

EECE 513: Design of Fault-Tolerant Systems

Wrapup or Synthesis

Basic concepts recap

- Fault → Error → Failure progression
- Terms/Definitions of Reliability, Availability, Maintainability and Safety
- Error detection, error recovery and avoidance
- Coverage and it's importance

Redundancy: Recap

- Parallel and Series Reliability
- Non-parallel and non-series reliability
- Impact of coverage
- Standby redundancy versus series-parallel

Redundancy: Continued

- Exponential failure distributions and bathtub curve (constant failure rate during lifetime)
- TMR can increase reliability for short-term missions, but not long-term missions
 - Need TMR Simplex for long-term missions
- Voter can be a source of unreliability in TMR

Fault Models: Recap

- Fault Models are very important
 - What ?
 - When ?
 - Where ?
- Faults can occur at multiple layers of the system stack (both h/w and s/w)
 - Tradeoff between fault masking and latency.
Lower levels have lower latency but less masking.

Architecture-level Fault tolerance

- High-reliability systems use a wide array of techniques for redundancy
- Commodity systems mostly use ECC/Parity
- ECC is not free – upto 25% performance degradation, also area/power overheads
 - But can substantially improve memory reliability
 - Chipkill ECC needed for memory module failures

Fault Injection

- Act of systematically perturbing program's execution to emulate h/w and s/w faults
- Needs to be systematic, reproducible and representative (also efficient if possible)
- Can be done at multiple levels, from h/w through pins or s/w emulated fault injection (SWIFI)
 - LLFI: Example of a compiler-driven SWIFI technique

Software FT - 1

- Software faults cannot be managed by simple replication as they may affect all replicas
- But majority of S/W faults are Heisenbugs [Gray'87]
 - Process pairs may be able to detect many of them in the Tandem Guardian System [Lee'93]
 - Refuted by later studies involving open-source software [Chen'99]

Software FT - 2

- Multi-version software may alleviate many bugs:
 - N-version programming
 - Recovery Blocks
 - N-Self-Checking Programming
- Mathematical model to show that for commodity s/w development, cost/benefit ratio is weighed more towards single-version software
- Robust data structures with redundant links etc.

Parallel Systems

- Efficient mechanisms to save and restore state
- Checkpointing each process individually in a parallel system can lead to inconsistent checkpoints
- Coordinated checkpointing the de-facto approach [Toueg et al] that solves the consistency problem
- Asynchronous checkpoints need to log messages: Can result in cascading roll-backs in worst case

Distributed Systems

- Distributed systems can have Byzantine faults where anything goes
- Agreement: Multiple nodes need to agree on common value (related to interactive consistency/consensus)
 - Cannot achieve agreement with more than $1/3^{\text{rd}}$ of nodes experiencing Byzantine faults
- Byzantine fault-tolerance algorithms perform many rounds of message exchanges -> high overheads
 - Message authentication can drastically reduce this

Closing thoughts

- I hope you had fun in this course, and that you learned something 😊
- Two announcements:
 - Project presentations on April 22nd 9 Am to Noon, Final Project reports on April 27th, 11:59 PM
 - Teaching evaluations are online until April 10th. I'd appreciate your comments or suggestions on improvement for this course