

**ECE 254 / CPS 225**  
**Fault Tolerant and Testable Computing Systems**

**Introduction**

Copyright 2011 Daniel J. Sorin  
Duke University

**General Course Information**

- **Professor: Daniel J. Sorin**
  - [sorin@ee.duke.edu](mailto:sorin@ee.duke.edu)
  - <http://www.ee.duke.edu/~sorin>
- **Course info**
  - <http://www.ee.duke.edu/~sorin/ece254/>
- **Office and office hours**
  - 209C Hudson Hall
  - Monday 11:15-12:00 (right after class)
  - Thursday 2:00-3:00
- **Prerequisite**
  - ECE 152 or CPS 104 or knowing how a computer works

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

2

**Course Objectives**

- **Learn about fault-tolerant computer systems**
  - Both hardware and software (more emphasis on hardware)
- **Learn how to read/evaluate research papers**
- **Learn how to perform research**

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

3

**Course Guidelines**

- **Students are responsible for:**
  - Homework and 1-page paper summaries - 35% of grade
  - Midterm exam - 20% of grade
  - Final exam - 25% of grade
  - Individual or group project - 20% of grade

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

4

## Project

- **The project is a half-semester-long assignment**
  - Written proposal (no more than 2 pages), due Oct 14
  - Written progress report (<= 3 pages), due Nov 4
  - Final document in conference/journal format (<= 10 pages), Nov 30
  - Final presentation (in class), Nov 30 and Dec 2
- **Groups of 2 or 3 are OK**
- **Get started early!** Talk to me about project ideas.
  - In case you missed it: **GET STARTED EARLY!**

## Academic Misconduct

- **I will not tolerate academically dishonest work.** This includes cheating on homework or exams and plagiarism on the project.
- **Be careful on the project to cite prior work and to give proper credit to others' research.**
- **Ask me if you have any questions. Not knowing the rules does not make misconduct OK!**

## Course Topics

- **Introduction**
  - Terminology and metrics
- **Faults and their Causes**
- **General Fault Tolerance Concepts**
  - Redundancy (spatial, temporal, information, etc.)
  - Forward recovery & backward recovery
- **Applying Theory to Real Systems**
  - Hardware: microprocessors, memory, disks, networks
  - Software
  - (Note: more emphasis on hardware than on software)*
- **Modeling/Evaluation**
- **Testing, Design for Test, Validation**

## Outline (of Intro)

- **Motivation for Fault Tolerance**
- **Goals of Fault Tolerant Computing**
- **Challenges (why this isn't easy)**
- **Some Examples**

## Motivation

- **Fault tolerance has always been around**
  - NASA's deep space probes
  - Medical computing devices (e.g., pacemakers)
  - But this had been a niche market until fairly recently
- **But now fault tolerance is becoming more important**
  - More reliance on computers
- **Extreme fault tolerance**
  - Avionics, car controllers (e.g., anti-lock brakes), etc.
- **High fault tolerance**
  - Commercial servers (databases, web servers), file servers, etc.
- **Some fault tolerance**
  - Desktops, laptops (really!), PDAs, game consoles, etc.

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

9

## Stuff Happens

- **We wouldn't need fault tolerance otherwise!**
- **Physical problems**
  - Melted wire
  - Toasted chip
- **Design flaws**
  - Incorrect logic (e.g., Pentium's FDIV, AMD's quad-core TLB bug)
  - Buggy software (e.g., Vista)
- **Operator error**
  - Incorrect software installation
  - Accidental use of `rm -R *`
- **Malicious attacks**
  - Security is beyond the scope of ECE 254

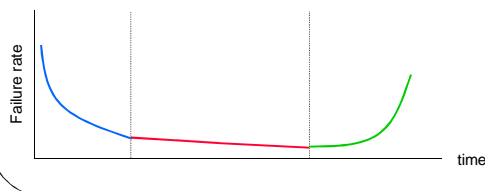
(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

10

## Physical Failures During Lifetime

- **Three phases of system lifetime**
  - Infant mortality
  - Normal lifetime
  - Wear-out period
- **Physical failures follow famous "bathtub curve"**



(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

11

## Outline

- **Motivation for Fault Tolerance**
- **Goals of Fault Tolerant Computing**
- **Challenges (why this isn't easy)**
- **Some Examples**

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

12

## Goals of Fault Tolerant Systems

- How can we deal with problems?
- **Option 1: Make problems less likely**
  - Tough to do!
  - Testing and design for test (DFT) can help avoid physical defects
  - Careful design reviews can help avoid design bugs
  - Training and practice can help avoid operator error
- **Option 2: Fail, but don't corrupt anything**
  - Example: ATM should shut down instead of passing out money
- **Option 3: Transparently tolerate problems**
  - Use hardware and/or software to mask fault effects
  - Key: use **redundancy** (a.k.a. spares or backups)
  - Example: having a co-pilot on an airplane

## Metrics of Goodness: Reliability

- **Reliability:**  $R(t)$  = probability that the system has been operating correctly and continuously from time 0 until time  $t$ , given that it was operating correctly at time 0
- Useful for measuring systems that can't be repaired or that will cause a catastrophe if they fail
  - Examples: satellites, pacemakers
- One measurement of reliability is the **Mean Time To Failure (MTTF)**
  - But the mean doesn't convey the whole story
  - E.g., even if mean is 100 years, some parts can fail in 1 day
- Related metric is **Failures in Time (FIT)**
  - FIT = number of failures per  $10^9$  hours
  - (No, I don't know where this name came from, or why  $10^9$  hours)

## Metrics of Goodness: Availability

- **Availability:**  $A(t)$  = probability that the system is operating correctly at time  $t$
- Useful for measuring systems that can be repaired or that aren't mission critical
  - Examples: file servers, desktops, telephone service
- **Availability =  $MTTF / (MTTF + MTTR)$** 
  - **MTTR** = Mean Time To Repair
  - Mean Time Between Failures (MTBF):  $MTBF := MTTF + MTTR$
- One unit of measurement is the "number of nines"
  - E.g., 5 nines means that  $A(t) = 0.99999$

## Other Terms/Metrics of Goodness

- **Safety:** won't fail in a dangerous way
  - Doesn't guarantee **liveness**, though
- **Maintainability:** ease of maintaining system
  - Somewhat vague metric (what are the units?)
- **Testability:** ease of testing system
  - Can quantify it by how many possible faults we can test for
  - Will cover this in more detail towards end of semester
- **Dependability:** ???
  - A truly vague term, which can be useful, e.g., the International Symposium on **Dependable** Systems and Networks (DSN)
- **Lots of -abilities ... not all of which are clearly and consistently defined**
  - One of my favorites is "performability"

## Architectural Vulnerability Factor (AVF)

- Recently, a new metric for architects to use: AVF
- Quantifies how vulnerable a storage structure is to transient faults
- Higher AVF  $\leftrightarrow$  higher probability that a transient fault will lead to an error that is visible (not masked) at the architectural level

## Outline

- Motivation for Fault Tolerance
- Goals of Fault Tolerant Computing
- Challenges (why this isn't easy)
- Some Examples

## Why Fault Tolerance Isn't Easy

- Fault tolerance can be solved to any arbitrary degree if you're willing to throw resources at the problem
- Resources to sacrifice:
  - System performance
  - Cost
  - Power
- Example: laptop running PowerPoint
  - Buy 2 different laptops (Dell w/AMD chip and a Mac w/Intel chip)
  - Run them more slowly to avoid certain hardware faults
  - Use 2 different operating systems (Windows 7 and Mac OS)
    - » No, neither one can be Vista
  - Use 2 versions of PowerPoint written by 2 independent groups
  - And only let skilled experts use either machine

## Trying Not To Lose Performance

- There are many FT approaches that sacrifice performance to tolerate faults
- Example 1
  - Periodically stop the system and checkpoint its state to disk
  - If fault occurs, recover state from checkpoint and resume
- Example 2
  - Log all changes made to system state in case recovery is needed
  - During recovery, undo the changes from the log
- Example 3
  - Run two identical systems in parallel
  - Compare their results before using them
- Example 4
  - Run software with lots of assertions and error checking

## Performance Issues

- **Most important not to degrade performance during fault-free operation**
  - This is the common-case → make it fast! (refer to [Amdahl's Law](#))
- **Somewhat less important not to degrade performance when a fault occurs**
  - This still might not be acceptable in certain situations (e.g., real-time systems)

## Trying Not To Increase Cost

- There are many FT approaches that sacrifice **cost** to tolerate faults
- **Example 1**
  - Replicate the hardware 3 times and vote to determine correct output
- **Example 2**
  - Mirror the disks (RAID-1) to tolerate disk failures
- **Example 3**
  - Use multiple independent versions of software to tolerate bugs
    - » Called N-version programming

## Trying Not To Increase Power

- There are many FT approaches that sacrifice **power** to tolerate faults
- **Examples 1, 2 & 3 (same as previous slide)**
  - Replicate the hardware 3 times and vote to determine correct output
  - Mirror the disks (RAID-1) to tolerate disk failures
  - Use multiple independent versions of software to tolerate bugs
- **Example 4**
  - Add continuously running checking hardware to system
- **Example 5**
  - Add extra code to check for software faults

## Outline

- Motivation for Fault Tolerance
- Goals of Fault Tolerant Computing
- Challenges (why this isn't easy)
- [Some Examples](#)

### But First, Levels of Fault Tolerance

- Fault tolerance can be at many levels in a system:
- **Application software**
  - Adding assertions to code
- **Operating system**
  - Protecting OS from hanging
- **Entire hardware system**
- **Hardware sub-system**
- **Circuits and transistors**

### Example 1: Telephone Switching System

- **Extreme availability**
  - Goal: <3 minutes of downtime per year
  - Goal: <0.01% of calls processed incorrectly
- **Uses physical redundancy to implement FER**
- **Hardware cost: about 2.5 times cost of equivalent non-redundant system**
- **Also uses:**
  - Error detecting/correcting codes (e.g., parity, CRC)
  - Watchdog timers
  - Many forms of diagnostics
  - Dynamic verification (“sanity program”)

### Example 2: IBM Mainframes

- **Lots of fault tolerance → high availability**
- **Note: IBM has produced mainframes since the 1960s, and they’ve changed their design and enhanced their fault tolerance several times since then**
- **Redundancy at many levels**
  - Redundant units within processor (e.g., register file)
  - Redundant processors
- **Diagnostic hardware for isolating faults**
- **Reliable operating system**
- **We’ll read much more about mainframes later ...**

### Example 3: My Laptop

- **Minimal fault tolerance**
- **Designed to be cheap and fast ... and obsolete in a few years**
- **May have parity or ECC on:**
  - Some bus lines
  - DRAM
  - Hard disk
- **Expected lifetime (as expected by me): 2 years**
- **Expected MTTRboot (also by me): 1 week**
- **Expected MTTRinstall: 6 months**
- **Expected probability of it successfully coming out of “hibernation” mode: 0**

### Example 4: Database Software (Oracle, DB2)

- **Lots of fault tolerance**
- **Can't afford to corrupt vital data**
- **Enforces "ACID" properties for transactions & data**
  - **A**tomicity: transactions are atomic
  - **C**onsistency: only valid data written to database
  - **I**solation: transactions don't interfere with each other
  - **D**urability: data written to database won't be lost
- **Implemented with logging and checkpointing**
- **Writes important data to disks**
  - Can even tolerate a fault that occurs while writing to disks

### Outline (of Intro)

- **Motivation for Fault Tolerance**
- **Goals of Fault Tolerant Computing**
- **Challenges (why this isn't easy)**
- **Some Examples**