Lecture Notes in Computer Science 4440

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Ben Liblit

Cooperative Bug Isolation

Winning Thesis of the
2005 ACM Doctoral Dissertation Competition

Springer

Author

Ben Liblit
University of Wisconsin–Madison
Computer Sciences Department
1210 West Dayton Street, Madison, WI 53706-1685, USA
E-mail: liblit@cs.wisc.edu

Library of Congress Control Number: 2007924082

CR Subject Classification (1998): D.2.4-5, D.2.2, F.3, F.2.2, I.2.8

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-71877-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-71877-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Markus Richter, Heidelberg
Printed on acid-free paper SPIN: 12046706 06/3180 5 4 3 2 1 0

To my parents, for their unwavering love and support throughout this and all my adventures

Foreword

Efforts to understand and predict the behavior of software date back to the earliest days of computer programming, over half a century ago. In the intervening decades, the need for effective methods of understanding software has only increased; software has spread to become the underpinning of much of modern society, and the potentially disastrous consequences of broken or poorly understood software have become all too apparent. Ben Liblit's work reconsiders two common assumptions about how we should analyze software and it arrives at some striking new results.

In principle, understanding software is not such a hard problem. Certainly a computer scientist studying programs appears to be in a much stronger position than, say, a biologist trying to understand a living organism or an economist trying to understand the behavior of markets, because the biologist and the economist must rely on indirect observation of the basic processes they wish to understand. A computer scientist, however, starts with a complete, precise description of the behavior of software—the program itself! Of course, the story turns out not to be so straightforward, because despite having a perfect description, programs are sufficiently complex that it is usually difficult or even impossible to answer many simple questions about them. Ben's first change of assumption comes from the observation that if programs are hard to understand, perhaps we could make use of some of the tools that biologists and economists use to understand their complicated systems: maybe it would be productive to regard programs as statistical processes and use statistical techniques to understand software. We can simply run the program, make some observations and, if we ask the right questions, learn something useful about program behavior. What questions to ask, and how to answer them, is the topic of the second half of this book.

The second key ingredient comes from asking the question: Which program runs should we use to gather the observations? Using test cases or automatically synthesized inputs is a bit unsatisfying, as these executions may not be representative of the reality of what users do with the software. And therein lies the answer: Use the runs of the program's users. These runs define the reality of how the software behaves in practice; in a real sense, these are the executions that matter. In a networked world it is possible to gather a small amount of information from every execution ever per-

formed and from that information build up a model of program behavior about which statistically meaningful statements can be made. How to actually gather that information in a way that is unobtrusive and efficient as well as statistically sound is the subject of the first half of this book.

The centerpiece of the monograph is an algorithm for isolating multiple bugs from sparsely sampled data taken from many thousands of program executions. The basic idea is to see which program events are strongly correlated with a subset of program failures, remove those failures from consideration and then, recursively, compute what events are correlated with the remaining failures. This algorithm has unique properties that complement other program analysis techniques; in particular, it is potentially able to find the root cause of any program failure without first requiring an explicit specification of the property to check. While Ben's work focuses on finding the causes of bugs, the underlying approach is much more general and should be adaptable to any program-understanding problem where one wants to discover which program events are strongly correlated with some observable behavior. The results Ben presents represent a new and fundamental approach to software analysis and should provide a source of ideas and inspiration to the field for many years to come.

January 2007 Alex Aiken

Preface

This book contains a revised version of the dissertation the author wrote in the Computer Science Division of the Department of Electrical Engineering and Computer Science of the University of California, Berkeley. The dissertation was submitted to the Graduate Division in conformity with the requirements for the degree of Doctor of Philosophy in December 2004. It was honored with the 2005 ACM Doctoral Dissertation Award in May 2005.

Abstract

Debugging does not end with deployment. Static analysis, in-house testing, and good software engineering practices can catch or prevent many problems before software is distributed. Yet mainstream commercial software still ships with both known and unknown bugs. Real software still fails in the hands of real users. The need remains to identify and repair bugs that are only discovered, or whose importance is only revealed, after the software is released. Unfortunately, we know almost nothing about how software behaves (and misbehaves) in the hands of end users. Traditional post-deployment feedback mechanisms, such as technical support phone calls or hand-composed bug reports, are informal, inconsistent, and highly dependent on manual, human intervention. This approach clouds the view, preventing engineers from seeing a complete and truly representative picture of how and why problems occur.

This book proposes a system to support debugging based on feedback from actual users. *Cooperative Bug Isolation* (CBI) leverages the key strength of user communities: their overwhelming numbers. We propose a low-overhead instrumentation strategy for gathering information from the executions experienced by large numbers of software end users. Our approach limits overhead using sparse random sampling rather than complete data collection, while simultaneously ensuring that the observed data is an unbiased, representative subset of the complete program behavior across all runs. We discuss a number of specific instrumentation schemes that may be coupled with the general sampling transformation to produce feedback data that we have found to be useful for isolating the causes of a wide variety of bugs.

Collecting feedback from real code, especially real buggy code, is a nontrivial exercise. This book presents our approach to a number of practical challenges that arise in building a complete, working CBI system. We discuss how the general sampling transformation scheme can be extended to deal with native compilers, libraries, dynamically loaded code, threads, and other features of modern software. We address questions of privacy and security as well as related issues of user interaction and informed user consent. This design and engineering investment has allowed us to begin an actual public deployment of a CBI system, initial results from which we report here.

Of course, feedback data is only as useful as the sense we can make of it. When data is fair but very sparse, the noise level is high and traditional manual debugging techniques insufficient. This book presents a suite of new algorithms for *statistical debugging*: finding and fixing software errors based on statistical analysis of sparse feedback data. The techniques vary in complexity and sophistication, from simple process of elimination strategies to regression techniques that build models of suspect program behaviors as failure predictors. Our most advanced technique combines a number of general and domain-specific statistical filtering and ranking techniques to separate the effects of different bugs and identify predictors that are associated with individual bugs. These predictors reveal both the circumstances under which bugs occur and the frequencies of failure modes, making it easier to prioritize debugging efforts. Our algorithm is validated using several case studies. These case studies include examples in which the algorithm found previously unknown, significant crashing bugs in widely used systems.

Acknowledgments

First and foremost, I wish to thank my advisor Alex Aiken. The best way I know to show my gratitude is to aspire to be for my students what Alex has been for me: teacher, collaborator, mentor, guide . . . role model.

Alice Zheng has been my statistically minded other half in what has evolved into a deeply rewarding cross-disciplinary collaboration. Thank you, Alice. I couldn't have done it without you.

This project has easily produced ten bad ideas for every good one. Michael Jordan met all of our brainstorming with patient explanations and unmatched expertise. It has been a pleasure and an honor having Mike on the team.

Mayur Naik's creativity, energy, and seemingly boundless enthusiasm have made him a delight to work with. Mayur, it's been great fun just trying to keep up with you.

The path balancing optimization is based on ideas first suggested by Cormac Flanagan.

This research began under the auspices of the Open Source Quality Project, whose faculty, student, and staff members have been a valuable source of insight and inspiration over the years. Special thanks are due to George Necula and his students for creating and maintaining the CIL C front end upon which our instrumentor is based.

I am indebted to the many members of the open source community who have supported our work. My thanks go out to the many anonymous users of our public deployment, and to the developers of the open source projects used in our public deployment and case studies. I am especially grateful to Luis Villa, Jody Goldberg, and Colin Walters for their longstanding interest in and encouragement of this research.

While conducting this research, my collaborators and I were supported in part by DARPA ARO-MURI ACCLIMATE DAAD-19-02-1-0383; DOE Prime Contract No. W-7405-ENG-48 through Memorandum Agreement No. B504962 with LLNL; NASA Grant No. NAG2-1210; NSF Grant Nos. EIA-9802069, CCR-0085949, ACI-9619020, and IIS-9988642; ONR MURI Grant N00014-00-1-0637; a grant from Microsoft Research; and a Lucent GRPW Fellowship.

January 2007 Ben Liblit

Contents

1 Introduction 1
- 1.1 Perfect, or Close Enough 1
- 1.2 Automatic Failure Reporting 2
- 1.3 The Next Step Forward 2
- 1.4 Cooperative Bug Isolation 4

2 Instrumentation Framework 7
- 2.1 Basic Instrumentation Strategy 7
  - 2.1.1 Sampling the Bernoulli Way 8
  - 2.1.2 From Blocks to Functions 10
  - 2.1.3 Interprocedural Issues 11
- 2.2 Instrumentation Schemes for Distributed Debugging 12
  - 2.2.1 Issues in Remote Sampling 13
  - 2.2.2 Counter-Based Instrumentation Schemes 14
  - 2.2.3 Additional Instrumentation Schemes 18
- 2.3 Performance and Optimizations 19
  - 2.3.1 Static Branch Prediction 24
  - 2.3.2 Weightless Functions 25
  - 2.3.3 Empty and Singleton Regions 27
  - 2.3.4 Local Countdown Caching 27
  - 2.3.5 Random Countdown Generation 30
  - 2.3.6 Path Balancing 31
  - 2.3.7 Statically Selective Sampling 33
  - 2.3.8 Optimization Recap 35
- 2.4 Adaptive Sampling 35
  - 2.4.1 Nonuniformity Via Multiple Countdowns 35
  - 2.4.2 Nonuniformity Via Non-Unit Site Weights 36
  - 2.4.3 Policy Notes 37
- 2.5 Realistic Sampling Rates 37

3 Practical Considerations 39
3.1 Native Compiler Integration 40
3.1.1 Static Site Information 41
3.2 Libraries and Plugins 41
3.3 Threads 43
3.3.1 Next-Sample Countdown 43
3.3.2 Predicate Counters 43
3.3.3 Compilation Unit Registry and Report File 44
3.3.4 Time Stamp Clock 44
3.3.5 Performance Evaluation 44
3.4 Privacy and Security 44
3.5 User Interaction 46
3.6 Status of the Public Deployment 49
3.6.1 Resource Requirements 49
3.6.2 Reporting Trends 50

4 Techniques for Statistical Debugging 55
4.1 Notation and Terminology 55
4.2 Predicate Elimination 56
4.2.1 Instrumentation Strategy 56
4.2.2 Elimination Strategies 57
4.2.3 Data Collection and Analysis 58
4.2.4 Refinement over time 59
4.2.5 Performance Impact 59
4.2.6 Limitations and Insights 60
4.3 Regularized Logistic Regression 61
4.3.1 Crash Prediction Using Logistic Regression 61
4.3.2 Data Collection and Analysis 63
4.3.3 Performance Impact 66
4.3.4 Limitations and Insights 67
4.4 MOSS: A Multiple-Bug Challenge 67
4.4.1 Nonuniform Sampling 70
4.4.2 Analysis Results 70
4.5 Iterative Bug Isolation and Elimination 71
4.5.1 Increase Scores 72
4.5.2 Statistical Interpretation 74
4.5.3 Balancing Specificity and Sensitivity 75
4.5.4 Redundancy Elimination 79
4.6 Case Studies 81
4.6.1 MOSS 81
4.6.2 CCRYPT 84
4.6.3 BC 84
4.6.4 EXIF 85
4.6.5 RHYTHMBOX 86

5 Related Work 89
5.1 Static Analysis 89
5.2 Profiling and Tracing 90
5.3 Dynamic Analysis 91

6 Conclusion 95

References 97