What To Do When Things Go Wrong: Recovery in Complex (Computer) Systems

Martin Rinard
MIT EECS, MIT CSAIL
rinard@csail.mit.edu

Abstract

We present and analyze a range of techniques for recovering from faults in complex hardware and software systems, from classical techniques that attempt to preserve the abstraction of perfection in the presence of faults to emerging techniques that adapt application functionality to transcend faults, overcome implementation errors in both the hardware and software, and adapt to the characteristics of the underlying execution environment.

Categories and Subject Descriptors D2.5 [*Testing and Debugging*]: Error Handling and Recovery

General Terms Recovery, Fault, Error

Keywords Recovery, Fault, Error

1. Abstract

Modern hardware and software systems are the most complex artifacts humans have ever built. Modularity enabled by digital interfaces that satisfy precise logical specifications has been a key prerequisite that has enabled us to build such complex systems. The inevitable presence of faults has always complicated the construction of modules that can satisfy their specifications in a wide range of environments. The standard goal has been to confine the effect of each fault within the module in which it occurs so that clients are oblivious to the fault and the system as a whole continues to execute without perturbation. We note in passing that (because, at least, of the basic physical characteristics of the devices and environments in which the systems are deployed) this goal is realizable only for some classes of faults. So the goal has been to obtain modules that satisfies their specification often enough so that an engineer using a module can productively believe that the module satisfies its specification (even though it may sometimes not).

We start by surveying a range of standard techniques that developers and engineers have developed for masking faults. Examples of these techniques include error-correcting codes, checkpoint followed by rollback and replay [8, 9, 16] and redundant computation. A critical component of many of these techniques is the mechanism for detecting faults. At this point these techniques are well understood, have been deployed widely in hardware and software systems, and are critical to ensuring the acceptable operation of these systems.

We then note that many faults are inherently unmaskable with traditional techniques. Consider, for example, a system that must provide output at regular time intervals. If the system loses part of the hardware resources that it needs to produce the result on time, traditional techniques do not aspire to change the computation so that it can produce the result with less resources. Another example is a software system with security vulnerabilities or implementation errors that are triggered by specific carefully crafted inputs. While it may be possible to monitor the system to detect the exploit or error, traditional techniques do not aspire to eliminate the vulnerability or error, mask the fault, and produce the correct result. One traditional response to the detection of such faults has been to stop the system to avoid potentially unanticipated negative consequences of a system that is operating outside of its anticipated operating envelope.

We next consider a more recent set of emerging techniques for surviving faults. Unlike most classical techniques, these techniques do not aspire to produce modules or systems that always satisfy their natural specifications. The goal is instead to ensure that the system respects key acceptability, correctness, or sanity conditions even in the face of faults that may be inherently unmaskable, unmaskable with traditional techniques, or even just too impractical or expensive to mask. Examples of such techniques include acceptability-oriented computing [17], task skipping [18], failure-oblivious computing [19], transactional function termination [21, 22], exiting infinite loops [3], automatic patch generation [15, 25], software rejuvenation [24], recovery-

AOSD'12, March 2530, 2012, Potsdam, Germany.
ACM 978-1-4503-1222-6/12/03.

oriented computing [2], heap overprovisioning [1], input rectification [11], and data structure repair [5–7, 12, 20, 26]. Because such techniques may have effects that easily propagate across module boundaries, understanding their acceptability often requires the adoption of an end-to-end perspective that takes the larger goals of the complete system into account.

Finally, we consider the emerging field of approximate computing. Inspired by the surprising ability that many software systems exhibit to tolerate perturbations or even large-scale changes to their execution (for example, loop perforation [13, 23], which simply skips iterations of time-consuming loops), researchers are now exploring a variety of techniques that purposefully adapt the functionality that a system provides to obtain benefits such as improved performance, reduced energy consumption, or robustness [4, 10, 14, 27]. The initial success of these techniques suggests that it may now be possible to move toward a world in which the dominant software systems exhibit unprecedented flexibility, malleability, and adaptability without requiring engineers to understand precisely how or why they are able to do so.

References

[1] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI*, pages 158–168, 2006.

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI*, pages 31–44, 2004.

[3] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *ECOOP*, pages 609–633, 2011.

[4] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. NavidPour. Proving programs robust. In *SIGSOFT FSE*, pages 102–112, 2011.

[5] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

[6] B. Demsky and M. C. Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, pages 176–185, 2005.

[7] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[8] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

[10] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. C. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, pages 199–212, 2011.

[11] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *ICSE*, 2012.

[12] M. Z. Malik, J. H. Siddiqui, and S. Khurshid. Constraint-based program debugging using data structure repair. In *ICST*, pages 190–199, 2011.

[13] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard. Quality of service profiling. In *ICSE (1)*, pages 25–34, 2010.

[14] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, pages 316–333, 2011.

[15] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102, 2009.

[16] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP*, pages 235–248, 2005.

[17] M. C. Rinard. Acceptability-oriented computing. In *OOPSLA Companion*, pages 221–239, 2003.

[18] M. C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, pages 324–334, 2006.

[19] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[20] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[21] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference, General Track*, pages 149–161, 2005.

[22] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS*, pages 37–48, 2009.

[23] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[24] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Trans. Dependable Sec. Comput.*, 2(2):124–137, 2005.

[25] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5), 2010.

[26] R. N. Zaeem and S. Khurshid. Contract-based data structure repair using alloy. In *ECOOP*, pages 577–598, 2010.

[27] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.