There's an app for that, but it doesn't work. Diagnosing Mobile Applications in the Wild

Sharad Agarwal Ratul Mahajan Alice Zheng Victor Bahl
Microsoft Research

Abstract— There are a lot of applications that run on modern mobile operating systems. Inevitably, some of these applications fail in the hands of users. Diagnosing a failure to identify the culprit, or merely reproducing that failure in the lab is difficult. To get insight into this problem, we interviewed developers of five mobile applications and analyzed hundreds of trouble tickets. We find that support for diagnosing unexpected application behavior is lacking across major mobile platforms. Even when developers implement heavyweight logging during controlled trials, they do not discover many dependencies that are then stressed in the wild. They are also not well-equipped to understand how to monitor the large number of dependencies without impacting the phone's limited resources such as CPU and battery. Based on these findings, we argue for three fundamental changes to failure reporting on mobile phones. The first is *spatial spreading*, which exploits the large number of phones in the field by spreading the monitoring work across them. The second is *statistical inference*, which builds a conditional distribution model between application behavior and its dependencies in the presence of partial information. The third is *adaptive sampling*, which dynamically varies what each phone monitors, to adapt to both the varying population of phones and what is being learned about each failure. We propose a system called MobiBug that combines these three techniques to simplify the task of diagnosing mobile applications.

Categories and Subject Descriptors
C.4 [Performance of systems] Reliability, availability, and serviceability
D.2.5 [Testing and debugging] Distributed debugging
C.2.4 [Distributed systems] Distributed applications
General Terms
Algorithms, Design, Measurement, Reliability
Keywords
Mobile applications, diagnosis

1. Introduction

A large number of applications are being written today for mobile phones. In just 3 years since its launch, the Apple iPhone App Store has over 200,000 applications. The Android market has over 50,000 applications. These applications are written by a variety of developers from hobbyists to experienced professionals.

Diagnosing unexpected application behavior is a major challenge for these developers today. They do not have low-level access to the OS and cannot easily learn conditions under which the application misbehaves; mobile OSes themselves have poor support for diagnosis (§2.1). Mobile applications operate in a highly dynamic environment, such as variable network connectivity. Without detailed information on the conditions under which the application misbehaves, it is difficult to reproduce misbehaviors that occur in the wild. We describe in §2.2 one case in which it took developers multiple weeks to reproduce and diagnose a crashing behavior.

The goal of this paper is to highlight and understand this problem. Since this domain is new and little is known about it today, we conducted two surveys to obtain detailed insight into the nature of the problem. The first survey (§2.2) consists of interviews with developers of five applications across two mobile platforms. The second survey (§2.3) is an analysis of hundreds of trouble tickets that detail failures experienced by users and actions taken by support engineers to diagnose them.

Our surveys confirm that the diagnostic support provided by current mobile OSes is limited. Many developers simply ignore the provided information and attempt to engineer their own systems. Due to the same limitation, support engineers find it tedious to diagnose failures that users experience.

The surveys also yield insights into the challenges in enabling effective diagnosis of mobile applications. First, mobile phones have fairly limited resources—they have limited processing and memory capacity, battery drain is a major issue, and network transfers over 3G can be expensive. Second, the dynamic conditions of the mobile environment make matters worse because some misbehaviors occur under uncommon conditions. Finally, because of the complexity of the environment, it is hard for developers to identify reliably all the factors on which the behavior of their application depends. These challenges are unfortunately in conflict. One way to get around the issues of rare, failure conditions and unknown dependencies is through continuous and detailed logging, but such logging can be very expensive for phone resources.

Current diagnostic systems, which have been designed for the PC or datacenter environments, are incapable of meeting these challenges. Most prior systems assume full availability of instrumented data, which is not feasible in this domain.

They use homogeneous instrumentation, both in terms of the type of data collected but also in the granularity of detail.

Based on observations from our surveys, we argue for three fundamental changes to failure reporting that, if implemented well, will significantly simplify mobile application diagnosis. The first is *spatial spreading*, by which monitoring is spread across phones, instead of each phone monitoring everything that it experiences. Such spreading can be achieved by directing dynamic instrumentation on each phone where each instrumentation point is turned on or off. The second is *statistical inference*, which allows us to deal with incomplete data by using a conditional distribution model between instrumentation points. The third is *adaptive sampling*, by which what a phone monitors changes with time as a dependency graph of conditions that the application behavior depends on is incrementally refined.

We outline a system, called MobiBug, that is designed around these three techniques. It treats monitoring as a coordinated global activity across phones, time and applications, and integrates it with the analysis itself. It can tune what a phone monitors based on available resources and what new information it is capable of contributing. The latter factor means that phones heavily represented in the population need to monitor less and that monitoring overhead for an application decreases with time.

2. Survey of current practices

The area of mobile application diagnosis is new and little is known about it today. Hence, we examine current practices to get detailed insights into the nature of the problem. Existing mobile OSes have built-in support to automatically log information about failures. We briefly summarize this support across 3 representative OSes in §2.1. We find that the information collected is limited in nature, and hence we survey 9 developers for 5 applications across 2 mobile OSes in §2.2 to understand how they cope. Finally, in some situations, debugging experts have the opportunity to interact directly with users while they are experiencing failures in the field. We analyze hundreds of trouble tickets describing such debugging sessions and summarize our findings in §2.3.

2.1 Diagnostics support in mobile OSes

Each time an application crashes on a phone, the mobile OS collects information about the crash. For example, on the Apple iPhone [1], this crash log includes:

- application name & version
- date & time of crash
- OS version
- exception type & error code

[1] Crashes on the iPhone are "silent" – the UI returns to the OS home screen without any other indication to the user that the application crashed. As a sample point of how often this occurs, on one author's iPhone 3Gs with 50 applications, 67 crash logs were collected over the last 6 months. The failing applications were both third-party ones including NPR Radio, Skype, Yahoo Messenger, Kindle and Facebook, but also those that shipped with the OS including Mail and Maps.

current practice	**app #1**	**app #2**	**app #3**	**app #4**	**app #5**
app. type	IM	search	backup	search	browser
mobile OS	WM 6	iPhone	WM 6	WM 6	WM 6
use crash logs from OS	yes	yes	yes	yes	yes
use custom logging	yes	yes	yes	yes	no
log app. performance	no	no	yes	yes	no
use custom logs in dogfood	yes	yes	yes	yes	no
use custom logs in release	no	no	no	no	no
transfer of custom logs	auto	manual	manual	manual	n/a
transfer is BW optimized	no	no	no	no	n/a
read user forums for details	yes	yes	yes	yes	no
failure analysis	manual	manual	manual	manual	mixed

Table 1: Summary of current practices across 5 mobile apps

- per-thread stack trace with function addresses & offsets
- version numbers of binaries that the application relies on

Android and Windows Mobile 6 collect almost the same information. While we have not examined all mobile OSes, we believe these three are representative of the state-of-the-art. The OSes do vary in how crash logs are transmitted. When an iPhone is connected to the desktop, iTunes collects these logs and uploads them to Apple. Android developers can use third-party crash reporting libraries [3, 2], but the upcoming Android 2.2 [1] will upload crash logs directly from the phone. Windows Mobile phones queue these logs and directly upload them to Microsoft when the phone is connected via Wi-Fi or USB. Logs are then available on-line to the application developer through iTunes Connect, or Android Market, or Windows Phone Developer Portal [4] respectively.

2.2 Developer interviews

To understand how developers use OS-generated crash logs and debug application failures, we interviewed 9 developers of 5 different applications for the iPhone and Windows Mobile. We briefly summarize our interview findings.

As Table 1 shows, developers for every application receive and examine crash logs from the field. Unfortunately, every team complained that the logs from the mobile OS are almost always insufficient to understand the nature of the failure. The included stack trace indicates what portion of the code failed, but gives little information about the environmental conditions that triggered the failure other than the hardware, OS and library versions.

Hence, many teams build custom logging. Using knowledge of their application, they identify parts of the environment that the application depends on (e.g. state of GPS). Logging is added to the application to periodically measure these dependencies. Unfortunately, they do not anticipate all the environmental conditions that impact their application. In subsequent debugging of failures, they uncover *some* of these unknown dependencies (e.g. status of network connections).

For all the applications, the developers adopted a two-stage release process. The application is first distributed among fellow employees. This stage, called "dogfood", allows developers to test the application beyond the lab. Then, after fixing any encountered bugs, the application enters the release stage where the public can download the application.

When custom logging is used, it is used exclusively during dogfood. The developers consider the performance overhead of logging to be acceptable for their fellow employees prior to release, but not for customers post-release. Most do not build

trouble tickets	type of problem	example
174	experienced failure	app. error message
99	app. developer question	what is the API to use GPS?
38	general question	how do I download updates?
34	feature requests	can you add email threading?
11	desktop software broken	re-install often solves problem

Table 2: Summary of 356 trouble tickets

automated transfers of custom logs back to them, and rely on educating their employees on how to manually transfer logs.

Without rich logging in public releases, many developers read user forums to learn about the conditions in which users experience failures, and sometimes contact those users to get more information or turn on logging on a case-by-case basis.

Developers for one of the applications described an example that is typical of their problematic debugging cases. They received negative feedback from users experiencing crashes. They found a common pattern—a specific phone model. The developers attempted to reproduce the failure in the lab by using the application extensively on that phone model. Weeks of testing yielded no failures until a developer inadvertently entered an elevator while using the application. Even with custom logging enabled, they were unable to find the culprit. They iteratively added logging for more environmental conditions and repeated their experiments. Eventually, they discovered that when the application downloads several web objects simultaneously, and the 3G signal rapidly degrades, the network stack on a particular driver or OS version on that hardware behaves unexpectedly.

2.3 Trouble ticket analysis

In the mobile ecosystem, engineers other than the developers can also be involved in diagnosing an application failure. The OS vendor, hardware manufacturer or mobile operator can offer product support to customers who can call in about any problems they are experiencing with their phones. We obtained a large corpus of trouble tickets from May 2008 for an organization that provides such support, and we now briefly summarize our analysis of it. During this period, support was limited to the Windows Mobile platform.

When a user calls in, the support engineer will document the trouble ticket in free form text. It includes transcriptions of voice conversations, emails, IMs, and sometimes the engineer's summary of the problem, cause and solution. We picked tickets at random and identified 356 that involve mobile phones and read through each one. Table 2 summarizes the type of problems users called about. 174 tickets involve failures they experienced. Of these, 74 are incomplete—the ticket is truncated or the mobile OS was re-installed without much debugging. We now focus on the remaining 100 tickets.

Figure 1 describes the debugging process in a typical trouble ticket. It is often lengthy and encompasses multiple days of phone, email and IM conversations. Early in the course of debugging, the main task is to identify the problem and the relevant support engineer, such as an email expert or a web expert. Once that engineer takes over the ticket, she will go through an iterative debugging process with the user.

The key challenge for the support engineer is to identify

- Reboot phone
- Email sync still fails
- Restart IIS server
- Email sync still fails
- Restart Exchange server
- Email sync still fails
- Create dummy account on Exchange server
- Setup phone emulator, configure it, attempt sync
- Email sync successful on emulator
- Delete email account on phone and reboot phone
- Recreate email account settings on phone
- Email sync still fails
- Check version of OS on mobile phone; does not match emulator
- Setup new emulator with that exact OS version
- Email sync fails on emulator
- Check server certificates
- Disable all authentication on server, reboot server
- Email sync succeeds
- Check mobile OS documentation on authentication
- Configured authentication type not supported on that mobile OS version
- Re-enable all authentication on server, reboot server

Figure 1: Typical debugging process after problem identification and re-routing to relevant engineer. Mobile phone is not syncing email with server and throwing a vague error message. Other phones that connect to the server work.

trouble tickets	symptom
73	Opaque error message
10	Unreachability
10	User action fails
2	Hang or crash
1	Out of memory
1	Drop in battery life
1	Phone slowdown
1	Phone beeping
1	Phone able to access disabled account

Table 3: Failure symptoms in 100 trouble tickets

the cause of the problem. Unfortunately, as we summarize in Table 3, often the error message thrown by the application or the OS is opaque – it provides little insight to the engineer. In fact, we found very different problem causes and solutions for similar error messages in these trouble tickets. This problem spanned multiple applications in our trouble tickets, and we argue is not unique to this OS – quite often, components of the OS or application or application's execution platform do not provide descriptive enough error messages that get propagated up to a reporting mechanism. This lack of information results in the engineer going through a process of elimination to find the culprit, as in Figure 1.

3. Goals & challenges

From our survey of current practices, we find that mobile OSes provide little information on application crashes. The stack trace and software and hardware versions do not identify many conditions that can contribute to failures, such as CPU utilization, state of network connections, software configuration, and status of sensors such as GPS. As a result, we find that both developers and support engineers largely ignore OS-generated crash logs. Developers attempt to build their own solution, while support engineers go through a tedious process of culprit elimination.

Our goal is to identify detailed information on conditions that lead to unexpected behavior in mobile applications. These conditions can be used by the developer to reproduce the fail-

system	dynamic or sampled instrumentation	dependency learning
Aguilera et al. [5]	✗	✓
Cohen et al. [12]	✗	✗
PinPoint [10]	✗	✗
Magpie [8]	✗	✗
CBI [15]	✓	✗
Holmes [11]	✓	✗
ABI [7]	✓	✗
Live Monitoring [14]	✓	✗
Paradyn [17]	✓	✗

Table 4: Comparison of prior work on failure diagnosis

ure in a lab, and sometimes eliminate or even identify possible culprits.

There are two main challenges to solving this problem. Developers are unable to collect all the relevant information without impacting the limited resources on phones. Even if they ignore this impact in specific situations, such as in the lab, they are unable to identify all the possible conditions that their application's behavior depends on. These unknown dependencies impact their application in the field because it is far more dynamic than the lab environment.

There are many requirements for an ideal solution. The combination of these two challenges requires the solution to learn dependencies automatically with limited data. For ease of failure reproducibility, a minimal set of contributing conditions should be identified. These conditions should be identified quickly so that the developer can patch the application. The solution should handle heterogeneous sources of information, including static conditions (e.g., OS version, GPS type), dynamic conditions (e.g., button clicks, CPU load, signal strength) and application behavior (e.g., crashes, performance counters).

4. Prior work on failure diagnosis

Although diagnosing mobile applications is a new domain, there is prior diagnosis work in other computer systems. However, as we summarize below and in Table 4, prior work does not sufficiently address the challenges and requirements of the mobile platform—monitor heterogeneous sources of information, handle limited phone resources by adapting instrumentation granularity and with dynamic or sampled instrumentation, and learn the unknown dependencies between instrumentation sites.

Most prior work uses fairly homogeneous instrumentation, i.e., information from one source. For example, Magpie [8] and DustMiner [13] deal solely with event streams, whereas statistical debugging [15, 16, 6] monitors predicates within single programs. In our domain, we need to examine the interaction between the application and mobile platform, which requires a set of very heterogeneous instrumentation sites across applications, OS and hardware, such as hardware models, button clicks, CPU load, signal strength and GPS state.

Even with homogeneous instrumentation, most systems operate at a fixed granularity between coarse to extremely fine grained. Coarse-grained instrumentation is cheaper to collect and more efficient to analyze, but does not yield as many clues as fine-grained instrumentation. Fine-grained instrumentation, on the other hand, generates a barrage of data, which turns root cause diagnosis into an extremely challenging needle-in-the-haystack problem. Diagnosis in the mobile domain demands the ability to adaptively adjust instrumentation granularity so as to avoid unnecessary overhead. In prior work, Pinpoint and others [10, 5, 9, 19] instrument the system at the component or sub-component level. Magpie [8] collects fine grained event information from both the kernel and user applications. Yuan [20] instruments system calls on a single desktop system, and Live Monitoring [14] uses data consistency checks within the program.

Most prior systems assume full availability of instrumented data and are poorly positioned to deal with mobile environments with limited resources. There are some recent works on adaptive instrumentation at the single-program level. Holmes [11] and Adaptive bug Isolation [7] dynamically collect intraprocedural path profiles from running programs. Live Monitoring [14] instruments AJAX-based web programs and is able to perform dynamic data structure consistency checks. Paradyn [17] dynamically instruments software binaries in order to debug common performance problems. These work are closer in spirit to what we wish to achieve, but none of them are designed to address resource limitations of the mobile domain.

5. Proposed solution: MobiBug

Despite the challenges that the mobile environment introduces, it provides opportunities that we can exploit. We propose first steps in a system called MobiBug to solving this problem. While there are multiple ways to build such a system, we argue for a few fundamental changes to failure reporting in mobile platforms.

5.1 Overview

MobiBug is a single platform for diagnosing all applications on a mobile OS. It is operated by the OS vendor, just as error reporting today. Such a platform reduces the application developer effort and limits the overhead on a phone by sharing common monitoring tasks among different applications. It also allows the mobile OS vendor to offer more effective product support to users. While we do not consider privacy issues in our design, we believe that the impact can be mitigated by restricting access to personally-identifiable information such as the phone number or IMEI, which are unlikely to be useful in diagnosis anyway.

In MobiBug, client modules on mobile phones measure data on application behavior and the environment. A centralized server sends instrumentation directives to these clients to measure specific pieces of data. The server is responsible for diagnosis, learning failure signatures, and generating new instrumentation directives. Application developers interface with the MobiBug server to obtain raw logs uploaded by phones as well as diagnosis findings computed automatically by MobiBug . Figure 2 provides a diagram of the various components.

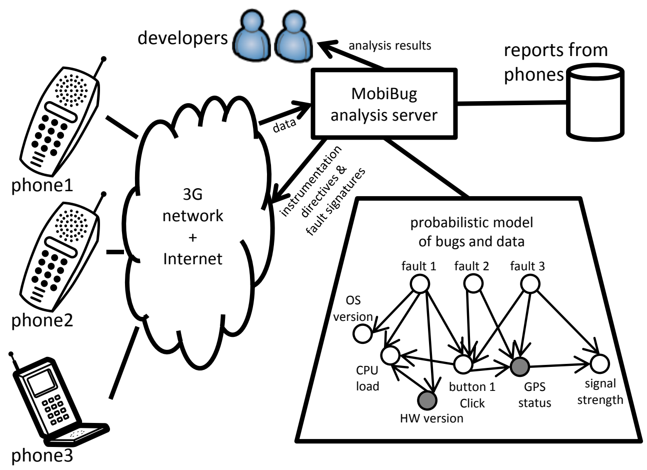

Figure 2: *Overview of* MobiBug

5.2 Reducing measurement overhead using spatial spreading

While individual phones have limited resources to measure all the data necessary for failure diagnosis, large numbers of each type of phone are sold on the market. There are a small number of phone manufacturers. Retail stores and users have little ability to modify the hardware. The OS that the mobile operator or phone manufacturer releases is difficult for end users or applications to modify. Thus there is a limited universe of unique phone environments that an application runs in, each with many instances. The opportunity is to spread the overhead of collecting diagnosis information across all these instances. We use this technique of *spatial spreading* in MobiBug to reduce the impact on each phone.

To achieve spatial spreading, we rely on *dynamic instrumentation*. The MobiBug server directs different phones to collect different information on different instrumentation sites in order to refine the diagnosis for previously un-seen failures. When a phone uploads measured data, the server performs signature matching against known failures. It collates the directives for each failure, and sends a random subset to the phone to monitor. The MobiBug client then dynamically updates instrumentation and maintains the measurements in a circular buffer.

Due to the variety of conditions that an application's behavior can depend on, MobiBug uses *heterogeneous instrumentation* sites to collect relevant information. It is heterogeneous in that very different types of information are collected across the OS, hardware and applications. Static or slow-changing information such as hardware platform and OS version need only be collected once during the initial sync and updated upon change. Dynamic information such as CPU load and network connection state are collected on an on-going basis, depending on directives from the MobiBug server.

Transferring large amounts of collected data to the MobiBugserver can be detrimental to the mobile operator's network and to the user's phone bill. Spatial spreading reduces the burden on each phone's network usage. Further, if the MobiBug client successfully matches the collected data against signatures for diagnosed failures downloaded from the server, then this data is not uploaded. For undiagnosed failures, the signature indicates the urgency of the bug – if it is non-critical (e.g. does not affect the user experience), the client will delay upload until the phone is tethered with a desktop PC.

5.3 Making sense of incomplete data using statistical inference

MobiBug compensates for limited resources on the phone by placing most of the analytical burden on the server. The snippets of incomplete data that each phone is measuring require more sophisticated computation to analyze. There are three main computational tasks for the server: filling in missing data, formulating and matching against failure signatures, and generating further instrumentation directives.

Statistical inference is a general tool to fill in missing data values and help guide diagnosis. It requires first learning a probabilistic model [18] of the data that captures the structure as well as parameters of conditional distributions between instrumentation points. Given such a model, standard probabilistic inference machinery can then tell us the probability of state z given observed values for variables X and Y. We illustrate this in §5.6 using the elevator bug as an example.

5.4 Refining the dependency graph using adaptive sampling

Building the probabilistic dependency graph can take a long time with incomplete data. To speed this process, we rely on *adaptive sampling*. First, some dependencies can be manually specified using developers' domain knowledge (e.g., the behavior of my map application depends on the GPS driver version). As we found from our survey of developers, they are often unable to specify all dependencies but are willing to tolerate measurement load during dogfood. Hence, second, we use heavy instrumentation during lab testing and dogfood to collect data to discover more dependencies. Finally, during release, the measurement overhead on each phone is restricted to avoid noticeable impact.

However, the user environment is more dynamic than either the lab or dogfood. Previously discovered dependencies may not be valid in the wild, or previous in-dependencies may be actual dependencies. MobiBug changes over time what each phone measures to validate existing (in)dependencies. The MobiBug server will occasionally send "hypotheses" of conditional (in)dependence and expected probabilities to the client. The client will then spend a small amount of resources to monitor those sites and check that the values are within the expected range. If the observed data that the client uploads consistently falls below a likelihood threshold under the current model, then the server must update or re-learn the model.

5.5 Failure diagnosis

There are two basic steps to failure diagnosis in MobiBug. We first match collected data against known failures. If the failure has not yet been fully diagnosed, we then generate directives for further instrumentation to help diagnose this failure.

Given the incomplete list of failure symptoms collected from the client, MobiBug's job is to determine whether they match against a known failure, or whether this is an instance of a new and previously unknown failure. In the latter case, the model needs to be expanded, and additional probes performed to distinguish the new failure from existing failures. As described earlier, we use adaptive sampling to change dynamic instrumentation on clients, thereby allowing MobiBug to learn additional symptoms that could distinguish failures.

Once unique failures have been identified, we merge all data from a failure and infer its dependency graph. This dependency graph contains the conditions on which the failure depends and is returned to the developer.

5.6 Example: The Elevator Bug

We now use the elevator bug from §2.2 in an example of how MobiBug might work. First, developers create a MobiBug profile for their application, and include any known dependencies. They suspect that the network type (3G, Edge, Wi-Fi, etc.) may be a factor. During dogfood, MobiBug aggressively collects data from participants' phones, on both application behavior (e.g., application specific counters such as number of active and completed requests, and error codes from various components) as well as the phone environment. It will collect the network type more frequently, but with a smaller probability still explore other dependencies. So one phone may be collecting the network type, CPU load, and speaker volume while the application is running. Another phone may collect the network type, speaker volume, and the network flow table.

Using this dataset, MobiBug builds the first version of its dependency graph, where it may decide that there is a probabilistic dependency between the application's behavior and CPU load, but has observed no other dependencies on the environment. During public release, MobiBug will primarily monitor the CPU load with application behavior, but with a very small probability explore other dependencies it had previously ruled out.

Suppose some phones observe error code Y that is uniquely associated with the elevator bug. Given Y, MobiBug looks at its dependency graph to determine additional features to monitor (if few are known, then this initial list may be randomly chosen). MobiBug uses this data to update its dependency graph. For instance, it should already know about the relationship between network type and number of completed network requests. But it may discover a previously unknown relationship between the number of active network downloads, the network signal strength, phone OS version, and error code Y. Each update to the structure of the probabilistic model should serve as a clue to the tester about the behavior of this bug.

6. Summary

Diagnosing unexpected application behavior on phones is difficult even for seasoned developers. Our survey of developers uncovered a need for more detailed information than what modern mobile platforms provide. A key challenge for them is collecting data on the application's behavior and its environmental conditions while having limited CPU, battery and network impact on the phone. In addition, they are unable to identify a priori all the possible dependencies for their application, partly because the lab environment is far less dynamic than users' mobile phones in the wild. Our analysis of trouble tickets finds that due to the lack of information, support engineers engage in a lengthy process of elimination of numerous possible culprits of failures.

While application diagnosis is a well-studied area in the desktop and enterprise domain, our findings open up this area into a new domain with unique challenges. We argue for fundamental changes to failure reporting in order to diagnose mobile application misbehavior. We do so by proposing the design of MobiBug around three techniques. Spatial spreading allows us to overcome the limited capabilities of each phone by sharing the overhead of monitoring across all the phones in the wild. Statistical inference can be used to make sense of this incomplete data to build dependency graphs. Adaptive sampling allows us to incrementally refine dependency graphs by changing what each phone monitors over time to test portions of it. We argue that if implemented well, these techniques will significantly reduce the problem of mobile application diagnosis.

7. References

[1] Android Developers Blog: Android Application Error Reports. http://android-developers.blogspot.com/2010/05/google-feedback-for-android.html.

[2] Application Crash Report for Android. http://code.google.com/p/acra/.

[3] Remotely log unhandled exceptions in your Android Applications. http://code.google.com/p/android-remote-stacktrace/.

[4] Windows Phone Developer Portal. http://developer.windowsphone.com.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SOSP*, 2003.

[6] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *ECML*, 2007.

[7] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In *ICSE*, 2010.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[9] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[10] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[11] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[12] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, 2004.

[13] M. Khan, H. K. Le, H. Ahmadi, T. Abdelzaher, and J. Han. DustMiner: Troubleshooting Interactive Complexity Bugs in Sensor Networks. In *Sensys*, 2008.

[14] E. Kiciman and H. J. Wang. Live monitoring: Using adaptive instrumentation and analysis to debug and maintain web applications. In *HotOS XI*, 2007.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[17] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE COMPUTER*, 1995.

[18] K. Murphy. An introduction to graphical models. 2001.

[19] R. R. Sambasivan, A. X. Zheng, E. Thereska, and G. R. Ganger. Categorizing and differencing system behaviours. In *HotAC II*, 2007.

[20] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.