the results on single-method queries only but not those on multiple-method ones for the entire category of arbitrary queries because the foregoing experiments have demonstrated that both the effectiveness and cost of these DDIA techniques were quite stable with varying query sizes. To obtain these cost-effectiveness metrics, the percentages of effectiveness gains were calculated from per-subject mean effectiveness shown in Table III; the factors of cost increase were derived from the mean post-processing time of Table VI (for the top plot) and the sum of static-analysis and runtime costs in Table VIII (for the bottom plot).

When considering the post-processing cost only, TCD_{ml} appeared to be the most cost-effective DIAPRO instance consistently for any subject, mainly because this technique combines the benefits of two forms of relatively cheap data: statement coverage and method-level dynamic points-to sets. In comparison, TD_{mil} had the lowest cost-effectiveness for all subjects but JMeter. Although TCD_{mil}, which uses the largest amount and finest form of dynamic data, was constantly the most effective DDIA instance among the six we compared, the top plot shows that its cost-effectiveness was among the lowest in most cases, suggesting that the effectiveness gains achieved by using the method-instance-level aliasing data did not well justify the extra overheads incurred by processing this data as using other forms of program information. For the majority of these subjects, the second most cost-effective technique was *TC*, which was anticipated from the foregoing study results where statement coverage mostly brought substantial effectiveness improvements at low costs.

As far as only the one-time (static-analysis and runtime) costs are concerned, however, DIVER had the best cost-effectiveness for all subjects but Ant and BCEL, for which *TC* and TCD_{mil} were the best, respectively. *TC* was also almost as cost-effective as DIVER for most subjects: In fact, DIVER was noticeably more cost-effective only for the two smallest subjects Schedule1 and NanoXML. Also except for these two subjects, at least one instance of DIAPRO had the cost-effectiveness either higher than or very close to DIVER. Both variants of *TD* were mostly the least cost-effective DDIA instances, primarily owing to the overly large overheads of collecting and using dynamic points-to sets relative to the effectiveness gains that DIAPRO obtained by using this data alone.

7.4.2. Repository Queries. The cost-effectiveness measurements of the new DIAPRO instances versus DIVER on repository queries are delineated in Figure 8 in the same format as Figure 7. When only the post-processing cost is concerned, TCD_{ml} was the most-effective analysis for XML-security but not as cost-effective as DIVER and *TC* for Ant. For PDFBox, all these techniques had almost identical cost-effectiveness, just as observed in the cases of arbitrary queries before,

which implies that although DIAPRO was not more cost-effective than DIVER for this particular subject, it still provided higher effectiveness that well paid off the extra overheads it incurred. Overall, for any of these three subjects, at least one DIAPRO instance was able to offer a more cost-effective option than DIVER, although TD_{mil} was found again the least cost-effective technique out of the six consistently for all subjects. On the other hand, *TC* seemed to be the best option for any subject as it attained by far the highest cost-effectiveness for Ant while for the other two subjects it was very close to the best.

Concerning the ont-time cost, *TC* achieved appreciably the greatest cost-effectiveness consistently for every individual subject, and only TCD_{mil} was close to it with Ant. Both variants of *TD* were constantly the least cost-effective instances, reaffirming again that the overheads of using dynamic points-to sets were relatively hard to be well paid off in contrast to statement coverage. Beyond DIVER and *TC*, both variants of *TCD* were reasonably close to DIVER in cost-effectiveness, implying that they could be better options over DIVER for tasks where higher precision of impact sets is desired since the additional costs and extra effectiveness benefits were well balanced.

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, implying that DIAPRO allows users to choose different best options for varying needs. For example, if developers are mainly concerned about longer static analysis and/or higher runtime overheads but willing to bear less precise impact computation, they would probably like to choose *TC* over other options or even not bother applying additional dynamic data such as statement coverage and dynamic points-to sets here but rather stick to DIVER. In contrast, developers who readily afford the one-time cost but are less tolerant for possibly long querying time may find that the effectiveness gain given by statement coverage and method-level dynamic alias analysis well justifies the extra costs incurred by using these additional data, and would pick TCD_{ml} as the best option. In all, our framework as a whole can indeed provide not only more precise and more cost-effective results than existing options but also a variety of cost-effectiveness choices for developers.

7.5. Summary of All Results

Out of the extensive results of our empirical studies, we highlight the major findings as follows:

— Exploiting static program dependencies in collaboration with various forms of dynamic program information, our hybrid approaches to dynamic impact prediction drastically pruned false-positive method-level impacts from the impact sets of approaches purely based on method-execution order like PI/EAS, by 65–72% on overall average.
— Utilizing additional forms of dynamic data including statement coverage and dynamic points-to sets beyond method-execution event traces, our three new DDIA techniques accomplished further improvements in the effectiveness of impact prediction beyond DIVER by up to 22%, which could mean substantial savings of inspection efforts given the usually large impact sets.
— Assuming arbitrary changes in methods, multiple-method queries received as effective impact sets as single-method queries did from any of our hybrid DDIA techniques (for the ten query sizes we studied at least), implying that the effectiveness of our framework is highly stable with respect to varying query sizes.
— Our framework appeared to be more effective on arbitrary queries than on repository queries in terms of absolute mean effectiveness numbers relative to the baseline, yet the improvements of DIAPRO over DIVER were generally consistent between the two categories of queries.
— Overall, adding any form of dynamic information to our framework resulted in statistically significant gains in DDIA effectiveness, with statement coverage generally having much stronger effects on the effectiveness than dynamic points-to sets, which were more impactful when applied at method-instance level than at method level though.
— Dynamic alias analysis at method-instance level was often much more expensive than it at method level and statement coverage, and also much less rewarding than the latter two with respect to the

additional effectiveness benefits it brought; in fact, adding dynamic aliasing data alone, especially at method level, to our DDIA framework was mostly quite ineffective.
— In contributing to the overall DDIA effectiveness, the static dependencies played a stronger role than either any single form of dynamic information used alone or multiple forms combined.
— The time cost of any of our DDIA approaches was reasonably acceptable for practical use, especially with respect to the effectiveness gains hence inspection-effort savings they attained; the storage costs of these approaches were negligibly small relative to commodity configurations.
— DIAPRO, with at least one of its instances, was generally more cost-effective than DIVER and different instances provided the best cost-effectiveness for different programs and for varying emphasis on different parts of the overall analysis overheads.
— Unifying both the two representative existing DDIA techniques (DIVER and PI/EAS) and the three new DIAPRO instances, including the two variants of *TD* and *TCD*, our framework as a whole provided a variety of cost-effectiveness options for various task scenarios in which developers have different requirements and preferences for dynamic impact analysis.

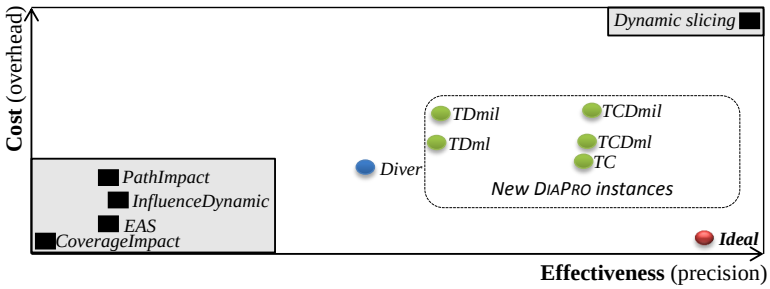

Fig. 9: An updated schematic illustration of tradeoffs between the cost and effectiveness dimensions in the DDIA design space in the same format as Figure 1 but with the new DDIA instances approximately situated in the currently untapped band.

In addition, putting the above results together can now help fill, with the new DDIA instances, the untapped region in the cost-effectiveness design space of DDIA as first depicted in Figure 1. Overall, considering the aggregate cost from the three phases, *TC* is the closest to the ideal, immediately followed by TCD_{ml} and then by TCD_{mil}; the two variants of *TD* achieve slightly higher effectiveness at much larger cost than DIVER, making them the lowest tier of cost-effectiveness with TD_{mil} being the farthest to the ideal tradeoff. Figure 9 gives an approximation of how the existing and newly proposed DDIA techniques compare in cost and effectiveness.

While these results were obtained from our particular experimental settings (including the limited number and scope of subject applications), the implications of our findings were not necessarily limited to the settings in this work. Also, these findings can inform developers regarding which techniques to choose as per their effectiveness expectations and time budgets, and can provide valuable guidelines to researchers for development of future impact-analysis techniques.

8. DISCUSSION

In this section, we briefly discuss main additional issues concerning both our study results and the technical approaches we proposed in general.

8.1. Comparing between Arbitrary and Repository Queries

First, we presented the empirical results around the four research questions on arbitrary and repository queries separately. The motivation to include and distinguish these two different types of queries was to comprehensively validate the merits of our approach with two complementary studies: The study on arbitrary queries focused on the scope and thoroughness, attempting to cover all possible queries of a program to avoid potential biases of sampling a subset of the program, yet

the results may not be representative of real-world use scenarios of impact analysis since in practice developers might query the impacts of some methods more likely than those of others; the study on repository queries focused on realistic usage scenarios, attempting to investigate impact sets that developers actually need, yet the results can be biased toward the chosen range of program revisions since developers may work on some particular sets of methods during specific period of time but on different ones during other periods. Importantly, for the latter study, we assumed implicitly that developers indeed seek impact sets of methods they change between program revisions as queries.

Moreover, in absolute terms, there have been noticeable differences in both effectiveness and cost as separate metrics and cost-effectiveness as an overall metric between the two categories of queries. And in all DIAPRO performed better on arbitrary queries than on repository ones. As we discussed before, the main reason for the difference in effectiveness might lie in the different constructs of the two groups of results: Compared to the repository group, the arbitrary group contained much more trivial queries which largely favored DIAPRO against the baseline. The reason for the difference in cost is most likely to consist in the appreciably larger subject (and/or input-set) sizes and the tremendously larger query sizes in the repository group than in the arbitrary group: Over 50 and even up to 500 (versus at most 10) concurrent query processing caused substantial extra overheads which apparently raised up the average post-processing cost; the largely grown subjects and inputs ended up with longer static analysis and more runtime overheads. In addition, recall that subjects in the repository group were chosen with an attempt to estimate the lower bound of the effectiveness of our techniques. Although the validity of the results on repository queries being indicative of the actual lower bound, even within the ten-subject pool, may have been affected by the changes in subject and input-set sizes, it should be reasonable for developers to expect at least the level of performance that lies between those on the two groups of queries in real-world evolution tasks. Finally, as stressed above, the motivation of including both categories of queries is to compare different DDIA techniques comprehensively rather than quantifying the performance of each technique in absolute terms.

8.2. Overcoming Possible Efficiency Hurdles

According to our extensive studies and detailed result analysis, the overall efficiency of our framework did not seem to be a barrier for its practical adoption. Yet, the exceedingly larger overheads DIAPRO incurred on certain subjects or queries than on others should be treated as a warning about the scalability of our techniques. To overcome such challenges, several improvements in regards to efficiency may be necessary. First, in the cases of impeding static-analysis time cost, we may abstract the dependence model from the current statement level to coarser (e.g., method) levels [Myers 1981; Loyall and Mathisen 1993]. In this regard, the method dependence graph (MDG) [Cai and Santelices 2015a] we recently developed could be utilized in place of the DIVER dependence graph in our framework. The MDG directly models data and control dependencies among methods with intraprocedural dependencies in PDGs abstracted away, which would also speed up the post-processing phase as a result of fewer dependencies to traverse during the iterative process of impact computation. Adopting the MDG in our framework, though, would need considerable implementation efforts in addition to empirical studies examining its influence on the soundness and effectiveness of the framework.

Second, in the cases of impeding post-processing cost for multiple-method queries that can not be effectively mitigated by querying parallelization (e.g., via multithreading), one solution would be to first compute and keep the impact sets of all single-method queries and then obtain the impact sets of multiple-method queries by simply taking the union of precomputed impact sets of member single-method queries— the overheads of such offline union operations are generally marginal as per our experience. Alternatively, impact sets can be computed online thus the overheads of tracing and processing of possibly long traces in the current post-processing phase would be eliminated, hence potentially large gains in querying efficiency. This latter solution can help address long post-processing time for single-method queries as well. Besides, it is worth noting that in the experiments of this work we used a commodity machine with relatively low CPU and memory

configurations that we have access to. Also, the current implementation of our framework is still a research prototype, which has not been tuned yet. Thus, even higher efficiency of DIAPRO may be expected from an optimized implementation running on a better-configured hardware platform.

8.3. Choosing the Best Technique

As suggested by our empirical results and related discussion, the cost and effectiveness of DDIA can be affected by a range of factors including the size of runtime inputs, variety of execution data, use of static information, query choice and composition, and various characteristics (such as source size, inter-component couplings, and application domain) of the subject program itself.

Also, when comparing the cost-effectiveness as a holistic measure of performance among different DDIA techniques, we found that different best options stood out for different subjects, and the best choice changed with varying emphases on different parts of the overall cost of DDIA. However, this judgement is subject to the assumption that developers would always take the most cost-effective analysis as the *best* option. In practice, though, it is also possible that developers would choose a DDIA technique preferably based on its effectiveness or efficiency alone even if it may not be the most cost-effective among all options. As an example, they may opt for better precision anyway no matter whether the added costs are best paid off— for instance, they could even choose the most expensive technique TCD_{mil} for its highest precision against all other DDIA techniques our framework offers, if the relatively highest cost remains affordable to them. For these developers, DIAPRO does provide more technical options than existing alternatives. On the other hand, for particular application needs, developers may go for the fastest technique despite its lower effectiveness than other techniques available: For example, to get a quick high-level picture of inter-module interactions in a complex system, a rough but fast analysis may indeed be more desirable than those that are more precise and cost-effective but slower. Note that these different requirements and preferences with impact-analysis techniques further necessitates a framework like DIAPRO that provides not just techniques of best cost-effectiveness in various scenarios but also those that may not be the most cost-effective in any task but are still in demand sometimes.

As we mentioned earlier, the impact sets give by DIAPRO are all safe relative to the execution set utilized by the analysis. Further, since DIVER, *TC*, *TD*, and *TCD* 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 DIVER, and similar inclusion relations hold for *TD* compared to DIVER, *TCD* to *TC*, and *TCD* to *TD*. At the same time, such incremental precision gains come with growing overheads in general. Accordingly, developers are suggested to adopt the DIAPRO instance that best fits their effectiveness need, time and storage budget, and availability of program information. Collectively, the diverse set of DDIA techniques we offer in a unified framework fills a large untapped area in the cost-effectiveness design space of DDIA as shown in Figure 1, potentially making dynamic impact prediction more attractive to developers for practical adoption.

9. RELATED WORK

Previous work related to ours can be summarized into the following three major categories: code-based impact prediction, hybrid DDIA, dynamic dependence analysis, and utilization of various dynamic program information.

9.1. Code-based Impact Prediction

The baseline PI/EAS comes from the method-level DDIA introduced by Law and Rothermel [Law and Rothermel 2003b] and its performance optimization EAS [Apiwattanapong et al. 2005] by Apiwattanapong and colleagues. Our DDIA techniques all utilize the method-execution events as by PI/EAS as the common form of dynamic data to quickly filter methods that cannot be impacted in the given executions (i.e., methods that never executed after the query). Unlike PI/EAS that relied solely on the method execution order, however, DIAPRO leverages static program dependencies in addition to the method events and other types of dynamic information to significantly improve the effectiveness of impact prediction against PI/EAS by pruning interprocedural dependencies

that are not exercised by the runtime data. The impact analysis using *static-execute-after* (SEA) relations [Jász et al. 2008] also exploits method execution order but is based on static call graph [Badri et al. 2005] instead of runtime traces towards a static approach to impact prediction.

A few *online* DDIA techniques [Breech et al. 2004; Breech et al. 2005] have been proposed to avoid tracing hence achieve higher querying efficiency than PI/EAS. As we mentioned earlier, DIAPRO could be implemented as online as well to deal with possible efficiency issues. Yet, typically online DDIA is limited to produce impact sets for a single execution only, hence multiple runs of the analysis would be required for obtaining impact sets with respect to multiple executions (e.g., from a test suite as commonly needed). Moreover, queries must be given before the runtime, thus obtaining impacts for different queries also necessitates rerunning the analysis—computing impact sets for functions of the entire program all at once is feasible [Breech et al. 2005] but would incur excessive runtime slowdown and memory consumption without relying on compiler extensions [Breech et al. 2004]. Also, these online techniques aim at performance gains over EAS without addressing its great imprecision as the major problem it suffers from. Therefore, we did not compare DIAPRO against any of such online DDIA techniques.

Various program information other than method-execution order has been exploited for predicting code-based impacts [Bohner and Arnold 1996; Li et al. 2013b], including static slicing [Acharya and Robinson 2011], coupling measures [Poshyvanyk et al. 2009; Kagdi et al. 2010], change-type classification [Sun et al. 2010; Ren et al. 2004], and concept lattices [Tonella 2003; Li et al. 2013a]. JRipple [Buckner et al. 2005; Petrenko and Rajlich 2009] attempts to improve impact-analysis precision by providing multiple levels of granularity of impact sets (e.g., class, method, and code-segment levels), which is similar to our approach in using code information only. However, like SEA, these are all *static* impact analyses which are more conservative (considering all possible program inputs) thus less precise than dynamic approaches. Although our framework includes a static-analysis phase where the static dependence graph is created, it leverages the dependence information in synergy with various dynamic data to predict impacts more effectively than using the static information alone. The COVERAGEIMPACT technique presented by Orso and colleagues [Orso et al. 2003] computes impact sets also using forward slicing, yet was shown to be even less precise than PI/EAS [Orso et al. 2004]. Other software artifacts than program code, such as code and change-request repositories [Canfora and Cerulo 2005] and version histories [Zimmermann et al. 2005], to predict change impacts and suggest future changes, different from our approach focusing on analyzing the program code itself.

9.2. Hybrid Approaches to DDIA

DDIA approaches employing both static and dynamic information (i.e., hybrid techniques) have also been explored before, such as INFLUENCEDYNAMIC [Breech et al. 2006] and its extension in [Huang and Song 2007]. While these techniques improve PI/EAS in terms of analysis precision, they model partial dependencies and exploit a single type of dynamic data only (i.e., the method event trace). And previous studies have shown that none of them achieved significant precision gains over PI/EAS [Breech et al. 2006; Huang and Song 2007]. SD-IMPALA [Maia et al. 2010] also follows a hybrid approach to DDIA, yet it focuses on improving the recall of DDIA and does that at the cost of penalizing precision. The technique uses call graphs as the static dependence model and is shown even less precise than its predecessor Impala [Hattori et al. 2008] and PI/EAS. In contrast, DIAPRO is built on a complete (albeit conservative) dependence model to accomplish significantly better precision not only than PI/EAS but even further beyond the latest peer approach DIVER by using diverse dynamic information. It is difficult to include INFLUENCEDYNAMIC in our study as its design is constrained to procedural languages such as C while our current DIAPRO implementation targets object-oriented software, and its environment (e.g., GCC with customization) is different from that of DIAPRO too. Our framework is also different from the integrated impact analysis presented by Gethers et al. [Gethers et al. 2012] in that all the program information utilized by DIAPRO are generated from the program and its input set under analysis, relying on no other external data such as repository history and change requests as required by the other approach.

Our DDIA framework reused a few components of DIVER [Cai and Santelices 2014], including the DIVER dependence graph and method event tracing. In contrast, this paper focuses on the unified DDIA framework by which we provide DDIA techniques of multiple levels of cost-effectiveness tradeoffs, and studies the effects of two additional types of dynamic data on that tradeoff. More importantly, the framework subsumes and generalizes hybrid DDIA where DIVER is one specific instance. This paper also supersedes its preliminary version in [Cai and Santelices 2015c] through both technical and empirical extensions substantially. First, the framework is presented in much greater detail including the process of dynamic data collection and, more importantly, the unified impact-computation algorithm, and with clearer motivation from the perspective of cost-effectiveness dimensions in the DDIA design space. Second, one more DDIA technique *TD*, including the two variants TD_{ml} and TD_{mil}, is added as another instance of the framework; and a more in-depth study on the effects of dynamic aliasing data on DDIA cost-effectiveness is enabled by the addition of this new instance to the empirical studies. Third, studies on arbitrary queries consisting of multiple methods and on a different category of queries, those based on real-world repository changes, are added with three more large subjects, and the analysis and discussion on the study results are largely expanded as well. In all, this paper presents our framework more completely and elaborately and evaluates it much more thoroughly than its previous version.

9.3. Dynamic Dependence Analysis

In general, DDIA can be regarded a particular form of forward dynamic dependence analysis and its resulting impact set of a given query corresponds to the forward dependence set of that query, albeit it commonly reports the dependence set at method level on top of a dependence model lighter than the traditional fine-grained dependencies based on the SDG [Horwitz et al. 1990]. The DDIA framework we proposed effectively computes a method-level forward dynamic dependence set as well, using an approximate static dependence graph with light dynamic information: In essence, given a query m, our DDIA techniques compute as impacts all methods that *may* (conservatively) dynamically dependent on m with respect to the concrete executions analyzed.

Previously, a few other approaches were proposed to explicitly represent program dependencies at method level as well [Loyall and Mathisen 1993; Myers 1981; Cai and Santelices 2015a]. Yet, unlike DIAPRO which targets dependencies among methods for specific set of executions, those approaches addressed only static dependencies that hold for any possible program executions. Forward dynamic slicing [Kamkar 1995; Cai and Santelices 2015d] could work as the finest-grained (statement-level) DDIA, yet in theory it would be overly expensive for a method-level impact analysis [Apiwattanapong et al. 2005; Law and Rothermel 2003a] 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 (although even more). While we assume that dynamic slicing would incur much higher cost than DIAPRO, it may still be worth comparing them empirically, especially on their cost-effectiveness, for the purpose of DDIA. In that regard, variants of dynamic slicing such as relevant slicing [Agrawal et al. 1993] and quasi slicing [Venkatesh 1991] will be of interest as well as they provide different levels of efficiency, precision, and/or recall. In fact, the cost-effectiveness tradeoffs of dynamic slicing have been studied in [Zhang and Gupta 2004], where different such tradeoffs are achieved through various algorithmic designs. In contrast, we investigate the cost-effectiveness tradeoffs of *method-level* DDIA realized via different configurations of static and dynamic program information.

9.4. Utilization of Execution Data

Statement coverage has been used in regression testing [Elbaum et al. 2002], test generation [Korel 1990], and in general as the quality metric of a test-suite [Weiser et al. 1985], but not yet been used 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 [Orso et al. 2003], which is much less precise than PI/EAS [Orso et al. 2004],

though. We used statement-level coverage information in *TC* and *TCD* to improve the effectiveness of predictive DDIA and showed that using statement coverage can greatly contribute to both the precision and the overall 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 [Mock et al. 2005]. 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 instances of DIAPRO, *TD* and *TCD*, we exploited dynamic points-to data too but for method-level 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.

10. CONCLUSION

In this article, we presented a framework DIAPRO that unifies a diverse set of DDIA techniques, hybrid approaches in particular, including two representative analyses previously proposed and three new hybrid techniques. Exploiting both static program dependencies and various dynamic data, including method execution trace, statement coverage, and dynamic points-to sets, DIAPRO aims to make predictive dynamic impact analysis more useful in software evolution practice and in a broader scope of dependence-based tasks as clients of impact analysis. While a rich body of previous research produced various techniques for dynamic impact prediction, they mostly tend to suffer from similar challenges in balancing cost and effectiveness: They are either efficient but too imprecise or more precise but too expensive, occupying two extreme (corner) areas in the cost-effectiveness design space of DDIA. In this context, the main goal of our framework is to fill the gap between these two extremes by providing a range of techniques that are not only cost-effective for practical use but also offer a variety of levels of cost-effectiveness tradeoffs. Through this framework, we also aim to examine the effects of various forms of program information, both static and dynamic, on the two dimensions of the DDIA design space, so as to inform about development of more advanced technical supports for future needs in software evolution and beyond.

Empirical results from extensive experiments on the DIAPRO framework demonstrated that, by combining multiple forms of program information, hybrid techniques can significantly upgrade the effectiveness of DDIA relative to purely dynamic approaches without losing the safety of results while remaining reasonably efficient. Although additional data generally also adds to the overhead of DDIA when giving more precise analysis results, the ultimate cost-effectiveness can be raised to higher levels by using some forms of program information over others. For instance, we have seen consistently that statement coverage is a much more cost-effective form of dynamic data than dynamic points-to sets in general, and static dependencies appear to contribute a lot more to the effectiveness of DDIA than any form of dynamic information. Further, in certain cases, both forms of dynamic data can achieve significantly better improvements in precision and cost-effectiveness when working together than either working alone. The finer-grained (method-instance-level) aliasing data, however, mostly incurred extra overheads that were not as well paid off as the other forms of data. Finally, our studies suggest that using arbitrary queries only may lead to overestimated performance of DDIA, and a comprehensive evaluation of a DDIA technique should also consider more realistic queries such as those based on repository changes in real-world software evolution.

In all, our approach generalizes various approaches to predictive DDIA while offering three new DDIA techniques that are more effective than existing alternatives all at reasonable costs. In addition, we showed that since different instances of our framework give the best cost-effectiveness in different budget situations or application contexts, the framework as a whole provides flexible options to meet various user needs for impact analysis. More broadly, since it essentially computes

forward dynamic dependencies at method level, our DDIA framework is potentially able to support a wide range of dependence-based tasks with multiple levels of cost-effectiveness options.

Several lines of future research would be worth exploring based on the presented framework. First, as we discussed earlier, recall is an integral part in the effectiveness of impact analysis in general, including predictive DDIA. While the recall of the current DIAPRO is identical to that of the baseline approach PI/EAS which is safe relative to the inputs to the analysis, measuring recall relative to actual impact sets of concrete changes using software repository mining techniques [Hassan and Holt 2006; Zimmermann et al. 2005] would complement our previous approach in that regard [Cai and Santelices 2015b] to provide additional insights into the cost-effectiveness of DDIA. Furthermore, given the imperfect recall with respect to concrete changes of existing DDIA techniques including ours, focusing on various potential improvements in recall [Hattori et al. 2008] of DDIA would be a rewarding direction complementary to our focus on precision in this work. Finally, the proposed DDIA framework can be further extended to support beyond *code-based* impact analyses [Li et al. 2013b], such as integrated approaches [Gethers et al. 2012] that incorporate code analysis, data mining [Canfora and Cerulo 2005], and information retrieval [Poshyvanyk et al. 2009] to exploit a larger variety of software artifacts (e.g., repository commits, running logs, developer information, and code documentation) than we presently considered.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their valuable comments that helped improve this paper.

REFERENCES

Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 746–765.

Hiralal Agrawal, Joseph R Horgan, Edward W Krauser, and Saul London. 1993. Incremental regression testing. In *Proceedings of IEEE International Conference on Software Maintenance*. 348–357.

Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques and Tools*.

M Ajrnal Chaumun, Hind Kabaili, Rudolf K Keller, and François Lustman. 1999. A change impact model for changeability assessment in object-oriented software systems. In *Proceedings of European Conference on Software Maintainance and Reengineering*. 130–138.

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2005. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 432–441.

Linda Badri, Mourad Badri, and Daniel St-Yves. 2005. Supporting predictive change impact analysis: a control call graph based technique. In *Proceedings of Asia-Pacific Software Engineering Conference*. 167–175.

Shawn A. Bohner and Robert S. Arnold. 1996. *An introduction to software change impact analysis*. In Software Change Impact Analysis, Bohner & Arnold, Eds. IEEE Computer Society Press, pp. 1–26.

B. Breech, A. Danalis, Stacey Shindo, and Lori Pollock. 2004. Online impact analysis via dynamic compilation technology. In *Proceedings of IEEE International Conference on Software Maintenance*. 453–457.

B. Breech, M. Tegtmeyer, and L. Pollock. 2005. A comparison of online and dynamic impact analysis algorithms. In *Proceedings of European Conference on Software Maintainance and Reengineering*. 143–152.

B. Breech, M. Tegtmeyer, and L. Pollock. 2006. Integrating influence mechanisms into impact analysis for increased precision. In *Proceedings of IEEE International Conference on Software Maintenance*. 55–65.

Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Vaclav Rajlich. 2005. JRipples: a tool for program comprehension during incremental change.. In *Proceedings of IEEE International Workshop on Program Comprehension*. 149–152.

Haipeng Cai and Raul Santelices. 2014. Diver: precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 343–348.

Haipeng Cai and Raul Santelices. 2015a. Abstracting program dependencies using the method dependence graph. In *Proceedings of IEEE International Conference on Software Quality, Reliability, and Security*. 49–58.

Haipeng Cai and Raul Santelices. 2015b. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *Journal of Systems and Software* 103 (2015), 248–265.

Haipeng Cai and Raul Santelices. 2015c. A framework for cost-effective dependence-based dynamic impact analysis. In *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering*. 231–240.

Haipeng Cai and Raul Santelices. 2015d. TracerJD: generic trace-based dynamic dependence analysis with fine-grained logging. In *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering*. 489–493.

Haipeng Cai, Raul Santelices, and Tianyu Xu. 2014. Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis. In *Proceedings of IEEE International Conference on Software Security and Reliability*. 48–57.

Gerardo Canfora and Luigi Cerulo. 2005. Impact analysis by mining software and change request repositories. In *Proceedings of IEEE International Symposium on Software Metrics*. 20–29.

Norman Cliff. 1996. *Ordinal methods for behavioral data analysis*. Psychology Press.

Cleidson RB de Souza and David F Redmiles. 2008. An empirical study of software developers' management of dependencies and changes. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 241–250.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 28, 2 (2002), 159–182.

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

Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. 2012. Integrated impact analysis for managing software changes. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 430–440.

Ahmed E Hassan and Richard C Holt. 2006. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering* 11, 3 (2006), 335–367.

L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio. 2008. On the precision and accuracy of impact analysis techniques. In *International Conference on Computer and Information Science*. 513–518.

Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12, 1 (1990), 26–60.

Lulu Huang and Yeong-Tae Song. 2007. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *Proceedings of International Conference on Software Engineering Research, Management and Applications*. 374–384.

Daniel Jackson and Martin Rinard. 2000. Software analysis: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 133–145.

Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. 2008. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of IEEE International Conference on Software Maintenance*. 137–146.

Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L Collard. 2010. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proceedings of IEEE Working Conference on Reverse Engineering*. 119–128.

Mariam Kamkar. 1995. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software* 31, 3 (1995), 197–214.

Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot - a Java bytecode optimization framework. In *Proceedings of Cetus Users and Compiler Infrastructure Workshop*. 1–11.

James Law and Gregg Rothermel. 2003a. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of IEEE International Symposium on Software Reliability Engineering*. 430–441.

James Law and Gregg Rothermel. 2003b. Whole program Path-Based dynamic impact analysis. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 308–318.

Bixin Li, Xiaobing Sun, and Jacky Keung. 2013a. FCA–CIA: An approach of using FCA to support cross-level change impact analysis for object oriented Java programs. *Information and Software Technology* 55, 8 (2013), 1437–1449.

Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013b. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability* 23, 8 (2013), 613–646.

Li Li and A Jefferson Offutt. 1996. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of IEEE International Conference on Software Maintenance*. 171–184.

Joseph P Loyall and Susan A Mathisen. 1993. Using dependence analysis to support the software maintenance process. In *Proceedings of IEEE International Conference on Software Maintenance*. 282–291.

Mirna Carelli Oliveira Maia, Roberto Almeida Bittencourt, Jorge Cesar Abrantes de Figueiredo, and Dalton Dario Serey Guerrero. 2010. The hybrid technique for object-oriented software change impact analysis. In *Proceedings of European Conference on Software Maintainance and Reengineering*. 252–255.

Markus Mock, Darren C Atkinson, Craig Chambers, and Susan J Eggers. 2005. Program slicing with dynamic points-to sets. *IEEE Transactions on Software Engineering* 31, 8 (2005), 657–678.

Frederick Mosteller and R. A. Fisher. 1948. Questions and Answers. *The American Statistician* 2, 5 (1948), 30–31.

Eugene M Myers. 1981. A precise inter-procedural data flow algorithm. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 219–230.

Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging field data for impact analysis and regression testing. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*. 128–137.

Alessandro Orso, Taweesup Apiwattanapong, James B. Law, Gregg Rothermel, and Mary Jean Harrold. 2004. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 491–500.

Maksym Petrenko and Václav Rajlich. 2009. Variable granularity for improving precision of impact analysis. In *Proceedings of IEEE International Conference on Program Comprehension*. 10–19.

Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering* 14, 1 (2009), 5–32.

Václav Rajlich. 2014. Software evolution and maintenance. In *Proceedings of the Conference on Future of Software Engineering*. 133–144.

Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. 432–448.

Per Rovegard, Lefteris Angelis, and Claes Wohlin. 2008. An empirical study on views of importance of change impact analysis issues. *IEEE Transactions on Software Engineering* 34, 4 (2008), 516–530.

Barbara G Ryder. 2003. Dimensions of precision in reference analysis of object-oriented programming languages. In *Compiler Construction*. 126–137.

Raul Santelices, Yiji Zhang, Haipeng Cai, and Siyuan Jiang. 2013. DUA-Forensics: a fine-grained dependence analysis and instrumentation framework based on Soot. In *Proceedings of ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*. 13–18.

Lajos Schrettner, Judit Jász, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. 2013. Impact analysis in the presence of dependence clusters using static execute after in WebKit. *Journal of Software: Evolution and Process* 26, 6 (2013), 569–588.

Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. 2010. Change impact analysis based on a taxonomy of change types. In *Proceedings of IEEE Computer Software and Applications Conference*. 373–382.

Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*. 51:1–51:11.

Paolo Tonella. 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering* 29, 6 (2003), 495–509.

G. A. Venkatesh. 1991. The semantic approach to program slicing. In *Proceedings of ACM Conference on Programming Language Design and Implementation*. 107–119.

Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying E. Ye. 2011. *Probability and Statistics for Engineers and Scientists*. Prentice Hall.

MD Weiser, John D Gannon, and Paul R McMullin. 1985. Comparison of structural test coverage metrics. *IEEE Software* 2, 2 (1985), 80–85.

Xiangyu Zhang and Rajiv Gupta. 2004. Cost effective dynamic program slicing. In *Proceedings of ACM Conference on Programming Language Design and Implementation*. 94–106.

Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.

Received June 2015; revised Jan 2016; accepted Feb 2016