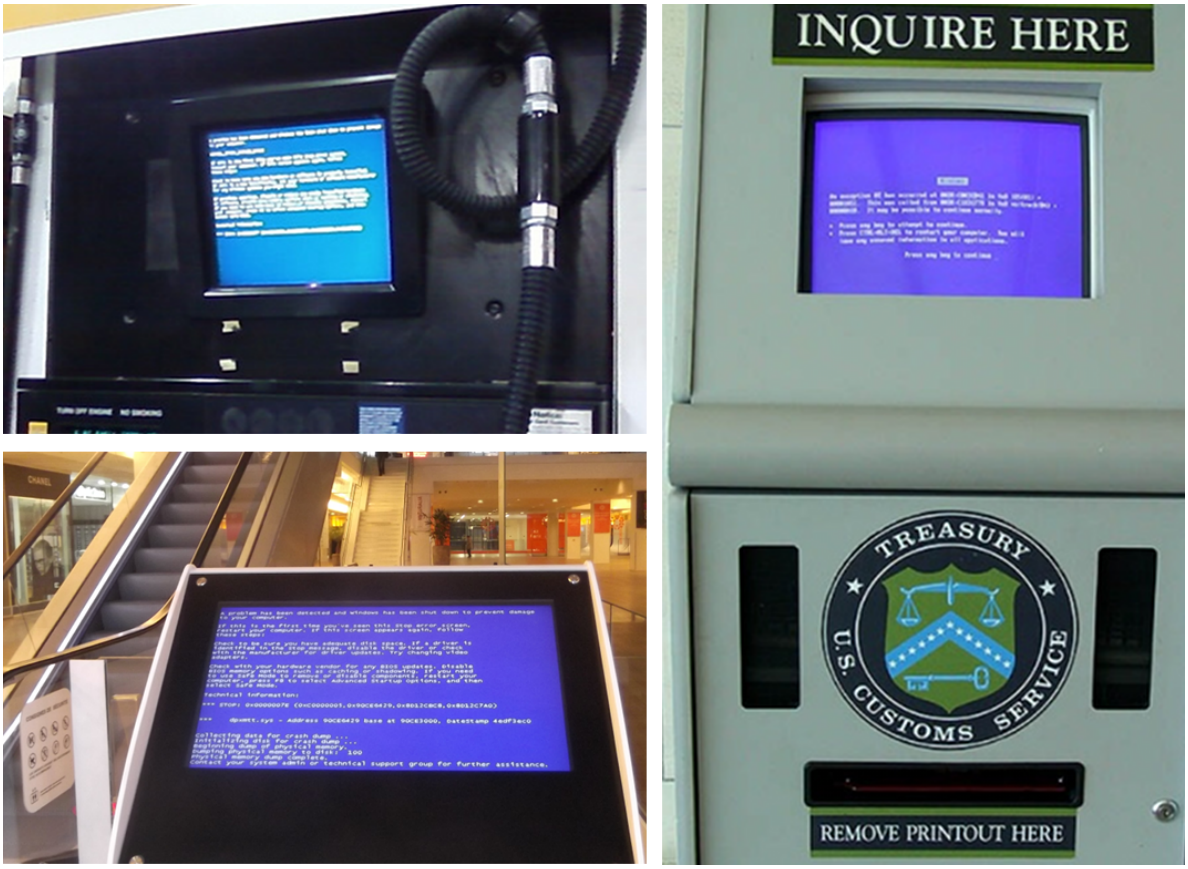



# Beyond efficiency *

**David H. Ackley**

*Esteem for efficiency should be tempered with respect for robustness.*

*Computer science often emphasizes processing efficiency, leaving robustness to be addressed separately. However, robustness requires* ***redundancy****, which efficiency eliminates. For safer and more scalable computing, we must embrace and manage this tradeoff.*

**You've seen them**: Crashed computers, frozen on the job. Fortunately the result is seldom worse than user inconvenience or owner embarrassment. Still, as computer scientists we wonder why the computer inside the machine is so often the weakest link.

Computers keep gaining new responsibilities. In everything from smartphones to cars to medical equipment, we need computers to be *robust*. They should be competent at their jobs, but also sensible about the unexpected, and prudent about the malicious.

Over the years we have learned much about how to keep computers working. Fields like fault tolerance[10] and software reliability[7] employ structured redundancy to enhance robustness. Data centers and other high-availability systems have benefited, but the techniques rarely reach the mass market. Meanwhile, many areas of computer science—for example, algorithm and database design, and programming generally—view redundancy as waste. A common perspec-

*This is an author's preprint of a *Viewpoint* essay accepted for publication in the *Communications of the ACM*, 2013.

tive, here stated categorically for emphasis, says software designers and programmers should assume a 100% reliable deployment platform, and the goal of software is 'CEO': ***C**orrectness and **E**fficiency **O**nly*.

That 'CEO Software' mindset has gone largely unchallenged because it has history and technology behind it. Our traditional digital architectures, error-correcting hardware, and fault-masking subsystems like TCP libraries work together to support it. Yet it is misleading and risky. It implies efficiency and robustness are separate, when actually they are coupled. Teaching it to our students perpetuates that misunderstanding.

CEO Software powers much of computing today, often with great success, but it is neither inevitable nor harmless in general. This essay reviews its origins, offers a canonical example of its hidden risks, and suggests opportunities for balancing efficiency and robustness throughout the computational stack.

**It is easy to blame** the woes of modern computing on flaky hardware, miscreants spreading malware, clueless users clicking, sloppy coders writing buggy code, and companies shipping first and patching later. Certainly in my programming classes I have a stern face for missing error checks and all program ugliness, but the deeper problem is our dependence on the basic von Neumann model of computation—a CPU with RAM, cranking out a vast daisy chain of ideal logical inferences. This is ultimately unscalable, as von Neumann himself observed[11] in 1948. We've stretched von Neumann's model far beyond what he saw for it. The cracks are showing.

The original concept of general-purpose digital computing amounts to this: ***Reliability** is a hardware problem; **desirability** is a software problem*. The goal was to engineer a "perfect logician" to flawlessly follow a logic 'recipe' provided later. Reliability was achieved through *massive redundancy*, using whole wires to carry individual bits, and amplifiers everywhere to squish out errors. Analog machines of the day could compute a desirable result, such an artillery firing solution, using less than ten amplifiers. Early digital machines did similar work more flexibly, but used thousands of amplifiers.

Meanwhile, computer scientists and programmers devised correct recipes for desirable computations. The huge costs but modest abilities of early hardware demanded those recipes be *ruthlessly efficient*. CEO Software was born. Many efficient algorithms were developed, along with efficiency enhancements such as keeping the processor busy with multiple recipes, sharing memory to speed task interactions, and caching intermediate results to save time.

However, if an intermediate result might—for any reason—be *incorrect*, reusing it might just make things worse. Sharing resources lets problems cascade between recipes. If efficiency is optimized, a single fault could corrupt a machine's behavior *indefinitely*—an unlikely outcome with random faults, but a fact of life with hostile security faults.

Computers used to have paid staff watching them. Software was small enough to debug credibly. Computing needs changed slowly. Leaving reliability to hardware worked well. The world has changed: In the hurly-burly of networked consumer computing, the more CEO Software we add to a system, the *more breakable* it gets. Today's crashed machines may be just canaries in the coal mine.

**To see how** robustness and efficiency can trade off, consider sorting a list by comparing pairs of items. This is considered a solved problem in computer science, with solid theory and many efficient algorithms. "Quick sort" and "merge sort" are particularly widely used. When sorting a long list, either one blows the doors off "bubble sort," which basically makes many passes over the list and swaps adjacent items if they compare out of order. Computer scientists love to hate bubble sort, because it is inefficient but easy to reinvent. It keeps popping up in software like a weed.

Efficient sorting algorithms make comparisons that often move items long distances. Bubble sort doesn't do that. It compares items repeatedly. It only moves items one position at a time. Bubble sorting a billion items would involve a billion billion comparisons and take years. *Quick sorting* a billion items took under ten minutes on my laptop: It is an *efficient* algorithm for big tasks.

Yet some tasks are small, like sorting my shopping list by price. Even inefficient bubble sort will do that in the blink of an eye. Is it always best to do even small jobs as fast as possible? Maybe not.

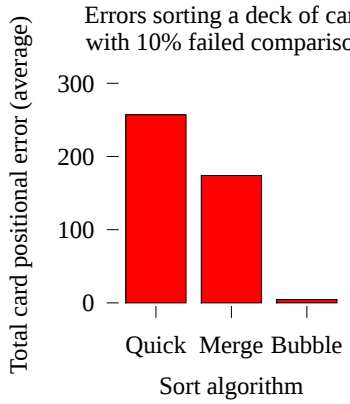
As a demonstration, imagine sorting just 52 items, like a shuffled deck of cards. I used[1] stock implementations of quick and merge sorts, and hand-coded a totally unoptimized bubble sort. Each algorithm picks pairs of cards to compare, and passes them to a 'card comparison component' to determine their order.

Here's the twist: Imagine that component is *unreliable*. Maybe malware corrupted it. Maybe sunspots cooked it. We don't know why. For this demonstration, the card comparison component usually works fine, but on 10% of comparisons it gives a random answer. It might claim $3\spadesuit > 7\diamondsuit$, or $Q\heartsuit = 9\clubsuit$, or whatever.

Given such faults, a sorting algorithm's output might well be incorrect. I scored the output error by adding up each card's distance from its proper position—ranging from 0, for the correct deck order, up to

1,352 (i.e., $52^2/2$), for worst cases like getting the whole deck backwards.

Averaged over 1,000 shuffled decks, here are the errors made by each sorting algorithm:




Now whose doors have fallen off? Bubble sort's inefficient repeated comparisons repair many faults, and its inefficient short moves minimize the damage of the rest.

I subsequently found that quick and merge become competitive with bubble if I repeat each of their comparisons six times and return the most common answer. But that felt like a cheap hack—an after-the-fact fix tailored to one situation. Here, the failed comparisons were *independent, identically-distributed* (IID) events, but real-world faults often occur in bursts or other patterns. For example, a mistaken person gives wrong directions even if you repeat the question. How far should you drive based on their answers? Or in this case, how far should a sorter swap data?

Bubble sort wins on some non-IID faults as well. Suppose our fallible comparator also suffered from a sticky output, so half the time it just repeated its previous answer. Given such awful directions, bubble's error rose to nearly 90—but quick exceeded 550, merge exceeded 650, and even their best-of-six variants both broke 250. Bubble sort is inefficient, but it is *robust*.

**Efficiency is key** when demands exceed resources. Yet many computers are usually idling. They could have been checking their work, but that's off the radar for CEO Software. Optimizing efficiency conspires with program correctness to undermine robustness. The more work done per program action, as with the long swaps of efficient sorters, the more serious the disruption from faulty action. Yet as long as the optimized code remains logically correct, with CEO Software there is no pushback. That risk is never even assessed, because fault-induced failures, no matter how severe, are billed to someone else.

Ironically, even fault tolerance researchers can succumb to the lure of CEO Software. With commendable candor, one group has argued that optimizing fault-free efficiency, "a practice that we have engaged in with relish…, is increasingly misguided, dangerous, and even futile."[6] Broader awareness of CEO Software's liabilities could help avoid such traps.

Over sixty years ago, von Neumann recognized the scalability limits of his namesake design. He predicted a future computing model, a "logic of automata," in which short runtimes would be preferred, and all computational operations would tolerate faults.[11] With transistor sizes continuing to shrink beneath churning piles of dubious software, plus fault-injecting attackers wielding lasers and X-rays[3], we continue to delay that future at our peril.

**Today's computing base** is optimized for the perfect logician, so the quest for robustness has many fronts. A clean-slate approach is robust, indefinitely scalable architectures[2], but research topics abound.

For example, it would help to have better visibility into efficiency-robustness tradeoffs. Existing software quality metrics could be expanded to include correctness degradation under various fault models. Bubble sort wasn't even designed to be robust. How will other algorithms behave?

To set the stage formally, it would help to have results on the tradeoffs between efficiency and robustness, perhaps like a Brewer's Theorem[8] extended to degrees of failure. A starting point might be *highly optimized tolerance*[4], which explicitly represents failure size.

We also need to understand robust applications programming better. The ongoing trend away from monolithic mainlines towards quick event handlers could be seen as compatible with von Neumann's short program runtimes.

More possibilities arise as we replace prima donna CEO Software with robust team player code that gives better than it gets. For example, Dijkstra's *self-stabilization*[9] technique continually "falls toward" correctness. It is a beautiful embodiment of the robust spirit, but requires extra work to interface with CEO Software, because it can violate correctness while stabilizing, which CEO Software cannot tolerate.

Finally, we could develop self-tuning algorithms that trade efficiency against robustness dynamically, based on workload properties, and resource availability and quality. Some existing work[5] uses failure-detection heuristics to switch from efficient to robust processing. More could be done in a computational model that supported signaling external resource characteristics, and internal robustness margins between components.

It's time to study and manage incorrectness in the interest of robustness. We should not shun the tradeoff, but rather, we should understand, engineer, and teach computation beyond correctness and efficiency only.

Robust computation now.

version 201301131528 word count 1724

# References


1. David H. Ackley. Bubble sort robustness demonstration code. http://livingcomputation.com/robusort2.tar, April 2012. Accessed Jan 2013.

2. David H. Ackley, Daniel C. Cannon, and Lance R. Williams. A movable architecture for robust spatial computing. *The Computer Journal*, 2012.

3. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, February 2006.

4. J. M. Carlson and John Doyle. Highly optimized tolerance: a mechanism for power laws in designed systems. *Phys. Rev. E*, 60:1412–1427, Aug 1999.

5. Ilwoo Chang, Matti A. Hiltunen, and Richard D. Schlichting. Affordable fault tolerance through adaptation. In *IPPS/SPDP Workshops*, pages 585–603, 1998.

6. Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

7. Tadashi Dohi and Bojan Cukic, editors. *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*. IEEE, 2011.

8. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

9. Marco Schneider. Self-stabilization. *ACM Comput. Surv.*, 25(1):45–67, March 1993.

10. Robert S. Swarz, Philip Koopman, and Michel Cukier, editors. *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*. IEEE Computer Society, 2012.

11. John von Neumann. The general and logical theory of automata. In L. A. Jeffress, editor, *Cerebral Mechanisms in Behaviour: the Hixon Symposium (1948)*, pages 15–19. Wiley, 1951. Also appears as pages 302–306 in A.H. Taub, editor, *John von Neumann Collected Works: Volume V – Design of Computers, Theory of Automata and Numerical Analysis*, Pergamon Press, 1963.


# Acknowledgments


For their comments and encouragement, the author thanks the readers of earlier versions of this essay, particularly including Brian Christian, Howard Shrobe, George Furnas, Jim Hollan, Michael Lesk, and Terry Jones. John Leslie King was particularly helpful during revisions, as were the vigorous reactions of the anonymous reviewers.


The internet has many pictures of crashed computers. In the opening image, clockwise from top left, from: gizmodo.com http://bit.ly/UZ1yeb, knick-knack.com http://bit.ly/13ovMsd, and wikipedia.org http://bit.ly/UTOlzG.

# Author


David H. Ackley (ackley@cs.unm .edu) is an associate professor of computer science at the University of New Mexico in Albuquerque, NM.