SEADS: Scalable and Cost-effective Dynamic Dependence Analysis of Distributed Systems via Reinforcement Learning

XIAOQIN FU, HAIPENG CAI, and WEN LI, School of Electrical Engineering and Computer Science, Washington State University
LI LI, Faculty of Information Technology, Monash University

Distributed software systems are increasingly developed and deployed today. Many of these systems are supposed to run continuously. Given their critical roles in our society and daily lives, assuring the quality of distributed systems is crucial. Analyzing runtime program dependencies has long been a fundamental technique underlying numerous tool support for software quality assurance. Yet conventional approaches to dynamic dependence analysis face severe scalability barriers when they are applied to real-world distributed systems, due to the unbounded executions to be analyzed in addition to common efficiency challenges suffered by dynamic analysis in general.

In this article, we present SEADS, a *distributed*, *online*, and *cost-effective* dynamic dependence analysis framework that aims at scaling the analysis to real-world distributed systems. The analysis itself is distributed to exploit the distributed computing resources (e.g., a cluster) of the system under analysis; it works online to overcome the problem with unbounded execution traces while running continuously with the system being analyzed to provide timely querying of analysis results (i.e., runtime dependence set of any given query). Most importantly, given a user-specified time budget, the analysis automatically adjusts itself to better cost-effectiveness tradeoffs (than otherwise) while respecting the budget by changing various analysis parameters according to the time being spent by the dependence analysis. At the core of the automatic adjustment is our application of a reinforcement learning method for the decision making—deciding which configuration to adjust to according to the current configuration and its associated analysis cost with respect to the user budget. We have implemented SEADS for Java and applied it to eight real-world distributed systems with continuous executions. Our empirical results revealed the efficiency and scalability advantages of our framework over a conventional dynamic analysis, at least for dynamic dependence computation at method level. While we demonstrate it in the context of dynamic dependence analysis in this article, the *methodology* for achieving and maintaining scalability and greater cost-effectiveness against continuously running systems is more broadly applicable to other dynamic analyses.

CCS Concepts: • **Theory of computation → Program analysis**;

Additional Key Words and Phrases: Distributed systems, cost-effectiveness, scalability, dynamic analysis, reinforcement learning

This work was partially supported by the National Science Foundation through the grant CCF-1936522.
Authors' addresses: X. Fu, H. Cai (corresponding author), and W. Li, School of Electrical Engineering and Computer Science, Washington State University, 355 NE Spokane St., Pullman, WA, 99163; emails: {xiaoqin.fu, haipeng.cai, li.wen}@wsu.edu; L. Li, Faculty of Information Technology, Monash University, Wellington Rd, Clayton VIC, Melbourne, Australia, 3800; email: li.li@monash.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
1049-331X/2020/12-ART10 $15.00
https://doi.org/10.1145/3379345

ACM Reference format:
Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. 2020. SEADS: Scalable and Cost-effective Dynamic Dependence Analysis of Distributed Systems via Reinforcement Learning. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 10 (December 2020), 45 pages.
https://doi.org/10.1145/3379345

1 INTRODUCTION

In response to scientific and societal demands on the scalability of data storage and computation, a growing number of modern software systems are *distributed* by design [22] (i.e., they are designed to leverage decentralized, high-performance computing infrastructure and resources). In particular, most (if not all) of these systems are supposed to be running continuously to provide an uninterrupted service. Examples of such distributed systems include financial/banking systems, web search, airline management systems, medical networks, and so on. Given their critical roles in our society and daily lives, the quality (e.g., reliability, resiliency, and security) of distributed systems is of paramount importance.

A fundamental strategy for understanding and validating software behaviors is to model runtime interactions among program entities as *dependencies* and then reason about program behaviors based on the dependence model [12, 45, 48]. Historically, dynamic dependence modeling and analysis [40, 61] served as the underpinning of a variety of code-based quality-assurance tool support, ranging from fault diagnosis [83] to security defense [7, 47]. Dynamic dependence analysis is important, because many application techniques in software quality assurance rely on dynamic dependence information, such as program optimization, performance monitoring, software testing, vulnerability detection, and so on [50]. For instance, the dependencies computed by a dynamic dependence analysis can be used to detect runtime sensitive data leaks. Essentially, to find where the leakages are, we would check if there are any sinks that are reachable from any sources via any chains of the dependencies. Similarly, software testing is also crucial for software quality assurance, for which dynamic dependencies can be utilized to detect defects in the software by searching among the dependencies of the program entities where faulty outputs are observed.

Compared with static approaches, dynamic dependence analysis has greater precision, as it focuses on specific, concrete executions. Developing a cost-effective dynamic dependence analysis, however, is challenging, especially given the known substantial overheads of dynamic analysis in general. Recent research has demonstrated the difficulties and complexity in balancing the cost and effectiveness in dynamic dependence analysis for single-process programs [18]. Developing such an analysis for most real-world distributed systems is even more challenging because of their typically larger size and greater complexity. Execution non-determinism, the variety of and uncertainties in runtime environments of distributed systems, and the unbounded executions (due to their continuously running nature) further exacerbate such challenges. As a motivating case, we recently applied an existing state-of-the-art dynamic dependence analysis approach for distributed programs [15] to Voldemort [4], a real-world, industry-scale distributed system (key-value store), for three minutes of its execution. The analysis did not finish after running for 12 hours on a high-performance server. Apparently, with this level of efficiency, current approaches are neither practically cost-effective nor scalable for distributed systems.

In this article, we address *common distributed systems*—continuously running distributed software whose constituent components[1] are (1) decoupled by networking facilities, (2) running in

[1]A component in such a system is defined as the collection of code entities that run in a single process separately from the rest of the software, with either one or more threads.

concurrent processes, and (3) communicating through message passing without a global timing mechanism. Our overarching goal is to unleash the power of dynamic dependence analysis to enable scalable tool support for assuring the quality of distributed systems. To that end, we have developed SEADS,[2] a dynamic dependence analysis framework that offers practical scalability and cost-effectiveness tradeoffs for distributed systems. Our analysis exploits both code (i.e., *static*) and execution (i.e., *dynamic*) data of a given system to compute runtime code dependencies, with varying analysis parameters (e.g., flow/context sensitivity of the static analysis and data granularity of the dynamic analysis) to enable variable cost-effectiveness tradeoffs during the dependence computation. The framework works purely at application level, only using data it produces by itself (e.g., gathering dynamic data via static instrumentation). In particular, SEADS realizes a novel dynamic analysis paradigm featuring three defining characteristics:

- First, SEADS is *distributed*. Our framework answers users' dependence queries by performing analyses both within a single process and across multiple processes (referred to as *intraprocess* analysis and *interprocess* analysis, respectively). The SEADS framework consists of one computing node that runs the interprocess analysis and a number of different other computing nodes each running the intraprocess analysis for one of the components/processes of the system under analysis (*SUA*). These intraprocess analyses constantly communicate with each other to synchronize their logic clocks to enable the interprocess analysis. This architecture brings two advantages of SEADS compared to conventional program analysis that typically runs in a centralized fashion (i.e., at one machine): (1) It naturally leverages the distributed computing resources of the SUA to gain in efficiency, and (2) by running intraprocess analysis that is only relevant to a component at the machine that runs the component, unnecessary network communications and consequent analysis delays are avoided, leading to further analysis accelerations.
- Second, SEADS is *online* and *continuous*. Given the uninterruptedly running nature of the kind of SUAs we target, an offline dynamic analysis would be largely impeded by the infeasibility of collecting and analyzing the unbounded execution traces. While offline analysis of partial execution traces is useful (for which SEADS can accommodate as well), SEADS particularly focuses on scenarios in which whole program execution traces need to be analyzed while the SUA runs continuously—for these situations, an offline analysis would be unscalable or even infeasible. An online design of the analysis in SEADS addresses this challenge by avoiding tracing that involves disk I/Os. Further, the continuous nature of SEADS enables on-demand querying capabilities that are desirable for a continuously running SUA.
- Third, SEADS is *cost-effective*. While the distributed architecture and online design contribute to its scalability, SEADS's primary scalability enabler is to continuously adjust its analysis configuration in an automated manner to achieve better cost-effectiveness tradeoffs. These tradeoffs are realized through the parameters (i.e., configuration items) of the static and dynamic analyses in SEADS that fall in multiple dimensions. There are two dimensions of static analysis parameters: *data_selection* and *sensitivity*. The *data_selection* dimension, including one parameter (*staticGraph*), concerns whether static data (i.e., static dependence) are used, while the *sensitivity* dimension, including two parameters (*context sensitivity* and *flow sensitivity*), influences the precision of the static dependence computation. Dynamic analysis parameters also fall in two dimensions: *data_selection* and *data_granularity*. The *data_selection* dimension, including two parameters (*methodEvent* and/or *statementCoverage*), determines which types of dynamic data are used, while the *data_granularity* dimension, including one parameter (*MethodInstanceLevel*), concerns the granularity of the dynamic data used in

[2]Short for Scalable and cost-Effective dynamic dependence Analysis of Distributed Systems.

SEADS. Automatically adjusting these analysis parameters is essential for addressing the potential challenge of heavy memory load with an online dynamic analysis, and the need for providing cost-effective results to user queries that come with a response time constraint (i.e., user budget for the average time cost of answering a query). This signature capability of SEADS is mainly offered by its novel design that automatically learns the best analysis configurations at different time during the SUA execution according to current configurations and associated costs with respect to the user-specified budget, using a reinforcement learning (Q-learning) methodology.

To evaluate the benefits and limitations of our framework, we implemented SEADS as a practical tool for Java. By applying it against eight real-world distributed systems, our evaluation revealed the efficiency and scalability merits of SEADS over a state-of-the-art conventional dynamic dependence analysis approach, at least for method-level dependence computation. With 10 randomly chosen queries and querying intervals of random lengths for each subject, SEADS demonstrated its compelling cost-effectiveness (i.e., competitive precision of the resulting runtime dependence sets at generally low time costs). Specifically, SEADS took on average 65 seconds for each query with negligible storage costs and less than 1× runtime slowdown (both in most cases and on average). In comparison, the online version of the state-of-the-art dynamic dependence analysis used as our baseline cannot answer any user queries within 12 hours for one of the subjects, while taking 197 seconds on average for each query with other subjects at a heavy overhead of up to 6× (and on average 3.3×) runtime slowdown. Meanwhile, SEADS achieved the scalability and substantially higher efficiency while achieving 82% of the precision attained by the baseline. As a result, the cost-effectiveness of the baseline is only 44% and 32% of that achieved by SEADS with respect to average response time and runtime overhead (slowdown), respectively.

Through SEADS, we contribute a methodology for (1) making a hybrid approach to dynamic dependence analysis *learnable* by decomposing the analysis algorithm into multiple dimensions each having a unique impact on the analysis cost and effectiveness, and (2) learning configurations of the algorithm on the fly to enable better cost-effectiveness tradeoffs than traditional approaches. To the best of our knowledge, this methodology has not been explored before. And SEADS is the first technique instantiating this methodology. By offering a scalable, continuous runtime dependence analysis, SEADS opens many doors for distributed software analysis by enabling a large number of dependence-based application/client techniques and tools that support quality assurance of distributed software. In sum, our main contributions in this article include:

- A distributed, online, and cost-effective dynamic dependence analysis framework for common distributed systems, which demonstrates a novel dynamic analysis methodology of overcoming the scalability and cost-effectiveness balancing challenges by reinforcement learning cost-effective analysis configurations on the fly according to current configurations and their costs with respect to the user-specified analysis time budget (Section 4).
- An open-source implementation of SEADS for Java that works with real-world continuously running distributed systems of different architectures, application domains, and scales (Section 5).
- An empirical demonstration and quantification of the scalability and cost-effectiveness advantages of the proposed technique over a conventional, state-of-the-art dynamic dependence analysis (i.e., without automatic configuration adjustment) for distributed programs as the baseline, in terms of detailed measures on precision losses and efficiency gains, against a diverse set of real-world distributed systems and their executions (Section 6).

The complete artifact package of SEADS has been made available here, including the source code and all the experimental scripts and datasets. This publicly accessible package not only

enables reproduction and replication of our work presented in this article, but also facilitates the development of further advanced approaches to assuring distributed software quality.

2 MOTIVATION

More and more industry-scale software systems are evolving to distributed systems that are continuously running. As they are part of the backbone of modern information technology infrastructure, assuring the quality of these systems is of paramount importance. One of the fundamental enablers for such tool support is dynamic dependence analysis, which, by modeling the interaction among code entities in a program, underlies a range of applications in testing, debugging, performance diagnosis and optimizations, security threat detection, and so on. For instance, tracking runtime dependencies backward from fault-revealing entities would guide the discovery of fault-inducing (i.e., faulty) entities in the program (i.e., the faults must propagate via dynamic dependencies of the fault-revealing entities on the faulty-inducing entities). For another example, tracking dynamic dependencies forward from a sensitive data accessing entity would guide the detection of information leaking (i.e., by checking if the sensitive data flows to any operations that send the data out of the program through the dynamic dependencies). These dependence-based applications essentially reason about the program's properties of interest (e.g., correctness and efficiency) via the dependence relationships among code entities at runtime. However, while there is an increasing demand for tool support for distributed software quality assurance, such tool support is largely lacking. Thus, in this context, our work focuses on common continuously running distributed systems.

To illustrate our motivation, Figure 1 shows an example of the source code excerpt from a real-world software project: Voldemort [4], a widely used distributed storage service (e.g., by LinkedIn to support a large part of the web site). As a distributed non-relational (NoSQL) data storage system, Voldemort enables high performance and availability via simple key-value data access. In the code snippet, a method `processEvents()` is in the class `ClientRequestSelectorManager`, and another method `main(String[] args)` is in the class `VoldemortServer`, as shown in Figure 1. These two classes are in different processes, *Store* and *Server*, which may be on separate machines (computing nodes). Thus, a dependence analysis approach for such a distributed system could be distributed to monitor these executed methods and to exploit computational resources in these nodes. These two methods communicate through message passing without a global clock, and they may have an implicit dependence relationship: The method `processEvents()` may be dependent on the method `main(String[] args)` in the execution. However, developers would hardly find the implicit dependencies between these methods only via reading the source code. Nonetheless, most existing dependence analyses rely on explicit dependencies, not capturing implicit ones. Therefore, we need a novel approach to infer these implicit dependencies.

As shown in Figure 1, the Voldemort system continuously runs to provide functional services. Tracing is unnecessary for an online dynamic analysis approach, which continuously runs and analyzes the system, along with the system execution. By contrast, offline dynamic techniques often analyze the program after the execution(s) have terminated, according to the collected traces whose storage and I/O costs might be expensive. If the execution is infinite (i.e., unbounded), the trace storage spaces would also be unbounded. Apparently, tracing the infinite execution is not practical. In addition, the terminating operation may also be impractical for a production system, because a common business flow (as realized by the system) should not be interrupted merely for dependence analyses. Thus, existing dependence analysis techniques, mostly offline, are generally unsuitable for continuously running distributed systems; and an online and continuous analysis is more desirable than an offline analysis for these systems.

```
// Voldemort Server Process
package voldemort.server;
public class VoldemortServer extends AbstractService {......
  public void createOnlineServices() {
    onlineServices.add(nioSocketService);
    onlineServices.add(socketService);
  ......}
  public static void main(String[] args) throws Exception {......
    final VoldemortServer server = new VoldemortServer(config);
    if(!server.isStarted())
        server.start();
  ......}
  public void startOnlineServices() {......
    for(VoldemortService service: onlineServices) {
        service.start();  }
  ......}
  protected void startInner() throws VoldemortException {......
    for(VoldemortService service: basicServices) {
        service.start();
    }
    startOnlineServices();
  ......}
......}
......
// Voldemort Store Process
package voldemort.store.socket.clientrequest;
public class ClientRequestExecutorFactory implements ResourceFactory<...> {......
  private class ClientRequestSelectorManager extends SelectorManager {......
    protected void processEvents() {......
      SocketChannel socketChannel = clientRequestExecutor.getSocketChannel();
      ......
      socketChannel.register(selector, SelectionKey.OP_CONNECT, clientRequestExecutor);
    ......}
  ......}
......}
```

Fig. 1. Code snippet from the Voldemort system as an example distributed system.

Voldemort has 20,406 methods in 115,310 non-blank non-comment Java source code lines. There may be serious cost-effectiveness and scalability problems for code analysis against distributed systems of such a scale. For instance, we spent 151 seconds in gaining dependence sets from a Voldemort integration test using a dynamic distributed-program dependence analysis technique DistIa [19] after terminating Voldemort processes and then gathering data from the execution traces. Though DistIa is very fast, its results (dependence sets) are very coarse. In contrast, the most precise extension of DistIa [15], a hybrid dependence analysis (a combination of static and dynamic analyses) solution utilizing both method-level and statement-level data to achieve a dynamic dependence abstraction, was twice as precise as DistIa for small- to medium-sized programs. Yet this analysis could not scale to Voldemort— even the first phase of the analysis (which has three phases) could not finish in more than 12 hours on an Ubuntu 16.04.3 LTS workstation with four 2.67 GHz processors, 512 GB DRAM, and 2 TB HDD (finally, we had to cancel the analysis). We concluded that this more precise analysis was too expensive to analyze an industry-scale distributed program due to the generally large size and great complexity of the target system. In general, the extension of DistIa suffers an impractical level of efficiency, hence a serious scalability problem, hence is subject to a very-low level of cost-effectiveness for large distributed systems.

From these examples, we see that, while representing the state-of-the-art in distributed program dependence analysis, both DistIa and its more precise extension are not suitable for common, continuously running distributed systems due to the scalability and cost-effectiveness problems. To resolve these problems, an approach should be able to quickly adjust itself. Moreover, in a varying environment such as a server performing lots of heavy tasks, merely one or a few adjustments may not be sufficient either. For example, after an analysis has adjusted itself to the optimal condition meeting the current user requirements, the runtime environment (e.g., operating system) or user requirements (e.g., time budgets) may change at the next second (for a different query), and then

there may come a deviation of the analysis from the previous, optimal condition. Thus, scalable and cost-effective dependence analysis approaches for distributed systems should be able to *continually* adjust themselves to meet varying requirements.

Following these observations, we develop SEADS as a distributed, online dynamic dependence analysis framework for common, continuously running distributed systems, which automatically and continually tunes itself to balance analysis cost and effectiveness. The technique addresses the scalability and cost-effectiveness balancing challenges faced by existing peer approaches through our novel approach using a reinforcement learning (Q-learning) strategy to learn cost-effective analysis configurations for the analysis algorithm to meet changing requirements.

3 BACKGROUND

In this section, we discuss some techniques and key concepts underlying SEADS.

3.1 Dependence Analysis for Single-process Programs

Analyzing dependencies among program entities of a software system can help developers better understand the structure and behaviors of the system. Thus, dependence analyses are very useful for users to develop, test, and maintain the system, because these tasks rely on the understanding of system structure and behaviors. Program dependencies can be deduced by both static and dynamic analyses. A static dependence analysis computes dependencies via analyzing the program code without executing the software. However, a dynamic dependence analysis infers dependencies from the data gathered during the execution(s) [56].

As a dynamic analysis approach, DIVER [16] computes dependence sets as impact sets by using dependence analysis techniques. As a recent advance in (offline) dynamic analysis, DIVER [16] achieves higher precision, hence provides a more cost-effective option over EAS-based approaches (which derive dynamic dependencies based on execution orders), such as PI/EAS [6]. DIVER utilizes a static dependence analysis to significantly decrease the size of the dependence set produced by PI/EAS. With significantly smaller resulting dependence sets, the cost-effectiveness tradeoff of DIVER is much higher even with the additional static dependence analysis cost. DIVER works in three technical phases: static analysis (Phase 1), runtime tracing (Phase 2), and post-processing analysis (Phase 3). DIVER first computes traditional control/data dependencies [41] and instruments the input program in Phase 1. In Phase 2, the instrumented version of the program is executed for tracing *entry* (i.e., program control entering a method) and *returned-into* (i.e., program control returning from a callee into a caller) events. In Phase 3, the technique computes the dependence set from the trace for any query given by the user.

An online dynamic analysis, DIVERONLINE [14] avoids execution tracing costs (e.g., space and I/O costs) that are ineluctable in offline analyses, via computing dependence sets during the executions of the program under analysis. In addition, DIVERONLINE provides an *All-in-One* analysis, which computes the dependence sets for all possible queries (methods), and then corresponding dependence sets are directly delivered to the user as the result within a short response time. As such, an *All-in-One* online dynamic dependence analysis may be a suitable solution for dependence analysis of large-scale software systems. In our SEADS framework, we leverage online dynamic analysis to compute dependencies within each component of the given SUA— we treat each component as a single-process program from the perspective of our analyses.

3.2 Dependence Analysis for Distributed Programs

For a complex distributed system with multiple processes, the developer needs to understand various (explicit and implicit) dependencies both within a single process and across multiple processes. Krinke proposed a slicing algorithm incorporating dependencies across distributed components

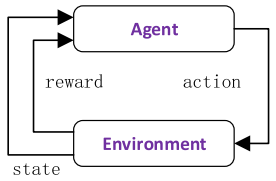

Fig. 2. The high-level workflow of Q-learning.

induced by socket-based message passing [51], but the dependencies were approximated over-conservatively, because they are computed through a purely static analysis. Another approach [10] infers various kinds of dependencies due to interprocess communications, but the approach potentially suffers a scalability problem due to its heavyweight nature, although it was not implemented and evaluated against real-world distributed systems.

To overcome the scalability challenges, a lightweight dynamic analysis for distributed programs, DistIa [19], was proposed. The analysis monitors and records method events and their timestamps during the system execution, and then approximates runtime dependencies among relevant methods, either within or across processes, based on the happens-before relations among execution events associated with those methods. For example, if a method A has the last returned-into event that was executed before the first entry event of another method B, the partial-order is A *before* B, and DistIa approximately supposes that B is dependent on A. Similarly, if one method D is dependent on another method C, C must execute before D; otherwise, C cannot affect D. Thus, dependencies computed by DistIa are safe for the executed methods. In our design of the Seads framework, we exploit the dependence inference based on happens-before relations as the basis for safe interprocess dependence approximation, as did DistIa.

3.3 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning to suggest a software agent taking actions in an environment to maximize the total reward in all possible successive actions [55]. The agent modifies its actions or control policies according to its interactions with the environment. RL requires less prior knowledge so it can be applied to environments where standard supervised learning or unsupervised learning approaches are not applicable [54]. Unlike supervised learning and unsupervised learning, RL does not need training data. As in a Markov decision process whose states are decided from previous states [64], an output of RL depends on the corresponding input, and the next input depends on the current output [1]. These characteristics make RL a generally suitable learning methodology for Seads.

In particular, as a particular type of RL, Q-learning uses the Bellman equation to minimize its cumulative cost [11]. When applied in software adaptations, it does not require explicit or exact descriptions of software systems and only needs state measurements in its feedback control loop [54]. Q-learning also has an exact capability of learning the next state according to the previous state only. Starting from an initial state, Q-learning tries to find a way to maximize cumulative reward values by selecting an action after measuring how good the action is in a particular state [79]. It is an off-policy and model-free algorithm, as it does not require an existing policy or a model [80].

The overall workflow of Q-learning is depicted in Figure 2, including the following steps: (1) initialize Q-learning components, such as the environment, the agent, Bellman equation parameters, and a Q-table (i.e., a lookup table calculating expected rewards); (2) at the current state, the agent selects an action referencing the maximal value in Q-table or by random; (3) the agent receives a state and a reward from the environment; (4) update the Q-table using the Bellman Equation [24]; and (5) repeat (2) through (4) until the learning meets predefined conditions (e.g., when the agent

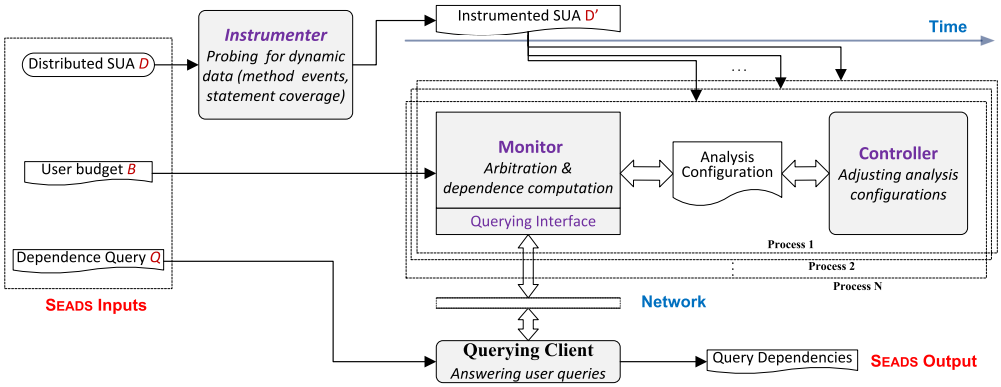

Fig. 3. An overview of the SEADS architecture and workflow, including its input, output, and key modules.

finishes the ultimate task assigned to it). In our SEADS framework where RL is applied for learning cost-effective analysis configurations at runtime, we particularly exploit Q-learning for automatically adjusting the configurations. Since SEADS provides a continuous dependence analysis, the predefined condition for terminating the learning process is when the SUA is terminated. That is, the learning process in SEADS repeats from (2) through (4) after finishing (1), until the SUA exits.

4 APPROACH

This section presents our technical approach with SEADS. We first give an overview of the architecture and workflow of SEADS, followed by elaborating each of its key modules.

4.1 Overview

The overall workflow of SEADS is shown in Figure 3. SEADS consists of several kinds of components, including an *instrumenter*, a set of *monitors* each for a process of the SUA, a set of *controllers* each for a process of the SUA, and a *querying_client*. It takes three **inputs** from the user: the distributed SUA D (in an executable format such as Java bytecode), a user budget B, and a dependence query Q. In particular, B is a response time constraint for the dependence analysis. In addition, since SEADS works at method level, Q is a method name given by the user for searching dependencies of the method, such as `voldemort.server.VoldemortServer: void main(java.lang.String[])`.

The *instrumenter* first inserts probes, which will monitor *entry* and *returned-into* events of all executed methods and/or monitor the coverage of all executed statements, into D to generate the instrumented SUA D'. Then, as *time* goes on (as indicated by the "Time" axis in the figure), D' continuously runs (in N distributed processes), and SEADS continually adjusts itself in its analysis configuration through a monitor and a controller running along with each process. In particular, during the execution of D', in each of its N processes, the *monitor* performs arbitration (deciding whether the adjustment is needed) and dependence computation, and the *controller* adjusts analysis configurations. As shown in Figure 3, the *monitors* and *controllers* of SEADS are distributed along with N processes: In each of the N processes of the SUA, SEADS has one monitor and one controller running in that process for controlling and performing the computation of dynamic intraprocess dependencies, respectively. In this way, all of the *monitors* and *controllers* together perform (per-process) intraprocess dependence analyses in a parallel and distributed manner naturally (with the same distributed architecture as that of the SUA itself). The *querying_client* computes interprocess dependencies on one machine (where the *querying_client* runs), which is the only dependence analysis step in SEADS that is not distributed.

The *querying_client* receives query Q from the user and sends it, through the *network* facility, to the *querying_interface* that directly communicates with the *monitor* in each process. After the dependence computation in a process has finished, the resulting dependencies are delivered back to the *querying_interface* attached to the monitor in that process, from which the *querying_client* receives the dependencies for the process. After the distributed dependence analyses in all processes are completed, the *querying_client* receives all intraprocess dependence sets and computes the overall dependencies (via an *interprocess analysis*) as the final **output** presented to the user.

Running example. To illustrate how SEADS works, we use the Voldemort system against an integration test as our running example. We consider querying the dependence set of a query `voldemort.server.VoldemortServer: void main(java.lang.String[])` (i.e., a method `main(String[] args)` in the class `VoldemortServer` of the *Server* process). For brevity, only part of the source code is shown in Figure 1.

4.2 Configuration

Our core idea for achieving better scalability and cost-effectiveness is to continually adjust analysis configurations according to (i) the user budget, (ii) the current and previous configurations, and (iii) time costs of dependence computations. The key insight underlying this design is that each analysis configuration represents a distinct tradeoff between the cost and effectiveness of our dependence analysis. Thus, in our framework analysis configurations play a critical role. Our hybrid approach leverages varying combinations of static and dynamic analysis techniques along with varied static/dynamic data (e.g., the static dependence graph, method events, and the statement coverage) while using different static and dynamic configuration parameters.

As follows, we first present the parameters (i.e., configuration items) considered in our static and dynamic analysis separately (referred to as *static configurations* and *dynamic configurations*, respectively), and then we describe our holistic (hybrid) analysis configuration encodings. The rationale (justification) of our selection of these particular analysis parameters is that they have not only different but also competing influences on the cost and effectiveness of the dependence analysis. First, their influences should be *different* from each other, because adjusting between two configurations that lead to the same cost-effectiveness tradeoff would be wasteful. Second, their influences on the cost and those on the effectiveness should be competing, because an analysis parameter that benefits or penalizes cost and effectiveness at the same time should be always set or dismissed, respectively, hence would be out of the scope of adjustments. Since the chosen parameters are known to be different and competing influence factors in static/dynamic dependence analysis in general, their selection is justified by our goal with SEADS of balancing the cost and effectiveness at runtime for better cost-effectiveness.

4.2.1 Static Configuration. There are two dimensions of static configuration parameters considered in SEADS: *data_selection* and *sensitivity*. In the *data_selection* dimension, we currently consider only one parameter, *staticGraph*, which concerns whether static data are used.

- The parameter *staticGraph* determines whether SEADS uses static dependencies (within each component of the SUA, represented as a dependence graph) to compute the dynamic dependencies of the given query. If the parameter is enabled, SEADS traverses the per-component static dependence graphs to infer more precise (runtime) dependencies with a higher time cost. Otherwise, such static dependence graphs would not be used, and then SEADS offers rough but rapid results (dependence sets) according to dynamic data only.

The *sensitivity* dimension, including two parameters (*context sensitivity* and *flow sensitivity*), is expected to bring a higher level of precision of the static dependence computation when the respective sensitivity is set (enabled) than when it is dismissed (disabled), as explained below.

- Context sensitivity concerns the awareness of the effects of varied calling contexts on analysis facts in a static analysis. A context-sensitive analysis distinguishes different calling contexts of methods and computes separate information for different calls of the same method. Conversely, a context-insensitive analysis treats all callsites of a method as one callsite [38, 70]. For example, if a method is called twice each at a different callsite, a context-sensitive analysis would distinguish these callsites when computing analysis facts (e.g., dependencies). As a result, if the analysis fact is valid only with respect to one callsite, the context-sensitive approach would be able to recognize the false result associated with the other callsite. A context-insensitive analysis, however, would not be able to do so, thus it may produce the false result. Meanwhile, differentiating calling contexts comes with an additional cost compared to not doing so. Therefore, a context-sensitive analysis generally computes more precise results with higher costs than a context-insensitive analysis.
- Flow sensitivity concerns the observance of control flow reachability in a static analysis. A flow-sensitive analysis approach takes into account the execution order (i.e., the control flow) of code entities (e.g., statements), whereas a flow-insensitive analysis does not consider the order [70], when computing analysis facts (e.g., dependencies between two code entities). For instance, if there are two definitions of a variable in the program, with one use of the same variable in between, a flow-sensitive analysis distinguishes the execution order of these definitions and the use hence reports one dependence induced by the first definition and the use. A flow-insensitive analysis, however, would additionally (and falsely) report the dependence due to the second definition and the use. Meanwhile, accounting for the execution order requires control flow reachability analysis, which incurs an additional cost compared to not doing so. Therefore, a flow-sensitive analysis is often more precise but also more expensive than a flow-insensitive analysis.

4.2.2 Dynamic Configuration. There are two dimensions of dynamic configuration parameters considered in SEADS: *data_selection* and *data_granularity*. The *data_selection* dimension concerns which types of dynamic data SEADS uses for its analysis. Specifically, there are two configuration parameters in this dimension: *methodEvent* and *statementCoverage*. They determine if the corresponding dynamic data are used, as elaborated below.

- The parameter *methodEvent* decides if SEADS uses method (*entry* and *returned-into*) events to compute dependencies. If the parameter is enabled, SEADS infers more precise dependencies from dynamic data (e.g., method events) with additional costs. Instead, if the parameter is disabled, SEADS coarsely but quickly computes dependence sets without method events.
- The parameter *statementCoverage* determines whether SEADS prunes the static dependence graphs using statement coverage information before applying the static dependencies in the hybrid computation of runtime dependencies. The pruning means that SEADS only considers statements covered in the SUA execution analyzed, with other statements dismissed while referring to the static dependencies. When the parameter *statementCoverage* is enabled, the dynamic dependence analysis is more expensive but more precise than otherwise.

In the *data_granularity* dimension, only one parameter, *MethodInstanceLevel*, is considered, which concerns the granularity of the dynamic data (method events) used.

Table 1. Holistic (Hybrid) Dependence Analysis Configuration Encoding

Encoding	Static Configuration			Dynamic Configuration		
	Data Selection	Sensitivity		Data Selection		Data Granularity
	StaticGraph	Context Sensitivity	Flow Sensitivity	Method Event	Statement Coverage	Method InstanceLevel
000000	No (0)	No (0)	No (0)	No (0)	No (0)	No (0)
000001	No (0)	No (0)	No (0)	No (0)	No (0)	Yes (1)
.....						
111110	Yes (1)	Yes (1)	Yes (1)	Yes (1)	Yes (1)	No (0)
111111	Yes (1)	Yes (1)	Yes (1)	Yes (1)	Yes (1)	Yes (1)

- The parameter *MethodInstanceLevel* is about whether Seads uses all method event instances to compute dependencies. If the parameter is enabled, Seads utilizes all instances of (*entry* and *returned-into*) events to compute dependencies more precisely at the cost of greater overheads (for monitoring and utilizing a greater amount of dynamic data). Otherwise, only the first *entry* and last *returned-into* events of each executed method are collected and used, thus the computation is faster but gives relatively rougher results (i.e., lower precision).

4.2.3 Holistic Analysis Configuration. The holistic configuration of Seads consists of both the static and dynamic configurations described above. We use three bits to encode the three parameters in the static configuration, thus there are eight possible combinations (static configurations). The first binary number 1 or 0 means whether Seads uses the static dependencies. The second and third binary numbers of 1 or 0 mean sensitivity (i.e., the sensitivity is enabled, as denoted by *Yes*) and insensitivity (i.e., the sensitivity is disabled, as denoted by *No*), respectively. The second bit represents the analysis being context-sensitive or context-insensitive, and the third bit indicates the analysis being flow-sensitive or flow-insensitive. Thus, the static configuration is encoded for eight possible values, from 000 through 111. In a similar manner, we utilize three bits to encode the three parameters in the dynamic configuration. The first through third bits represent parameters *MethodEvent*, *statementCoverage*, and *methodInstanceLevel*, respectively. This way, the dynamic configuration has 8 possible values, ranging from 000 to 111.

Therefore, the holistic (i.e., hybrid) configuration, including static and dynamic configuration parameters, is encoded as a 6-bit binary number that ranges from 000000 through 111111. The first to third bits are encoded as the static configuration parameters, and the fourth to sixth bits are used as the dynamic configuration parameters. As shown in Table 1, the hybrid configuration parameters are encoded in order from the left to the right as: *staticGraph*, *context-sensitivity*, *flow-sensitivity*, *methodEvent*, *statementCoverage*, and *methodInstanceLevel*.

Among the possible 64 (2^6) hybrid configurations, some are invalid and thus are never used in Seads, because certain configuration parameters are dependent on others and they are meaningful only with other parameters enabled. For example, three parameters (*context-sensitivity*, *flow-sensitivity*, *statementCoverage*) depend on the parameter *staticGraph*. If the parameter *staticGraph* is disabled, meaning that Seads does not use the static data (i.e., the static dependence graph), then the three relevant parameters (*context-sensitivity*, *flow-sensitivity*, and *statementCoverage*) are meaningless—statement coverage data are only used for pruning the static dependence graph in Seads. Thus, configurations 001xxx, 010xxx, 011xxx, and 0xxx1x are invalid, where x is a bit that can be 0 or 1. Another example is that the parameter *methodInstanceLevel* depends on the parameter *methodEvent*, thus configurations xxx0x1 are invalid. In addition, configuration 000000

means that no data is utilized in the analysis, which is also invalid. In total, there are 38 invalid configurations and 26 valid configurations in SEADS.

Illustration. As shown in Table 1, the configuration 111110 indicates that all the six parameters but *methodInstanceLevel* are enabled. Under this configuration, SEADS would perform a hybrid analysis of dynamic dependencies, utilizing the static dependencies first computed within each component of the SUA through a context- and flow-sensitive analysis and then pruned with statement coverage, as well as method execution events collected at method level (i.e., only two events are kept per executed method). In terms of its internal workings, for the running example, after the two Voldemort processes started, the SEADS monitor in each process checks whether the configuration file *Configuration.txt* exists or not. If it is found, the monitor reads the configuration from this file. Otherwise, SEADS uses the initial configuration 111111 (i.e., all the six parameters are enabled) to gain the highest possible precision of the dependence analysis.

4.3 Instrumenter

The *instrumenter* of SEADS inserts probes to the SUA D to produce its instrumented version D' that will continuously run. During the execution, the probes monitor and record the *entry* and *returned-into* events per executed method. We probe for these events, because they suffice for inferring the happens-before relations among executed methods as shown before [16], while the happens-before relations enable the approximation of dynamic dependencies among methods both within and across processes. The instrumenter also probes for statement coverage, another kind of dynamic data considered. For greater monitoring efficiency, only branches are probed, which suffices for inferring statement coverage from branch coverage as we did before [18]. In particular, besides the branches associated with explicit predicates, we also treat the entry of each method as a special branch (i.e., *entry branch*), whose `true` edge leads to the entry (execution) of the method.

Illustration. For the running example, SEADS traverses the bytecode of Voldemort to create its instrumented version with probes monitoring *entry* and *returned-into* events of each method as well as statement coverage during the execution. For example, for the source code shown in Figure 1, SEADS adds a probe before the method `server.start ()` (line 11) to monitor its *entry* events. Moreover, another probe is inserted after the method `server.start ()` to monitor *returned-into* events of this method. In addition, as an example of statement coverage probing, SEADS inserts a probe at the entry (branch) of the `main` method (before Line 9) and another probe for the branch at Line 10 to tell whether the `true` edges are covered. If so, statements control-dependent on the edges will be inferred as being covered as well.

4.4 Monitor

After the instrumentation, the instrumented SUA D' continuously runs in its N distributed processes. Through the instrumentation, the monitor and controller modules for each process of D' are launched upon the start of that process, and then also continuously run along with the process, as shown in Figure 4. During the execution of the process, as the core component of SEADS, the monitor module determines *when* the analysis configuration needs to be adjusted (i.e., *arbitration*) and computing dynamic dependencies with a current configuration (i.e., *dependence computation*). The module consists of two submodules: a processor and a gatherer. The gatherer focuses on collecting dynamic data (method execution events and/or statement coverage), which feed the processor for the dependence computation therein. More specifically, the processor computes (updates) the dynamic dependencies for all possible queries (i.e., methods exercised at least once so far) when (a) the time, since the previous round of dependence computation (updating) exceeds a threshold (e.g., 5 minutes) and (b) the number of method-execution events accumulated since the previous

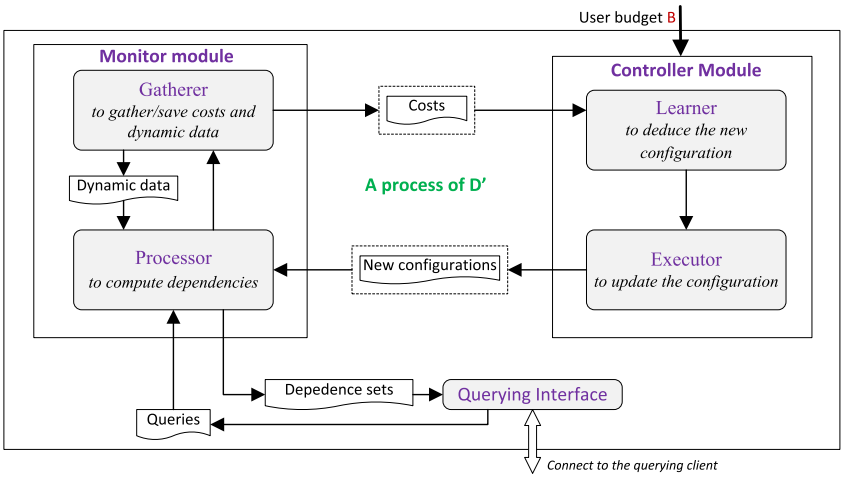

Fig. 4. The monitor and controller modules running along with the instrumented SUA.

round of dependence computation exceeds another threshold (e.g., 1,000). Both thresholds are part of SEADS's settings, customizable by users.

The dependencies need to be computed for *all* possible queries for two reasons. First, SEADS aims to answer arbitrary queries at arbitrary times, thus it cannot assume which queries users would send and when. Second, SEADS performs online analyses, thus it does not keep all execution data, while the query dependencies it computes must respect all the dynamic data available up to the query arrival time. The online nature of the analysis also justifies the computation being continually redone (updated)—since the dynamic data used by the analysis come as streaming data.

We set both conditions, (a) and (b) above via the two thresholds, for triggering the dependence recomputation/updating, because the speed at which the dynamic data arrive can vary widely across different SUAs. If we consider (a) only, when the time reaches the threshold, there may be still too few new dynamic data available to deserve the updating (i.e., *trivial recomputation*); if we consider (b) only, the dynamic data may arrive too fast such that SEADS thrashes between two rounds of updating (i.e., *overly busy recomputation*).

Next, we elaborate on the two roles of the monitor running with each SUA process: arbitration and dependence computation. We then describe how SEADS interacts with users, responding to user queries and delivering query dependencies back to users.

4.4.1 Arbitration. Algorithm 1 shows the arbitration pseudo-code that decides when and how to trigger dependence computations and configuration adjustments. In this algorithm, several variables are used to denote the inputs: *method* as the executed method ID (an integer uniquely representing a method), *gCounter* as the counter of method events, and *LastT* as the time when the previous analysis round completed. *TC* and *TT* denote the aforementioned time and event number threshold, respectively. Moreover, *QU* is a method event queue that records the return-into event(s) of methods as the method ID(s) and the entry event(s) as the minus method ID(s)—we simply use negative values here to indicate entry events as opposed to returned-into events. SEADS first initiates *gCounter*, *QU*, and *LastT*, and sets the current configuration *TCN* as 111111 for the most precise (but potentially the slowest) analysis to start with (line 1). The variable *oldTCN* is used to keep the previous configuration, initialized as None. The values of *sgc_T*, *sgl_T*, and *d_T* are timeouts for constructing/loading the static dependence graph and computing dependencies, respectively: These values are empirically allocated from the total user-given budget *B* (line 2). To illustrate how

ALGORITHM 1: Triggering the Dependence Computation and Configuration Adjustment

let $method$ be the executed method
let $gCounter$ be the number counter of method events
let TC and TT be the thresholds of event number and analysis time interval, respectively
let $LastT$ be the time of the last computation
let B be the user budget
let sgc_T, sgl_T, and d_T be timeouts of constructing/loading the static graph and computing dependencies, respectively
let QU be the method event queue
let TCN and $oldTCN$ be current and immediately previous configurations, respectively
let $isTimeOut$ be the boolean value to record timeouts

1: Set $gCounter$ = 0, $LastT$ = 0, QU = $\emptyset$, TCN =111111, $oldTCN$ =None
2: Assign sgc_T, sgl_T, and d_T from B
3: **while** true **do**
4: **if** event($method$)==entry **then**
5: $gCounter$++
6: Add (-$method$) to QU
7: **if** event($method$)==returnInto **then**
8: $gCounter$++
9: Add $method$ to QU
10: **if** $gCounter > TC$ *and* ($CurrentTime$ - $LastT$) > TT **then** // $CurrentTime$ is the current system time
11: Read current configuration parameters
12: $isTimeOut$ = false
13: **if** $Configuration\_staticGraph$ **then** // if the *staticGraph* parameter is enabled
14: **if** Static configuration parameters are different between TCN and $oldTCN$ **then**
15: Construct a new static (dependence) graph
16: **if** (Not $isTimeOut$) *and* ($constructTime > sgc\_T$) **then**
17: Cancel the static graph construction, and set $isTimeOut$ = true
18: **if** The static graph $exists$ **then**
19: Load the static graph
20: **if** (Not $isTimeOut$) *and* ($loadTime > sgl\_T$) **then**
21: Cancel the static graph loading, and set $isTimeOut$ = true
22: **if** not $isTimeOut$ **then**
23: Call the processor to compute dependencies with TCN
24: **if** (Not $isTimeOut$) *and* ($ComputeTime > d\_T$) **then**
25: Cancel the dependence computation, and set $isTimeOut$ = true
26: $gCounter$ = 0, $LastT$ = $CurrentTime$
27: Call the gatherer to record the time cost of the analysis
28: Call the controller to obtain new configuration $newTCN$
29: $oldTCN$ = TCN
30: TCN = $newTCN$

the monitor works, let us consider a concrete example: The user budget is 60 seconds, out of which we allocate *sgc_T*, *sgl_T*, and *d_T* as 42 s, 12 s, and 6 s, respectively; let TC=1,000, TT=5 (minutes), and the method being processed be `voldemort.server.VoldemortServer: void startInner()` (id=15700); and let `voldemort.server.VoldemortServer: void main(java.lang.String[])` be the query.

The algorithm proceeds with an infinite loop arbitrating dependence computations and configuration adjustments (lines 3–30), via invoking (collaborating with) the controller module for the same process of this monitor. During the execution, in each *entry* event of the *method*, SEADS increments *gCounter* by one and adds minus *method* (e.g., −15,700) to QU (lines 4–6). In each *returned-into* event, SEADS also increments *gCounter* by one (e.g., 2 now) but adds *method* (e.g.,

15,700) to *QU* (lines 7–9). If *gCounter* is greater than *TC* and the time span (between the current time and the last computation time *LastT*) is greater than *TT* (i.e., both conditions (a) and (b) are satisfied), SEADS will start a new round of analysis (e.g., updating dynamic dependencies for all possible queries) as detailed below. To start with, SEADS reads the current configuration and set *isTimeOut* as false (lines 10–12). For example, after 1,000 events occurred and 5 minutes passed, *gCounter*>*TC* (1,000) and (the current system time - *LastT*)>*TT* (5 minutes). Then, SEADS reads the current configuration TCN=111111 with *isTimeOut*=`false`. If the *staticGraph* parameter is enabled and at least one of the static analysis parameters varies between the current and immediately previous configurations, SEADS constructs a new static dependence graph (lines 13–15) using the static configuration. For example, with TCN=111111 now, the first to third bits (three static parameters) are all 1 (enabled). Thus, SEADS constructs the new static dependence graph with both context sensitivity and flow sensitivity applied.

The static analysis, including constructing and loading the static dependence graph, reuses DIVER [16] and DIVERONLINE [14], as described earlier in Section 3. Recall that each monitor only deals with the single process associated with it, thus the static dependence analysis here only targets the code of the SUA component that runs in the process. When the static dependence graph is ready, SEADS loads the static graph (lines 18–19). Moreover, the processor is invoked to compute dependencies with the current configuration *TCN* (line 23), as detailed in Algorithm 2 (Section 4.4.2). When *isTimeOut* is false, if any part of the static or dynamic analysis—(i) constructing and (ii) loading the static dependence graph and (iii) computing dynamic dependencies (costing *constructTime*, *loadTime*, and *computeTime*, respectively)—runs timeout, SEADS would cancel the respective part of the analysis and set *isTimeOut* as true (lines 16–17, 20–21, 22–25). For example, with the current configuration TCN=111111, SEADS has not finished constructing the static graph in *sgc_T* time (42 s). Thus, the graph construction is canceled. Then, *isTimeOut*=`true`, hence SEADS skips the static graph loading and dependence computation. After resetting *gCounter* and *LastT*, SEADS calls the gatherer to collect the time costs of above analyses (i.e., constructing/loading the static graph, computing dependencies) under the current configuration and then calls the controller to obtain the next configuration *newTCN* (lines 26–28), as detailed later in Algorithm 3 (Section 4.5). For example, we now have *gCounter*=0, the time cost=43 seconds and *newTCN*=000101. Finally, the algorithm updates the current (*TCN*) and previous configuration (*oldTCN*) accordingly for the next arbitration iteration (lines 29–30). For example, now *oldTCN*=111111 and *TCN*=000101.

Illustration. For the example considered, the analysis starts with *TCN*=111111, for SEADS to compute the most precise dependence sets possible with respect to its current design. When there are more than 1,000 new method events and it is over 5 minutes since the last analysis round, with the *staticGraph* parameter enabled, SEADS attempts to construct a static graph with context-sensitivity and flow-sensitivity enabled. In *sgc_T* time (42 s), however, the graph construction did not finish and thus it was canceled. Then, *isTimeOut* is set to be true and no static graph is created; hence, SEADS skips the static graph loading, and further skips dependence computation also. Suppose immediately afterwards the controller produces the next (new) configuration *newTCN* 000101, which indicates two parameters (*methodEvent* and *methodInstanceLevel*) are enabled while the other four disabled (i.e., no static dependencies nor statement coverage are used). With this new configuration, SEADS is able to finish the entire round of dependence analysis within the total budget time (60 s). Since only the method events are used for the analysis, the dependencies are inferred immediately based on happens-before relationships according to the partially ordered sequence of execution methods. In this example, the sequence in the `Server` and `Store` process of the Voldemort system (in Figure 1) is shown in Figure 5 and Figure 6, respectively.

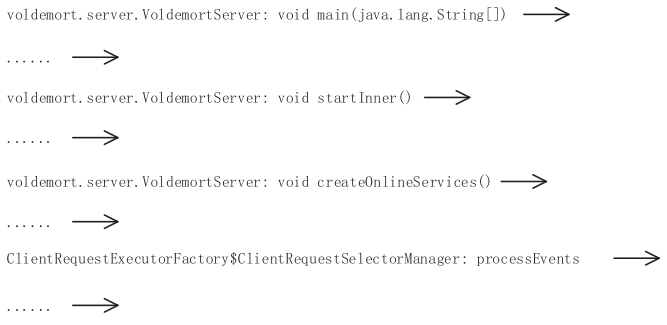

Fig. 5. An example of partially ordered sequence of executed methods in the Server process of Voldemort.

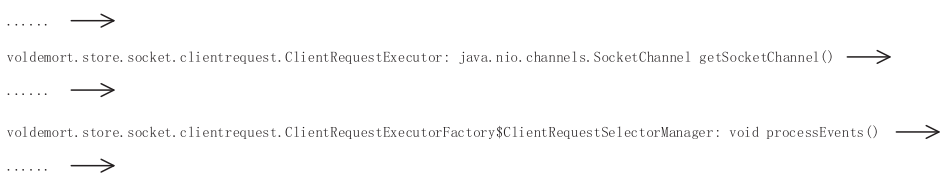

Fig. 6. An example of partially ordered sequence of executed methods in the Store process of Voldemort.

4.4.2 Dependence Computation. When SEADS calls the processor to compute dependencies, the online analysis based on DIVERONLINE [14] is adopted, avoiding execution tracing to economize analysis costs, such as storage and disk I/O costs. Algorithm 2 gives the pseudo-code of the online algorithm to compute dependencies. In Algorithm 2, QU is the same method event queue as in Algorithm 1, and $DS(m)$ is the dependence set for the method m. First, four configuration parameter variables (i.e., $Configuration\_staticGraph$, $Configuration\_methodEvent$, $Configuration\_statementCoverage$, and $Configuration\_methodInstanceLevel$) are read from the current configuration (line 1). For example, from the current configuration 000101, we have these variables assigned 0, 1, 0, 1, respectively. If the parameter *methodInstanceLevel* is disabled, SEADS filters the first *entry* and the last *return-into* events from the event sequence in QU (lines 2–3). For example, since *methodInstanceLevel* is enabled, SEADS skips the filtering. If *staticGraph* and *statementCoverage* are both enabled, the static dependence graph is pruned according to the statement coverage (lines 4–5). For example, since both *staticGraph* and *statementCoverage* are disabled, SEADS skips the pruning. Then, SEADS traverses QU to compute dynamic dependencies corresponding to the method events in QU (lines 6–24), as elaborated as follows:

For each event e in QU, let m be the corresponding method of e and $DS(m)$ be the dependence set of m that is empty initially (line 7). For example, for $e = -15700$, $DS(e)$ is empty now, where 15,700 is the ID of method `voldemort.server.VoldemortServer: void main(java.lang.String[])`.) If the parameter *methodEvent* is enabled and the value of e is negative (i.e., e is an *entry* event) and m is executed, SEADS adds m itself into the dependence set $DS(m)$ (lines 8–10). For example, since the parameter *methodEvent* is enabled and $e = -15{,}700$ (< 0), the method (of ID = 15,700) is added to $DS(m)$.

If parameters *methodEvent* and *staticGraph* are enabled, SEADS adds dependencies via calling a subroutine `AddDSForEntry` for the negative e value (*entry* event) or calling another subroutine `AddDSForReturnInfo` for the positive e value (*returned-into* event) (lines 11–15). For example, as the parameter *staticGraph* is not enabled, SEADS skips both subroutines. We leveraged DIVERONLINE [14] to develop these two subroutines, in which SEADS traverses the static dependence graph

ALGORITHM 2: Computing dependencies

let $Configuration\_staticGraph$ be $staticGraph$ parameter of current configuration
let $Configuration\_methodEvent$ be $methodEvent$ parameter of current configuration
let $Configuration\_statementCoverage$ be $statementCoverage$ parameter of current configuration
let $Configuration\_methodInstanceLevel$ be $methodInstanceLevel$ parameter of current configuration
let $DS(m)$ be dependence set for method m

```
1: Read current configuration parameter settings
2: if Not Configuration_methodInstanceLevel then
3:     QU=getFirstLastInstances(QU)
4: if Configuration_staticGraph and Configuration_statementCoverage then
5:     Prune the static graph with the statement coverage
6: for each method event e ∈ QU do
7:     m=abs(e), DS(m) = ∅
8:     if Configuration_methodEvent then
9:         if e < 0 then
10:            DS(m) ∪ = {m}
11:        if Configuration_staticGraph then
12:            if e < 0 then
13:                AddDSForEntry(m)
14:            else
15:                AddDSForReturnInto(m)
16:        else
17:            if e < 0 then
18:                for each last returned-into event e′ that happens after e ∈ Q do
19:                    m′=abs(e′)
20:                    DS(m) ∪ = {m′}
21:    else
22:        if Configuration_staticGraph then
23:            AddDSForEntry(m)
24:            AddDSForReturnInto(m)
```

to add dependencies into $DS(m)$, using different dependence propagation rules for the *entry* and *returned-into* event of the method (m), respectively.

If the parameter *methodEvent* is enabled and the parameter *staticGraph* is disabled, upon each negative e (i.e., an *entry* event), SEADS adds all methods whose last (returned-into) event in QU happened after e, into the dependence set $DS(m)$ (lines 16–20). For example, since parameter *methodEvent* is enabled and the parameter *staticGraph* is disabled, SEADS adds all methods whose last (returned-into) event in QU happened after e (−15,700) into the dependence set $DS(m)$.

If the parameter *methodEvent* is disabled and the parameter *staticGraph* is enabled, SEADS simply calls these two subroutines `AddDSForEntry` and `AddDSForReturnInfo` to add dependencies into $DS(m)$ (lines 21–24): In this situation, these two subroutines compute the dependencies by traversing the static dependence graph without utilizing any dynamic data. With the example configuration being considered (*methodEvent* is enabled and *staticGraph* is disabled), SEADS skips both subroutines here.

Note that for each process the monitor only computes the runtime dependencies *within* the process (referred to as *intraprocess dependencies*). Dependencies *across* processes (referred to as *interprocess dependencies*) will be computed for a given query Q by the *querying_client* of SEADS after it receives the intraprocess dependencies of Q from each of the SUA's processes.

Illustration. Consider the same example used for illustrating the arbitration algorithm above. With the new configuration 000101, the partially ordered sequences of methods are those in

```
{ voldemort.server.VoldemortServer: void main(java.lang.String[]),
voldemort.server.VoldemortServer: void startInner(),
voldemort.server.VoldemortServer: void createOnlineServices(),
......,
voldemort.store.socket.clientrequest.ClientRequestExecutorFactory$ClientRequestSelectorManager: void processEvents(),
...... }
```

Fig. 7. An example set of intraprocess dependencies of a query in the `Server` process.

```
{ voldemort.store.socket.clientrequest.ClientRequestExecutor: java.nio.channels.SocketChannel getSocketChannel()
......,
voldemort.store.socket.clientrequest.ClientRequestExecutorFactory$ClientRequestSelectorManager: void processEvents()
...... }
```

Fig. 8. An example set of intraprocess dependencies of a query in the `Store` process.

Figures 5 and 6. The pruning of the first *entry* and last *return-into* events is not executed, because the parameter *methodInstanceLevel* is enabled in this configuration. With this configuration, static dependence analysis and graph pruning are skipped, too. Next, in the loop of traversing the queue QU, m and $DS(m)$ are initiated first. With the parameter *methodEvent* enabled, for each e that is an *entry* event (the e value is negative), SEADS adds all methods whose last returned-into events in the queue QU happened after e, into the dependence set of the method m (i.e., Lines 17–20). As a result, SEADS computes the runtime dependencies for all possible queries (i.e., methods executed in any process). For example, for the query method `voldemort.server.VoldemortServer: void main(java.lang.String[])` exercised in the `Server` process, the dependence set at a particular querying time is partially shown in Figure 7, while a resulting dependence set of the query method `voldemort.store.socket.clientrequest.ClientRequestExecutor: java.nio.channels.SocketChannel getSocketChannel()` executed in the `Store` process is partially shown in Figure 8. Note that each of these dependence sets includes intraprocess dependencies only, despite the existence of methods executed in more than one process (e.g., the method `voldemort.store.socket.clientrequest.ClientRequestExecutorFactory $ClientRequestSelectorManager: void processEvents()` in this illustrative example).

4.4.3 Querying Interface. To offer the querying service to users, the monitor module for each process includes a *querying_interface* to receive the dependence query Q from and send corresponding dependence sets back to, the *querying_client* module of SEADS, both through the *network* facility (see Figure 3). When a query is received at (the *querying_interface* of) the *monitor* module, there are two situations, dealt with by the interface differently as follows:

(1) The *monitor* is in the middle of computing/updating the dependence sets for all possible queries. In this situation, the *querying_interface* will wait until the dependence computation/updating is completed to return the dependence set of Q.
(2) The *monitor* is performing arbitration functionalities, but not computing/updating dependencies—that is, it has completed a previous round of dependence computation/updating and is waiting for the next round. In this situation, the *querying_interface* will immediately return the most recently computed dependence set of Q.

As mentioned earlier, while the *querying_interface* (attached to the monitor) for each process computes intraprocess runtime dependencies, the *querying_client* module of SEADS derives

```
{ voldemort.server.VoldemortServer: void main(java.lang.String[]),
......,
voldemort.server.VoldemortServer: void startInner(),
voldemort.server.VoldemortServer: void createOnlineServices(),
voldemort.server.niosocket.NioSocketService: void <init>(...),
......,
voldemort.store.socket.clientrequest.ClientRequestExecutorFactory: void <init>(...),
voldemort.store.socket.clientrequest.ClientRequestExecutorFactory$ClientRequestSelectorManager: void processEvents(),
...... }
```

Fig. 9. The dependence set for an example query returned to the user.

interprocess dependencies, hence produces the final dependence set, while merging all the per-process intraprocess dependence sets, for the user-supplied query Q. Once it has received Q, the *querying_client* sends it to the *querying_interface* for each process, and then waits for all the per-process interfaces to return their respective intraprocess dependence sets for all possible queries. The reason is that all these dependence sets may be needed for deriving interprocess dependencies for Q. More specifically, the *querying_client* will identify the process P_i where Q was executed first (i.e., where the earliest first entry event of Q occurred). If no process exercised Q, an empty dependence set would be returned immediately back to the user. Otherwise, the final dependence set of Q, noted as $fDS(Q)$, is initialized as the intraprocess dependence set of Q returned from the *querying_interface* for P_i (noted as $intraDS(Q, i)$).

Then, for each other process P_j:

(1) if P_j also exercised Q, $intraDS(Q, j)$ is straightforwardly merged into $fDS(Q)$; otherwise,
(2) for each method m exercised in P_j, $intraDS(m, j)$ is straightforwardly merged into $fDS(Q)$ if the last returned-into event happens after the first entry event of Q.

This merging process implicitly derives and adds to $fDS(Q)$ the interprocess dependencies for Q according to the happens-before relationships among method execution events across all processes.

Illustration. Suppose when the two monitors, one for the Voldemort `Server` process and the other for the `Store` process, have received the same dependence query `voldemort.server.VoldemortServer: void main(java.lang.String[])` from their respective *querying_interface*, both are just performing routine arbitration (not in the middle of computing/updating dependencies). Thus, both monitors have all of their dependence sets (computed in the previous round of analysis) ready. Further suppose the `Server` process executed this query while the other process did not. The intraprocess dependencies of the query `voldemort.server.VoldemortServer: void main(java.lang.String[])` in the `Server` process include `voldemort.server.VoldemortConfig: int getBdbCleanerLookAheadCacheSize()`, `voldemort.server.VoldemortServer: voldemort.server.StoreRepository getStoreRepository()`, and a few other methods. In the `Store` process, only one method, `voldemort.store.stats.Tracked: java.lang.String toString()`, has its last returned-into event happened after the first entry event of the query here, and includes in its dependence set the method `voldemort.store.socket.clientrequest.ClientRequestExecutorFactory$ClientRequestSelectorManager: void processEvents()` and a few others, besides itself. Then, according to the process of deriving the final (cross-process) dependencies as described above, the *querying_client* will return to the user the final dependence set as shown in Figure 9. Here the two methods in the `Store` process are added to the dependence set due to their implicit dependencies on the query via happens-before relations.

ALGORITHM 3: Configuration Adjustment using Q-learning

let TCN and $newTCN$ be current and new configurations
let $Qtable$, $actions$, $rewards$, $states$, γ, α, and ϵ be components and parameters used by Q-learning
let T be overall dynamic dependence analysis time cost with the current configuration
let B be the user budget
let $probability$ be the possibility to select the action according to the largest value in $Qtable$

1: Initiate Q-learning components: $learner$, $Agent$, $Qtable$, $actions$, $rewards$, and $states$
2: Set Bellman equation parameters γ, α, and ϵ between 0 and 1
3: Update $reward$ = 1/(B - T) * 1,000.
4: Update $Qtable$ using the Bellman equation
5: $probability$=random(0,1)
6: **if** $probability$ < = (1 - ϵ) **then**
7: Take the best action according to the largest value in $Qtable$
8: **else**
9: Randomly take an action
10: Compute a new configuration $newTCN$ according the action and TCN

4.5 Controller

For each process of the SUA, SEADS runs a controller to adjust analysis configurations in SEADS. As shown in Figure 4, the controller takes the costs of the current configuration and user budget B as inputs to determine which next configuration the analysis should use to achieve a better cost-effectiveness (than with the current configuration) while respecting the user budget (i.e., containing the total analysis cost under the budget). Recall that the analysis in SEADS is distributed, thus the controller associated with each process of the SUA is only responsible for adjusting the configuration for the analysis of that process—the controllers across all processes work independently of one another.

Each controller module consists of two submodules: a learner and an executor. The learner utilizes the data from the gatherer (i.e., the analysis costs under the current configuration) while referring to the user budget, to adjust the configuration—the output is a configuration that may be the same as or different from the current configuration. Then, the executor updates SEADS to the new configuration: It takes the learner's output and simply prepares for transferring the new configuration to the collaborating monitor. Specifically, the preparation is realized by serializing the configuration to an external file that is later loaded by the processor in the monitor. Next, we elaborate the learner's inner workings for configuration adjustment.

SEADS makes decisions on new analysis configurations using a reinforcement learning methodology, in particular the Q-learning method (Section 3). Supervised learning, which needs a large training set, is not appropriate for the configuration adjustment in SEADS, because there are not enough data for training when SEADS starts with a particular SUA. Meanwhile, since the dynamics of execution may vary widely across different SUAs, learning from other SUAs beforehand may not be effective either. Thus, given the unpredictably changing *environment* during the execution of an SUA, reinforcement learning, which is not subject to those constraints, is more suitable. Moreover, Q-learning as a special type of reinforcement learning is particularly appropriate for configuration adjustments in our framework, because dependence computation time costs constantly vary during the execution without an existing policy or a strict model of the adjustments [80]. Therefore, we employ Q-learning as an off-policy and model-free learning strategy for SEADS.

In Q-learning as applied in SEADS, an agent receives a state (i.e., the current configuration) from the environment and takes an action (i.e., selecting a new configuration), either from a Q-table or by a random exploration of possible actions. As a consequence, the agent receives feedback in

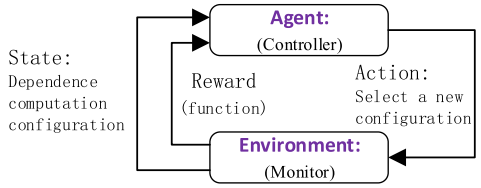

Fig. 10. The interactions between the agent and environment of Q-learning in SEADS.

terms of a reward computed according to the action's performance. As shown in Figure 10, a state represents the current dependence computation configuration, and the *monitor* is the environment while the *controller* is the agent. With the user budget and analysis time cost, a reward is calculated and sent back to the agent as feedback. In the case of a positive reward, the corresponding action is encouraged (i.e., reinforced); otherwise, the action is discouraged. Q-learning uses the reward to update the Q-table whose largest value will be presumably selected as the future action [32]. In other words, for SEADS, the larger the reward is, the more possible the corresponding configuration is selected.

Q-learning updates the Q-table according to the Bellman equation to find optimal policies and value functions [43]. With this equation, the value in the Q-table (i.e., QV) for the next state is computed as follows:

$$QV = QV + \alpha * [reward + \gamma * \max\{q|q \in Q\text{-}table\} - QV], \quad (1)$$

where γ and α are two equation parameters. In particular, γ is a discount factor between 0 to 1 to determine the importance of future rewards. If γ is 0, the Q-learning agent only considers current rewards; if γ is 1, the agent strives for a long-term high reward. The other parameter, α, indicates the learning rate between 0 to 1 to control how much the difference between previous and new QV values is considered. If α is 0, the agent only exploits prior knowledge; if α is 1, the agent considers only the most recent information to explore possibilities and ignores prior knowledge. Another relevant parameter, ϵ, as part of the learning algorithm, is used to control the agent taking an action (i.e., selecting a new configuration) either from the Q-table or by a random exploration of possible actions. If a randomly calculated variable value is less than or equal to (1 - ϵ), the agent uses the epsilon greedy strategy [84] to take the best action according to the largest value in the Q-table. Otherwise, the agent randomly selects an action [26].

Algorithm 3 shows the pseudo-code of the learning process for configuration adjustment. In the first place, SEADS initiates Q-learning components (e.g., *learner*, *Agent*, *Qtable*, *actions*, *rewards*, ϵ, and *states*) and Bellman equation's parameters: γ and α (lines 1–2). For example, we set the values in the Q-table as all zeros and the parameters γ, α, and ϵ as 0.9, 0.9, and 0.2, respectively. Since γ=0.9 is slightly lower than 1, the Q-learning agent prefers a long-term high reward rather than the current reward. Also, α is 0.9, thus the agent prefers for the most recent data (e.g., reward, values in the Q-table). In addition, ϵ = 0.2, thus (1 - ϵ) = 1 - 0.2 = 0.8 = 80%. Therefore, the possibility that the agent takes the best action according to the largest value in the Q-table is 80% while the possibility that the agent randomly selects an action is 20%.

Then, the *reward* and *Qtable* are updated (lines 3–4). The *reward* is defined as 1/(the user budget B - the current dependence analysis time cost T) * 1,000—the rationale is that the closer the analysis cost is to the user budget (the learning goal here), the higher the reward. For example, suppose the user budget B is 6,0000 (ms) and the current analysis time cost T is 43,000 ms. Then the *reward* is 1/(60,000 - 43,000) * 1,000 = 0.0588 as used to update the Q-table. In our algorithm, the action means the transfer from the current state (configuration) to the next state (configuration). If the randomly calculated *probability* value equals or is less than (1 - ϵ), the Q-learning *Agent*

uses the epsilon greedy strategy [8, 77] to take the best action according to the largest value in the *Qtable* (lines 5–7). For example, suppose SEADS randomly calculated the *probability* value as 0.813, which is greater than 0.8. Then, SEADS skips taking the best action according to the largest value in *Qtable*. Otherwise, the *Agent* randomly selects an action (lines 8–9). For example, the *Agent* may randomly take an action (selecting the next configuration). Last, a new configuration *newTCN* (e.g., 000101) is computed according to the action and the current configuration (line 10).

Illustration. For the running example, we first initiate the values in the *Qtable* as all zeros and set Q-learning parameters γ, α, and ϵ as 0.9, 0.9, and 0.2, respectively. The user budget B is 60,000 (ms), and the current time cost T is 43,000 ms after SEADS canceled the static dependence graph construction because of the timeout. Then the *reward* is 1/(60,000 - 43,000) * 1,000 = 0.0588, which is used to update the Q-table. After the *reward* and *Qtable* were updated, the *probability* calculated randomly is 0.813 (> 0.8). And 1 - ϵ = 1 - 0.2 = 0.8. Thus, the *Agent* randomly selects 000101 as the new configuration, which only has two parameters enabled: *methodEvent* and *MethodInstanceLevel*. With this new configuration 000101, SEADS will quickly infer rough and rapid dependence sets only from the dynamic data: full (instance-level) events of all executed methods.

5 IMPLEMENTATION AND LIMITATIONS

We have implemented SEADS as a tool for Java to work with various distributed systems used in the real world. In this section, we discuss some key technical issues for the implementation, as well as its current limitations.

5.1 Analysis of Intraprocess Dependencies

SEADS reused our Java dynamic dependence analysis tools, DIVER [16] and DIVERONLINE [14], both based on the Soot [53] bytecode manipulation and instrumentation framework. In particular, we reused relevant code from DIVER implementation for (1) constructing the static dependence graph for each SUA component, (2) instrumenting and probing for the two kinds of method execution events, and (3) for computing dynamic dependencies at method level within each process with the hybrid dependence analysis approach that utilizes the per-component static dependencies and the intraprocess method execution events. We then reused DIVERONLINE when developing the online version of the dynamic dependence analysis for each process. We also leveraged the capabilities of both tools in handling exceptional handling constructs (e.g., `catch` and `finally` blocks) and computing the data/control flow facts induced by those constructs. This is important for making SEADS work for modern real-world Java systems, since exception-handling code is prevalent in those systems.

In addition, we leveraged the Indus Java code analysis (slicing) framework [65] to compute threading-induced static dependencies. In particular, we reused their code for inferring inter-thread *ready*, *synchronization*, and *interference* dependencies. The fact that Indus is also built on Soot facilitated our integration of these analysis parts into our SEADS framework.

5.2 Inferring Interprocess Dependencies

SEADS derives interprocess dependencies from partially ordered method execution events across all SUA processes throughout its execution, similarly to what we did in DISTIA [19], although we further utilized static dependencies and statement coverage data to refine intraprocess dependencies. Thus, we reused relevant code from DISTIA implementation, mainly for probing for and monitoring message-passing events (i.e., the event of a message being sent and the event of a message being received) that are used to partially order the method execution events at runtime. This

includes handling various kinds of network I/Os in such ways that the runtime monitoring and processing of the message-passing events do not affect the original communication semantics of the SUA, which is crucially important for our Seads tool to work with real-world distributed systems of varied architectures. We also leveraged the capabilities of DistIa in handling exceptional control flows to address message-passing events initiated in exception-handling constructs of Java.

5.3 Responding to User Queries

To enable users to interact with and query the continuously running Seads, we implement the *querying_client* as a command-line tool, which takes user queries and collaborates with the *querying_interface* in each process of the SUA in a classical client/server architecture. More specifically, to avoid potential interferences of user querying with the continuous analysis in Seads, each *querying_interface* communicates with the *querying_client* via a Java *non-blocking* IO (NIO) socket channel [23]. This way, Seads deals with the user interaction *asynchronously* with its internal workings. In contrast, using network I/O working in a blocking and synchronous mode for the communication would be inefficient. Users would typically launch the *querying_client* tool on demand (i.e., whenever dependence querying is needed). Recall that the users here are commonly dependence-based client analyses/techniques that utilize the runtime dependencies to enable particular applications (e.g., diagnosing performance issues and security threats).

5.4 Reuse of Prior Techniques/Tools

We have leveraged a few of our own prior techniques and tools in developing Seads, including Diver, DiverOnline, and DistIa, as described earlier in Sections 3 and 5. While not our main contribution (the key novel contribution of this work is the use of RL to learn, hence tune analysis configurations at runtime), customizations and improvements were necessary for our non-trivial reuse of these existing techniques/tools in Seads. We considerably extended the dynamic interprocess dependence computation algorithm of DistIa, a lightweight dynamic analysis for distributed programs. DistIa was immediately reused only when Seads works at a particular configuration (000100), which only has one analysis parameter (i.e., *methodEvent*) enabled: The analysis uses only one form of dynamic data (i.e., executed method events) while without using any static information of the SUA. For other configurations, Seads only utilizes the algorithm of DistIa for inferring the happen-before relations between method events; it adapted the algorithm to work also with all of the other configurations and for some of them incorporated static dependencies and statement coverage.

Also, we enhanced the algorithms of the Diver/DiverOnline dynamic analysis frameworks for constructing the static dependence graph, probing for dynamic data, and computing dynamic intraprocess dependencies. Diver/DiverOnline can only deal with a single-process program in building a single static dependence graph for the program. For a distributed program, Seads needed to build such a graph for each of the distributed (and decoupled) components of the program. To that end, it had to improve the static dependence analysis algorithm in Diver/DiverOnline to identify and traverse all of the entry points of the distributed program before starting the static analysis with each entry point. Accordingly, in utilizing the static dependencies and method execution events to compute the dynamic dependencies within each process, Seads had further to first identify the right dependence graph (i.e., of the component corresponding to the process) and the method events that only belong to the process. In addition, Diver or DiverOnline only represents one of the multiple static configurations in Seads (i.e., with the parameter *context sensitivity* disabled and the parameter *flow sensitivity* enabled). Seads extended these prior tools to support all of the four static configurations (i.e., for the static analysis to be context/flow sensitive/insensitive).

Table 2. Experimental Subjects

Subject (version)	#Method	#SLOC	Test type
NioEcho (r69)	27	412	Integration
MultiChat (r5)	37	470	Integration
OpenChord (v1.0.5)	736	9,244	Integration
Thrift (v0.11.0)	1,941	14,510	Integration
xSocket (v2.8.15)	2,209	15,760	Integration
ZooKeeper (v3.4.11)	5,383	62,194	Integration, Load, System
Netty (v4.1.19)	12,389	167,961	Integration
Voldemort (v1.9.6)	20,406	115,310	Integration, Load, System

5.5 Limitations

During the instrumentation, SEADS needs to insert probes into the bytecode of the SUA to monitor method events. If administrators do not allow to modify the bytecode, SEADS has no way to work. SEADS targets continuously running distributed systems, offering online dynamic dependence analysis (querying) capabilities with practical scalability and cost-effectiveness tradeoffs. If no querying is performed before the SUA is terminated (hence SEADS exits accordingly), SEADS would not provide useful results, since it does not dump (or save in other ways) its analysis results.

Also, SEADS tries to provide the most cost-effective result (dependence set) achieved within a response time constraint (i.e., the user budget for the average time cost for processing a dependence query). However, our current controller (Q-learning) algorithm may not be optimal—it does not necessarily choose the next configuration that is optimal (i.e., the configuration with which the dynamic dependence computation/updating does not necessarily have the optimal cost-effectiveness tradeoff possible with respect to our framework). For instance, the Q-learning algorithm might take a wrong action (especially when the selection is random—see Algorithm 3) at certain steps. As a result, the dynamic dependence analysis, as informed by the controller, may not be always the most cost-effective as it could be. In addition, if the user sets an improper budget (e.g., one that is far off the typical time cost for answering a dependence query against the particular SUA), the analysis configuration adjustment by the controller can be even less effective (i.e., leading the analysis in SEADS to be further away from optimal cost-effectiveness balances). However, when the user does not specify a budget, SEADS would have to use a default budget, which may not be desirable to the user or not suitable for the given SUA.

6 EVALUATION

Aiming to assess the scalability/efficiency and cost-effectiveness of SEADS and its merits in these regards, our evaluation was guided by the following research questions:

- **RQ1:** How efficient is SEADS in terms of its query response time in an average case?
- **RQ2:** How scalable and efficient is SEADS in terms of its analysis overheads?
- **RQ3:** How cost-effective is SEADS in terms of query response time and analysis overheads?

6.1 Experiment Setup

We applied SEADS against eight Java distributed systems, as shown in Table 2. As study subjects, these systems typically run continuously. The sizes of these subjects are measured as the number of methods defined in the subject source code (the second column *#Method*) and the number of Java source code lines excluding blank lines and code comments (the third column *#SLOC*).

The last column shows the test types, including integration, load, and system tests. We chose these subjects with different architectures, domains, and scales. NioEcho [72] is a simple system whose server echoes any message from the clients. MultiChat [36] is a chat system whose clients broadcast messages to all other clients through the server. OpenChord [75] uses distributed hash tables to provide peer-to-peer network services. Thrift [5] is an application development framework that has a code generation engine for developing scalable cross-language services. xSocket [76] is a framework based on NIO for constructing high-performance, scalable software systems. ZooKeeper [2, 44] is a coordination system providing distributed synchronization and group services. Netty [63] is an asynchronous NIO client-server framework used to rapidly develop network applications. Voldemort [3] is a distributed key-value storage system used by LinkedIn.

In each integration test, we started several server/client instances and performed various operations, to cover main subject functionalities in respective SUA components. For NioEcho, we started a server and a client, and next sent random text messages from the client to the server, and then waited for the echoing of each message. For MultiChat, we sent random text messages from a client to the server and then broadcasted these messages to all other clients. For OpenChord, a peer-to-peer system, we first started three nodes, A, B, and C and then performed following operations: On a machine (node) A, we created an overlay network; on other nodes B and C, we joined the network; on the node C, we inserted a new data entry to the network; on the node A, we searched and then removed the data entry; on the node B, we listed all data entries. For Voldemort, after we started a server and a client, the ordering of our operations is: adding a key-value pair, finding the key for its value, removing the key, and retrieving the pair. For ZooKeeper, we started two instances of a server and a client, and our operations were: creating two nodes, searching them, looking up their attributes, updating their data association, and removing these two nodes.

Particularly for Thrift, xSocket, and Netty, which are all libraries/frameworks, we developed one sample application program of each of them to cover their major functional features, and then exercised these subjects via executing corresponding applications. For Thrift, we developed a calculator application with a server and a client. Some basic arithmetic operations were sent from the client to the server, and the calculation results were sent back from the server to the client. For xSocket, after a server and a client started, the client sent text messages to the server. For Netty, we started a client sending messages to the server. In our developed application programs, each client interacts with the server regularly and infinitely. Besides the integration tests, the load and system tests were downloaded as parts of software packages from respective official project websites.

We note that all of these test cases for our subjects were used as runtime inputs to trigger the subject executions to generate the dynamic information needed by SEADS's analyses, yet we did not aim to use them to test the subject systems or to address specific testing problems. In the article, we did not develop and evaluate any specific dependence-based applications (e.g., testing) but rather focus on the foundational dependence analysis itself; accordingly, our evaluation focuses on assessing the efficiency, effectiveness (i.e., precision), scalability, and cost-effectiveness of SEADS versus the baseline. Recall that SEADS targets SUAs that run continuously; thus, in real deployment settings, the executions analyzed by SEADS are supposed to be uninterrupted. For the purpose of our evaluation experiments, however, we ran each SUA for as long as it needed for ten randomly selected sample queries to be processed by SEADS. And these queries were sent from the *querying_client* with a random interval between 5 to 15 seconds. In our evaluation results, we thus report the cost and effectiveness measures with respect to such executions for the subject systems. For the controller, we set relevant parameters as follows: sgc_T=42 s, sgl_T=12 s, d_T=6 s, TC=1,000, TT=1 mins for the two smallest subjects (NioEcho and MultiChat) and 5 mins for others. In addition, with respect to the likely great variety of user budgets in practice, we specified the user budget for each subject also randomly, ranging from 14 to 200 seconds.

Table 3. Time (in Seconds) and Storage costs (in MB) and Precision (Ratios) of SEADS versus DODA

Execution	Normal Run Time	DODA			SEADS			Precision	Storage
		Run Time	Slow Down	Response Time	Run Time	Slow Down	Response Time		
NioEcho	158.37	228.17	44.07%	14.39	214.15	35.22%	13.71	100.00%	2.00
MultiChat	148.96	241.89	62.39%	15.12	223.67	50.15%	14.32	100.00%	2.00
Openchord	233.78	606.87	159.59%	51.36	359.37	53.72%	25.33	77.44%	14.00
Thrift	199.87	573.49	186.93%	45.23	345.19	72.71%	23.82	90.68%	25.00
xSocket	380.59	1,817.61	377.58%	170.82	772.38	102.94%	65.37	83.25%	21.00
Netty	589.39	4,218.85	615.80%	409.56	1,226.16	108.04%	115.16	87.12%	105.00
Zookeeper Integration	598.34	3,543.47	492.22%	343.19	1,139.25	90.40%	103.21	66.29%	96.00
Zookeeper Load	616.16	3,804.37	517.43%	368.43	1,209.77	96.34%	111.97	65.55%	96.00
Zookeeper System	598.17	3,676.91	514.69%	355.56	1,183.97	97.93%	107.94	63.28%	96.00
Voldemort Integration	398.06	-	-	-	791.37	98.81%	69.62	-	200.00
Voldemort Load	194.44	-	-	-	731.71	276.32%	67.58	-	200.00
Voldemort System	355.78	-	-	-	719.38	102.20%	66.96	-	200.00
Average:	372.66	2,079.07	330%	197.07	743.03	99%	65.41	81.5%	88.08

6.2 Results and Analysis

In this section, we illustrate and discuss our empirical results related to the research questions. One of the key innovations in SEADS is its capability of adjusting analysis configuration at runtime through reinforcement learning to automatically achieve better cost-effectiveness than techniques with a fixated analysis configuration. To show the impact and merits of this innovation, and given the absence of a directly comparable peer technique, we created and used an online version of D^{2}ABS [15], the state-of-the-art dynamic dependence analysis for distributed programs (to the best of our knowledge), as the baseline in our evaluation (referred to as DODA). In terms of implementation, DODA is essentially a variant of SEADS that does not change its analysis configurations on the fly but constantly uses a fixated configuration (111111, for the highest precision possible within our analysis infrastructure)—otherwise, these two tools are not different.

Table 3 presents our major experimental results, including the cost and effectiveness measures of SEADS versus the baseline DODA against the 12 SUA executions (as listed in the first column). The third to fifth and sixth to eighth columns list the total runtime (*Run Time*)—the continuous running time during which we sampled ten random queries with random intervals, runtime slowdown (*SlowDown*), and average query response time (*Response Time*), for each instrumented SUA execution with DODA and SEADS, respectively. The normal runtime (the second column) for each SUA execution was the total length of execution of the SUA against the same sequence of runtime inputs used for driving the corresponding instrumented SUA execution. To enable the comparison between our approach and the baseline, we ensured that, for each SUA execution, the runtime inputs to the SUA were exactly the same during the analysis by both techniques, in addition to feeding them with the same sequence of queries.

In the ninth column (*Precision*), we report the average precision of SEADS via a relative measure (to the baseline): For each SUA execution and each query, the measure was computed as the ratio of the size of query dependence set computed by DODA to the size of the query dependence set computed by SEADS. We chose to compute and report the relative measure for two reasons. First, we do not have the ground-truth dependence sets available for the sample queries, and we are not aware of an existing tool that scales to and works with our subject systems to compute such ground truths. Second, the main goal of our evaluation is to validate the scalability and cost-effectiveness merits (over conventional analyses that have a fixated configuration) of SEADS due to its on-the-fly adjustable analysis configurations and its ability to learn the configurations to adjust to. With respect to this goal, measuring relative cost and effectiveness differences between SEADS and the baseline, which represents the conventional analyses, should suffice. Also, we only report precision as the effectiveness metric, assuming the design (of adjusting analysis configurations) of SEADS does not affect the recall of the dynamic dependence analysis relative to the baseline—because the baseline always uses the most precise configuration while SEADS sacrifices precision for better scalability and overall cost-effectiveness. To validate this hypothesis, we compare the dependence sets between the two techniques against each query—we confirm that SEADS does not sacrifice recall if its dependence set includes the dependence set produced by DODA for the same query.

The last column (*Storage*) lists the total storage costs (disk space taken) of SEADS, ranging from 2 MB for the two smallest subjects (NioEcho and MultiChat) to 200 MB on the largest system (Voldemort), for an average of 88 MB across all the 12 executions. These are the space costs mainly incurred by storing the static analysis data (i.e., the static dependence graph for each SUA component) and the instrumented versions of the SUAs. As these costs with the baseline are almost the same as those with SEADS for each SUA execution, we omitted the numbers for the baseline.

Results for DODA against the Voldemort executions are unavailable (hence, missing from the table) because the baseline did not scale to the system: We killed the analysis after running it for 12 hours. For this reason, the relative precision of SEADS for these executions is missing also. Next, we discuss major findings and observations from these results to answer our research questions.

6.2.1 RQ1: Efficiency: Response time. The response time, as shown in the fifth and eighth columns, is the user's waiting time, since a dependence query is sent out until the user receives the dependence set in return. Over all SUA executions, SEADS took 65.41 seconds on average to respond to random user queries with random intervals and user-specified budgets. For individual executions, SEADS took the shortest response time (13.71 seconds) on average against the NioEcho execution, most plausibly due to its smallest size. However, SEADS took the longest average response time (115.16 seconds) on Netty, the largest system among our subject SUAs. Yet looking at this efficiency measure across all the SUA executions reveals no consistent correlation between subject sizes and the average response time. One reason is because the source size of a subject is not the only factor that affects this efficiency measure—for example, the complexity of the execution analyzed is another major factor here. In fact, the three SUA executions for the same SUA Voldemort saw noticeably different average response time with SEADS. Other factors, such as the time cost of network communication and that of merging dependence sets while deriving inter-process dependencies at the *querying_client* after it receives all per-process dependence sets, also have non-trivial effects on the response time perceived by the user.

In addition, the variations in the response time have to do with how SEADS works: When it receives a query, SEADS may be in one of two possible situations, as mentioned earlier: (1) computing/updating dependence sets for all queries or (2) being idle during the time interval between two consecutive rounds of dependence computation. In the first case, the relevant dependence sets will be delivered back to the user after the computation is done. Depending on the timing of query

arrivals, the query response time can be substantial in this case. This situation often occurs when the user requests a query for an SUA of a large code size or a great execution complexity, for which the dependence computation may take a relatively long time. Of course, even in this situation, the response time can still be fairly short: For example, for small subjects such as NioEcho and MultiChat, any round of dependence computation is very fast, so SEADS can deliver newly computed dependence sets shortly after receiving a dependence query. Otherwise, in the latter case, the computed results (i.e., dependence sets) are sent back to the *querying_client* at once, since the results (from the most recent round of dependence computation) are already available. In consequence, the response time is generally short in this case.

For the same query requests as sent to SEADS, DODA took 197 seconds by average over the 12 SUA executions, with the average response time for individual SUA executions ranging from 14.39 seconds on NioEcho to 409.56 seconds on Netty. In particular, for Voldemort, DODA could not answer any user query within 12 hours—as mentioned earlier, we had to kill, after that long time, the analysis by DODA against this subject SUA (for any of its three executions, because the scalability challenge mainly lies in the static dependence analysis part). In practice, a user (either a human or an application/client analysis using the dynamic dependence results) is not very likely to wait even longer than 12 hours for querying a dependence set. That is, DODA may suffer a serious scalability problem that impedes its practical adoption to industrial-scale distributed systems. In contrast, for the three executions of Volemort studied, SEADS took a bit over one minute to respond, highlighting the scalability and efficiency advantages of our approach over the conventional dependence analysis.

It is worth noting that the efficiency advantage in terms of mean response time of SEADS over the baseline was generally more significant with larger-scale SUA executions. As shown, the gap in this efficiency measure between the two techniques tended to increase when the SUA grows in source size and execution complexity—for instance, for the two smallest SUAs, the average response time of DODA was very close to that of SEADS (the difference was less than one second on average for each query); for a medium-scale SUA such as Thrift, SEADS was about 2× faster; and for the largest SUAs, SEADS was over 3× faster. This further implies that the efficiency/scalability merits of SEADS over DODA are especially important and needed for large, complex real-world distributed systems.

Answer to RQ1: SEADS is more than 3x faster than the baseline by responding to user queries within about 65 seconds versus the baseline taking 197 seconds for the same queries, on average over 12 unique system executions of eight SUAs. Particularly, the baseline could not be applied to large-sized distributed systems, such as Voldemort, due to its severe scalability issues. In contrast, SEADS was able to scale to such systems and respond to user queries reasonably fast, demonstrating the scalability advantage of our approach over the conventional dynamic dependence analysis approach.

6.2.2 RQ2: Efficiency: Analysis Overheads. Beyond the average response time, we further gauged the efficiency/scalability of SEADS relative to the baseline in two measures of analysis overheads: the runtime slowdown caused by the instrumentation, and the storage costs. The measurement of runtime slowdown followed the standard computation. In our study in particular, we were concerned about the overall slowdown of each technique during the *entire continuous execution* of each SUA that we considered. Thus, this measure was computed as the percentage of increase between the *Normal Run Time* (the second column of Table 3) and the *Run Time* of each technique (the third and sixth columns). Specifically, a runtime slowdown of DODA was calculated as $(T_D\text{-}T_n)/T_n$, where T_D was the runtime of the SUA instrumented by DODA and T_n was the runtime of

the original SUA. For example, the original NioEcho executed 158.37 seconds, and the runtime of the DODA-instrumented NioEcho was 228.17 seconds. As a result, the runtime slowdown of DODA in this case was (228.17 - 158.37)/158.37 = 0.4407 = 44.07%, as shown in the fourth column (*Slow Down*). Similarly, a runtime slowdown of SEADS was computed as $(T_S\text{-}T_n)/T_n$, where T_S was the runtime of the SUA instrumented by SEADS (the sixth column *Run Time*). For instance, the runtime of NioEcho instrumented by SEADS was 214.15 seconds. Thus, we calculated the runtime slowdown caused by SEADS as (214.15 - 158.37)/158.37 = 35.22% (the seventh column *Slow Down*).

From Table 3, we can see that the runtime slowdown of SEADS ranged from 35% (NioEcho) to 276% (Voldemort load test), with 99% on average over the 12 SUA executions. In comparison, the runtime overheads of DODA ranged from 44% (NioEcho) to 616% (Netty), with 330% on average. Recall that both techniques perform dynamic dependence analysis *online*. Thus, the slowdown was resulting from the time cost of the online analysis (e.g., collecting analysis data and computing dependencies, both during the instrumented SUA execution). Therefore, the slowdown measures are connected to the factors that influence the online analysis costs. Intuitively, these factors are similar to those that caused the variations in the average response time in relation to the source size and execution complexity of the subject SUAs. Generally, the slowdown of either technique was greater against larger SUAs with more complex executions, as expected.

Although the runtime of DODA-instrumented Voldemort is absent in the table as explained earlier, we know that for each of three Voldemort executions the time was at least 12 hours. Since DODA did not answer any dependence query within that period of time, the corresponding slowdown measures were not meaningful and thus omitted—albeit they would be extremely high: at least (12*3600 - 398.06/398.06) = 4,319,900%. For the other nine SUA executions for which the individual runtime slowdown measures were available for both techniques, SEADS was consistently more efficient than the baseline. Similar to their contrasts in average response time, the advantage of our approach over DODA was increasingly significant for SUAs of growing size and execution complexity. For instance, while for the two smallest and simplest SUAs the slowdowns of both techniques were close, DODA incurred over 2× greater slowdown than SEADS for a medium-scale SUA xSocket; for the large-scale SUAs like Netty and ZooKeeper, DODA's slowdown was 4–5× greater. Overall, SEADS was over 3.3× as efficient as the conventional approach to dynamic dependence analysis in terms of the slowdown measure, further elucidating the merits of the on-the-fly analysis configuration adjustments through reinforcement learning in our approach.

In terms of the other overhead measure, storage cost, the two techniques compared were very close both for any individual SUA execution and overall. Thus, SEADS did not have substantial advantages in this regard. Yet given the generally negligible storage costs as shown in (the last column of) Table 3—no more than 200 MB—the results on this overhead measure do not affect the efficiency and scalability advantages of SEADS against the baseline otherwise.

Answer to RQ2: For the SUAs and their executions that DODA can scale to, SEADS incurred mostly no more than 1x run-time slowdown, compared to DODA causing 2–6x slowdown. On overall average, SEADS caused a 99% slowdown versus 330% by the conventional dynamic dependence analysis, with greater advantages against larger-scale systems. Both techniques were highly and similarly scalable in terms of storage costs.

6.2.3 RQ3: Cost-effectiveness: Precision-cost Ratios. To evaluate the cost-effectiveness of our approach, we need to first compute the precision by comparing the average sizes of query dependence sets computed by DODA and SEADS. The precision for each SUA execution, as shown in the ninth column (*Precision*) of Table 3, is the average ratio of the size of the dependence set for each

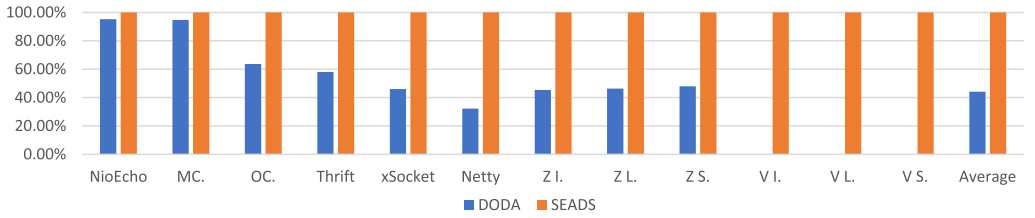

Fig. 11. Comparisons (y axis) of the cost-effectiveness expressed as the ratios of the precision to the response time of DODA and SEADS per execution (x axis). The higher the ratio, the more cost-effective.

query computed by DODA to the size of that computed by SEADS for the same query. For each dependence query, we also compared the content of both dependence sets, and found that *SEADS's dependence set always subsumed the dependence set given by DODA* for any of the queries involved in our evaluation—this confirms that SEADS, although sacrificing precision by adjusting analysis configuration to achieve higher scalability and efficiency, had no loss in recall. These relative effectiveness measures essentially treated the baseline results as ground truths. Thus, given the equally 100% recall of both techniques, we only considered the relative precision of SEADS (with DODA precision as constantly 100%) when computing the cost-effectiveness of both techniques as the ratio of effectiveness to cost.

Our results show that, for the nine SUA executions for which the baseline dependence sets were available to enable the relative precision measurement for SEADS, the precision achieved by SEADS ranged from 63% to 100%, for an overall average of 82%. In the best cases, for the two smallest SUA and simplest SUA executions (i.e., NioEcho and MultiChat), SEADS did not lose any precision relative to the baseline. The reason was mainly because the online analysis by SEADS constantly incurred time costs lower than the user budgets even with the most precise analysis configuration for these subjects; thus, SEADS did not need to switch away from the highest-precision configuration it started with throughout the entire online analysis. Likewise, SEADS had the lowest precision of 63.28% for Zookeeper system test, most plausibly because SEADS experienced the most aggressive and frequent adjustments of its analysis configuration to maintain scalability and efficiency with respect to the given user budgets. That is, the average (relative) precision (over the 10 queries) that SEADS achieved for an SUA execution had to do with the size and complexity of the SUA execution, which was reflected in part in the two efficiency metrics (response time and runtime slowdown). Indeed, the numbers in Table 3 generally revealed a connection between the precision (the ninth column) and those efficiency metrics (the seventh and eighth columns)—the shorter average response time and smaller slowdown mostly went with higher precision, despite the absence of an always consistent correlation. Such potential interplays between precision and costs further necessitate the assessment of cost-effectiveness as a holistic measure.

To compute the cost-effectiveness measure for each technique, for each SUA execution, we calculated the ratio of the average precision (over the 10 queries) to one of the two cost measures we considered: average response time (over the 10 queries), and runtime slowdown (overall during the entire execution across the 10 queries). Accordingly, we had two measures, each with respect to one of the two cost measures, in our cost-effectiveness assessment and comparison between the two techniques, shown in Figure 11 (with average response time as the cost factor) and Figure 12 (with the runtime slowdown as the cost factor), respectively. For ease of presentation with respect to space constraints, we use abbreviations in both figures as follows (on the x axis): MC. for MultiChat, OC. for OpenChord, V for Voldemort, Z for ZooKeeper, I. for integration test, L. for load test, and S. for system test. In particular, to highlight the merits of our approach over the

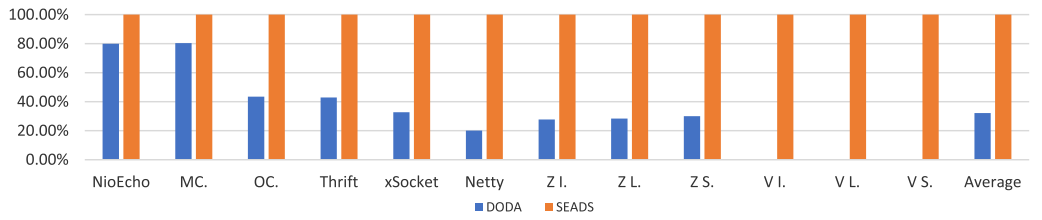

Fig. 12. Comparisons (y axis) of the cost-effectiveness expressed as the ratios of the precision to the runtime slowdown of DODA and SEADS per execution (x axis). The higher the ratio, the more cost-effective.

baseline, either figure only shows how SEADS compared to DODA in terms of the cost-effectiveness per SUA execution (indicated along the y axis), as the percentage of DODA's cost-effectiveness over the cost-effectiveness of our approach, rather than showing the individual cost-effectiveness measure numbers of each technique separately. This is why both figures show the bars for SEADS constantly corresponding to 100%, whereas the cost-effectiveness of the baseline is shown as a fraction of that of our approach. The rationale of doing so is to normalize the measure values, especially ironing out the large differences in cost measures across these SUA executions. Because DODA could not be applied to Voldemort, the corresponding cost-effectiveness measures were zero. We do not consider storage costs as another cost factor in computing the cost-effectiveness because they were almost negligible (only 88 MB on average and 200 MB at most), hence did not affect much the efficiency/scalability of either technique.

In Figure 11, we can see that the cost-effectiveness (with respect to response time) of DODA was about 44% of that of SEADS on average. For individual SUA executions, DODA and SEADS had very close cost-effectiveness against the two smallest and simplest SUAs, NioEcho and MultiChat. In RQ1 and RQ2, we observed that the efficiency and scalability advantages of SEADS over the baseline were more prominent when applied to larger and more complex systems. This comparative trend applies here also: The cost-effectiveness merits of our approach were more substantial with SUAs of large scale and more complex executions, compared to DODA. For instance, DODA cost-effectiveness was around 60% of that of SEADS for medium-scale SUAs like Open chord and Thrift, while the ratio went down to less than 45% for larger systems like ZooKeeper and further down to 30% for even the larger SUA Netty; at one extreme of this trend, the baseline cost-effectiveness was zero for Voldemort, the most challenging SUA in our study.

Concerning the runtime slowdown as the cost factor, Figure 12 shows the cost-effectiveness contrasts between the two techniques in the alternative measure. Overall, the cost-effectiveness with respect to this cost measure of DODA was only about 32% of that of SEADS on average, substantially lower than the cost-effectiveness with respect to average user-query response time as shown in Figure 11. The reason is apparently because the efficiency and scalability advantages of SEADS over the baseline in terms of runtime slowdowns were greater than those in terms of average response time. However, compared to the cost-effectiveness variations in relation to the underlying SUA executions analyzed in Figure 11, the variations in the cost-effectiveness with respect to runtime slowdown were similarly associated with the scale of the SUA executions—the cost-effectiveness advantages of our approach were greater against larger and more complex SUAs.

Put together, SEADS was substantially more cost-effective than DODA, regardless of the cost factor concerned with. This implies that, although looking at the precision losses alone seemed to suggest that SEADS sacrificed effectiveness significantly in exchange for higher scalability and efficiency, the precision sacrifices were well paid off by the gains in efficiency and scalability, resulting in the ultimate merits in cost-effectiveness overall. Thus, our methodology of adjusting

analysis configurations at runtime appeared to be a scalable and cost-effective solution to the dynamic dependence analysis of large-scale, real-world distributed systems.

Answer to RQ3: On overall average, the precision of SEADS was about 82% relative to the baseline, with equal recall between the two techniques. This effectiveness loss of SEADS was far outweighed by its efficiency and scalability gains compared to the baseline, though. As a result, the average cost-effectiveness of the baseline was only 44% and 32% of that of our approach, with respect to average user-query response time and run-time slowdown, respectively. Moreover, the cost-effectiveness advantages of our approach were even greater for larger-scale and more complex distributed systems.

6.3 Accuracy Validation based on Ground Truth

As we discussed earlier, for the evaluation of our technique SEADS versus the baseline DODA, ground-truth dependence sets for the studied subjects and executions are unavailable, while there are no existing tools available to compute them either. Meanwhile, thoroughly identifying all ground truth needed with manual effort would be too expensive to be practical. Nevertheless, it is still useful to manually construct partial ground truth to validate the accuracy of both techniques to provide confidence about their correctness.

Methodology. Given the tedious nature of the manual process, we limited the validation scale to 10 randomly chosen sample queries per subject execution for which the method-level dependence sets computed by DODA included no more than 30 methods. For each of the 12 subject executions, we manually constructed the ground-truth dependence set TD for each of such queries q through understanding the subject code and tracing the program execution paths starting at the query. We then computed the precision and recall of DODA for each query q according to TD and the dependence set DD produced by DODA for q. For SEADS, we assessed its precision and recall for the same query q by comparing DD against each of the dependence sets SD produced by SEADS for q at each of five querying times with random intervals during SEADS's continuous analysis.

Results. Our manual study revealed that DODA had overall average precision of 97.7% with constantly 100% recall for the 10 random queries across the 12 subject executions, according to our manual ground truth. The imprecision was mainly attributed to the interprocess dependence approximation that was based on happens-before relations between method events across processes. For SEADS, we validated that for each query q, the dependence set given by DODA was always a subset of that given by SEADS (i.e., DD-SD==$\emptyset$), which suggested that SEADS had the same recall as DODA did (i.e., 100%).

We also checked dependencies reported by SEADS but not by DODA (i.e., SD-DD) and found that all of those dependencies were false positives, further supporting the 100% recall of the baseline. Our results showed that these false positives led SEADS to a lower precision than DODA—80% on overall average with respect to the manual ground truth, or 81% with respect to the baseline dependence sets (which is consistent with the results from our empirical evaluation in Section 6.2).

6.4 Speed of Configuration Learning

As SEADS uses Q-learning to learn analysis configurations for obtaining and maintaining scalability and better cost-effectiveness of the dynamic dependence analysis, its controller module takes time to learn, hence start producing reasonably good decisions (i.e., the next configuration to switch to). In fact, Q-Learning as an iterative learning method widely used in approximate dynamic programming for Markov decision processes (MDPs) computes an optimal MDP policy

Table 4. The Number of Iterations and Learning Time (in seconds) of Q-learning in SEADS

Execution	#Iteration	Time
NIOEcho	1	36.73
MultiChat	1	41.38
OpenChord	2	327.19
Thrift	2	316.70
xSocket	3	639.37
ZooKeeper_integration	4	1013.73
ZooKeeper_load	4	1097.26
ZooKeeper_system	4	1053.82
Netty	4	1132.98
Voldemort_integration	3	697.35
Voldemort_load	3	632.42
Voldemort_system	3	621.94
Overall Average:	**3**	**634.24**

through multiple iterations such that the averaged dynamics could be desired with convergence properties [60]. To see how fast SEADS can learn cost-effective configurations, we have collected the numbers of Q-learning iterations and learning time before SEADS started computing more cost-effective results (i.e., when the controller started stably choosing the next configuration that led to more cost-effective dynamic dependence computation than would the current configuration) in our empirical evaluation (Section 6).

Table 4 shows the number of learning iterations (*#Iteration*) and learning time (*Time*) in seconds for each of our 12 subject executions (*Execution*). As we described earlier, executions for all of the subjects other than Zookeeper and Voldemort were driven by respective integration tests. Generally, SEADS tended to take more iterations to learn better configurations for more complex system executions, intuitively because of the greater variations in the dynamics of these executions. Note that the learning time included the time cost of dependence computations across the corresponding iterations. According to the efficiency results in Table 3, SEADS took longer time to compute dependencies, hence responded more slowly to dependence queries for more complex system executions. This explains the observation here that the learning time was generally longer as well for those system executions. On overall average, SEADS needed three rounds of learning and 634 seconds before it started achieving more cost-effective results.

6.5 Optimality of Configuration Learning

From our evaluation results, we see that although SEADS has scalability and cost-effectiveness advantages over DODA, SEADS is not optimal and suffers an inability to maximally utilize the user given time budget while having noticeable room for improvement in precision (only 81.5% on average relative to the baseline). An immediate explanation for this inability is that the controller in SEADS did not always give the optimal (i.e., most cost-effective possible) configuration for the dependence analysis algorithm to take, as we mentioned earlier in Section 5.5. Our further examination suggested that a plausible underlying reason for the lack of optimality in our configuration learning is that currently SEADS simply uses a generic Q-learning algorithm whose reward function is not optimized for a particular subject—the algorithm is not aware of the particular

characteristics of the subject and does not consider the different execution characteristics across different subject systems.

Towards gaining the optimality of configuration learning in SEADS in the future, we have multiple directions to pursue. One possible idea is to develop a control-theoretical method using a feedback control mechanism to predict optimal configurations for the analysis algorithm so it can utilize as much of the user-specified analysis time budget as possible to obtain the highest level of precision possible when the configuration adjustment is arbitrated. Another idea is to optimize the Q-learning algorithm used by the SEADS controller to provide optimal cost-effectiveness tradeoffs for the analysis algorithm by learning a reward function that is specific to (i.e., parameterized for) each particular subject execution. We will also consider exploring other popular learning methods to optimize the controller underlying our dynamic dependence analysis, such as multi-step Q-learning [60], self-adapt neural network [27], multi-modal deep learning [66], and so on.

6.6 Threats to Validity

Our empirical results are subject to various common kinds of validity threats according to Reference [82]. We describe below each major kind and discuss how we control or mitigate relevant threats.

Internal validity. The major threat to internal validity concerns potential mistakes in the implementation of SEADS, DODA, and our experimental procedure. Errors in any of these implementations would compromise the validity of our empirical results and our conclusions drawn based on the results. However, SEADS is based on Soot [53], a framework that has matured over a decade. Many of the key components of SEADS and DODA, including the code for static instrumentation, static dependence analysis, runtime monitoring/profiling, and hybrid computation of dynamic dependencies, were drawn from tools developed in previous work [14, 16, 19] which have been debugged and tuned for years. To minimize the threats concerning the experimentation scripts and newly developed components of SEADS and DODA (e.g., the controller backed by the Q-learning algorithm), we conducted careful code review and manual inspection against simple samples (e.g., the two smallest SUAs) and cases (e.g., queries with relatively small dependence sets) to ensure functional correctness.

Another kind of internal validity threat is that the *instrumenter* of SEADS may cause false negatives and false positives. First, our instrumenter is static, thus it cannot instrument dynamically loaded code, which is not available during the instrumentation phase. As a result, SEADS cannot catch *entry* and *returned-into* events of methods invoked in the dynamically loaded code, which may lead to false negatives in its profiling step, hence later in its dynamic dependence computation. Dynamic instrumentation would overcome this threat, but it would cause a portability problem, because it typically needs to customize underlying platforms such as runtime systems or operating systems. Second, the message-passing API list used by the instrumenter might be incomplete, thus SEADS may miss some message-passing events at runtime that are necessary for timing synchronization across distributed processes. This incompleteness could lead to incorrect partial ordering of method execution events, hence false positives and/or false negatives in the dynamic dependence analysis in SEADS. To mitigate this implementation-wise threat, SEADS currently includes the most commonly used kinds of message-passing APIs by default. Users can readily supplement the default list with other such APIs used in their systems under SEADS's analysis.

External validity. One threat to the external validity of our results lies in the representativeness of the subject SUAs and their executions used in our evaluation study. The SUAs we chose may not well represent all real-world distributed systems that SEADS could apply to, and the executions considered for each chosen SUA may not have exercised all the typical behaviors of that SUA

(or may not reflect its representative operational profiles). If the differences between our sample SUA executions used and representative distributed system executions are significant, users of SEADS may experience its performance and merits differently from what we reported here. We have attempted to reduce these threats by considering subject SUAs of varying size, architecture, and application domains, as well as execution scenarios of varied kinds. Nevertheless, to minimize such threats, we would need to use real operational scenarios of real-world distributed systems in their actual deployment settings.

Construct validity. The baseline chosen for the comparative evaluation is not ideal—to avoid potential biases, we would need to use a state-of-the-art online dynamic dependence analysis tool developed by others (rather than ourselves) that at least works with some (if not able to scale to all) real-world distributed systems. Even more desirably, we would want to use as the baseline a *scalable and cost-effective* dynamic dependence analysis tool for distributed systems, which may achieve the scalability and cost-effectiveness (potentially at different levels from those offered by our approach) through different methodologies from ours. Given the unavailability of such desirable baselines, we used DODA as an alternative. This choice may have led to biases in our evaluation, since DODA was also developed by ourselves. To reduce this threat, we built DODA on top of the state-of-the-art dynamic dependence analysis for distributed software systems while ensuring both DODA and SEADS share underlying analysis infrastructure and utilities as far as possible.

In addition, given the unavailability of actual ground-truth dependence set for each query, we only considered relative measures to evaluate the precision and recall of SEADS with respect to the baseline's results, which constitutes another threat to construct validity. Also, the dependence queries chosen may not represent real queries sent by users (either human users or application analyses/tools) in practice. A similar threat concerns the user budgets considered for each subject SUA execution. To reduce these threats, we randomly chose the queries and sent them at random intervals, and specified user budgets also randomly, trying to cover a variety of these inputs to SEADS in real use scenarios. Finally, we only observed continuous executions within a limited amount of time and used a limited number of queries in the evaluation. As a result, users interacting with real-world distributed systems for a much longer time with much more queries sent at different intervals from those in our experimental setup could obtain performance results with our techniques that are different from what we reported.

Conclusion validity. The main conclusion validity threat lies in the generalizability of our evaluation results and conclusions. Due to the limited number and diversity of subject SUAs and SUA executions we considered, as well as the discussed concerns with the baseline choice and other experimentation settings, we do not claim that our results (e.g., those supporting the substantial cost-effectiveness advantages of SEADS over DODA) generalize to an arbitrary real-world distributed system in all operational scenarios while in comparison to any other relevant baseline approaches. Finally, our conclusions based on the results of the dynamic dependence analysis at method level may not generalize to finer-grained levels (e.g., statement level), at which the analysis would face much greater scalability and efficiency challenges.

7 DISCUSSION

In the section, we discuss additional aspects of our technical approach and empirical evaluation.

7.1 Objectives of Empirical Evaluation

Compared to the baseline approach DODA that prioritizes analysis precision with a fixated configuration, SEADS sacrifices (relaxes) analysis precision to contain the analysis cost within the user-specified time budget. It is intuitive that the price for a less precise and accurate analysis is

generally lower. Thus, it was well expected that SEADS would be less precise and more efficient than the baseline. Yet, we cannot simply assume and expect that the precision loss would be always outweighed by the efficiency gains, hence the overall cost-effectiveness would be higher than the baseline. Therefore, while the general evaluation results on the efficiency merits of our approach are not surprising, it is still important to deeply measure the precision loss and efficiency gains of SEADS compared with DODA as done in our empirical evaluation, to explicitly validate and quantify the cost-effectiveness and scalability advantages of our approach.

7.2 System-level Scalability Challenges

A distributed system may start with a very large number of concurrent processes in its executions. It is also likely that the number of processes in the execution of a long-running (or continuously running) system increases to be very large at runtime. In either situation, the scale of the distributed system execution may generally cause a scalability challenge to SEADS when applied against such systems. The goal of SEADS is to address the scalability issue (with dynamic dependence analysis of long/continuously running distributed systems) caused by the great overhead of both of its static and dynamic analysis parts. Although our evaluation results indicate that the overall cost/overhead dealt with by SEADS was dominated by the cost of its static analysis (i.e., computing the static intracomponent dependencies) part, generally the cost of its dynamic analysis part can become substantial, too (e.g., when the system execution has a very large number of processes to start with or added in the middle of the execution). Such substantial dynamic analysis costs would trigger SEADS to adjust its analysis configurations to strive for a good cost-effectiveness tradeoff and maintain scalability.

Meanwhile, when there are a large number of processes in the system execution, a challenge to the underlying platform and infrastructure (e.g., computing power and network bandwidth) would arise. If these underlying resources cannot scale to the distributed system execution itself, the execution of SEADS would certainly be impeded as well. Thus, since it works purely at application level, SEADS deals only partially with the system-level scalability issues that involve the runtime platform and computing infrastructure of the distributed system under analysis.

7.3 Analysis on Execution Slices

While SEADS aims to address scalability and cost-effectiveness balancing challenges for dynamic dependence analysis based on long-/continuously running executions (hence long/unbounded traces), it is also feasible and useful to adopt a dynamic analysis technique on a (relatively short) slice of the executions (i.e., trace slice). SEADS can be readily adapted to work on a trace slice, too. However, there are two issues to be considered for such an analysis against a long-/continuously running system. The first one is that the dependence analysis working on a trace slice may be incomplete: Without previous runtime states (e.g., the events occurring prior to the start of the slice) of the system being considered, the dependencies computed for the slice are not complete with respect to the system's entire execution up to the end of the slice. The second one is that some dynamic dependence-based applications may produce incorrect analysis results if only a slice of the execution is analyzed. For instance, in a dynamic taint analysis (based on dynamic dependencies), when a source is executed beyond a slice while a sink that is actually reachable from the source at runtime is executed within the slice, the dynamic information flow path from the source to the sink (hence the sensitive data leak) would be missed. In this article, we address dynamic dependence analysis with complete executions of continuously running systems (i.e., unbounded traces).

7.4 Online versus Offline Analysis

While we designed SEADS as an online technique, offline dependence analysis and its applications (e.g., impact analysis) are generally useful as well. In fact, our previous techniques (e.g., DIVER, DISTIA) that compute dynamic dependencies for dynamic impact prediction are mostly offline dynamic analysis approaches. Yet, we note that these prior techniques only address short-running programs producing traces that are not very large, while considering the entire traces. They would not work with long-/continuously running programs that produce infinite and voluminous traces.

For long-/continuously running programs, when we need to consider entire execution traces (we exemplified such situations in Section 7.3), an offline analysis is generally infeasible, because the execution traces are unbounded, hence cannot be completely serialized. However, an offline analysis would work well when (1) the program does not run very long and whole program execution traces are not very large, even though entire execution traces need to be analyzed, or (2) only a part of the execution traces needs to be analyzed. For both cases, SEADS would readily accommodate as well by just considering the trace part (slice) of interest.

We also note that although we have referred to impact analysis as an example application of the dynamic dependence analysis offered by SEADS, impact analysis itself is just one of the many possible applications based on dynamic dependencies. In this article, our focus is more generally on dynamic dependence analysis rather than on its particular application to impact analysis. Also, with SEADS, we mainly address the situations in which entire program execution traces need to be analyzed while these traces are unbounded.

8 RELATED WORK

There are several classes of prior work that are most related to ours: *program dependence analysis*, *dependence analysis for distributed systems*, *analysis with variable cost-effectiveness*, and *adaptive software systems and techniques*.

8.1 Program Dependence Analysis

Dependence analyses are used to infer dependence relationships among program entities and to further reason about the relationships for coding, debugging, and testing software systems [50, 61]. We can compute program dependencies using static, dynamic, or hybrid approaches. A static dependence analysis approach computes dependencies without referring to the program executions, while a dynamic dependence analysis solution utilizes the execution data of programs for the computation [56]. Another type of analysis, called hybrid analysis, combines static and dynamic analysis techniques [69] to deduce dependencies. According to the nature of the analysis result (dependencies), hybrid analysis can be regarded as a special type of dynamic analysis, as opposed to purely dynamic analysis, since its results are dynamic: Its results only hold for particular executions utilized (as opposed to static analysis producing results that hold for all possible executions). In the article, we focus on the *hybrid* approach to *dynamic* dependence analyses.

One example of dynamic analysis is dynamic program slicing. For instance, Korel and Laski [48] utilized arrays and dynamic data structures to significantly reduce the slice size for a finer localization of program fault(s). Zhang and Gupta introduced a dynamic program slicing algorithm OPT based on a precise dynamic dependence graph (dyDG) representation that is rapidly traversable [86]. Hybrid approaches have also been proposed for dependence computations. For instance, DIVER employs a static dependence analysis to significantly reduce the computation time of the final dynamic dependence analysis [16]. Existing dependence analysis approaches mostly focused on single-threaded systems. For example, besides DIVER, TRACERJD is also a dynamic dependence analysis approach, including a static analysis phase, for single-threaded programs

[17]. In addition, Rus and Rauchwerger proposed a hybrid dependence analysis for automatic parallelization to achieve almost maximum parallelism with minimum runtime overhead using an integrated compiler via seamlessly merging static and dynamic analysis techniques together [69]. Concurrent program slicing deals with multi-threaded but mostly single-process code, as exemplified in References [35, 37] as dynamic approaches—most existing approaches in this domain are static [21, 35, 51, 58, 65]. However, for multi-process applications, SimEvo employs static and dynamic analysis techniques to identify system-level concurrent dependencies [85], thus it can also be considered a hybrid analysis approach.

SEADS clearly differentiates itself from these existing dependence analysis approaches in that it targets distributed software, which executes in multiple, distributed processes each including single or multiple threads. Moreover, the defining distinction of SEADS lies in its changing analysis configurations on the fly, versus existing approaches commonly using a fixated analysis configuration. Meanwhile, SEADS leverages the state-of-the-art hybrid dependence analysis approaches (i.e., References [16, 19]) for computing intraprocess dependencies.

8.2 Dependence Analysis for Distributed Systems

Some dependence analyses have enabled numerous client applications for distributed systems, including impact prediction [33, 62, 74] and performance optimization [46]. Yet these analysis approaches are static and limited to specialized systems (e.g., DEBS [57] and RMI-based systems) or a customized language that relies on developer annotations [25]. As a dynamic analysis for commonly deployed distributed systems, DISTIA, including its basic and enhanced versions, can compute dependencies both within and across processes by exploiting the partial ordering of executed methods. Moreover, the enhanced version of DISTIA is much more cost-effective than the basic version, because the former prunes methods that do not satisfy the message-passing semantics to improve the cost-effectiveness [19]. Relative to these approaches, SEADS differs clearly in that it computes dynamic dependencies while achieving an even greater level of cost-effectiveness than DISTIA by utilizing static dependencies and statement coverage when the user budget allows.

DISTTAINT [29], a purely application-level dynamic information flow analyzer for common distributed programs, computes intraprocess and interprocess dependencies from globally partial-ordered execution method events to handle implicit dependencies with high analysis precision at a fine-grained (statement) level. DISTTAINT achieves practical cost-effectiveness and resolves multiple technical challenges, including applicability, portability, and scalability challenges, through a principled, multi-phase analysis strategy [31]. The key difference between DISTTAINT and SEADS is that the former as an information flow analyzer is essentially a targeted dependence analysis: For a given query, only the dependencies through which the query reaches to known targets (i.e., the sinks) are computed. Thus, compared to SEADS as a general dependence analysis, DISTTAINT has a much narrower analysis scope. Moreover, like other existing dependence analyses, DISTTAINT uses a fixated configuration during the entire analysis. Also, DISTTAINT is an offline dynamic analysis, as opposed to the online analysis in SEADS.

8.3 Analysis with Variable Cost-effectiveness

Most existing analysis approaches commonly suffer from challenges of balancing the analysis cost and effectiveness: They are either precise but too expensive or efficient but too imprecise. To deal with these challenges, one existing solution is to offer variable cost-effectiveness balances to satisfy varying user needs. For example, the DIAPRO framework provides flexible cost-effectiveness choices for a variety of levels of cost-effectiveness tradeoffs with the best options for variable user requirements and budgets [18]. By combining the static and dynamic data, DIAPRO unifies PI/EAS [6], DIVER [16], and three dependence-based dynamic impact analysis techniques: one using

coverage and trace, one using aliasing and trace, and the other using all these dynamic data (aliasing, coverage, and trace). Another example is D^{2}Abs [15], which aims at practical scalability and offers various levels of cost-effectiveness tradeoffs in the dynamic dependence analysis for distributed programs. Its most precise computation of runtime dependencies has been used to measure interprocess communication (IPC) coupling in distributed systems [30]. To achieve different cost-effectiveness tradeoffs, D^{2}Abs provides four versions each enabling and disabling different analysis steps. In contrast to these peer approaches, which achieve variable but a small number of (four or less) cost-effectiveness levels via the same number of versions of an analysis with each still utilizing a fixated analysis configuration, Seads achieves a much greater and easily extensible number of cost-effectiveness levels in one analysis that adjusts its configurations on the fly.

8.4 Adaptive Software Systems and Techniques

Self-adaptive systems automatically adapt themselves to continuously meet requirements in dynamic, uncertain, and unpredictable environments [71]. Self-adaptive systems also monitor relevant changes in the environments and adapt themselves to ensure adaptations [28]. For example, Bodden [13] developed a dedicated domain-specific language and intermediate representations to express self-adaptive static analyses. These representations allow for an automatic adaptation of the code of the analysis, both ahead-of-time (through static analysis) and just-in-time during the execution of the analysis. In addition, Heo et al. proposed a new technique for developing a resource-aware program analysis [39], which monitors the resource usage of the analysis and adjusts the analysis's behaviors by coarsening program abstraction usually using less memory and time to meet the constraint on the usage of physical resources (e.g., memory). The analysis adjusts itself many times under the direction of a controller to decide how much the analysis should coarsen the abstraction, with the varying computation of the analysis. Seads shares a similar goal with these adaptive approaches, but it adjusts itself (in its configuration) concerning the analysis cost and user budget both in terms of time as opposed to hardware resources (e.g., memory). Also, Seads is a dynamic analysis versus the peer approaches being static. However, some control-theoretical approaches have been designed for self-adaptation, such as References [34, 49, 59], and [81]. These approaches generate and update explicit architectural models of themselves for self-adaptation [71].

As a computational intelligence subfield, machine learning has been widely applied to provide self-adaptation capabilities. As a subfield of machine learning, RL refers to a set of trial-and-error methods by which an agent could learn how to make good decisions through interactions with an environment. The adaptive nature of RL makes it very appropriate for self-adaptation [9]. Cardellini et al. presented an Elastic and Distributed data stream processing Framework (EDF) to autonomously control elastic data stream processing applications. EDF elastically self-adapts at runtime to prevent resource wastage and match the workload. In EDF, several distributed self-adaptation policies were designed, including a model-free RL (Q-learning) solution and a model-based RL [52] solution. The framework utilizes different levels of available knowledge about system dynamics, where distributed agents learn the most valuable reconfiguration actions [20].

In addition, Zhao et al. proposed a framework for self-adaptation through combining an offline learning phase to create adaptation rules for goal settings and an online adaptation phase to make adaptation decisions using the generated rules. At these two phases, there are two key self-adaptation capabilities of the framework: (1) at the offline phase, the framework automatically learns adaptation rules from different goal settings; (2) at the online phase, the framework automatically evolves adaptation rules from the environment and user goals [87]. Wan et al. presented another self-adaptation framework to handle the complexities of software changes using a rule-based RL self-adaptation planning method [78]. Also, Hrabia et al. presented an approach

that integrates RL into a hybrid decision-making and planning framework for online learning from beneficial behaviors' experiences of robots depending on the environment. In the framework, RL is used to improve self-adaptation and to deal with dynamically changed target conditions [42].

For managing multiple containers deployed across multiple host nodes (machines), Rossi et al. developed Elastic Docker Swarm (EDS) extending a container management tool *Docker Swarm* with self-adaptation capabilities. EDS is based on (model-based) RL to adapt the deployment of container-based applications at runtime in a decentralized manner. For the sake of comparison, the authors also designed approaches based on Q-learning and Dyna-Q [73] algorithms [68]. Fabiana Rossi [67] also proposed decentralized policies based on RL to adapt the application deployment, in terms of container migrations and elasticity. As a first step, the author designed and evaluated Q-learning, Dyna-Q, and model-based RL solutions to exploit system dynamics.

In sum, these prior approaches generally focus on bringing self-adaptation capabilities to various kinds of software applications through RL, adapting the behaviors of the software itself to the runtime environments and/or system-level resources. In comparison, SEADS also leverages the RL methodology to achieve adaptation capabilities, but it aims to adapt a foundational program analysis of software applications (as opposed to adapting the software itself) to better cost-effectiveness tradeoffs while targeting continuously running distributed software in particular.

9 CONCLUSION AND FUTURE WORK

We have presented SEADS, a distributed, online, scalable, and cost-effective dynamic dependence analysis framework for common continuously running distributed systems with concurrent and decoupled processes. SEADS itself is distributed, with nodes (each analyzing one of the distributed processes of the SUA) to compute dependencies in parallel using the SUA's distributed computing resources. SEADS is online to avoid disk I/O and storage costs of dynamic data tracing. To achieve practical scalability and cost-effectiveness, SEADS automatically and continually adjusts its analysis configurations during the SUA execution, according to previous configurations, corresponding costs, and the user-defined budget, using a reinforcement learning (Q-learning) strategy.

Our evaluation targeted eight real-world Java distributed systems with continuous executions. The results revealed that SEADS offers a scalable, cost-effective, online, and continuous dynamic dependence querying service, with practical runtime overheads (99% slowdown), acceptable response time (65 seconds), and almost-negligible storage costs (only 88 MB) in an average case. We also demonstrated that SEADS has scalability and cost-effectiveness advantages over our baseline, the online version of a state-of-the-art dynamic dependence analysis for distributed program but using a fixed (highest-precision) analysis configuration: (1) SEADS was 3× faster than the baseline to respond to the user (65 seconds vs. 197 seconds); (2) SEADS was 3.3× as efficient as the baseline in terms of the runtime slowdown caused by the online analysis; (3) SEADS was more cost-effective than the baseline that only attained 44% and 32% of SEADS's cost-effectiveness with respect to average response time and runtime slowdown, respectively; and (4) SEADS scaled to enterprise-scale SUAs, such as Voldemort, while the baseline could not.

Through SEADS, we demonstrated a novel methodology for achieving a scalable and cost-effective online dynamic dependence analysis, as a fundamental dynamic analysis that has numerous applications in various domains (e.g., debugging, testing, security defending, and performance-tuning distributed systems). Thus, one immediate next step is to develop practical client analyses and tools for these application problems for real-world, enterprise-scale distributed systems. Another future direction is to develop an optimal self-adaptive dynamic dependence analysis framework that provides optimal cost-effectiveness tradeoffs at arbitrary querying time.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful and constructive comments, which helped greatly improve our manuscript revision process.

REFERENCES

[1] David H. Ackley and Michael L. Littman. 1990. Generalization and scaling in reinforcement learning. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 550–557.
[2] Apache. 2015. ZooKeeper. Retrieved from https://zookeeper.apache.org/.
[3] Apache. 2017. Voldemort. Retrieved from https://github.com/voldemort.
[4] Apache. 2017. The Voldemort Project. Retrieved from https://www.project-voldemort.com/voldemort/.
[5] Apache. 2018. Thrift. Retrieved from https://thrift.apache.org/.
[6] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2005. Efficient and precise dynamic impact analysis using execute-after sequence. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*. 432–441.
[7] Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Vol. 10. 1–14.
[8] Bryan Auslander, Stephen Lee-Urban, Chad Hogg, and Héctor Munoz-Avila. 2008. Recognizing the enemy: Combining reinforcement learning with strategy selection using case-based reasoning. In *Proceedings of the European Conference on Case-Based Reasoning*. 59–73.
[9] Erick de A. Barboza, Carmelo J. A. Bastos-Filho, Joaquim F. Martins-Filho, Uiara C. de Moura, and Juliano R. F. de Oliveira. 2013. Self-adaptive erbium-doped fiber amplifiers using machine learning. In *Proceedings of the SBMO/IEEE MTT-S International Microwave & Optoelectronics Conference (IMOC'13)*. 1–5.
[10] Soubhagya Sankar Barpanda and Durga Prasad Mohapatra. 2011. Dynamic slicing of distributed object-oriented programs. *IET Softw.* 5, 5 (2011), 425–433.
[11] E. N. Barron and H. Ishii. 1989. The Bellman equation for minimizing the maximum cost. *Nonlin. Anal. Theor. Meth. Applic.* 13, 9 (1989), 1067–1090.
[12] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. 2006. Theoretical foundations of dynamic program slicing. *Theoret. Comput. Sci.* 360, 1–3 (2006), 23–41.
[13] Eric Bodden. 2018. Self-adaptive static analysis. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results*. 45–48.
[14] Haipeng Cai. 2018. Hybrid program dependence approximation for effective dynamic impact prediction. *IEEE Transa. Softw. Eng.* 44, 4 (2018), 334–364.
[15] Haipeng Cai and Xiaoqin Fu. 2020. *D^2ABS: A Framework for Dynamic Dependence Analysis of Distributed Programs*. Technical Report. Washington State University, Pullman, Pullman, WA.
[16] Haipeng Cai and Raul Santelices. 2014. Diver: Precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of the International Conference on Automated Software Engineering*. 343–348.
[17] Haipeng Cai and Raul Santelices. 2015. TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. 489–493.
[18] Haipeng Cai, Raul Santelices, and Douglas Thain. 2016. DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *ACM Trans. Softw. Eng. Methodol.* 25, 2(2016).
[19] Haipeng Cai and Douglas Thain. 2016. DistIA: A cost-effective dynamic impact analysis for distributed programs. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 344–355.
[20] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized self-adaptation for elastic data stream processing. *Fut. Gen. Comput. Syst.* 87 (2018), 171–185.
[21] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng. 2000. Bandera: Extracting finite-state models from java source code. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*. 439–448.
[22] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley Publishing Company.
[23] Manoj Debnath. 2018. Understanding asynchronous socket channels in Java. Retrieved from https://www.developer.com/java/data/understanding-asynchronous-socket-channels-in-java.html.
[24] I. Capuzzo Dolcetta and Hitoshi Ishii. 1984. Approximate solutions of the Bellman equation of deterministic control theory. *Appl. Math. Optim.* 11, 1 (1984), 161–181.
[25] Patrick Eugster and K. R. Jayaram. 2009. EventJava: An extension of Java for event correlation. In *Proceedings of the European Conference on Object-oriented Programming*. 570–594.
[26] Eyal Even-Dar and Yishay Mansour. 2003. Learning rates for q-learning. *J. Mach. Learn. Res.* 5, Dec. (2003), 1–25.

[27] Chun-guo Fei, Zheng-zhi Han, Hou-jun Tang, and Guo Wei. 2006. Self-adapt hybrid chaotic neural network and its application to TSP. *J. Syst. Simul.* 12 (2006).
[28] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. 2015. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Trans. Softw. Eng.* 42, 1 (2015), 75–99.
[29] Xiaoqin Fu and Haipeng Cai. 2019. A dynamic taint analyzer for distributed systems. In *Proceedings of the ACM International Symposium on the Foundations of Software Engineering*. 1115–1119.
[30] Xiaoqin Fu and Haipeng Cai. 2019. Measuring interprocess communications in distributed systems. In *Proceedings of the IEEE International Conference on Program Comprehension*. 323–334.
[31] Xiaoqin Fu and Haipeng Cai. 2020. Scaling application-level dynamic taint analysis to enterprise-scale distributed systems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*.
[32] Takayasu Fuchida, Kathy Thi Aung, and Atsushi Sakuragi. 2010. A study of Q-learning considering negative rewards. *Artif. Life Robot.* 15, 3 (2010), 351–354.
[33] Joshua Garcia, Daniel Popescu, Gholamreza Safi, William G. J. Halfond, and Nenad Medvidovic. 2013. Identifying message flow in distributed event-based systems. In *Proceedings of the ACM International Symposium on the Foundations of Software Engineering*. 367–377.
[34] David Garlan, S.-W. Cheng, A.-C. Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (2004), 46–54.
[35] Dennis Giffhorn and Christian Hammer. 2009. Precise slicing of concurrent programs. *Autom. Softw. Eng.* 16, 2 (2009), 197–234.
[36] GoogleCode. 2015. MultiChat. Retrieved from https://code.google.com/p/multithread-chat-server/.
[37] Diganta Goswami and Rajib Mall. 2000. Dynamic slicing of concurrent programs. In *Proceedings of the IEEE International Conference on High Performance Computing*. 15–26.
[38] Christian Hammer and Gregor Snelting. 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.* 8, 6 (2009), 399–422.
[39] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware program analysis via online abstraction coarsening. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*. 94–104.
[40] Susan Horwitz and Thomas Reps. 1992. The use of program dependence graphs in software engineering. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*. 392–411.
[41] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.* 12, 1 (1990), 26–60.
[42] Christopher-Eyk Hrabia, Patrick Marvin Lehmann, and Sahin Albayrak. 2019. Increasing self-adaptation in a hybrid decision-making and planning system with reinforcement learning. In *Proceedings of the Computer Software and Applications Conference*, Vol. 1. 469–478.
[43] Chi-fu Huang, Henri Pages, et al. 1992. Optimal consumption and portfolio policies with an infinite horizon: Existence and convergence. *Ann. Appl. Probab.* 2, 1 (1992), 36–64.
[44] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference*, Vol. 8. 9.
[45] Daniel Jackson and Martin Rinard. 2000. Software analysis: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 133–145.
[46] K. R. Jayaram and Patrick Eugster. 2011. Program analysis for event-based distributed systems. In *Proceedings of the International Conference on Distributed Event-based Systems*. 113–124.
[47] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Vol. 47. 121–132.
[48] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Inf. Proc. Lett.* 29, 3 (1988), 155–163.
[49] Jeff Kramer and Jeff Magee. 2007. Self-managed systems: An architectural challenge. In *Proceedings of the Conference on Future of Software Engineering*. 259–268.
[50] Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. 2019. Towards efficient, multi-language dynamic taint analysis. In *Proceedings of the ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 85–94.
[51] Jens Krinke. 2003. Context-sensitive slicing of concurrent programs. In *Proceedings of the ACM International Symposium on the Foundations of Software Engineering*. 178–187.
[52] Leonid Kuvayev and Richard S. Sutton. 1996. Model-based reinforcement learning with an approximate, learned model. In *Proceedings of the 9th Yale Workshop on Adaptive and Learning Systems*. 101–105.
[53] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot—A Java bytecode optimization framework. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop*. 1–11.

[54] Frank L. Lewis and Kyriakos G. Vamvoudakis. 2010. Reinforcement learning for partially observable dynamic processes: Adaptive dynamic programming using measured output data. *IEEE Trans. Syst., Man, Cybern., Part B (Cybern.)* 41, 1 (2010), 14–25.
[55] Michael L. Littman and Csaba Szepesvári. 1996. A generalized reinforcement-learning model: Convergence and applications. In *Proceedings of the International Conference on Machine Learning*, Vol. 96. 310–318.
[56] Wes Masri. 2015. Dependence Analysis. Retrieved from https://www.sciencedirect.com/topics/computer-science/dependence-analysis.
[57] Gero Mühl, Ludger Fiege, and Peter Pietzuch. 2006. *Distributed Event-based Systems*. Vol. 1. Springer Science & Business Media.
[58] Mangala Gowri Nanda and S. Ramesh. 2006. Interprocedural slicing of multithreaded programs with applications to Java. *ACM Trans. Prog. Lang. Syst.* 28, 6 (2006), 1088–1144.
[59] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 1998. Architecture-based runtime software evolution. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*. 177–186.
[60] Jing Peng and Ronald J. Williams. 1996. Incremental multi-step Q-learning. *Mach. Learn.* 22 (1996), 226–232.
[61] A. Podgurski and L. A. Clarke. 1990. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.* 16, 9 (1990), 965–979.
[62] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. 2012. Impact analysis for distributed event-based systems. In *Proceedings of the International Conference on Distributed Event-based Systems*. 241–251.
[63] Netty project. 2020. Netty: Home. Retrieved from https://netty.io/index.html.
[64] Martin L. Puterman. 2014. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.
[65] Venkatesh Prasad Ranganath and John Hatcliff. 2007. Slicing concurrent Java programs using Indus and Kaveri. *Int. J. Softw. Tools Technol. Transf.* 9, 5–6 (2007), 489–504.
[66] Seyed Mahdi Roostaiyan, Ehsan Imani, and Mahdieh Soleymani Baghshah. 2017. Multi-modal deep distance metric learning. *Intell. Data Anal.* 21, 6 (2017), 1351–1369.
[67] Fabiana Rossi. 2019. Self-management of containers deployment in decentralized environments. In *Proceedings of the IEEE World Congress on Services*, Vol. 2642. 315–318.
[68] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. 2019. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *Proceedings of the International Conference on Cloud Computing*. 329–338.
[69] Silvius Rus and Lawrence Rauchwerger. 2005. *Hybrid Dependence Analysis for Automatic Parallelization*. Technical Report, TR05-013. Texas A&M University.
[70] Barbara Ryder. 2003. Dimensions of precision in reference analysis of object-oriented programming languages. In *Compiler Construction*. Springer, Berlin, Heidelberg, 126–137.
[71] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. 2018. Control-theoretical software adaptation: A systematic literature review. *IEEE Trans. Softw. Eng.* 44, 8 (2018), 784–810.
[72] SourceForge. 2015. NioEcho. Retrieved from http://rox-xmlrpc.sourceforge.net/niotut/index.html# The code.
[73] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. The MIT Press.
[74] Simon Tragatschnig, Huy Tran, and Uwe Zdun. 2014. Impact analysis for event-based systems using change patterns. In *Proceedings of the ACM Symposium on Applied Computing*. 763–768.
[75] Bamberg University. 2015. Open Chord. Retrieved from http://sourceforge.net/projects/open-chord/.
[76] Vice. 2018. xSocket. Retrieved from http://xsocket.org/.
[77] Andre Violante. 2019. Towards Data Science, Simple Reinforcement Learning: Q-learning. Retrieved from https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56.
[78] Jiewen Wan, Qingshan Li, Lu Wang, Liu He, and Yvjie Li. 2017. A self-adaptation framework for dealing with the complexities of software changes. In *Proceedings of the International Conference on Software Engineering and Service Science*. 521–524.
[79] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Mach. Learn.* 8, 3–4 (1992), 279–292.
[80] Paweł Wawrzynski and Andrzej Pacut. 2004. Model-free off-policy reinforcement learning in continuous environment. In *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2. 1091–1096.
[81] Danny Weyns, Sam Malek, and Jesper Andersson. 2012. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.* 7, 1 (2012), 1–61.
[82] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. Experimentation in software engineering: An introduction. *The Kluwer International Series in Software Engineering*. Springer Science & Business Media.
[83] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A survey on software fault localization. *IEEE Trans. Softw. Eng.* 42, 8 (2016), 707–740.
[84] Michael Wunder, Michael L. Littman, and Monica Babes. 2010. Classes of multiagent Q-learning dynamics with epsilon-greedy exploration. In *Proceedings of the International Conference on Machine Learning*. 1167–1174.

[85] Tingting Yu. 2017. SimEvo: Testing evolving multi-process software systems. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 204–215.

[86] Xiangyu Zhang and Rajiv Gupta. 2004. Cost effective dynamic program slicing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Vol. 39. 94–106.

[87] Tianqi Zhao, Wei Zhang, Haiyan Zhao, and Zhi Jin. 2017. A reinforcement learning-based framework for the generation and evolution of adaptation rules. In *Proceedings of the International Conference on Autonomic Computing*. 103–112.

Received January 2020; revised July 2020; accepted July 2020