TAOPT: Tool-Agnostic Optimization of Parallelized Automated Mobile UI Testing

Dezhi Ran
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
dezhiran@pku.edu.cn

Zihe Song
University of Texas at Dallas
Richardson, USA
zihe.song@utdallas.edu

Wenyu Wang
University of Illinois at Urbana-Champaign
Champaign, USA
i@wenyu.io

Wei Yang
University of Texas at Dallas
Richardson, USA
wei.yang@utdallas.edu

Tao Xie
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
taoxie@pku.edu.cn

Abstract

The emergence of modern testing clouds, equipped with a vast array of real testing devices and high-fidelity emulators, has significantly increased the need for parallel automated mobile testing to optimally utilize the resources of testing clouds. Parallel testing aligns perfectly with the characteristic of rapid iteration cycles for mobile app development, where testing time is limited. While numerous tools have been proposed for optimizing the testing effectiveness on a single testing device, it remains an open problem to optimize the parallelization of automated mobile UI testing in terms of resource and time utilization. To optimize the parallelization of automated mobile UI testing, in this paper, we propose TAOPT, a fully automated, tool-agnostic approach, which improves the parallelization effectiveness of any given testing tool without modifying the tool's internal workflow. In particular, TAOPT conducts online analysis to infer loosely coupled UI subspaces in the App Under Test (AUT). TAOPT then manages access to these subspaces across various testing devices, guiding automated UI testing toward distinct subspaces on different devices without knowing the testing tool's internal workflow. We apply TAOPT on 18 highly popular mobile apps with three state-of-the-art automated UI testing tools for Android. Evaluation results show that TAOPT helps the tools reach comparable code coverage using 60% less testing duration and 62% less machine time than the baseline on average. In addition, TAOPT consistently enhances automated UI testing tools to detect 1.2 to 2.1 times more *unique* crashes given the same testing resources.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Dynamic graph algorithms**.

Keywords: UI Testing, Mobile Testing, Parallel Testing, Android, Parallelization, Graph Partition, Online Algorithm

ACM Reference Format:
Dezhi Ran, Zihe Song, Wenyu Wang, Wei Yang, and Tao Xie. 2025. TAOPT: Tool-Agnostic Optimization of Parallelized Automated Mobile UI Testing . In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3716282

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03…$15.00
https://doi.org/10.1145/3676641.3716282

1 Introduction

Given the importance of quality assurance for mobile apps, automated User Interface (UI) testing [1, 4, 14, 15, 21, 27, 36, 40–42, 47, 53, 60, 63, 64, 66, 69, 70, 78] has been developed to automatically generate and execute test inputs for Apps Under Test (AUTs). UI testing involves exploring an app's UI space, which encompasses all possible UI states, and the transitions between them. The UI transitions, triggered by interactions such as clicks or swipes, form a cohesive structure known as the UI transition graph, representing how users navigate through the app. While traditional efforts focus on enhancing the coverage of AUTs' UI spaces [69] and identifying unique crashes on a single device, the advent of testing clouds [31–33, 37, 48, 55], equipped with a wide range of real devices and emulators, underscores the necessity of parallel automated mobile testing (in short as parallelized testing) to leverage these resources. Yet, the task of effectively *parallelizing testing of an AUT across multiple devices* on testing clouds remains a significant challenge. In this context, a *parallelization strategy* akin to those used in distributed and parallel

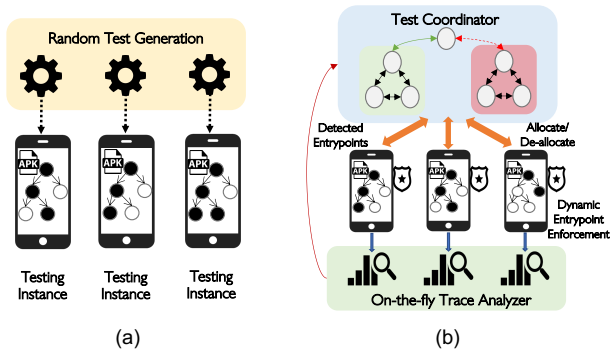

Figure 1. A conceptual illustration of parallelized testing with a testing cloud. Parallelized testing is not coordinated in (a) and coordinated by TaOPT in (b).

computing [6, 17, 34, 49, 61] is essential for coordinating automated UI testing across various *testing instances* within a testing cloud, as conceptually depicted in Figure 1, to align with the rapid evolution of AUTs [37, 39] and the limited testing timeframe [10]. Aiming to minimize *overlapping explorations*[1], task distribution in parallelization strategies for mobile UI testing, i.e., dividing the app's UI space into decoupled components or functionalities (e.g., login, browsing, or checkout functionalities), is significantly more complex than the static allocation of workloads in traditional distributed or parallel computing [6, 17, 49, 61].

The complexity of parallelization for mobile UI testing arises from the need for real-time dynamic task partitioning due to the dynamic nature of UI spaces in two major aspects. First, constructing UI space based on static app structure, such as partitioning according to static analysis [35] or the Activity component [10, 24], struggles with applicability and accuracy in most of apps on App Market. The scale of industrial apps [65] and widely adopted obfuscation techniques [18] make existing static analysis approaches fail to produce complete and accurate UI space structures. Second, the UI space requires not just the structure of UI transitions but also the probabilities associated with the transitions, and the probabilities can vary significantly across different testing tools, affecting the coupling among UI states. The inconsistent transition probabilities for testing tools necessitate their direct executions on the AUT to obtain UI space information for effective parallelization (as shown in our preliminary study in Section 3.3).

Despite the necessity of a dynamic parallelization strategy, it is challenging yet critical to design such an effective strategy. The main problem is that this strategy has to operate *with only local information* during exploration. To avoid redundant testing, dynamic parallelization needs to be initiated before the AUT's UI space has been fully constructed and the global information of the entire UI space has been known. However, if partitioning decisions rely on only local information, they may result in flawed parallelization and diminish the overall effectiveness of parallelized testing. We show examples in Section 2, and our preliminary study in Section 3.3 further confirms this point.

To address the preceding challenge, in this paper, we formulate the problem of achieving effective parallelization with local information as an online Min-Conductance Graph Partitioning problem [11, 44] (Section 4.1). We prove that accurate online graph partitioning within a reasonable time frame ($O(n^2 \log n)$) is feasible when the subgraphs are Globally Sparse and Locally Dense (GS-LD) [68]. The key insight is that exploring the graph will accumulate sufficient information of each subgraph of vertices before jumping out of the subgraph, making the local information about the subgraph sufficient for effective parallelization (Section 4.2).

Meanwhile, our preliminary study reveals that the AUT's UI spaces usually consist of what we define as **loosely coupled UI subspaces**, which are inherently GS-LD (Section 3.2). These subspaces are groups of UI states where the exploration by a UI testing tool rarely goes from one subspace to another. This phenomenon is attributed to mobile apps' design, which often contains multiple relatively independent functionalities [30, 38]. Figure 2 shows an example of an online shopping app. When exploring the account-logistics functionality, the design of mobile apps tends to confine the testing tool inside the account setting area, reducing transitions to other functionalities. By allocating each testing instance to explore one subspace, the overlapping explorations across testing instances are naturally minimized and the tool's behavior is preserved, achieving an effective parallelization strategy (as the workflow of our approach shown in Figure 4).

To make the preceding insights into a practical and generally applicable solution, we further propose TaOPT, a general and automated approach for flexibly optimizing parallelized testing of *any* given automated UI testing tool. Figure 4 presents the workflow of TaOPT. TaOPT monitors the UI transitions (i.e., UI hierarchy changes along with the triggering UI actions) during the working of the tool, and analyzes the UI transitions with an effective online algorithm (described in Section 5.2) to identify UI subspaces. Once the UI subspaces are identified, instead of modifying the testing tool or AUT, TaOPT controls the entrypoints to the UI subspaces with the Toller framework [64]. To satisfy different needs of balancing testing duration and testing resources, TaOPT supports two parallelization modes (duration-constrained mode and resource-constrained mode).

We evaluate the effectiveness of TaOPT by experimenting on three state-of-the-art/practice tools (Monkey [21],

[1]Overlapping explorations refer to the common scenarios (suggested by our preliminary study in Section 3.2) where separate testing instances redundantly traverse identical functionalities, decreasing testing effectiveness.

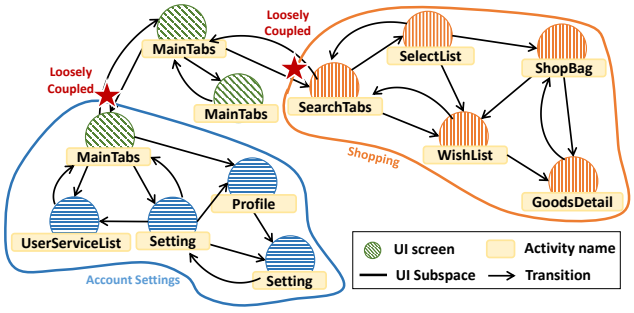

Figure 2. Motivating example: UI transition graph of online shopping app under automated UI exploration.

Ape [26], and WCTester [72, 78]) and 18 highly popular industrial apps used by recent work [64–66]. Evaluation results show that TAOPT consistently and substantially improves the parallelization effectiveness of all three testing tools, saving 65.9%, 50.1%, and 65.9% testing resources for Monkey, Ape, and WCTester to reach the same testing effectiveness, respectively. When given the same testing resources, TAOPT consistently improves the testing effectiveness of the three testing tools, covering 14.6% more methods and detecting 1.64× unique crashes on average. TAOPT helps reduce overlapping explorations (measured by abstracted UI screens [60]) by 68.9% on average and by 90.1% for the most advanced tool Ape, while preserving the three testing tools' behavior by covering more than 95% of the methods that the three testing tools can cover without TAOPT. These results indicate TAOPT's high value to generally improve the parallelization effectiveness.

In summary, this paper makes the following main contributions:

- A formulation of loosely coupled UI subspaces with theoretical guarantees to improve parallelized testing.
- An automated approach named TAOPT [51] to optimize the parallelization of any automated UI testing tool.
- Extensive evaluations of TAOPT, demonstrating generalization and effectiveness to improve parallelized testing.

2 Motivating Example

In this section, we motivate loosely coupled UI subspaces with a real-world example. Figure 2 shows a UI transition graph of a popular online shopping app collected from automated UI exploration. Each circle shows a UI screen that users or testing tools can see and interact with, while each edge indicates a potential transition. The app's main shopping functionalities, indicated by the top right solid-line area, enable users to browse products and make purchases. Meanwhile, users can manage their accounts on a separate set of screens, enclosed by the bottom left solid line. As shown by the graph, there is an apparent disconnection between the screens of these two sets of functionalities, suggesting that they are not highly related. It is desirable to parallelly and separately explore these two sets of functionalities for the efficiency of automated UI testing, achievable by partitioning the app UIs into two subspaces and avoiding overlapping explorations.

To partition the app UIs based on functionalities, a straightforward strategy is to leverage UI-related code units, such as Android activities [10]. This strategy will significantly limit the context of explorations in the given example: as shown in Figure 2, the main shopping functionalities are implemented using several different activities, such as *ShopBagActivity* and *GoodsDetailActivity*. If we explore these activities separately, we will not be able to cover core functionalities such as adding goods to the shopping bag and checking out. Consequently, this partitioning strategy will substantially limit the effectiveness of parallelized testing.

To tackle the preceding limitations, TAOPT performs online dynamic analysis on AUT UI transitions. Initially, the test generation tool explores the app freely on each device while TAOPT keeps monitoring the behavior. After the tool has explored certain functionalities (e.g., Shopping) non-trivially by frequently transitioning among the corresponding activities for sufficient amount of time on one device, TAOPT reports that a loosely coupled UI subspace is found. The tool will then no longer be able to enter these functionalities by UI actions on other devices; for example, the button leading to *SearchTabsActivity* (as indicated by the star) will be disabled on the main screen. Consequently, the tool can focus on testing the rest of functionalities (such as Account Settings) on other devices.

3 Preliminary Study

In this section, we conduct a preliminary study of the necessities and opportunities to improve parallelized testing with general parallelization strategies. First, given the fact that most automated UI testing tools [21, 26, 72, 78] are randomized, we first study whether automated UI testing can be effectively parallelized by the intrinsic randomness when using different random seeds. Second, we apply a simple activity-based parallelization strategy to a testing tool to study its general parallelization effectiveness for different exploration strategies. In summary, our study aims to answer the following research questions:

- **RQ1:** How effective is the intrinsic randomness of testing tools for parallelizing automated UI testing?
- **RQ2:** How general is the activity-based parallelization strategy for parallelizing an automated UI testing tool?

3.1 Study Setup

Baseline parallelized testing setting. We conduct parallelized testing by running $n = 5$ testing instances simultaneously for $l_p = 1$ hour in wall-clock time. For each testing instance, we simply let the test generation tool explore the

AUT for $l_p = 1$ hour without interruption, interference or coordination across testing instances. We use different random seeds for different testing instances to maximize the randomness across different testing instances with the hope of reducing overlapped exploration across different testing instances.

Parallelization effectiveness metrics. We use the overlap of methods covered and UI subspaces covered by different testing instances as the parallelization effectiveness metrics.

Measuring overlaps of covered methods. We use *Jaccard similarity* [54] to measure the overlap of method coverage achieved by two testing instances. Given two sets of covered methods A and B by two testing instances, the Jaccard similarity between A and B is $\text{Jaccard}(A, B) = \frac{\|A \cap B\|}{\|A \cup B\|}$. For parallelized testing consisting of n testing instances, we calculate the Average Jaccard Similarity (AJS) across all pairs of covered method set by two different testing instances with the following equation:

$$\text{AJS} = \frac{1}{C(n,2)} \sum_{A \neq B} \text{Jaccard}(A, B) \quad (1)$$

where C_n^2 is the number of combinations of n items taken 2 at a time, and A, B are two covered method sets achieved by two different testing instances.

Measuring overlaps of UI subspace exploration. In addition to measuring the coverage overlap, we also measure the number of overlapped UI subspaces covered by different testing instances. We apply an offline UI subspace partition algorithm (based on the algorithm introduced in Section 5.2) on the traces and count each UI subspace's occurrences in different testing instances. The algorithm segments regions conservatively, requiring both low inter-region transition probabilities and high internal cohesion before partitioning, ensuring that identified subspaces reflect genuine UI independence and supporting the study validity.

The setup of the testing platform, subject apps, and automated UI testing tools is the same as our evaluation setup (Section 6.1).

3.2 RQ1: Parallelization via Intrinsic Randomness

Prevalent and increasing overlap of covered methods. Figure 3 presents the overlapping explorations measured by overlaps of covered methods by different testing instances, from which we have two observations. First, all automated UI testing tools studied suffer from the overlapping explorations and the most advanced tool, Ape, suffers the most from overlapping explorations, indicating the general need for a parallelization strategy for any automated UI testing tool. Second, the intrinsic randomness of testing tools cannot effectively parallelize automated UI testing to address the overlapping explorations. While the covered methods of different testing instances at the beginning can be divergent due to the randomness of tools, the overlap quickly increases

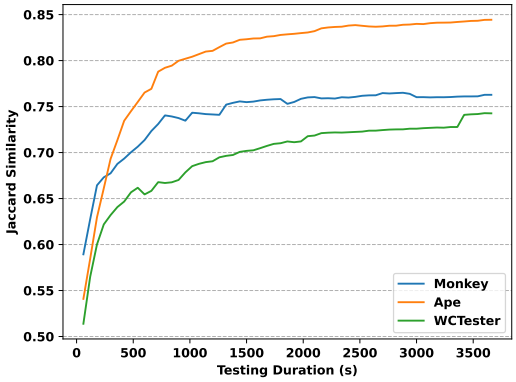

Figure 3. Overlaps of methods covered by different testing instances in non-coordinated (baseline) parallelized testing.

with parallelized testing. At the end of the parallelized testing, the Jaccard similarity of covered methods by Ape's testing instances reaches 0.84. Supposing that two testing instances cover 100 methods, respectively, 91 methods are covered by both testing instances, leaving the second testing instance increase only 9 newly covered methods and wasting 91% testing resources due to overlapping explorations.

Prevalent overlap of UI subspace exploration. Table 1 presents the results of UI subspace overlap across different testing instances. "Overlap freq." refers to the overlapping frequency, quantifying the number of testing instances (out of five) that explore the same UI subspace during parallel testing. Among the identified 209 UI subspaces, 202 UI subspaces (97% of all UI subspaces) are explored by more than one testing instance, and 76 UI subspaces (36% of all UI subspaces) are explored by all testing instances. Consequently, without coordination, different testing instances tend to repetitively explore the same UI subspace, resulting in overlapping explorations and decreasing the effectiveness of parallelized testing.

Answer to RQ1: The prevalent overlapped exploration decreases the parallelization effectiveness, not addressed by the intrinsic randomness of testing tools.

3.3 RQ2: Activity-based Parallelization Strategy

To investigate the applicability of simple activity-based parallelization implemented in ParaAim [10], we implement the parallelization strategy with the WCTester testing tool. Since the original paper of ParaAim [10] claims the difficulty of designing a parallelization strategy generally applicable to any testing tool, we choose to adapt WCTester whose exploration strategy is also activity-based, being the most similar to the parallelization strategy.

Poor generalization of activity-based parallelization. Table 2 presents the results of method coverage achieved by WCTester under different parallelization settings. For each parallelized testing setting, the covered methods are

Table 1. Overlaps of UI subspace exploration.

Overlap freq.	1/5	2/5	3/5	4/5	5/5
# of subspaces	7 (3%)	9 (4%)	57 (27%)	60 (29%)	76 (36%)

Table 2. Method coverage of WCTester under different parallelization settings.

App Name	Baseline	Parallel	Rel. Improve.
AbsWorkout	9483	7679	-19.0 %
AccuWeather	20692	20306	-1.9 %
AutoScout24	38653	13758	-64.4 %
Merriam-Webster	10398	7936	-23.7 %
Duolingo	12852	12789	-0.5 %
Filters For Selfie	1145	2559	+123.5 %
GoodRx	15841	6171	-61.0 %
Google Chrome	11329	9307	-17.8 %
Google Translate	11553	5030	-56.5 %
WEBTOON	25139	13860	-44.9 %
Marvel Comics	6565	1115	-83.0 %
Ms Word	12928	6590	-49.0 %
Quizlet	39675	31062	-21.7 %
Sketch	8061	4149	-48.5 %
TripAdvisor	27743	16356	-41.0 %
Trivago	33043	20275	-38.6 %
UC Browser	23813	19690	-17.3 %
Zedge	63574	67569	+6.3 %
Average	**20693**	**14788**	**-28.5%**

calculated by accumulating all unique methods covered by different testing instances. If one method is covered by more than one testing instance, the method is counted only once.

From Table 2, it is surprising that the activity-based parallelization strategy generally reduces the testing effectiveness on 89% apps, achieving 28.5% method coverage drop on average compared to baseline performance (i.e., running testing instances without any coordination) with the same testing resources. A major reason for the effectiveness drop is that WCTester prioritizes the UI actions that trigger Activity transitions. Considering that WCTester has a quite similar exploration strategy to the parallelization strategy, this phenomenon demonstrates the poor generalization of the activity-based parallelization strategy. In summary, with the fast development of exploration strategy [26, 47, 53, 60], it is necessary to design a general parallelization strategy applicable to any testing tool.

Answer to RQ2: Existing parallelization strategies cannot be generalized to parallelize different automated UI testing tools. A generally applicable parallelization strategy remains an unresolved challenge.

4 Problem Formulation

In this section, we formalize and prove our intuition of loosely coupled UI subspaces with graph theory.

4.1 Parallelizing Automated UI Testing as Min-Conductance Graph Partitioning

We first formalize the design of parallelization strategy as an optimization problem of graph partitioning.

Automated UI testing can be modeled as a random walking process on a stochastic directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P})$, where $\mathcal{V}$ denotes UI state set, $\mathcal{E}$ denotes UI action set triggering transitions between UI states, and $\mathcal{P} : \mathcal{E} \rightarrow [0, 1]$ denotes the probabilistic transition function defining the likelihood of selecting UI actions in $\mathcal{E}$ at each UI state by the test generation tool.

In parallelized automated UI testing, $N \geq 2$ testing instances are executed on $\mathcal{G}$, each covering a subgraph $\mathcal{G}_i \subset \mathcal{G}$. A *parallelization strategy* $\mathcal{S}$ represents a particular combination of N subgraphs. These subgraphs of differing testing instances exhibit overlaps, i.e., overlapping explorations. The goal of parallelized automated UI testing is to minimize these overlapping explorations (i.e., $\mathcal{G}_i \cap \mathcal{G}_j = \emptyset$). Thus, designing a parallelization strategy can be reduced to optimizing a graph partitioning problem [28], where each graph partitioning solution corresponds to a parallelization coordination solution.

An ideal partitioning should preserve the behavior of the test generation tool on the AUT. Partitioning the graph removes a set of edges (denoted as $cut(s)$) to yield K disjointed subgraphs, and the probability sum of edges in $cut(s)$ should be minimized. This formulated optimization problem closely aligns with the Minimum Conductance Graph Partitioning Problem [11, 44] (shortened to *MC-GPP*), a widely studied and known NP-hard problem [28, 57]. In our setting where the graph is weighted and directed, we extend the original MC-GPP definition. The *conductance* from $\mathcal{G}_1$ to $\mathcal{G}_2$, $\phi(\mathcal{G}_1, \mathcal{G}_2)$, is then defined as follows:

$$\phi(\mathcal{G}_1, \mathcal{G}_2) = \frac{\sum_{i \in \mathcal{G}_1, j \in \mathcal{G}_2} p(i, j)}{\min\{|vol(\mathcal{G}_1)|, |vol(\mathcal{G}_2)|\}} \tag{2}$$

where a subgraph's volume $vol(\mathcal{G}_x) = \sum_{i \in \mathcal{G}_x, j \notin \mathcal{G}_x} p(j, i) - p(i, j) + 2 \times \sum_{i \in \mathcal{G}_x, j \in \mathcal{G}_x} p(i, j)$. Intuitively, $\phi(\mathcal{G}_1, \mathcal{G}_2)$ represents the tool's transition probability from $\mathcal{G}_1$ to $\mathcal{G}_2$. Designing an optimal parallelization strategy is equivalent to optimize the following target:

$$\text{MC-GPP:} \quad \mathcal{G}^* = \arg\min_{s \in \Omega} \max_{\mathcal{G}_i, \mathcal{G}_j \in s} \phi(\mathcal{G}_i, \mathcal{G}_j) \tag{3}$$

Here, $\Omega = s = (\mathcal{G}_1, \mathcal{G}_2, .., \mathcal{G}_k)$ represents the space of all possible k-way partitions of $\mathcal{G}$.

While minimizing the conductance between two UI subspaces, two possible scenarios arise: (1) $\phi(\mathcal{G}_1, \mathcal{G}_2) \approx 0$ and $\phi(\mathcal{G}_2, \mathcal{G}_1) \approx 0$: the two UI subspaces scarcely interact. As demonstrated in our motivating example in Section 2, the main shopping functionality and the account settings functionality of the online shopping app in Figure 2 are connected solely through the Account Tab. (2) $\phi(\mathcal{G}_1, \mathcal{G}_2) \gg 0$ and $\phi(\mathcal{G}_2, \mathcal{G}_1) \approx 0$: the UI subspace $\mathcal{G}_1$ can easily transition

to $\mathcal{G}_2$ but the reverse is infrequent. Both cases are referred to as *loosely coupled* UI subspaces.

4.2 Exploiting Local Density of Loosely Coupled UI Subspaces for Online Partitioning

Based on the preceding formulation, we further address the challenge of global parallelization with local information by exploiting the Global Sparsity and Local Density (*GS-LD*) property of $\mathcal{G}$, the graph representation of the AUT's UI subspace. Generally speaking, mobile apps are component-based software, comprising multiple functionalities that are relatively independent [30, 38], and UI states implementing a specific functionality are tightly interconnected. Given this property, we prove that parallelization with local information can yield satisfactory results.

Theorem 1. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be n-complete graphs (i.e., complete graph with n vertices) connected by an edge c. Assume that the probability of selecting each edge is equal to $1/n$, and the probability of selecting edge c is far less than $1/n$. Given $N \geq O(n^2 \log n)$, the graph conductance between $\mathcal{G}_1$ and $\mathcal{G}_2$ is expected to be statistically smaller than the graph conductance inside $\mathcal{G}_1$.*

Proof Sketch. The central idea is that every edge inside the n-complete subgraph has probability $\frac{1}{n}$ from a given vertex, while the cross edge linking the two subgraphs has a much smaller probability $\frac{1}{\alpha n}$, with α far greater than 1. By sampling the edges sufficiently many times (on the order of $n^2 \log n$), standard Chernoff/Hoeffding bounds [13] guarantee that the observed frequency of any internal edge stays well above that of the rarer cross edge. As a result, the online exploration discovers that internal edges appear much more frequently, leading it to identify the correct separation between the two subgraphs with high probability.

Proof of Theorem 1. Within G_1, each vertex v_i has $n-1$ incident edges. For each *internal* edge $e = (v_i, v_j)$, we denote its *true probability* $p_e = \frac{1}{n}$ when sampling from v_i. For the *cross* edge $c = (v_i, u_j)$ between G_1 and G_2, by assumption that its probability is $p_c = \frac{1}{\alpha n}$ with $\alpha \gg 1$. Let $freq(e)$ be the empirical frequency of choosing e over N steps (restricted to the times when the traversal starts from vertex v_i).

Focus on a single vertex v_i in G_1. Over the course of N total samples, let m_i be the number of times that we *start* a step from v_i. Then $freq(e)$ for edges e emanating from v_i can be viewed as $freq(e) = \frac{X_e}{m_i}$, where $X_e \sim \text{Binomial}(m_i, p_e)$ and $E[X_e] = m_i p_e$. Based on standard Chernoff inequality [13], if we have $X \sim \text{Binomial}(m_i, p)$, then for $0 < \delta < 1$:

$$\Pr[X \leq (1-\delta) m_i p] \leq \exp(-\frac{\delta^2}{2} m_i p).$$

Hence for each edge e in G_1, we have high-probability bounds around $\frac{m_i}{n}$.

Our main goal is to show

$$(\min_{e \in E_1} freq(e)) > freq(c),$$

with high probability, where E_1 denotes the set of all internal edges in G_1.

Lower bound for internal edges. Since there are at most n edges leaving each vertex $v_i \in G_1$, we can union-bound over those edges to get (with high probability) a uniform lower bound on the frequency at each vertex. Then union-bound over all n vertices in G_1. Provided that $m_i \approx N/n$ is sufficiently large (on the order of $\Omega(n \log n)$), the Chernoff bound shows that each internal edge e satisfies $freq(e) \geq \frac{1}{n} - \varepsilon$ for some small $\varepsilon > 0$ with high probability, uniformly over all edges $e \in E_1$.

Upper bound for cross edge c. Similarly, for the cross edge c, each time when we start from $v_i \in G_1$, the probability is $p_c = \frac{1}{\alpha n}$ with $\alpha \gg 1$. If m_i is again $\approx \frac{N}{n}$, then $X_c \sim \text{Binomial}(m_i, \frac{1}{\alpha n})$ implies $freq(c) = \frac{X_c}{m_i} \approx \frac{1}{\alpha n}$ up to small fluctuations. For large α, we have $\frac{1}{\alpha n} \ll \frac{1}{n}$.

Based on the preceding information, choosing $N \geq Cn^2 \log n$ (for some sufficiently large C) ensures with high probability that

$$\min_{e \in E_1} \underbrace{freq(e)}_{\gtrsim \frac{1}{n}} > \underbrace{freq(c)}_{\lesssim \frac{1}{\alpha n}}.$$

Hence, the estimated conductance between G_1 and G_2 (proportional to $freq(c)$) remains significantly smaller than the internal conductance in G_1.

By standard arguments linking edge-frequency estimations to partitioning conductance (i.e., if a cross edge is observed to be much rarer than internal edges, the partition $\{G_1, G_2\}$ naturally follows), we conclude that after $N \geq Cn^2 \log n$ steps of online sampling, *with high probability* the subspace partition G_1 vs. G_2 is correctly identified by comparing internal vs. cross-edge frequencies. Since local sampling has revealed that cross-edge frequencies are significantly lower, the system recognizes that G_1 and G_2 form two distinct UI subspaces, completing the proof. □

Theorem 1 provides three key theoretical insights for implementing TAOPT. First, it proves that effective online partitioning is theoretically tractable, validating our dynamic partitioning approach during testing. Second, it shows that least frequently traversed edges serve as natural subspace boundaries, enabling a principled implementation for identifying subspace entry points. Third, it reveals that partition accuracy improves with longer exploration time but increases overlap-explore risk, allowing us to balance these factors through hyperparameters.

5 The TaOPT Approach

Figure 1(b) depicts the overview of TaOPT's architecture, which consists of two main modules. The *trace analyzer* (detailed in Section 5.2) monitors UI transition traces (i.e., logs of UI screen changes caused by the testing tool) and identifies loosely coupled UI subspaces, while the *test coordinator* (detailed in Section 5.3) schedules UI subspaces among testing instances (i.e., mobile devices/emulators with the testing tool running on them) based on the results of UI subspace identification.

The initial input of TaOPT is a set of mobile devices or emulators, an App Under Test (AUT), and an automated UI testing tool. During the testing process, TaOPT also takes the UI transition traces as inputs. The output of TaOPT is a list of UI subspaces of the given testing tool on the AUT, and a schedule of dedicating each UI subspace to one testing instance. The outputs of TaOPT are dynamically generated and immediately used during the testing process, explained in the workflow section (Section 5.1).

5.1 Workflow of TaOPT

Figure 4 presents TaOPT's workflow. Depending on the chosen mode - *duration-constrained* or *resource-constrained* - the coordinator initiates one or more testing instances (step 0). At the beginning, TaOPT has no information about the AUT's UI space and lets the testing tool explore the AUT on each testing instance (step 1).

During the tool exploration, TaOPT monitors the UI transition (step 2) on each testing instance and analyzes the UI transition traces (step 3) with the online trace analysis algorithm (detailed in Section 5.2). Upon identification of a new UI subspace (step 4), TaOPT dedicates the detected UI subspace to a specific testing instance (step 5) while barring other testing instances from exploring it (step 5).

For example, in the third phase of Figure 4, UI subspace X (colored in red) is dedicated to testing instance A, and UI subspace Y (colored in green) is dedicated to testing instance B. The subsequent exploration in testing instance A cannot access UI subspace Y anymore, and the subsequent exploration in testing instance B cannot access UI subspace X anymore. TaOPT also allocates a new testing instance C (step 6) if there are available mobile devices/emulators. The newly allocated testing instance C cannot access either UI subspace X or Y, but can access other parts of the UI space.

In the subsequent testing process, the three testing instances explore different parts of the UI space (step 7) with reduced overlapping explorations. TaOPT repeats steps 1 to 7 until the end of testing.

5.2 UI Subspace Identification with a Trace Analyzer

To make TaOPT generally applicable to any UI test generation tool without modifying tools' internal logic or modifying the AUT, we implement a trace analyzer with the Toller framework [64]. Toller monitors and reports immediately any UI action along with the context AUT UI without modifying the test generation tool or the AUT. The logs of Toller form *UI transition traces*, defined as a sequence of UI screens interspersed with corresponding UI actions.

Algorithm 1 presents the online trace analysis algorithm FindSpace to identify the entrypoint to a UI subspace based on the UI transition traces. FindSpace takes a UI transition trace S and a parameter $l_{\min}$ as inputs. FindSpace examines each UI screen p in S to determine whether the exploration following screen p constitutes a loosely coupled UI subspace relative to the UI space preceding screen p. As can be seen, FindSpace targets both higher-level irreversible transition of UI exploration space and lower-level difficulties of exercising a specific functionality. Let $S[0:p]$ and $S[p:N]$ represent the sets of UI screens explored before and after screen p, respectively. FindSpace calculates a score (Line 11), signifying the degree of UI screen overlap or mutual transition between $S[0:p]$ and $S[p:N]$. For each screen s in $S[0:p]$, the algorithm counts its appearances in $S[p:N]$ (Line 7), abstracting each screen to avoid excessive counts of similar screens. This abstraction removes text associated with UI elements [5, 60]. The CountIn function (Line 7) calculates the tree similarity [66] of the two abstracted UI hierarchies to determine the times of the appearances of s in $S[0:p]$. The score is then normalized (Lines 9- 11) to find the screen p_{out} with the minimum score, denoting $S[p_{out}:N]$ as a UI subspace with its entrypoint at p_{out}.

The parameter $l_{\min}$ controls the minimum time to explore $S[p_{out}:N]$. On the one hand, the larger $l_{\min}$, the more thorough $S[p_{out}:N]$ is explored, and the trace analysis has more information about the potential UI subspace. On the other hand, the larger $l_{\min}$ means that a longer time period is needed before a UI subspace can be identified, likely missing opportunities for effective parallelization. To satisfy different needs of balancing testing duration and testing resources, we use two sets of empirically determined parameters: $l_{\min}^{long} =$ 5 minutes for the resource-constrained mode, and $l_{\min}^{short} = 1$ minute for the duration-constrained mode. Subspaces identified using l^{long}min are confidently accepted at once, while those identified with l^{short}min are accepted only when reported by at least two testing instances.

5.3 Parallelization with a Test Coordinator

To satisfy different needs of balancing testing duration and testing resources, TaOPT supports two parallelization modes, namely the duration-constrained and resource-constrained modes. The duration-constrained mode requires two user-specified parameters: the number of testing instances to run concurrently $d_{\max}$ as well as the test time budget l_p per testing instance. The test coordinator maintains exactly $d_{\max}$ concurrent testing instances throughout the testing period l_p, immediately launching a new instance whenever one

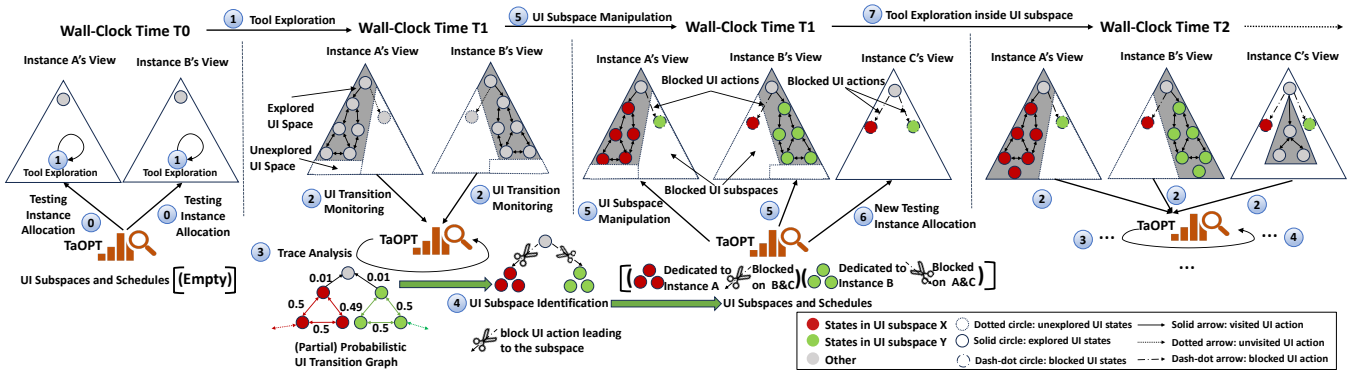

Figure 4. The workflow of TaOPT. Steps with the same numbers represent the same operation.

Algorithm 1: FindSpace: Identifying loosely coupled UI subspace via trace analysis.

Input: List of UI screens S along with their timestamps T, the threshold after partition $l_{\min}$
Output: Indexes of entrypoints of the UI subspace, or nil

$N, p_{\text{out}}, score_{\min} \leftarrow |S|, nil, 1$
$p_{\max} \leftarrow \max\{p : p \in [0, N-1] \wedge T[p] \leq T[N-1] - l_{\min}\}$
$sample\_size \leftarrow |\text{Set}(S[p_{\max}+1 : N])|$
foreach $p \in 1$ to p_{max} **do**
  $overlap\_size \leftarrow 0$
  foreach $s \in \text{Set}(S[0:p])$ **do**
    $overlap\_size \leftarrow overlap\_size + \text{CountIn}(s, S[p : N])$
  end
  $overlap\_score \leftarrow \frac{overlap\_size}{N-p}$
  $purity\_score \leftarrow \text{Sigmoid}(\frac{|\text{Set}(S[p:N])|}{sample\_size} - 1)$
  $score \leftarrow overlap\_score + 2 * purity\_score - 1$
  if $score < score_{min}$ **then**
    $score_{\min}, p_{\text{out}} \leftarrow score, p$
  end
end
return p_{out}

is de-allocated. In contrast, the resource-constrained mode requires only the total machine hour budget T. Starting with a single testing instance, the number of concurrent instances dynamically adjusts based on the number of detected UI subspaces and the testing progress within each subspace.

During the testing (i.e., the testing tool's exploration of the AUT), UI subspaces and entrypoints can be identified by the trace analyzer (detailed in Section 5.2). The test coordinator dedicates each UI subspace to one testing instance and manages testing instance allocation/de-allocation.

First, the test coordinator broadcasts UI subspace entrypoint information to all running testing instances. For the testing instance that identifies a UI subspace, the test coordinator grants access to both the entrypoint and the corresponding UI subspace. For other testing instances, the test coordinator blocks the entrypoint, i.e., blocking access to the corresponding UI subspace. Specifically, TaOPT uses the Toller framework [64] for screen update notifications. Upon each screen update, TaOPT obtains a UI hierarchy, identifies UI elements matching any blocked entrypoint, and instructs Toller to disable these elements before the test generation tool can interact with them.

Second, when any available mobile devices are not running and a new UI subspace X is identified, the test coordinator will allocate a new testing instance and block the entrypoint to the UI subspace X on the new testing instance.

Finally, the test coordinator monitors the testing progress of testing instances. If one testing instance does not discover new UI screens for $l_{\min}^{short} = 1$ minute, the test coordinator will de-allocate the testing instance. After de-allocation, the response varies by mode. In the duration-constrained mode, a new testing instance is immediately allocated with all entrypoints to identified UI subspaces blocked. In the resource-constrained mode, new testing instance allocation is deferred until new UI subspaces are identified.

6 Evaluation

Our evaluation answers the following research questions:

- **RQ3:** How much testing duration can TaOPT reduce compared with baselines?
- **RQ4:** How many testing resources (i.e., machine time of all testing instances) can TaOPT reduce compared with baselines?
- **RQ5:** Given the same testing resources or duration, how much effectiveness can TaOPT improve (measured by code coverage and unique crashes) compared with baselines?
- **RQ6:** How effectively can TaOPT reduce overlapping explorations and preserve the behavior of UI testing tools?

6.1 Evaluation Setup

Parallel run settings. We conduct parallel runs with three settings: (1) baseline parallelization (i.e., parallelized testing without coordination across testing instances, the same

Table 3. Overview of used industrial apps.

App Name	Version	Category	#Inst
AbsWorkout	4.2.0	Health & Fitness	10m+
AccuWeather	7.4.1-5	Weather	100m+
AutoScout24	9.8.6	Auto & Vehicles	10m+
Duolingo	3.75.1	Education	100m+
Filters For Selfie	1.0.0	Beauty	10m+
GoodRx	5.3.6	Medical	10m+
Google Chrome	65.0.3325	Communication	10b+
Google Translate	6.5.0	Books & Reference	1b+
Marvel Comics	3.10.3	Comics	10m+
Merriam-Webster	4.1.2	Books & Reference	10m+
Ms Word	16.0.15	Personal	1b+
Quizlet*	6.6.2	Education	10m+
Sketch	8.0.A.0.2	Art & Design	50m+
TripAdvisor*	25.6.1	Food & Drink	100m+
Trivago	4.9.4	Travel & Local	50m+
UC Browser	13.0.0.1288	Communication	1b+
WEBTOON*	2.4.3	Comics	100m+
Zedge	7.34.4	Personalization	100m+

Notes: "#Inst" denotes the approximate number of downloads. Apps with asterisks (*) after their names are the ones that require a login to access most features.

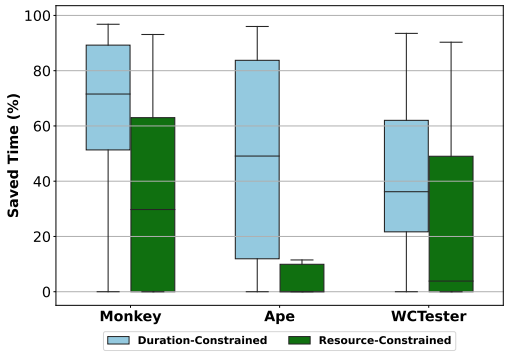

Figure 5. Statistics of testing duration saved by TAOPT.

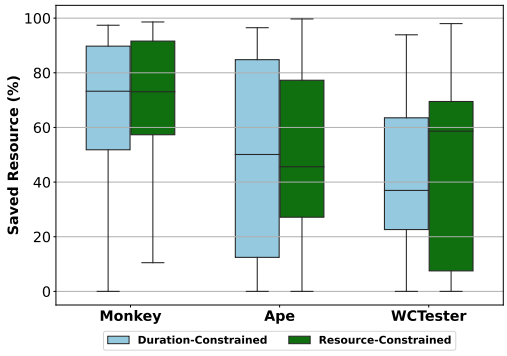

Figure 6. Statistics of testing resources saved by TAOPT.

setting as used in Section 3), (2) TAOPT with the resource-constrained mode, and (3) TAOPT with the duration-constrained mode.

For baseline parallelization and TAOPT with the duration-constrained mode, we set the parallel testing time $l_p = 1$ hour and the number of testing instances $d_{\max} = 5$. For each *baseline parallelization*, we simply start $d_{\max}$ testing instances at once and let the test generation tool explore the AUT for l_p without interruption or interference.

For TAOPT with the resource-constrained mode, we launch one testing instance at start and allocate $l_m = l_p \times d_{\max} = 5$ machine hours. There is no more than $d_{\max}$ concurrent testing instances at any time in the parallelized testing. The parallelized testing is ended until all machine hours are used.

Automated UI testing tools. Three state-of-the-art/practice Android UI testing tools are involved in our study: Monkey [21], WCTester [72, 78], and Ape [26]. Monkey [21] is one of the most widely used tools in industrial settings, and WCTester [72, 78] is a state-of-the-practice tool used to test WeChat, a highly popular app with over one billion monthly active users. Ape [26] is an advanced model-based testing tool exhibiting superior effectiveness on industrial apps [53, 64–66].

Subject apps. We conduct the evaluation on highly popular industrial apps that are widely used for automated UI testing evaluation [53, 64–66]. We obtain 18 apps that run properly on our x64 Android emulators as presented in Table 3.

Test platform. All experiments are conducted on the official Android x64 emulators running Android 6.0 on a server with four AMD EPYC 7H12 64-Core Processors. Each emulator is allocated with 4 dedicated CPU cores, 2 GB RAM, and 2 GB internal storage. Emulator data are stored on an in-memory disk for minimal mutual influences caused by disk I/O bottlenecks. Hardware graphics acceleration is also enabled with two Nvidia GeForce RTX 3090 Graphics Cards to ensure the responsiveness of emulators. We manually write auto-login scripts for apps (with '*' marks in Table 3) requiring a login to access their main functionalities. Each of these scripts is executed only once before the corresponding app starts to be tested in each testing instance.

Coverage collection. We collect the method coverage as a fine-grained code coverage metric achieved by each testing instance, using the MiniTrace [25] tool from Ape. By modifying DalvikVM/ART, the tool does not require app instrumentation, avoiding unexpected issues from modifying industrial apps in our experiments.

Crash collection. We consider only crashes originating from apps' bytecode. Code locations in stack traces are used to identify unique crashes [64–66]. We obtain stack traces by monitoring Android Logcat [22] messages.

Table 4. Statistics of cumulative code coverage.

App Name	Baseline			TaOPT (Duration)			TaOPT (Resource)		
	Mon.	Ape	WCT.	Mon.	Ape	WCT.	Mon.	Ape	WCT.
AbsWorkout	7558	9421	9483	9363 (+23%)	11135 (+18%)	9979 (+5%)	8181 (+8%)	10046 (+6%)	9836 (+3%)
AccuWeather	13366	22916	20692	23020 (+72%)	24724 (+7%)	25066 (+21%)	22924 (+71%)	23239 (+1%)	24746 (+19%)
AutoScout24	31196	41629	38653	42368 (+35%)	43464 (+4%)	39061 (+1%)	43058 (+38%)	43434 (+4%)	39406 (+1%)
Merriam-Webster	6322	10180	10398	10444 (+65%)	11286 (+10%)	10494 (+0%)	10403 (+64%)	10663 (+4%)	10454 (+0%)
Duolingo	12952	16702	12852	15509 (+19%)	16811 (+0%)	15262 (+18%)	15440 (+19%)	16782 (+0%)	15534 (+20%)
Filters For Selfie	4611	2727	1145	4507 (-2%)	4446 (+63%)	2587 (+125%)	4806 (+4%)	4702 (+72%)	2575 (+124%)
GoodRx	16274	17751	15841	16158 (0%)	18817 (+6%)	16091 (+1%)	16363 (+0%)	17928 (+0%)	15085 (-4%)
Google Chrome	9874	12787	11329	12151 (+23%)	12835 (+0%)	12601 (+11%)	12034 (+21%)	13618 (+6%)	12268 (+8%)
Google Translate	8986	11074	11553	10510 (+16%)	11680 (+5%)	11088 (-4%)	10512 (+16%)	10743 (-2%)	10409 (-9%)
WEBTOON	26206	27673	25139	27153 (+3%)	28186 (+1%)	25885 (+2%)	27546 (+5%)	31135 (+12%)	27845 (+10%)
Marvel Comics	6650	5709	6565	7276 (+9%)	6703 (+17%)	6379 (-2%)	5961 (-10%)	6675 (+16%)	5137 (-21%)
Ms Word	13477	14291	12928	13970 (+3%)	14325 (+0%)	13837 (+7%)	13397 (0%)	14571 (+1%)	13290 (+2%)
Quizlet	43127	50559	39675	49624 (+15%)	52500 (+3%)	43099 (+8%)	49177 (+14%)	50828 (+0%)	46683 (+17%)
Sketch	7198	9730	8061	9528 (+32%)	9617 (-1%)	8401 (+4%)	9508 (+32%)	9676 (0%)	7532 (-6%)
TripAdvisor	22411	29496	27743	26669 (+18%)	31105 (+5%)	29569 (+6%)	24720 (+10%)	30317 (+2%)	27590 (0%)
Trivago	36919	20403	33043	40321 (+9%)	20426 (+0%)	34041 (+3%)	38863 (+5%)	40166 (+96%)	35041 (+6%)
UC Browser	26436	30938	23813	30624 (+15%)	32989 (+6%)	25662 (+7%)	28958 (+9%)	32407 (+4%)	26622 (+11%)
Zedge	62380	71475	63574	79476 (+27%)	85187 (+19%)	81322 (+27%)	64658 (+3%)	92585 (+29%)	75283 (+18%)
Average	**19774**	**22525**	**20693**	**23815**	**24235**	**22801**	**22583**	**25528**	**22518**
Average Δ	-	-	-	**+21% (+20.4%)**	**+9% (+7.6%)**	**+13% (+10.2%)**	**+17% (+14.2%)**	**+14% (+13.3%)**	**+11% (+8.8%)**

*Notes: **Average** Δ stands for the average improvement. Δ = (#TaOPT − #Baseline) ÷ #Baseline × 100% in the corresponding tool. Mon. and WCT. represent Monkey and WCTester, respectively.*

Table 5. Statistics of distinct crashes.

App Name	Baseline			TaOPT (Duration)			TaOPT (Resource)		
	Mon.	Ape	WCT.	Mon.	Ape	WCT.	Mon.	Ape	WCT.
AbsWorkout	2	2	1	5 (+150%)	7 (+250%)	1 (0%)	4 (+100%)	4 (+100%)	2 (+100%)
AccuWeather	0	2	0	1 (+100%)	1 (-50%)	1 (+100%)	0 (0%)	2 (0%)	0 (0%)
AutoScout24	0	0	0	1 (+100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Merriam-Webster	0	1	1	2 (+200%)	1 (0%)	1 (0%)	2 (+200%)	4 (+300%)	0 (-100%)
Duolingo	0	1	0	0 (0%)	1 (0%)	1 (+100%)	0 (0%)	2 (+100%)	2 (+200%)
Filters For Selfie	1	0	0	0 (-100%)	0 (0%)	0 (0%)	1 (0%)	0 (0%)	0 (0%)
GoodRx	1	2	1	2 (+100%)	0 (-100%)	0 (-100%)	1 (0%)	1 (-50%)	0 (-100%)
Google Chrome	0	0	0	0 (0%)	1 (+100%)	0 (0%)	0 (0%)	0 (0%)	1 (+100%)
Google Translate	1	3	2	8 (+700%)	2 (-33%)	0 (-100%)	6 (+500%)	10 (+233%)	5 (+150%)
WEBTOON	1	2	0	0 (-100%)	3 (+50%)	2 (+200%)	0 (-100%)	1 (-50%)	0 (0%)
Marvel Comics	2	2	0	2 (0%)	3 (+50%)	0 (0%)	2 (0%)	2 (0%)	0 (0%)
Ms Word	1	0	1	3 (+200%)	0 (0%)	0 (-100%)	1 (0%)	0 (0%)	0 (-100%)
Quizlet	5	1	0	3 (-40%)	3 (+200%)	0 (0%)	2 (-60%)	2 (+100%)	1 (+100%)
Sketch	0	1	0	2 (+200%)	0 (-100%)	0 (0%)	0 (0%)	0 (-100%)	0 (0%)
TripAdvisor	0	4	1	3 (+300%)	3 (-25%)	3 (+200%)	0 (0%)	2 (-50%)	0 (-100%)
Trivago	0	1	0	2 (+200%)	2 (+100%)	1 (+100%)	1 (+100%)	1 (0%)	0 (0%)
UC Browser	2	1	0	0 (-100%)	1 (0%)	0 (0%)	0 (-100%)	1 (0%)	0 (0%)
Zedge	1	2	1	3 (+200%)	3 (+50%)	1 (0%)	4 (+300%)	3 (+50%)	1 (0%)
Total	**17**	**25**	**8**	**37**	**31**	**11**	**24**	**35**	**12**
Average Δ	-	-	-	**+117% (+118%)**	**+27% (+24%)**	**+22% (+38%)**	**+52% (+41%)**	**+35% (+14%)**	**+13% (+50%)**

*Notes: 'Δ = (#TaOPT − #Baseline) ÷ #Baseline × 100% for the corresponding tool. **Average** Δ stands for the average improvement. Δ = (#TaOPT − #Baseline) ÷ #Baseline × 100% in the corresponding tool. Mon. and WCT. represent Monkey and WCTester, respectively.*

6.2 RQ3. Reduction of needed testing duration

We investigate how much testing duration can be reduced by TaOPT while maintaining the same test effectiveness compared to baseline parallelization. Using cumulative code coverage over time as our effectiveness metric, we calculate testing duration reduction by identifying when TaOPT achieves the same code coverage as baseline runs' full-duration (1 hour) results. We then calculate unused testing duration remaining at this point and compute the reduction percentage by dividing remaining testing duration by total allocated testing duration.

Figure 5 presents the statistics of the testing duration saved by TaOPT on different apps. TaOPT-conducted parallel runs use substantially less testing duration to achieve

the same average code coverage with baseline runs, with 64.0%, 48%, 41.0% less testing duration on average needed by Monkey, Ape, and WCTester in duration-constrained parallel runs. While the resource-constrained mode of TaOPT is not expected to save substantial testing duration, the resource-constrained mode can still save testing duration on 12, 6, and 9 apps for Monkey, Ape, and WCTester, respectively.

In conclusion, TaOPT helps save substantial testing duration, being especially beneficial for testing tasks requiring timely completion.

6.3 RQ4. Reduction of needed machine time

We investigate how many testing resources (measured in machine time) can be reduced by TaOPT when reaching the same test effectiveness as the baseline parallelization. To calculate the reduction of needed machine time, we find the remaining machine time not used by TaOPT-conducted testing instances when they achieve the same code coverage that is achieved by baseline runs using the total machine time (i.e., 5 machine hours). By dividing the remaining machine time by the total machine time, we obtain the reduction of needed machine time.

Figure 6 presents the statistics of the testing resources saved by TaOPT on different apps. We find that TaOPT-conducted parallel runs use substantially less machine time to achieve the same average code coverage with the baseline parallelization runs, with 65.9%, 50.1%, 47.6% less, and 64.6%, 48.9%, 42.5% less machine time needed by Monkey, Ape, and WCTester in resource-constrained and duration-constrained parallel runs, respectively.

Notably, the duration-constrained mode achieves comparable or sometimes better testing resource savings than the resource-constrained mode. To better understand this observation, we conduct non-parallelization testing with single testing instances of $l_p \times d_max = 5$ hours, using the same machine hours but without any parallelization. In the 5-hour non-parallelization run, Monkey, Ape, and WCTester cover 19133, 22771, and 17851 methods, respectively. We find out that baseline parallelization runs generally achieve comparable or even higher method coverage (shown in Table 4) compared to non-parallelization runs, despite overlapping explorations. The results explain the substantial testing resource savings in the duration-constrained mode and suggest that dividing testing budget across multiple testing instances can be more effective than single long-duration runs. TaOPT further enhances this advantage by reducing overlapping explorations.

In conclusion, TaOPT achieves substantial testing resource savings, reducing testing resource costs by up to 65.9%. Given the high costs of mobile testing resources (e.g., AWS Device Farm's rate of $0.17 per device minute for real devices [55]), TaOPT's efficiency improvements translate to significant economic benefits for testers.

6.4 RQ5. Test effectiveness improvement

This RQ aims to find out that given the same quantity of testing resources (i.e., machine time), whether TaOPT is able to help testing tools achieve better test effectiveness in terms of cumulative method coverage and numbers of distinct crashes. For each parallel run, its cumulative method coverage and distinct crashes are calculated as the union of distinct methods covered and crashes triggered in each testing instance, respectively.

Table 5 shows the statistics of distinct crashes and Table 4 shows the statistics of cumulative code coverage. As can be seen, TaOPT-conducted parallel runs consistently achieve higher code coverage and discover more unique crashes compared with the baseline runs, covering 14.6% more methods and detecting 1.64× unique crashes on average. To prevent potential skew in the average metric caused by outliers, we also present the percentage increase or decrease in code coverage and crashes detected for each app compared to the baseline. Most cases (81.5%) demonstrate positive growth in cumulative code coverage compared to the baseline. Similarly, for crash detection, both modes consistently detect a higher or equal number of crashes in most cases (78.7%), demonstrating TaOPT's effectiveness across different apps.

To validate whether the improvement resulted by substantially changing the testing tool's behavior, we then examine how TaOPT changes the behavior of the test generation tool by examining the method coverage overlap between baseline parallelization runs and TaOPT-enhanced runs. We calculate the Jaccard similarity [54] between the sets of methods covered by baseline parallelization and TaOPT-enhanced runs, and the proportion of methods covered by baseline parallelization but not by TaOPT-enhanced runs. Specifically, the Jaccard similarity between the duration-constrained mode of TaOPT and baseline runs are 0.77, 0.86, and 0.85 for Monkey, Ape, and WCTester, respectively. In other words, 3.7%, 3.4%, and 3.3% methods covered by baseline parallelization runs are not covered by the duration-constrained mode of TaOPT for Monkey, Ape, and WCTester, respectively. the Jaccard similarity between the resource-constrained mode of TaOPT and baseline parallelization runs is 0.77, 0.81, and 0.83 for Monkey, Ape, and WCTester, respectively. In other words, 5.0%, 5.1%, and 5.3% methods covered by baseline parallelization runs are not covered by the resource-constrained mode of TaOPT for Monkey, Ape, and WCTester, respectively. In conclusion, TaOPT consistently improves the parallelization effectiveness of three testing tools without compromising the effectiveness of the original testing tools, demonstrating the generalization of TaOPT and the effectiveness of using loosely coupled UI subspaces for parallelization.

6.5 RQ6. Reduction of overlapped explorations

This RQ aims to find out how TaOPT improves the effectiveness of parallelized testing by studying its capability of

Table 6. UI overlap measured by the average # of occurrences of distinct UIs.

App Name	Baseline			TaOPT (Duration)			TaOPT (Resource)		
	Mon.	Ape	WCT.	Mon.	Ape	WCT.	Mon.	Ape	WCT.
AbsWorkout	128.3	40.7	64.0	14.1	34.1	69.2	12.1	26.1	62.8
AccuWeather	12.1	17.0	39.6	3.2	18.2	17.9	3.1	10.0	14.2
AutoScout24	29.3	9.0	14.8	4.4	6.7	14.6	7.0	7.6	13.9
Merriam-Webster	7.9	45.3	22.7	7.4	27.6	44.3	5.7	22.7	30.5
Duolingo	67.3	37.2	23.3	14.4	27.2	68.9	25.1	29.4	27.7
Filters For Selfie	5.1	2398.5	2755.0	4.9	30.6	1355.9	3.7	37.4	1990.7
GoodRx	18.1	18.8	21.9	20.2	22.2	21.2	14.2	22.2	20.5
Google Chrome	79.4	10.3	12.8	10.0	10.7	16.3	8.8	12.7	14.2
Google Translate	20.8	17.4	20.7	9.3	15.4	20.3	10.5	16.7	21.4
WEBTOON	25.5	12.7	18.6	14.5	12.1	12.9	19.5	13.0	10.6
Marvel Comics	16.6	53.7	40.1	9.0	18.0	38.1	9.0	19.1	40.5
Ms Word	8.9	28.2	36.2	6.0	17.4	36.6	5.3	16.1	28.3
Quizlet	14.0	7.8	10.4	7.7	6.7	10.4	7.0	7.2	14.9
Sketch	28.4	11.1	43.2	35.5	15.3	38.7	31.9	11.4	38.4
TripAdvisor	52.1	8.1	6.9	19.2	8.0	7.6	19.1	7.8	7.2
Trivago	11.2	8.2	20.1	5.3	5.7	15.7	5.9	7.8	17.7
UC Browser	36.5	8.0	22.5	12.6	7.7	15.6	9.0	7.3	18.6
Zedge	10.1	8.7	673.7	8.0	10.0	38.3	16.2	10.9	38.7
Average	**31.8**	**152.3**	**213.7**	**11.4**	**16.3**	**102.4**	**11.8**	**15.9**	**133.9**
Δ	-	-	-	**64.5%**	**89.5%**	**52.1%**	**64.5%**	**90.1%**	**37.6%**

reducing overlapping explorations. We measure the average number of occurrences of distinct UI screens observed during testing across all testing instances, since TaOPT uses the UI screens and their similarity to determine the UI subspace. We represent UI screens by their abstract UI hierarchies (using the strategy from previous work [66]) to avoid being overly sensitive to screen content changes. Table 6 shows the statistics of overlapped UI screens for three parallel runs, where the Δ = (#Baseline − #TaOPT) ÷ #Baseline × 100% represents the relative overlap reduction for the corresponding tool. As can be seen, TaOPT-conducted parallel runs have substantially smaller UI overlaps on average compared with the baseline runs, with 64.5%, 90.1%, 37.6% fewer, and 64.5%, 89.5%, 52.1% fewer per-UI occurrences by Monkey, Ape, WCTester in resource and duration-constrained parallel runs, respectively. Notably, TaOPT reduces the most overlapping explorations in Ape, which has the highest overlapping explorations found in our preliminary study (detailed in Section 3.2). In conclusion, TaOPT effectively reduces overlapping explorations and substantially improves the parallelization effectiveness with high generalization.

7 Discussion

While we focus TaOPT on automated UI testing, the core concepts of TaOPT, i.e., detecting loosely coupled subspaces and the theoretical foundation of graph partitioning, can be generalized beyond GUI testing. First, our approach can be adapted to any event-driven system where the program state space can be partitioned based on event transitions. Examples include network protocols and distributed systems, where states and transitions can be analyzed similarly to UI screens and actions. Second, the theoretical foundation of our approach is particularly promising for generalization, as many software systems naturally exhibit the property of being globally sparse (loose coupling among major components) but locally dense (tight coupling within components).

8 Threats of validity

The main external threat comes from the environmental dependencies of our subject apps. Specifically, part of our subject apps require network access to maintain the main functionalities. Toward minimizing the effects of such environmental dependencies, we ensure the consistency of our experimental environment during the experiment process, make each parallel run include five testing instances, and use aggregated metrics. The major internal threat to our work would be the potential faults from both the implementation of Toller and the setup of all Android testing tools involved in our experiments, which may affect our experimental results. To mitigate these internal threats to the validity of our work, we output relevant logs and the used metrics for each experiment, and manually inspect the samples of experimental logs to ensure correctness.

9 Related work

Automated UI testing for Android. Existing tools are mainly in the form of automatic input generation, generally divided into three categories. (i) Random UI testing [40, 42, 50, 70]. For example, Monkey [21] generates random inputs without considering the semantics of app UIs, often serving as the baseline of comparison. (ii) Model-based UI testing [9, 14, 26, 27, 36, 39, 53, 60]. Badge [53] guides test input generation by dynamically constructing a stochastic

model of the app's behavior and leveraging this model to balance between exploiting known promising paths and exploring new UI states. (iii) Systematic exploration [1, 4, 41]. A3E [4] constructs systematic exploration traces using both dynamic and static analyses.

Parallel testing. Existing parallel testing includes parallelizing mutation testing [43], testing distributed software [34], and symbolic execution [7, 58]. Different from traditional programs, the control and data dependencies within UI-based apps are hard to obtain statically. Additionally, there exist efforts [10, 62, 67] aiming to enable parallel UI testing on Android. PATS [67] proposes a master-slave framework, where the master device performs initial explorations and dispatches tasks to slaves as new UI states are discovered. Such strategy is highly susceptible to overlapping explorations, mainly due to many UI transitions being bidirectional in real-world apps; for example, one can often press Back to return to the previous screen. ParaAim [10] proposes partitioning an Android app for parallel testing on activity granularity. This simple strategy is easily vulnerable to uneven partitioning, where various activities convey different quantities of app features, especially when using Fragments [23], and cannot be generalized to any testing tools.

Graph partitioning. Graph partitioning [28] is a popular and general model frequently used to formulate numerous practical applications in various domains [8]. Given that the general graph partition problem is NP-hard [28], numerous heuristics are proposed [3, 20] to efficiently approximate the optimal results under different relaxations of the underlying graphs. Previous work [3] studies the problem of partitioning graphs into subgraphs that are of similar sizes. There are also extensive research efforts focusing on practical scenarios where partitions are conducted based on local graph information [2, 29, 46, 59, 71].

Trace and log analysis. TaOPT is instantiated with trace and log analysis techniques. Existing work applies log analysis with various techniques. (i) Anomaly detection [19, 45, 73, 77]. LogRobust [73] uses deep neural networks to capture the context in sequences and learn the importance of different events automatically. (ii) Cause locating [12, 16, 56, 74, 79]. In the collected logs, Kairux [74] searches for the non-failure instruction sequence that has the longest common prefix with the fault sequence to assist in the cause locating of distributed-system faults. (iii) Performance-issue detection [52, 66, 75, 76]. VET [66] identifies performance issues of automated UI testing tools from logs collected from previous testing runs. Their analysis also supports our observation that loosely coupled UI spaces are prevalent across different testing tools and mobile apps.

10 Conclusion

In this paper, we have looked into the opportunities of reducing testing duration and/or saving testing resources in the context of parallelization for automated mobile UI testing. Specifically, we have presented TaOPT, a parallelization approach that automatically manipulates the App Under Test (AUT) to guide a UI test generation tool. TaOPT conducts on-the-fly trace analysis to infer loosely-coupled AUT UI subspaces and coordinates testing instances to explore each UI subspace, substantially reducing the overlapping explorations across testing instances. To evaluate TaOPT, we have applied it on 18 highly popular industrial apps with three state-of-the-art/practice tools for automated mobile UI testing. Our evaluation results have shown that TaOPT substantially and consistently improves the parallelization effectiveness of the three automated mobile UI testing tools.

Acknowledgments

Tao Xie is the corresponding author. This work was partially supported by NSFC under Grant No. 92464301, No. 623B2006, an Amazon Trust AI Research Award, and the Tencent Foundation/XPLORER PRIZE.

References

[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *FSE*.
[2] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local graph partitioning using pagerank vectors. In *FOCS*.
[3] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *SPAA*.
[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA*.
[5] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *ASE*.
[6] Kenneth P Birman. 1993. The process group approach to reliable distributed computing. *Commun. ACM* 36, 12 (1993).
[7] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *EuroSys*.
[8] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent advances in graph partitioning. *Algorithm engineering* (2016).
[9] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: a multi-agent model-based test generation system. In *AST*.
[10] Chun Cao, Jing Deng, Ping Yu, Zhiyong Duan, and Xiaoxing Ma. 2019. ParaAim: testing Android applications parallel at activity granularity. In *COMPSAC*.
[11] Jeff Cheeger. 2015. A lower bound for the smallest eigenvalue of the Laplacian. In *Problems in analysis*. Princeton University Press.
[12] An Ran Chen. 2019. An empirical study on leveraging logs for debugging production failures. In *ICSE*.
[13] Herman Chernoff. 1952. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics* (1952).
[14] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*.
[15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: are we there yet?. In *ASE*.
[16] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse debugging of failures

in deployed software. In *OSDI*.
[17] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008).
[18] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android obfuscation techniques: A large-scale investigation in the wild. In *SecureComm*.
[19] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS*.
[20] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. 1999. Fast approximate graph partitioning algorithms. *SIAM J. Comput.* 28, 6 (1999).
[21] Google. 2021. Android Monkey. https://developer.android.com/studio/test/monkey.
[22] Google. 2021. Logcat command-line tool. https://developer.android.com/studio/command-line/logcat.
[23] Google. 2023. Android Fragment. https://developer.android.com/reference/android/app/Fragment.
[24] Google. 2024. Introduction to Android activities. https://developer.android.com/guide/components/activities/intro-activities.
[25] Tianxiao Gu. 2021. MiniTrace. http://gutianxiao.com/ape.
[26] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *ICSE*.
[27] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*.
[28] Juris Hartmanis. 1982. Computers and intractability: a guide to the theory of NP-completeness (Michael R. Garey and David S. Johnson). *Siam Review* 24, 1 (1982).
[29] Avinatan Hassidim, Jonathan A Kelner, Huy N Nguyen, and Krzysztof Onak. 2009. Local graph partitions for approximation and testing. In *FOCS*.
[30] Jianjun Huang, Yousra Aafer, David Perry, Xiangyu Zhang, and Chen Tian. 2017. UI driven Android application reduction. In *ASE*.
[31] Kobiton. 2022. Kobiton: Mobile Device Testing. https://kobiton.com/.
[32] Sauce Labs. 2022. Sauce Labs. https://saucelabs.com/.
[33] LambdaTest. 2022. LambdaTest: Cross Browser Testing Cloud. https://www.lambdatest.com/.
[34] Alexey Lastovetsky. 2005. Parallel testing of distributed software. *Information and Software Technology* 47, 10 (2005).
[35] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of Android apps: A systematic literature review. *Information and Software Technology* 88 (2017).
[36] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A lightweight UI-guided test input generator for Android. In *ICSE-Companion*.
[37] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, and Tianyin Xu. 2023. Virtual device farms for mobile app testing at scale: A pursuit for fidelity, efficiency, and accessibility. In *MobiCom*.
[38] Yi Liu, Yun Ma, Xusheng Xiao, Tao Xie, and Xuanzhe Liu. 2023. LegoDroid: Flexible Android app decomposition and instant installation. *Science China Information Sciences* 66, 4 (2023).
[39] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *ASE*.
[40] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *ESEC/FSE*.
[41] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented evolutionary testing of Android apps. In *FSE*.
[42] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *ISSTA*.
[43] Pedro Reales Mateo and Macario Polo Usaola. 2013. Parallel mutation testing. *Software Testing, Verification and Reliability* (2013).
[44] Bojan Mohar. 1989. Isoperimetric numbers of graphs. *Journal of combinatorial theory, Series B* 47, 3 (1989).
[45] Cristina Monni and Mauro Pezzè. 2019. Energy-based anomaly detection a new perspective for predicting software failures. In *ICSE-NIER*.
[46] Lorenzo Orecchia and Zeyuan Allen Zhu. 2014. Flow-based algorithms for local graph clustering. In *SODA*.
[47] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA*.
[48] Perfecto. 2022. Perfecto: Web & Mobile App Testing | Continuous Testing. https://www.perfecto.io/.
[49] Michael J Quinn. 1994. *Parallel computing theory and practice*. McGraw-Hill, Inc.
[50] Dezhi Ran, Zongyang Li, Chenxu Liu, Wenyu Wang, Weizhi Meng, Xionglin Wu, Hui Jin, Jing Cui, Xing Tang, and Tao Xie. 2022. Automated visual testing for mobile apps in an industrial setting. In *ICSE-SEIP*.
[51] Dezhi Ran, Zihe Song, Wenyu Wang, Wei Yang, and Tao Xie. 2025. Implementation of TaOPT. https://github.com/PKU-ASE-RISE/TaOPT.
[52] Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A runtime framework for LLM-based UI exploration. In *ISSTA*.
[53] Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. 2023. Badge: Prioritizing UI events with hierarchical multi-armed bandits for automated UI testing. In *ICSE*.
[54] Raimundo Real and Juan M Vargas. 1996. The probabilistic basis of Jaccard's index of similarity. *Systematic biology* 45, 3 (1996).
[55] Amazon Web Services. 2022. AWS Device Farm. https://aws.amazon.com/device-farm/.
[56] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on Hadoop clouds. In *ICSE*.
[57] Jiří Šíma and Satu Elisa Schaeffer. 2006. On the NP-completeness of some graph cluster measures. In *SOFSEM*.
[58] Matt Staats and Corina Păsăreanu. 2010. Parallel symbolic execution for structural test generation. In *ISSTA*.
[59] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *KDD*.
[60] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *ESEC/FSE*.
[61] Vaidy S. Sunderam. 1990. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience* 2, 4 (1990).
[62] Porfirio Tramontana, Nicola Amatucci, and Anna Rita Fasolino. 2020. A technique for parallel GUI testing of Android applications. In *ICTSS*.
[63] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *ICSE*.
[64] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An infrastructure approach to improving effectiveness of Android UI testing tools. In *ISSTA*.
[65] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *ASE*.
[66] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: Identifying and avoiding UI exploration tarpits. In *ESEC/FSE*.
[67] Hsiang-Lin Wen, Chia-Hui Lin, Tzong-Han Hsieh, and Cheng-Zen Yang. 2015. PATS: A parallel GUI testing framework for Android applications. In *COMPSAC*.
[68] Sinead A Williamson and Mauricio Tec. 2020. Random clique covers for graphs with local density and global sparsity. In *UAI*.

[69] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *FASE*.

[70] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droid-Fuzzer: Fuzzing the Android apps with intent-filter tag. In *MoMM*.

[71] Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. 2017. Local higher-order graph clustering. In *KDD*.

[72] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: Are we really there yet in an industrial case?. In *FSE*.

[73] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. 2019. Robust log-based anomaly detection on unstable log data. In *ESEC/FSE*.

[74] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The inflection point hypothesis: A principled debugging approach for locating the root cause of a failure. In *SOSP*.

[75] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *OSDI*.

[76] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. Lprof: A non-intrusive request flow profiler for distributed systems. In *OSDI*.

[77] Zilong Zhao, Sophie Cerf, Robert Birke, Bogdan Robu, Sara Bouchenak, Sonia Ben Mokhtar, and Lydia Y Chen. 2019. Robust anomaly detection on unreliable data. In *DSN*.

[78] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated test input generation for Android: Towards getting there in an industrial case. In *ICSE-SEIP*.

[79] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *ESEC/FSE*.