DOI:10.1145/2961890

To view the accompanying paper, visit doi.acm.org/10.1145/2958738 rh

# Technical Perspective
# Toward Reliable Programming for Unreliable Hardware

By Todd Millstein

"IT'S NOT A bug; it's a feature!" Though this sentence is often meant as a joke, sometimes a bug really *is* a feature—when the benefits of tolerating the bug outweigh its negative impact on applications.

Designers of emerging hardware architectures are taking this point of view in order to increase energy efficiency, which is a critical concern across the computing landscape, from tiny embedded devices to enormous datacenters. Techniques such as a low-voltage mode for data-processing components and a low refresh rate for memory components can significantly decrease energy consumption. But they also increase the likelihood of *soft errors*, which are transient hardware faults that can cause an erroneous value to be computed or retrieved from memory.

Ultimately, whether these techniques should be considered bugs or features rests on the ability of software systems, and their developers, to tolerate the increase in soft errors. Fortunately, a large class of applications known as *approximate computations* is naturally error-tolerant. A book recommendation system approximates an unknown "ideal" recommendation function, for example, by clustering users with similar tastes. With enough users and data about these users, sporadic errors in the clustering computation are unlikely to cause noticeably worse recommendations. Similarly, an audio encoder can likely tolerate sporadic errors that introduce additional noise without affecting the user experience.

Even so, no application can tolerate an unbounded number of errors. At some point the book recommendations will be random and the music will be unlistenable. How can the implementers of these applications gain assurance that the quality of service will be acceptable despite the potential for soft errors?

The computing industry and research community have developed many tools and techniques for finding bugs and validating properties of programs. However, for the most part those approaches do not help to answer the question here. The issue in this setting is not whether a bug exists, but how likely the bug is to occur and how it will affect the application's behavior. Further, the bug is not in the application but rather in the underlying hardware platform. Finally, it's not even clear how to specify a desired quality-of-service level; traditional program logics based on a binary notion of truth and falsehood are not up to the task.

The following paper by Carbin et al. addresses these challenges in the context of an important subproblem. The authors introduce the notion of a *quantitative reliability specification* for a variable, which specifies a minimal acceptable probability that the variable's computed value will be correct despite the potential for soft errors. For example, a developer may desire a particular variable's value to be correct 99% of the time. The authors introduce a language for providing such specifications as well as an automated code analysis to verify them. Separately, the authors and other researchers have tackled complementary problems, such as how to bound the maximum effect that soft errors can have on a variable's value.

The power of the authors' approach comes from its generality. Despite my example here, reliability specifications are relative rather than absolute. For example, the reliability specification for a function's return value is defined in terms of the reliability probabilities of the function's arguments and so must hold for all possible values of those probabilities. Further, the approach is parameterized by a separate hardware reliability specification that provides specific probabilities of soft errors for different operations (for example, reading from memory, performing an addition). Therefore the approach is oblivious to the particular details of the hardware architecture and the causes of its soft errors.

These choices not only make the approach more general; they also enable the authors to recast the problem in a manner that is surprisingly amenable to traditional program verification techniques. Their analysis validates reliability specifications by determining the probability that each variable's computation incurs no soft errors, since that is a lower bound on the variable's probability of being reliable. By abstracting away the specific reliability probabilities of function inputs as well as of individual operations, the problem essentially becomes one of counting the number of operations that can incur soft errors and that can affect a variable's value, a task that is well suited to automated program analysis.

This work is part of an exciting stream of recent research that adapts and extends traditional program verification techniques to reason about probabilistic properties, which are abundant in modern software systems. I am hopeful this research agenda will lead to general ways of building robust systems out of potentially unreliable parts, where the notion of unreliability is broadly construed—not only soft errors, but also faulty sensor and other environmental inputs, untrusted libraries, and approximate computations themselves. The more tools we have to reason about unreliability, the more bugs we can turn into features. C

**Todd Millstein** is a professor of computer science at UCLA, Los Angeles, CA.