Summary of Research, Teaching, and Innovation Philosophy
Armando Fox

Research Summary

Cheap Recovery + Statistical Monitoring

Dependability is one of the most pressing problems facing systems designers and operators today. Despite improved development processes and tools, all production-quality software has difficult-to-reproduce bugs, including Heisenbugs, race conditions, resource leaks, and environment-dependent bugs, all of which can trigger prolonged outages when they strike live systems. Faced with failures, operators’ main concern is rapid recovery. To make things worse, due to high line counts, rapid churn, and the limited impact and cultural resistance to “correct-by-design” approaches, our understanding of these complex systems of black boxes is poor except at the coarsest grain, impeding both diagnosis and repair.

We address this challenge with two key innovations. First, since availability is defined as MTTF/(MTTF+MTTR), improvement through reducing MTTR is just as valid as through lengthening MTTF, and is an under-explored area where there may be opportunities for real contributions. Second, we use machine learning techniques to characterize the “normal” behavior of complex systems and then detect possible problems by looking for anomalous behavior; since achieving the highest detection rates with such techniques results in some false positives, we architect systems to expose low-cost “do no harm” recovery techniques that are always safe to try, which is synergistic with our first goal of lowering MTTR.

We find that if recovery is made sufficiently cheap, it leads to a change in thinking about systems from “normal-mode vs. recovery-mode” to “always adapting, always recovering”. We provide a simple invariant that can be exploited by any recovery policy: rebooting a whole node or a fine-grained set of software components, though not guaranteed to solve the problem, is always safe to try—it results in no loss of correctness and modest but tolerable loss of performance. This simplifies recovery policy and allows for aggressive failure monitoring based on anomaly detection algorithms, enabling the application of a vast existing literature on pattern recognition, data mining/clustering, etc. to the problem of dependability. The effect on end users is that most transient failures are perceived as additional latency ranging from hundreds of milliseconds to a few seconds, well below established thresholds for user-perceptible service quality. The idea of “always safe to reboot” is also incorporated into iROS, a software infrastructure for interactive ubiquitous computing that balances flexibility and robustness as first-class goals.

Reducing MTTR by 1–2 orders of magnitude

Our position paper and initial proof-of-concept [4, 2] argued for rebooting as a well-understood recovery mechanism with salubrious properties, and proposed that the disadvantages associated with rebooting (long latency and possible state loss leading to correctness problems) should be addressed by rethinking system design to tolerate rebooting better. This resulted in design guidelines for “crash-only software” [5]: exploit component boundaries to provide localized recovery (as we did in adding “microreboots” to an open-source J2EE server [6]), and protect application-level state that must survive reboots by putting it in a state store that is itself optimized for fast safe reboot-based recovery [14]. Our prototype uses “microreboots” of subsets of components to recover from most reboot-curable failures 1–2 orders of magnitude faster than a full reboot. In essence, our prototype can be managed as if it were a farm of stateless servers, techniques for which are well understood. (A similar system does the same for persistent hash tables [11] but hasn’t yet been integrated into our testbed.)

Application-generic Failure Detection and Localization

Our failure-analysis work relies on extracting information from a real running system. We have used data-clustering methods for localizing failures in arbitrary componentized applications [8]. To characterize the behavior of and detect possible low-level failures in arbitrary componentized applications, we used machine learning techniques such as decision trees and probabilistic context-free grammars to detect and localize otherwise elusive failures with no a priori application-specific knowledge [13]. These approaches detect and localize 89–96% of injected faults (depending on the fault type and the specific technique), compared to 20–70% for current techniques, and allow trading off detection rate against false positive rate; since we have

\[1\] Mean time to failure and mean time to recovery.
reduced recovery cost by 1–2 orders of magnitude through microrebooting, our microrebootable application server tolerates up to 97% false positives while still maintaining the same overall availability as reboot-based recovery, hence it makes sense to tune in favor of better detection rates [7].

To determine what to recover, we use dynamic fault injection to characterize failure propagation [3], resulting in more detailed and more accurate inter-component dependency graphs than could be obtained by readily-available static application descriptions.

Implications

Applying statistical learning theory (SLT) to extract system models is appealing not only as a way to guide automated recovery, but as a way to help human operators understand the behavior of the system and the impact of various kinds of failures [9]. The combination of these techniques with cheap recovery is not only a concrete step towards “self-managing” systems, but also may allow applying linear feedback control to system dependability as well. Control theory has a long history of success in physical systems whose behaviors are, happily, constrained by physical laws (e.g. “bounded input–bounded output”). In contrast, software systems have no such constraints other than those of the hardware on which it runs, and those typically bear little relevance to the way software actually fails. Further, the sensors and control inputs to which controllers of physical systems are attached seem absent from software: it is rarely the case that a single “low-level” property such as CPU load is a good predictor of a high-level property such as end-to-end server response time. However, we can use SLT to synthesize higher-order sensors and control inputs from dynamically-changing collections of lower-level sensors and “effectors”. For example, we can use data clustering to understand which requests are the likeliest contributors to an excessively-high workload, then use microrebooting as fine-grained load-shedding (control input) to repair the problem. In other words, such techniques potentially allow us to model these systems as control loops with reasonable forward gains and short feedback times. More interestingly, although control theory has been used in simple cases to optimize performance in complex servers, we have been able to use SLT techniques to effectively derive metrics of correctness based on low-level software “sensors”; this holds out the possibility of applying control loops to optimize for correctness (e.g., minimize deviation from known-correct behavior) rather than strictly performance.

The invariant of “always safe to try recovery” brings such systems a step closer to having a “physical control input”—indeed, it brings their behavior closer to dynamic adaptation than to traditional recovery. The application of SLT may make it possible to expose synthetic sensors that derive a measured property at a high level of abstraction from collections of low-level observations. A systematic methodology for combining these to apply control theory to complex software systems will be a definite step forward, and more generally, moving toward a “physics” for software will allow us to reason rigorously about the behavior of these systems to make them as dependable as society expects and deserves.

Robustness in interactive workspace design

I have also done a body of work in software infrastructure for interactive workspaces. The main goal of that work was to design a software infrastructure with robustness as a first-class goal but still amenable to the application programming requirements of interactive workspaces. To this end, the entire architecture was based on tuplespaces [12] but modified to fit the “always safe to reboot” philosophy [15]; it proved sufficiently flexible for developing user-interface toolkits [16] and physical user-interface prototyping [17], and robust enough to “package” as an appliance [1] and subsequently deploy in Stanford’s Meyer Library (http://teamspace.stanford.edu) for general student use with no dedicated system administrator. Others who did previous work in this area have willingly adopted our infrastructure, so there is hope that it will have lasting value as both a development artifact and as the embodiment in ubiquitous computing of some important systems lessons about designing for robustness.

Innovation philosophy

As researchers we must periodically ask ourselves “What is the deep research question or result?” The history of systems research suggests that sometimes a simple but powerful idea, taken to an extreme, can lead to fundamental shifts in technology and provide order-of-magnitude improvements in ways that are often only deep in retrospect. RISC, for example, is not obviously a “deep” idea yet its impact resulted in qualitative advances in microarchitecture and compiler technology that would not have come about any other way, and it quantified design lessons such as “make the common case fast and rare case correct”, which has become a systems mantra.

“Reduce recovery time to improve availability” and “design systems to be reboot-safe” may amount to codification of common sense practices, but explored in the extreme, the combination has led to a qualitative change in detecting and recovering from otherwise elusive failures and a perspective on how to bring other fields of knowledge to bear on system dependability. If this eventually lets us develop some “science” in systems building, the science will have proceeded from the bluntest of recovery instruments, rebooting.
Summary

Our ability to design and deploy large complex systems has outpaced our ability to deterministically predict their behavior except at the coarsest grain. Statistical approaches, which can find patterns and detect deviations in data whose semantics are initially unknown, will be a powerful tool not only for monitoring and online adaptation of these systems but for helping us better understand their structure and behavior. Fast safe microrecovery tolerates nontrivial false positive rates, allowing us to improve sensitivity and coverage to detect failures that would go unnoticed by existing techniques. We argue [10] that a generic platform for pervasive integration of SLT methods and microrecovery would hasten the adoption of these techniques into distributed systems, which would in turn provide a new foundation for the construction of self-managing systems.

Teaching

My teaching philosophy coming in was to emphasize integration of teaching and research, inspire passion across a range of teaching levels by connecting the material with the “real world”, mentor outside the classroom, and encourage students to learn as much from their peers as from the instructor. My generally high student evaluations and ASSU and Tau Beta Pi awards suggest that I have had some success in doing this.

My CS241 course, as well as two joint graduate courses taught with UC Berkeley, have led to numerous research mini-projects that constituted many students’ first exposure to research and opportunity to interact with high-profile “technical advisors” in a setting less intimidating than a conference talk. These mini-projects have resulted in about a dozen refereed conference papers and two Ph.D. theses, as well as ongoing collaborations between Stanford and Berkeley students, exposing them to a range of Departmental cultures and perspectives they wouldn’t get at either institution alone.

Lack of time has limited my involvement with undergraduate advising recently, but I have tried to closely mentor a modest number of outstanding undergraduates. I suspect that my Society of Women Engineers Professor of the Year award is due as much to this as to in-class performance.

Although research distinctions are important, students are the reason I chose academia over industrial research, and therefore student-selected distinctions for mentoring and teaching mean more to me personally than research distinctions. We are privileged to get the very best students at all levels, and they deserve the best mentoring. I’m looking forward to helping two of my own graduating students seek faculty positions of their own this year and to continue improving in this area.

A Note on Professional Responsibility

Paul Hilfinger used to say of his compiler exams that they would cover “the sum total of human knowlege, but with a strong emphasis on compilers.” This is a good metaphor for our responsibility as academic professionals: we should be ambassadors for everything higher education can do, including significant public outreach and public service. Such activity is good for the University and good for the world. I look forward to further developing these career areas beyond my University research and teaching.

References


