Fault−Tolerant Computing

6.911 Architectures Anonymous

Chris Laas

The Problems

- Defects: manufacture errors
  - As complexity and density of hardware increases, perfect−chip yield decreases.
  - Results in faulty logic, flaky connections (mostly opens, shorts, or stuck at 1 or 0)
- Faults: run−time errors
  - Caused by noise, radiation, metastability.
    - (DRAM chips make great alpha particle detectors!)
  - Results in unpredictable, random, independent errors.

The Solution

- Redundancy: if Foo is flaky, use lots of Foos.
  - A statistical game: you can never reach 100% reliability, but you can often come arbitrarily close.
  - Foo might be a computing element, a network element, an input device, etc.: any might be the reliability bottleneck.

Teramac paper

- Manufacturing defects which can be detected
  - Defects are mapped out once.
  - Map is used by compiler to route around bad LUTs and bad interconnects: this is made possible by the highly redundant fat−tree nature of the interconnects.
  - "It’s the wires": conscious design decision to out−wire Rent’s Rule.

Error−Correcting Codes

- Protects against random, independent bit errors
  - Long−haul networks, networks in noisy environments
  - Medium−to−long−term storage (DRAM, hard drives)
- How Does It Work: append n−k redundant bits to a transfer block of k bits
  - Linear block codes: redundant bits are a linear combination of the input block
  - Straightforward, but in−depth, linear algebra (mod 2) to find a linear encoding which can correct t errors or detect 2t errors.

Error−Correcting Codes

- Efficiency
  - Encoding requires a n−k by k matrix to be applied to the k−vector (mod 2, so can be implemented with ANDs and XORs)
  - Decoding requires a same−size multiplication, plus an (n−k)−bit table lookup for error correcting.

Byzantine Generals

- Theoretical result: given certain (strong) assumptions, reliability can be proven.
  - Definition of reliability:
    - All nonfaulty processors must use the same input value (so they produce the same output).
    - If the input unit is nonfaulty, then all nonfaulty processors use the value it provides as input (so they produce the correct output).
    - In general, the "input unit" could be a processor or something else.
  - In short, the results must be consistent and not "bad".

Byzantine Generals

- Solution: majority voting
  - Basically, every unit tells every other unit "he said <foo>". Then every unit (except "him") tells every other unit "he said she said <foo>", and so on, recursively, until the recursion bottoms out. Then a majority vote is used to determine whether "he said she said Glorb said ... he said <foo>", upon which all nonfaulty processors will agree. Then these values can be used in a majority vote to determine whether "she said Glorb said ... he said <foo>"; and so on, stripping off a "someone said" each time, until, at the top of the chain, there is an agreement on "<foo>".

Byzantine Generals

- Limitations
  - Assumes complete, perfect connectivity
  - If more than 1/3 of processors are faulty, correctness is not guaranteed.
- Variations
  - Can be modified to work around incomplete connectivity
  - If signed messages can be used, any number of faulty processors can be dealt with

Byzantine Generals

- Perfect fault−tolerance? No!
  - Depends on strong assumptions which are not true 100% of the time in a non−perfect system.
    - Strong connectivity assumptions (perfect transmission, high regularity of graph)
    - Clock skew, an entire fault−prone system in itself
    - More than 1/3 of the processors will be faulty, with some nonzero probability
    - If signed messages are used, there is always non−zero probability of a signed message being (either accidentally or maliciously) correctly forged

The Real World Today

- Real fault−tolerant systems use redundant routing, ECC, and majority voting (Byzantine Generals), and it seems to work.
- Also techniques such as transactions, assertions, periodic consistency audits, and pervasive timeouts are used to attain high uptimes: defensive programming.

The Real World Tomorrow

- Most of the research into fault tolerance today seems to focus on the deplorable fact that most faults are due to design errors, programmer error, and operator error, and not due to noise or defects.
  - More radical techniques include things like automated logic checking ("what were you THINKING?")

Questions

- From a system reliability perspective, error correction is a dangerous idea to pursue: the belief that an ECC system is "safe from errors" can lead designers to build a much more fragile system, which the infrequent mis−corrected error can have a much greater effect upon. (Witness the reliability of modern systems versus older ones which assumed errors would occur) What are good ways to deal with this issue?

Questions

- How does the fault−tolerant system on the Space Shuttle work?
- What level of fault tolerance does NASA employ in its spacecraft? Is it Byzantine? Programming errors aside, I would expect corrupted or failed components to be the biggest problem for a computer in space.

Questions

- How applicable are approaches like the Byz. General to real system stability? It seems like it would be better to analize what happens during average failures than absolute worst−case scenerios. Failure is sort of a statistical analysis, so it seems logical to do overall statistics.