Fault Tolerant Computing

6.911 Architectures Anonymous

Chris Laas

April 11, 2000

1 Summary

As the limits of computer technology are pushed into the domains of the very small, the very cheap, and the very robust, the near-perfect reliability which had been granted by the digital abstraction has eroded. Small components may be cheap and fast, but it is an ever greater challenge to produce the correct answer every time, an immovable constraint of the design. (`gcc --wrong-answer-very-quickly` isn't in terribly high demand.) To compound the problem, the environment is anything but friendly to computer systems: a typical system is under mechanical, thermal, radiative, and electromagnetic stresses which may interfere with the operation of the machine.

The set of errors that can occur are divided into two broad categories, defects and faults. Defects, which are errors made in the hardware during manufacture, become far more common as the complexity and density of the circuits increases, resulting in ever lower perfect-chip yields as time goes on. The most common results of chip defects are faulty logic and flaky connections, which usually manifest as opens, shorts, or stuck at 1 or 0.

On the other hand, faults are errors made during a calculation, at run time. For example, electromagnetic noise or particle radiation can cause undesirable energy spikes in a circuit; in fact, this can be exploited to use DRAM chips as reliable and cheap alpha particle detectors. In addition, the inherently statistical nature of metastable circuits (such as registers) cause an low, but nonzero error rate, which can be decreased (by slowing the clock rate) but not eliminated. Run-time faults typically result in unpredictable, random, and independent errors; in contrast, defects usually cause highly predictable and correlated errors.

The generic engineering solution to the problem of flaky components is redundancy: using multiple unreliable components in a coordinated, mutually verifying way can increase the reliability of the complete system by orders of magnitude. For example, if two identical, redundant components are each down 0.1% of the time, their failure modes are completely independent and detectable, and the rest of the system (including the arbitrator which determines which component to trust) can be approximated as 100% perfect, then the whole system should be down only 0.0001% of the time. As the example demonstrates, it's never possible to reach 100% reliability, but it is often possible to come arbitrarily close to the limit. In general, accomplishing this is more difficult than it seems, because any component of the system (be it computing element, network element, input/output device, or even redundancy arbitrator) can be the reliability bottleneck of the system.

2 Defect-tolerant architectures

As manufacturing processes push the size and speed limits of silicon circuitry, and as the sheer number of components per chip balloons, the yield of perfect chips from a fab inevitably drops, wasting precious resources. While the mainstream solution to this problem is ever-more-precise manufacturing processes in ever-more-pristine clean rooms, the cost of such fabs is approaching billions of dollars. Another way to increase effective yields is to design some amount of redundancy into the system; for example, RAM chips are often fabricated with a few extra lines of storage space, which are wired in as replacements for any lines which turn out to be faulty.

A particularly ambitious system of this type is Teramac, a custom configurable computer designed by HP Labs. Teramac consists of 65,536 LUTs (distributed among about a thousand FPGAs) connected via crossbars in a fat-tree network. This extremely flexible architecture had few critical paths (the only components that needed to be perfect were the power and clock wires) and highly redundant network connectivity through the fat-tree; this allowed the compiler to map out the defects in the system once, and simply route around bad LUTs and interconnects when compiling. In particular, the conscious design decision to out-wire Rent's Rule (Teramac used Rent's Rule exponents of 2/3 to 1, as opposed to the "normal" exponents of 1/2 to 2/3) allowed the system to function normally despite defects in 10% of logic cells and 10% of interchip interconnects. Most importantly, the defect tolerance scaled well: if a wire or component were physically damaged, the loss of computational capacity (after recompiling) was roughly proportional to the fraction of damaged parts.

3 Fault-tolerant signal transmission

Several forms of networks (long-haul networks, modems, cell phones), and of medium to long term storage (DRAM, hard drives, backup tapes) are susceptible to data corruption by noise in the environment. Such corruption often manifests as random, independent bit errors, which can be protected against by a class of linear block codes known as error correction codes. In brief, such codes append a certain number of redundant bits (which are a linear combination of the bits of the input block) to a transfer block; the receiver then checks the redundant bits against the input to determine whether there has been an error in transmission, and can sometimes correct the error. One can visualize the algorithm as placing the correct blocks as points in an n-dimensional space; then, any block within a certain "distance" of a correct block can be corrected, and any block outside all the n-spheres thus defined are uncorrectable. Thus, in general, it is straightforward to find a linear encoding which can correct up to t errors, or detect up to $2t$ errors.

The encoding process is quite efficient: it requires an $n - k$ by k matrix to be applied to a k-vector input block. All arithmetic is mod 2, and so can be implemented with ANDs and XORs. However, decoding requires a same-size multiplication, but also requires an $(n - k)$-bit table lookup for error correcting: this can easily outstrip memory resources if the redundant block size n is large.

4 General majority-vote fault-tolerance

Transmitting redundant bits using error correcting codes is sufficient to protect against random, independent bit errors in storage and network elements; however, in the presence of correlated errors and unpredictably faulty computational elements, a more general solution in necessary. The problem can be expressed like this: imagine that a group of generals of the Byzantine army are camped around an enemy city, communicating only by messenger, and are deciding whether to attack, or to retreat. However, the situation is complicated by the fact that some of them may be traitors, and they can't tell which. They must ensure that

1. All loyal generals decide upon the same plan of action, and

2. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Note that the "traitorous" faulty components might conspire or only lie some of the time: the model allows for any sort of correlated errors, or even for malicious intelligences working against the system.

The solution to this most general problem is, essentially, a form of majority voting. For Alice to tell all the other generals "we should retreat!", Alice first sends a message to every other general, saying "Alive says: we should retreat!" This, however, doesn't satisfy the generals, since Alice might be a traitor, telling Bob "we should retreat!" and telling Carl "we should attack!" Thus, every other general (for example, Bob) tells every other general "Bob says: Alice says we should retreat." But again, we run into the same problem, and hence each general chimes in with "Dave says: Bob says Alice says we should retreat" and "Dave says: Carl says Alice says we should attack."

The recursion finally bottoms out at n levels deep, where n is the number of units, since one processors is left out at every level of recursion: Bob need not send a message "Bob says: Bob says Alice says we should retreat". At that point, every processor takes for granted the statement received (which is similar to "Zed says Yak says... Bob says Alice says we should retreat") and performs a majority vote to determine the "true" next-level statement. In other words, if both Zed and Zach say "Yak says... Bob says Alice says we should retreat", but Zeeb says "Yak says... Bob says Alice says we should attack", the unit decides that Zed and Zach are right. This process is used to climb the recursion tree, until, at the top of the chain, there is a final result: "Alice says we should retreat."

Given the strong assumption of *perfect connectivity* among the generals (each general can send messages to any other general, and no general can pretend to be another), this procedure is guaranteed to result in agreement among all the loyal generals so long as no more than 1/3 of the generals are traitors. The algorithm can be modified to cope with incomplete connectivity, so long as the subgraph of loyal generals is connected (otherwise, there would be no way to get a message from one loyal connected component to another). Also, if signed, unforgeable messages can be used, any number of faulty processors can be dealt with, and many of the inefficiencies and details of the procedure can be smoothed over.

Although the theoretical result is a proof of perfect operation, any implementation of the algorithm is still not perfectly fault-tolerant, because the proof relies on strong assumptions which are not true 100% of the time in a non-perfect system. In real systems, one must worry about:

1. Lack of complete connectivity in any practical system.
2. Clock skew: synchronization is an entire fault-prone system in itself.
3. With nonzero probability, more than 1/3 of the nodes will be faulty.
4. If signed messages are used, there is always a non-zero probability of a signed message being correctly forged, either accidentally or maliciously.

5 Conclusion: The Real World

Real fault-tolerant (in buzzword jargon, "ultrareliable") systems use all of the techniques of redundant routing, ECC, and majority voting in various subsystems; it's a proven technology. Most practical implementations of majority voting use simpler, more efficient mechanisms that that outlined above, however. A common implementation is to use an arbitrator to count the votes and determine the correct result; however, care must be taken to ensure that the arbitrator itself is fault-tolerant.

Current systems tend to focus on software techniques, such as transactions, assertions, periodic consistency audits, and pervasive timeouts to attain high uptimes; in general, defensive programming is becoming an ever-more-important tool. Along these lines, most of the research into fault tolerance today seems to focus on the deplorably fact that most faults are due to design errors, programmer errors, and operator error, rather than noise or defects, which are adequately defended against by the techniques outlined above. For example, tools are being developed to perform automated logic checking on programs, determining if a program actually does what its designer intended it to do. Finally, this problem is ultimately human in origin, and the optimal solutions are also human in nature: an effective technique is to have multiple development teams independently design programs to perform a certain task, and then use majority voting on the results of the different programs in the actual production system. Such techniques are used in systems where reliability is critical, such as life support.

References

[1] Shu Lin. *An Introduction to Error-Correcting Codes*. Prentice-Hall, Englewood Cliffs, NJ, 1970, 330pp, pp. 33-57.

[2] James R. Heath, Philip J. Kuekes, Gregory S. Snider, R. Stanley Williams. *A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology*. Science, Vol. 280, 12 June 1998, pp. 1716-1721.

[3] Leslie Lamport, Robert Shostak, Marshall Pease. *The Byzantine Generals Problem*. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, pp. 382-401.