Comparison Criticality in Sorting Algorithms

Thomas B. Jones
Dept. of Computer Science
The University of New Mexico
Albuquerque, NM 87131

David H. Ackley
Dept. of Computer Science
The University of New Mexico
Albuquerque, NM 87131

***Abstract*—Fault tolerance techniques often presume that the end-user computation must complete flawlessly. Though such strict correctness is natural and easy to explain, it's increasingly unaffordable for extreme-scale computations, and blind to possible preferences among errors, should they prove inevitable. In a case study on traditional sorting algorithms, we present explorations of a *criticality* measure defined over *expected fault damage* rather than probability of correctness. We discover novel 'error structure' in even the most familiar algorithms, and observe that different plausible error measures can qualitatively alter criticality relationships, suggesting the importance of explicit error measures and criticality in the wise deployment of the limited spare resources likely to be available in future extreme-scale computers.**

Keywords. Fault-intolerance, fault tolerance, criticality, sorting algorithms, robust-first computing

I. Beyond Strict Correctness

A computation is often envisioned as an abstract mathematical function, faultlessly mapping provided inputs to desired outputs. Of course, a real computation is performed by some necessarily fallible physical device or devices—a process that may or may not yield the intended outputs, raising the question of what should be considered acceptable. The strict 'all-or-nothing' approach to correctness and error, for example, views *any* fault-induced alteration of the input-output mapping as a total failure of the computation. Strict correctness is simple and seems honorable—but it also implies that all errors are equally bad, no matter how harmless or catastrophic.

Though such puritanical rigidity can be satisfied for small programs, the probability of perfection for large computations declines precipitously [1]. Extreme-scale users, understandably reluctant to discard resource-intensive computational results lightly, will increasingly choose to judge some errors worse than others, and thus abandon—if only informally and implicitly—the strict boolean view of correctness. We argue it is better to do that explicitly—and sooner rather than later—to understand better the trade-offs and potentials of graduated correctness or error measures that offer 'partial credit', making distinctions finer than just right and wrong.

A. Criticality

In this paper we define *criticality*, a method of comparing program behavior when specific faults do or do not occur, with respect to some given error measure. Expanding on a previous demonstration [2], we explore the criticality of comparison operations in traditional pairwise sorting algorithms. Across four strategies for scoring sorting error, we observe both expected and unexpected interactions among the algorithms, their efficiencies, and their behaviors under faults.

The rest of this section considers related work, focusing on fault tolerance as well as measures of the 'sortedness' or (conversely) disorder of a list, which we re-purpose as error measures for fallible sorting algorithms. Then Section II explains the criticality method, and Section III presents case study results, illustrating how criticality offers insights into algorithmic behavior in realms beyond strict correctness. Finally, Section IV briefly discusses future work and offers conclusions.

B. Fault tolerance

Multifaceted research and development on the reliability and dependability of computing systems has been ongoing for over four decades [3]. One main branch of that work begins, conceptually, with a *fault-intolerant* computation [4]—designed to terminate immediately on any uncorrected fault—and then strives to preserve and protect that fragile core using fault tolerance techniques based in hardware, such as [5]–[8], or in software, such as [9]–[13].

A related thrust emphasizes *degradable performance* and *performability* [14], evaluating system capability and reliability together to support graceful system degradation. By considering degrees of performance, performability moves distinctly beyond all-or-none systemic perfection, but it still presumes strict correctness for each subtask or component. For example, a degradable performance measure might involve variable job completion delays or changes in link failure probabilities [15], [16]—while still demanding strict correctness

©2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Paper as accepted for FTXS2014; DOI 10.1109/DSN.2014.74

of each job computed and messages conveyed over each link. A performability framework could be extended to handle graduated error measures within individual work components, by adding some suitable error measure, but we have yet to find such efforts in the literature.

There is some research, however, that envisions faults altering the function computed, and requiring evaluation at that level. For example, the *selective reliability* approach, discussed by [17], develops error bounds on computations that are divided into higher and lower reliability sections. The present work in some ways complements that approach, seeking to identify computational steps most likely to need high reliability. Rather than hardware bit-flip faults, our case study model considers faults at the algorithmic level, in the comparison operations performed during sorting.

As another example, *approximate design* [18] aims to increase energy efficiency by running chips at voltages that allow transient errors. An approach called *Application Resilience Characterization* (ARC) [19], based on dynamic binary instrumentation [20], supports approximate design by helping programmers understand how their applications may function in fault-prone environments. Like selective reliability and our criticality method, ARC also reaches beyond the "fault-intolerant core" framework. A difference is we have applied criticality to classical deterministic algorithms, while ARC has been predominantly applied to function approximations—such as machine learning algorithms—for which perfect correctness is rarely the norm. Another difference is that criticality quantifies the impact of faults in individual operations—however they are defined—while ARC categorizes inner loop lines of code qualitatively as *resilient* or *sensitive*.

Interestingly, [21] develops robust floating point iterations for several traditionally exact algorithms, including sorting. They work within a fault-intolerant strict correctness error model, and demonstrate sorting small lists perfectly even with as many as half the floating point operations failing.

C. Measuring Sortedness

In contrast with that work, our interest is in better understanding tradeoffs and algorithmic behaviors when the end user may be willing to accept less than strictly correct results. To investigate sorting as an example, we must confront the question of what "sort of sorted" might mean. Fortunately, sorting is an extremely well-studied topic, and researchers have defined a variety of *sortedness measures*— [22] is one survey—that quantify the notion of 'partially correct sorting'. These measures have traditionally been used to measure a list's 'presortedness'—its degree of disorder before sorting—but they are also usable as measures of output quality of a potentially fallible sorting algorithm.

Existing sortedness measures include *inversions error*—the number of items immediately preceding a smaller item, and *max displacement*—the maximum distance any item must be moved to reach its correct position. In this paper, we explore those measures, as well as all-or-none *strict correctness*, and a measure called *positional error* discussed in the next section.

II. Methodology

Arbitrary faults can have arbitrary effects on the execution of a program, and in the general case little can be said. For this case study:

- We consider only sorting programs based on pair-wise comparisons. The program input is a random permutation of the numbers 0..51, modeling a shuffled deck of cards.
- We presume the existence of an error measure that maps any permutation of the input data into a scalar value from 0.0 meaning "perfectly sorted" to 1.0 meaning "maximally unsorted."
- We consider only faults in the pair-wise sorting comparisons. Such faults fit naturally into our adopted sortedness measures, but of course they are only one of many possibilities. In particular, we presume the data items are never corrupted.
- We collapse a fault's impact on a given comparison down to a scalar called the "criticality" of that fault at that comparison, by averaging across possible inputs and other possible faults.

The rest of this section provides details.

A. Comparison Criticality Defined

Intuitively, the goal of the criticality method is to isolate the *additional program error* due to some specific fault on some specific computational step. The impact of any specific fault depends on three factors, which we call the 'fault mode', the 'error measure', and the 'fault pattern'.

The *fault mode* determines how the computational step acts during a particular type of fault. Specifically, when comparisons fail in our case study, the result is as if the comparison was performed backwards. If the operation $\mathrm{Compare}(3, 4)$ faulted it would return $>$, as if the performed operation had been $\mathrm{Compare}(4, 3)$.

The *error measure* specifies how badly any given computational result deviates from its correct outcome. In this study we tested four different sorting error measures. The first error measure, *strict correctness*, is 0 if the output is perfectly sorted and 1 otherwise. Definitions for the other three error measures—*normalized inversions error* [22], *normalized max displacement* [22], and *normalized positional error* [2]—appear below in equations 1, 2, and 3, respectively.

In those equations $L(i)$ is the position of item i in the output list L, while $L[i]$ is the inverse operation: The value of the ith item in list L. Since we sort lists of distinct numbers from 0 to $N-1$ the ith item in a correctly sorted list is equal to i. Note that each error measure is normalized to $[0,1]$ by dividing by the maximum possible value of that error measure.

$$\mathrm{Inv}(L) = \frac{\sum_{i=0}^{N-2} \frac{L[i]-L[i+1]}{|L[i]-L[i+1]|} + 1}{2N-2} \tag{1}$$

$$\mathrm{MaxDis}(L) = \frac{\max_{i=0}^{N-1} |i - L(i)|}{N-1} \tag{2}$$

$$\mathrm{PosErr}(L) = \frac{\sum_{i=0}^{N-1} |i - L(i)|}{\mathrm{MaxPosErr}(N)} \tag{3}$$

The normalization factor $\mathrm{MaxPosErr}(N)$—the maximum positional error for an input list of size N—is equal to the positional error when a list is reverse sorted:

$$\begin{aligned} \mathrm{MaxPosErr}(N) &= \sum_{i=0}^{N-1} |N - (2i) - 1| \\ &= 2\left\lfloor \left(\frac{N}{2}\right)^2 \right\rfloor \end{aligned} \tag{4}$$

Finally, the *fault pattern* specifies when and where faults occur during the computation. In general the fault pattern may require a matrix to represent computational steps in space and time, but in this case, since the tested sorting algorithms are sequential, a single bit vector suffices, indexed by the number of comparisons so far executed [1].

Given a fault, a fault mode, a fault pattern, and a set of program inputs, the output of the program is completely determined, and its quality can be assessed by the error measure. The *criticality* of a fault, then, is simply the error measure value when the fault occurs minus its value when the fault is absent. For extremely small computations it is possible to calculate a fault's criticality exactly, but more typically the combinatorics make direct calculation intractable. This study uses Monte Carlo sampling to estimate the error measure values with and without the fault, and the difference of those estimates are reported in the next section.

The criticality for a failure at each comparison index was obtained by taking a sample of 1000 fault pattern-input pairs for each comparison in the algorithm. Inputs were randomly generated so that each list item had a uniform probability of occurring in any location in the list. Fault-patterns were sampled from a binomial distribution set to produce $true$ bits at rates of 0%, 10%, and 20% so that comparisons not under consideration would fail at a consistent i.i.d. *background failure rate*[2]. The average output error was measured for each of these fault pattern-input pairs—once with the comparison under consideration forced to fail and once with it succeeding. This gave us two *average conditional error measures* whose difference was then taken as the comparison's criticality. See the top graph in figure 2 for an example of how this was done.

[1] In this paper comparison index c refers to the cth comparison executed by the algorithm

III. Results and Discussion

We tested quick sort, merge sort, and bubble sort, using permutations of the $N = 52$ numbers $[0, 1, ..., 51]$ as input. For each algorithm, we estimated comparison criticality with respect to the four error measures presented in Section II-A, at background failure rates of 0%, 10%, and 20%. In this section. we briefly touch on a few expected and unexpected phenomena we have observed.

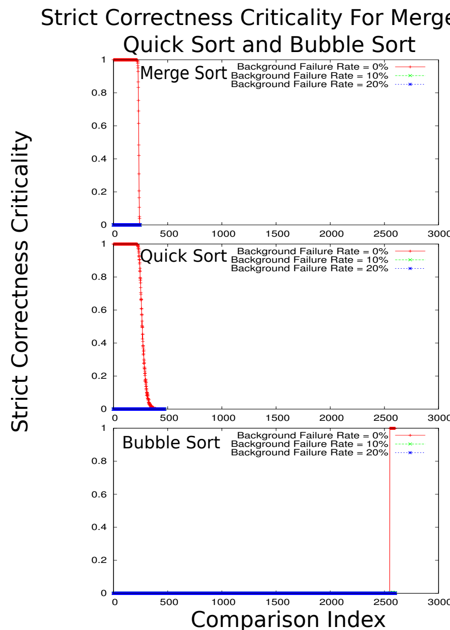
A. Strict Correctness Hides Most Criticality Structure

To get a feel for criticality in general—and to see some of the liabilities of all-or-none correctness—Figure 1 shows estimated criticalities under the strict correctness error measure. While the figure does make clear that quick and merge sorts perform many fewer comparisons than bubble sort, relatively little other structure is revealed. Despite averaging over random input permutations, the strict correctness criticality of each comparison is usually either 1 or 0: Any given comparison is either maximally critical or not at all. Given 0% background failures (*red* curve), for example, there will only be a single fault. For bubble sort, a failure is critical if that fault is in any of the last N comparisons (seen at about comparison 2600), but otherwise it's harmless. By contrast, with merge and quick sorts nearly every comparison is critical—if any comparison fails, the output will not be strictly correct. The last comparisons for quick and merge sorts show intermediate criticalities because, depending on the specific input permutation tested, the algorithm will sometimes finish before that comparison is reached, so a failure at that comparison index is sometimes harmless.

Given strict correctness, if the background failure rate is appreciably non-zero (e.g., 20%, $blue$ curve) all comparisons became non-critical in all three algorithms: Since the output will essentially never be strictly correct, the occurrence or absence of any one fault makes no difference.

[2] Note that a *background* failure rate of 0% does not mean 'no failures whatsoever.' Instead it means there are no failures *other than* the one failure being induced in the comparison under consideration.

Fig. 1. **Extremal values dominate in a plot of strict correctness criticality ('Boolean criticality'; y axis) vs. the comparisons executed during a sort ('Comparison index'; x axis): Most faults are either maximally critical or not at all. See text for details.**



B. Graduated Criticality Reveals Algorithmic Structure

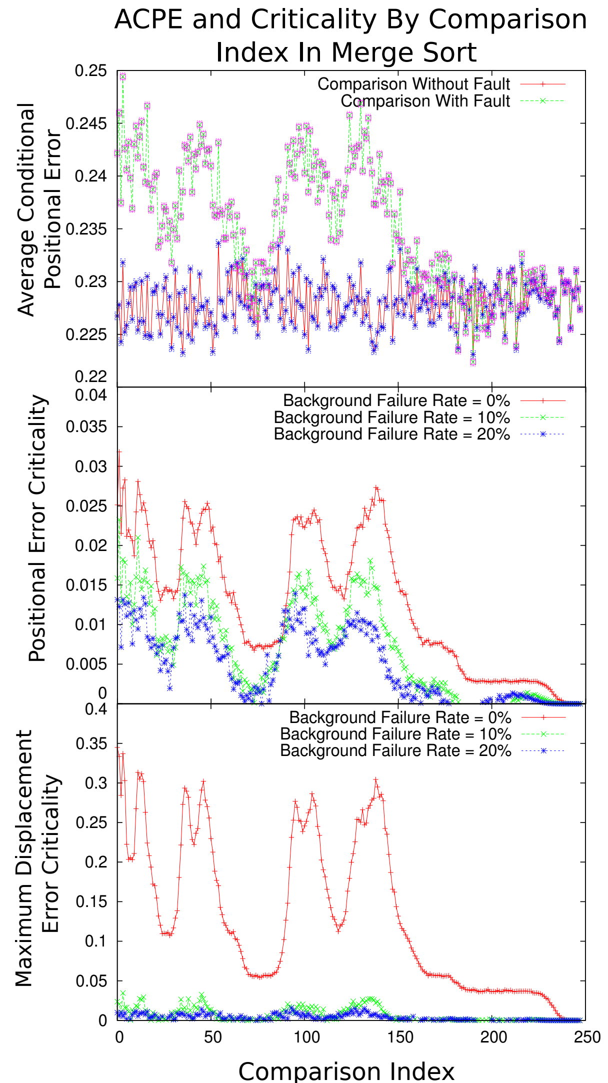
As a second example, Figure 2 shows average conditional positional error and criticality for the merge sort algorithm. The criticality of a fault at a given comparison index—illustrated in the middle graph—is simply equal to the difference between the top and bottom lines in the first graph of Figure 2—the estimated error when the fault does occur less that when it doesn't.

We note two striking aspects in the middle graph in Figure 2. First, the positional error measure reveals a fractal criticality structure for the merge sort algorithm. In retrospect at least this makes sense given the depth-first recursion used in this merge sort implementation. Comparisons at the deepest recursive levels—when two items are merged into a length 2 sublist—are also the most critical comparisons; the deeper "criticality valleys" reflect the larger merges.

Second, that recursive criticality structure is strikingly persistent across background fault rates. Even at a background failure rate of 20% we can still see four distinct 'humps' in merge sort's criticality results. This implies that criticality structure is robust when the right algorithm and error measure are used. Note that the the criticality falls off at larger background failure rates since criticality measures additional error due to a fault and at higher background failure rates so much damage has already been done to the output that it becomes difficult for faults to do even more damage.

Next, when comparing the middle graph of figure 2 to the bottom graph we see that max displacement error reveals a structure that is similar to that revealed by positional error. However, unlike positional error which reveals a structure that is persistent in the face of background faults, the structure of max displacement error quickly collapses as the number of background errors increases.

Fig. 2. **The average conditional positional error curves (*top* graph), corresponding to the estimated error with and without the fault at the given comparison index, and the positional error criticality (*middle* graph), both based on a 10% background error rate. Note that the purple and blue boxes are error bars. Both positional error criticality and max displacement error criticality reveal the fractal structure of the recursive merge sort algorithm (*bottom* graph). See text for details.**



The existence of such criticality structures suggest the possibility of deploying *computational armoring* strategically to the most critical comparisons. Methods like triple modular redundancy [23]—applied here to list comparisons—are expensive, and targeting via criticality might help increase the "bang for the buck" of

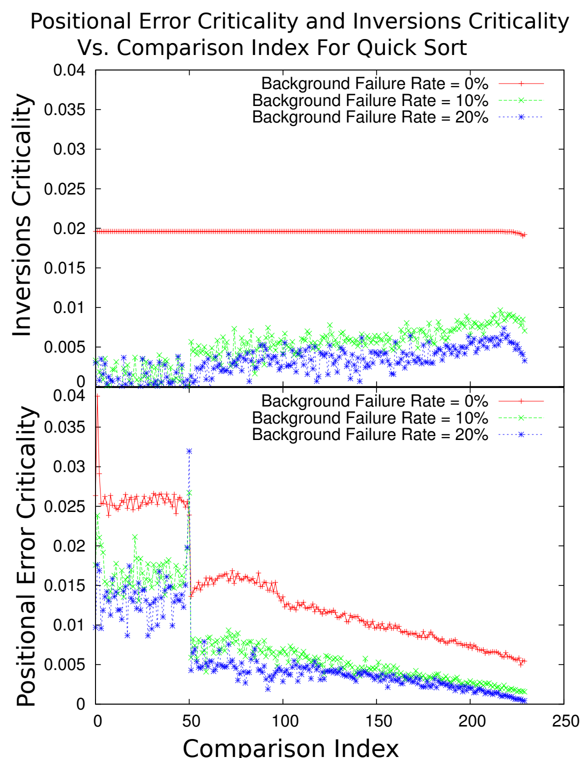
such techniques.

C. Criticality Structures Depend on Error Measures

The strict correctness error measure is a degenerate case, but we also found that the comparison criticality can vary greatly given different graduated error measures. Using quick sort, for example, we measured comparison criticality for the first 250 comparisons using both normalized positional error and inversions error. In Figure 3 we see the results: With positional error criticality, the first $N = 52$ comparisons have much greater criticalities than all other comparisons in the sort. This occurs because the first N comparisons of quick sort are responsible for sorting the list into a 'top' half and 'bottom' half with all items less than the pivot in the bottom half and all items greater than the pivot in the top half. A faulty comparison in the first N comparisons leads to the miscompared item being placed, on average, about $N/2$ away from its correct position.

On the other hand, for inversions error criticality, the first N comparisons have *lower* criticalities than all other comparisons. We suspect this is because items misplaced in either the top or bottom half of the list will tend to move toward the center where the halves meet, so even many faults in the first N comparisons will tend to add only a single inversion to the error measure.

Fig. 3. **Different error measures generate different criticality structures. See text for details.**

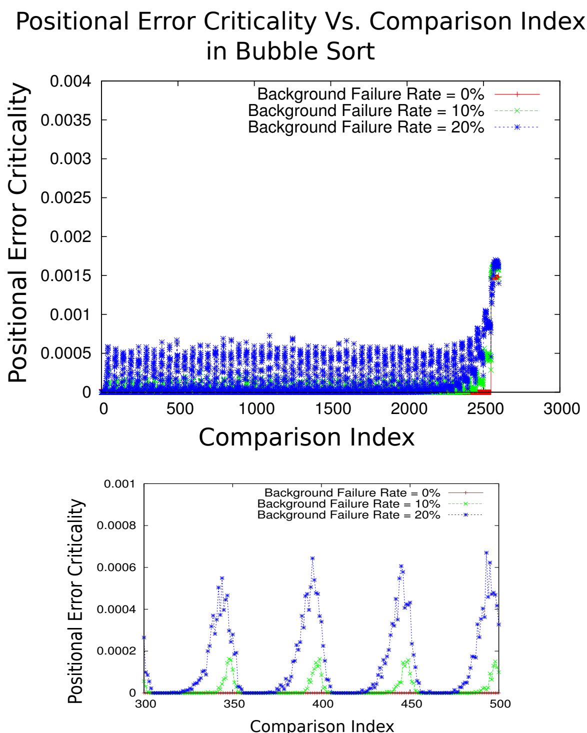


Though positional error and inversions error, like all error measures, agree on the meaning of perfectly sorted, they measure significantly different list properties, and their criticality structures are therefore quite different. Although we may hope to find general principles, it is important to understand that wise choice of error measure requires not only sensitivity to likely faults, but also to the needs of a computation's end-user.

D. Criticality Structure is Highly Dependent on the Algorithm

As a final illustration, consider the bubble sort results in Figure 4. Bubble sort's $O(N^2)$ comparisons give it a great deal of redundancy, so the criticalities in Figure 4 are much smaller than in, say, Figure 3—but in addition, the details of its criticality structures emerge most clearly at relatively high background error rates. It is unsurprising that bubble sort's last N comparisons are most critical, but we, at any rate, hadn't anticipated the small but distinct length N periodic structure throughout bubble sort's execution, indicating increased criticality in the last half of each pass through the list.

Fig. 4. **Periodic criticality in bubble sort facing a non-zero background error rate. The bottom graph is a subgraph of the top graph. See text.**



All three sorting algorithms displayed structures related to the input size of $N = 52$: First N criticality in quick sort, last N criticality in merge sort, and a period N oscillation in bubble sort.

IV. Future Work and Conclusions

Moving forward, we want to apply criticality to other problems and algorithms, especially those traditionally approached via strict correctness. We also hope to investigate additional fault modes, selected to help hardware designers *minimize computational damage* by triaging and shaping faults, rather than trying to suppress them all. And finally, though the presented data was obtained numerically, the observed relationships between input size and criticality raise the possibility of developing predictions of comparison criticality analytically, or by extrapolation from small instances.

In this paper we have introduced the idea of criticality, and demonstrated the application of *graduated* error measures—beyond strict correctness—in a case study of traditional sorting algorithms. We observed that the algorithms possessed previously-unseen criticality structures, which can change significantly depending on the chosen error measure, and which can persist in the face of rising background failures.

The "fault-intolerant core" computing style has scaled vastly beyond what von Neumann predicted for it over sixty years ago [24]. The extreme-scale computing community—tasked with "living in the future"—is increasingly perceiving now what will later become common wisdom: Computing must embrace fault tolerance and robustness *throughout* the computational stack, all the way to the end-user.

Extreme scale will lead the way.

References

[1] K. Ferreira, J. Stearley, J. H. Laros III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 44.

[2] D. H. Ackley, "Beyond efficiency," *Communications of the ACM*, vol. 56, no. 10, pp. 38–40, 2013.

[3] *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*. IEEE, 2013.

[4] A. Avižienis, "Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing," *SIGPLAN Not.*, vol. 10, no. 6, pp. 458–464, Apr. 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808469

[5] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 25–36.

[6] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 87–98, 2002.

[7] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 98–109.

[8] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*. IEEE, 2001, pp. 214–224.

[9] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[10] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented edac protection against seus," *Reliability, IEEE Transactions on*, vol. 49, no. 3, pp. 273–284, 2000.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63–75, 2002.

[12] ——, "Control-flow checking by software signatures," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 111–122, 2002.

[13] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 2003, pp. 137–143.

[14] J. F. Meyer, "On evaluating the performability of degradable computing systems," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 720–731, 1980.

[15] R. Ghosh, K. S. Trivedi, V. K. Naik, and D. S. Kim, "End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach," in *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*. IEEE, 2010, pp. 125–132.

[16] J. P. Sterbenz, E. K. Çetinkaya, M. A. Hameed, A. Jabbar, S. Qian, and J. P. Rohrer, "Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation," *Telecommunication systems*, vol. 52, no. 2, pp. 705–736, 2013.

[17] J. Elliott, M. Hoemmen, and F. Mueller, "Resilience in numerical methods: A position on fault models and methodologies," *arXiv preprint arXiv:1401.3013*, 2014.

[18] K. V. Palem, L. N. Chakrapani, Z. M. Kedem, A. Lingamneni, and K. K. Muntimadugu, "Sustaining moore's law in embedded computing through probabilistic and approximate design: retrospects and prospects," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2009, pp. 1–10.

[19] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 113.

[20] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[21] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi, "A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 161–170.

[22] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 441–476, 1992.

[23] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, Apr. 1962. [Online]. Available: http://dx.doi.org/10.1147/rd.62.0200

[24] J. von Neumann, "The general and logical theory of automata," in *Cerebral Mechanisms in Behaviour: the Hixon Symposium (1948)*, L. A. Jeffress, Ed. Wiley, 1951, pp. 15–19, 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.