Two-Level Adaptation for Budget-Constrained Continuous Dynamic Dependence Analysis

XIAOQIN FU, Washington State University, USA
HAIPENG CAI*, University at Buffalo, USA

Dynamic dependence analysis is essential for software performance optimization, debugging, regression testing, and security analysis. In long-running and distributed systems, continuously performing this analysis while balancing cost and precision under strict time budgets is a persistent challenge. Current state-of-the-art (SOTA) approaches tackled this problem but suffer from inefficient budget utilization and/or imprecise dependence results, limiting their practicality in real-world software engineering workflows. We introduce GDist, a novel two-level adaptation framework that self-tunes analysis parameters to optimize cost-effectiveness for continuous dynamic dependence analysis. GDist integrates a decision-tree-based learning model, tailored to system executions, with domain knowledge about analysis precision levels, ensuring a more precise and budget-observing adaptation strategy. We evaluate GDist on 12 real-world distributed systems and in two key applications: regression test reduction and vulnerability detection. Results show that GDist improves budget utilization by 29% and precision by 18% over SOTA. Specifically, GDist reduces regression testing costs by 31% while maintaining test effectiveness and lowers the cost of identifying true-positive vulnerabilities by 36% in enterprise-scale systems. Its lightweight adaptation and the inherent interpretability of decision trees make it well-suited for scalable, cost-aware software maintenance and security analysis. These merits position GDist as a practical and adaptive solution for modern software engineering challenges.

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

Additional Key Words and Phrases: Dynamic analysis, budget constraint, distributed systems, security

ACM Reference Format:
Xiaoqin Fu and Haipeng Cai. 2026. Two-Level Adaptation for Budget-Constrained Continuous Dynamic Dependence Analysis. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE160 (July 2026), 24 pages. https://doi.org/10.1145/3808167

1 Introduction

Distributed systems, often running continuously to provide uninterrupted services, play crucial roles in modern society [15, 31]. Thus, assuring the quality of these systems is crucial, for which dynamic dependence analysis (**DDA**) is a fundamental technique [37, 42]. This technique empowers many client tasks in software engineering (e.g., program slicing [67], regression testing [22, 48], fault localization [71], vulnerability discovery [38]). It computes run-time dependencies among code entities (e.g., methods) from program executions. When its subjects (e.g., distributed programs) run continuously, the analysis needs to be continuous (i.e., run as long as the subject runs) also.

Generally, different DDAs offer different cost-effectiveness tradeoffs [14, 18, 20]. Also, in real-world software development, developers are practically subject to budget constraints (e.g., maximal analysis time cost affordable) when using an analysis tool [31, 63]. For example, security analyses

*Haipeng Cai is the corresponding author.

Authors' Contact Information: Xiaoqin Fu, Washington State University, Pullman, USA, xiaoqin.fu@wsu.edu; Haipeng Cai, University at Buffalo, Buffalo, USA, haipengc@buffalo.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE160
https://doi.org/10.1145/3808167

integrated into CI/CD pipelines must complete within tight deployment windows to avoid delaying critical software updates. In particular, when analyzing a distributed system under active attack, defenders must identify root causes or dependencies linked to vulnerabilities swiftly to mitigate threats before further exploitation occurs. Thus, a *more cost-effective analysis*, which offers more precise results (e.g., program dependencies) without exceeding the cost budget, *is more desirable.*

However, applying DDA in practice, especially finding a sweet spot in the *balance between analysis precision and cost within the given budget*, is challenging. Precise analyses tend to incur excessive overhead while more efficient approaches sacrifice accuracy and produce misleading or incomplete dependence information. This challenge is aggravated for DDA of distributed programs (**DDADP**) due to greater size and complexity of the (distributed) systems under analysis (***SUA****s*) than single-process programs and unbounded traces produced in the (continuous) SUA execution which require much higher costs to collect and analyze. As shown in our motivating study (§2), existing techniques can suffer 44% imprecision or fail to answer a dependence query in 12 hours, making them unsuitable for real-world deployment in time- or resource-sensitive settings. These issues underscore the urgent need for a more cost-effective form of DDA (especially DDADP).

Like other dynamic analyses, a DDADP often has different options (i.e., values of analysis parameters constituting its algorithmic configuration) each of which affects its precision and cost differently. Selecting and combining these options appropriately may lead to a solution to the above challenge. Yet due to the run-time dynamics of a distributed SUA and its continuously-running nature, as well as the differences among different SUAs, it is quite challenging to pick the "best" configuration *once for all.* If done manually, tuning the configuration (i.e., setting/adjusting the analysis parameters) hence cost-effectiveness tradeoff for different executions of the same or different SUAs (i.e., *Level-1* adaptation) is clearly too tedious hence impractical. On top of that, further tuning the configuration and tradeoff for the dynamics (e.g., varying run-time behaviors/complexity at different times) of every single SUA execution (i.e., *Level-2* adaptation) is even more challenging.

To address these challenges, we have developed GDist, a novel, Greedy approach to automatically balancing cost-effectiveness of continuous dynamic dependence analysis, applied to Distributed programs. GDist learns a simple decision-tree-based model, built via a greedy algorithm, to achieve adaptation. It starts with characterizing the dynamics of (i.e., uncertainty in) a given SUA execution and uses the results to construct the initial model (decision tree)—*to enable Level-1 adaptation.* Then, during the continuous analysis, GDist keeps updating the model upon new dynamics of the SUA execution observed on the fly, which enables its awareness of and handling the uncertainty. Meanwhile, it monitors the cost-effectiveness of the analysis to decide when to tune, while consulting the model to choose a potentially more optimal configuration—*to enable Level-2 adaptation.* Notably, a key advantage of GDist is that, during both the model initialization and updating, it *incorporates human a-priori knowledge* (heuristics) about the comparative precision levels of various analysis configurations. The decision tree underlying it also makes GDist *inherently interpretable*, versus the limited explainability of existing approaches (including the black-box use of RL).

We implemented GDist for Java and applied it to 12 real-world distributed system executions. Our results showed significant improvements of GDist over the state of the art (SOTA). On overall average, GDist achieved 17.9% increase in precision (91.6% vs 77.7%) and 28.5% in budget utilization rate (92.7% vs 72.1%), both statistically significant and large, while maintaining efficiency/scalability. We also demonstrated the benefits of GDist for regression test selection and vulnerability discovery as two of its applications. Compared to the SOTA, its higher precision led to 31% higher regression-test-reduction rates (53.6% vs 22.6%) and 36.1% less dependencies to inspect (46 vs 72) for confirming vulnerabilities in enterprise-scale distributed system executions.

GDist demonstrates a new adaptation methodology for continuous dynamic dependence analysis, evaluated for but not limited to distributed programs, hence benefiting a range of software analyses

that utilize dynamic dependencies. We considered a particular analysis configuration space for ease of presentation and evaluation purposes, although GDist's two-level adaptation approach is not limited to a specific configuration space. Our key contributions are:

- A novel, simple yet better (more effective and transparent) adaptation approach to self-balancing the cost and effectiveness of DDADP that achieves two-level adaptation (§3 and §4).
- An open-source implementation [8] of GDist that works with diverse, real-world distributed systems/executions of various architectures, domains, and sizes (§5.1).
- Extensive evaluations of GDist, with two representative application studies of DDADP (time-constraint-aware regression test reduction and vulnerability detection), that demonstrate its significant superiority over the SOTA peer technique (§5.3 and §6).

2 Background and Motivation

Dynamic dependence analysis tracks control and data dependencies among program statements during execution. It has been widely used in software engineering (e.g., debugging [39, 57], fault localization [19, 83], concurrency bug diagnosis [79], and security analysis [64]). Compared to static dependence analysis, it provides higher accuracy by capturing runtime behaviors. However, this benefit comes at a cost: it often incurs significant runtime overhead.

These costs are further amplified in distributed systems, where multiple processes or services execute concurrently across different nodes [36]. Dependence relations may span machine boundaries and rely on communication channels, such as message passing or remote procedure calls. Tracking such dependencies continuously requires monitoring diverse execution contexts, synchronizations, and data flows [41]. In particular, two major challenges arise: (1) *scalability*—fine-grained tracking quickly becomes infeasible as the number of nodes, messages, and interleavings grows, leading to prohibitive runtime slowdowns; and (2) *precision*—coarse-grained approximations reduce costs but risk conflating distinct causal chains, which can obscure root causes of failures or vulnerabilities. Thus, dynamic dependence analysis in distributed systems fundamentally faces a tension between maintaining precision and staying within an acceptable cost budget.

To illustrate this challenge, consider a distributed key–value store (e.g., Voldemort [3]) with multiple client-facing frontends, a replicated backend, and a load balancer. When a frontend intermittently crashes, developers may rely on dynamic dependence analysis to determine whether the failure originated from (i) a malformed client request, (ii) a faulty replication message, or (iii) a race condition involving the balancer. A high-precision configuration of the analysis (e.g., context-sensitive with fine-grained data-flow tracking) can reveal the exact causal chain across nodes. However, this incurs substantially greater runtime slowdown, exceeding the monitoring budget. Conversely, a coarse configuration (e.g., context-insensitive with reduced flow tracking) respects the cost budget but merges unrelated dependences, leaving the root cause ambiguous.

Table 1. Cost-effectiveness tradeoffs by existing DDADPs

SUA execution	DistIa		D^2Abs		Seads		
	Cost	Prec	Cost	Prec	Cost	Prec	BUR
NioEcho	3s	74%	228s	100%	214s	100%	91%
OpenChord	5s	71%	607s	100%	359s	77%	63%
Zookeeper	38s	61%	3,453s	100%	1,139s	66%	52%
Voldemort	190s	56%	>12h	n/a	860s	61%	35%

This example illustrates the central problem: *how can we adapt dynamic dependence analysis in distributed systems to balance precision and cost, while respecting a strict resource budget?* Most dynamic analyses offer a fixed cost-effectiveness tradeoff. For instance, Diver [17] and DiverOnline [20] together offer two tradeoff options for single-process software, similarly to DistIa [21] doing the same for distributed programs. DiaPro [20] and D^2Abs [16] offer slightly more (5 and 4, respectively) options. Yet, these prior works rely on the users to choose which option/version to use per SUA or SUA execution (*lacking Level-1 adaptation*), and the analysis has a fixed configuration hence only fixed cost-effectiveness balance once started

(*lacking Level-2 adaptation*). Relevant approaches [7, 51] seem viable, yet they only deal with a single objective hence are inapplicable to the scenario of two (*competing*) objectives (cost and effectiveness)—while applicable techniques (e.g., deep learning [45]) rely on sizable training data that may not be available in practical use scenarios.

Table 1 shows the average-case (time) cost and precision (*Prec*) per dependence query of two SOTA non-adaptive DDADPs against four SUA executions (as detailed in §5.1) each against its integration test. Here we use the *most-precise* setting with D^2Abs. Apparently, *the best balance varies with different SUA executions and budgets.* For example, for a 11-minute budget, the best-balance offeror for ZooKeeper would be DistIa, but D^2Abs for OpenChord and NioEcho. If the budget changes, the best tool may change too. It is clear that *manually tuning per budget value and SUA execution is not a practical option.* Note that for one system (Voldemort) D^2Abs provided no result (*n/a*) within 12 hours (thus we killed the job). These results serve as clear practical evidence of the severity of cost and imprecision issues with existing DDAs. Therefore, *cost-effectiveness balancing is not merely a matter of optimization; it can be a deal maker versus a deal breaker.*

Seads [40] has attempted to achieve *automatic* cost-effectiveness balancing through reinforcement learning (RL). While a major advance, Seads suffers from often low precision while with large wastes of the given budget—unable to effectively utilize the budget to push up the precision. As shown in Table 1 (last 3 columns), although offering more tradeoff options and better balance overall than the other two analyses, Seads had low precision especially for large systems while wasting the remaining budget—the budget utilization rate (*BUR*) is extremely low except for the smallest system NioEcho. One reason lies in its *lack of Level-1 adaptation* as it uses the RL model in a generic, black-box manner. More importantly, the sophisticated RL process takes long time to learn before starting to make good decisions (choosing better configurations).

Configuration tuning and hyperparameter optimization (including multi-objective ones) solve relevant problems, yet existing approaches (e.g., general-purpose methods/frameworks like SMAC [50] and Hyperopt [9], or NSGA-II [29]/III [28]) are not practically suitable for what we aim to address (e.g., due to their black-box/offline nature). We discuss/compare with them in detail later (§5.3.5).

3 Approach Overview

As shown in Figure 1, GDist takes five **inputs** from users: the distributed SUA D and its run-time inputs, a user budget B (response time upper-bound) that often varies across different executions of D or different SUAs, a dependence query Q (i.e., the name of a method invoked in D), and a few hyperparameters. Then, GDist works as follows via five modules (noted as numbered boxes):

① **Instrumentation**. This module aims to generate run-time information used by the dynamic analysis, by instrumenting D. The output is the instrumented version of D, noted as D'.

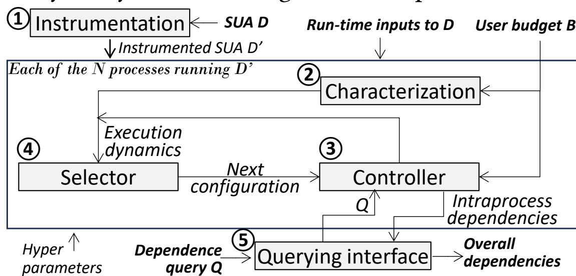

Fig. 1. Overview of GDist architecture and workflow.

② **Characterization**. This module learns (decision-tree) model parameters specifically optimized for D. GDist first starts D'. Then, it randomly samples a set of queries (i.e., methods) from D and computes their dependence sets, while recording the time costs and precision of its analysis for each query at each configuration. These recorded data are used to represent the execution dynamics (i.e., run-time characteristics) of D, which consists of (1) n configurations, (2) time cost of (i.e., that of the dependence analysis at) each configuration, and (3) precision of (i.e., that of the dependence analysis at) each configuration. Next, GDist iterates between the `Controller` and `Selector`.

③ **Controller**. Via this module, GDist decides when to self-tune (i.e., selecting the next analysis configuration) and when to compute/update dependencies. For instance, in mainstream DDADPs the dependence computation/updating is triggered when (1) a dependency query arrives, or (2) the computation time interval is longer than a user-defined threshold (e.g., 5 minutes) and the #method events exceeds another threshold (e.g., 1000). The adaptation is triggered when the dependence computation is about to take more time than the budget B or has finished within the budget. After the dependencies have been computed, they are sent to the `Querying interface`.
④ **Selector**. This module uses the decision-tree-based approach to select the next (potentially more optimal) configuration according to the budget B and the execution dynamics resulting from the `Characterization` or `Controller` module. The selected configuration is sent to `Controller` for the next round of dependence computation/updating.
⑤ **Querying interface**. Via this interface, users send a dependency query Q and receive corresponding results (i.e. dependencies of Q as a set of methods that have dynamic dependency on Q). After all (intraprocess) dependencies in the N processes are computed, GDist gathers them to compute the interprocess dependencies of Q. The overall (intraprocess and interprocess) dependencies of Q are the **outputs** of GDist as the response to the user query.

As justified earlier (§2), a DDADP should offer *both Level-1 and Level-2 adaptations* (as defined below). With the above design, GDist achieves that overarching goal as follows:

- **Level-1** (*Inter-execution adaptation*; ①→②→④): Before analyzing a new SUA execution, at this level, GDist performs lightweight characterization to collect its initial execution dynamics. Based on these characteristics, it builds the initial decision tree model to map characterization results to good initial decisions (configuration choices). This stage helps "personalize" the starting point for each SUA execution based on its analysis difficulty and cost profile.
- **Level-2** (Intra-execution adaptation; ③↔④): While analyzing the current SUA execution, at this level, GDist monitors the cost usage and precision gain in real-time. If the cost is approaching the user-specified budget or the analysis appears inefficient, it dynamically updates the configuration. Specifically, GDist switches to the "next best" configuration that promises the highest increase in precision per unit cost, while still keeping the remaining analysis under budget.

The intuition and justification behind this two-level adaptation strategy are: (1) SUAs exhibit diverse runtime behaviors that can be profiled and mapped to effective configurations; and (2) this mapping is captured via a budget-aware, interpretable decision tree trained on sampled queries, with precision further refined at runtime through intra-execution adaptation.

Specifically, Level-1 performs coarse-grained adaptation by partitioning the global configuration space using high-level contextual signals (e.g., SUA scale, budget tier). Level-2 then conducts fine-grained adaptation (configuration selection) within this reduced space to balance cost and precision. The two layers are synergistic: Level-1 ensures robust generalization by avoiding poor global modeling, while Level-2 provides local agility to react to runtime variations. This combination improves stability, responsiveness, and robustness across diverse executions.

We adopt a decision-tree–based model for its interpretability, low inference cost, and ability to encode human/domain knowledge — critical for continuous dynamic analysis. Complex alternatives (e.g., SMAC, Hyperopt) or black-box models (e.g., neural nets, ensembles) impose high latency and require costly runtime optimization. In contrast, decision trees enable fast, transparent/traceable configuration decisions without online learning, supporting both efficiency and explainability.

4 Technique Details

The adaptation GDist aims at is achieved via automatic configuration tuning. The configuration space represents the design space of the analysis algorithm. The key novelty and contribution of GDist lies in how to better tune (i.e., adjust/set parameters in) the configuration, not in how to

compute the dependencies. Thus, GDist's design is orthogonal to the design of this configuration space itself (i.e., which parameters should be included in it). Yet for ease of presentation, we assume, without loss of generality, that the dependence analysis can utilize some or all of various static/dynamic data (e.g., static dependencies, method events, and statement coverage). Accordingly, it has various analysis parameters controlling how to generate and use such program information, includes parameters regarding the collection and use of static program information (i.e., static parameters, such as whether static dependence graph is utilized and whether the static dependencies are computed in a context- and/or flow-sensitive manner) and those of dynamic information (i.e., dynamic parameters, such as whether all instances of a type of execution event of a method are distinguished or conflated into a single event). In fact, this configuration space has been pretty common among all the mainstream continuous dynamic dependence analyses [16, 21, 40].

4.1 Instrumentation

GDist inserts probes to the SUA (program D) for collecting the dynamic program information used by the dependence analysis under adaptation in GDist. For instance, method *entry* and *returned-into* as well as message-passing events are probed for, as these events are the major form of dynamic data used by mainstream DDADPs to infer happens-before relations hence dynamic dependencies among executed methods, both within and across processes. Another example is statement coverage, also commonly probed by DDAPS for statement-level dependence computation.

4.2 Characterization

GDist computes the SUA's initial *execution dynamics* by running the instrumented system against the given inputs. The goal is to initialize our decision-tree-based model for this specific SUA, hence realizing *Level-1 adaptation*. Leveraging a priori human knowledge that *the most precise dependence set $\hat{\mathcal{S}}$ is typically achieved at the highest-precision configuration (e.g., all static and dynamic parameters are turned on) for any query*, GDist first takes this configuration for each sample query. Then, the precision of the dependence set $\mathcal{S}$ of a sample query at any other configuration is computed as the size ratio of $\hat{\mathcal{S}}$ over $\mathcal{S}$. Intuitively, the more queries sampled, the better the execution dynamics can be captured hence the better the Level-1 adaptation be achieved, but also the higher the overhead. Thus, a *tradeoff* needs to be made here. The result of this characterization phase is initial execution dynamics, consisting of *the (mean over all sample queries) precision and cost for each configuration.*

These sample queries are randomly picked to efficiently characterize a system without exhaustively analyzing all methods, allowing us to obtain unbiased estimates of cost and precision for each configuration. Since only averaged configuration-level statistics—not query-specific data—are used, sampling variance has limited impact and is further corrected during Level-2 adaptation.

4.3 Controller

The controller makes decisions on when to compute dependencies and when to self-tune, per the triggering conditions described earlier (§3). During the continuous dependence analysis, GDist keeps measuring its cost and precision, and attempts to adjust its configuration when the cost is *about* to be too high (exceeding budget) or precision is too low (not fully utilizing budget)—i.e., cost-effectiveness should be improved, and uses the decision-tree model to select a new configuration for better cost-effectiveness tradeoff. GDist records the cost of each major step (e.g., computing static dependencies, gathering statement coverage) of dependence computation. This fine-grained cost measurement enables to trigger `Selector` well *before* a timeout actually occurs.

To deal with continuous executions—which is typical for all mainstream DDADPs, the dependence analysis adapted by GDist works as an *online* analysis, allowing users to request for dependencies of any query anytime during the SUA execution. Thus, whenever dependence computation is triggered at time t, GDist (re)computes the dependence set of any method that has executed at

least once (by time t) as a query, at the current configuration C. Then, the time cost associated with C is updated to the mean of per-query time costs to obtain the *new/updated execution dynamics*.

Since it focuses on *adaptation, not dependence analysis itself*, GDIST assumes standard dependence computation in mainstream DDADPs, which first computes intraprocess dependencies using static/dynamic data per a given configuration. For example, if the parameters are set to use static dependencies and method events, the analysis builds a static dependence graph of the SUA and then computes dynamic dependencies using different dependence propagation rules based on method event types (i.e., entry, returned-into) and static dependence categories (e.g., method call parameter passing, heap variable def-use) [20]. If statement coverage is additionally set to be utilized, the graph is pruned per the covered statements [16]. If static dependencies are opted out, dynamic dependencies are computed using method events only [21].

4.4 Selector

A key novelty of GDIST is that it incorporates a-priori human knowledge, meaning that GDIST's algorithms are designed per the a-priori knowledge about the relative/comparative precision level of its analysis at each configuration and that the analysis is generally more precise under some configurations than under others. For example, a flow-sensitive analysis is more precise than a flow-insensitive one for building the static dependence graph and the dynamic dependencies are less precise when computed using method-execution events only than also using statement coverage.

4.4.1 Overview. Intuitively, there are configurations at which the (dependence) analysis may get *potentially* greater cost-effectiveness than at others. We refer to such configurations as *signature configurations* or simply signatures. We can use these signatures to form a hierarchy (i.e., tree) of configurations from which the next ones may be predicted by searching through the hierarchy. With this intuition, the `Selector` consists of two components as depicted in Figure 2. When triggered (§4.3), the `Selector` starts with *Signature Identification* using the most recent execution dynamics of the SUA and signature count m (a user customizable setting). Then, *Configuration Prediction* determines the next configuration per the same execution dynamics, budget B, and signatures identified.

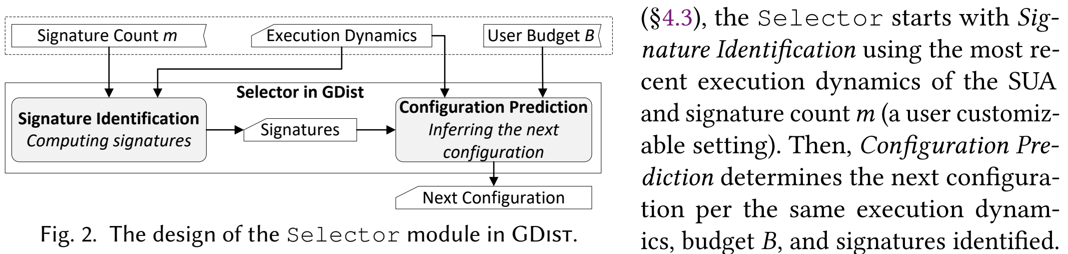

Fig. 2. The design of the `Selector` module in GDIST.

This process iterates, after it is initiated, when these signatures are identified according to the result of (i.e., the initial execution dynamics obtained from) `Characterization`. We illustrate each of these two steps with concrete examples to help walk through the full `Selector` pipeline.

4.4.2 Signature Identification. Without loss of generality, we consider n configurations in the configuration space of the dependence analysis adapted by GDIST. The *Signature Identification* component takes the precision and time cost associated with each of these configurations along with the signature count m as its inputs, and it outputs m signatures. Given this purpose and the definition of signatures, it is generally recommended that $m \in (2, n/2)$. The rationale is that if m is too small, the prediction would be excessively imprecise, while if m is too large, the model would be too complex hence likely overfitted (to the initial execution dynamics).

Specifically for the hybrid analysis addressed by GDIST, we set $m \le min(n_s, n_d)$, where n_s and n_d is the number of purely static and purely dynamic configurations (i.e., those with only static or dynamic parameters turned on), respectively. The rationale is that this allows us to obtain a good tradeoff between the precision of configuration prediction and the risk of model overfitting.

Algorithm 1 shows the pseudo code of *Signature Identification*. GDIST first picks the configuration with which the analysis precision is the highest (and the time cost is often also, but not necessarily,

Algorithm 1: Signature Identification

Input : Execution dynamics (including n configurations $C_1, C_2, ..., C_n$, the time cost $T(C_i)$ and precision $P(C_i)$ of each (C_i) $(i \in [1, n])$) and the signature count m (part of the hyperparameters)
Output : m signatures $S_1, S_2, ..., S_m$

```
1  configSet = {C_1, C_2, ..., C_n}        ▷ initially, all configurations are considered candidate signatures
2  SG_1 = argmax_{C_j} P(C_j), C_j ∈ configSet        ▷ pick the first signature (i.e., highest-precision configuration
3  remove SG_1 from configSet        ▷ remove the one already picked, so as to consider remaining configurations for signatures
4  for i=2 to m do
5      SG_c1 = argmax_{C_j} P(C_j), C_j ∈ configSet        ▷ consider two candidates: the first is one for the (next) highest precision
6      SG_c2 = argmin_{C_k} T(C_k), C_k ∈ configSet        ▷ the second candidate is the configuration for the lowest cost
7      P_fallRatio1 = (P(SG_{i-1}) − P(SG_c1))/P(SG_{i-1})        ▷ precision falling ratio of the first candidate
8      P_fallRatio2 = (P(SG_{i-1}) − P(SG_c2))/P(SG_{i-1})        ▷ precision falling ratio of the second candidate
9      T_fallRatio1 = (T(SG_{i-1}) − T(SG_c1))/T(SG_{i-1})        ▷ time cost falling ratio of the first candidate
10     T_fallRatio2 = (T(SG_{i-1}) − T(SG_c2))/T(SG_{i-1})        ▷ time cost falling ratio of the second candidate
11     Diff_1 = T_fallRatio1 − P_fallRatio1        ▷ time cost versus precision falling ratio difference of the first candidate
12     Diff_2 = T_fallRatio2 − P_fallRatio2        ▷ time cost versus precision falling ratio difference of the second candidate
13     SG_i = Diff_1 >= Diff_2 ? SG_c1 : SG_c2        ▷ pick the candidate with greater cost-effectiveness as a new signature
14     remove SG_i from configSet        ▷ to consider remaining configurations for signatures
15 return {SG_i | i ∈ [1, m]} sorted non-descendingly by their time costs
```

the highest) among all the n configurations, as the first signature (lines 1–2). Then, the next m-1 signatures are identified one by one (loop of 4–14), each from two candidates: the one associated with the (next) highest precision (line 5) and the one with the (next) lowest time cost (line 6).

To pick a signature between the two candidates, GDist computes the precision falling ratio ($P_{fallRatio1}$, lines 7–8) and time cost falling ratio ($T_{fallRatio}$, lines 9–10) of each candidate over the immediately previous signature. Next, the differences between these two ratios are calculated per candidate (lines 11–12). The candidate that has the larger difference is selected as a signature (line 13). The rationale is intuitive: the larger difference indicates a greater time cost decrease (i.e., falling) associated with a lesser precision decrease (hence greater resulting cost-effectiveness). Lastly, GDist sorts the m signatures by their time costs, as the outputs of Algorithm 1.

Table 2. Example execution dynamics

Configuration	Time Cost (ms)	Precision
C_1 101010	$T(C_1)$ 460,476	$P(C_1)$ 90.49%
C_2 101101	$T(C_2)$ 673,958	$P(C_2)$ 93.26%
C_3 101111	$T(C_3)$ 777,716	$P(C_3)$ 98.98%
C_4 111111	$T(C_4)$ 993,012	$P(C_4)$ 100.0%

For example, consider the xSocket system [74] (see §5.1) running an integration test where one server and two clients are launched, and each client sends a different text message to the server. Table 2 shows the *execution dynamics* just before Algorithm 1 is triggered, with configurations encoded as binary strings (per §5.1). Assume $m = 2$. GDist first selects C_4 as the first signature SG_1 due to its highest precision. It then evaluates two candidates for SG_2: C_3 (highest remaining precision) and C_1 (lowest cost). For C_3, the relative precision drop from C_4 is $P_{fallingratio1}$=(100-98.98)%/100%=1.02%, and the relative cost reduction is $T_{fallingratio1}$=(993,012-777,716)/993,012= 21.68%, yielding a differential benefit $Diff_1$=(21.68-1.02)%=20.66%. For C_1, the differential benefit is similarly computed as $Diff_2$ = 44.12%. Since C_1 offers better overall cost-effectiveness (greater time reduction with less precision loss), it is selected as SG_2. With $m = 2$, the process returns the final signature set ordered by non-descending cost: $\{C_1, C_4\}$.

4.4.3 Configuration Prediction. After the signatures are identified, the `Selector` predicts the next configuration, as outlined in Algorithm 2. It takes the same execution dynamics (i.e., $T(C_i)$, $P(C_i)$, $i \in [1, n]$) as in Algorithm 1, m signatures identified there, and budget B as inputs. With these inputs, GDist builds a decision tree DT (lines 3–18) and then traverses it to predict the next configuration at one of the tree leaves (lines 19–29). The DT has three types of nodes: *root*, *leaf*, and ϵ. The root node encodes the budget B; each leaf node represents a candidate configuration; and each ϵ node is a condition node used for branching during a search on DT.

The tree is built in a rule-based, cost-aware manner over a given budget B and signature configurations $S_1, S_2, \ldots, S_m$ ordered by increasing cost $T(S_i)$, with the root node representing B. If $B < T(S_1)$, a low-cost leaf node is created. If $B = T(S_i)$ for some i, an exact-match branch is created.

Algorithm 2: Configuration Prediction

```
Input  : signatures from Algorithm 1 (S_1, S_2, ..., S_m sorted non-descendingly by time cost); budget B; execution dynamics (i.e., configurations C_i, time
         cost T(C_i) and precision P(C_i) of the analysis at each configuration C_i (i ∈ [1, n]))
Output : the next configuration
1  let ϵ_t ∈(0,1) be the threshold used for branching at ϵ nodes (default value 0.5)
2  let DT=(V, E) be a decision tree: vertex set V and edge set E, both empty
   /* build the decision tree DT                                                          */
3  v_B = create a node that represents the user budget B                    ▷ the single root node of DT
4  add v_B to V as the root node
5  v_l = node for argmin_{C_j} T(C_j), j ∈ [1, n]                           ▷ pick the lowest-cost configuration
6  add v_l as a leaf node to V and an edge (v_B, v_l) with the label "B < T(S_1)" to E
7  for i=1 to m do
8      v_{S_i} = create a node that represents S_i
9      add v_{S_i} as a leaf node to V and an edge (v_B, v_{S_i}) with the label "B==T(S_i)" to E
10     if i < m then
11         v_ϵ = create a node that contains the label "ϵ=(B-T(S_i))/(T(S_{i+1})-T(S_i))"
12         add v_ϵ as an ϵ node to V and an edge (v_B, v_ϵ) with the label "B > T(S_i) && B < T(S_{i+1})" to E
13         v_lc = node for argmin_{C_j} T(C_j), T(C_j)>T(S_i) && T(C_j)<T(S_{i+1}), j ∈ [1, n]   ▷ picked for the lowest cost
14         add v_lc as a leaf node to V and an edge (v_ϵ, v_lc) with the label "ϵ < ϵ_t" to E
15         v_hp = node for argmax_{C_k} P(C_k), T(C_k)>T(S_i) && T(C_k)<T(S_{i+1}), k ∈ [1, n]  ▷ picked for the highest precision
16         add v_hp as a leaf node to V and an edge (v_ϵ, v_hp) with label "ϵ >= ϵ_t" to E
17 v_h = node for argmax_{C_k} P(C_k), k ∈ [1, n]                            ▷ pick the highest-precision configuration
18 add v_h as a leaf node to V and an edge (v_B, v_h) with the label "B > T(S_m)" to E
   /* traverse DT to predict the next configuration                                        */
19 if B < T(S_1) then
20     return the configuration on the leaf of an edge labeled "B < T(S_1)"   ▷ return the globally lowest-cost configuration
21 for i=1 to m do
22     if B == T(S_i) then
23         return S_i                                           ▷ return the signature configuration S_i immediately
24     if i < m && B > T(S_i) && B < T(S_{i+1}) then
25         compute ϵ as (B - T(S_i)) / (T(S_{i+1}) - T(S_i))
26         l_ϵ = (ϵ < ϵ_t) ? "ϵ < ϵ_t" : "ϵ >= ϵ_t"   ▷ prepare the ϵ condition label based on the corresponding condition
27         return the configuration on the leaf node of the edge with the label l_ϵ
28 if B > T(S_m) then
29     return the configuration on the leaf of the edge with the label "B > T(S_m)"   ▷ return the globally high-precision configuration
```

If $B \in (T(S_i), T(S_{i+1}))$ for some i, a fractional-ϵ branch is created, wherein the relative budget slack $\epsilon = \frac{B-T(S_i)}{T(S_{i+1})-T(S_i)}$ is compared to a threshold ϵ_t, guiding the choice between the lowest-cost configuration in that cost interval if $\epsilon < \epsilon_t$, or the highest-precision one if $\epsilon \geq \epsilon_t$. Finally, if $B > T(S_m)$, the highest-precision configuration across all candidates is used.

Specifically, GDist first adds the root node (lines 3–4). It then adds a node and an edge such that when the budget is lower than the cost with the lowest-cost signature, the configuration with the globally lowest cost will be selected (lines 5–6). Each of the edge labels indicates the condition under which the edge can be traversed through. For each (S_i) of the m signatures (loop of 7–16), one node and an edge from the root to the node with label "B==$T(S_i)$" are added (lines 8–9), to allow for straightforward prediction when the budget right hits the cost with one of signatures (S_i).

Also, for each signature S_i, $i \neq m$ (lines 10–16), GDist adds an ϵ node "ϵ=(B-$T(S_i)$)/($T(S_{i+1})$-$T(S_i)$)" and an edge connecting it to the root to accommodate a situation in which the budget falls in between the costs with two adjacent signatures S_i and S_{i+1}. Further, two leaf nodes and corresponding edges connecting them to the ϵ node are added to enable prediction in such situations. Specifically, among all configurations falling in between the two adjacent signatures in terms of their associated time costs, GDist identifies two candidate configurations to select from and uses the two leaf nodes to represent them: one (C_{lc}) with the lowest cost and the other (C_{hp}) with the highest precision. Accordingly, the two edges are labeled with conditions to help select the more optimal configuration between these two candidates, such that (1) when the ϵ value is closer to $T(S_i)$, C_{lc} will be selected, while (2) when the ϵ value is closer to $T(S_{i+1})$, C_{hp} will be selected. The rationale lies in the guiding principle of our methodology (achieving highest precision possible without exceeding the budget). A threshold $\epsilon_t \in (0, 1)$ is used for user preferences regarding which candidate is more likely to pick. We chose ϵ_t=0.5 by default for an equal preference. Lastly, GDist

adds a node and an edge such that when the budget is greater than the cost with the highest-cost signature, the configuration with the globally highest precision will be selected (lines 17–18).

Once the tree is built, the prediction is made by traversing it with a concrete value of B and $T(C_i)$, $i \in [1, n]$. If the budget is lower than the cost with the lowest-cost signature, the globally lowest-cost configuration is selected (lines 19–20). Otherwise, if the budget exactly matches the cost with a signature, the signature configuration is selected (lines 22–23); or if the budget falls in between the costs of any two adjacent signatures, the configuration is predicted based on the relevant ϵ's evaluation outcome against the threshold ϵ_t (lines 24–27). Lastly, the highest-precision configuration is selected if the budget is greater than the highest-cost signature (lines 28-29).

Table 3. Example decision conditions

Condition Case	Next Configuration (when)	
	X	Y
$B < T(S_1)$	000101	-
$B = T(S_1)$	101010 (S_1)	-
$T(S_1) < B < T(S_2)$	100010 ($\epsilon < 0.5$)	110010 ($\epsilon >= 0.5$)
$B = T(S_2)$	1011101 (S_2)	-
$T(S_2) < B < T(S_3)$	111010 ($\epsilon < 0.5$)	100100 ($\epsilon >= 0.5$)
$B = T(S_3)$	101111 (S_3)	-
$T(S_3) < B < T(S_4)$	100111 ($\epsilon < 0.5$)	110111 ($\epsilon >= 0.5$)
$B = T(S_4)$	111111 (S_4)	-
$B > T(S_4)$	111111 (S_4)	-

For example, we use the same xSocket scenario considered for illustrating Algorithm 1 (§4.4.2) to illustrate how GDist builds and traverses the decision tree for self-configuration as shown in Algorithm 2. Assume that all four configurations in Table 2 are identified as signatures. Based on the execution dynamics from either the initial characterization or the latest update, GDist constructs the tree by enumerating all possible budget-related decision conditions (as shown in Table 3).

The decision logic depends on how the user budget B compares to the time costs $T(S_i)$ of the signature configurations S_1 to S_4: If $B = T(S_i)$, then S_i is directly selected; If $B < T(S_1)$, the lowest-cost configuration (000101) is chosen; If $B > T(S_4)$, the highest-precision (also the highest-cost here) configuration (111111) is selected; If $T(S_i) < B < T(S_{i+1})$ for some $i \in [1, 3]$, the choice is made as per how the value ϵ compares to the threshold ϵ_t.

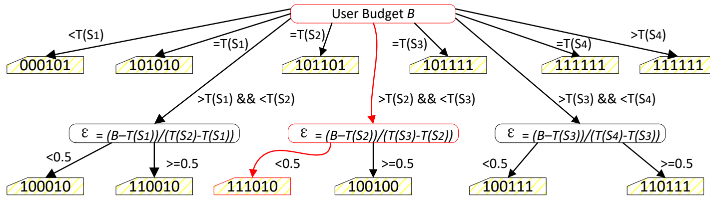

Fig. 3. The decision tree constructed for inferring the next configuration for xSocket with an example selection (as marked in red boxes/lines) process, illustrating the inherent interpretability and transparency of GDist.

This logic is encoded into the decision tree shown in Figure 3. For a concrete example, suppose the user budget is B=700,000 ms. From Table 2, we have $T(S_2)$=673,958 and $T(S_3)$=777,716, so we compute ϵ=(700,000-673,958)/(777,716-673,958)=0.25 < 0.5 (the default threshold ϵ_t). Thus, GDist follows the "ϵ<0.5" branch and selects configuration 111010 as the next-best choice. This process—highlighted in red in the figure—demonstrates GDist's transparent and explainable decision-making at runtime.

4.5 Querying Interface

GDist uses its querying client to interact with the user, via which the user sends a dependency query Q to the querying interface within each process of the instrumented SUA D'. When the intraprocess dependence computation in a process has finished, resulting dependencies are sent back to the client via the querying interface in that process. Once intraprocess analyses in all processes are completed, the client performs an interprocess analysis, which splices all the intraprocess dependence sets received based on the timestamps associated with the partially ordered method events while leveraging message-passing semantics to prune false-positive dynamic dependencies [40].

4.6 Usability

When using GDist, for a given system execution, users may send any queries (i.e., method names) to it and the **budget** B can be set (1) as a desired/acceptable value per the user's task schedule; or (2) based on the time costs corresponding to different levels of precision of the dependence analysis.

GDist has four **hyperparameters**. Two are used in `Controller`: (1) dependence computation interval threshold, set mainly based on system/execution complexity and how frequently users want the dependencies to be updated during the continuous analysis, and (2) threshold number of events, set mainly based on memory capacity and system execution complexity (i.e., the #events being generated). The (3) number of samples used in `Characterization` is set based on users' tradeoff between Level-1 tuning cost and effectiveness. The (4) signature count *m* used in `Selector` is set based on the size of the configuration space of the dependence analysis (see §4.4.2).

4.7 Theoretical Analysis

Our adaptation strategy is a greedy, rule-based policy that selects locally optimal configurations based on statistics from sampled space. Like many greedy heuristics, it offers predictable and fast behavior but does not guarantee convergence to a global optimum, nor exhibiting known bounds on regret or tradeoff optimality unless the cost–precision space exhibits special structure (e.g., sub-modularity/matroid properties). GDist does not assume/exploit such structures.

The optimality we target is to find, under a fixed cost budget, a configuration that achieves the highest attainable precision. This optimality is theoretically achievable when (i) cost and precision are monotonic across configurations, and (ii) the marginal precision gain has diminishing returns (i.e., the gain per unit cost decreases with cost). That is, we assumed that the configurations have a partial order by precision/cost, although we did not observe pathological counterexamples in our experiments. In edge cases (e.g., a higher-cost configuration yields same or even lower precision than a cheaper one), our system records the observed precision and makes decisions based on it accordingly (e.g., simply skipping the less cost-effective configuration).

4.8 Other Limitations

First, GDist includes a `Characterization` phase to characterize initial execution dynamics. Thus, GDist answers user queries *after* this phase completes. Users may not accept the delay.

Second, GDist's target scenarios are where users want maximal precision as long as the cost does not exceed given budgets. These scenarios are prevalent in many applications where high accuracy is critical to avoid false positives or missed threats, and where available resources (e.g., time) must be fully leveraged for the best outcomes. For example, in vulnerability detection, the goal is to identify true-positive vulnerabilities with minimal noise (e.g., false positives), and utilizing the entire budget ensures the analysis explores as many paths or scenarios as possible for comprehensive coverage. For another example, security or functional correctness checks in CI/CD pipelines must operate within a fixed budget to meet deployment deadlines, where maximizing precision ensures confidence in releases, while higher budget utilization improves coverage and reduces post-deployment risks. Yet there may be scenarios where gaining precision is unimportant (e.g., users just want a given/fixed level of precision; then, maximizing budget use would not make sense). That is where GDist misfits.

5 Evaluation

Our evaluation of GDist itself is guided by three research questions.

RQ1 *How cost-effective is GDist in terms of its precision and user budget utilization improvements?*
RQ2 *How efficient/scalable is GDist in terms of its time costs and run-time slowdowns?*
RQ3 *How significant are the cost-effectiveness and efficiency improvements achieved by GDist?*
RQ4 *How sensitive is GDist's performance to budget and hyperparameter settings?*

5.1 Experimental Setup

We implemented GDist as an open-source tool and applied it to eight real-world distributed Java systems, most of which are industrial systems, as shown in the first column (**Subject (version)**) of Table 4. Also listed are the numbers of non-blank non-comment Java source code lines (**#SLOC**),

the number of methods defined (**#Method**), and the types of run-time inputs (**#Tests**) for each subject SUA. MultiChat [46] is a chat app that broadcasts messages received from one client to others; NioEcho [73] is an echo service for any message sent by clients; OpenChord [77] is a peer-to-peer system using a distributed hash table to store key-value pairs; Zookeeper [4] is a distributed coordination service for synchronization and consistency as used by Apache Hadoop and Yahoo; Voldemort [3] is a distributed key-value store used by LinkedIn; and three libraries (i.e., Thrift [5], xSocket [74], and Netty [69]) are used for developing various distributed systems. These subjects were commonly used in prior/related works [11, 21, 35], and they cover various levels of scales as well various types of domains and architectures. We included the two small subjects to demonstrate GDist's scalability across SUA sizes and its merits for small and large systems.

Table 4. Experimental subjects and run-time inputs

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

In each integration test, we started 2–5 computing (e.g., server/client) nodes and performed various operations, to cover the main functionalities of each SUA. For the three libraries, we developed a sample application for each and exercised these subjects through the applications. In addition, load and system tests were used whenever available, as downloaded from respective subject repositories. Further details regarding execution setup and run-time inputs are given in our artifact package [8].

Baseline. To show the impact and merits of our novel contributions with GDist, without other peer tools available for comparison, we only selected Seads [40] as our baseline, a state-of-the-art (SOTA) dynamic dependence analyzer for distributed systems to the best of our knowledge. In fact, GDist is the most recent SOTA peer technique for GDist to compare with fairly—as per our motivating study (§2) and shown in Table 1, Seads is the best existing DDADP and the only one that offers flexible cost-effectiveness tradeoffs albeit only capable of adapting at Level-2.

Configuration space. As a practical tool, GDist needs to work on top of a concrete configuration space, which is also required for evaluation purposes. To enable a fair comparison with the SOTA baseline Seads [40] on their *adaptation capabilities*, we consider the same configuration space as in Seads. In this way, any performance difference between GDist and Seads will be attributed to the new adaptation approach, rather than being confounded by other factors.

Specifically, the configuration space includes three static parameters and three dynamic parameters. The static parameters include (1) *static Graph*, which specifies whether static dependencies (represented in a graph called *static graph*) are computed and used by the DDADP, and (2) *context Sensitivity* and (3) *flow Sensitivity*, which specify the analysis for constructing the static graph should be context- and flow-sensitive, respectively. The dynamic parameters include (4) *method Event* and (5) *statement Coverage*, specifying if method entry/returned-into (including message-passing) events and statement coverage, respectively, are collected and used, as well as (6) *method InstanceLevel* specifying if instances of each type of execution events of a method are used separately or all such instances are conflated as one event. Each of these parameters, in the above order, is encoded by a bit where 0 or 1 indicates the associated analysis option is disabled or enabled, respectively.

We performed all experiments on Ubuntu 22.04 machines with four 2.67GHz processors, 512GB DRAM, and 2TB HDD. The two `Controller` hyperparameters, which are shared by Seads, are set the same as in [40] (for fair comparisons). The `Characterization` hyperparameter (#samples) is set to 20 and the `Selector` hyperparameter (signature count m) is set to 8. By the principles laid out earlier (§4.6), the budget was set (in ms) as: Nioecho:2000, Multichat:1500, Openchord:60000, Thrift:50000, xSocket:40000, Voldemort:94000, ZooKeeper:50000, and Netty:90000, as defaults. If the budget is set too low, GDist can still handle it (i.e., choosing the cheapest configuration).

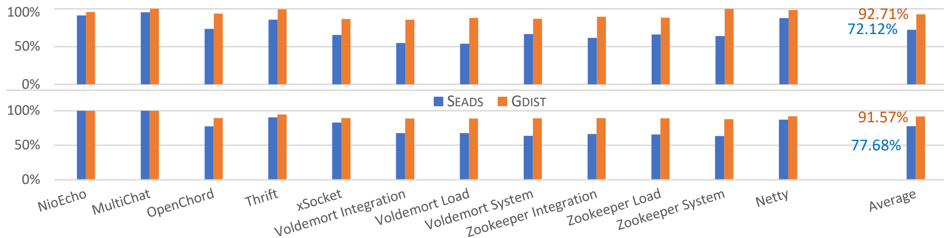

Fig. 4. Budget utilization rate (y axis, top) and precision (y axis, bottom) of GDist vs. Seads per subject execution (x axis).

5.2 Evaluation Methodology

In our targeted scenarios (§4.8), the user wants highest precision possible (provided that recall is not compromised), but the specific time cost is not relevant as long as it does not exceed the budget [13, 33, 40]. Yet a higher precision generally comes at a greater cost. Thus, *while a higher budge utilization rate (i.e., greater time expense within budget constraints) may not intuitively be more desirable from a user's perspective, it typically signifies the greater justifiable ability of an adaptive, budget-constrained dynamic analysis to push up precision*. Hence, the analysis should use as much of the budget to maximize precision, and GDist does *not aim to save budget*, but to maximally utilize (without exceeding) it. Accordingly, both the precision and budget utilization rate can be used to measure the performance of DDADP adaptation as the key cost-effectiveness metrics.

More specifically, the user budget utilization rate of an analysis with respect to a user query is the ratio of the analysis's actual query response time to the user's response time budget. Furthermore, we calculate budget utilization rate improvement of our technique over the baseline for a particular query as $(U_D\text{-}U_S)/U_S$, where U_D and U_S is the budget utilization rate of GDist and Seads, respectively. Similarly, the precision improvement is calculated as $(P_D\text{-}P_S)/P_S$, where P_D is the precision of GDist and P_S is that of Seads. In a continuous, online analysis, ground-truth dependencies are hard to obtain automatically. Thus, we compute precision as the dependence set size ratio relative to the high-precision configuration as described earlier (§4.2). Note that the dependence analysis precision is computed the same way as described in §4.2.

To measure the overall improvements, we computed the weighted average of per-query improvements across all the subject executions (with 20 randomly chosen queries per execution), where the weight for an execution is #methods in that execution as shown in the third column of Table 4. Similarly, we calculated the weighted average of precision improvements. Regarding efficiency, we compare to the baseline in terms of run-time slowdown as the main metric. We also compare both techniques in other relevant aspects (e.g., `Characterization` cost). Lastly, we measure the differences in cost-effectiveness and efficiency between the two techniques in terms of statistical significance (p values) and effect size. To account for the randomness (for sampling `Characterization` and testing queries) involved, we repeated our experiments 10 times for cross-validation and report the means (with *stdev* <5% of means).

5.3 Results and Analysis

5.3.1 RQ1: Cost-Effectiveness. Figure 4 contrasts GDist over Seads per SUA execution and the overall averages across the 12 executions. In both relevant metrics, GDist exhibited substantial advantages. Notably, the improvement was generally greater with larger-scale SUAs and more complex SUA executions, suggesting highly promising merits of our approach in practice. For smaller systems, Seads performed better (almost as well as GDist did) for two reasons. First, smaller systems have less variations in their (code) complexity, making missing *Level-1* adaptation a relatively lesser issue. Second, smaller systems also have less variations in their executions, making longer time to learn effective adaptation (due to the slower RL process) less detrimental. Overall, GDist was 17.9% more precise while utilizing user budget 28.5% more effectively than the baseline.

SEADS uses the generic Q-learning algorithm with the same reward function for any SUA and execution. This blackbox use of RL led to suboptimal self-tuning, despite the known promise of RL for enabling adaptation to changing environments in general.

A closer look into these results revealed that, for all the 240 queries, SEADS had more than 10% cases in which the actual query response time exceeded the budget. One reason was that, without knowledge about specific SUA executions, SEADS opportunistically always starts with the presumably most-precise configuration 111111 before gradually tuning down if necessary, and this configuration was constantly associated with the highest time cost among all configurations. In comparison, there were very few (<1% in total) such cases seen by GDIST because of its improved ability to predict more optimal analysis configuration due to its use of the a-priori human knowledge and the knowledge about (via the initial execution dynamics of) each specific SUA.

Imprecision. Dynamic analysis is often expected to have no false positives. Yet GDIST computes dependencies in an approximate manner and it is adaptive: it can be imprecise when switching to less-precise configurations for a lower cost.

Recall. When GDIST pushes up precision by better utilizing budget than SEADS, it could compromise recall. Without ground-truth dependence sets available, we manually examined the dependencies reported by SEADS but not by GDIST. We verified they were all false positives thus that GDIST had the same recall (100%[40]) as SEADS.

GDIST achieved substantially (28.5% and 17.9%, respectively) higher budget utilization rates and precision, hence greater cost-effectiveness, than the state of the art, with much fewer failures to observe the given budget constraint.

5.3.2 RQ2: Efficiency. To enable fair comparison, GDIST targets the same dependence analysis as SEADS as the optimization object. So, their differences in the cost for most parts (e.g., instrumentation, dependence computation) are expected to be negligible, as we confirmed in our experiments. Thus, we focus on the characterization cost and run-time overhead, which reflect the major design differences between the two techniques.

Table 5 lists this cost incurred by GDIST, which ranged from 6 seconds for the two smallest subjects (NioEcho and MultiChat) to 25 minutes for the largest subject (Netty). This implies that for an industry-scale system, the user would need to wait almost half an hour until GDIST can start serving cost-effectively, which may or may be not acceptable in practice.

Table 5. Characterization costs (in seconds)

SUA Execution	Time
NioEcho	5.68
MultiChat	5.60
Chord	17.63
Thrift	232.20
xSocket	993.01
ZooKeeper Integration	1,233.65
ZooKeeper Load	1,375.04
ZooKeeper System	1,302.87
Voldemort Integration	1,367.31
Voldemort Load	1,489.73
Voldemort System	1,457.26
Netty	1,497.17

However, there are several points to note. First, as discussed earlier (cf. §4.2), this cost can be traded off with the better self-tuning performance. Second, this is a one-off cost for a given SUA execution, during which the `Characterization` is done once for any number of queries. The longer the execution, the better this cost is paid off. Thus, given the cost-effectiveness merits of GDIST and the long-/continuous-running nature of real-world distributed systems, this cost can be readily paid off. Third, SEADS did not incur as much waiting time because the budget given was never large enough for it to switch to the highest-cost configuration; otherwise, SEADS users would have to wait similar amounts of time.

Figure 5 shows the run-time slowdown of GDIST versus SEADS against the 12 SUA executions. As expected, GDIST incurred higher overhead, because it immediately triggers the computation of dependence sets for all possible queries when the trigger conditions (§3) are met. In contrast, SEADS commonly delivers the results computed in the previous round of dependence computation when a user query arrives, thus it has less frequent and delayed re(computation) of dependencies, resulting in lower run-time slowdown. Yet as shown in the figure, the differences between the two techniques were generally small in this regard.

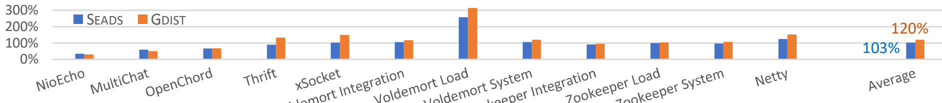

Fig. 5. The run-time slowdown (y axis) per subject execution (x axis).

The storage costs of GDIST are negligible and close to SEADS, ranging from 2MB to 200MB (mean 88MB) across the 12 subject SUA executions. These space costs were incurred by storing the instrumented version of each SUA and the static analysis data (i.e., the static dependence graph).

Like other efficiency numbers (e.g., dependence computation time), the numbers on characterization costs, run-time slowdown, and storage costs appeared to grow linearly or sub-linearly with growing subject sizes and execution complexity (in terms of #method events), suggesting that GDIST scaled gracefully to large systems.

Compared to the baseline, GDIST incurred 16.5% higher run-time slowdown and similarly negligible storage costs. It incurred extra costs for characterization, which can be readily paid off by its merits.

5.3.3 RQ3: Significance of Improvements. We did a set of paired Wilcoxon signed-rank tests [78] and computed Cliff's delta [24] on the budget utilization rate, precision, and run-time slowdown per SUA execution between the baseline and GDIST. We chose this non-parametric test as we cannot assume about (e.g., the normality of) the underlying data distribution. We applied paired statistical tests, where the results of GDIST and the baseline were paired on each SUA execution (10 repetitions per SUA). The p values (at 0.95 confidence level) and effect sizes are listed in Table 6.

Table 6. P-values and effect sizes of the differences in budget utilization rate, precision, and run-time slowdown between SEADS and GDIST

Measure	Utilization	Precision	Slowdown
p-value	0.0115	0.0327	0.1913
Effect size	-0.889	-0.656	-0.337

The p-value numbers revealed that the differences in budget utilization rate and precision between the two techniques are strongly statistically significant. Moreover, the absolute values of the effect sizes indicate that the differences were statistically large, while the signs of these numbers further show that GDIST (in the 2nd groups) had significantly greater precision and budget utilization rate.

On the other hand, the difference in run-time slowdown between the two techniques was not statically significant nor large. These statistical results revealed that our technique is significantly more cost-effective while being similarly efficient relative to the baseline.

The improvements of GDIST over the state of the art were statistically significant and large in cost-effectiveness, while their difference in run-time slowdown was insignificant.

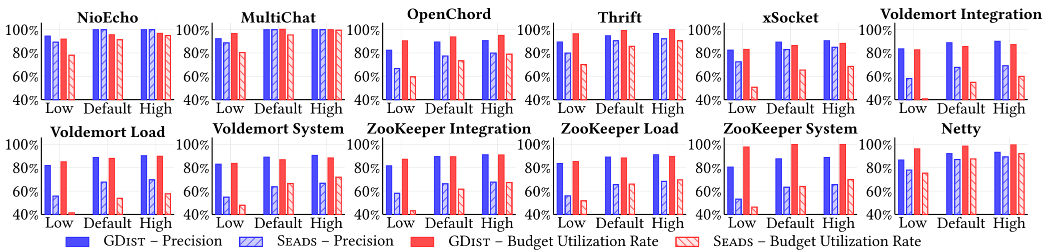

Fig. 6. Effectiveness (precision and budget utilization, y axis) of GDIST vs. SEADS at various budgets (x axis).

5.3.4 Budget/Parameter Sensitivity. To assess the robustness of GDIST, we compare it to the baseline at two other representative budget levels: Low (50%) and High (150%) of the Default (100%) budgets (§5.1), while keeping all other factors fixed. As Figure 6 shows, GDIST maintains relatively stable precision even with lower budgets. With higher budgets, precision generally increases, but not as much from Default to High compared to Low to Default. Similar variations are seen in budget

utilization rates, and on the baseline albeit with larger magnitudes. These reflect GDist's better adaptive budget usage. However, the Default budgets were used earlier to assess GDist within tighter (but not clearly insufficient) resource constraints, as a stronger test of its adaptability.

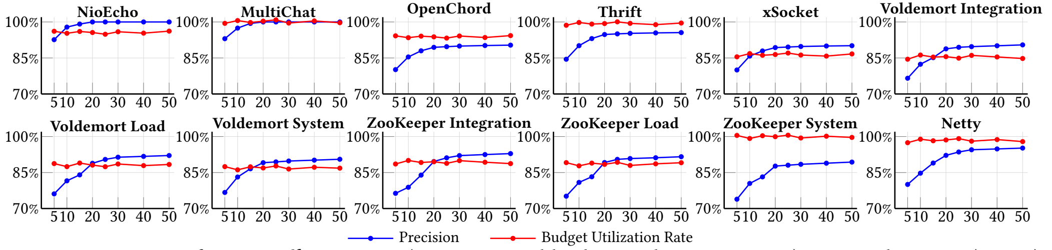

Fig. 7. Sensitivity of GDist effectiveness (precision and budget utilization, y axis) to sample count (x axis).

We also evaluate the impact of two hyperparameters on GDist: sample and signature counts (not applicable to the baseline). For sample count, we varied the number of samples (from 5 to 50) used in its offline characterization, with other factors fixed. As Figure 7 shows, the budget utilization rate is mostly very stable as expected, since the cost of characterization is not included in the query-time budget. Across all subjects, increasing sample counts led to monotonic or plateauing gains in precision, with early saturation in small subjects (e.g., NioEcho) and longer tails in large subjects (e.g., ZooKeeper, Netty), consistent with their configuration space size and diversity.

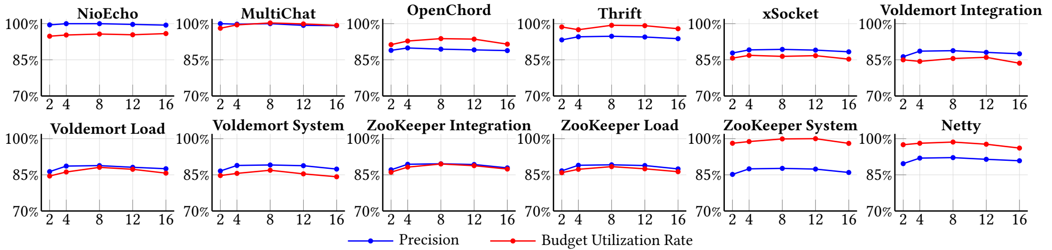

Fig. 8. Sensitivity of GDist effectiveness (precision and budget utilization, y axis) to signature count (x axis).

For signature count m, we varied it from 2 to 16 used in GDist's runtime adaptation, while fixing other parameters (at default values). As Figure 8 shows, as m increases, budget utilization rate rises and precision initially improves across all subjects. Small subjects reach precision saturation quickly (often at m=4 or 8) and show minimal budget utilization change. Medium subjects benefit moderately from increasing m, peaking at m=8. Large subjects experience more notable gains in both budget utilization and precision up to m=8∼12, after which precision often declined. The reasons are that (1) too-small m makes the decision tree less expressive, thus leading to overly generic configuration predictions, while (2) too-large m increases the number of configuration-space partitions and hence the model's complexity—which in turn increases the risk of overfitting to early signature observations. These results show GDist's tolerance to signature granularity variations.

The performance of GDist and its advantages over the baseline are reasonably stable across various budget levels and hyperparameter (sample count and signature count) settings.

5.3.5 Comparison to Other Approaches. Beyond Seads, general-purpose configuration tuning and hyperparameter optimization frameworks (e.g., SMAC [50], Hyperopt [9, 10], Optuna [2], or multi-objective optimizers like NSGA-II [29] and NSGA-III [28, 52]) seem to be potential baselines. Yet, they are not practically applicable to scenarios GDist addresses: continuously-running dynamic dependence analysis under budget constraints, as explained by the differences outlined in Table 7.

Table 7. Comparison of GDistwith existing configuration tuning/hyperparameter optimization frameworks.

Aspect	GDist (i.e., what we need)	Existing (e.g., SMAC/Hyperopt/NSGA)
Online tuning per execution	✓(adapts per SUA execution, at Level-2)	✗(assume static datasets/benchmarks)
Domain knowledge usage	✓(uses monotonicity/precision hierarchy)	✗(black-box, not amendable to the usage)
Budget-aware execution	✓(enforces runtime budget constraints)	✗(not designed for runtime constraints)
Explainability of decisions	✓(offers tree-based, traceable choices)	✗(no interpretable reasoning)
Dynamic feedback-driven	✓(adapts to observed cost/precision)	✗(offline, iterative, not runtime-reactive)

In theory, one could attempt to adapt these frameworks by (1) pre-profiling many executions to build an offline dataset, (2) mapping configurations to cost/precision ahead of time, and (3) using SMAC/Hyperopt/NSGA as static recommenders. However, such an approach faces three critical challenges: (1) infeasibility of exhaustive profiling: running dozens of full analyses to build a tuning dataset is prohibitively costly; (2) lack of per-execution adaptability: these frameworks cannot react to execution-specific behaviors or runtime variations; and (3) no budget-constraint enforcement: they optimize expected objective functions, but do not enforce hard runtime budgets.

By contrast, GDist was explicitly designed to handle these issues: it characterizes execution-specific dynamics (Level-1), uses this to guide per-execution adaptation (Level-2), and makes deterministic, explainable decisions towards the highest possible precision under a given budget. Also, GDist is not merely a configuration tuner—it co-designs when to tune (via characterization) and how (via tree-guided model), disparate from black-box optimizers.

To demonstrate that these frameworks are misfit for our problem setting, we use the cost and precision profiles gathered during GDist's characterization phase for ZooKeeper and Netty, and apply SMAC and Hyperopt (two popular optimization techniques) as static recommenders over this data. Each tool is given access to the full cost–precision mapping up front (an unrealistic best-case assumption for these two), and selects a configuration under each given cost budget, for 5 times.

As Table 8 shows, even under these assumptions favorable to the baselines, GDist consistently yields higher precision while satisfying budget constraints at low costs. In contrast, SMAC and Hyperopt either fail budget compliance (*BudgComp*) (by recommending over-budget configurations) or sacrifice precision, while incurring much higher costs. Also, unlike GDist, they cannot adapt to runtime-specific behaviors. The performance difference arises because GDist does not perform iterative black-box optimization at runtime—instead, it leverages pre-characterized data and a decision-tree-based model for instantaneous, self-adaptive decision-making.

Table 8. GDist vs. other approaches

System	Method	Precision	BudgComp	Time
ZooKeeper (Integration)	GDist	**0.89**	**5/5**	**<1s**
	SMAC	0.83	4/5	~12m
	Hyperopt	0.81	2/5	~9m
Netty	GDist	**0.92**	**5/5**	**<2s**
	SMAC	0.86	3/5	~13m
	Hyperopt	0.84	2/5	~10m

Note that this experiment is a *static*, *offline* simulation and does not reflect real-world applicability for these baselines, which would require exhaustive profiling and lack runtime adaptability. GDist, by contrast, collects data incrementally and adapts online with budget constraint enforcement.

6 Application Studies

We demonstrate two practical applications, in *software testing* in general and *security analysis* in particular, of adaptive/self-tuning dynamic dependence analysis through the use of DDADP.

6.1 Regression Test Selection

During software evolution, developers often perform regression testing to ensure that the code changes made do not break the original functionalities of the software. As the change is mostly incremental and small, it is unnecessary to always run the entire regression test suite. Instead, only the tests that cover the program entities (e.g., methods) changed or those impacted by the changes need to be executed. This test reduction can speed up testing hence is widely adopted in practice. As in other evolution/maintenance tasks, developers have a limited time budget for selecting the regression tests after making a set of code changes, especially when rapid application development (RAD) or an agile process is followed.

Given an SUA S, we consider its two versions S_1 and S_2 during the system evolution. Suppose the regression test suite for S_1 is T_1. Let S_M be the set of methods deleted or modified between S_1 and S_2. We run S_1 against T_1 and query GDIST (versus three other DDADPs as baselines) to obtain the (forward) dependence set (as the change-impact set) of each method in S_M. Then, we unionize all the per-method impact sets to obtain the total impact set IM_{T_1}. To regression-test S_2, we only select test cases in T_1 that cover one or more methods in IM_{T_1}.

In this study, we considered the two largest industrial systems used in §5 and retrieved for each two historical versions from its current GitHub repository: ZooKeeper [6] (S_1 corresponds to commit id *585...814* with 6 regression tests, and S_2 corresponds to *17b...bbe*) and Voldemort [60] (S_1 corresponds to commit id *51b...e0c* with 7 regression tests, and S_2 corresponds to *570...27b*). The time budget was generously set as 0.5 hour for each dependence query.

As shown in Figure 9, DISTIA achieved a 0% test reduction rate (due to its excessive imprecision hence identifying all methods covered by the original test suite as impacted). D^2ABS did not reduce any test either, as it did not finish hence return no result within the budget. Overall, SEADS was able to reduce 22.6% of the tests, much lower than GDIST's 53.6%, because the much higher precision of GDIST within the budget. We also validated that this test reduction did not reduce the overall test coverage of the impacted code.

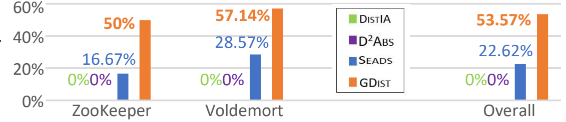

Fig. 9. Regression test reduction rates (y axis) enabled by GDIST vs. baselines, for two industrial systems and overall (x axis).

6.2 Vulnerability Discovery

Dynamic dependence analysis is a key enabler for a range of software security techniques/tools. For instance, recently proposed dynamic information flow analyses (DIFAs) [38, 56] which discovered high-profile security bugs are dynamic dependence analysis at their core. One of them [38] employed DDADP for successful vulnerability discovery in industry-scale distributed systems.

Given an SUA S, we consider its prior-to-release version S_R and suppose its test input set is T_R. We run S_R against T_R continuously for 10 hours to simulate the system's field deployment setting, in which S is supposed to provide uninterrupted service. The goal is to check the system execution against information flow vulnerabilities focusing on *exposure of sensitive information* (CWE-200 [61]). To that end, top 20 most sensitive security-relevant APIs in the Java SDK that can take user input are considered information sources as in [38]. We query GDIST (versus the other three DDADPs) to obtain the dependencies of each of the 30 sources. Then, we examine if these dependencies include any information sinks (critical APIs in the JDK that can leak data out of the system execution [38]) to confirm the presence/absence of CWE-200 vulnerabilities if any.

We consider ZooKeeper against its system test as in §5. Given the total monitoring time of 10 hours and the 20 dependence queries, as well as post-dependence-query inspection costs, the time budget for each dependence query is set as 20 minutes. From the dependence sets returned by GDIST, we confirmed 3 previously reported vulnerabilities (CVE-2014-0085 [1], CVE-2018-8012 [25], and CVE-2019-0201 [26]), and one new vulnerability (leaking the port number of the server) which has been verified and fixed by the developer. This discovery was achieved by inspecting no more than 50 methods on average per query (source). In contrast, 70+ and 200+ methods would have to be inspected per source if using SEADS and DISTIA, respectively. Again, D^2ABS ended up with timeout, providing no help with this task. This advantage of GDIST was again due to its greater cost-effectiveness as a result of its superior automatic cost-effective balancing capabilities.

GDIST's merits led to 31% higher regression-test reduction rates (in software testing) and 36.1% less dependencies inspected for vulnerability discovery (in software security) for enterprise-level distributed systems, as two examples of the practical applications of GDIST.

7 Discussion

We discuss how generalizable GDist is expected to be and how it may be extended, as long as the implications and validity threats of its results.

7.1 Generalizability and Extensibility

For ease and fairness of our evaluation, we only considered a relatively small (6-parameter) configuration space. Yet our adaptation methodology can readily work with larger spaces. For instance, for the dynamic dependence analysis in GDist, other configurations could include parameters (e.g., object and field sensitivity) of an underlying pointer analysis, or choices of additional program information (e.g., dynamic points-to sets, and if used, at object-level or object-instance-level). Parameters of application/client analyses can be part of the configurations as well (e.g., using message-passing events or not for interprocess dependence computation).

GDist is also by no means limited to binary parameters (e.g., varying depth of calling context or object/field sensitivity). In fact, the current GDist implementation already supports 20 different sizes of the event queue used for its online dependence analysis in the configuration space, making it work with 1,280 configurations—albeit for fair comparison with the SOTA approach we fixated this parameter at the same queue size used in Seads. For even larger space, GDist may scale by adjusting the sample and/or signature counts. Its design is not tied to a specific space or space size.

To extend GDist to other software analyses that utilize dynamic dependencies, changes might be needed to: (1) the configuration space, (2) the cost and precision metrics, and (3) possibly updating the thresholds and decision rules used in signature identification. The effort would be to determine the #purely static configurations and #purely dynamic ones, hence choosing the signature count in between, as described in §4.4. The key human knowledge needed is what is already incorporated in GDist's decision-tree modeling/algorithms, which will not change for other/new configurations. Users do not need to specify any relationships between the analysis configurations either: Algorithm 1 automatically identifies signature configurations, and then the relationships are sorted out during the decision-tree construction, also automatically.

7.2 Lessons Learned and Key Takeaways

Balancing cost and effectiveness is fundamental for program analysis, especially dynamic analysis, where tradeoffs must be tuned not just per SUA but per execution. A fixed tradeoff across executions reduces cost-effectiveness and may be infeasible—either failing to complete within budget or yielding overly imprecise results. Manual tuning is impractical, and existing tools (e.g., D^2Abs [16]) offer only limited pre-configured options, leaving much of the burden on users. Our results show that self-tuning not only alleviates this burden but also enhances practicality and effectiveness.

Our results also clearly demonstrate that a simpler (decision tree) approach can outperform a more sophisticated (RL) method. GDist not only achieves higher budget utilization and precision while better adhering to budget constraints, but also offers natural explainability—its decisions are transparent and interpretable, unlike RL's decision-making which is less explainable. RL explainability techniques (e.g., [47, 62]) exist, yet they are post-hoc and approximate, often computationally expensive, and not designed to enforce (e.g., runtime budget) constraints during decision-making.

Two key factors justify GDist's advantages. First, it enables both Level-1 and Level-2 tuning, whereas Seads only performs Level-2 tuning, applying the same generic RL model initialization across all SUAs and executions. This forces RL to implicitly learn Level-1 variations without explicit guidance, making the task overly difficult and often ineffective. Second, RL is not a magic bullet—its effectiveness depends on task-specific design. Unlike Seads, GDist is application-aware, leveraging `Characterization` to extract execution-specific insights, leading to better adaptation.

Second, the RL model in Seads struggled because the reward function did not generalize across all SUA executions, and finding a universally suitable function may be infeasible. Without proper

rewards, the model often mispredicted configurations, leading to ineffective self-tuning. In contrast, GDist deterministically selects configurations that maximize precision gains with minimal cost increase. This is achieved by combining (1) human knowledge of precision levels across configurations and (2) execution-specific insights from characterizing SUA dynamics.

7.3 Threats to Validity

An *internal validity* threat is that the GDist tool and our experimental scripts might be subject to implementation errors. One mitigating factor is that for our evaluation of Seads we used its original artifact package. To further minimize this threat, we validated the functional correctness of our tool and scripts via careful code review and manual inspections against the smallest subjects (Nioecho and MultiChat) and queries with small dependence sets.

The main threat to *external validity* is that our study subjects may not represent all real-world distributed systems and, as with any dynamic analysis, the run-time inputs used may not cover all their behaviors. We thus selected diverse systems in terms of architecture, code size, and application domains, and considered as diverse input types as available.

Given the dataset limitations, we do not claim that our results generalize broadly to all real-world distributed systems and their executions—the main threat to *conclusion validity*.

8 Related Work

Dependence analysis of distributed programs. Prior work [43, 68, 75] applies dependence analysis to impact prediction and debugging. GDist focuses on adaptive cost-effectiveness tuning rather than novel dependence analysis itself.

Tuning configurable systems. Configuration tuning tools like Unicorn [51] optimize system performance but handle only a single objective. In contrast, GDist simultaneously balances two competing objectives (cost and effectiveness) under budget constraints.

Self-adaptive software/systems. Unlike self-adaptive systems [23, 32, 34, 44, 54, 72, 81], which adapt *software itself* to runtime environments, GDist adapts *analysis of software* to dynamic executions, ensuring continuous cost-effectiveness within budget constraints.

Adaptive program analysis. Prior adaptive analyses [13, 49, 53, 58, 59, 66, 80] mainly focus on static analysis. Tools for optimizing analysis configurations [12, 27, 30, 55, 65, 70, 76, 82] adapt at a single level (e.g., per target program). GDist, however, employs a *two-level adaptation* to continuously optimize cost-effectiveness at runtime using a decision-tree-based model. Compared to Seads [40], GDist introduces a *decision-tree-based adaptation design* with novel *Characterization* and *Selector* modules. While incurring small runtime overhead, it significantly improves budget utilization and precision, hence superior cost-effectiveness overall.

9 Conclusion

We presented GDist, a novel, two-level self-adaptation approach for continuous dynamic analysis with respect to user-given budget constraints, and demonstrated its merits in the context of dependence analysis for distributed systems. Our results show that GDist has significant advantages over existing solutions. Based on decision-tree modeling, GDist is also inherently interpretable with transparent adaptation logic. Via two typical applications of dynamic dependence analysis, we also showed practical benefits of the improvement GDist brings in cost-effectiveness balancing.

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) through the grant CCF-2505223, as well as by the U.S. Office of Naval Research (ONR) through the grant N000142512252.

Data Availability

We have released all of our code and datasets to facilitate reproduction, replication, and reuse, as found in our artifact package, which will be made publicly available.

References

[1] 2014. Vulnerability Details : CVE-2014-0085. https://www.cvedetails.com/cve/CVE-2014-0085/.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2623–2631.

[3] Apache. 2015. Voldemort. https://github.com/voldemort.

[4] Apache. 2015. ZooKeeper. https://zookeeper.apache.org/.

[5] Apache. 2018. Thrift. https://thrift.apache.org/.

[6] Apache Software Foundation. 2022. apache/zookeeper. https://github.com/apache/zookeeper.

[7] Matthew Arnold, Martin Vechev, and Eran Yahav. 2008. QVM: An efficient runtime for detecting defects in deployed systems. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. 143–162.

[8] The authors. 2025. GDist Artifact Package. https://figshare.com/s/424c699ba40f3eb9827a.

[9] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. 2015. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery* 8, 1, Article 014008 (2015), 24 pages.

[10] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems (NIPS)*, Vol. 24. 2546–2554.

[11] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D Ernst. 2020. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–38.

[12] Dirk Beyer and Matthias Dangl. 2018. Strategy selection for software verification based on Boolean features: A simple but effective approach. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II 8*. Springer, 144–159.

[13] Eric Bodden. 2018. Self-adaptive static analysis. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 45–48.

[14] Haipeng Cai. 2018. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *IEEE Transactions on Software Engineering* 44, 4 (2018), 334–364.

[15] Haipeng Cai. 2025. A survey of program analysis for distributed software systems. *ACM Computing Surveys (CSUR)* 57, 12 (2025), 1–45.

[16] Haipeng Cai and Xiaoqin Fu. 2021. D^2ABS: A Framework for Dynamic Dependence Analysis of Distributed Programs. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4733–4761.

[17] Haipeng Cai and Raul Santelices. 2014. DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning. In *Proceedings of International Conference on Automated Software Engineering*. 343–348.

[18] Haipeng Cai and Raul Santelices. 2015. A Framework for Cost-effective Dependence-based Dynamic Impact Analysis. In *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering*. 231–240.

[19] Haipeng Cai, Raul Santelices, and Siyuan Jiang. 2016. Prioritizing Change Impacts via Semantic Dependence Quantification. *IEEE Transactions on Reliability* 65, 3 (2016), 1114–1132.

[20] Haipeng Cai, Raul Santelices, and Douglas Thain. 2016. DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *ACM Transactions on Software Engineering and Methodology* 25, 2, Article 18 (2016), 50 pages.

[21] Haipeng Cai and Douglas Thain. 2016. DistIA: A cost-effective dynamic impact analysis for distributed programs. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 344–355.

[22] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 809–820.

[23] Tao Chen, Ke Li, Rami Bahsoon, and Xin Yao. 2018. FEMOSAA: Feature-guided and knee-driven multi-objective optimization for self-adaptive software. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 2, Article 5 (2018), 50 pages.

[24] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[25] CVE. 2018. CVE-2018-8012. https://nvd.nist.gov/vuln/detail/CVE-2018-8012.

[26] CVE. 2019. CVE-2018-8012. https://nvd.nist.gov/vuln/detail/CVE-2019-0201.

[27] Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. 2017. Predicting rankings of software verification tools. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*. 23–26.
[28] Kalyanmoy Deb and Himanshu Jain. 2014. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014), 577–601.
[29] Kalyanmoy Deb, Agrawal Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.
[30] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. 2017. Empirical software metrics for benchmarking of verification tools. *Formal Methods in System Design* 50 (2017), 289–316.
[31] Chandan Dhal, Xiaoqin Fu, and Haipeng Cai. 2023. A control-theoretic approach to auto-tuning dynamic analysis for distributed services. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 330–331.
[32] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. 2010. FUSION: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 7–16.
[33] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, et al. 2017. Control strategies for self-adaptive software systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11, 4 (2017), 1–31.
[34] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. 2015. Supporting Self-Adaptation via Quantitative Verification and Sensitivity Analysis at Run Time. *IEEE Transactions on Software Engineering* 42, 1 (2015), 75–99.
[35] Xiaoqin Fu and Haipeng Cai. 2019. A dynamic taint analyzer for distributed systems. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1115–1119.
[36] Xiaoqin Fu and Haipeng Cai. 2019. Measuring interprocess communications in distributed systems. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 323–334.
[37] Xiaoqin Fu and Haipeng Cai. 2020. Scaling application-level dynamic taint analysis to enterprise-scale distributed systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 270–271.
[38] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 2093–2110.
[39] Xiaoqin Fu, Haipeng Cai, and Li Li. 2020. Dads: dynamic slicing continuously-running distributed programs with budget constraints. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1566–1570.
[40] 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 Transactions on Software Engineering and Methodology* 30, 1, Article 10 (2020), 45 pages.
[41] Xiaoqin Fu, Boxiang Lin, and Haipeng Cai. 2022. DistFax: A toolkit for measuring interprocess communications and quality of distributed systems. In *Proceedings of the ACM/IEEE International Conference on Software Engineering: Companion Proceedings*. 51–55.
[42] Xiaoqin Fu, Asif Zaman, and Haipeng Cai. 2025. DistMeasure: A framework for runtime characterization and quality assessment of distributed software via interprocess communications. *ACM Transactions on Software Engineering and Methodology* 34, 3, Article 74 (2025), 53 pages.
[43] Joshua Garcia, Daniel Popescu, Gholamreza Safi, William GJ Halfond, and Nenad Medvidovic. 2013. Identifying Message Flow in Distributed Event-Based Systems. In *Proceedings of ACM International Symposium on the Foundations of Software Engineering*. 367–377.
[44] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. On the impact of applying machine learning in the decision-making of self-adaptive systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 104–110.
[45] Chengyue Gong and Xingchao Liu. 2021. Bi-objective trade-off with dynamic barrier gradient descent. In *35th Conference on Neural Information Processing Systems (NeurIPS)*. 1–13.
[46] GoogleCode. 2015. MultiChat. https://code.google.com/p/multithread-chat-server/.
[47] Samuel Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. 2018. Visualizing and understanding atari agents. In *International Conference on Machine Learning*. PMLR, 1792–1801.
[48] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. 2019. Triggerflow: Regression testing by advanced execution path inspection. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 330–350.

[49] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware Program Analysis via Online Abstraction Coarsening. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 94–104.
[50] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.
[51] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2022. Unicorn: Reasoning about Configurable System Performance through the Lens of Causality. In *Proceedings of the European Conference on Computer Systems*. 199–217.
[52] Himanshu Jain and Kalyanmoy Deb. 2014. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part II: Handling constraints and extending to an adaptive approach. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014), 602–622.
[53] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.
[54] Cody Kinneer, David Garlan, and Claire Le Goues. 2021. Information reuse and stochastic search: Managing uncertainty in self-* systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 15, 1 (2021), 1–36.
[55] Ugur Koc, Austin Mordahl, Shiyi Wei, Jeffrey S Foster, and Adam A Porter. 2021. SATune: a study-driven auto-tuning approach for configurable software verification tools. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 330–342.
[56] Wen Li, Ming Jiang, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.
[57] Xiangyu Li and Alessandro Orso. 2020. More accurate dynamic slicing for better supporting software debugging. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 28–38.
[58] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.
[59] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 129–140.
[60] LinkedIn / Microsoft. 2022. voldemort/voldemort. https://github.com/voldemort/voldemort.
[61] MITRE/NVD. 2022. CWE-200: Exposure of Sensitive Information to an Unauthorized Actor . https://cwe.mitre.org/data/definitions/200.html/.
[62] Alexander Mott, Daniel Zoran, Mike Chrzanowski, Daan Wierstra, and Danilo Jimenez Rezende. 2019. Towards interpretable reinforcement learning using attention augmented agents. *Advances in Neural Information Processing Systems* 32 (2019), 12360–12369.
[63] Ning Nan and Donald E Harter. 2009. Impact of Budget and Schedule Pressure on Software Development Cycle Time and Effort. *IEEE Transactions on Software Engineering* 35, 5 (2009), 624–637.
[64] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*. 1–17.
[65] Eugene Nudelman, Kevin Leyton-Brown, Alex Devkar, Yoav Shoham, and Holger Hoos. 2004. SATzilla: An algorithm portfolio for SAT. *Solver description, SAT competition* 2004 (2004), 1–2.
[66] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. *ACM SIGPLAN Notices* 50, 10 (2015), 572–588.
[67] Santosh Pani, Shashank Mouli Satapathy, and GB Mund. 2013. Slicing of programs dynamically under distributed environment. In *Proceedings of International Conference on Advances in Computing*. Springer, 601–609.
[68] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. 2012. Impact Analysis for Distributed Event-Based Systems. In *Proceedings of the ACM International Conference on Distributed Event-Based Systems*. 241–251.
[69] Netty project. 2020. Netty: Home. https://netty.io/index.html.
[70] Cedric Richter and Heike Wehrheim. 2019. PeSCo: Predicting Sequential Combinations of Verifiers: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25*. Springer, 229–233.
[71] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight Fault Localization Using Multiple Coverage Types. In *Proceedings of IEEE/ACM International Conference on Software Engineering*. 56–66.
[72] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. 2017. Control-Theoretical Software Adaptation: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 44, 8 (2017), 784–810.
[73] SourceForge. 2015. NioEcho. http://rox-xmlrpc.sourceforge.net/niotut/index.html#Thecode.
[74] SourceForge. 2018. xSocket. http://xsocket.org/.
[75] Simon Tragatschnig, Huy Tran, and Uwe Zdun. 2014. Impact Analysis for Event-based Systems Using Change Patterns. In *Proceedings of Annual ACM Symposium on Applied Computing*. 763–768.

[76] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V Nori. 2014. Mux: algorithm selection for software model checkers. In *Proceedings of the Working Conference on Mining Software Repositories*. 132–141.

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

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

[79] Dasarath Weeratunge, Xiangyu Zhang, William N Sumner, and Suresh Jagannathan. 2010. Analyzing concurrency bugs using dual slicing. In *Proceedings of the International Symposium on Software Testing and Analysis*. 253–264.

[80] Shiyi Wei and Barbara G Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[81] Terence Wong, Markus Wagner, and Christoph Treude. 2022. Self-adaptive systems: A systematic literature review across categories and domains. *Information and Software Technology* 148 (2022), 106934.

[82] Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. 2012. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge* (2012), 57–58.

[83] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the International Conference on Software Engineering*. 272–281.