1

Introduction

There are no significant bugs in our released software that any significant number of users want fixed.

–Bill Gates quoted in FOCUS Magazine

Real software is buggy. Real users can make it better. Cooperative Bug Isolation (*CBI*) seeks to leverage the huge amount of computation done by the end users of software. By gathering a little bit of information from every run of a program performed by its user community, we are able to make inferences automatically about the causes of bugs encountered in the field.

1.1 Perfect, or Close Enough

Many computer scientists think of a program as either correct (i.e., it meets some specification) or incorrect (i.e., it does not meet some specification). But industrial software development is as much about economics as computer science. Software quality is a monetary balancing act among engineers' salaries, time to market, user expectations, and other business concerns. We ship software when it seems correct enough to neither embarrass us nor alienate users. We ship software with known bugs that are not worth fixing, and users uncover new bugs that developers never imagined. An observer in residence at the game development studios of Electronic Arts (EA) wrote that

> The largest sin at EA is not delivering a title on time. ... Making an outstanding game, but delivering it late, is not as profitable as making an acceptable quality game on time. EAers talk about "maximum on-time-quality." [52]

Clearly practitioners use something other than a Boolean notion of correctness, but such a notion has been difficult to quantify. In-house testing can only guess at field usage patterns, and poor guesses can leave users in bad shape. A seemingly obscure, low-priority bug that was difficult to reproduce in the testing lab may turn out to affect large numbers of users on a regular basis. Technical support channels provide one way for post-deployment feedback to reach engineers, but traditionally these mechanisms have been informal and overly dependent on human intervention.

B. Liblit: Cooperative Bug Isolation, LNCS 4440, pp. 1–6, 2007.

1.2 Automatic Failure Reporting

Industry critics have said that many software vendors treat their customers like beta testers. If that is so, then we are not yet using these thousands or millions of testers as effectively as we could. Traditionally, most software failures produce a grumpy user and no diagnostic feedback, which benefits no one.

This situation is starting to change as a result of ubiquitous Internet connectivity. KDE, GNOME, Mozilla/Netscape, and Microsoft have all deployed automated, opt-in crash reporting systems. These systems gather key information about program state after a failure has occurred: stack trace, register contents, and the like. By sending this information back to the development organization, the user community helps developers effectively triage bugs that cause crashes and focus on the problems experienced by the most users.

However, automatic crash reporting systems create a new problem: developers who are overwhelmed with bug reports, many of which may be redundant, and who must prioritize their work in terms of which bug fixes are likely to provide the greatest net benefit in the shortest amount of time. As of this writing, the Bugzilla bug tracking database for the open source Mozilla web browser project shows 58,661 open bugs; an additional 104,764 have been marked as duplicates of bugs already reported [44]. Mozilla augments manual bug reporting with an automated crash feedback system. This system currently shows 186,180 automated crash reports for Mozilla Firefox 2.0 over a ten day period, accounting for 5,024,104 hours of "testing" by end users [45]. As early as 2002, Microsoft's Watson error reporting service had collected crash reports from half a million separate programs. Experience with Watson has shown that one percent of software errors cause fifty percent of user crashes [41].

We believe that ubiquitous crash reporting is progress in the right direction, but we also believe that existing approaches only scratch the surface of what is possible when developers and users are connected by a network. For example, crash reporting systems in mainstream end user environments do not gather any information about what happened before the crash. Trace information leading up to the failure may contain critical clues to the actual problem. Also, crash reporting systems report no information for successful runs, which makes it difficult to distinguish anomalous (crash-causing) behavior from innocuous behavior common to all executions. In general, the information gathered by crash reporting systems is not very systematic, and in all feedback systems of which we are aware (crash reporting or otherwise) the subsequent data analysis is highly manual.

1.3 The Next Step Forward

The high level of redundancy exhibited by existing crash reporting systems suggests that there is great potential to harness the user community as a distributed, brute force bug hunting resource. Because the most important bugs are those that happen most often to the most users, it is not necessary to trace program behavior in a complete,