D^{2}ABS: A Framework for Dynamic Dependence Analysis of Distributed Programs

Haipeng Cai and Xiaoqin Fu

Abstract—As modern software systems are increasingly developed for running in distributed environments, it is crucial to provide fundamental techniques such as dependence analysis for checking, diagnosing, and evolving those systems. However, traditional dependence analysis is either inapplicable or of very limited utility for distributed programs due to the decoupled components of these programs which run in concurrent processes at physically separated machines.

Motivated by the need for dependence analysis of distributed software and the diverse cost-effectiveness needs of dependence-based applications, this paper presents D^{2}ABS, a framework of dynamic dependence analysis for distributed programs. By partially ordering distributed method execution events and inferring causality from the ordered events, D^{2}ABS computes method-level dependencies both within and across process boundaries. Further, by exploiting message-passing semantics across processes, and incorporating static dependencies and statement coverage within individual components, D^{2}ABS offers three additional instantiations that trade efficiency for better precision. We present the design of the D^{2}ABS framework and evaluate the four instantiations of D^{2}ABS on distributed systems of various architectures and scales using our implementation for Java. Our empirical results show that D^{2}ABS is significantly more effective than existing options while offering varying levels of cost-effectiveness tradeoffs. As our framework essentially computes whole-system run-time dependencies, it naturally empowers a range of other dependence-based applications.

1 INTRODUCTION

In response to scientific and societal demands on the scalability of data storage and computation, a rising number of modern software systems are *distributed* by design [1] to leverage decentralized, high-performance computing infrastructures and resources. In fact, most critical software and services today, such as financial systems, web search, airline services, and medical networks, are all distributed systems in nature [2]. The quality (e.g., reliability and security, among other quality factors) of these systems is thus of paramount importance. Significant advances in these regards have been made in the areas of parallel and distributed computing, yet mainly from *coarse*, *system-level* perspectives such as those of architecture, networking, and resource management. In contrast, *code-level* quality via deeper analyses of programs in distributed systems has not been much studied, and there is a lack of tool support for code quality assurance for distributed software.

To attain and sustain various quality factors of distributed systems, it is crucial to model and reason about the complex interactions among code entities via *dependencies* in these systems. Historically, dependence analysis has been a foundational methodology that underlies a wide range of code-based software engineering tasks and associated techniques [3], [4], [5]. Generally, a dependence analysis can be static or dynamic (either purely dynamic or mixed with some static analysis).

Static dependence analysis is a core static analysis that computes and reasons about *possible* dependencies among program entities. As it attempts to produce results that hold for all possible execution scenarios, it is known to be imprecise in general. For distributed systems, it has to be even more conservative, hence is subject to even greater imprecision, due to the *implicit* interactions among the components of these systems that are decoupled by networking facilities [6]. As a result, there are no *explicit* references or invocations among code entities across those components [7], [8], on which existing dependence analysis approaches typically rely. In consequence, existing approaches are largely limited to centralized software which mostly runs in a single process (whether the program is single- or multi-threaded) hence has explicit interactions (via invocations or references) among code entities. There is also no trivial adaptation of existing static dependence analyses to distributed programs [9], [10], [11].

In contrast, *dynamic* dependence analysis computes and reasons about dependencies among program entities that are *exercised* in a particular execution of the program. It has two advantages relative to static dependence analysis. First, dynamic analysis has a generally better potential for discovering implicit relationships among code entities because it looks at concrete executions of the code. For instance, implicit calling relationships via reflection in Java can be readily resolved by dynamic analysis which simply identifies the call targets actually invoked. The second merit lies in its greater precision than static approaches [12], because it is only concerned about dependence relations with respect to specific program executions [13] instead of considering all possible ones. For developers working with these concrete executions (e.g., those driven by regression tests), dynamic dependence analysis can be a powerful technique to assist them with development and maintenance tasks (e.g., selecting/prioritizing regression tests that cover program entities dependent on changed locations), as it narrows down the search space of complex dependency relations to the focused executions of a program. Therefore, in this paper, we focus on dynamic dependence analysis of distributed systems.

However, developing a dynamic dependence analysis for distributed programs faces multiple challenges. The first challenge is that, despite its generally greater potential for resolving implicit dependencies, *it remained unknown how exactly the implicit code*

- *Haipeng Cai and Xiaoqin Fu are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA. E-mail: haipeng.cai@wsu.edu, xiaoqin.fu@wsu.edu*

Manuscript received April 1, 2018; revised August 26, 2015.

dependencies among distributed components may be inferred from distributed program executions. The majority of current approaches to dynamic analysis of distributed program dependencies (e.g., [14], [15], [16]) were designed only for procedural programs [15]. For distributed object-oriented programs, *backward* dynamic slicing algorithms have been developed, yet it is still unclear whether they can work with real-world systems [14], [15]. Also, for many maintenance and evolution tasks (e.g., impact analysis), *forward* dependence analysis would be needed instead (e.g., given a query, entities that transitively depend on the query are considered potentially impacted).

As the second challenge, dynamic dependence analysis is subject to a common, fundamental difficulty with balancing the analysis cost and effectiveness, evidenced by relevant prior works on centralized programs [12], [13], [17]. To deal with its substantial analysis overheads, dynamic analysis has been mostly studied in two directions. In one direction, efficient analysis often provides fast answers to dependence queries [18], [19], yet with coarse results that require tremendous efforts for post-analysis inspection [20], [21], [22]. In the other direction, highly precise analysis saves developers' inspection efforts by producing few false positives, yet incurring analysis overheads that may not be practically affordable [23], [24]. Frameworks offering better cost-effectiveness tradeoffs along with more such tradeoff options also have been studied recently [12], [25]. However, the frameworks were developed for centralized, single-process software, without a straightforward path to extending them for distributed programs [26]. Given their typically large size, great complexity, and long executions, *distributed systems represent a software domain where achieving good cost-effectiveness balances in dynamic dependence analysis is even more challenging*.

The third challenge lies in the practicality barriers induced by the distributed nature of the program under analysis. As other dynamic analyses, a dynamic dependence analysis has to collect certain forms of execution data of the program before analyzing them. System-level approaches to data collection (i.e., using dynamic instrumentation or customized run-time environment) are non-intrusive but would cause portability issues. Purely application-level approaches (i.e., analyzing the application itself without any run-time environment modification) are much more portable, yet they could also be intrusive as they commonly use static instrumentation to collect execution data needed, which may cause interference with the execution of the original program. In fact, in our early effort [27], we experienced substantial difficulties in making such static instrumentation work with real-world distributed software. For example, common ways of probing for communications among distributed components led to abnormal system behaviors in network I/O [26] hence system crashes. As it turned out, *designing non-interfering probing for application-level dynamic analysis for distributed software is not trivial*.

To address these challenges, we develop D^{2}ABS[1], a framework of dynamic dependence analysis for the most commonly deployed kind of distributed systems—where components communicate through message passing via standard socket facilities and/or their encapsulations. By exploiting the happens-before relation [28] between method execution events and the semantics of message passing among distributed components, the framework abstracts dynamic dependencies to method level for a given system and its executions both within and across its concurrent processes.

1. D^{2}ABS stands for Distributed program Dependence ABStraction.

Our approach offers *rapid* results that are safe [29] relative to (i.e., guaranteed to hold for) the analyzed executions, while relying on neither well-defined inter-component interfaces nor message-type specifications needed by peer approaches (e.g., those for distributed event-based systems—DEBS [7], [9], [10]).

Further, utilizing intra-component *static* dependence analysis [17], [30] that incorporates threading-induced and exception-driven dependencies, and whole-system *dynamic* statement coverage data, D^{2}ABS trades efficiency for precision by pruning false-positive dependencies. As a whole, the framework unifies four dynamic dependence analysis algorithms each providing a distinct level of cost-effectiveness balance. D^{2}ABS naturally empowers dynamic impact analysis of distributed programs that accommodates different usage scenarios with varying demands for precision and budgets for analysis and inspection costs.

We evaluate D^{2}ABS on eight distributed Java software, including six enterprise-scale systems, and demonstrate that it is able to work with large, complicated distributed systems using blocking and/or non-blocking (e.g., selector-based [31]) communication. In the absence of more advanced prior peer techniques, we use our coarse dynamic dependence analysis purely based on control flows [26] as the baseline, and measure the effectiveness of the three more precise abstraction techniques against it in the application context of dynamic impact analysis. We also gauge the costs of each technique, including those for different phases of our framework (static analysis, run-time profiling, and post-processing for dependence querying), and compare against the baseline.

Our results show that, without using static dependencies or dynamic coverage data, D^{2}ABS can reduce the size of baseline impact (dependence) sets to be inspected by over 15% on average. The average cost was 73 seconds to finish the one-time instrumentation and 61 milliseconds for answering a query, with run-time overhead of 8%. Incorporating static dependence analysis in our framework largely further reduced baseline impact (dependence) sets by 41% on average, at the cost of reasonable additional overheads for most of our subject systems. Exploiting whole-system statement coverage additionally even further enhanced the precision of our analysis (54% impact-set reduction of baseline results), which causes generally negligible increases in total analysis overheads compared to just incorporating static dependencies.

The development of D^{2}ABS started with our preliminary work DISTIA [26], where we presented the first two levels of abstraction (i.e., based on method-level control flows with/without message-passing semantics exploited to enhance effectiveness). Technically, this paper represents a substantial expansion of that earlier work by developing two more levels of abstraction, using *method-level static* dependencies with/without *run-time statement-level* coverage data. Experimentally, we expand the scale of empirical study by including two more real-world subject distributed programs and examining an additional research question (on variable cost-effectiveness). We used the basic abstraction in [26] as the baseline (instead of using all methods covered in an execution as baseline impacts) in assessing the cost and effectiveness of these newly added levels of abstraction. We also conducted extensive statistical analyses to understand how various forms of program information used in our framework contribute to its effectiveness.

The main contributions of this work include:

- A framework of dynamic dependence analysis for distributed programs that unifies four analysis algorithms, each providing a distinct level of cost-effectiveness tradeoff.
- An extensive evaluation of our framework for dynamic im-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

```
1  public class S {              /* the Server component */
2    Socket ssock = null;
3    public S (int port) {
4      ssock = new Socket (port);
5      ssock.accept();
6    }
7    char getMax(String s) {...}
8    void serve() {
9      String s = ssock.readLine();/* message receiving */
10     char r = getMax(s);
11     ssock.writeChar(r);           /* message sending */
12   }
13   public static int main(String[] a) {
14     S s = new S(33);
15     s.serve();
16     return 0;
17   }
18 }
19
20 public class C {              /* the Client component */
21   Socket csock = null;
22   public C (String host, int port) {
23     csock = new Socket (host,port);
24   }
25   void shuffle (String s) {...}
26   char compute (String s) {
27     shuffle(s);
28     csock.writeLine(s);          /* message sending */
29     return csock.readChar();  /* message receiving */
30   }
31   public static int main (String[] a) {
32     C c = new C ('localhost', 33);
33     System.out.println( c.compute(a[0]) );
34     return 0;
35   }
36 }
```

Fig. 1: An example distributed program D consisting of two components: S (server component) and C (client component).

pact analysis against subject programs of varied sizes and domains that shows its promising effectiveness and scalability, as well as flexible cost-effectiveness tradeoffs.

- An empirical investigation of how the use of various forms of data in the framework affects its cost-effectiveness.
- An *open-source* implementation of the framework that works with diverse, large enterprise distributed systems with either or both of blocking and non-blocking communication.

In the rest of this paper, we first motivate our work while introducing a running example for illustration purposes in Section 2. To present our technical approach, we start with the definitions of dependence analysis related terms in Section 3. Then, we give an overview in Section 4, followed by details on the design, instantiations, and implementation of the D^{2}ABS framework presented in Sections 5, 6, and 7, respectively. We evaluate D^{2}ABS in Section 8, where we describe our experimental setup and methodology before discussing empirical results, and then discuss additional issues with our evaluation and the use of our technical approach in Section 9. Prior work related to D^{2}ABS is discussed in Section 10, before we give concluding remarks in Section 11.

2 Motivating and Working Example

In this section, we illustrate a motivating use scenario of dynamic dependence analysis with a simple distributed program as an example, for its application in change impact analysis.

When maintaining and evolving a distributed program which consists of multiple components, the developer needs to understand potential change effects not only in the component where the change is proposed, but also in all other components. By design, the components that constitute a distributed system are decoupled as a result of *implicit* invocations and/or references among them, realized via networking-based message passing. This design paradigm, however, greatly reduces the utility of existing dependence analysis and its client analysis techniques (e.g., impact analysis). Consider the program D of Figure 1, which consists of a server and a client component, implemented in classes `S` and `C`, respectively. The client retrieves the largest character in a string by delegating the task to the server (Lines 26-28), which finishes the task and sends the result back to the client (Lines 8-12).

Suppose the developer proposes to change `S::getMax` as part of an upgrade plan for the server, and thus needs to determine which other parts of the program may also have to be changed. Having an available set I of inputs, the developer wants to perform a dynamic impact analysis to get a safe estimation of the potential impacts of the candidate changes with respect to the program executions against I. Note that static approaches would be largely disabled by the implicit communication between these two components (via message passing, which is in this case realized through a network socket).

To accomplish this task, method-level dynamic impact analyses of varying cost-effectiveness tradeoffs (e.g., [19], [22], [25]) seem able to offer the developer with many options. However, since there are no explicit dependencies between `S` and `C`, existing approaches would predict impacts within the *local* component (i.e., where the changes are located; `S` in this case) only. In consequence, the developer would have to ignore impacts in *remote* components (`C` in this case), or make a worst-case assumption that all methods in remote components will be impacted.

One major difficulty here is the lack of explicit invocations or references among the two decoupled components [8], [9], [32], whereas traditional, dependence-based approaches often rely on such explicit information to compute dependencies for impact prediction. Lately, various analyses that are not based on code dependencies have also been proposed [7], [8], [10]. While efficient for *static* impact analysis, these approaches are limited to systems of special types such as distributed *event-based* systems (DEBS) [33], or rely on specialized language extensions like EventJava [34]. Other approaches are potentially applicable in a wider scope yet depend on information that is not always available, such as execution logs of particular patterns [35], or suffer from overly-coarse granularity (e.g., class-level) [7], [10], [35] and/or unsoundness [6], in addition to imprecision, of analysis results.

This example illustrates the need for a dynamic impact analysis of distributed programs at method level, a specific application of dynamic dependence analysis. Also, developers would need varied tradeoffs between precision and overhead of such analyses in different task scenarios [13], [25]. For example, if the developer aims at a *quick, high-level* understanding about the system behavior (concerning how `S::getMax` interacts with other methods across the system), a fast analysis with relatively low precision would be desirable than a precise yet much slower analysis. Yet, given a sufficient time budget, if the developer is tasked to fix a bug in `S::getMax`, it would be more desirable to have higher precision so that the developer only needs to inspect a few methods when deciding how to apply the bug fixes in `S::getMax` itself and necessary changes in other methods. Earlier studies have suggested such diverse needs when performing impact analysis (e.g., due to various types of change requests [36]) and its application tasks (e.g., due to varied usage scenarios [37]); Another common reason lies in varying resource (e.g., time) budget constraints developers are subject to in conducting these tasks [23], [36].

Thus, more generally, a framework of dynamic dependence analysis that offers various levels of cost-effectiveness would be required to support a wide range of (dependence-based) tasks in developing and maintaining distributed software. In the rest of this paper, we demonstrate how to address these challenges and needs with our framework D^{2}ABS. We also use the program D in many working examples throughout the paper to illustrate the overall design ideas and detailed inner workings of our framework.

3 PRELIMINARIES

This paper addresses the topic of program dependence analysis, particularly focusing on dynamic dependence analysis of distributed programs. Thus, it is helpful to explicitly define what the various kinds of dependencies to be computed are. These definitions provide the basis of all the dependence inference algorithms offered by our D^{2}ABS framework.

D^{2}ABS addresses code dependencies at the method level, which are defined based on the classical notions of control and data dependencies [3] often defined at statement level by default.

Definition 1. *Given two statements s_1 and s_2 in a program, s_2 is* data dependent *on s_1 if s_2 may read (use) a variable v that is written (defined) at s_1.*

Definition 2. *Given two statements s_1 and s_2 in a program, s_2 is* control dependent *on s_1 if s_1 computes a branching decision that determines whether s_2 may be executed or not.*

More formal definitions of data/control dependencies can be found in [4]. Based on these statement-level dependencies, the method-level dependencies can then be defined as follows.

Definition 3. *For two arbitrary methods m_1 and m_2 in a program, m_2 is* method-level dependent *on m_1 if there exist a statement s_1 in m_1 and another statement s_2 in m_2 such that s_2 is data or control dependent on s_1. Without loss of generality, we treat formal parameters of a method as defined at the entry of the method and as data dependent on the corresponding actual parameter at each call site for that method. In other words, m_2 is data dependent on m_1 if m_1 defines a variable that m_2 might use, while m_2 is control dependent on m_1 if m_1 computes a decision that may determine whether m_2 (or part of it) executes [17].*

We consider a code dependence to be either a data or control dependence, each defined broadly—other kinds of dependencies essentially fall in either category (e.g., interference dependencies as a kind of data dependence and synchronization dependencies as a kind of control dependence [38]). In general, there are four kinds of data dependencies: flow (true) dependence, anti-dependence, input dependence, and output dependence. We focus on flow/true dependencies while disregarding the other kinds of data dependencies as in prior works on program dependence modeling and analysis [4], [38], [39]. Also, while control dependencies were differentiated as strong versus weak in [4], we focus on strong control dependencies as later in [39]. Then, we have the following definition for the general dependence notion.

Definition 4. *Given two program entities e_1 and e_2, whether they are statements, methods, or components, e_2 is* dependent *on e_1 if there is any data or control dependence of e_2 on e_1.*

Here a component is a particular kind of (coarse-grained) program entity which is common in a distributed program.

Definition 5. *In a distributed program, a* component *is the collection of code that runs in a separate process from the rest of the program.*

We further define a few other dependence related terms according to the different scopes of code dependencies.

Definition 6. *Suppose a statement s_2 is dependent on a statement s_1. If s_1 and s_2 are in the same method, the dependence is* intraprocedural*; otherwise, the dependence is* interprocedural*.*

Definition 7. *Given two program entities e_1 and e_2, whether they are statements or methods, suppose e_2 is dependent on e_1. If e_1 and e_2 are in the same thread, the dependence is* intra-thread*; otherwise, the dependence is* interthread*. Interthread dependencies are induced by multithreading constructs in a program, thus we also refer to them as* threading-induced *dependencies.*

Definition 8. *Given two program entities e_1 and e_2, whether they are statements or methods, suppose e_2 is dependent on e_1. If e_1 and e_2 are in the same component, the dependence is* intra-component*; otherwise, the dependence is* intercomponent*.*

By default, program dependencies are *static*, meaning that they are defined with respect to any possible execution of the program. In contrast, dynamic dependencies are defined with respect to a particular, concrete program execution.

Definition 9. *Given two program entities e_1 and e_2, whether they are statements, methods, or components, if e_2 is dependent on e_1 and that dependence is exercised in a concrete execution X of the program against a particular run-time input I_X, then e_2 is* dynamic(ally) dependent *on e_1 (with respect to X or I_X).*

Then, in accordance with different kinds of static dependencies, we have the corresponding kinds of dynamic dependencies as well. In particular for distributed programs, dependencies across process boundaries can be defined as the dynamic version (projection) of dependencies across components.

Definition 10. *Given two components c_1 and c_2 in a distributed program running in an execution X as two processes p_1 and p_2, respectively, if c_2 is dynamic(ally) dependent on c_1 with respect to X, then p_2 is* interprocess dependent *on p_1 (with respect to X). If c_1 and c_2 are the same components, then p_2 is* intra-process dependent *on p_1 (with respect to X).*

Finally, we define terms concerning the original input and ultimate output of a dependence analysis as follows:

Definition 11. *Given a program entity e as the input, whether it be a statement, method, or component, a (static or dynamic) dependence analyzer produces as the output the set DS of entities of the program that are found to be dependent on e. The input entity e is the* dependence query *(or simply* query*) and the corresponding output set DS is the* dependence set *of the query.*

4 D^{2}ABS OVERVIEW

This section gives an overview of our D^{2}ABS framework, focusing on its architecture, workflow, configuration, and application scope.

4.1 D^{2}ABS Architecture

Figure 2 delineates the high-level conceptual (layered) architecture of our framework, elucidating its key design elements and their relationships. The bottom layer consists of a set of static **code**

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

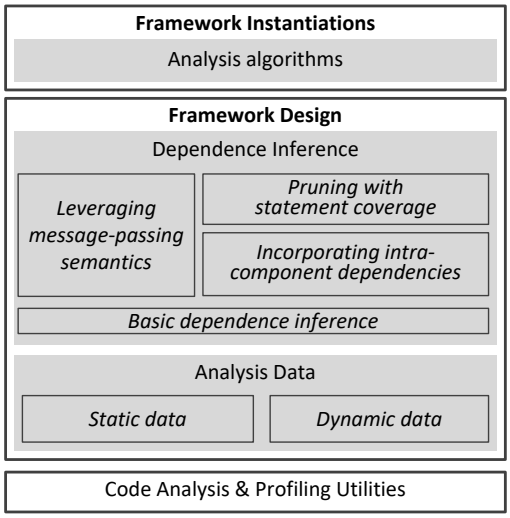


Fig. 2: The architecture of D^{2}ABS highlighting its key design elements, including *analysis data* used and the underlying *dependence inference* rules followed by various *instantiations* of the framework. Each layer relies on the layer immediately below it.

analysis and profiling utilities (e.g., bytecode parser and instrumenter) that enable the computation of, and actually purvey, the various forms of analysis data about the distributed program under analysis as utilized by the framework. The data includes *static data* directly computed from the program's code (for example, the static dependence of `C::compute` on `C::shuffle` in the client component), and *dynamic data* about the program's execution (for example, the event of a program D's execution entering `C::shuffle` at Line 25, or that of the execution returning into `C::compute` at Line 29 after invoking `Socket::readChar`).

By using these analysis data in different, combinatorial ways, the framework infers dynamic dependencies through different dependence inference rules. Among these rules, *basic dependence inference* provides the basic form of dependence approximation solely based on the execution order among methods. For example, the dynamic dependence of `S::getMax` on `C::shuffle` is inferred simply from the fact that `S::getMax` executed after `C::shuffle`. On top of this basic inference rule, *leveraging message-passing semantics* additionally requires at least one occurrence of message passing between two processes to infer the dynamic dependence between two methods executed across the two processes. For example, the dynamic dependence of `S::getMax` on `C::shuffle` is inferred from both the execution order between them and the fact that the client process (in which `C::shuffle` is executed) did communicate with the server process (in which `S::getMax` is executed).

Also on top of the basic inference rule, *incorporating intra-component dependencies* additionally requires two methods within the same component to have static dependence in order to affirm dynamic dependence between them. For example, the dynamic dependence of `C::compute` on `C::shuffle` is inferred from both the execution order between them (i.e., `C::compute` executed after `C::shuffle`) and the fact that within the client component there is indeed static dependence of `C::compute` on `C::shuffle`. Finally, on top of the *incorporating intra-component dependencies* rule, *pruning with statement coverage* further requires the statements that are actually responsible for the static dependence between the two methods to be covered in order to infer the dynamic dependence between them. For example, suppose there is a condition that guards against the call to `C::shuffle` at Line 27 and this condition is evaluated false during the execution of D being considered, then the call statement would not be covered hence the dynamic dependence of `C::compute` on `C::shuffle` would not be affirmed.

In Figure 2, the boxes representing the four dependence inference rules are spatially placed so as to indicate the semantic relationships among the rules. Together with the analysis data, these dependence inference rules form our **framework design**.

Based on this design, D^{2}ABS spawns various **framework instantiations** that each corresponds to a concrete dynamic dependence analysis algorithm for distributed programs, by using different combinations of static/dynamic data and applying different dependence inference rules. Accordingly, each of these instantiations is expected to have a level of costs (e.g., time for producing the analysis data and for computing dependencies with the data) and a level of effectiveness (e.g., precision) that are both different from those offered by others. For example, producing static dependencies among methods within each of the two components of program D and using these static dependencies for computing dynamic dependencies in an execution of D would contribute to the precision and total cost of the dynamic dependence analysis, *differently* from what producing and using the statement coverage data would do. On the other hand, these instantiations compute dynamic dependencies through a set of unified analysis algorithms. These algorithms are abstracted as the unified workflow of D^{2}ABS, as described next (Section 4.2).

Details on the design of our framework are presented in Section 5, including analysis data and dependence inference, given in Sections 5.1 and 5.2, respectively. Details on the instantiations of our framework are presented in Section 6, including key analysis algorithms given in Section 6.1.

4.2 D^{2}ABS Workflow

The overall workflow of our framework is depicted in Figure 3, where the three primary **user inputs** are the program Y under analysis, a set I of program inputs for Y, and a query set M. An *optional* input, a message-passing API list L can also be specified to help D^{2}ABS identify program locations for profiling inter-process communications during the execution of Y driven by I. In practice, typically this list would not be necessary since the framework implementation for a language (e.g., Java) could handle most commonly used network I/Os for message passing (e.g., in Java distributed software) as a built-in feature. This built-in feature would suffice for precisely and completely capturing inter-process communications as required for the dynamic dependence analysis in our framework. The **D^{2}ABS output** is a set of dependencies of M (i.e., the set of methods dynamically dependent on any method in M) computed from the given inputs in **four steps** as annotated in the figure and described below.

As mentioned earlier, the key motivation for D^{2}ABS is to enable dynamic dependence analysis for distributed programs while offering varied tradeoffs between analysis precision and efficiency. To that end, D^{2}ABS utilizes both static and dynamic data about the program. These data are of different forms each coming with a different level of *cost* (i.e., the time cost incurred by generating and utilizing the data) and *benefit* (i.e., the data's contribution to the dependence analysis effectiveness). At a high level, the four steps of D^{2}ABS are about *generating* these data (**Step** ❶,❷,❸) and then *utilizing* them for dependence computation (**Step** ❹).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

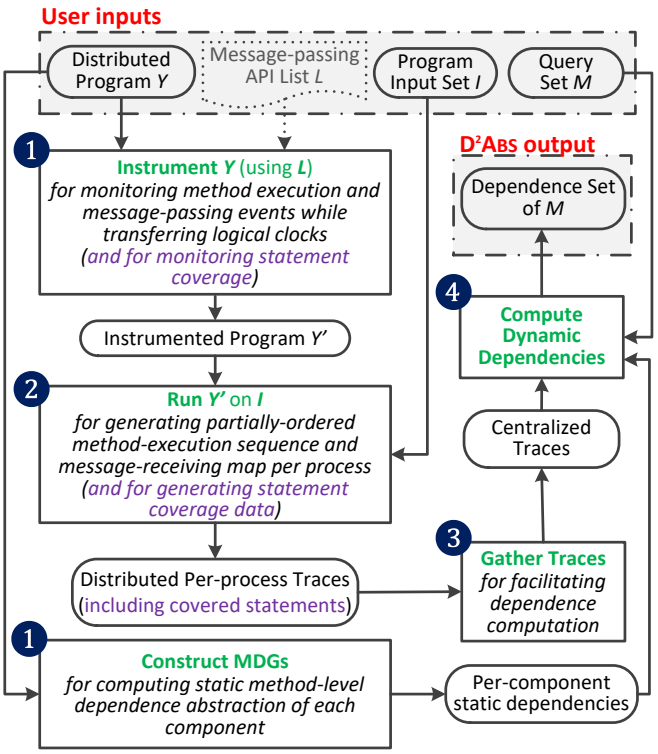


Fig. 3: The unified workflow of D^{2}ABS (consisting of four key steps as numbered), including its inputs and outputs.

The workflow of D^{2}ABS as regards how these steps collaborate is elaborated below.

The *first step* (**Step** 1) of D^{2}ABS has two aims: (1) computing the only kind of *static data* (static dependencies per component) through a static analysis of Y, and (2) preparing for collecting different forms of *dynamic data* (method execution events, message-passing events, and statement coverage) through static instrumentation of Y. The results of this step are the per-component static dependence graphs and the instrumented program Y'.

The *second step* (**Step** 2) aims to profile the dynamic data that is probed for in the previous step. Specifically, this is done by executing Y' on the given input set I, during which all the dynamic data instrumented are generated and collected (serialized into respective disk files). An additional item of data, *message-receiving map*, is derived from the message-passing events and serialized as part of the resulting execution trace. The map records for each process the timestamp when it receives the first message from any other processes. The tracing in each process is performed concurrently with that in others during the system execution.

In the *third step* (**Step** 3), the execution traces separately generated in individual processes are gathered to the machine where the dependence computation is to be performed. This assisting step is necessary as the dependence computation in D^{2}ABS is offline and centralized, while the tracing is performed in a distributed manner.

In the *fourth step* (**Step** 4), D^{2}ABS takes all the static and dynamic data as inputs to compute the dependence set of the query set M, using a unified dependence analysis algorithm. The algorithm automatically chooses different combinations of the collected data and utilizes different dependence inference rules accordingly, as per user configurations.

Illustration. For the example program D of Figure 1, D^{2}ABS first constructs an intra-component static dependence graph for the server and client component separately. For example, the client's graph includes the dependence of `C::main` on `C::compute` and that of `C::compute` on `C::shuffle`, while the server's graph includes the dependence of `S::serve` on `S::getMax` and that of `S::main` on `S::serve`.

It then instruments at each message-passing API (e.g., `Socket::readLine` and `Socket::writeLine`) and every other method call (e.g., the call to `shuffle` in `C::compute`). The instrumentation also probes for all branches in order to gather statement coverage data at runtime. Except for branches induced by ordinary conditional statements, the entry point of each method (e.g., prior to Line 14 in `S::main`) is also treated as a branch so that statement coverage can be inferred from covered branches in a consistent way (e.g., if the entry point of `S::main` is covered, we infer that all the statements in this method that are not guarded by any predicates are covered).

In the second step, D^{2}ABS runs the instrumented version of D against a client input—the server does not use the user input. During this execution, the events about message passing (e.g., the message including the string s is sent from the client to the server at Line 28 after the call `csock.writeLine(s)`) and the events about method invocation (e.g., the execution enters `C::shuffle`, and then returns into `C::computer`) are monitored. The message-receiving map here records the timestamp at which the server process received the first message from the client (e.g., 9) and the timestamp at which the client process received the first message from the server (e.g., 13). At the end of the execution, on the server and client machine separately, D^{2}ABS serializes the sequence of synchronized method events in the respective process (e.g., the event of entering `S::main` at timestamp 0 followed by the event of entering `S::S` at timestamp 1 in the server process) into a trace file; it then computes and serializes into a disk file the statement coverage in each process (e.g., statements 2 through 16 in the server process), and dumps the map to yet another disk file.

In the third step, D^{2}ABS gathers the two method event trace files to a machine where its dynamic dependence computation is performed (e.g., the server machine). In the fourth step, the two static dependence graphs (one for the server component and the other for the client component), the two statement coverage files, and the message-receiving map are also gathered to that machine. Now, depending on which of the four dependence inference rules is adopted, D^{2}ABS computes, for a given query (e.g., `S::getMax`), the dynamic dependence set accordingly (e.g., {`S::serve`, `S::main`, `C::compute`, `C::main`} with the *basic dependence inference* rule).

4.3 D^{2}ABS Configuration

A main merit of our framework lies in its offering variable and flexible cost-effectiveness tradeoffs in dependence analysis. This merit enables D^{2}ABS to accommodate varying developer needs (e.g., due to their different precision expectations and cost budgets). In essence, the flexibility of D^{2}ABS is attributed to its customizability, realized via varied configurations. As depicted in Figure 3 (via notes in the parentheses), the framework allows users to choose which data to be utilized by the dependence analysis algorithm. For example, the user may choose not to utilize statement coverage in the analysis. Then, accordingly, the instrumentation and profiling (in **Steps** 1 **and** 2, respectively) for statement coverage would be skipped. Similarly, if the user does not want to

use static dependencies in the dynamic dependence analysis, the static (dependence) analysis in **Step** ❶ would be ignored.

Due to the flexible configuration, D^{2}ABS can be instantiated as different dynamic dependence analyzers each offering a potentially distinct level of cost-effectiveness. Some instantiations have been realized, as described in Section 6. Moreover, rather than a tool itself, D^{2}ABS as a framework focuses on computing dependencies which enable a range of practical client/application tools to address particular quality problems with distributed systems (e.g., correctness and security as exemplified in Section 10).

4.4 Application Scope

D^{2}ABS targets the most common type of distributed systems, as opposed to specialized ones such as DEBS and others relying on special inter-component interfaces (e.g., RMI—remote method invocation [40]) or message-type specifications (e.g., CORBA–common object request broker architecture [41]). As exemplified by the running example program D of Figure 1, a *common distributed system* has three characteristics according to the definition in [1]: (1) the system consists of multiple components located at networked computers, (2) these components communicate and coordinate their actions only by passing messages, and (3) the components run concurrently in multiple processes without a global clock. These systems represent the applicability scope of D^{2}ABS, although D^{2}ABS only deals with the software part of such systems (as opposed to system-level concerns such as distributed computing infrastructure/platform and resource management), referred to as *distributed software*. More specifically, as we focus on code analysis, the actual analysis scope of D^{2}ABS is the program (code) of distributed software, referred to as *distributed program*.

We further confine as a *component* the code entities that runs in a separate *process* from the rest of the system, where each process may host one or multiple *threads*. For example, the two components in the example program D, `server` and `client`, each runs in a separate process and they communicate only through message passing (via a network socket). These components are marked for illustration purposes—D^{2}ABS does not recognize functionality roles of components/processes; it identifies probing points for inter-process communication (IPC) events based on the given list of message-passing APIs.

5 The D^{2}ABS Framework Design

In this section, we elaborate the framework design of D^{2}ABS, as shown in Figure 2. The two major design elements that constitute the framework are the analysis data utilized for dynamic dependence analysis of distributed programs, and the dependence inference rationales (rules) according to which dynamic dependencies are derived from the analysis data.

5.1 Analysis Data

This section presents the various forms of program data utilized by D^{2}ABS for its dependence analysis of variable cost-effectiveness. As noted earlier, D^{2}ABS uses one form of static, and three forms of dynamic, data in its analysis, justified by the different levels of cost and benefit of each of these forms of data which can contribute to the variable cost-effectiveness D^{2}ABS aims at.

5.1.1 Static Data

To trade analysis cost for better precision (than purely dynamic dependence analysis), D^{2}ABS exploits the static dependencies within each component of the distributed program under analysis (noted as *intra-component dependencies*). Treating each component as a single-process program, traditional dependence analysis can be utilized for this purpose. However, immediately adopting the traditional dependence model would compromise the scalability of D^{2}ABS, given the known heavyweight nature of computing fine-grained (statement-level) dependencies [12], [22], [42]. Thus, we employ the method-level dependence analysis algorithm developed for single-process Java software [17], [30] to compute the static dependencies among methods. Here we summarize the key ideas of this abstraction algorithm (for intra-thread dependencies) and then discuss its extension for including threading-induced (i.e., interthread) dependencies.

Intra-thread dependencies. Specifically, for each component c, the *method dependence graph* MDG_c is constructed to represent the static dependencies among methods, with intraprocedural dependencies abstracted as summaries [30]. Each method of c is represented as a node of MDG_c, where each edge represents data or control dependence between two methods. The intraprocedural dependence summaries for each method are represented by a mapping from incoming to outgoing flows, as a result of an intraprocedural reachability analysis on the procedure dependence graph [3] of the method. For the sake of scalability, interprocedural dependencies are computed in a flow-insensitive manner (although at the cost of compromising precision). Further, since we do not perform any analysis of static dependencies alone but use them as contexts for dynamic dependence refinement, the computation of interprocedural dependencies in MDG_c is also context-insensitive. Discarding context- and flow-sensitivity largely enhances the scalability of the static intra-component dependence analysis [17], hence potentially that of D^{2}ABS when incorporating the static dependence information in the dynamic dependence analysis.

Threading-induced dependencies. Realistic distributed systems typically run multiple threads in each component (process). Therefore, D^{2}ABS also includes threading-induced dependencies in its static intra-component dependence computation. In particular, two additional types of control dependencies, *synchronization dependencies* and *ready dependencies* [43], and an additional type of data dependence, *interference dependencies* [38], are considered in our framework. For Java programs, *synchronization dependencies* are induced by the use of synchronization blocks while *ready dependencies* are due to the `wait-notify` synchronization mechanism. *Interference dependencies* are generally a result of data sharing between threads, caused by one thread using a variable defined in another thread. Each of these three types of dependencies lies between two different methods across two different threads. For each component c, D^{2}ABS adds these interthread dependencies (each as an edge) to the MDG_c computed for c in the static dependence analysis step described above.

Given the prevalence of exception-handling constructs in modern languages (e.g., Java), D^{2}ABS considers static dependencies induced by exceptional control flows [39] (e.g., those via `try/catch` and `finally` blocks in Java).

Illustration. For example, in the program D of Figure 1, the server component is composed of one class S which includes four methods (`S::S`, `S::main`, `S::getMax`, and `S::serve`).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

TABLE 1: A full sequence of (i.e., instance-level) method execution (and message-passing) events of the program D

Server process		**Client process**	
Event	Timestamp	Event	Timestamp
`S::main`$_e$	0	`C::main`$_e$	0
`S::init`$_e$	1	`C::init`$_e$	1
`S::init`$_i$	2	`C::init`$_i$	2
`S::init`$_r$	3	`C::init`$_r$	3
`S::main`$_i$	4	`C::main`$_i$	4
`S::serve`$_e$	5	`C::compute`$_e$	5
E_C(C,S)	-	`C::shuffle`$_e$	6
`S::getMax`$_e$	**10**	`C::shuffle`$_i$	7
`S::getMax`$_i$	11	`C::shuffle`$_r$	8
`S::getMax`$_r$	12	`C::compute`$_i$	9
`S::serve`$_i$	13	E_C(C,S)	-
E_C(S,C)	-	E_C(S,C)	-
`S::serve`$_r$	14	`C::compute`$_r$	**14**
`S::main`$_i$	15	`C::main`$_i$	15
`S::main`$_r$	16	`C::main`$_r$	16

Thus, the MDG for this component has four nodes—library functions (e.g., `Socket::readLine`) are not modeled on this graph. This graph has, among others, an edge between the node for `S::main` and the node for `S::S` representing the data dependence between them as induced by the return variable s at Line 14, and an edge between the node for `S::main` and the node for `S::serve` representing the data dependence between them via the instance field $ssock$. Other example edges are between the node for `S::serve` and the node for `S::getMax` for the dependence between them via the return variable r and the dependence between them via the parameter s, both at Line 10.

5.1.2 Dynamic Data

Our framework utilizes only lightweight run-time (i.e., dynamic) information, at method and statement levels. In particular, D^{2}ABS uses three forms of dynamic data as mentioned earlier: *method-execution events* and *message-passing events*, which record the occurrence and timing of method calls and IPCs, respectively, and *statement coverage*, which records which statements are exercised during the execution. In the general context of distributed systems, an *event* is defined as any happening of interest observable from within a computer [28]. More specifically, events in a DEBS are often expressed as messages transferred among system components and defined by a set of attributes [10], [33]. While it also deals with message passing in distributed systems, currently D^{2}ABS neither makes any assumption nor reasons about the structure or content of the messages (doing so would compromise its applicability—it would be applicable to a narrower scope).

Method execution events. As the very basic form of dynamic data for its dynamic dependence analysis at the method level, D^{2}ABS monitors and utilizes method execution events. We define a method execution event $E_I(c)$ as an occurrence of method execution within a component c, where E stands for "Event" and I for "Internal"—such events are *internal* to the component [26]. We differentiate three subcategories of such events: *entering a method*, *returning from a method*, and *returning into a method*, denoted as m_e, m_r, and m_i, respectively, for the method m.

Note that D^{2}ABS captures both the return (from) and returned-into events for each method. For example, for a call to a method g in a method f, a *return from* event, associated with g, indicates the event that the program control gets out of the scope of g, marking the end of one execution instance of g. Differently, the corresponding *return into* event, associated with f, indicates the event that the program control gets back into the scope of f, marking the continuation (after the callsite targeting g) of one execution instance of f. However, we distinguish them during instrumentation only and treat them equally in the analysis algorithms of D^{2}ABS. In sequential program executions, a method m is potentially affected by any changes in the query q if m, or part of it, executes after q. Thus, monitoring method entry and returned-into events suffices for retrieving such execute-after relations. In the case of concurrent (single-process) programs, however, m is potentially affected by the changes also if m, or part of it, executes in parallel with q. Therefore, method return events need to be monitored as well to correctly identify dependence relations from interleaving method executions in multiple threads [19].

Message-passing events. Besides method execution events, D^{2}ABS monitors message-passing events, used to reason about the effects of IPCs in the distributed system on the dynamic dependencies among methods across all the processes of the system's execution. We define a message-passing event $E_C(c1,c2)$ as the occurrence of a message transfer between two components $c1$ and $c2$ where $c1$ initiates the event which is attempted to reach $c2$. Here E stands for "Event" and C for "Communication"—such events are *communication* events [26]. Further, according to the direction of message flow, we distinguish two major subcategories of such events: *sending a message* to a component and *receiving a message* from a component. We refer to the process running the component that sends and receives the message as the *sender process* and *receiver process*, respectively.

Statement coverage. For containing analysis costs, most kinds of program information D^{2}ABS uses are at the method level. To provide cost-effectiveness options with more emphasis on precision, D^{2}ABS also considers using statement coverage as a form of statement-level data. More specifically, it records which statements are covered during the system execution, separately for each component of the system. This fine-grained form of data is used to refine the per-component (method-level) static dependencies so that they can contribute towards enabling an even more precise dynamic dependence analysis.

Illustration. As an example, Table 1 shows the method execution events along with their timestamps, captured during an execution of the program D of Figure 1. The method name `init` denotes the constructor of a class. These events are recorded separately for each process and are listed in the order of their timestamps. For instance, in the server process, the event of (the program execution) entering the method `S::main` at timestamp 0 was denoted as `S::main`$_e$ associated with that timestamp. This is when the server process just started. Likewise, the event `C::init`$_r$ with timestamp 3 indicates that the program execution returned from the method `C::init` at that time, after it was called by the method `C::main`. Immediately after this event, at timestamp 4, the execution returned into the caller, `C::main`, denoted as `C::main`$_i$. Message-passing events are also captured. For instance, E_C(C,S) in the client process indicates the event that the client sent a message (the string s after being shuffled) to the server, triggered by the call `csock.writeLine(s)`. In this simple program, there are no branches. As a result, during the system execution, in accordance with the events of Table 1, all statements of D are covered.

5.2 Dependence Inference

In this section, we present the various dependence inference rules underlying the diverse cost-effectiveness tradeoffs D^{2}ABS offers via its different instantiations. We first describe the basic

inference of dependence relations among executed methods (Section 5.2.1). We then discuss three ways to refine the basic inference so as to improve its precision at additional costs: (i) leveraging message-passing semantics (Section 5.2.2), (ii) incorporating intra-component static dependencies (Section 5.2.3), and (iii) pruning spurious dynamic dependencies using statement coverage (Section 5.2.4). Note that while conceptually dynamic analysis does not produce spurious dependencies, such dependencies can be resulting from the *dependence approximations*, as adopted in our framework. For example, for high efficiency, we may conclude (approximately) that a method $m2$ depends on another method $m1$ if $m2$ executes after $m1$. This dependence can be spurious because the execution order used for the dependence approximation here does not necessarily imply true dependence.

We use dynamic impact analysis as an illustrating application (client analysis) of D^{2}ABS, for which the dependence relation between two methods derived by D^{2}ABS immediately corresponds to the impact relation between the two methods: a method $m2$ is considered to be impacted by a method $m1$ (i.e., $m1$ impacts $m2$) if $m2$ is considered to depend on $m1$. Thus, for our dependence-based dynamic impact analysis application, we may use "*be impacted by*" and "*depend on*" exchangeably hereafter.

Note that while our dependence inference rules may seem intuitive and straightforward, they appear to be so after we have revealed them—these rules have not been presented before, especially in the context of dynamic analysis for common distributed programs. In particular, the partial-ordering algorithm underlying the basic inference was originally proposed for event synchronization in distributed systems [28]. Yet it has not been exploited for modeling code-based dynamic dependencies. Thus, as opposed to computational challenges, exploring how to infer dependencies between two code entities (e.g., methods) in distributed programs comes more with methodological challenges—it was not known (1) which information, among various possible kinds, to utilize, (2) how to combine different kinds of information to derive dependencies, and furthermore (3) how to infer the dependencies in different yet cost-effective ways.

5.2.1 Basic Dependence Inference

One challenge to developing D^{2}ABS is to infer dependence relations based on execution order in the presence of asynchronous events over concurrent multiprocess executions. Fortunately, maintaining a logical notion of time per process to discover just a partial ordering of method execution events suffices for that inference. The dependence relation between any two methods can be semantically over-approximated by the *happens-before* relation between relevant execution events of corresponding methods; and the partial ordering of the execution events reveals such happens-before relations [28]. Formally, given two methods $m1$ and $m2$, we have

$$m1_e \prec m2_r \vee m1_e \prec m2_i \Longrightarrow m2 \textit{ depends on } m1 \quad (1)$$

where $\prec$ denotes the *happens-before* relation. Without loss of generality, either of $m1_e \prec m2_r$ and $m1_e \prec m2_i$ implies that "$m2$ executes after[2] or in parallel with $m1$, thus $m2$ may be affected by (any changes in) $m1$", hence the dependence (impact) relation between $m1$ and $m2$. For example, with respect to the example execution of program D shown in Table 1, `C::init`$_e \prec$`C::main`$_i$ implies that `C::main` depends on `C::init`, and `C::compute`$_e \prec$`C::shuffle`$_r$ implies that `C::shuffle` depends on `C::compute`.

Based on the above inference, for a given query q, computing the dependence set $DS(q)$ of q is reduced to retrieving methods, from multiprocess method execution event sequences, that satisfy the partial ordering (as defined in [28]) of the execution events of candidate methods as follows:

$$DS(q) \;=\; \{m \mid q_e \prec m_i \vee q_e \prec m_r\} \quad (2)$$

Note that only method execution events are directly used for the inference, whereas message-passing events are utilized to maintain the partial ordering of method execution events across processes. While this basic dependence inference leads to potentially excessive imprecision due to its overly conservative nature, it only requires highly lightweight dynamic analysis and computation, which implies great efficiency. Therefore, this basic inference still provides a useful cost-effectiveness option.

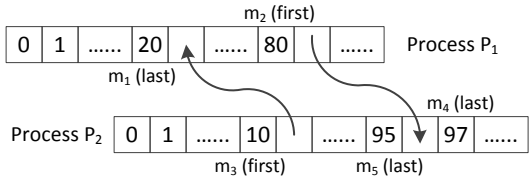


Fig. 4: Inter-process impact constrained by message passing.

5.2.2 Leveraging Message-Passing Semantics

The above basic inference leads to a safe yet possibly *overly* conservative approximation of dynamic dependencies (impacts), because it is based purely on happens-before relations between method execution events. Recall that message passing is the only communication channel between processes in the systems we address to propagate data flows or control decisions. Let ${P_i}^m$ denote a method m that is executed in a process P_i. Then, a method ${P_1}^m$ in process P_1 would not depend on a method ${P_2}^m$ in process P_2 if P_1 never received a message from P_2 *before* the last execution event of ${P_1}^m$. This is true even if ${P_1}^m$ is executed after ${P_2}^m$ in the whole-system method execution event partial ordering, since there would be no data or control flow between the two processes that can possibly affect ${P_1}^m$.

For example, Figure 4 illustrates how message passing could constrain the dependence relation defined by Equation 1. Each numbered cell represents the first or last execution event of a method with the number indicating the time (stamp) when that event occurs, and each empty cell represents a message receiving or sending event—these events are considered separately from method execution events (see Section 5.1.2). The arrowed line on the left and right indicates the *first* message passing from P_2 to P_1 and that from P_1 to P_2, respectively. As per the basic inference, m_1 in P_1 would be inferred as dependent on m_3 in P_2, because m_1 was executed after m_3. With the message-passing semantics considered, however, this dependence is deemed as spurious hence should be pruned. The reason is because P_1 received the first message from P_2 after the last time m_1 was executed. Similarly, a spurious dependence of m_5 on m_2 (as produced per the basic inference) should also be pruned. On the other hand, m_4 is still considered dependent on m_2 even with the message-passing semantics considered, because not only was m_4 executed after m_2 was first executed, but also P_2 received a message from P_1 prior to the last execution event of m_4.

Generally, considering the message-passing semantics, ${P_i}^m$ potentially depends on ${P_j}^{m'}$ only if (i) the first execution event

2. Hereafter throughout the paper, *execute before/after* relations between methods are referred to in accordance with the *happens-before/after* relations between associated method execution events.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

of ${P_j}^{m'}$ happens before the last execution event of ${P_i}^{m}$—this essentially enforces the basic inference rule, (ii) P_j sends at least one message (directly or transitively) to P_i—this is a quick check against spurious dependencies, because if there is no message passing between two processes, two methods across the processes should not have dependence relationships, and (iii) the last execution event of ${P_i}^{m}$ happens after the first message receiving event in P_i from P_j, and the message is sent after the first execution event of ${P_j}^{m'}$—this last condition ensures that the message, which represents interprocess data flow, can possibly be affected (written to or defined) by ${P_j}^{m'}$ and then possibly affect (be read or used by) ${P_i}^{m}$. Formally, let $T_F(m)$ and $T_L(m)$ denote the time (stamp) of the first and last execution event of a method m, respectively, and let $T_S(P_i, P_j)$ denote the time (stamp) of the event of P_i receiving the first message from process P_j, we define a customized form of partial order relation between two methods ${P_i}^{m}$ and ${P_j}^{m'}$ as follows (T_S is the *message-receiving map*):

$$ {P_j}^{m'} \prec {P_i}^{m} := \begin{cases} T_L({P_i}^{m}) \geq T_F({P_j}^{m'}), & \text{if } i = j \\ T_S(P_i, P_j) \neq \text{null} \wedge T_L({P_i}^{m}) \geq T_S(P_i, P_j) \geq T_F({P_j}^{m'}), & \text{if } i \neq j \end{cases} \tag{3} $$

With this constrained definition of $\prec$, a more precise dependence inference is obtained from Equation 2 since the potential method-level dependencies identified by the basic inference that do not satisfy the additional constraints (ii) and (iii) will be pruned. Given the scope of distributed systems we address, this pruning does not compromise the safety of resulting dependence sets.

Note that while both exploit the happens-before relations among method execution events that are synchronized through message-passing events, the basic inference and the inference leveraging message-passing semantics utilize different information of those relations. The basic inference only uses these relations to partially order the method events, disregarding the data flow implications of message passing. In comparison, the more advanced inference additionally leverages the data-flow semantics of the message-passing events (i.e., a message sending or receiving event implies a data write/definition or read/use).

5.2.3 Incorporating Intra-Component Dependencies

With the static dependencies of a component c and method execution events of the process P_c that executes c, the dependence/impact relation between two methods ${P_c}^{m1}$ and ${P_c}^{m2}$ in P_c is inferred from both the partial ordering of execution events associated with these two methods and the static dependencies between them. Formally, we define another constrained partial order relation as follows:

$$ {P_c}^{m1} \prec {P_c}^{m2} := \begin{array}{r} T_L({P_c}^{m2}) \geq T_F({P_c}^{m1}) \ \wedge \\ {P_c}^{m1} \xrightarrow{MDG_c} {P_c}^{m2} \end{array} \tag{4} $$

where ${P_c}^{m1} \xrightarrow{MDG_c} {P_c}^{m2}$ denotes the *dynamic* dependence of ${P_c}^{m2}$ on ${P_c}^{m1}$ with respect to the (static) dependence graph MDG_c and the method execution events in P_c. In order to avoid imprecision induced by straightforward transitive dependence computation, this dynamic dependence is not simply determined through reachability on the graph. Instead, the dependence is computed iteratively according to the semantics of method execution events (e.g., a method-entry event indicates that data flow facts associated with the method's parameters start propagating forward) while traversing the graph (as described in Section 6.1.3). Applying this constrained partial order relation to Equation 2 leads to a more precise dependence inference through pruning false positives within individual processes, as detailed below.

Given a query method q defined in component c and executed in process P_c, methods that execute after q but are not statically dependent on q will be pruned from the trace of P_c. For example, in the client process of program D, `C::shuffle` executed after `C::compute`. Yet, on the MDG for the client component of D, `C::shuffle` does not depend on `C::compute`. Thus, `C::shuffle` will be pruned from the (forward) dependence set of `C::compute` that would be produced as per the basic inference. To apply similar pruning based on static dependencies for every other component c' and corresponding process $P_{c'}$, we need to identify a method similar to q that serves as the starting point of the dynamic dependence/impact propagation (noted as *spark method*). [3] We safely choose the first method in $P_{c'}$ that executed after $P_{c'}$ receives a message from P_c as the spark method of c' for query q. For example, if the query is in the client component of program D, `S::getMax` would be the spark method for the server component of D. Then, methods that are partially ordered after but not statically dependent on the spark method will be pruned from the trace of $P_{c'}$.

5.2.4 Pruning with Statement Coverage

Conceptually, the intra-component dependencies in D^{2}ABS aim to directly model dependencies at the method level. However, in terms of the concrete (graph) representation of these dependencies, there may be multiple edges (i.e., interprocedural dependencies) between two methods (e.g., data dependencies each due to the passing of a different parameter from the caller to the callee). These edges are not conflated into a single dependence edge. The rationale is that, by doing so, D^{2}ABS can avoid imprecision accumulation during transitive dependence propagation across methods, threads, and processes. Specifically, all the individual interprocedural dependencies computed at statement level are kept to represent method-level dependencies. Thus, for any two nodes of an MDG_c, there may be multiple edges. D^{2}ABS further uses edge annotations to denote the type of interprocedural dependence for each edge (e.g., *interference*, *synchronization*, *traditional control*, etc.). These edge annotations are used later in the precise transitive dynamic dependence analysis algorithm of D^{2}ABS.

Given this underlying representation of the static dependencies in D^{2}ABS, our framework provides an additional option of collecting statement coverage data and utilizing the data to further prune possible spurious dynamic dependencies. A method $m2$ is considered to (dynamically) depend on a method $m1$ if there is at least one static interprocedural dependence of $m2$ on $m1$ on the corresponding MDG_c that has both statements associated with the dependence covered during the execution analyzed by our framework. Otherwise, the method-level dependence between $m1$ and $m2$ should be pruned. This pruning rule can be formally defined as another constrained partial order relation as follows:

$$ {P_c}^{m1} \prec {P_c}^{m2} := \begin{array}{r} T_L({P_c}^{m2}) \geq T_F({P_c}^{m1}) \ \wedge \\ {P_c}^{m1} \xrightarrow[\text{at lease one covered}]{MDG_c} {P_c}^{m2} \end{array} \tag{5} $$

where the notations are the same as in Equation 4. The only difference is that *at least one* of the interprocedural dependencies

3. Without loss of generality, given a dependence query q, we define a *spark method* P_c^{sm} specific to a process P_c as the first method sm executed in P_c that is (transitively) dependent on q.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

TABLE 2: The four D^{2}ABS instantiations defined by analysis data and dependence inference rules used

			Instantiation			
			Basic	**Msg+**	**Csd+**	**Scov+**
dependence inference rule (Equation)			1	3	4	5
analysis data	static	static dependencies	✗	✗	✓	✓
	dynamic	method execution events	✓	✓	✓	✓
		message-passing events	✗	✓	✗	✗
		statement coverage	✗	✗	✗	✓

between $m1$ and $m2$ on MDG_c, in addition to their existence, must be exercised (*covered*) during the execution. For example, in the server process of the working example program D, `S::getMax` executed after `S::serve` and the MDG of the server component also indicates that `S::getMax` (statically) depends on `S::serve` (because the former uses variable s which is defined in the latter). Now suppose the statement `char r = getMax(s);` is not covered (e.g., suppose there is an unexercised predicate guarding this statement) during the analyzed execution. In this case, `S::getMax` would be pruned from the (forward) dynamic dependence set of `S::serve` that would be produced as per the *incorporating intra-component dependencies* inference rule (i.e., Equation 4). Note that like the pruning based on static intra-component dependencies, the pruning rule here based on statement coverage only applies within a single process (albeit possibly across multiple threads). Also, similarly, applying this constrained partial order relation to Equation 2 leads to yet another level of precision of dynamic dependence analysis in D^{2}ABS due to the removal of possible false positives associated with statements that are not covered in the analyzed execution.

6 Framework Instantiations

As we discussed in Section 2, users would need variable levels of cost-effectiveness balances to accommodate different dependence-based task scenarios. Now that D^{2}ABS aims to provide a dynamic dependence analysis framework that empowers a range of dynamic-dependence-based applications, it should have the capabilities for offering dynamic dependencies at variable cost-effectiveness levels so as to meet the diverse application needs. To streamline this vision, D^{2}ABS provides flexible options for users to enable/disable certain parts of its analysis data and some steps of its analysis algorithms, hence to accommodate different usage scenarios that need varied cost-effectiveness tradeoffs (e.g., some application tasks prioritize precision over efficiency, while some others accept relatively rough/low-precision dependencies in exchange for high efficiency/scalability). The current D^{2}ABS implementation provides different command-line options for choosing the various tradeoff levels, as the tool's user interface.

In particular, currently D^{2}ABS unifies four instantiations, each corresponding to a version of D^{2}ABS that offers a distinct level of cost-effectiveness. By presenting these instantiations, we do not intent to investigate all possible levels of such tradeoffs; rather, the goal here is to demonstrate the capabilities of our framework design for enabling varying dynamic dependence analysis of distributed programs to offer variable levels of cost-effectiveness.

Table 2 outlines the four instantiations (2nd row) of D^{2}ABS, defining each in terms of the analysis data (static and/or dynamic, 4th–7th rows) and dependence inference rule used (3rd row). The rationale for having these four is two-fold. First, the **Basic** version uses only one form of data: method execution events. While message-passing events are used to partially order these method execution events, they are not further utilized in the analysis algorithm for this instantiation. Second, from the 5th to 7th columns, each instantiation aims to enhance the **Basic** version in a different way, by adding message-passing events only, static dependencies only, or statement coverage along with static dependencies. Statement coverage needs to go with static dependencies, because it cannot be used along with other forms of data in our framework—the static dependencies are the only other form of data with statement-level details. There could be more instantiations of D^{2}ABS. We study these four as they have intuitively more *differentiable* cost-benefit tradeoffs than other possible instantiations, and because these four would suffice for the purpose of demonstrating the capabilities of our framework in offering diverse cost-effectiveness levels.

- **Basic** version. This is the simplest, most lightweight instantiation of D^{2}ABS, which computes dynamic dependencies among methods simply using the **basic** inference (Section 5.2.1). It only utilizes method execution events and their global partial ordering, essentially a dependence approximation purely based on interprocess control flows.
- **Msg+** version. This instantiation exploits the **m**ess**a**ge-passing semantics to prune false-positive dynamic dependencies *across* processes in the **Basic** version (Section 5.2.2). It utilizes method execution events and per-process message-receiving maps, essentially a dependence approximation based on interprocess control *and* data flows.
- **Csd+** version. This version enhances the **Basic** version by incorporating intra-**c**omponent **s**tatic **d**ependencies to prune false-positive dynamic dependencies *within* individual processes (Section 5.2.3). It utilizes method execution events and per-component static dependence analysis, essentially a dependence approximation based on *interprocess* control flows and *intra-component* (data and control) dependencies.
- **Scov+** version. This is a further refinement of the **Basic** version, incorporating statement coverage in **Csd+** to prune false-positive dynamic dependencies *within* individual processes (Section 5.2.4). In essence, **Scov+** utilizes both method- (i.e., execution events and static dependencies) and statement-level data for a method-level dynamic dependence analysis that is supposedly more precise than **Csd+**.

TABLE 3: Cost-effectiveness tradeoffs of D^{2}ABS instantiations

D^{2}ABS Instantiation	static analysis time	runtime overhead	querying time	effectiveness (precision)
Basic		✓	✓	✗
Msg+			✗	✓
Csd+	✗		✗	✓
Scov+	✗	✗	✗	✓

To further justify our decision for having these different instantiations of our framework, Table 3 shows what the cost-effectiveness factors are that each instantiation attempts to trade for and off: ✗ indicates a factor that is traded off (i.e., compromised) while ✓indicates a factor that is traded for (i.e., prioritized). For instance, the **Basic** version trades effectiveness (in terms of precision) for efficiency in terms of both runtime overhead and querying time, while **Scov+** trades all the efficiency factors for greater effectiveness. Note that in general which factor is considered *traded off* versus which is considered *traded for* is a relative notion: for example, relative to **Basic**, **Msg+** trades efficiency for effectiveness, but relative to **Csd+**, **Msg+** trades effectiveness for efficiency. Here in Table 3, the tradeoff made by **Basic** is determined based on its nature (suffering very low

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

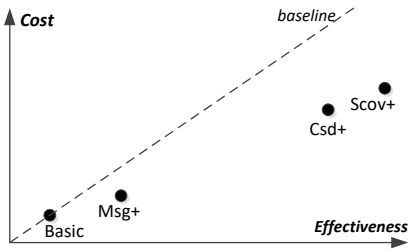


Fig. 5: The cost-effectiveness space explored by D^{2}ABS through its four instantiations. The coordinates of these instantiations are not based on empirical measurements but only provide estimated contrasts among them in terms of their cost-effectiveness levels.

precision for achieving high efficiency), while the tradeoff made by the other three instantiations is determined all relative to **Basic**.

To complement Table 3 which gives the factors involved in the tradeoff made by each instantiation, Figure 5 further gives estimated, relative comparisons of cost-effectiveness levels among the four instantiations—the coordinate for each instantiation does not correspond to its exact cost-effectiveness measure. The dashed line ($y = x$) represents the baseline, indicating the level of cost-effectiveness achieved by the **Basic** version. Accordingly, a point below this line means the associated analysis is better than the baseline, and the further the point is from the line the more cost-effective the associated analysis. As depicted, through its various instantiations each representing a distinct level of cost-effectiveness, our framework provides variable cost-effectiveness tradeoffs as a whole, meeting diverse application needs.

6.1 Analysis Algorithms

In this section, we present the unified dependence analysis (abstraction) algorithm in D^{2}ABS. It is referred to as *unified* because it generalizes all of the varying instantiations of the framework, using different dependence inference rules according to the different selections of analysis data. For obtaining each form of static/dynamic data, D^{2}ABS has a dedicated algorithm as well. Among those algorithms, we present two in detail—one for monitoring message-passing events and the other for partially ordering method execution events; the others are straightforward.

6.1.1 Monitoring Message-Passing Events

Preserving the partial ordering of method execution events is at the core of D^{2}ABS, since the partially-ordered method execution events are the default form of data used by any instantiation of the framework. For that purpose, we adopt the *Lamport time-stamping* (LTS) approach [28] based on a logical notion of time. The idea is to maintain a synchronized logic clock across processes of the system execution analyzed, through monitoring message-passing events. From these events, the *message-receiving map* is also derived, but only for the **Msg+** version which uses the data to prune spurious dependencies resulting from the basic inference.

Algorithm 1 summarizes our algorithm for monitoring message-passing events while carrying and synchronizing the logic clocks based on the original LTS algorithm. The logical clock C of the current process is initialized to 0 upon process start, as is the global variable `remaining`, which tracks the remaining length of data most recently sent by the *sender* process. The role of `remaining` is to avoid the message-passing event monitoring causing interference with the original message-passing semantics

Algorithm 1: Monitoring message-passing events

let C be the logical clock of the current process
remaining = 0 // remained length of data to read

```
1: function SENDMESSAGE(msg)// on sending a message msg
2:     sz = length of sz + length of C + length of msg
3:     if using the Msg+ version then
4:         sz += length of the sender process id sid
5:         pack sz, C, sid, and msg, in order, to d
6:     else
7:         pack sz, C, and msg, in order, to d
8:     end if
9:     write d
10: end function
11: function RECVMESSAGE(msg)// on receiving a message msg
12:     read data of length l into d from msg
13:     if remaining > 0 then
14:         remaining -= l
15:         return d
16:     end if
17:     retrieve and remove data length k from d
18:     retrieve and remove logical clock ts from d
19:     remaining = k - length of k - length of ts - l
20:     if ts > C then
21:         C = ts
22:     end if
23:     increment C by 1
24:     if using the Msg+ version then
25:         retrieve and remove sender process id sid from d
26:         add (sid, C) to message-receiving map if sid∉its key set
27:         remaining -= length of sid
28:     end if
29:     return d
30: end function
```

of the analyzed system execution, as further elaborated later in our discussion of practical challenges. The rest of this algorithm consists of two parts, triggered upon the occurrence of IPC (i.e., message-passing) events during system executions.

The first part is the run-time monitor SENDMESSAGE triggered online upon each message-sending event. The monitor piggybacks (prepends) two extra data items to the original message: the total length sz of the data to send, and the present value of the local logical clock C (of this *sender* process) (lines 2–7); then, it sends out the packed data (line 8). The sender process id is also packed in the **Msg+** version (lines 3–5).

The second part is the other monitor RECVMESSAGE, which is triggered online upon each message-receiving event. After reading the incoming message into a local buffer d (line 10), the monitor decides whether to simply update the size of remaining data and return (lines 11–13), or to extract two more items of data first: the new total data length to read, and the logical clock of the peer *sender* process (lines 14–15). In the latter case, the two items are retrieved, and then removed also, from the entire incoming message. Next, the remaining data length is reduced by the length of the data already read in this event (line 16), and the local logical clock (of this *receiver* process) is compared to the received one, updated to the greater, and incremented by 1 (lines 17–19). In the **Msg+** version, the sender process id is retrieved as well (line 21) and a new entry is added to the message-receiving map if this is the first message received from that sender process (line 22). Lastly, the monitor returns the message as originally sent in the system (i.e., with the prepended data taken away).

Note that D^{2}ABS monitors message-passing events initiated by any message-passing APIs exercised during the analyzed system execution, regardless of the message passing happening

across processes or between threads within the same process—the monitoring probes were inserted during the static instrumentation (i.e., **Step** ❶ shown in Figure 3) when the message passing targets cannot be resolved. The purpose of monitoring these events is to synchronize the timing of the method execution events across processes. Thus, the monitoring of events of message passing between threads in a process and the processing of the events (to synchronize method execution events within that process) would be of no avail, because such threads already have a synchronized timing mechanism (logic clock). On the other hand, the unnecessary monitoring only incurs a small extra cost while not affecting the correctness of our dependence analysis algorithms.

Illustration. For the example program D, `Socket::readLine` and `Socket::readChar` are message-receiving APIs, while `Socket::writeLine` and `Socket::writeChar` are considered message-sending APIs. During the static instrumentation, D^{2}ABS instruments for each callsite of any of these message-passing APIs to probe for corresponding message-passing events. For instance, during the execution of the instrumented version of D, an invocation of the run-time monitor SENDMESSAGE will be triggered immediately upon the call `ssock.writeChar(r)` at Line 11. Through this monitor, the current logic clock of the server process will be piggybacked to the message (i.e., character r) hence passed to the client process. Similarly, during the execution, an invocation of the run-time monitor RECVMESSAGE will be triggered immediately upon the call `csock.readChar(r)` at Line 29. Through this monitor, the client process retrieves the logic clock of the server process from the received message. In this way, D^{2}ABS lets all the processes of the system execution get each other's current logic clock, so as to synchronize the timing throughout the distributed execution.

Practical challenge. To avoid interfering with the message-passing semantics of the original system, D^{2}ABS keeps the length of remaining data (with the variable `remaining` in the algorithm) to determine the right timing for logical-clock (and sender id) retrieval. In real-world distributed programs (e.g., Zookeeper [44]), it is common that a *receiver* process may obtain, through several reads, the entire data sent in a single write by its peer *sender* process. For example, a first read just retrieves the data length so that an appropriate size of memory can be allocated to take the actual data content retrieved in a second read. Therefore, not only is it unnecessary to attempt retrieving the prepended data items (data length and logical clock) in the second read since the first one should have already done so, but also such attempts can break the original network I/O protocols. D^{2}ABS addresses this issue by piggybacking the length of data to send and tracking the remaining length of data to receive.

To illustrate our solution to this practical challenge, consider the example program D of Figure 1 again. Suppose the server process reads the string sent (at Line 28) by the client process through a second read (also by calling the same message-receiving API `Socket::readLine`) in addition to the one at Line 9. Further suppose the length of the string sent is 16 and the first read only retrieves 8 characters of the string. Then, as per Algorithm 1 (lines 11–30), upon the first read, RECVMESSAGE is invoked for the first time, where 8 bytes are retrieved from the message and put in the buffer d. At this time, `remaining`==0, thus the monitor proceeds to retrieve different parts of the first message segment (e.g., logical clock) and update `remaining`. Upon the second read, RECVMESSAGE is invoked again but it just updates `remaining` and returns the buffer d after retrieving the remaining characters of the original string.

6.1.2 Partial Ordering Method Execution Events

The **Basic** dependence inference in D^{2}ABS relies on the execution order of methods that is deduced from the timestamps attached to all method execution events. These timestamps are calculated based on each local logic clock, which is synchronized through D^{2}ABS's monitoring message-passing events as described earlier.

As proved in [19], recording the *first* entry and *last* returned-into (or return) events only is equivalent to tracing the full sequence of those events for our dynamic dependence inference that is purely based on the method execution order. Thus, instead of keeping the timestamp for every method execution event occurrence (as shown in Table 1), for the **Basic** version, D^{2}ABS only records two key timestamps for each method m: the one for the first instance of m_e, and the one for the last instance of m_i or m_r, whichever occurs later. As per Equation 3, this simplified tracing also suffices for the **Msg+** version. For the **Csd+** and **Scov+** versions, however, the full sequence of (i.e., instance-level) method execution events is required for effectively pruning false positives with intra-component dependencies [17], [22]—the hybrid dependence analysis algorithm needs to traverse each instance of every method execution event to *precisely* recognize false positives. Thus, the online monitor of method execution events records either the two key timestamps for each method, or the timestamp for every method execution event, depending on which version is used.

For each process, we use an integer counter for time-stamping, which is updated using the per-process logical clock. Meanwhile, the logical clock C_i of each process P_i is maintained as follows:

- Initialize C_i to 0 upon the start of P_i.
- Increase C_i by 1 upon each method execution event in P_i.
- Update C_i upon each message-passing event occurred in P_i via the two online monitors shown in Algorithm 1.

Finally, for the offline dependence computation in D^{2}ABS, the online algorithm here also dumps per-process method execution-event sequences as traces upon the program termination (in each distributed component). Additionally, in the **Msg+** version, the message-receiving maps are also serialized as part of the per-process traces. The **Scov+** version further carries statement-coverage information in the per-process traces.

Illustration. For the example program D, every time when the server process receives the current logic clock of the client process, it tries to update its own current logic clock with that received clock according to the algorithm described above. Suppose the server and client components are deployed on two distributed machines, and S' (i.e., the instrumented server component) starts before C' (i.e., the instrumented client component). When running concurrently (e.g., against an example input set I={"hello"}), S' and C' generate two method-event sequences in two separate processes, as listed *in full* in the first two and last two columns of Table 1, respectively. After the last instance of event `C::main`$_i$, the client prints 'o' (which is the maximum character in the input string). As shown, logical clocks are updated upon message-passing events. For instance, the logical clock of the server process is first updated to 10 upon the event E_c(C,S) originated in the client process, which is greater by 1 than the current logical clock of the client process. Later, the client logical clock is updated to 14 upon E_c(S,C). The method execution events are time-stamped

Algorithm 2: Computing dynamic dependencies

let $P_1, P_2, ..., P_n$ be the n concurrent processes of the system
let q be the query method
let $MDG[P_i]$ be the MDG of the component executed in P_i
let $AM[P_i]$ be the spark method of P_i with respect to q

```
1: locDS = ∅, extDS = ∅, comDS = ∅
2: for i=1 to n do
3:     mdg = (using the Csd+ or Scov+ version)?MDG[P_i]:null
4:     if using the Scov+ version then
5:         covStmts = obtainCovStmts(tr(P_i), MDG[P_i])
6:         updateMDG (mdg, covStmts)
7:     end if
8:     ts_q = computeIntraDS(q, locDS, tr(P_i), mdg)
9:     if ts_q==null then continue
10:    end if
11:    for j=1 to n do
12:        if i == j then continue
13:        end if
14:        if using the Csd+ or Scov+ version then
15:            if using the Scov+ version then
16:                covStmts = obtainCovStmts(tr(P_j), MDG[P_j])
17:                updateMDG (MDG[P_j], covStmts)
18:            end if
19:            computeIntraDS(AM[P_j], extDS, tr(P_j), MDG[P_j])
20:            continue
21:        end if
22:        if using the Msg+ version ∧ S(tr(P_j))[P_i]==null then
23:            continue
24:        end if
25:        for each method m∈ keyset(R(tr(P_j))) do
26:            if R(tr(P_j))[m] ≥ ts_q then
27:                if using the Msg+ version then
28:                    if R(tr(P_j))[m] ≥ S(tr(P_j))[P_i] then
29:                        extDS ∪= {m}
30:                    end if
31:                else
32:                    extDS ∪= {m}
33:                end if
34:            end if
35:        end for
36:    end for
37: end for
38: comDS = locDS ∩ extDS
39: return locDS, extDS, comDS
```

by these logical clocks while message-passing events are not, as marked by '-' (i.e., not applicable). In this way, all the method execution events are time-stamped with a synchronized timing such that they are partially ordered in the whole system execution.

6.1.3 Dynamic dependence analysis

During system executions, the online method-execution-event monitor generates event traces concurrently (and typically on distributed machines). Since it computes dependencies offline, D^{2}ABS gathers these traces after their completion to one machine, and computes dependence sets there as outlined in Algorithm 2. For the **Csd+** version, the MDGs of individual components are also utilized; and statement coverage is further computed and utilized on top of the MDGs for the **Scov+** version.

For a detailed analysis, we refer to the process where the query is first executed as *local process* versus all other processes as *remote process*, and dependencies (represented by the dependent methods) in local and remote processes as *local dependencies* and *remote dependencies*, respectively. For a given query q, D^{2}ABS computes its dependence set as three subsets: *local dependence set*, *remote dependence set*, and their intersection called *common dependence set* (denoted as *locDS*, *extDS*, and *comDS*, respectively, in Algorithm 2).

The algorithm takes the query q, n per-process traces, and n per-component MDGs as inputs, and outputs the three subsets (line 27) all initialized as empty sets (line 1). Per-process spark methods can be readily derived from the message-receiving maps. The algorithm traverses the n processes (loop 2–25) taking each as the local process (line 2) against all others as remote processes (lines 9–10) to first compute the local dependence set (line 4). Then, the remote dependence set is computed (loop 11–25) based on Equation 4 if **Csd+** or **Scov+** is used (lines 11–16), or on Equation 3 if **Msg+** is used (lines 17–25). In particular, if the **Scov+** is used, the static dependencies need to be pruned based on statement coverage before they are used for local (lines 4–6) or remote (lines 12–14) dependence analysis, as per Equation 5.

The subroutine `computeIntraDS` computes the dependence set (to return via the second argument) of the given query (taken as its first argument) *within* the given process (indicated by the third argument). If the MDG of the component associated with this process is given as null (as the last argument), it will be simply ignored by the subroutine, which will identify as dependants the methods whose last execution is not earlier than the first execution of the query as in EAS [19]. Otherwise (i.e., in **Csd+** or **Scov+** version), the MDG will be utilized along with the trace for the process (given as the third argument) to compute the dependence set using DIVER [22]. The key ideas of DIVER's hybrid dependence analysis are summarized below for self-containing purposes.

Generally, the DIVER computation carefully checks whether a method execution event e in the trace leads the dependence (impact) originated in the query to propagate to the method associated with e by referring to the MDG while using different propagation rules. The rationales underlying these rules are (1) through static dependencies induced by parameter or return-value passing, impact can only propagate between two adjacently executed methods, and (2) through other kinds of static dependencies (those induced by definition-use relations between heap variables), the dependence can propagate from a method a to any method that executed after a. The subroutine `computeIntraDS` unionizes the newly computed dependence set with the one passed in, and returns the first execution event time ts_q of q if q is found in the input trace and `null` otherwise (line 8).

For the **Scov+** version, subroutine `obtainCovStmts` is needed to compute the covered statements (line 5) from the coverage information contained in the trace of the given process (passed in as the first argument). For the sake of efficiency, D^{2}ABS instruments for and monitors covered branches in previous phases. Now in the dependence analysis, covered statements are derived according to the branch coverage and control dependencies on the MDG for the component corresponding to the given process (passed in as the second argument). Then, the third subroutine `updateMDG` is invoked, which prunes dependencies with at least one statement not included in the covered statements (line 6).

For the **Csd+** or **Scov+** version, the external dependence set is computed similarly (line 15), but using the spark method of a remote process P_j as the query (the first argument) and always taking the MDG of the component that P_j executes (the last argument). Computation of statement coverage and MDG pruning with the result for the remote process (lines 13–14) are both similar to those for the computation of local dependencies.

Further, for the **Msg+** version, *tr(P)* denotes the trace of process P, *R(t)* denotes the hashmap from each executed method

to its last execution event time in trace t, *S(t)* denotes the hashmap from the id of each sender process to the event time when the remote process receives the first message from the sender, and *keyset(.)* returns the key set of a hashmap. If the (remote) process P_j never received any message from the (local) process P_i, no remote impacts would be found in P_j (lines 17–18).

For a single test input, the dependence-computation algorithm computes the dependence set of one method at a time; for multiple methods in the query set, the result is the union of all the one-method dependence sets computed separately (e.g., in parallel). Similarly, the dependence set for multiple tests is the union of all per-test dependence sets. These treatments are similar to those adopted in union slicing [45].

Illustrations. To illustrate the unified dynamic dependence analysis in D^{2}ABS, we now give a running example for each of its four instantiations, Still consider the program D of Figure 1 as the program under analysis. Suppose the message-passing and method execution events have been captured as described in the previous two illustrations (i.e., at the end of Sections 6.1.1 and 6.1.2). Further, suppose the MDGs for S and C have been constructed also, and suppose the query set M={`S::serve`}.

By inferring dynamic dependencies from the happens-before relationships among method execution events according to their timestamps (i.e., according to Equation 2), the **Basic** version produces {`S::getMax`, `S::serve`, `S::main`, `C::shuffle`, `C::compute`, `C::main`} as the dependence set of M. The reason is because, as shown in Table 1, `S::serve` first executed at timestamp 5, which is no greater than the timestamp at which any of the method in this dependence set executed the last time (e.g., 8 for `C::shuffle` and 12 for `S::getMax`).

By exploiting message-passing semantics, the **Msg+** version prunes `C::shuffle` from this dependence set according to Equation 3. In this case, the client process is P_i, the server process is P_j, `S::serve` is ${P_j}^{m'}$, and `C::shuffle` is ${P_i}^{m}$. Thus, here $T_S(P_i, P_j)$=13, $T_L({P_i}^{m})$=8, $T_F({P_j}^{m'})$=5 (see Table 1), hence the $T_L({P_i}^{m}) \geq T_S(P_i, P_j) \geq T_F({P_j}^{m'})$ is not satisfied. The intuitive explanation is this: the impact of the computation in `S::serve` in the server process can propagate to the client process via the message sent by the server at timestamp 13, because `S::serve` executed before the message was sent; yet `C::shuffle` never executed in the client process after the message was received, thus it was too late for `C::shuffle` to be impacted by the computation in `S::serve`. As a result, `C::shuffle` has no dynamic dependence on `S::serve`.

By utilizing the static dependencies of both components, the **Csd+** version prunes `S::main` from the dependence set produced by the **Basic** version according to Equation 4. The reason is because there is no static dependence of `S::main` on `S::serve`, thus the static dependence graph of the server component does not have the edge between these two methods that is required by the *incorporating intra-component dependencies* rule.

In this simple example, since none of the methods contain branches, the fact that a method executed means that every statement in that method is covered. Thus, statement coverage data did not lead to further reduction of the dependence set. As a result, the **Scov+** version produces the same dependence set as **Csd+** does for the given query set.

Now let us respond to the challenge to the developer in the use scenario that motivated D^{2}ABS. As demonstrated, D^{2}ABS can compute dynamic dependencies (hence predict impacts) across distributed components (processes). For an example application of these dependencies in impact analysis, if the developer plans for a change to method `serve` in the server, the methods `compute` and `main` in the client, in addition to the other server method (`S::getMax`), are potentially affected and thus need impact inspection by the developer before applying that change.

6.1.4 Analysis Soundness and Result Safety

The soundness of D^{2}ABS relies on that of its static and dynamic analysis. Our discussion and characterization on the analysis soundness and results safety of D^{2}ABS refer to relevant definitions and discussions in [29]: a sound static analysis produces information that holds for all possible program executions, while a sound dynamic analysis produces information that holds for the analyzed execution alone. As per these definitions, our static analysis is not sound, because of its inability to deal with dynamic language constructs in Java (e.g., reflection, native code invocation via JNI, etc.)—the analysis does not see the relevant code at compile time thus its results may not hold for the executions involving such code. In fact, sound static analysis for a language that allows for dynamic code constructs is generally rare [46].

Also per the definition in [29], our dynamic analysis that only uses dynamic data is sound as it captures all the dynamic dependencies (at method level) exercised in the analyzed executions and does so always in a conservative manner: for instance, the **Basic** version over-approximates dynamic dependencies between two methods just according to their happens-before relationships. However, in two instantiations of D^{2}ABS, **Csd+** and **Scov+**, the dynamic analysis utilizes static dependencies also, which are produced by the static analysis that is unsound; thus, the resulting dynamic dependencies may not hold for some executions (i.e., those involving dynamic code constructs). Therefore, considering all of its instantiations, D^{2}ABS is unsound as a whole.

Traditionally, the result of a sound analysis is considered *safe* and that of a unsound analysis unsafe. Thus, we regard the resulting dependence sets of D^{2}ABS as unsafe in light of the overall unsoundness of our (hybrid) dynamic (dependence) analysis. Note that here we use "soundness/unsoundness" as a property of an analysis while using "safety/unsafety" to characterize the results of the analysis. Leveraging additional analyses (e.g., statically resolving the targets of reflective calls) to mitigate the unsoundness and unsafety issues is part of our future work.

7 IMPLEMENTATION

D^{2}ABS consists of three main modules: a static analyzer, two sets of run-time monitors, and a post-processor. Careful treatments are crucial for a non-interfering implementation as recapped below. We released the source code of D^{2}ABS along with our empirical study results at https://bitbucket.org/wsucailab/d2abs.

Static analyzer. The static analyzer first instruments the input program such that all relevant events are monitored accurately, which is crucial to the accuracy of D^{2}ABS. We used Soot [47] for the instrumentation in two main steps. First, D^{2}ABS inserts probes for the three types of method execution events in each method, for which we reused relevant modules of DIVER [22], a hybrid impact analysis that is built on Soot and uses method execution event traces also. The second step is to insert probes for message-passing events, for which D^{2}ABS uses the list L of specified message-passing APIs to identify probe points: L includes the prototype of each API used in the input system for network I/Os.

If L is not specified, a list of basic Java network I/O APIs is used covering two common means of blocking and non-blocking communication: Java Socket I/O [48] and Java NIO [49] (both are *socket-based*). While not immediately addressing implicit dependencies in distributed programs by itself, this API list is critical for the effectiveness of dependence analysis in our framework: it immediately affects whether D^{2}ABS can completely and precisely capture message-passing events hence how accurately it can model interprocess control and data flow. Fortunately, it seemed to suffice that our implementation handles the most commonly used network I/O mechanisms for message passing, at least in Java distributed software—for the diverse systems in our evaluation study, we did not need to specify the list but just used the built-in support of our framework implementation.

For the **Csd+** version, the static analyzer proceeds with constructing the MDG for each component of the input program, by reusing the algorithms for abstracting method-level dependencies in single-process programs [17], [30]. The MDG includes three classes of threading-induced dependencies: *ready*, *synchronization*, and *interfere* dependencies, computed partly in reference to the implementation of the Indus project [43]. The MDGs are serialized to disk, to be used by the post-processor.

Run-time monitors. The two sets of run-time monitors implement the two online algorithms: the first focuses on monitoring method execution events and the second is dedicated to preserving the partial ordering of them. The first set again reuses relevant parts of DIVER [22]. For the second set, instead of invoking additional network I/O API calls to transfer logical clocks, the monitors *take over* the original message passing so that they can piggyback the three extra data items (i.e., the data length, logical clock, and sender id) to the original message. In the message-passing event monitors, we carefully manage these extra data items with sophistication in order to support non-blocking I/Os (e.g., Java NIO) and account for complications and variety in communication implementations of real-world distributed systems (e.g., those that would cause interference with original communication semantics as discussed in Section 6.1.1).

Post-processor. The post-processer is the module that actually answers dependence analysis queries. It starts by gathering distributed traces with a helper script which passes per-process traces to the offline dependence-computation algorithm. To compute the dependence set following Algorithm 2, the post-processor retrieves the partial ordering of method execution events by comparing the associated timestamps. For the **Csd+** and **Scov+** versions, the MDG is deserialized from the disk file dumped by the static analyzer. Additionally for the **Scov+** version, branches covered are retrieved from the traces as well and covered statements are identified using control dependencies in the MDG: if a branch is covered, all statements that are control dependent on that branch are all considered covered. Since our MDG construction addresses exception-driven control flows as well, coverage of exception-handling constructs is handled by our statement coverage computation as well.

8 EVALUATION

We evaluate D^{2}ABS in the context of its application for dynamic impact analysis, with which an impact query corresponds to a dependence query (a dependence of $m2$ on $m1$ implies $m1$ impacts $m2$). We seek to answer the following research questions:

- **RQ1**: Are **Msg+** and the two new levels of abstraction in D^{2}ABS more effective than the basic DISTIA [26]?
- **RQ2**: Which kind of program information used by D^{2}ABS contributes the most to its effectiveness?
- **RQ3**: How efficient and scalable are the different versions of D^{2}ABS in terms of various analysis overheads?
- **RQ4**: Does D^{2}ABS as a whole offer variable cost-effectiveness tradeoffs hence accommodate diverse use scenarios?

TABLE 4: Statistics of experimental subjects

Subject (version)	#SLOC	#Methods	Test type	#Cov.M.
MultiChat (r5)	470	37	integration	25
NIOEcho (r69)	412	27	integration	26
xSocket (v2.8.15)	15,890	2,204	integration	391
Thrift (v0.11.0)	12,366	1,459	integration	266
Open Chord (v1.0.5)	38,084	736	integration	354
ZooKeeper (v3.4.11)	62,450	4,813	integration	749
			system	817
			load	798
Voldemort (v1.9.6)	163,601	17,843	integration	2,048
			system	1,242
			load	1,323
Freenet (v0.7.0)	196,281	16,673	integration	2,477

8.1 Experiment Setup

We evaluated D^{2}ABS on eight distributed Java programs, as summarized in Table 4. The size of each subject is measured by the number of non-comment non-blank Java source lines of code (*#SLOC*) and number of methods defined in the subject (*#Methods*) that we actually analyzed. The last two columns list the type of test input used in our study (one test case per type) and the number of methods executed at least once in the respective test (*#Cov.M.*). We used each of these methods as a dependence query. For each subject and input type, the ratio of the fifth column to the third column gives the method-level coverage of the test input (e.g., the method-level coverage of the integration test input for Thrift is 266/1,459).

8.1.1 Subject Systems

We chose these subjects such that varied system scales and architectures, application domains, and uses of either and both of blocking and non-blocking I/Os are all considered.

- MultiChat [50] is a chat application where multiple clients exchange messages via a server broadcasting the message sent by one client to all others.
- NioEcho [51] is an echo service via which the client just gets back the same message as it sends to the server.
- xSocket [52] is an NIO-based library for building high-performance network applications.
- Thrift [53] is a framework for scalable cross-language services development.
- Open Chord is a peer-to-peer lookup service based on distributed hash table [54].
- ZooKeeper [44], [55] is a coordination service for distributed systems to achieve consistency and synchronization.
- Voldemort [56] is a distributed key-value storage system used at LinkedIn.
- Freenet [57] is a peer-to-peer data-sharing platform offering anonymous communication.

Some of these systems use Socket I/O or Java NIO only, while others use both mechanisms, for message passing among their components. For all subjects, we checked out from their official repositories the latest stable versions or revisions as shown in (the parentheses of) Table 4.

8.1.2 System Executions

We chose test inputs to cover different types of inputs when possible, including system test, integration test, and load test, which we assume exercise typical, overall system behaviour instead of a small specific area of the system. The integration tests were created manually as elaborated below, while the other types of inputs come with the subjects from their respective repositories.

In each integration test, we started two to five server and client nodes on different machines and performed client operations that cover main system services—specially for peer-to-peer systems, we operated on all nodes, and for ZooKeeper we started a container node in addition.

- For MultiChat and NioEcho, the client requests were sending random text messages.
- For ZooKeeper, the ordered client operations were: create two nodes, look up for them, check their attributes, change their data association, and delete them.
- For Open Chord, the operations were in order: create an overlay network on machine (node) A, join the network on machines B and C, insert a new data entry to the network on C, look up and then delete the data entry on A, and list all data entries on B.
- For Voldemort, the operations were: add a key-value pair, query the key for its value, delete the key, and retrieve the pair again.
- For Freenet, we first uploaded a file to the network with a note on node A, shared it to nodes B and C, then on B and C accepted the sharing request and the note, followed by downloading the file and replying with a note.
- The remaining two subjects, xSocket and Thrift, are frameworks/libraries, thus we needed to develop user applications for exercising these subjects. Each of the two applications consists of two components, a server and a client. For xSocket, one client sends a text message to the server, followed by the second client sending a different message to the server. The Thrift application implements a calculator, for which the operations were addition, subtraction, multiplication, and division of two numbers.

Our D^{2}ABS implementation handled the varied system architectures and network I/O mechanisms represented by the chosen subject systems, including blocking and non-blocking message passing among the distributed components of these systems. Thus, no user-specified list of message-passing APIs was needed for our evaluation experiments. Given the diversity of our study subjects, this implies that users of D^{2}ABS would commonly not need to specify the optional API list L (Figure 3).

8.2 Experimental Methodology

To answer our research questions, we evaluate D^{2}ABS as a holistic framework through assessing the cost and effectiveness of its four instantiations (Section 6). In particular, we compare the **Msg+** version and the two newly added versions, together noted as *advanced versions*, against the **Basic** version (i.e., basic DISTIA [26]) as the baseline, so as to understand the contribution of message-passing semantics, static dependencies, and coverage data to effectiveness improvements and overhead increases in D^{2}ABS. We considered every method of each subject as a query, yet we report results only for queries executed at least once in one process—only such queries have a non-empty impact set. For a method executed in more than one processes, we took it as executed in each process as a separate query.

For each query, we measure the effectiveness of D^{2}ABS by comparing it to the baseline in terms of of impact-set (dependence-set) size ratios, and examine the composition of the impact set concerning its two subsets: *local impact set* (i.e., local dependence set) and *remote impact set* (i.e., remote dependence set). For average-case analyses, we added the set of common impacts (i.e., common dependence set) into both of the two subsets. Accordingly, we measure the effectiveness of D^{2}ABS with respect to two subsets (local and remote impact sets) relative to the corresponding baseline results as well. In essence, we assess precision improvement through impact-set reduction as a *relative* measure. We use the relative effectiveness measures for two reasons. First, computing the (precision and/or recall) measures in absolute terms requires ground-truth dependence sets of all possible queries, which are neither available for our subject systems and executions nor automatically computable (due to the lack of capable tools available to us). Second, given that the baseline has been evaluated with respect to its effectiveness before [26], the main goal of this evaluation is to assess the relative improvements of the advanced versions over the baseline, for which the relative measures are sufficient. Besides the impact-querying time, we report the static-analysis and run-time costs of D^{2}ABS, and storage costs, all as efficiency metrics.

For a fair comparison, all versions used the same test input for computing the impact set of each query. This was realized by first recording our manual inputs per subject and test type and then replaying accordingly across different D^{2}ABS versions. The machines used were all Linux workstations with an Intel i5-2400 3.10GHz CPU and 8GB DDR2 RAM.

8.3 Results and Analysis

In this section, we report and discuss the empirical results of our evaluation on D^{2}ABS, focusing on the effectiveness and efficiency of D^{2}ABS separately and both together as a cost-effectiveness measure, in order to answer the four research questions.

8.3.1 RQ1: Effectiveness

Figure 6 illustrates the effectiveness results of D^{2}ABS, with every single plot depicting the result (i.e., relative precision) distribution for each subject and input type together shown as the plot title (hereafter, the input type is omitted for subjects for which only an integration test is available and utilized). Each chart includes at most nine boxplots showing that data distribution for one of three categories (on x axis): the holistic impact set (*A* for all) and its two subsets (*L* for local and *R* for remote), each produced by one of the three advanced versions of D^{2}ABS if applicable. Two exceptions were Voldemort and Freenet, for which the **Csd+** results are missing because computing the static intra-component dependencies did not finish after ten hours (thus we terminated the analysis). The same was applied to **Scov+** since it subsumes **Csd+** in terms of the inclusion of the static dependence analysis.

Particularly in Figure 6, for each query, the common dependencies were removed from the two (i.e., local and remote) subsets of the impact set. This is done for a sanity check of the pruning strategy of each advanced version: **Msg+** only prunes remote dependencies while the other two only prune local dependencies. Each underlying data item of the chart indicates the effectiveness metric (i.e., impact-set size ratio to the baseline, as shown on y

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

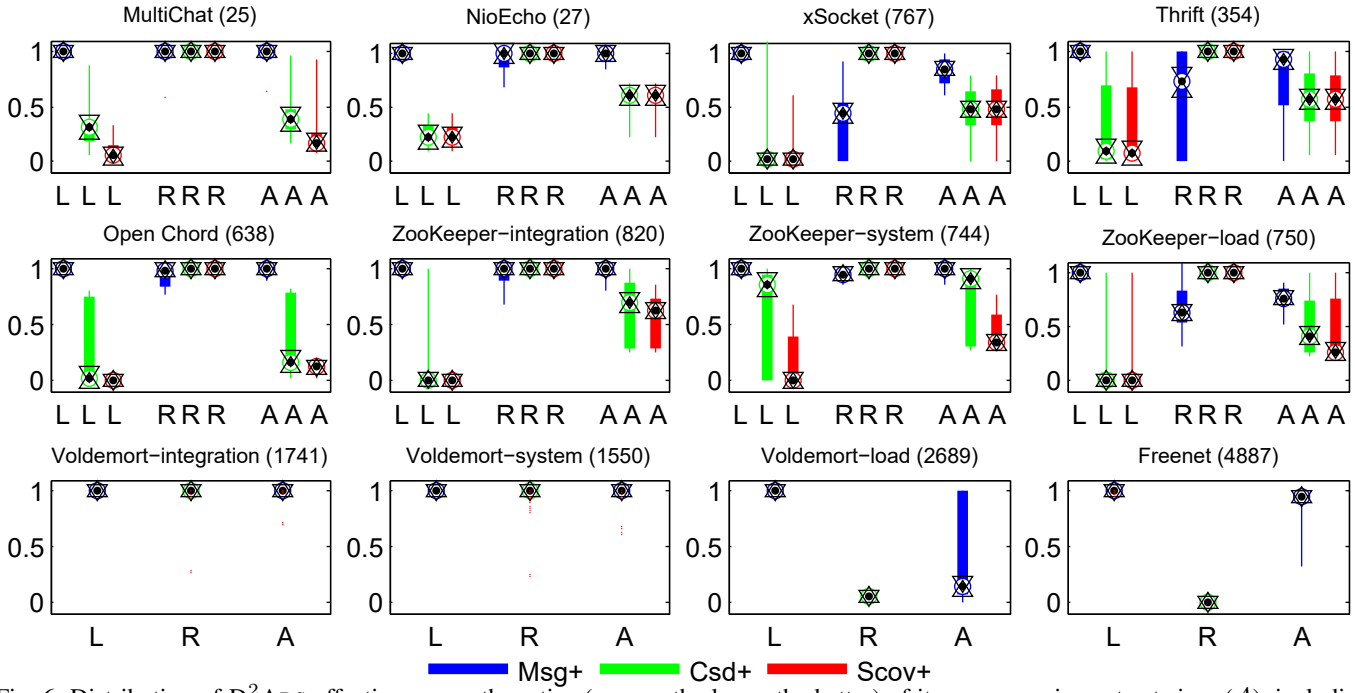


Fig. 6: Distribution of D^{2}ABS effectiveness as the ratios (y axes, the lower the better) of its per-query impact-set sizes (A), including those of the local (L) and remote (R) subsets (x axes), to the baseline, per subject and input type (with #queries in parentheses).

axis) for one query. The total numbers of queries involved are shown in the parentheses. In each boxplot, the circled dot within the bar represents the median. To facilitate the comparison of effectiveness differences among the three D^{2}ABS versions, we also showed in each boxplot a pair of triangular marks, which indicate the comparison interval of the median. The statistical meaning of these intervals is: the medians of any two groups are significantly different at the 5% significance level if their intervals do not overlap.

The results indicate that all the three advanced versions are noticeably more effective than the baseline (basic DISTIA). Incorporating the message-passing semantics reduced the baseline impact sets by 5% up to over 20% in most cases (according to the 75% quartiles). Compared to the two smallest subjects, the large, real-world systems saw generally higher effectiveness improvements by the **Msg+** version. Examining the two subsets of each impact set confirms that **Msg+** only reduced remote impact sets only while not changing the local impact sets relative to the baseline, consistent with the rationale of this technique. Note that the ratios with respect to *all* (rightmost three boxplots of each chart) impact sets are mostly higher than those with respect to the *remote* subsets. This is because each *all* impact set includes the corresponding common impact set as well, while sizes of the common impact sets of **Msg+** are consistently smaller than those of the baseline (due to smaller remote impact sets).

The **Csd+** and **Scov+** versions of D^{2}ABS further enhanced the baseline effectiveness by pruning only local impact sets, according to the analysis algorithm (Algorithm 2). Just as **Msg+** version did not change the local impact sets, the results confirmed that neither **Csd+** nor **Scov+** changed the remote impact sets. Since the static dependencies incorporated in **Csd+**, and statement coverage in **Scov+** additionally, are both limited to individual components (Equations 4 and 5), the reduction of overall impact sets by **Csd+** and **Scov+** was ascribed to the decreases in local impact-set sizes. Again, here we are looking at the two subsets with common dependencies removed; if we look at the the complete local and remote impact sets, any of the three advanced versions would have reduced both local and remote impact sets in most cases, because reducing one of the two subsets led to the reduction of the common set hence the reduction of the other subset.

The impact-set size ratio distribution of **Csd+** and **Scov+** (i.e., the two framework instantiations newly introduced in this paper) shows their substantial advantage over both the baseline and the **Msg+** version. In all of the subjects to which these two new versions were applied, utilizing local static dependencies largely cut off the baseline local impact sets, leading to drastic reduction of the overall baseline impact sets by 20% to over 50% for the majority of the queries (per the 75% quartiles). For instance, for Open Chord (first row, first column of Figure 6), the impact-set size ratio (over the baseline) by **Msg+** centered around 90% for virtually any query, while the ratio by **Csd+** was below 50% for 75% of the queries. As expected, leveraging statement coverage data moved further along in pruning the local impact sets. The results demonstrated substantial improvements of **Scov+** over **Csd+** for most queries in most subjects, which is particularly true of larger subjects (e.g., Open Chord and ZooKeeper) compared to smaller ones (e.g., xSocket and Thrift).

Complementary to Figure 6, Table 5 gives the mean effectiveness results (computed from the same individual data points which the figure depicts the distribution for). Results of the two new D^{2}ABS versions against Voldemort and Freenet were missing because of the reason mentioned above. On average, **Msg+** was not always effective, with a worse case of reporting 99% of the baseline impact set for Open Chord. Although it attained the highest reduction of over 50% (for Voldemort-load), the reduction was less than 20% in all other cases. In comparison, **Csd+** pruned baseline impact sets mostly by 40% or more, and 30% in the worse case. The highest effectiveness was achieved by **Scov+**

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

TABLE 5: Mean effectiveness improvement (in terms of impact-set size ratios) of the three new instances of D^{2}ABS over the baseline

Subject & test input	Average baseline effectiveness (impact-set sizes)			Mean effectiveness improvement versus baseline								
				Msg+ version			Csd+ version			Scov+ version		
	Local	Remote	All	Local	Remote	All	Local	Remote	All	Local	Remote	All
MultiChat	14.0	4.5	18.5	100%	96.28%	97.12%	36.85%	100%	49.27%	19.21%	100%	34.77%
NioEcho	11.4	11.3	21.7	100%	93.35%	96.59%	38.69%	100%	66.20%	38.07%	100%	65.85%
xSocket	203.2	281.1	356.3	100%	38.62%	84.30%	12.45%	100%	49.40%	11.46%	100%	49.08%
Thrift	80.7	102.1	128.0	100%	60.60%	80.80%	33.59%	100%	56.76%	31.17%	100%	55.50%
Open Chord	248.0	245.9	276.0	100%	93.36%	98.95%	48.32%	100%	54.76%	0.61%	100%	14.60%
ZooKeeper-integration	284.1	265.5	492.1	100%	94.46%	96.96%	29.77%	100%	70.03%	0.51%	100%	62.96%
ZooKeeper-system	687.1	641.3	929.5	100%	95.12%	97.23%	55.31%	100%	70.64%	18.13%	100%	44.49%
ZooKeeper-load	577.1	560.7	837.8	100%	66.65%	82.73%	15.50%	100%	48.58%	7.25%	100%	47.79%
Voldemort-integration	654.6	578.1	1052.7	100%	89.13%	95.69%	-	-	-	-	-	-
Voldemort-system	522.1	478.6	842.9	100%	86.97%	94.01%	-	-	-	-	-	-
Voldemort-load	286.9	709.9	967.6	100%	4.42%	47.91%	-	-	-	-	-	-
Freenet	2787.8	2780.0	2956.6	100%	3.11%	93.24%	-	-	-	-	-	-
weighted average	**1191.6**	**1251.8**	**1494.7**	**100%**	**41.06%**	**84.85%**	**31.86%**	**100%**	**58.75%**	**9.95%**	**100%**	**46.02%**

on Open Chord, with an over 85% reduction of baseline results on average. Our manual inspection of the code of this subject in comparison to other subjects revealed that it contains much more extensive control structures exercised during the analyzed executions, which justifies the best effectiveness of statement-coverage-based pruning for this subject.

The last row of the table shows the weighted (by the number of queries) averages of per-subject effectiveness measures. Given the differences in the subjects' code and executions, these overall averages may not well represent an average-case situation across the subjects. Nevertheless, these numbers provide a summary metric to facilitate comparisons. In all, **Msg+**, **Csd+**, and **Scov+** pruned 15%, 41%, and 54% of baseline results, respectively, on average across all applicable cases. Given the safety of all resulting impact sets produced by any of the four compared techniques (Section 5.2), these reduction ratios are translated to precision improvements by 17.6%, 69.5%, 117.4% (in terms of ratios rather than absolute precision values)[4], respectively.

In practical application scenarios, these effectiveness improvements imply that developers can save the time that would be spent on inspecting up to half of the impacts/dependencies given by the baseline approach. The first four columns of Table 5 list the baseline impact set sizes, which inform the implications and significance of these inspection-effort savings. For instance, for ZooKeeper-load, **Csd+** would save developers effort on examining 430 (false-positive) methods. On overall average, **Msg+**, **Csd+**, and **Scov+** would save such efforts for inspecting 226, 617, and 807 methods that are false positives, respectively. The numbers for the local and remote impact sets further validated our expectations on where (local/remote) each advanced version would prune the baseline results the most. The **Msg+** local impact sets are always the same (100%) as the baseline's, so are the remote impact sets from **Csd+** and **Scov+**. This consistency further validated the analysis algorithms underlying these three advanced versions.

RQ1: The three advanced instances of D^{2}ABS achieved substantial effectiveness (precision) improvements over DISTIA, with an impact-set size reduction of 15% (with **Msg+**), 41% (with **Csd+**), and 54% (with **Scov+**). The implication of these improvements is to save developers' effort of examining 226 to 807 false-positive methods in the baseline impact sets.

8.3.2 RQ2: Contributing Factors

To understand the contributing factors in the effectiveness improvements of D^{2}ABS over the baseline, we examined the effectiveness results again but with a focus on the contrast among the three advanced D^{2}ABS versions evaluated against the baseline: **Msg+**, **Csd+**, and **Scov+**. The motivation for understanding these contributing factors is to gain knowledge on the pros and cons of varied design decisions (in terms of data use) in dynamic dependence analysis of distributed programs.

As shown in Table 5, the two versions that utilize static dependencies (**Csd+** and **Scov+**) are both considerably more effective than **Msg+** (achieving 37% and 26% smaller impact sets on overall weighted average, respectively, as shown in the bottom row of the table). In comparison, **Msg+** is more effective than the baseline, with much smaller magnitude of improvements though (14% smaller impact sets on average). Thus, in terms of average-case precision, it appeared that message-passing semantics contributed less than static dependencies. This is intuitively because message-passing semantics itself is a very coarse form of (component/process-level) information. Also, normally message passing occurs among distributed processes fairly often and bidirectionally, as we observed in our subjects' executions. As a result, the constraint (Equation 3) is generally easy to satisfy. Comparing between **Csd+** and **Scov+** indicates that adding more program information (i.e., statement coverage) led to further precision improvements. On the other hand, the gain of 11% is only half of that (26%) achieved by **Csd+** over **Msg+**. Thus, static dependencies seem to contribute more than statement coverage too. Not only were such contrasts among the three advanced versions observed in an average case, Figure 6 revealed similar contrasts for the majority of individual queries in all relevant subjects individually.

Figure 6 also enables statistical comparisons of the contributions made by varied forms of program information used in D^{2}ABS. Comparing the medians (circled dots) across the three advanced versions for each subject and test type further corroborates the considerable effectiveness advantage of **Scov+** over **Csd+**, and the even greater improvements of **Csd+** over **Msg+**. These comparison intervals indicate that, in a median case, (1) **Csd+** was *significantly* more effective than **Msg+** in 7 out of the 8 applicable cases (with the only exception of ZooKeeper-system), and (2) **Scov+** was *significantly* more effective than **Csd+** for 3 of the 8 cases (MultiChat, ZooKeeper-system, and ZooKeeper-load).

Beyond the average- and median-case comparisons, we further conducted two statistical analyses: (1) paired Wilcoxon signed-rank tests [58] to assess the statistical significance (at the 0.95 confidence level) of effectiveness differences among the four D^{2}ABS versions, and (2) effect sizes in terms of Cliff's Delta [59] (in a paired setting with $\alpha = .05$) to assess the magnitude of the effectiveness differences. In both analyses, the two groups

4. An impact-set size ratio r corresponds to a precision increase by (1-r)/r.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

TABLE 6: p-values and effect sizes (in parentheses) with respect to impact-set sizes between all pairs among the four D^{2}ABS instances

Subject & test input	baseline:Msg+	baseline:Csd+	baseline:Scov+	Msg+:Csd+	Msg+:Scov+	Csd+:Scov+
MultiChat	5.62E-01 (.08)	**6.20E-05** (.68)	**2.62E-06** (.68)	**1.17E-04** (.60)	**4.55E-06** (.68)	**6.30E-04** (.80)
NioEcho	2.92E-01 (.33)	**2.31E-09** (.93)	**7.97E-10** (.96)	**2.48E-09** (.93)	**7.37E-10** (.96)	9.58E-01 (.04)
xSocket	**5.29E-38** (.71)	**1.12E-181** (.86)	**1.08E-181** (.86)	**1.11E-133** (.70)	**1.08E-133** (.70)	1.00E+00 (.00)
Thrift	**3.58E-18** (.56)	**1.22E-63** (.77)	**1.67E-67** (.80)	**2.17E-24** (.50)	**6.35E-27** (.53)	6.34E-01 (.10)
Open Chord	6.33E-02 (.25)	**3.47E-158** (.81)	**3.25E-243** (.99)	**2.93E-157** (.80)	**3.52E-243** (.99)	**6.62E-87** (.79)
ZooKeeper-integration	**5.34E-03** (.28)	**1.01E-40** (.40)	**4.68E-35** (.36)	**5.35E-50** (.42)	**3.88E-48** (.42)	**2.01E-12** (.55)
ZooKeeper-system	**1.49E-13** (.64)	**7.97E-88** (.63)	**2.31E-221** (.94)	**5.15E-57** (.47)	**9.35E-221** (.94)	**5.26E-48** (.63)
ZooKeeper-load	**9.83E-106** (.56)	**5.50E-163** (.80)	**7.31E-154** (.78)	**1.05E-69** (.47)	**1.01E-64** (.45)	1.20E-01 (.47)
Voldemort-integration	**4.95E-12** (.18)	-	-	-	-	-
Voldemort-system	**8.34E-15** (.20)	-	-	-	-	-
Voldemort-load	**0.00E+00** (.58)	-	-	-	-	-
Freenet	**5.96E-168** (.49)	-	-	-	-	-

were the impact-set sizes given by each pair of techniques compared. Both analyses are nonparametric, allowing us to lift the assumption about the normality of the distribution of underlying data points. Table 6 lists the Wilcoxon p values along with corresponding effect sizes (in parentheses) for all relevant comparison groups for all applicable subjects and test input types. Statistically significant results are highlighted in boldface. We differentiate four levels of effect strength as per Cliff's Delta (d) values [60]: *negligible* ($|d|\leq 0.147$), *small* ($0.147<|d|\leq 0.33$), *medium* ($0.33<|d|\leq 0.474$), and *large* ($|d|>0.474$).

The results (3rd to 6th columns of Table 6) indicate that the two new D^{2}ABS versions (**Csd+** and **Scov+**) were strongly significantly more effective, with mostly very strong (large) effect sizes, than both the baseline and **Msg+**. Meanwhile, **Msg+** impact set sizes were significantly different from (smaller than) those of the baseline for all cases but the two smallest subjects and Open Chord. Also, the majority of these significant cases came with a large effect size. These observations with **Msg+** and the baseline were similar to those between **Csd+** and **Scov+**. In all, results of the statistical analyses corroborated what we observed from the earlier comparisons on medians and averages about the strengths of the effects of various program information on D^{2}ABS's effectiveness. On the other hand, the numbers of Table 6 indicate significant (in terms of p values) and large (in terms of effect sizes) improvements of the three advanced versions of D^{2}ABS over the basic DISTIA.

RQ2: Among the various forms of program information used by D^{2}ABS, static dependencies combined with statement coverage contributed the most to its effectiveness, followed by static dependencies alone and then by message-passing semantics. The intra-component dependencies are the most contributing, single form of data, implying that incorporating static dependence analysis may benefit greatly to the precision of dynamic dependence analysis of distributed programs.

8.3.3 RQ3: Efficiency

Tables 7 and 8 list all relevant costs of the baseline and the three advanced versions of D^{2}ABS, respectively. The costs reported include the time cost of static analysis, run-time overhead measured as ratios of the execution time of the original program (*Original run*) over the execution time of the instrumented one (*Instr. run*), and impact querying time. The time costs are reported in milliseconds (ms), and storage costs in KB which include the disk space taken by serialized MDGs (only for **Csd+** and **Scov+**) and execution traces (for all of the four instantiations).

With the **Basic** version (baseline), the static analysis generally took longer for larger subjects, as expected, yet still below 3 minutes even on the largest system Freenet. Run-time and querying

TABLE 7: Efficiency results of the baseline analysis

Subject & input	Original run (ms)	time costs in milliseconds (ms)				space (KB)
		Static analysis	Instr. run	Runtime overhead	Querying (stdev)	
MultiChat	5,461	12,817	5,735	5.02%	4 (2)	7
NioEcho	3,213	13,365	3,619	12.64%	4 (2)	5
xSocket	7,753	24,842	8,470	9.25%	6 (2)	85
Thrift	9,751	23,143	10,241	5.03%	7 (1)	18
Open Chord	4,856	14,533	4,931	1.54%	8 (5)	73
ZooKeeper-integration	37,239	39,124	38,396	3.11%	10 (2)	94
ZooKeeper-system	15,385		18,565	20.67%	24 (6)	132
ZooKeeper-load	94,187		98,891	4.99%	22 (5)	142
Voldemort-integration	17,755	132,536	18,662	5.11%	22 (7)	315
Voldemort-system	11,136		12,232	9.84%	19 (4)	197
Voldemort-load	21,066		21,198	0.63%	29 (5)	782
Freenet	54,794	165,174	61,876	12.92%	114 (17)	527
Overall average	**33,213.7**	**73,854.9**	**36,273.6**	**8.07%**	**49.3 (44.9)**	**395.7**

costs are consistently correlated to subject sizes as well as the type of inputs and the size of execution traces (the seventh column), with the worst case seen by ZooKeeper and Freenet, respectively. Nevertheless, the run-time overhead was at worst 21% and the longest querying time was just a tenth of one second. The last row (of Tables 7) gives the average costs weighted by the number of queries studied in each subject and input type. Overall, the baseline needed only 74 seconds for static analysis and incurred only 8% runtime overhead. Both the querying and storage costs were negligible.

The **Msg+** version shares the same instrumentation as the **Basic** version (second column of Table 8). The additional per-process message-receiving maps incurred negligible overheads. Thus, we did not show the cost breakdown but only the querying time for this framework instantiation. Our results show the additional data utilized only caused very small increase in the querying costs, which remained well below one second per query. The efficiency results of both the **Basic** and **Msg+** versions were highly consistent with those obtained in our preliminary studies [26].

The **Csd+** version incurred much higher costs in any of the three phrases of D^{2}ABS than the baseline and **Msg+**. Since static dependencies are not just affected by the size of a program but more by the its logic complexity, the static-analysis costs were not linearly correlated to subject sizes. Moreover, very-large systems (Voldemort and Freenet) were not successfully analyzed by this version as mentioned earlier, thus we missed the results accordingly. Among the six remaining subjects, the largest took the longest time of about 53 minutes. This should be readily acceptable for systems at such a scale. Also, this is a one-time cost (for the single program version analyzed by D^{2}ABS) as the instrumented code and the constructed MDG can be reused for any inputs and for computing any queries afterwards. The runtime overheads were considerably larger too, almost 5x with Open Chord, since **Csd+** requires full sequences of method execution events. Further, traversing the MDG and longer traces led to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

TABLE 8: Time cost breakdown (in milliseconds) and storage cost (in KB) of the three advanced versions of D^{2}ABS

Subject & input	Msg+	Csd+ time costs in milliseconds				Csd+	Scov+ time costs in milliseconds				Scov+
	querying (stdev)	Static analysis	Instr. run	Runtime overhead	Querying (stdev)	Storage costs (KB)	Static analysis	Instr. run	Runtime overhead	Querying (stdev)	Storage costs (KB)
MultiChat	4 (2)	175,474	25,298	363.25%	46 (21)	84	221,839	26,246	380.61%	19 (7)	85
NioEcho	5 (2)	277,903	8,147	153.56%	52 (19)	133	367,268	9,716	202.40%	34 (12)	136
xSocket	8 (2)	781,925	19,156	147.08%	394 (114)	6,332	859,158	21,683	179.67%	256 (39)	6,540
Thrift	9 (7)	401,621	21,397	119.43%	775 (48)	4,407	462,480	23,924	145.35%	337 (17)	4,438
Open Chord	10 (6)	196,958	29,078	498.81%	21,643 (16,291)	4,109	264,532	37,319	668.51%	100 (256)	4,173
ZooKeeper-integration	12 (2)		84,063	125.74%	8,855 (15)	15,689		95,520	156.51%	8,850 (15)	15,708
ZooKeeper-system	28 (5)	3,188,103	37,704	145.07%	21,737 (7,208)	28,402	3,374,107	48,119	212.77%	23,233 (216)	36,804
ZooKeeper-load	28 (8)		303,009	221.71%	16,313 (939)	28,412		412,846	338.33%	13,988 (1,773)	35,058
Voldemort-integration	25 (7)		-	-	-	-		-	-	-	-
Voldemort-system	22 (5)	-	-	-	-	-	-	-	-	-	-
Voldemort-load	42 (13)		-	-	-	-		-	-	-	-
Freenet	141 (24)	-	-	-	-	-	-	-	-	-	-
Overall average	**61.3 (56.1)**	**846,958.6**	**88,705.8**	**209.42%**	**12,411.8 (11,619.7)**	**15,599.8**	**914,834.8**	**114,809.6**	**283.91%**	**4,897.6 (5,444.4)**	**18,378.6**

much higher querying costs. Nevertheless, even as the worse case, querying the impact set of a method in ZooKeeper against the system test took 22 seconds, which is reasonably affordable. On overall (weighted) average, **Csd+** took 14 minutes for static analysis and 12 seconds for answering an impact-set query with 2x run-time overhead, over the 8 cases it was applied to.

The **Scov+** version adds the collection and use of statement coverage data to **Csd+**, thus it is supposedly the most heavyweight instance of our framework. Our results, however, show that this additional data did not incur much additional static analysis costs or run-time slowdown (68 seconds and 0.7x, respectively). Noticeably, not only did the statement coverage contribute significantly to the effectiveness of our framework (see RQ2), the additional static analysis and run-time overheads are also paid off by the savings of querying costs. The statement coverage data helped prune the static dependencies before they were used for dynamic dependence analysis (see Lines 6 and 14 of Algorithm 2). As a result, the querying costs incurred by **Scov+** were only 40% of those by **Csd+** on overall average. As for **Csd+**, storage costs of **Scov+** were also negligible (37MB in the worse case, and 18.4MB on overall average).

In sum, the **Basic** and **Msg+** versions were highly efficient in both time and space dimensions, thus they are readily scalable to large distributed systems. The **Csd+** and **Scov+** versions were considerably more expensive, mainly because of the substantial cost of static dependence analysis. Nevertheless, for systems of small to medium levels of size and complexity, they still provide compelling options, offering much higher effectiveness at reasonable costs. To very large and complex systems, more efficient static dependence analysis would be necessary to scale up these more precise D^{2}ABS versions.

RQ3: The **Basic** and **Msg+** versions of D^{2}ABS were highly efficient and scalable to large distributed systems, taking less than 1.5 minutes for static analysis and less than 1 second for querying, with 8% run-time overhead. The **Csd+** and **Scov+** versions were significantly more expensive, yet their costs remain reasonable and practically acceptable for systems of small to medium levels of size and complexity, with 15 minutes for static analysis, 12 seconds for answering a query, and 2.8x overhead on average. Storage costs were negligible for the entire framework.

8.3.4 RQ4: Cost-Effectiveness Tradeoffs

In practice, developers need to consider both the cost and effectiveness of an analysis tool (i.e., the balance between these two factors) to make their decisions on tool selection [13], [37], [61], [62], [63]. To investigate the capabilities of our framework in offering variable cost-effectiveness, we put together the effectiveness improvements and overhead increases of the three advanced versions over the baseline. Specifically, we compute the cost-effectiveness as the ratio of improvement percentage of mean precision (from Table 5) to the increase factor of average costs (from Tables 7 and 8), relative to the baseline. We consider two different classes of cost separately: (1) the per-query cost for answering each impact-set query, which is different from query to query, and (2) the one-time cost for static analysis and profiling, which is incurred once for all queries. Storage costs are not considered here since they were all quite trivial in all cases with any of the four D^{2}ABS versions.

Figure 7 shows the cost-effectiveness (y axis) comparisons among the three advanced versions for all the cases each version was applied to (as listed on the x axis), for the querying cost (top) and one-time costs combined (bottom). To better differentiate the results of the three techniques compared, the cost-effectiveness is shown in a logarithmic scale (with a base 10) after the effectiveness improvement percentage was enlarged by 1,000 times. As shown, in terms of querying costs, **Msg+** appeared to be the most cost-effective version for all cases but the two smallest subjects and Open Chord (for which **Scov+** was the most cost-effective). This is mainly because of the highly lightweight nature of **Msg+**. As a result, while the costs of the two new D^{2}ABS versions were not very high in an absolute term, the ratios of these costs to those of **Msg+** were large, outweighing their precision improvement percentages over **Msg+**. Among the two new versions, however, **Scov+** was consistently most cost-effective than **Csd+**, due to the substantially higher precision yet relatively small cost increases of **Scov+** over **Csd+**.

Concerning the costs of the first two phases combined, the general contrasts among the three advanced versions were similar to what we observed in the cases in which only querying costs are considered in the cost-effectiveness measure. Except for Open Chord, **Msg+** always outperformed the other two versions. The reason mainly lies in the costs of static dependence analysis incurred by **Csd+** and **Scov+** being substantially larger than even the total cost of **Msg+**. On the other hand, the advantage of **Scov+** over **Csd+** was of a lesser extent (albeit still noticeable) in contrast to their comparisons when only querying costs were concerned. This is because **Scov+** was always more expensive than **Csd+** for the first two phases.

Overall, **Msg+** achieved the highest cost-effectiveness among the three advanced versions due to its high efficiency and scalability, along with significant precision advantage over the baseline.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3124795, IEEE Transactions on Software Engineering

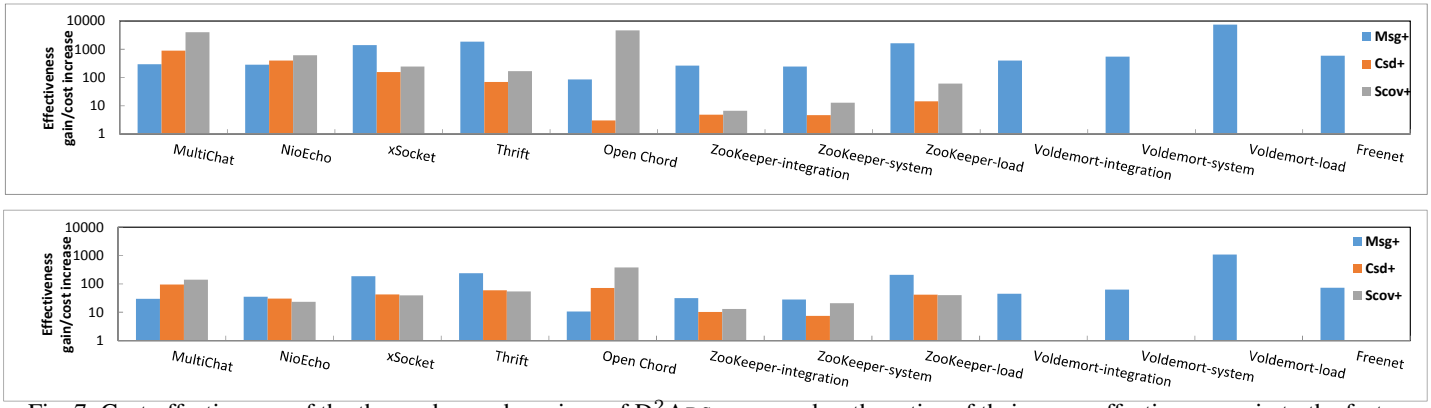


Fig. 7: Cost-effectiveness of the three advanced versions of D^{2}ABS expressed as the ratios of their mean effectiveness gain to the factor of increase in the average querying cost (top) and total cost of the first two phases (bottom), both against the baseline DISTIA. The higher the ratio, the better (more cost-effective).

The two new versions, **Csd+** and **Scov+**, remain practically cost-effective options, especially for software systems of medium or smaller sizes and complexity or for users who need much higher precision even at the cost of considerably higher overheads. In particular, **Scov+** was consistently more cost-effective than **Csd+**. Thus, where having less false negatives are of a higher priority, **Scov+** would generally be a better option.

RQ4: D^{2}ABS provides variable and flexible cost-effectiveness balance options to accommodate diverse needs. The **Basic** and **Msg+** versions provide a relatively rough but rapid solution, between which **Msg+** offers the best cost-effectiveness when high precision is not a priority; otherwise, **Scov+** is the most cost-effective technique, especially for systems to which it scales. These tradeoffs provide guidance for developers to choose which D^{2}ABS version to use in varied situations.

8.4 Threats to Validity

The main threat to *internal* validity lies in possible errors in our D^{2}ABS implementation and experiment scripts. To reduce this threat, we did a careful code review for our tools and manually validated the correctness of their functionalities and analysis results against the two smallest subjects. An additional such threat concerns possible missing (remote) impacts due to network I/Os that were not monitored at runtime. However, we checked the code of all subjects and confirmed that they only used the most common message-passing APIs monitored by our tool when executing the program inputs we utilized. In general, for arbitrary distributed systems, the accuracy of D^{2}ABS relies on the identification of all such API calls used in the system. Yet another threat is that there might be hidden dependencies between methods induced by external storage I/Os: for example, a method reading a disk file is hidden-dependent on a method that writes the same file. Similar hidden dependencies can also induced by database accesses. Currently, D^{2}ABS does not consider such hidden dependencies; instead, it focuses on dependencies among code entities due to memory accesses and network communications.

The main threat to *external* validity is that our study results may not generalize to all other distributed programs and input sets. Dynamic analysis is *commonly* subject to the limited coverage of the run-time inputs that drove the analyzed execution. Nevertheless, the limited size and representativeness of the input set available to us and used for each subject in our evaluation study constituted a validity threat. To reduce this threat, we have chosen subject programs of various sizes and application domains, including the six industry-scale systems in different domains. In addition, we considered different types of inputs, including integration, system, and load tests whenever they were available. Many of these tests came as part of the subjects, except for the integration tests which we created according to the official online documentation (quick-start guide) of these systems.

The main threat to *construct* validity concerns the metrics used for the evaluation. Without directly comparable peer techniques in the literature, we used the **Basic** version of our framework as the baseline approach to assess the improvements made by the advanced versions of the same framework. While it is intuitive to compare D^{2}ABS with a technique that it extends, this choice of baseline might cause potential biases. Also, we did not have the ground-truth impact sets to compute precision and recall metrics in absolute terms. A precise statement-level forward dynamic slicer would be able to generate the ground-truth needed, yet such a slicer is not currently available to us. Thus, we used the impact-set size ratio as a *relative* effectiveness measure. Yet, referring to the *same* baseline enabled our comparison of the three advanced versions of D^{2}ABS. While not ideal, such relative comparisons and measures suffice for our goal of understanding how the varied design factors in the dynamic dependence analysis of distributed programs affect the cost-effectiveness tradeoffs.

Finally, a *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). Also, the present study only considered potential changes in single methods for each query, while in practice developers may plan for changes in multiple methods at a time, which may lead to different results. To minimize this threat, we adopted the strategy for all experiments and calculated the metrics for every possible query. To reduce possible biases in our statistical analyses, we chose non-parametric hypothesis testing and effect size measures that do not rely on the normality of underlying data's distribution.

9 Discussion

In this section, we discuss two additional pertinent issues with our technical approach: validation of D^{2}ABS's analysis results and selection of D^{2}ABS instantiations for practical use.

9.1 Result Validation

As discussed in Section 6.1.4, the analysis results of D^{2}ABS are in general unsafe because its analyses are unsound overall. In principle, this unsafety *implies* imperfect recall. Yet in practice, perfect recall can still be obtained, at least for queries that do not involve dependencies induced by dynamic language constructs (i.e., reflection and JNI in our case of Java). Also, as discussed in Sections 8.4, due to the lack of ground truth, we could not compute recall and precision in absolute terms. Instead, we gained confidence about recall of each advanced version of D^{2}ABS as per the conservative nature of its dependence pruning relative to the baseline, and used the impact-set size ratios, also relative to the baseline, to measure precision indirectly. These strategies suffice for the purpose of studying the improvements of these advanced versions and their underlying techniques over the baseline, which is indeed our main goal with this work.

Nevertheless, it is still important to understand the actual analysis accuracy of the dependence sets produced by (any instantiation of) our framework. Unfortunately, we are not aware of any existing automated tool that can compute the ground-truth dependence set of any given query against the distributed programs and their executions used in our evaluation; neither is it possible to produce all such ground truth manually. Thus, we extended the manual inspection adopted for the preliminary version of this work [27], following a prior methodology aiming at a similar purpose [12] as summarized as follows.

For each of our studied subjects and executions, we randomly chose ten queries for which none of the D^{2}ABS instantiations produced a dependence set that includes more than 50 methods—we had to limit the scale of this manual study because of its tedious and heavyweight nature. Then, for each chosen query, we manually produced the ground-truth dynamic dependence set by in-depth code review and step-through tracking of the executions (like step-over debugging) while leveraging available documentation of respective subject systems. Using these ground-truth dependencies, we computed the precision and recall for each of the chosen queries.

With these cases, the average (over all the subject executions and the ten queries per execution) precision of **Basic**, **Msg+**, **Csd+**, and **Scov+** was 55.1%, 64.8%, 93.4%, and 98.6%, respectively. Importantly, for all these cases, the recall of any instantiation was 100%—indeed, the dependence set of each of these sample queries did not include any dependencies induced by reflective or JNI calls. This confirmed that the pruning carried out by the three advanced versions only removed false-positive dependencies.

Beyond these quantitative results, our manual study revealed that even in system executions where there were extensive message exchanges among the distributed processes, message passing that constrained interprocess data flow (see Figure 4) was not prevalent. This helped explain the observation that pruning spurious dependencies computed by **Basic** based on message-passing semantics was limited, hence the moderate precision improvement of **Msg+** over **Basic**. On the other hand, we frequently observed that there was not any (static) control or data dependence between methods that were partially ordered (via method-level control flow). This was why incorporating static dependencies in **Csd+** helped gain a leap in precision over **Basic** and **Msg+**. Finally, we found it was not very common in our studied cases that method invocations are guarded by predicates, which justifies pruning the static dependencies through statement coverage only led to a relatively lesser advantage of **Scov+** over **Csd+**.

9.2 Instantiation Selection

As shown, our framework offers flexible options of cost-effectiveness tradeoffs through the four instantiations each providing a distinct level of such tradeoffs. For a user with a particular task that relies on the dynamic dependencies computed by D^{2}ABS, a practical problem is to how to choose the right instantiation (tool) out of the varied alternatives. Thus, we make recommendations in this regard as guidelines as follows.

Figure 7 suggests that the most cost-effective option varied both with different subject systems (of varying code size and complexity) and with different executions (of varying execution complexity) of the same systems. Yet the order of the four D^{2}ABS instances in terms of effectiveness (precision) and (time/space) cost as earlier presented in Figure 5 and justified in Section 6 still holds as corroborated by our empirical results (e.g., those for RQ1 and RQ3). Overall, for systems and executions of a small to medium size and complexity, the user might want to choose **Scov+** given that it tended to offer the highest cost-effectiveness in most cases in our evaluation study. For large-scale and highly-complex distributed systems and executions, though, the user would be most likely recommended to opt for **Msg+** given the highest level of cost-effectiveness it offered.

However, there are also situations in which the user's best option may not be the most cost-effective one. For instance, the user would choose the instantiation that offers higher precision for the system and execution on hand as long as the added costs are still affordable, even though the extra costs are not best paid off. For another example, the fastest instantiation might be the best choice to the user when the small efficiency advantages (over the second fastest) matter, although this fastest tool is not as cost-effective as other options. In these situations, the user should choose the instantiation that satisfies the cost or effectiveness priority with respect to the particular task. Note that the presence of these situations justifies not only offering the most cost-effective analysis techniques but also providing those of other levels of cost-effectiveness, just as D^{2}ABS did.

10 Related Work

In preliminary work [26], we developed an early prototype of D^{2}ABS that includes the the **Basic** and **Msg+** versions. This paper extends that prior work both technically by adding **Csd+** and **Scov+**, and empirically through a larger-scale and more extensive evaluation. Beyond dynamic impact analysis exemplified as one of its applications, we also have applied D^{2}ABS to the run-time behavior characterization [64], dynamic slicing [65], and dynamic information flow analysis [66] of distributed systems. A recent work on automatic, on-the-fly tuning of the cost-effectiveness trade-offs in dynamic dependence analysis of distributed programs [67] was also built on top of D^{2}ABS. In comparison, this paper provides an official formulation of dynamic dependence analysis of distributed software as a much needed foundation that underlies all of those (dependence-based) applications.

Other prior works most related to ours fall in four categories: *dependence analysis of distributed programs*, *impact analysis* (as an application of dependence analysis), *logging for distributed systems*, and *dynamic partial order reduction*.

10.1 Dependence Analysis of Distributed Programs

Historically, dependence analysis has underlaid a wide range of code-based software engineering tasks and associated tech-

niques [4], [5]. In particular, a large body and variety of dependence-based approaches have been proposed over the past few decades to support development and maintenance in general [3], [68], [69], [70], [71], [72], and fault diagnosis [73], [74], [75] and security defense in particular [76], [77], [78]. For instance, for maintaining and evolving a software system, it is essential to assess the influence of given program entries of interest on the rest of the program [70], [79] for change planning (deciding whether to realize changes at candidate change locations) and fulfillment (deciding where to realize the changes) [20], [36], [79].

Using fine-grained dependency analysis, a large body of work attempted to extend traditional slicing algorithms to concurrent programs [38], [43], [80], [81], [82] yet mostly focused on centralized, and primarily multithreaded, ones. For those programs, traditional dependence analysis was extended to handle additional dependencies due to shared variable accesses, synchronization, and communication between threads and/or processes (e.g., [38], [80]). While D^{2}ABS also handles multithreaded programs, it targets multiprocess ones running on distributed machines, and aims at method-level dependence analysis instead of fine-grained slicing. For systems running in multiple processes where interprocess communication is realized via socket-based message passing, an approximation for static slicing was discussed in [80]. Various dynamic slicing algorithms have been proposed too, earlier for procedural programs only [32], [83], [84], [85], [86] and recently for object-oriented software also [14], [15], [16]. And a more complete and detailed summary of slicing techniques for distributed programs can be found in [82] and [15].

A few other static analysis algorithms for distributed systems exist as well but focus on other (special) types of systems, such as RMI-based Java programs [87], different from the common type of distributed systems [1] D^{2}ABS addresses. At coarser levels, researchers resolve dependencies in distributed systems too but for different purposes such as enhancing parallelization [88], system configuration [89], and high-level system modeling [11], [90], or limited to static analysis [6], [7], [10]. In contrast, D^{2}ABS performs code-based analysis while providing more focused dependencies (impacts) relative to concrete program executions than static-analysis approaches.

Existing static [23] and dynamic [24], [91] dependence analyses (e.g., static/dynamic slicing) for single-threaded programs are related to the computation of intra-thread dependencies in D^{2}ABS. While it might be appealing to extend these analyses for distributed programs (e.g., via the happens-before relations among methods executed across processes), that may not be expected to work practically against real-world distributed systems. Among the presented D^{2}ABS instances, **Scov+** is the closest to those analyses in terms of granularity and program data use. Nevertheless, **Scov+** does a much more lightweight approximation of the dependencies those analyses would compute. Yet even **Scov+** did not scale to large systems like Voldemort and Freenet. Thus, the extensions of those analyses will give cost-effectiveness options where the cost may be too high to be practically adoptable. This was why we did not incorporate such analyses and provide corresponding cost-effectiveness options in our framework.

10.2 Impact Analysis

The EAS approach [19] which partially inspired D^{2}ABS is a performance optimization of its predecessor PATHIMPACT [18]. Many other dynamic impact analysis techniques also exist [70], aiming at improving precision [22], recall [92], efficiency [93], and cost-effectiveness [13], [25] over PATHIMPACT and EAS. However, these techniques did not address distributed or multiprocess programs that we focus on in this work. Two recent advances in dynamic impact analysis, DIVER [22] and the unified framework in [13], [25], utilize hybrid program analysis to achieve higher precision and more flexible cost-effectiveness options over EAS-based approaches, but still target single-process programs only. On the other hand, the **Csd+** version of D^{2}ABS was initially motivated by these prior works to incorporate the static intra-component dependencies for effectiveness improvement.

An impact analysis for distributed systems, Helios [7] can predict impacts of potential changes to support evolution tasks for DEBS. However, it relies on particular message-type filtering and manual annotations in addition to a few other constraints. Although these limitations are largely lifted by its successor Eos [10], both approaches are *static* and limited to DEBS only, as is another technique [8] which identifies impacts based on change-type classification yet ignores intra-component dependencies hence provides incomplete results. While sharing similar goals, D^{2}ABS targets a broader range of distributed systems than DEBS using *dynamic* analysis and without relying on special source-code information (e.g., interface patterns) as those techniques do. Unlike our dependence-based approach, a traceability-based solution [79] is presented in [94] which relies on a well-curated repository of various software models. The dynamic impact analysis for component-based software in [95] works at architecture level, different from ours working for distributed programs at code level.

10.3 Logging for Distributed Systems

Targeting high-level understanding of distributed systems, techniques like logging and mining run-time logs [11], [35] infer inter-component interactions using textual analysis of system logs, relying on the availability of particular data such as informative logs and/or patterns in them. D^{2}ABS utilizes similar information (i.e., the Lamport timestamps) but infers the happens-before relation between method execution events mainly for code-level dependence analysis. Also, D^{2}ABS automatically generates such information it requires rather than relying on existing information in the original programs.

The Lamport timestamp used is related to vector clocks [96], [97] used by other tools, such as ShiVector [90] for ordering distributed logs and Poet [98] for visualizing distributed systems executions. While we could utilize vector clocks also, we chose the Lamport timestamp as it is lighter-weight yet suffices for D^{2}ABS. In addition, unlike ShiVector, which requires accesses to source code and recompilation using the AspectJ compiler, D^{2}ABS does not have such constraints as it works on bytecode.

Tracing message passing was explored before but for different purposes, such as overcoming non-determinacy/race detection [99] and reproducing buggy executions [100]. Tools like RoadRunner [101] and ThreadSanitizer [102] targeted multithreaded, centralized programs. None of these solutions work for large, diverse distributed systems without perturbations as D^{2}ABS did.

10.4 Dynamic Partial Order Reduction (DPOR)

DPOR has been used to improve the performance of model checking concurrent software by avoiding the examination of independent transitions [103]. Sharing the spirit of DPOR, especially distributed DPOR [104], D^{2}ABS synchronizes method

execution events for efficient computation of partial ordering of methods *within* processes, and exploits message-passing semantics to avoid partially ordering independent methods *across* processes. However, unlike existing DPOR techniques which target multi-threaded programs, D^{2}ABS focuses on multi-process, distributed programs. On the other hand, DPOR may be adopted for dynamic dependence analysis of distributed programs with extensions/adaptations. First, the execution conditions (e.g., entered, returned) of methods can be modeled as method-level states (as opposed to statement-level states in the original DPOR). Second, to represent state transitions across processes, the identifiers of parent processes may prefix thread identifiers. Finally, partial ordering algorithms like LTS may be used to determine transition dependence at the process level. Accordingly, independent transitions can be identified to help further identify methods that have no dependence between them, so as to speed up the dependence computation overall.

11 Conclusion

Modeling and reasoning about program dependencies has long been a fundamental approach to many advanced techniques and tools for various software engineering tasks, ranging from testing and debugging to performance optimizations and security defense. However, traditional dependence models and analysis algorithms, which assume explicit references among code entities, cannot be readily applied to distributed systems with full potential, because the architectural design of these systems encourages implicit references among decoupled components via networking facilities such as socket.

To unleash the power of dependence analysis for distributed software, we presented D^{2}ABS, a framework for dynamic dependence analysis that safely approximates run-time code dependencies at the method level both within and across process boundaries. D^{2}ABS offers cost-effective dependence analysis by partially ordering method execution events, exploiting message-passing semantics, incorporating static intra-component dependencies, and leveraging whole-system statement coverage data. Blending multiple forms of program information, D^{2}ABS offers flexible cost-effectiveness balances via four instantiations, to accommodate varying time and other resource budgets in diverse task scenarios in distributed software development and maintenance.

We motivated and described the design of this framework in its application for dynamic impact analysis, and evaluated its effectiveness and efficiency also in the context of impact prediction. Our empirical results with large real-world distributed programs have shown the superior effectiveness of advanced D^{2}ABS versions over its basic, purely control-flow based version as a baseline by safely reducing false positives of the baseline by 15% to 54% at the cost of varied but reasonable analysis overheads. There are many applications of D^{2}ABS beyond impact analysis. As immediately next steps, we plan to apply the framework for performance diagnosis and security defense of distributed systems.

References

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, 2011.

[2] J. Dean, "Designs, lessons and advice from building large distributed systems," *Keynote from LADIS*, pp. 1–73, 2009.

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

[4] 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*, vol. 16, no. 9, pp. 965–979, 1990.

[5] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 1992, pp. 392–411.

[6] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 262–292, 1996.

[7] D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic, "Impact analysis for distributed event-based systems," in *Proceedings of International Conference on Distributed Event-Based Systems*, 2012, pp. 241–251.

[8] S. Tragatschnig, H. Tran, and U. Zdun, "Impact analysis for event-based systems using change patterns," in *Proceedings of ACM Symposium on Applied Computing*, 2014, pp. 763–768.

[9] K. Jayaram and P. Eugster, "Program analysis for event-based distributed systems," in *Proceedings of International Conference on Distributed Event-Based System*, 2011, pp. 113–124.

[10] J. Garcia, D. Popescu, G. Safi, W. G. Halfond, and N. Medvidovic, "Identifying message flow in distributed event-based systems," in *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, 2013, pp. 367–377.

[11] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2014, pp. 468–479.

[12] H. Cai, "Hybrid program dependence approximation for effective dynamic impact prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 334–364, 2018.

[13] H. Cai, R. Santelices, and D. Thain, "Diapro: Unifying dynamic impact analyses for improved and variable cost-effectiveness," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 2, p. 18, 2016.

[14] D. P. Mohapatra, R. Kumar, R. Mall, D. Kumar, and M. Bhasin, "Distributed dynamic slicing of Java programs," *Journal of Systems and Software*, vol. 79, no. 12, pp. 1661–1678, 2006.

[15] S. S. Barpanda and D. P. Mohapatra, "Dynamic slicing of distributed object-oriented programs," *IET software*, vol. 5, no. 5, pp. 425–433, 2011.

[16] S. Pani, S. M. Satapathy, and G. Mund, "Slicing of programs dynamically under distributed environment," in *Proceedings of International Conference on Advances in Computing*. Springer, 2012, pp. 601–609.

[17] H. Cai and R. Santelices, "Method-level program dependence abstraction and its application to impact analysis," *Journal of Systems and Software*, vol. 122, pp. 311–326, 2016.

[18] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2003, pp. 308–318.

[19] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2005, pp. 432–441.

[20] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, 2012, pp. 51:1–51:11.

[21] H. Cai and R. Santelices, "A comprehensive study of the predictive accuracy of dynamic change-impact analysis," *Journal of Systems and Software*, vol. 103, pp. 248–265, 2015.

[22] ——, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of International Conference on Automated Software Engineering*, 2014, pp. 343–348.

[23] M. Acharya and B. Robinson, "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems," in *Proceedings of IEEE/ACM International Conference on Software Engineering, Software Engineering in Practice Track*, May 2011, pp. 746–765.

[24] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2003, pp. 319–329.

[25] H. Cai and R. Santelices, "A framework for cost-effective dependence-based dynamic impact analysis," in *International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 231–240.

[26] H. Cai and D. Thain, "DistIA: A cost-effective dynamic impact analysis for distributed programs," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 344–355.

[27] ——, "Distea: Efficient dynamic impact analysis for distributed systems," *arXiv preprint arXiv:1604.04638*, 2016.

[28] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[29] D. Jackson and M. Rinard, "Software analysis: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 133–145.

[30] H. Cai and R. Santelices, "Abstracting program dependencies using the method dependence graph," in *International Conference on Software Quality, Reliability and Security (QRS)*, 2015, pp. 49–58.

[31] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto, "Software model checking for distributed systems with selector-based, non-blocking communication," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 169–179.

[32] D. Goswami and R. Mall, "Dynamic slicing of concurrent programs," in *IEEE International Conference on High Performance Computing*, 2000, pp. 15–26.

[33] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems*. Springer, 2006, vol. 1.

[34] P. Eugster and K. Jayaram, "EventJava: An extension of Java for event correlation," in *Proceedings of European Conference on Object-Oriented Programming*. Springer, 2009, pp. 570–594.

[35] J.-G. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured logs analysis," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 91–96, 2010.

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

[37] C. R. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 241–250.

[38] M. G. Nanda and S. Ramesh, "Interprocedural slicing of multithreaded programs with applications to Java," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 6, pp. 1088–1144, 2006.

[39] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Trans. Softw. Eng. Method.*, vol. 10, no. 2, pp. 209–254, 2001.

[40] E. Pitt and K. McNiff, *Java. RMI: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[41] J. Siegel and D. Frantz, *CORBA 3 fundamentals and programming*. John Wiley & Sons New York, NY, USA:, 2000, vol. 2.

[42] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. on Prog. Lang. and Systems*, vol. 12, no. 1, pp. 26–60, 1990.

[43] D. Giffhorn and C. Hammer, "Precise slicing of concurrent programs," *Automated Software Engineering*, vol. 16, no. 2, pp. 197–234, 2009.

[44] Apache, "ZooKeeper," https://zookeeper.apache.org/, 2015.

[45] A. Beszedes, C. Farago, Z. Mihaly Szabo, J. Csirik, and T. Gyimothy, "Union slices for program maintenance," in *Proceedings of IEEE International Conference on Software Maintenance*, Oct. 2002, pp. 12–21.

[46] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto." *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[47] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a Java bytecode optimization framework," in *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[48] Oracle, "Java Socket I/O," http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html, 2015.

[49] ——, "Java NIO," http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html, 2015.

[50] GoogleCode, "MultiChat," https://code.google.com/p/multithread-chat-server/, 2015.

[51] SourceForge, "NioEcho," http://rox-xmlrpc.sourceforge.net/niotut/index.html#Thecode, 2015.

[52] Vice, "xSocket," http://xsocket.org/, 2018.

[53] Apache, "Thrift," https://thrift.apache.org/, 2018.

[54] Bamberg University, "Open Chord," http://sourceforge.net/projects/open-chord/, 2015.

[55] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *Proceedings of USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[56] Apache, "Voldemort," https://github.com/voldemort, 2015.

[57] The Freenet team, "The Free Network," https://freenetproject.org/, 2015.

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

[59] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 1996.

[60] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*. Citeseer, 2006, pp. 1–51.

[61] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[62] M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change?" *Journal of Systems and Software*, vol. 43, no. 1, pp. 19–27, 1998.

[63] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 307–317.

[64] X. Fu and H. Cai, "Measuring interprocess communications in distributed systems," in *IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2019, pp. 323–334.

[65] X. Fu, H. Cai, and L. LI, "Dads: Dynamic slicing continuously-running distributed programs with budget constraints," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tool*, 2020, pp. 1566–1570.

[66] X. Fu and H. Cai, "A dynamic taint analyzer for distributed systems," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tool*, 2019, pp. 1115–1119.

[67] X. Fu, H. Cai, W. Li, and L. LI, "Seads: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, 2020, (impact factor 2.5; journal-first paper).

[68] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *Software Engineering, IEEE Transactions on*, vol. 17, no. 8, pp. 751–761, 1991.

[69] J. P. Loyall and S. A. Mathisen, "Using dependence analysis to support the software maintenance process," in *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*. IEEE, 1993, pp. 282–291.

[70] 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, pp. 613–646, 2013.

[71] A. Orso, S. Sinha, and M. J. Harrold, "Classifying Data Dependences in the Presence of Pointers for Program Comprehension, Testing, and Debugging," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 2, pp. 199–239, 2004.

[72] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 2009, pp. 10–19.

[73] G. K. Baah, A. Podgurski, and M. J. Harrold, "The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[74] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Proc. of Symp. on Principles of Program Lang.*, Jan. 1993, pp. 384–396.

[75] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug 2016.

[76] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 482–493.

[77] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis." in *OSDI*, vol. 10, 2010, pp. 1–14.

[78] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Acm Sigplan Notices*, vol. 47, no. 7. ACM, 2012, pp. 121–132.

[79] S. A. Bohner and R. S. Arnold, *An introduction to software change impact analysis*. Software Change Impact Analysis, IEEE Comp. Soc. Press, pp. 1–26, Jun. 1996.

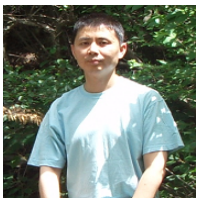
[80] J. Krinke, "Context-sensitive slicing of concurrent programs," in *Proceedings of ACM International Symposium on the Foundations of Software Engineering*, vol. 28, no. 5, 2003, pp. 178–187.

[81] J. Xiao, D. Zhang, H. Chen, and H. Dong, "Improved program dependence graph and algorithm for static slicing concurrent programs," in *Advanced Parallel Processing Technologies*. Springer, 2005, pp. 121–130.

[82] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[83] B. Korel and R. Ferguson, "Dynamic slicing of distributed programs," *Applied Mathematics and Computer Science*, vol. 2, no. 2, pp. 199–215, 1992.
[84] J. Cheng, "Dependence analysis of parallel and distributed programs and its applications," in *Proceedings of Advances in Parallel and Distributed Computing*, 1997, pp. 370–377.
[85] E. Duesterwald, R. Gupta, and M. Soffa, "Distributed slicing and partial re-execution for distributed programs," in *Languages and Compilers for Parallel Computing*. Springer, 1993, pp. 497–511.
[86] M. Kamkar and P. Krajina, "Dynamic slicing of distributed programs," in *Proceedings of IEEE International Conference on Software Maintenance*, 1995, pp. 222–229.
[87] M. Sharp and A. Rountev, "Static analysis of object references in RMI-based Java software," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 664–681, 2006.
[88] K. Psarris and K. Kyriakopoulos, "An experimental evaluation of data dependence analysis techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 3, pp. 196–213, 2004.
[89] F. Kon and R. H. Campbell, "Dependence management in component-based distributed systems," *IEEE concurrency*, vol. 8, no. 1, pp. 26–36, 2000.
[90] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst, "Shedding light on distributed system executions," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 598–599.
[91] W. Masri, N. Nahas, and A. Podgurski, "Memoized forward computation of dynamic slices," in *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*. IEEE, 2006, pp. 23–32.
[92] 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 *Proceedings of European Conference on Software Maintainance and Reengineering*, 2010, pp. 252–255.
[93] B. Breech, M. Tegtmeyer, and L. Pollock, "A comparison of online and dynamic impact analysis algorithms," in *Proceedings of European Conference on Software Maintainance and Reengineering*, 2005, pp. 143–152.
[94] H. M. Sneed, "Impact analysis of maintenance tasks for a distributed object-oriented system," in *Proceedings of the IEEE International Conference on Software Maintenance*, 2001, p. 180.
[95] T. Feng and J. I. Maletic, "Applying dynamic change impact analysis in component-based architecture design," in *ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2006, pp. 43–48.
[96] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, 1988, pp. 56–66.
[97] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.
[98] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten, "Poet: Target-system independent visualizations of complex distributed-application executions," *The Computer Journal*, vol. 40, no. 8, pp. 499–512, 1997.
[99] R. H. Netzer and B. P. Miller, "Optimal tracing and replay for debugging message-passing parallel programs," *The Journal of Supercomputing*, vol. 8, no. 4, pp. 371–388, 1995.
[100] R. Konuru, H. Srinivasan, and J.-D. Choi, "Deterministic replay of distributed java applications," in *International Parallel and Distributed Processing Symposium*, 2000, pp. 219–227.
[101] C. Flanagan and S. N. Freund, "The roadrunner dynamic analysis framework for concurrent programs," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2010, pp. 1–8.
[102] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71.
[103] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005, pp. 110–121.
[104] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Distributed dynamic partial order reduction based verification of threaded software," in *Model Checking Software*. Springer, 2007, pp. 58–75.

Haipeng Cai is an assistant professor of computer science at Washington State University, Pullman. He obtained his PhD in computer science from University of Notre Dame in 2015. His current research interests are software engineering and software systems with a focus on program analysis and its applications to software reliability and security.



Xiaoqin Fu is currently pursuing his PhD in computer science at Washington State University, Pullman. His research is focused on developing program analysis techniques to support software maintenance and evolution of distributed systems.