An Empirical Comparison of Dynamic Impact Analysis Algorithms

Alessandro Orso,[‡] Taweesup Apiwattanapong,[‡] James Law,[†]
Gregg Rothermel,[†] and Mary Jean Harrold,[‡]

[‡]College of Computing
Georgia Institute of Technology
Atlanta, Georgia
{orso, term, harrold}@cc.gatech.edu

[†]Computer Science Department
Oregon State University
Corvallis, Oregon
{law, grother}@cs.orst.edu

Abstract

Impact analysis — determining the potential effects of changes on a software system — plays an important role in software engineering tasks such as maintenance, regression testing, and debugging. In previous work, two new dynamic impact analysis techniques, `CoverageImpact` *and* `PathImpact`*, were presented. These techniques perform impact analysis based on data gathered about program behavior relative to specific inputs, such as inputs gathered from field data, operational profile data, or test-suite executions. Due to various characteristics of the algorithms they employ,* `CoverageImpact` *and* `PathImpact` *are expected to differ in terms of cost and precision; however, there have been no studies to date examining the extent to which such differences may emerge in practice. Since cost-precision tradeoffs may play an important role in technique selection and further research, we wished to examine these tradeoffs. We therefore designed and performed an empirical study, comparing the execution and space costs of the techniques, as well as the precisions of the impact analysis results that they report. This paper presents the results of this study.*

1 Introduction

As software systems evolve, changes made to those systems can have unintended or even disastrous effects [7]. Software change impact analysis is a technique for predicting the potential effects of changes before they are made, or measuring the potential effects of changes after they are made. Applied before modifications, impact analysis can help maintainers estimate the costs of proposed changes, and select among alternatives. Applied after modifications, impact analysis can alert engineers to potentially affected program components requiring retesting, thus reducing the risks associated with releasing changed software.

Many impact-analysis techniques have been presented in the literature. In this paper, we focus on dependency-based impact analysis techniques [2]. Dependency-based impact-analysis techniques proposed to date (e.g., [1, 2, 8, 12, 14, 16]) rely on static analysis based on forward slicing or transitive closure on call graphs to identify the syntactic dependencies that may signal the presence of important semantic dependencies [13].

Recently, two new dependency-based impact analysis algorithms, `CoverageImpact` [10] and `PathImpact` [5, 6], have been proposed. These algorithms differ from previous impact analysis algorithms in their reliance on dynamic information about program behavior. This information is computed in the form of execution data for a specific set of program executions, such as executions in the field, executions based on an operational profile, or executions of test suites. Naturally, the impact estimates computed in this manner reflect only observed program behavior; however, these estimates may be particularly useful in cases where safety is not required because they provide more precise information about impact — relative to a specific program behavior — than static analyses [5, 6, 10].

Although `CoverageImpact` and `PathImpact` are similar in overall approach, an analysis of the techniques suggests that they may behave quite differently in practice. `PathImpact` is expected to be more precise than `CoverageImpact`, but also more expensive in terms of space and time usage. Analytical comparisons, however, do not reveal the extent to which relative costs and benefits of heuristics emerge in practice, when they are applied to real programs, real changes, and real input data. Practitioners wishing to adopt dynamic impact analysis techniques, and researchers seeking to improve them, need to understand the cost-benefits tradeoffs that hold for those techniques in practice.

We have therefore designed and performed an experiment comparing the cost and precision of `CoverageImpact` and `PathImpact` applied to a set of non-trivial Java programs. We considered several

versions of each program. For each version we performed impact analysis using both techniques and compared (1) the precision of the results, (2) the cost of the analyses in terms of space, and (3) the cost of the analyses in terms of time. The results of the study support the analytical claim that tradeoffs exist between the two techniques, and provide data on the extent to which these tradeoffs actually surface in practice. The results also show that the techniques have complementary strengths.

In the next section of this paper, we summarize the `CoverageImpact` and `PathImpact` techniques and provide a model for use in assessing their costs and benefits. Section 3 presents our experiment design, results, and analysis. Section 4 discusses the implications of our results, and Section 5 presents conclusions.

2 Dynamic Impact Analysis

Let P be a program on which we are performing impact analysis and let C be a set of methods $\{m\}$ in P in which changes have been or may be made. The impact analysis techniques we consider compute an *impact set* — a set of methods potentially impacted by changes in C.

To illustrate the techniques we use a simple example. Figure 1 provides the example in terms of a call graph for a program P. The edge connecting nodes D and C is a back-edge in the call graph and, thus, indicates recursion.

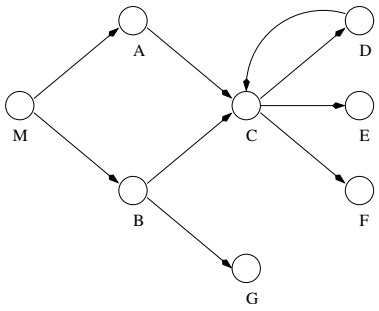

Figure 1. Call graph for program P.

Because the techniques we consider perform dynamic impact analysis, we provide, as part of our example (Figure 2) dynamic execution data corresponding to possible executions of program P. Each line in this figure corresponds to a a trace of an execution of P, and consists of an identifier for the execution followed by a list of methods called and return statements encountered during that execution. Each `r` represents the return from the most recently called method, and `x` represents the exit from the program. For example, the trace for Exec2 corresponds to an execution in which M is called, M calls B, B calls C, C returns to B, B calls G, G returns to B, B returns to M, and M exits.

```
Exec1: M B G r G r r A C E r r r x
Exec2: M B C r G r r x
Exec3: M A C E r D r r r x
Exec4: M B C E r F r D r r ... C E r F r D r r r x
```

Figure 2. Traces for P. The dots in the last trace indicate that the sequence CErFrDrr is repeated several times.

2.1 Coverage-Based Dynamic Impact Analysis

The `CoverageImpact` technique relies on lightweight instrumentation to collect dynamic information from a running software system. As the instrumented program executes, it records coverage information in the form of bit vectors, and uses these bit vectors to compute the impact set. The bit vectors contain one bit per method. A value of 1 in the bit for method m in the vector for execution e indicates that m was covered in e, and a value of 0 indicates that m was not covered in e. For example, for the executions shown in Figure 2, the coverage information consists of the set of bit vectors shown in Table 1.

Exec ID	M	A	B	C	D	E	F	G
Exec1	1	1	1	1	0	1	0	1
Exec2	1	0	1	1	0	0	0	1
Exec3	1	1	0	1	1	1	0	0
Exec4	1	0	1	1	1	1	1	0

Table 1. Coverage bit vectors for the execution traces in Figure 2.

Given a set of changed methods C, `CoverageImpact` uses the bit vectors collected during execution to determine an impact set. We provide an overview of this analysis; the complete algorithm is presented in Reference [10]. To identify the impact set, `CoverageImpact` computes, for each method m in C, a dynamic forward slice based on the coverage data for executions that traverse m. The impact set is the union of the slices thus computed.

`CoverageImpact` computes each dynamic slice in three steps. First, using the coverage information, it identifies the executions that traverse m and adds the methods covered by such executions to a set of *covered methods*. Second, it computes a static forward slice from m and adds the methods included in the slice to a set of *slice methods*. (To compute such a slice, we start slicing from all variable definitions in method m.) Third, it takes the intersection of covered methods and slice methods. The result of the intersection is the impact set.

To illustrate, consider the impact set computed by `CoverageImpact` for the program and executions in our example (Figures 1 and 2, Table 1) for change $C = \{\text{A}\}$ (i.e., only method A is modified). The executions that tra-

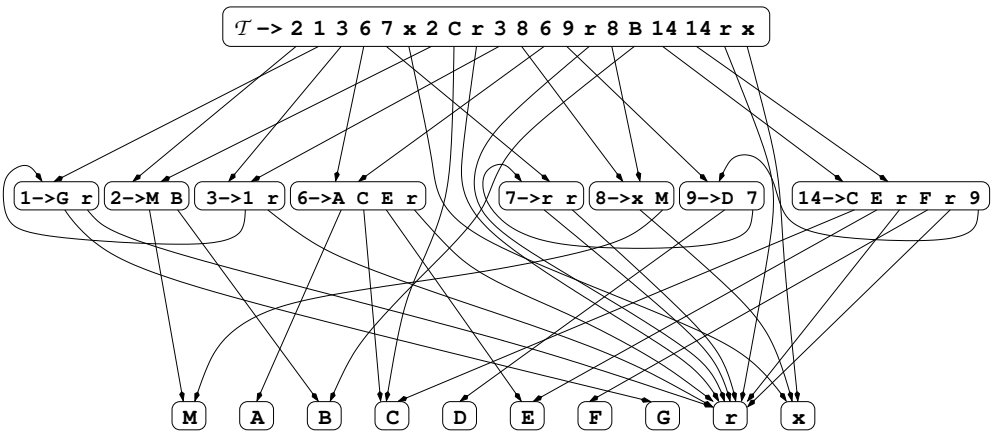

Figure 3. Whole-path DAG for the execution traces in Figure 2.

verse A are Exec1 and Exec3, and the covered methods include M, A, B, C, D, E, and G. Assume that the slice methods for method A consists of methods M, A, C, D, E, and F. The resulting impact set computed by `CoverageImpact`, which is the intersection of covered methods and slice methods, is then { M, A, C, D, E }.

2.2 Path-Based Dynamic Impact Analysis

The `PathImpact` [5] technique also relies on instrumentation to collect dynamic information from a running software system. As the instrumented program executes, it records multiple execution traces, of the form shown in Figure 2. `PathImpact` first processes these traces sequentially using the SEQUITUR compression algorithm [9] to obtain a compact representation called a whole-path DAG [4] (directed acyclic graph). The execution traces can be processed as they are generated and need not be stored, and the whole-path DAG is used instead of the traces to calculate the change impact set. Given the execution traces in Figure 2, and assuming that the sequence CErFrDrr is repeated two times in Exec4, the resulting DAG is shown in Figure 3. The compression algorithm creates rules, shown as numbered interior nodes in the DAG, to remove repetition within and across traces.

`PathImpact` walks the DAG to determine an impact set given a set of changes. The complete algorithm for performing this DAG walk is presented in Reference [6]. Intuitively, one way to visualize its operation is to consider beginning at a changed method's node in the DAG, walking through the DAG by performing recursive forward and backward in-order traversals at each node, and stopping when any trace termination symbol, `x`, is found. By traversing forward in the DAG, the algorithm finds all methods that execute after the change and therefore could be affected by the change. By traversing backward in the DAG, the algorithm determines all methods into which execution can return.

To illustrate, consider the impact set computed by `CoverageImpact` for the program and the executions in our example (Figures 1 and 2, Table 1) for change $C = \{\text{A}\}$ (i.e., only method A is modified). (This same example is used to illustrate `CoverageImpact`.) `PathImpact` starts at leaf node A, and walks through the DAG, as described above. The resulting impact set computed by `PathImpact` is $\{$M, A, C, D, E$\}$.

2.3 Cost-Benefits Tradeoffs

`CoverageImpact` and `PathImpact` are both safe with respect to the dynamic information used for impact analysis: that is, neither technique excludes from its identified impact set any method that may be affected relative to the available dynamic information. `CoverageImpact` and `PathImpact` can differ, however, in terms of precision and cost.

Where precision is concerned, `PathImpact` is expected to be more precise than `CoverageImpact`. Consider, for example, the computation of the impact set for the example in Figure 1, when method D is changed. Assume, for this example, that execution Exec3 (Figure 2) is the only dynamic information available. In this case, `PathImpact` builds a DAG for Exec3 only and then, based on that DAG, computes impact set $\{$M, A, C, D$\}$. In contrast, in this same case, `CoverageImpact` computes the impact set $\{$M, A, C, D, E$\}$. `CoverageImpact` includes E in this set because (1) E is in the static forward slice computed from the point of the change, D, (2) E is covered by execution Exec3, and (3) using coverage, rather than trace, information, it is impossible to identify whether a method is executed before or after another method. This imprecision may occur, for instance, in the presence of recursion or calls within loops. Note that there could also be cases in which `CoverageImpact` is more precise than `PathImpact`. For example, if method M contains a call to D followed by a call to C, and D has no

data dependences that involve C, the slice computed from D would not include D. In such a case, D would not be in the impact set for `CoverageImpact`, whereas it would be in the impact set for `PathImpact` (as discussed above).

Where cost is concerned, `PathImpact` is analytically more expensive than `CoverageImpact` in terms of both space and time. Consider, for example, the computation of the impact set for our example for any given change, using execution Exec4 as dynamic information. To compute the impact set, `CoverageImpact` stores the coverage information as a bit vector of fixed size (8 bits in this case) and performs the analysis using only this bit vector. In contrast, `PathImpact` must process the entire trace to generate a DAG ($4 + 8 * n$ elements, where n is the number of times the sequence CErFrDrr is repeated) and then must walk the DAG to compute the impact set.

We next describe these tradeoffs in greater detail.

2.3.1 Relative Precision

Consider two impact sets, IS_A and IS_B, computed by safe impact analysis techniques A and B for the same program, change, and dynamic information. The number of methods in IS_A and not in IS_B can be used to measure the relative imprecision of A with respect to B, and vice versa. Any elements that are in the set identified by A and not in the set identified by B represent imprecision: cases in which the method identified as impacted is not in fact impacted by the modification (relative to the given set of executions). Such imprecision forces engineers to spend unneeded time investigating potential impacts. To measure the relative precision of `CoverageImpact` with respect to `PathImpact`, we thus measure the differences in the sizes of their impact sets.

2.3.2 Relative Cost in Space and Time

To compare the costs of `CoverageImpact` and `PathImpact` we define a model of a general process for impact analysis (Figure 4). `CoverageImpact` and `PathImpact` are two instances of this process, distinguished by the type and amount of information they collect and the actors that play different roles in the processes.

Initially, a program is instrumented to collect dynamic information while it executes. The types of instrumentation and information produced depend on the impact analysis performed. `CoverageImpact` requires the addition of a probe at the entry of each method, whereas `PathImpact` requires two or more probes per method, one at the entry and one at each exit.

The instrumented program is executed to collect the dynamic information produced by the probes in the code. For `CoverageImpact`, the dynamic information consists of a token per executed method, whereas for `PathImpact`, the dynamic information consists of two tokens per method, one produced at the entry and one at the exit.

The dynamic information is processed, while being produced, to create the execution data needed by the impact analysis techniques, which is then stored on disk. For `CoverageImpact`, the processor sets, for each token, the bit for the corresponding method in the bit vector that contains coverage information. For `PathImpact`, the processor builds the DAG based on the sequence of tokens read from the trace, as described in Section 2.2.

Impact analysis is performed using the stored execution data and information on the changes in the program. Table 2 summarizes the characteristics of `CoverageImpact` and `PathImpact` relative to this model.

Based on this model, we identify the following costs, in space and time, for dynamic impact analysis techniques.

Space cost is the space required to store the execution data on disk. For `CoverageImpact`, space cost consists of the size of the bit vectors that contain method-coverage information, one per execution. For `PathImpact`, space cost is the size of the DAGs. Depending on the context, there could be one DAG per execution (e.g., if executions are performed at different user sites), or one DAG for the whole set of executions. In our study, we measure space cost in both cases.

Time overhead is the additional time required for the execution of the program due to instrumentation and processing of dynamic information. For `CoverageImpact`, time overhead is caused by the execution of one probe for each method invocation and by the updating of the coverage bit vector. For `PathImpact`, time overhead is caused by the execution of two probes for each method invocation and by the time required to process the traces and build the DAG.

We do not consider the cost of the last step of the process modeled in Figure 4, the generation of the impact sets, for three reasons. First, this step can be performed off-line with respect to the execution of the program. Second, in our preliminary work, we have observed it to be relatively low cost for both techniques considered. Third, maintainers are typically willing to wait reasonable times for impact analysis results [2]. In contrast, overhead involved in program execution is an important cost, especially when execution data are collected from deployed programs running on users' platforms.

3 Empirical Study

The foregoing analyses of precision and cost suggest potential differences between `CoverageImpact` and `PathImpact`; however, such analyses cannot predict the extent to which these differences will surface when applied to actual software systems, versions, and execution data. Our goal, then, is to empirically investigate the relative costs and benefits of the techniques in practice. Toward this end, we designed and performed a controlled experiment. The

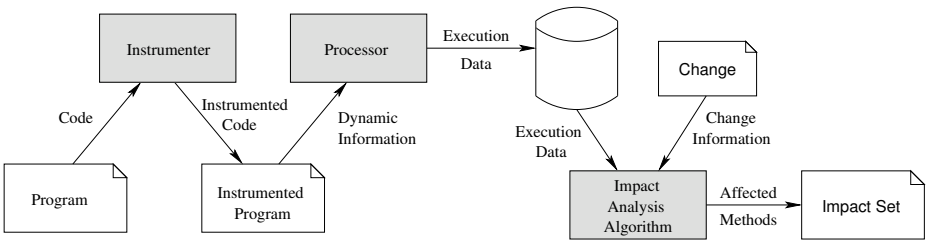

Figure 4. Model of the dynamic impact analysis process.

	`CoverageImpact`	`PathImpact`
Instrumenter	Inserts probes at method entries	Inserts probes at method entries and exits
Dynamic Information	One token per method call (method id)	Two tokens per method call (method id, return)
Processor	Bit vectors builder	DAG builder
Execution Data	Bit vectors	DAGs
Impact Analysis Algorithm	Coverage based	Path based

Table 2. Characteristics of `CoverageImpact` and `PathImpact` with respect to the model of Figure 4.

following subsections present our variables and measures, experiment setup and design, threats to validity, and data and analysis. Further discussion of the results and their implications follows in Section 4.

3.1 Variables and Measures

3.1.1 Independent Variables

Our experiment manipulated one independent variable: impact analysis technique, the two categorical factors being `CoverageImpact` and `PathImpact`.

3.1.2 Dependent Variables and Measures

To investigate our research questions we use three dependent variables — precision, space cost, and time overhead — derived from the discussion and cost model presented in Section 2.3.

To evaluate the precisions of `CoverageImpact` and `PathImpact`, we measure the relative sizes of the impact sets computed by the techniques on a given program, change set, and set of program executions. We report and compare such sizes in relative terms, as a percentage over the total number of methods in the sets.

As discussed in Section 2.3, the primary factor in determining space cost for `CoverageImpact` is the size of the bit vectors representing test coverage of methods. The primary factor in determining space cost for `PathImpact` is the size of the DAGs constructed by the algorithm following its processing of a set of traces. Thus, to evaluate the relative space usage of `CoverageImpact` and `PathImpact`, we measure and compare these sizes, in terms of disk space required to store the corresponding information.

To evaluate relative execution costs for `CoverageImpact` and `PathImpact`, we measured the time required to execute an instrumented program on a set of test cases, gathered the execution data (bit vectors or DAGs) needed for subsequent impact analysis, and output that information to disk. We compare the execution costs for the two techniques to each other, and to the cost of executing a non-instrumented program on the same set of test cases.

3.2 Experiment Setup

3.2.1 Object of Analysis

As objects of analysis we used several releases of three programs: NANOXML, SIENA, and JABA (see Table 3).

Program	Versions	Classes	Methods	LOC	Test Cases
NanoXML	6	19	251	3279	216
Siena	8	24	219	3674	564
Jaba	11	355	2695	33183	125

Table 3. Objects of analysis

NANOXML, an XML parser written in Java [15], is a component library that has been used with various application programs. SIENA (Scalable Internet Event Notification Architecture) [3] is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks. JABA (Java Architecture for Bytecode Analysis)[1] is a framework for analyzing Java programs. For this study, we used several consecutive versions of each of these systems, as indicated in Table 3.

To provide input sets for use in determining dynamic behavior, for NanoXML and Siena, we used test suites created for earlier experiments [11]. These test suites consist of specification-based test cases created using the user documentation and interface descriptions for the programs. For JABA, we used its existing regression test suite.

[1] `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`

3.3 Experiment Design and Analysis Strategy

As a representation of `CoverageImpact`, we used an implementation created at Georgia Tech for use in earlier studies [10]. In this implementation, we approximated static forward slicing by computing simple reachability on the interprocedural control-flow graph. As a representation of `PathImpact`, we used an implementation created at Oregon State University for use in earlier studies [5, 6].

As change sets for use in assessing impacts, we used the actual sets of changes from each version of our subject programs to each subsequent version.

Given this infrastructure, to conduct the experiment, we first ran the instrumented programs on all workloads and gathered bit vectors and DAGs using the two implementations; for each version of the program being considered, we did this both for the entire test suites and for each individual test case. To gather time-overhead data, we measured, for each test execution, the time required to run the program and build and store the bit vector or the DAG; to measure such time, we used differences in system times, obtained through calls to the standard library, at the beginning and at the end of the execution. We then collected size measures on the bit vectors and DAGs. Finally, we applied each impact analysis technique to each set of bit vectors or DAGs, for each change set considered, and collected information about the size of the generated impact sets. We collected all data on four dedicated 2.80 GHz Pentium4 PCs, with 1 GB of memory, running GNU/Linux 2.4.21.

Because we collected time, space, and precision data for both the entire test suites and each individual test case, for each of the cost factors we report two different sets of results: individual test execution results and test-suite execution results. *Individual test execution results* are the results that we computed by considering each test case in a test suite independently and then averaging the results across all test executions. *Test suite execution results* are the results that we computed considering the entire test suite at once. Considering both sets of results let us perform a more thorough comparison of the two techniques.

3.4 Threats to Validity

Like any empirical study, this study has limitations that must be considered when interpreting its results. We have considered the application of the impact-analysis techniques studied to only three programs, using one set of test data per program, and cannot claim that these results generalize to other scenarios. However, the system and versions used are real, non-trivial software systems, the change sets used are actual change sets, and the test suites used represent suites that are or could be realistically used in practice. Nevertheless, further studies with other subjects are needed to address questions of external validity.

Threats to construct validity concern the appropriateness of our measures for capturing our dependent variables. Our measures of time and space capture some aspects of these variables, but other factors (e.g., the amount of memory rather than disk consumed) may be important in particular situations. Other components of effectiveness, such as the relative usefulness of the generated impact sets, could also be considered. Finally, different approaches to implementing the impact-analysis techniques could alter the relative importance of specific cost factors, and require different measures. For example, given a version of either impact-analysis tool in which the processor operates asynchronously as a second process, overall execution and space costs would vary depending on (1) the relative speeds of the executing instrumented program and the processor, and (2) the kind of buffering used between the processes.

Threats to internal validity concern our ability to draw conclusions about the connections between our independent and dependent variables. In this experiment, our primary concern involves instrumentation effects, such as errors in our algorithm implementations or measurement tools, that could affect outcomes. To control for these threats, we validated the implementations and tools on known examples.

A second threat to internal validity involves drawing conclusions based on specific algorithm implementations. Our implementations are research prototypes and were not constructed with efficiency in mind, nor was their construction controlled for differences that could impact relative efficiency. Furthermore, we have approximated static slicing with reachability, which, in some cases, may result in less precise results for `CoverageImpact`. Note that these limitations do not apply to our space cost results, for which we are able to obtain accurate measures using our prototypes.

3.5 Results and Analysis

In the following sections we present and analyze the data obtained for each of our three dependent variables, in turn, using descriptive and inferential statistics. Section 4 provides additional discussion.

3.5.1 Precision

Figure 5 summarizes the precision results observed in our studies with two graphs; the graph on the left represents individual test execution results, and the one on the right represents test-suite execution results. In both graphs, each version of each program occupies a position along the horizontal axis, and the relative impact set size for that program and version are represented by a vertical bar — grey for the `CoverageImpact` technique and black for the `PathImpact` technique. The height of the bars represents the impact set size relative to the number of methods in the program, expressed as a percent, for each program and version. By normalizing the impact set size in this way we can

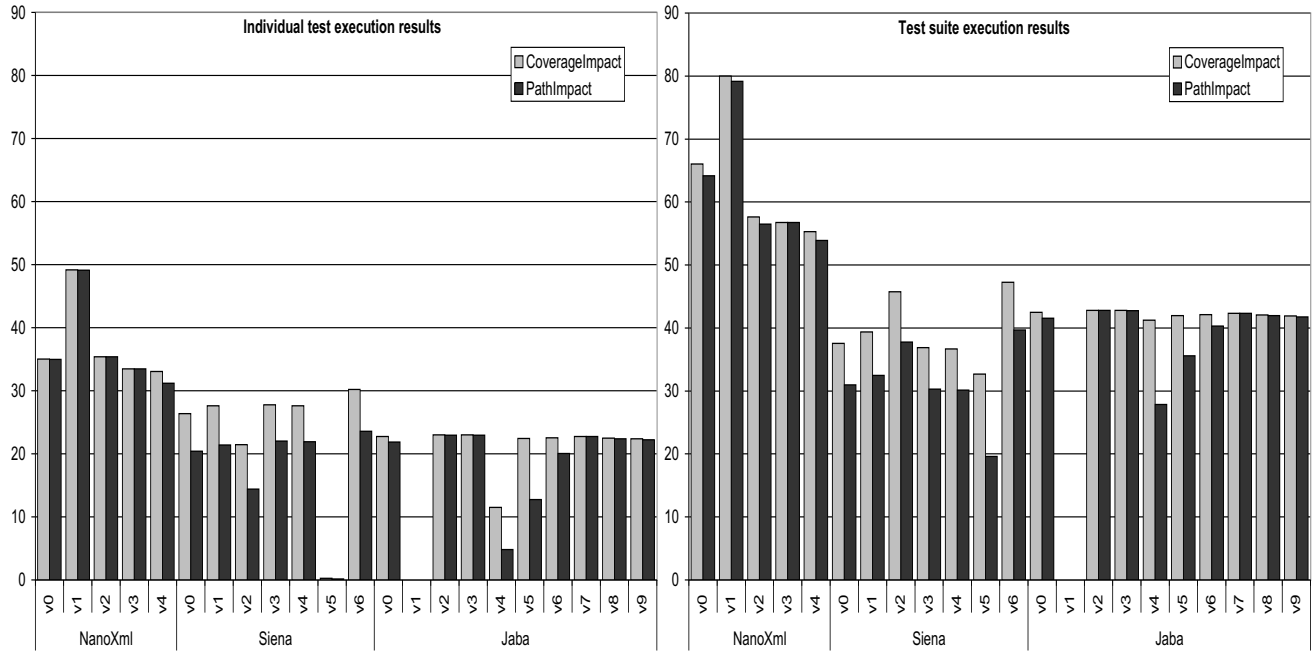

Figure 5. Precision results, expressed as percentage of methods in the impact sets.

Program	v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
NanoXml	0.9512	0.9933	0.9930	1.0000	0.0997	–	–	–	–	–
Siena	**0**	**0**	0.1332	**0**	**0**	0.6778	**0**	–	–	–
Jaba	0.0653	N/A	0.9328	0.9331	**0**	**0**	**0**	1	0.8667	0.7377

Table 4. ρ-values from t-tests on individual test execution precision results, per program and version.

make comparisons across programs and versions that have different numbers of methods.

The graphs show that, in many cases, `PathImpact` is more precise than `CoverageImpact`. The difference in relative precision between the two techniques varies across programs and versions: it ranges from 0 to 9.7, with an average of 2.7, for the individual test execution results, and from 0 to 13.4, with an average of 3.8, for the test suite results.

We formally investigated the differences between the precision of the two techniques using paired t-tests across individual test execution results within each program version. The null hypothesis was that the mean of the impact set sizes for the two techniques was equal. T-tests for which the ρ-value is less than or equal to a level of significance α of 0.05 are statistically significant. Table 4 summarizes the results of this analysis, per program and version, listing the ρ-values that result from each t-test. Entries in which results are statistically significant (ρ-value < 0.05) are shown in bold. (Blank entries occur due to the different numbers of versions available per program. The ρ-value for version $v1$ of JABA cannot be computed because the changes for this version were not traversed by any test case and, thus, the size of the impact sets is zero for both techniques.)

The ρ-values in the table indicate that there are statistically significant differences in the impact sets computed by `CoverageImpact` and `PathImpact` on eight out of 21 cases (38%) — for five versions of Siena and three versions of JABA. For the other 13 cases (62%), the results are not statistically significantly different for an α of 0.05.

3.5.2 Space Cost

Figure 6 summarizes the test-suite execution results for space costs that we observed in our studies. For each program and version, the graph in the figure represents space cost data with a vertical bar — grey for the `CoverageImpact` technique and black for the `PathImpact` technique. The height of the bars depicts the absolute size of the impact analysis data stored by the technique, expressed in KB.

The graph shows that, for all programs and versions, `CoverageImpact` requires from seven to 30 times less space than `PathImpact` to store execution data. However, the size required by `PathImpact` is never larger than 1.1 MB per suite, which makes the technique practical in cases in which space cost is not severely constrained (for example, when collecting execution data in house rather than from deployed software [10]).

For space cost, we do not show a graph for individual test execution results because, for those results, the differ-

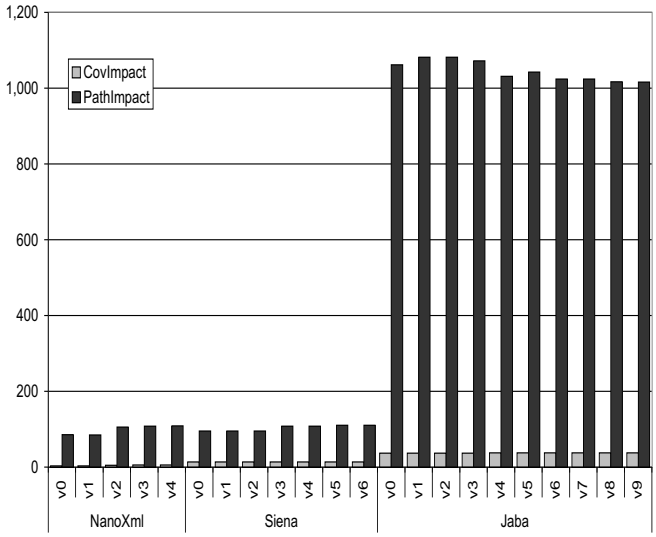

Figure 6. Space cost results, in KB.

	Individual test execution			Test suite execution		
Program	*Uninst*	*Cov*	*Path*	*Uninst*	*Cov*	*Path*
NanoXml-v0	21	86	506	4,536	18,379	2,354,081
NanoXml-v1	29	94	435	6,261	20,189	862,384
NanoXml-v2	33	99	474	7,123	21,235	1,199,627
NanoXml-v3	38	103	548	8,287	22,198	1,226,769
NanoXml-v4	39	103	538	8,348	22,259	1,372,292
Siena-v0	658	694	868	370,999	391,575	1,807,538
Siena-v1	658	693	868	370,904	391,115	1,807,192
Siena-v2	658	694	889	371,130	391,231	1,786,140
Siena-v3	658	694	889	371,163	391,386	1,944,428
Siena-v4	658	694	890	371,211	391,270	1,944,974
Siena-v5	658	696	908	371,205	392,606	1,754,441
Siena-v6	658	695	908	371,249	293,093	1,754,009
Jaba-v0	413	427	58,959	51,586	53,344	57,749,814
Jaba-v1	408	425	69,677	51,044	43,097	54,127,217
Jaba-v2	410	427	53,461	51,229	53,393	56,437,190
Jaba-v3	412	430	53,089	51,605	53,710	60,012,216
Jaba-v4	414	433	56,316	51,769	54,099	54,899,038
Jaba-v5	417	434	56,327	52,226	54,270	57,053,813
Jaba-v6	412	433	56,835	51,611	54,085	54,544,107
Jaba-v7	414	484	56,171	51,856	60,459	57,015,465
Jaba-v8	462	476	56,877	57,709	59,495	54,411,568
Jaba-v9	466	479	57,110	58,246	59,824	54,823,308

Table 5. Time cost results, in ms.

ences are even more extreme. This is because `PathImpact` can generate a more compressed DAG when processing traces for a whole test-suite than when processing traces for executions of single test cases individually; conversely, `CoverageImpact` always computes a bit vector of fixed size, per execution. Therefore, only when gathering bit vectors for a whole test suite does their cumulative size become comparable to the size of a DAG for the same suite.

3.5.3 Time Overhead

Table 5 summarizes the time cost results observed in our studies. For each program and version, the table reports the time required to execute the uninstrumented program (*Uninst*), the time required to execute an instrumented program and gather and store the bit vectors (*Cov*), and the time required to execute an instrumented program and gather and store the DAGs (*Path*).

Like Figure 5, Table 5 reports both individual test results, in which we gather the execution times for each test case independently and then average them across all test cases, and test-suite results, in which we gather the execution time required to execute the whole test suite at once.

As the table shows, the execution times for `PathImpact` are consistently higher than those for `CoverageImpact` As with the space results, the difference in time overhead between the two techniques varies considerably in the table: it ranges from 174 ms (SIENA $v0$) to 69,252 ms (JABA $v1$), for the individual test execution results, and from 842,205 ms (NANOXML $v1$) to 59,958,506 ms (about 16 hours, on JABA $v3$), for the test-suite results. The table also shows the degree to which both `CovImpact` and `PathImpact` impose overhead on the execution time for the uninstrumented programs, and the extent to which that overhead differs between the two techniques.

We used paired t-tests to investigate the differences between time overhead of the two techniques, across individual test execution results, by program and version. We also employed paired t-tests to compare the costs of each technique to the cost of executing the non-instrumented program. The null hypotheses for these three t-tests are that (1) the time costs of the two impact analysis techniques are equal, (2) the time cost of `CoverageImpact` is negligible with respect to the time cost of running the uninstrumented program, and (3) the time cost of `PathImpact` is negligible with respect to the time cost of running the uninstrumented program. Table 6 summarizes the results of these analyses, per program and version, listing the ρ-values that result from each t-test, with bold indicating significance.

The ρ-values in the first part of the table show that the differences in time costs of `CoverageImpact` and `PathImpact` are statistically significant in 20 out of 22 cases (in all cases except for SIENA versions $v0$ and $v1$). The ρ-values in the second part of the table show that there is no significant difference in the time cost of running `CoverageImpact` and the time cost of running the uninstrumented program for SIENA and JABA, whereas the difference is significant for NANOXML. Finally, the ρ-values in the third part of the table show that there is a significant difference in the time cost of running `PathImpact` and the time cost of running the uninstrumented program for all programs and versions.

4 Discussion

The goal of our study was to explore and understand the practical tradeoffs for `CoverageImpact` and `PathImpact`. Overall, the results of the study show clearly that such tradeoffs do exist in practice and that there can be significant differences between the two techniques. In

Program	v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
CoverageImpact vs PathImpact										
NanoXml	**0**	**0**	**0**	**0**	**0**	-	-	-	-	-
Siena	0.0557	0.0547	**0.0308**	**0.0311**	**0.0303**	**0.0191**	**0.0185**	-	-	-
Jaba	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
CoverageImpact vs Uninstrumented										
Program	v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
NanoXml	**0**	**0**	**0**	**0**	**0**	-	-	-	-	-
Siena	0.6991	0.7042	0.7057	0.7040	0.7062	0.6876	0.6954	-	-	-
Jaba	0.3906	0.3094	0.2899	0.3056	0.2574	0.3298	0.2268	0.2139	0.3554	0.4211
PathImpact vs Uninstrumented										
Program	v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
NanoXml	**0**	**0**	**0**	**0**	**0**	-	-	-	-	-
Siena	**0.0235**	**0.0235**	**0.0125**	**0.0126**	**0.0124**	**0.0069**	**0.0069**	-	-	-
Jaba	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Table 6. ρ-values from t-tests on individual test execution time cost results, per program and version, for three dimensions of comparisons.

particular, `CoverageImpact` was considerably less expensive than `PathImpact` in terms of space cost and time overhead, whereas `PathImpact` was more precise than `CoverageImpact` in the impact sets computed.

A closer look at the data shows that, for the cases considered, the relative precision of the techniques did not appear to be related to the size of the programs. NANOXML and SIENA are of comparable size, but the relative precision of the techniques varied considerably for these two programs. Moreover, the relative precision varied across different versions of the same program. From these observations, we conjecture that, in addition to depending on the program and on the set of dynamic data considered, relative precision is also affected by the location of changes. To illustrate, consider again the example provided in Section 2.3, where we show a case in which `PathImpact` is more precise than `CoverageImpact`. For the same example, we would see no difference in precision were we to consider changes in methods other than D, such as changes involving method E or method A.

To further investigate this conjecture, we computed the impact sets for one version of JABA, the largest of our subject programs, for 2,695 changes, each consisting of a change in a single method. We then compared the impact sets calculated by each technique on each of these changes. The results confirmed our intuition: the relative difference in precision between the two techniques varied widely across changes, from 0 to 42.79. Being able to recognize the types of changes for which differences in precision are particularly high would aid further investigation of the reasons for such differences.

The results of our study also show that the size of the program, and especially the size of the execution traces,[2] affected to a large extent the differences with respect to cost in space and time for the two techniques. Because `CoverageImpact` requires almost no processing of the dynamic information, its time overhead tended to be negligible. (The reason why the relative overhead was not negligible for program NANOXML is because its execution times are so short — less than 50 milliseconds — that even the small overhead due to the storing of the bit vectors to disk becomes relevant.) Moreover, `CoverageImpact` requires no path information, and its space cost is always one bit per method in the program. The situation is different for `PathImpact`, for which the cost in both time and space tends to grow with the size of the traces. The test-suite execution times for JABA provide an example of this behavior, with times for `PathImpact` on the order of 15 hours.

To further study this observation, we performed impact analysis using the two techniques on a handful of executions that generated traces from one to two orders of magnitude larger than those considered for the experiments in this paper. Although the number of data points collected is too small to be statistically significant, these additional observations confirmed the trend observed in the presented data — that the cost in both time and space for `PathImpact` grows with the size of the traces.

5 Conclusion

In this paper, we have presented an empirical comparison of two dynamic impact analysis techniques, `CoverageImpact` and `PathImpact`, that perform impact analysis based on execution data gathered from the field, from operational profile data, or from test-suite executions. In the study, we compared the two techniques based on (1) the precision of the impact sets they compute, (2) their cost in terms of space, and (3) their cost in terms of time. The

[2]Due to space limitations, we do not report data about the size of execution traces in this paper. However, the reader can use execution times as an indicator of such size: in general, the longer a program runs, the larger is the trace it produces.

results of the study show that tradeoffs do exist for the two techniques, and that the techniques have complementary strengths and weaknesses.

Given these findings, we can draw several conclusions and outline several future research directions.

First, it seems that `PathImpact`, at least as currently defined, may not be practical for use with programs that generate large execution traces, due to its large time overhead. Part of the overhead observed in these studies is due to the use of an implementation that is not optimized for speed and, in theory, the technique may be made linear in the size of the traces [4]. Nevertheless, trace sizes can be in the order of several GB, and the (almost) constant cost required to process each element in the trace is not negligible.

Second, `CoverageImpact` appears to be practical for the programs that we considered in our experiment. However, for some programs and changes, the technique can be significantly less precise than `PathImpact`, which may result in wasted resources (e.g., in all cases in which maintainers need to consider the methods in the change sets for other tasks). Additional studies will be needed to assess the extent to which these findings generalize to larger programs.

Third, dynamic impact analysis techniques focus on behavior relative to specific program inputs, in contrast to static techniques that consider all possible behaviors. Empirical comparisons of `PathImpact` and `CoverageImpact` to an implementation of static slicing [6, 11] have shown that the latter can identify much larger impact sets than the former, and thus, in cases in which conservative estimates are not required, can lead to unnecessary expense. Similar comparisons involving other static slicing techniques will be needed to provide a complete picture of the tradeoffs.

Fourth, because `PathImpact` and `CoverageImpact` have such complementary cost benefits, an interesting area for future work is to investigate ways to combine them, to define a hybrid approach that is not as expensive as `PathImpact`, yet is more precise than `CoverageImpact`. For example, static analysis could be used to identify groups of methods that may cause imprecision and collect only partial traces for those methods.

Finally, our study has examined constructs directly measuring the time, space, and precision costs associated with dynamic impact analysis techniques themselves. Ultimately, however, the practical potential of these techniques must be assessed in terms of the engineering approaches they are intended to assist, approaches such as predictive risk assessment and regression testing guidance.

Acknowledgements

This work was supported in part by National Science Foundation awards CCR-0306372, CCR-0205422, CCR-9988294, CCR-0209322, and SBE-0123532 to Georgia Tech, and by the NSF Information Technology Research program under Award CCR-0080900 to Oregon State University.

References

[1] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the International Conference on Software Maintenance*, pages 292–301, Sept. 1993.

[2] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Press, Los Alamitos, CA, USA, 1996.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computing Systems*, 19(3):332–383, August 2001.

[4] J. Larus. Whole Program Paths. In *Proceedings of SIGPLAN PLDI 99*, pages 1–11, Atlanta, GA, May 1999. ACM.

[5] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, Nov. 2003.

[6] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering*, pages 308–318, May 2003.

[7] J. L. Lions. ARIANE 5, Flight 501 Failure, Report by the Inquiry Board. *European Space Agency*, July 1996.

[8] J. P. Loyall, S. A. Mathisen, and C. P. Satterthwaite. Impact analysis and change management for avionics software. In *Proceedings of the IEEE National Aeronautics and Electronics Conference, Part 2*, pages 740–747, July 1997.

[9] C. Nevill-Manning and I. Witten. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the IEEE Data Compression Conference*, pages 3–11, 1997.

[10] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 128–137, Sept. 2003.

[11] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metadata to support the regression testing of component-based software. In *Proceedings of the International Conference on Software Maintenance*, pages 716–725, November 2001.

[12] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, Englewood Cliffs, NJ, 1998.

[13] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79, Sept. 1990.

[14] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the ACM Worshop on Program Analysis for Software Tools and Engineering*, pages 46–53, Oct. 2001.

[15] M. D. Scheemaecker. NanoXML: A small XML parser for Java. http://nanoxml.n3.net, 2002.

[16] R. J. Turver and M. Munro. Early impact analysis technique for software maintenance. *Journal of Software Maintenance*, 6(1):35–52, Jan. 1994.