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

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

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

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

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

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

I. Introduction

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

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

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

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

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

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

We implemented this framework and its three instances above for Java, and evaluated them using seven subjects of up to 220K lines of code in size.[1] Our results show that these new analyses are generally all cost-effective, with continuous and significant precision gains over the baseline approach PI/EAS at reasonable costs. In addition, among the three instances, *TC* attained the best cost-effectiveness in most cases as regards the overall analysis time overhead, which implies

[1]The download is publicly available at http://nd.edu/~hcai/diver.

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

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

The main contributions of this paper include:

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

II. MOTIVATION

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

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

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

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

III. BACKGROUND

This section presents necessary background with an example program used for illustration purposes. In Figure 1, program E inputs two integers `a` and `b` in its entry method `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* impact analysis [12] which assumes no knowledge about actual changes to programs. Such analysis takes a program P, a test suite T, and a set of methods M and outputs an *impact set* containing the methods in P potentially impacted by M when running T.

One example technique is PATHIMPACT [12], which collects runtime traces of executed methods. For each method m in M that is queried for its impacts, PATHIMPACT uses the method execution order found in the runtime traces of P for T. The analysis identifies as impacted m and all methods executed in any trace after m.

Figure 1 shows an example trace of E for PATHIMPACT (bottom left), where `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. 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.

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

INFLUENCEDYNAMIC [7] tries to improve the precision of EAS by considering method dependencies in addition to method traces. It models interface-level data dependencies via parameter passing and returns between methods, using an influence graph on which impacts are computed with impact propagation through intraprocedural dependencies ignored. For an example, given the same query set {M2}, example program

```
public class A {
    static int g; public int d;
    String M1(int f, int z) {
        int x = f + z, y = 2, h = 1;
        if (x > y)
            M2(x, y);
        int r = new B().M3(h, g);
        String s = "M3val: " + r;
        return s;}
    void M2(int m, int n) {
        A a2 = _trans(this);
        C.M5(a2);
        int w = m - d;
        if (w < 0)
            g = m / w;}}
```

```
public class B {
    static short t;
    int M3(int a, int b) {
        int j = 0;
        t = - 4;
        if ( a < b )
            j = b - a;
        return j;}
    static double M4() {
        int x = A.g, i = 5;
        try {
            i = x / (i + t);
            new A().M1(i, t);
        } catch(Exception e) { }
        return x;}}
```

```
public class C {
    static boolean M5(A q) {
        long y = q.d;
        boolean b = B.t > y;
        q.d = -2;
        return b;}
    public static void
      M0(String[] args) {
        int a = 0, b = 3;
        A o = new A();
        String s = o.M1(a, b);
        double d = B.M4();
        String u = s + d;
        System.out.print(u);}
}
```

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

IaPro: 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. 1: The example program E and its execution traces used by PathImpact and IaPro for illustration.

E, and execution trace as used above, the resulting impact set will be the entire program as PI/EAS produced.

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

IV. Technique

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

A. Overview

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

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

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

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

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

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

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

B. Trace Only (TO)

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

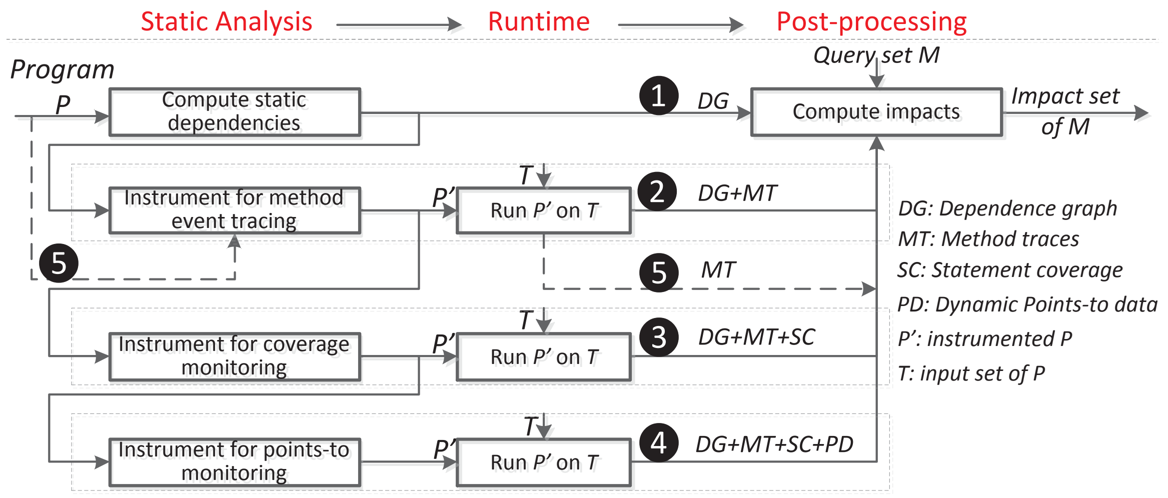

Fig. 2: A unified framework for DDIA that incorporates static dependence information and various forms of dynamic data to achieve multiple-level cost-effectiveness: The marked paths (circled 1 to 5) illustrate five of its instances we studied.

DIVER [16], *TO* finds methods from the trace directly or transitively dependent on the query using the dependence graph. Details about this technique are presented in [16], and here we recap only the key ideas as follows.

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

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

C. Trace plus Coverage (TC)

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

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

D. Full Combination (FC)

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

FC thus attempts to further improve the precision of DDIA by exploiting dynamic points-to data. This technique collects the full set of allocation sites that each pointer points to during program execution, by monitoring memory addresses of pointer targets. Then, during the post-processing phase, spurious aliasing-induced data dependencies can be identified by checking the intersection of relevant points-to sets. Given a data dependence $s1 \rightarrow s2$ in the dependence graph, for example, *FC* 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 collaboration with the method trace, which contains all method execution instances, dynamic points-to sets can be collected and employed at two granularity levels: method level and method-instance level. The method level strategy maintains a single points-to set of each heap variable (exercised by the runtime input) in a method m that contains allocation sites pointed to by that variable for all instances of m. In contrast, the method-instance level data includes such points-to sets for each instance of m separately.

Once spurious dependencies are identified, they are used for pruning false impacts from the method execution trace. The method level data can be applied either before or during impact computation, similar to the pre-pruning and post-pruning strategies described for *TC* above. However, only post-pruning can be adopted with the method-instance level data as the dependence graph is static and does not incorporate information about method execution instances.

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

E. Other Instantiations

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

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

F. Illustration

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

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

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

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

V. EVALUATION

This section presents our empirical evaluation of the DDIA framework we proposed, by studying the IAPRO techniques as its representative instances: *TO*, *TC*, and *FC* (both variants). Our main goal with this empirical study was twofold. First, we wanted to assess the precision of the new techniques and its practicality in terms of time and space costs against the existing most cost-effective DDIA PI/EAS as the baseline approach.[2] Second, we intended to analyze the effects of different types of dynamic data on the cost-effectiveness of DDIA in order to inform future design of better DDIA techniques.

A. Experiment Setup

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

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

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

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

[2]The *TO* technique was described before but only initially evaluated [16].

TABLE I: STATISTICS OF EXPERIMENTAL SUBJECTS

Subject	#LOC	#Tests	#Methods	#Queries
Schedule1	290	2,650	24	20
NanoXML	3,521	214	282	172
Ant	18,830	112	1,863	607
XML-security	22,361	92	1,928	632
JMeter	35,547	79	3,054	732
Jaba	37,919	70	3,332	1,129
ArgoUML	102,400	211	8,856	1,098

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

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

B. Experimental Methodology

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

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

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

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

For a further analysis of those effects, we intended to assess the statistical significance of the differences in precision among the IAPRO instances. To that end, we performed a set of paired Wilcoxon signed-rank tests [25] 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 normality of underlying data distribution. 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 [26].

C. Results and Analysis

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

Overall, *TO* greatly reduced the sizes of baseline impact sets, with *TC*, FC_{ml}, and FC_{mil} continuously improving the relative precision, albeit slightly. Also, except for Schedule1, there always existed cases in which IAPRO cut some off the entire impact set reported by the baseline analysis—IAPRO reports empty impact sets for queries with empty method body—leading to the minimal size ratios of 0%. And except for Ant and XML-security in addition to Schedule1, IAPRO was always able to prune some false impacts—The maximal ratios are constantly below 100% for other subjects.

The largest gains in precision were seen with Ant, XML-security and JMeter, for which IAPRO drastically prunes the baseline impacts by over 80%. ArgoUML also got a substantial impact-set reduction from IAPRO, which reported no more than 35% of the methods produced by the baseline. With Schedule1 and Jaba, IAPRO improved relatively less, probably because these programs have dense dependencies among their entities as a result of tight inter-function couplings.

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

The complementary results on precision shown in the left five columns of Table II further demonstrated the effects of the additional dynamic data beyond method traces, where the means of impact-set size ratios reveal that both *TC* and *FC* gained only little in precision on top of *TO*. Between the two forms of dynamic information, however, statement coverage appears to be more effective than dynamic points-to data. Consistent with the boxplots, the overall mean ratios, as listed in the bottom row of the table, show that relative advantage of statement coverage. Note that, as in other tables too, these *overall* numbers are not simple averages over the per-subject means listed in the table but weighted averages by the number of queries per subject as shown in the last column of Table I.

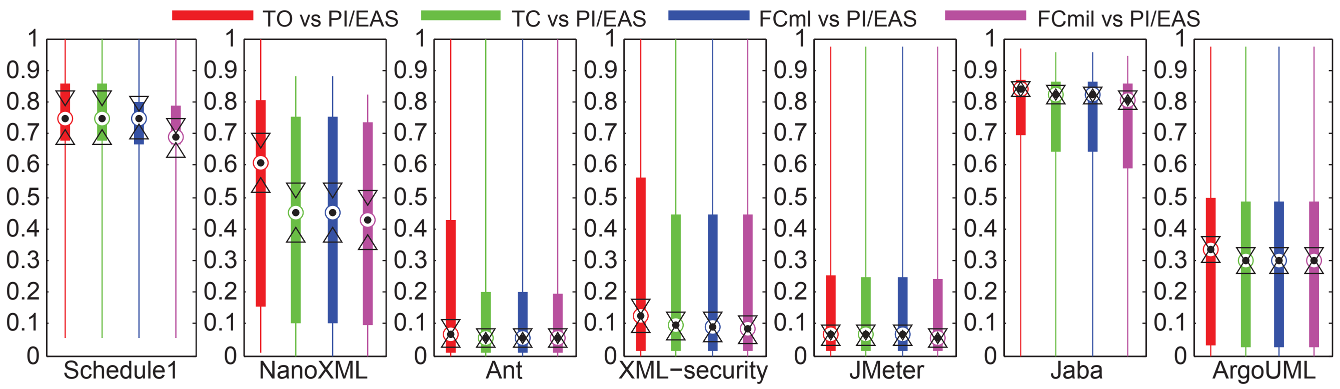

Fig. 3: Precision of the IAPRO techniques expressed as impact-set size ratios against PI/EAS (the lower the ratio, the better).

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

Subject	**Mean IS size ratio to PI/EAS (%)**				**Query time in seconds: mean (stdev)**				
	TO	*TC*	FC_{ml}	FC_{mil}	PI/EAS	*TO*	*TC*	FC_{ml}	FC_{mil}
Schedule1	71.3	71.3	66.6	65.1	0.7 (0.3)	14.6 (6.0)	15.7 (5.9)	19.2 (5.8)	44.3 (6.7)
NanoXML	51.7	45.6	45.5	43.5	0.1 (0.1)	6.2 (8.8)	6.4 (8.9)	5.6 (7.7)	7.9 (10.4)
Ant	25.7	17.2	17.2	16.9	0.1 (0.1)	3.2 (7.6)	3.4 (7.9)	3.3 (7.2)	5.2 (9.7)
XML-security	28.8	24.8	24.6	24.1	0.1 (0.1)	7.4 (9.6)	8.0 (10.4)	8.2 (10.5)	16.9 (20.9)
JMeter	18.8	18.2	18.1	17.6	0.1 (0.1)	2.3 (7.8)	2.3 (7.9)	1.8 (5.6)	2.2 (6.2)
Jaba	66.9	63.8	63.3	61.5	0.3 (0.2)	78.3 (82.5)	99.7 (102.5)	82.6 (83.0)	105.2 (99.7)
ArgoUML	31.5	29.4	29.2	29.2	0.1 (0.1)	15.9 (58.2)	15.9 (57.8)	12.6 (42.8)	15.8 (49.9)
Overall	**38.3**	**34.8**	**34.6**	**33.8**	**0.1 (0.2)**	**26.4 (60.0)**	**32.0 (72.1)**	**26.7 (57.9)**	**35.1 (70.8)**

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

Subject	**Prof.**	**Static analysis cost in seconds**				**Runtime cost in seconds**						**Execution data size in MB**				
		PI/EAS	*TO*	*TC*	*FC*	Normal	PI/EAS	*TO*	*TC*	FC_{ml}	FC_{mil}	PI/EAS	*TO*	*TC*	FC_{ml}	FC_{mil}
Schedule1	13	5	6	11	17	4	10	16	19	36	91	1.0	8.2	10.2	14.8	20.2
NanoXML	12	11	14	25	39	1	1	5	7	14	26	0.4	2.4	3.0	3.5	4.9
Ant	29	27	142	170	311	1	2	2	4	9	25	1.0	2.0	3.8	5.1	7.2
XML-security	37	33	158	190	280	4	5	15	20	30	70	0.5	3.8	4.2	5.9	11.6
JMeter	51	38	372	408	764	12	13	15	28	42	61	0.5	0.8	1.1	1.6	2.4
Jaba	62	55	289	326	600	11	12	14	25	51	115	2.5	15.0	16.8	23.0	24.8
ArgoUML	190	172	7,465	7,542	11,998	8	10	11	23	38	87	1.4	7.3	8.1	11.4	17.8
Overall	**82**	**73**	**2,047**	**2,115**	**3,392**	**8**	**9**	**12**	**21**	**35**	**76**	**1.3**	**6.8**	**7.9**	**10.9**	**14.3**

Finally, method-instance level dynamic alias data led to slightly larger precision gain than method-level data, which is most noticeable with Schedule1, NanoXML and Jaba, possibly because of relatively heavier use of pointers in these subjects. Overall, IAPRO reduces the baseline impact sets by 61%–66%, implying an increase in precision of about 160%–200%.

2) Efficiency: Table II summarizes the means and standard deviations (in the parentheses) of query costs incurred by the baseline and all IAPRO instances, in the right five columns. As expected, IAPRO incurred higher cost during post-processing than the baseline because of the larger execution data they traversed (full method execution trace by IAPRO versus two integers per method by PI/EAS). Among the IAPRO instances, query cost generally grows as more dynamic data is used. An exception is FC_{ml}, though, for which adding method-level dynamic aliasing data even reduced the cost with respect to *TC* for five out of the seven subjects. A possible reason is that spurious alias-induced data dependencies account for a relatively large portion of all dependencies in these subjects. In consequence, pruning those spurious DDs using dynamic points-to sets sped up searches on the reduced dependence graph during the process of impact computation.

Naturally, higher costs are associated with subjects of denser dependencies (e.g., Jaba and ArgoUML), longer traces (e.g., Jaba), and/or larger test input (e.g., Schedule1). The lowest efficiency of IAPRO was seen with Jaba, which has a much larger dependence graph and longer traces than other subjects. In addition, the generally large standard deviations suggest the post-processing time fluctuates greatly across different queries in most subjects. The *overall* numbers show that the query cost of IAPRO is about half a minute per query. In fact, the average cost for the other six subjects than Jaba would be 10–20s. In all, the query cost of IAPRO looks reasonable. Further, multiple queries can be easily parallelized since their computation is independent of each other.

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

The rest of Table III shows the time costs of the first two phases of DDIA techniques studied here, including the uncaught-exception profiling costs incurred by all IAPRO techniques (*Prof.*) and the execution time of inputs on the original (uninstrumented) programs (*Normal*). Mostly, both the profiling cost and static-analysis time tend to increase with the program size, with the peak number seen by the largest subject ArgoUML. Similar to previous observations, the method-instance level dynamic alias data again led to the greatest cost growth, in both the static-analysis and runtime phases. Nonetheless, the profiling is generally cheap, as is the generation of addition dynamic data at runtime. Static analysis is mostly efficient too, except for the largest subject ArgoUML. However, as the runtime phase, the static analysis incurs one-time costs for all queries for the same program version, and it can be incorporated in nightly builds in practice.

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

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

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

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

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

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

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

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

D. Threats to Validity

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

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

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

Finally, a *conclusion* threat is the appropriateness of our statistical analyses. To reduce this threat, we used a non-parametric hypothesis test which makes no assumptions about the distribution of the underlying data (e.g., normality).

TABLE IV: Wilcoxon p-values for Null Hypotheses that There Are No Differences in Means with respect to Various Paired Contrasts among PI/EAS and IaPro Techniques (Confidence Level=0.95)

Subject	PI/EAS vs *TO*	*TO* vs *TC*	*TO* vs FC_{ml}	*TO* vs FC_{mil}	*TC* vs FC_{ml}	*TC* vs FC_{mil}	FC_{ml} vs FC_{mil}
Schedule1	1.73E-05	1	0.4854	0.2261	0.4854	0.2261	0.5523
NanoXML	9.41E-24	0.0053	0.0052	0.0001	0.9922	0.1298	0.1325
Ant	8.06E-130	0.0033	0.0025	0.0014	0.9126	0.7766	0.8583
XML-security	1.22E-104	0.1029	0.0507	0.0196	0.733	0.4659	0.6965
JMeter	2.71E-181	0.6369	0.5411	0.1745	0.8892	0.3777	0.4571
Jaba	6.43E-42	0.0384	0.0129	0.0001	0.5083	0.0184	0.0423
ArgoUML	1.77E-176	0.1575	0.0995	0.0906	0.8086	0.7703	0.9599
Fisher Overall	**0**	**0.0006**	**4.99E-05**	**2.79E-09**	**0.9935**	**0.1449**	**0.4326**

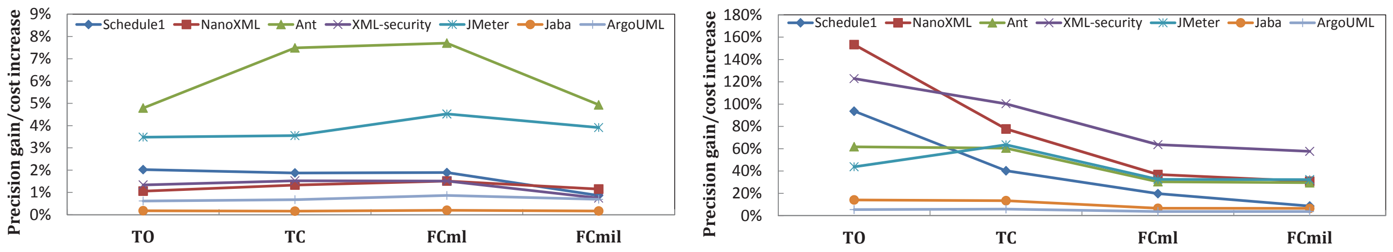

Fig. 4: Cost-effectiveness of the IaPro techniques expressed as the ratios of their precision gain to the increase in the average query cost (left) and total cost of the first two phases (right), both against PI/EAS.

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

VI. Related Work

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

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

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

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

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

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

VII. CONCLUSION

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

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

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

VIII. ACKNOWLEDGMENTS

This work was partially supported by ONR Award N000141410037 to the University of Notre Dame.

REFERENCES

[1] V. Rajlich, "Software evolution and maintenance," in *Proceedings of the Conference on Future of Software Engineering*, 2014, pp. 133–144.

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

[3] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *FSE*, 2012, pp. 51:1–51:11.

[4] P. Rovegard, L. Angelis, and C. Wohlin, "An empirical study on views of importance of change impact analysis issues," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 516–530, 2008.

[5] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *TOPLAS*, vol. 12, no. 1, pp. 26–60, 1990.

[6] L. Li and A. J. Offutt, "Algorithmic analysis of the impact of changes to object-oriented software," in *ICSM*, 1996, pp. 171–184.

[7] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating influence mechanisms into impact analysis for increased precision," in *ICSM*, 2006, pp. 55–65.

[8] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[9] A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *ICSE*, 2004, pp. 491–500.

[10] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *FSE*, 2003, pp. 128–137.

[11] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "Jripples: A tool for program comprehension during incremental change." in *Proc. of 9th Int'l Workshop on Program Comprehension*, 2005, pp. 149–152.

[12] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE*, 2003, pp. 308–318.

[13] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *ICSE*, 2005, pp. 432–441.

[14] H. Cai, R. Santelices, and T. Xu, "Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis," in *International Conference on Software Security and Reliability*, 2014, pp. 48–57.

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

[16] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *ASE*, 2014, pp. 343–348.

[17] C. R. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *ICSE*, 2008, pp. 241–250.

[18] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers, "Program slicing with dynamic points-to sets," *Software Engineering, IEEE Transactions on*, vol. 31, no. 8, pp. 657–678, 2005.

[19] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Impact analysis in the presence of dependence clusters using static execute after in webkit," *Journal of Software: Evolution and Process*, vol. 26, no. 6, pp. 569–588, 2013.

[20] M. Ajrnal Chaumun, H. Kabaili, R. K. Keller, and F. Lustman, "A change impact model for changeability assessment in object-oriented software systems," in *CSMR*, 1999, pp. 130–138.

[21] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a Java Bytecode Optimization Framework," in *Cetus Users and Compiler Infrastructure Workshop*, 2011, pp. 1–11.

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

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

[24] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *EMSE*, vol. 10, no. 4, pp. 405–435, 2005.

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

[26] F. Mosteller and R. A. Fisher, "Questions and answers," *The American Statistician*, vol. 2, no. 5, pp. 30–31, 1948.

[27] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich, "Static execute after/before as a replacement of traditional software dependencies," in *ICSM*, 2008, pp. 137–146.

[28] L. Huang and Y.-T. Song, "Precise dynamic impact analysis with dependency analysis for object-oriented programs," in *Int'l Conf. on Softw. Engg. Res., Manag. & Apps*, 2007, pp. 374–384.

[29] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero, "The hybrid technique for object-oriented software change impact analysis," in *CSMR*, 2010, pp. 252–255.

[30] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio, "On the precision and accuracy of impact analysis techniques," in *Int'l Conf. on Computer and Information Science*, 2008, pp. 513–518.

[31] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 159–182, 2002.

[32] B. Korel, "Automated software test data generation," *Software Engineering, IEEE Transactions on*, vol. 16, no. 8, pp. 870–879, 1990.

[33] M. Weiser, J. D. Gannon, and P. R. McMullin, "Comparison of structural test coverage metrics," *IEEE Software*, vol. 2, no. 2, pp. 80–85, 1985.

[34] M. Kamkar, "An overview and comparative classification of program slicing techniques," *J. Syst. Softw.*, vol. 31, no. 3, pp. 197–214, 1995.

[35] J. Law and G. Rothermel, "Incremental dynamic impact analysis for evolving software systems," in *ISSRE*, 2003, pp. 430–441.

[36] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *ICSM*, 1993, pp. 348–357.

[37] G. A. Venkatesh, "The semantic approach to program slicing," *SIGPLAN Notice*, vol. 26, no. 6, pp. 107–119, 1991.