19
Dependability for Embedded Systems

Distributed Embedded Systems

Philip Koopman

November 11, 2015

Some slides based on material by Dan Siewiorek

Carnegie Mellon

Preview

- **Dependability overview**
  - Definitions; approaches

- **How and why things break**
  - Mechanical / Hardware / Software
  - Intro to reliability computations

- **Designing systems for failure detection & recovery**
  - Practical limits of fault tolerant design
  - Environment & other sources of problems
  - How to (and not to) design a highly available system

Definitions

- **Source for this section:**
  - *Fundamental Concepts of Dependability (2001)*
  - Avizienis, Laprie & Randell; fault tolerant computing community

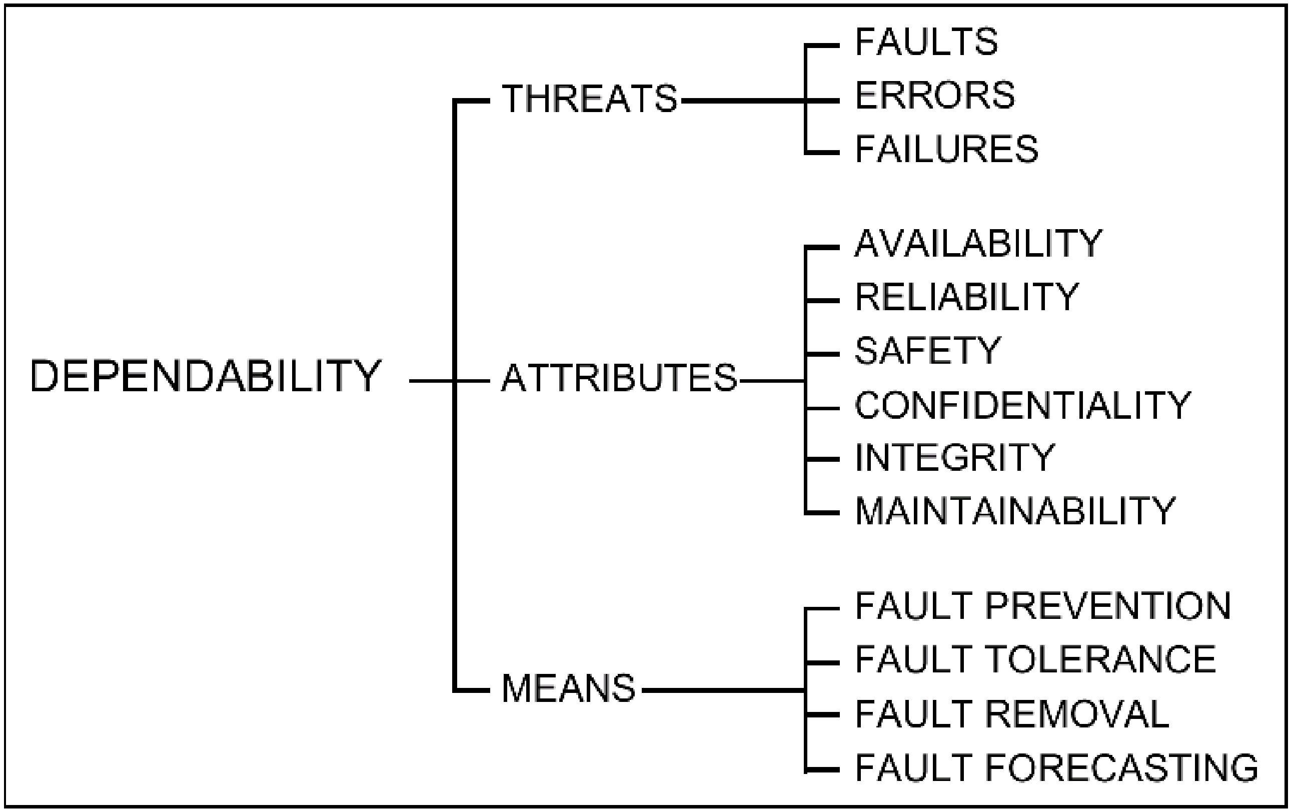


Figure 1 - The dependability tree [Avizienis/Laprie/Randell 01]

Threats:

- Failure = system does not deliver service
- Error = system state incorrect (may or may not cause failure)
- Fault = defect; incorrect data; etc. (*potential* cause of error *if activated*)

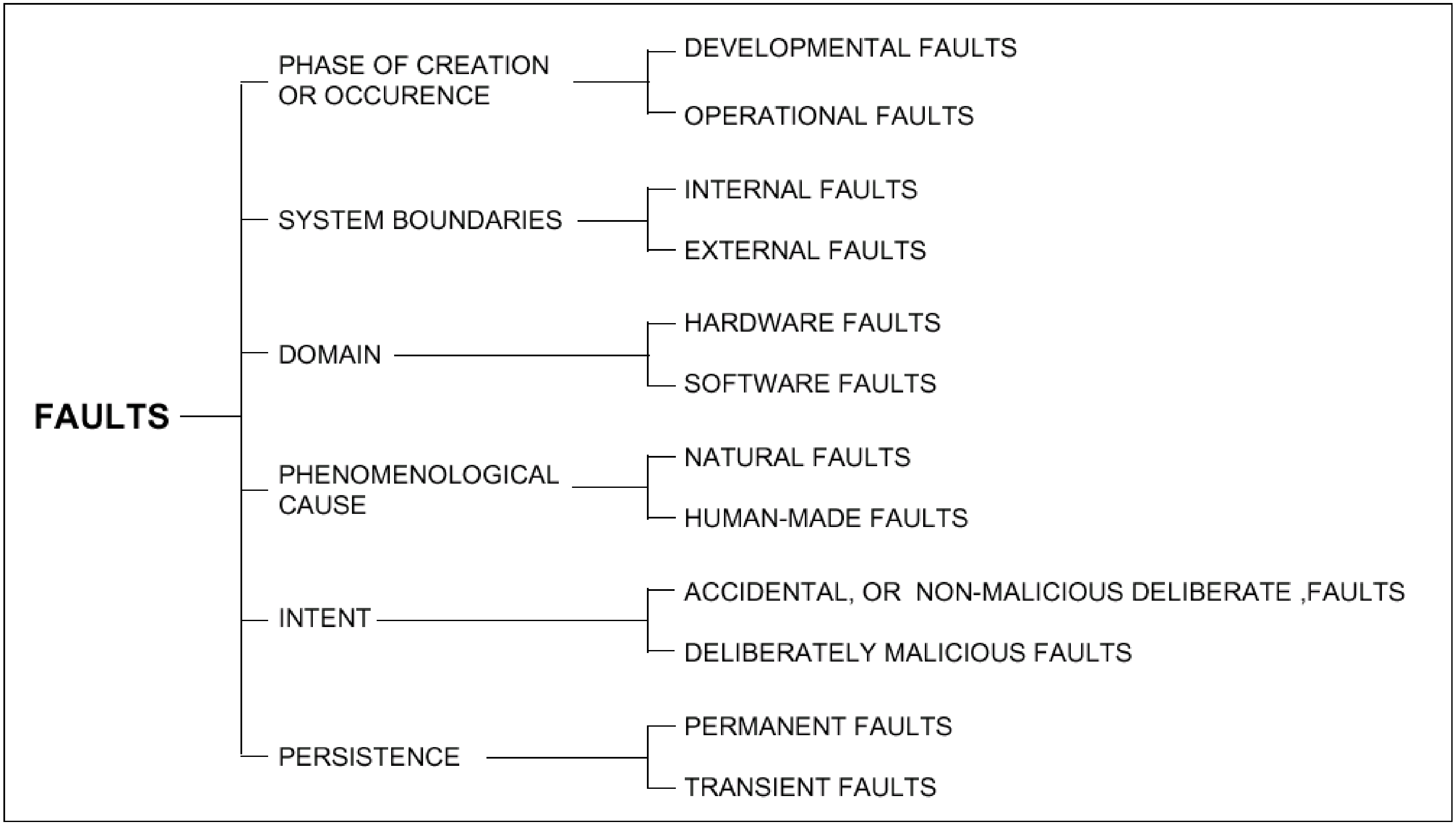


Figure 3 - Elementary fault classes [Avizienis/Laprie/Randell 01]

Primary Attributes of Dependability

- **Dependability** (covers all other terms):
  - The ability to deliver service that can justifiably be trusted
- **Availability:**
  - Readiness for correct service – "Uptime" as a percentage
- **Reliability:**
  - Continuity of correct service
    - (How likely is it that the system can complete a mission of a given duration?)
- **Safety:**
  - Absence of catastrophic consequences on the user(s) and the environment
- **Confidentiality:**
  - Absence of unauthorized disclosure of information
- **Integrity:**
  - Absence of improper system state alterations
    - (Note: security involves *malicious* faults; confidentiality & integrity)
- **Maintainability:**
  - Ability to undergo repairs and modifications
- **MTBF: Mean Time Between Failures**
  - Time between failures, including down-time awaiting repair from previous failure

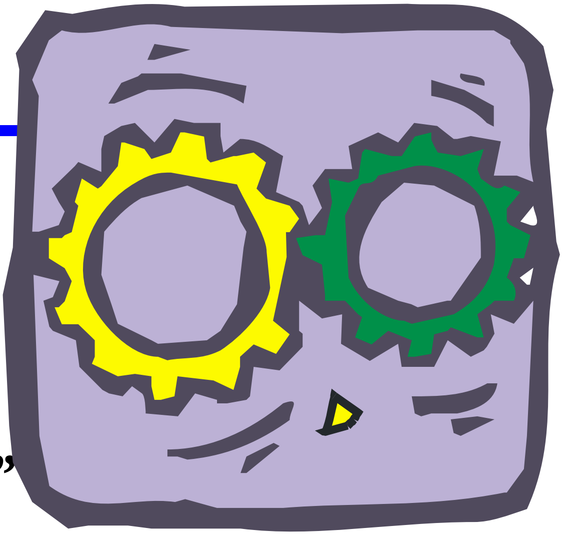
[Avizienis/Laprie/Randell 01]

Means to Attain Dependability

- **Fault prevention – "get it right; keep it right"**
  - Avoid design defects (good process; tools; etc.)
  - Shielding, robust design, good operational procedures *etc.* for runtime faults
- **Fault tolerance – "when things go wrong, deal with it"**
  - Error detection
  - Recovery
  - Fault handling
- **Fault removal – "if you don't get it right at first, make it right"**
  - Verification & validation (design phase)
  - Corrective & preventive maintenance (operations)
- **Fault forecasting – "know how wrong you expect to be"**
  - Simulation, modeling, prediction from process metrics (data from models)
  - Historical data, accelerated life testing, etc.   (data from real systems)

(Note: many of above require accurate fault detection to know when something needs to be fixed or avoided)

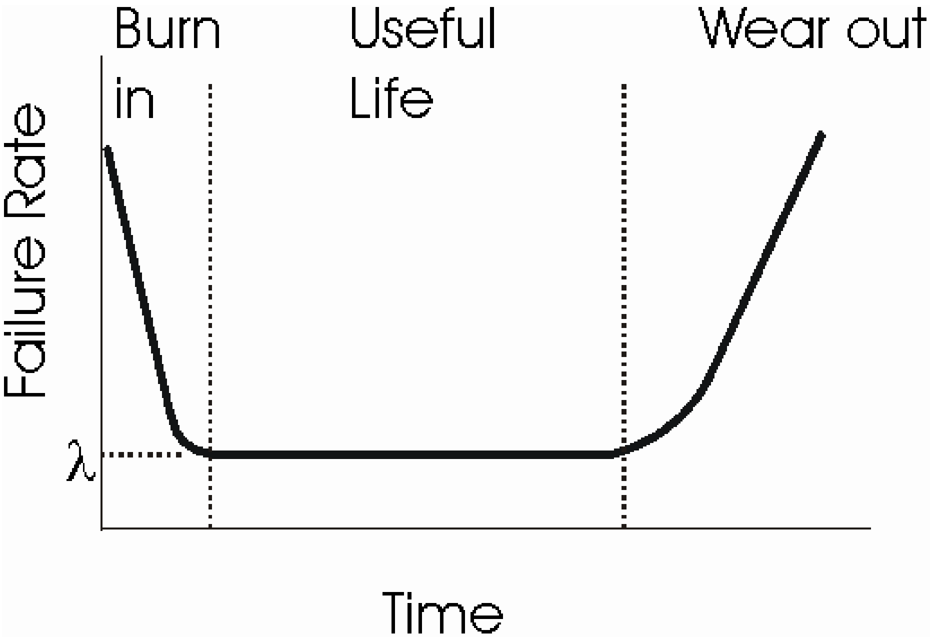
Generic Sources of Faults



- **Mechanical – *"wears out"***
  - Deterioration: wear, fatigue, corrosion
  - Shock: fractures, stiction, overload
- **Electronic Hardware – *"bad fabrication; wears out"***
  - Latent manufacturing defects
  - Operating environment: noise, heat, ESD, electro-migration
  - Design defects (*e.g.*, Pentium FDIV bug)
- **Software – *"bad design"***
  - Design defects – some code doesn't break; it comes pre-broken
  - "Code rot" – accumulate run-time faults
    - Sometimes this "feels" like wearout or random failures, but it is a design fault
- **Outside influence – operation outside intended design limits**
  - People: Mistakes/Malice
  - Environment: natural disasters; harsh operating conditions

Definition of Reliability

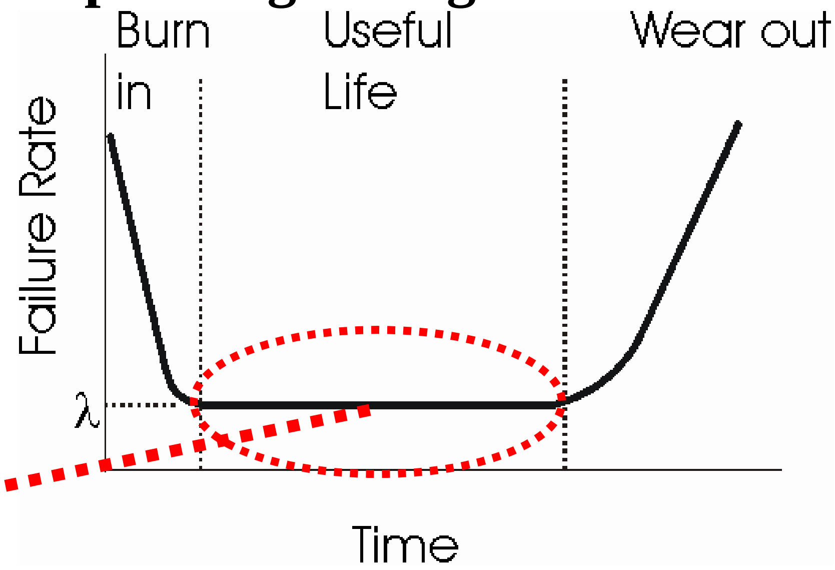
- **Reliability is the probability that a system/product will perform in a satisfactory manner for a given period of time under specified operating conditions**
  - **Probability**: usually assumes constant, random failure rate
    - Typical assumption: system is in “useful life” phase
  - **Satisfactory:** usually means 100% working or *m* of *n* redundancy
  - **Time:** usually associated with mission time = a period of continuous operation
  - **Specified** operating conditions: all bets are off in exceptional situations



Burn-in issues can be dramatically reduced with good quality control (e.g., 6-sigma) – but only from high quality suppliers!

Reliability Calculations

- **R(t) is probability that system will still be operating in single mission at time t**
  - Start of mission is time 0
  - Assume no repairs made during single mission
  - ○ is <u>constant</u> failure rate (in same units as time, e.g., per hour during <u>useful life</u>)



$$\boxed{R(t) = e^{-\lambda t}}$$

  - Assumption: *<u>independent faults!</u>*

  - Example: if ○ is 25 per million hours, for an 8-hour mission:

$$R(8) = e^{-(25*10^{-6})(8)} = 99.98\%$$

(*i.e.*, the system will complete an 8-hour mission without failure 9,998 times of 10,000)

MTTF & Availability

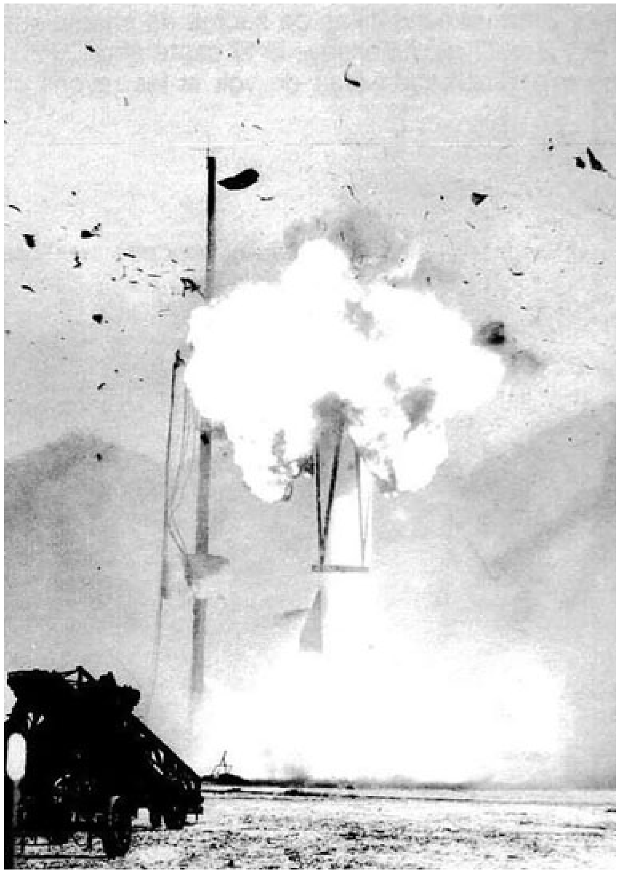
- **MTTF = Mean Time To Failure**
  - (related to MTBF, but doesn't include repair time)
  - MTTF can be represented as the **failure rate ○** (typically in per million hours)
- **Counter-intuitive situation is:**
  How many systems are still working at t=MTTF?
  mission time = MTTF, t = 1/○, **R(○) = <u>37%</u>**

$$R\left(t = \frac{1}{\lambda}\right) = e^{-\lambda t} = e^{-\lambda \frac{1}{\lambda}} = e^{-1} = 36.79\%$$

- **Availability is up-time computed over life of system, not per mission**
  - As an approximation (assuming detection time is small):
    Availability = 1 - ((Time to repair one failure) / MTBF)
  - Example: repair time of 10 hours; MTBF of 1000 hours
    - Availability = 1 – (10 / 1000) = 0.99 = 99%

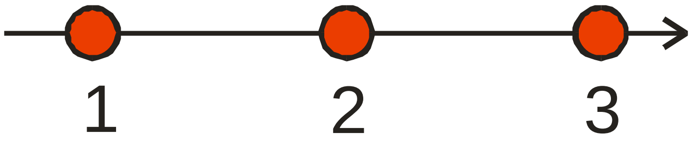
Origins of Reliability Theory

- **WWII: Modern reliability theory invented**
  - To improve V-2 German rocket
  - For US Radar/electronics
- **Problem: *<u>Misleading</u>* mechanical analogy:**
  - "Improved" V-2 rockets kept blowing up!
  - "Chain is as strong as its weakest link"
    - So strengthen any link that breaks to improve system
  - Assumes failures based only on over-stress effects
  - Works for simple mechanical components (chains), not electronic components



Serial Equivalent Reliability

- **Serial reliability – It only took one failure to blow up an original V-2**
  - Any single component failure causes system failure



$$\boxed{R(t)_{SERIAL} = R(t)_1 R(t)_2 R(t)_3 = \prod_i R(t)_i}$$

  - Example for mission time of 3 hours:
    - $\circ_1$ = 7 per million hours: $R(3)_1 = e^{-3*7*10^{-6}} = 0.999979$
    - $\circ_2$ = 2 per million hours: $R(3)_2 = e^{-3*2*10^{-6}} = 0.999994$
    - $\circ_3$ = 15 per million hours: $R(3)_3 = e^{-3*15*10^{-6}} = 0.999955$
    - $R(3)_{TOTAL} = R(3)_1\ R(3)_2\ R(3)_3$
      $= 0.999979 * 0.999994 * 0.999955 = 0.999928$

      which equates to 72 failures per million missions of 3 hours each

    - Serial system reliability can be dominated (but *not* solely determined) by worst component

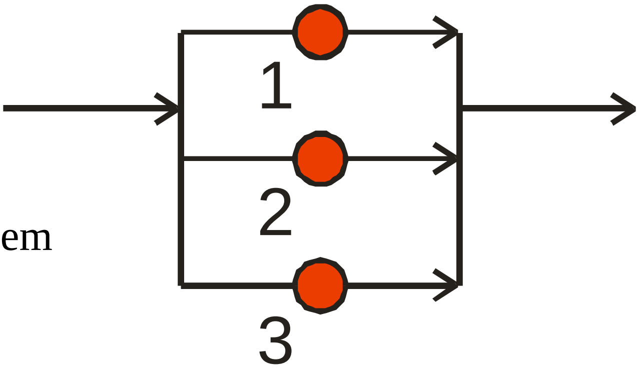
Parallel Equivalent Reliability

Parallel reliability

- Simple version -- assume only 1 of N components needs to operate

$$R(t)_{TOTAL} = 1 - \left[\left(1 - R(t)_1\right)\left(1 - R(t)_2\right)\left(1 - R(t)_3\right) \right]$$

$$\boxed{= 1 - \prod_i \left(1 - R(t)_i\right)}$$



  - Computed by finding (1 – unreliability) of composite system

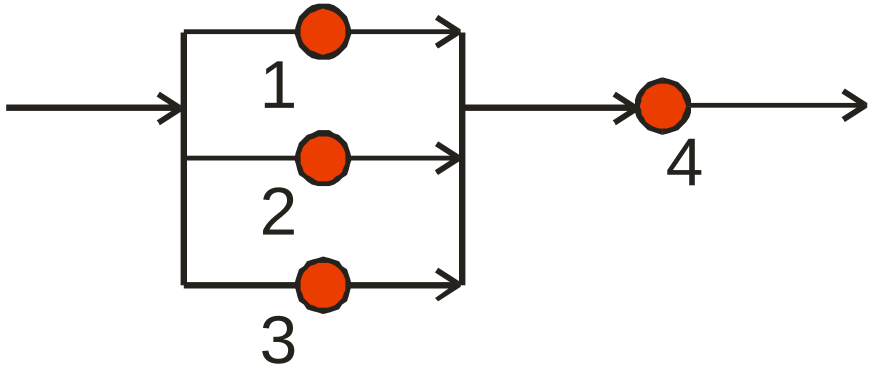
- Example for mission time of 3 hours
  - ○$_1$ = 7 per million hours: $R(3)_1 = e^{-3*7*10^{-6}} = 0.999979$
  - ○$_2$ = 200 per million hours: $R(3)_2 = e^{-3*200*10^{-6}} = 0.999400$
  - ○$_3$ = 15000 per million hours: $R(3)_3 = e^{-3*15000*10^{-6}} = 0.955997$
  - $R(3)_{TOTAL} = 1\text{-}[(1\text{-}R(3)_1)(1\text{-}R(3)_2)(1\text{-}R(3)_3)]$
    $= 1\text{-}[(1\text{-}0.999979)(1\text{-} 0.999400)(1\text{-} 0.955997)] = 0.999\ 999\ 999\ 45$

    which equates to 550 failures per trillion missions of 3 hours in length
  - You can make a very reliable system out of moderately reliable components (but only if they are all *strictly* in parallel!)

- More complex math used for M of N subsystems
  - These may also be a “voter” that counts for a serial reliability element!

Combination Serial/Parallel Systems

- **Recursively apply parallel/serial equations to subsystems**



- Example for mission time of 3 hours
  - o_1 = 7 per million hours: $R(3)_1 = e^{-3*7*10^{-6}} = 0.999979$
  - o_2 = 200 per million hours: $R(3)_2 = e^{-3*200*10^{-6}} = 0.999400$
  - o_3 = 15000 per million hours: $R(3)_3 = e^{-3*15000*10^{-6}} = 0.955997$
  - o_4 = 2 per million hours: $R(3)_4 = e^{-3*2*10^{-6}} = 0.999994$
  - $R(3)_{PARALLEL} = 1-[(1-R(3)_1)(1-R(3)_2)(1-R(3)_3)] = 0.999\ 999\ 999\ 45$
  - $R(3)_{TOTAL} = R(3)_{PARALLEL}\ R(3)_4 = 0.999\ 999\ 999\ 45 * 0.999994 = 0.999994$

- Note that a relatively reliable serial voter dominates reliability of redundant system!

Pondering Probability: Reliability *vs.* Availability

- **Availability is per generic unit of time**
  - Analogous to "I've tossed N heads in a row; what's probability of heads next time?"
    (answer: still 50%)
  - Instantaneous measure – no notion of duration of time
  - E.g.: "Our servers haven't crashed in 5 years minus 1 hour. What is chance they will crash in the next hour?"

- **Reliability is for a continuous mission time**
  - Analogous to the next N coin tosses in a row coming up "heads" (N is mission length)
    (answer: one chance in 2^N)
  - Assumes everything is working and is has no pre-mission memory
  - Assumes failures are independent
  - The longer the mission, the less likely you will get lucky N hours in a row
  - E.g.: "What is the chance our server will run for 5 years without crashing?"

Common Hardware Failures

- **Connectors**
  - Especially wiring harnesses that can be yanked
  - Especially if exposed to corrosive environments

- **Power supplies**
  - Especially on/off switches on PCs
  - Batteries

- **Moving mechanical pieces**
  - Especially bearings and sliding components

How Often Do Components Break?

- **Failure rates often expressed in failures / million operating hours (“Lambda” ○) or “FIT” (Failures in time, which is per billion operating hours)**

Military Microprocessor	**0.022 /Mhr**
Automotive Microprocessor	**0.12 (1987 data)**
Electric Motor	**2.17**
Lead/Acid battery	**16.9**
Oil Pump	**37.3**
Human: single operator best case	**100 (per Mactions)**
Automotive Wiring Harness (luxury)	**775**
Human: crisis intervention	**300,000 (per Mactions)**

- **We have no clue how we should quantitatively predict software field reliability**
  - Best efforts at this point based on usage profile & field experience

Data Sources – Where Do I Look Up o?

- **Reliability Analysis Center (US Air Force contractor)**
  - *http://rac.alionscience.com/*
  - Electronic Parts Reliability Data (2000 pages)
  - Nonelectronic Parts Reliability Data (1000+ pages)
  - Nonoperating Reliability Databook (300 pages)

- **Recipe books:**
  - Recipe book: MIL-HDBK-217F
  - Military Handbook 338B: Electronic Reliability Design Handbook
  - Automotive Electronics Reliability SP-696

- **Reliability references:**
  - Reliability Engineering Handbook by Kececioglu
  - Handbook of Software Reliability Engineering by Lyu…
    - … where you'll read that this is still a research area

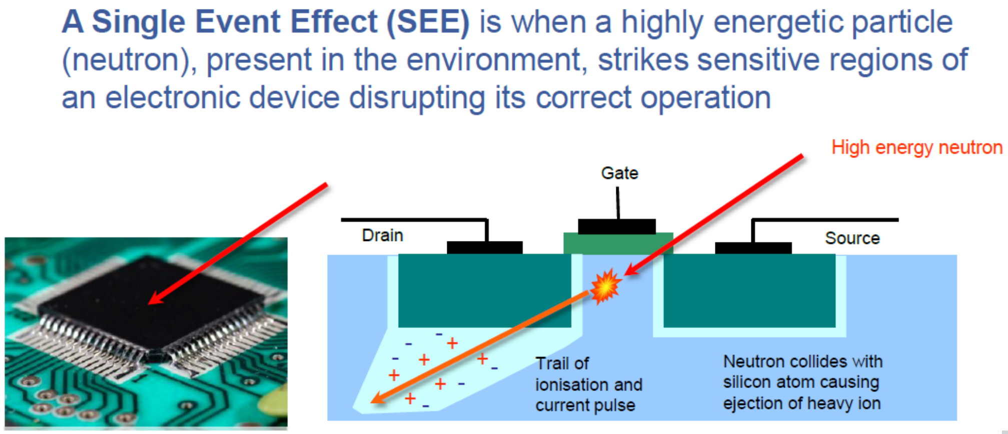
Data Sources

- **Reliability Analysis Center (RAC); USAF Contractor**
  - Lots of data for hardware; slim pickings for software

EPRD-97
Electronic Parts Reliability Data
A Compendium of Commercial and Military Device Field Failure Rates
Volume 1
RAC
EPRD-97
Electronic Parts Reliability Data
A Compendium of Commercial and Military Device Field Failure Rates
Volume 2
RAC
Reliability Analysis Center
NPRD-95
Nonelectronic Parts Reliability Data 1995
RAC
Reliability Analysis Center
Nonoperating Reliability Databook
RAC
Reliability Analysis Center
Introduction to Software Reliability: A State of the Art Review
RAC
Reliability Analysis Center

Transient Faults Matter Too

- **Software: they can happen in essentially all software ("crashes")**
  - Software is deterministic, but sometimes a "random" fault model is useful
    - Race conditions, stack overflows due to ISR timing, etc.
- **Hardware: 10x to 100x more frequent than permanent faults**
  - Electromagnetic Interference
  - Random upsets due to cosmic rays = "Soft Errors" (yes, really)



A Single Event Effect (SEE) is when a highly energetic particle (neutron), present in the environment, strikes sensitive regions of an electronic device disrupting its correct operation

High energy neutron
Gate
Drain
Source
Trail of ionisation and current pulse
Neutron collides with silicon atom causing ejection of heavy ion

Radiation strike causing transistor disruption (Gorini 2012)

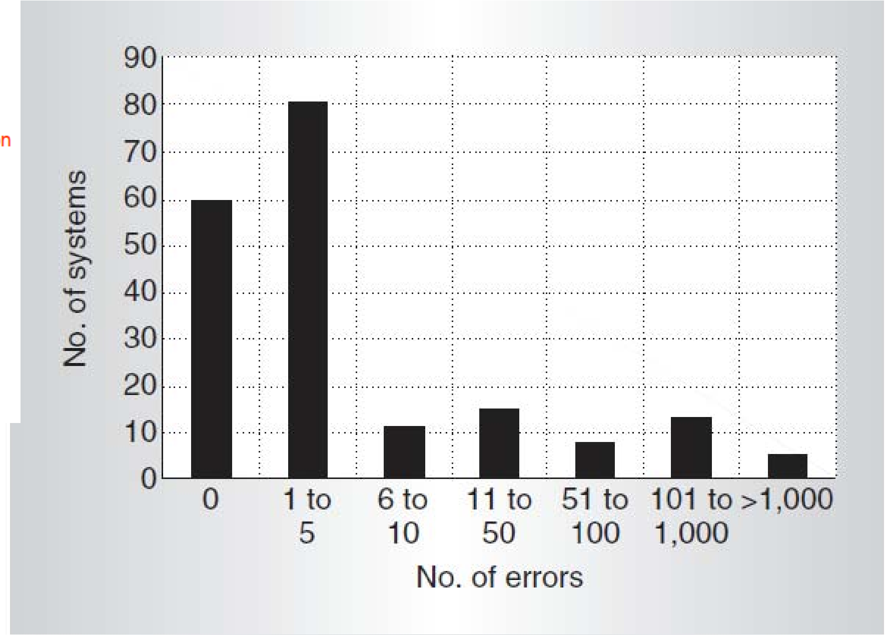
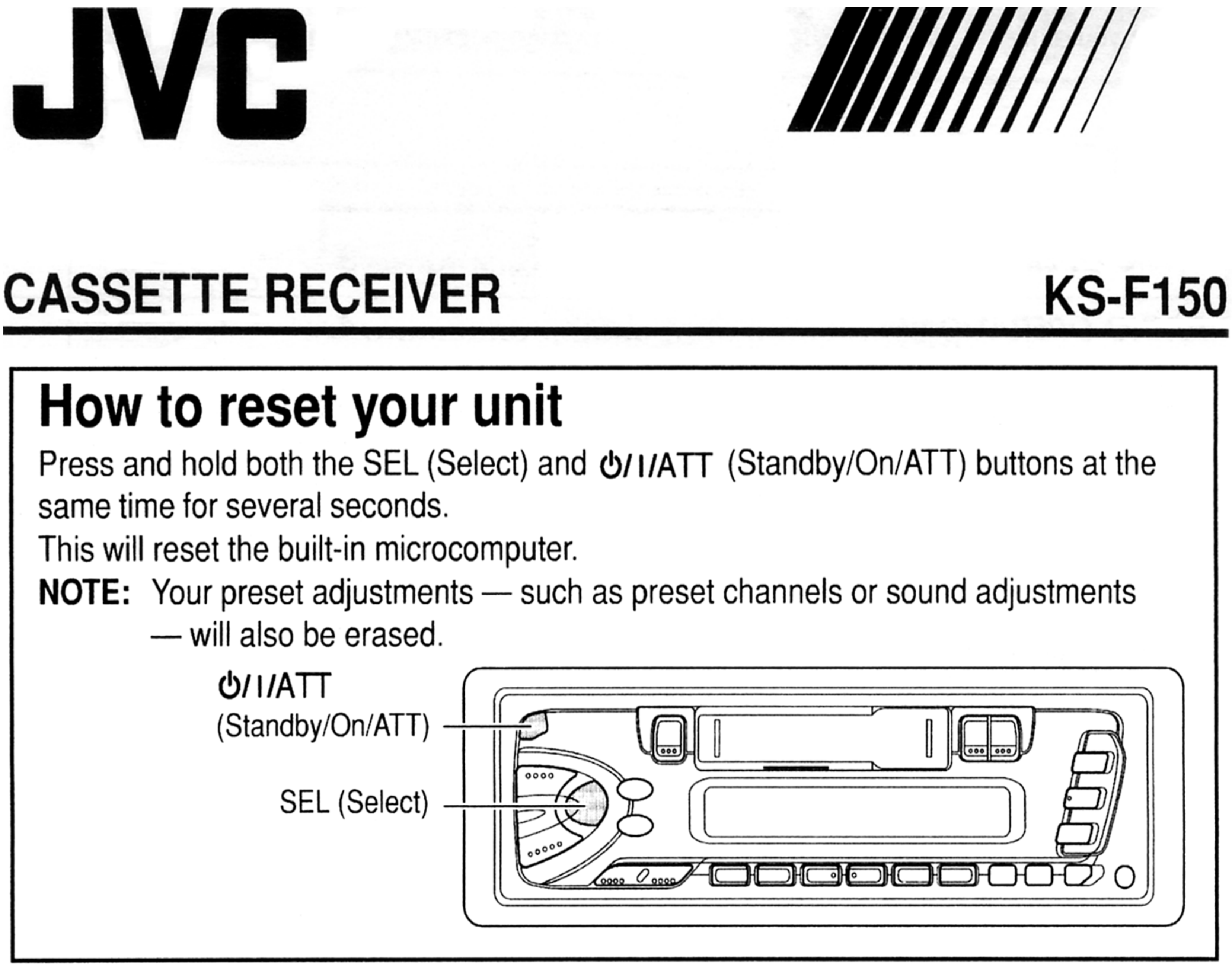


Figure 2. Histogram of the number of memory single-bit errors reported by 193 systems over 16 months.

(Constantinescu 2003, p. 16)



JVC

CASSETTE RECEIVER KS-F150

How to reset your unit

Press and hold both the SEL (Select) and ⏻/I/ATT (Standby/On/ATT) buttons at the same time for several seconds.

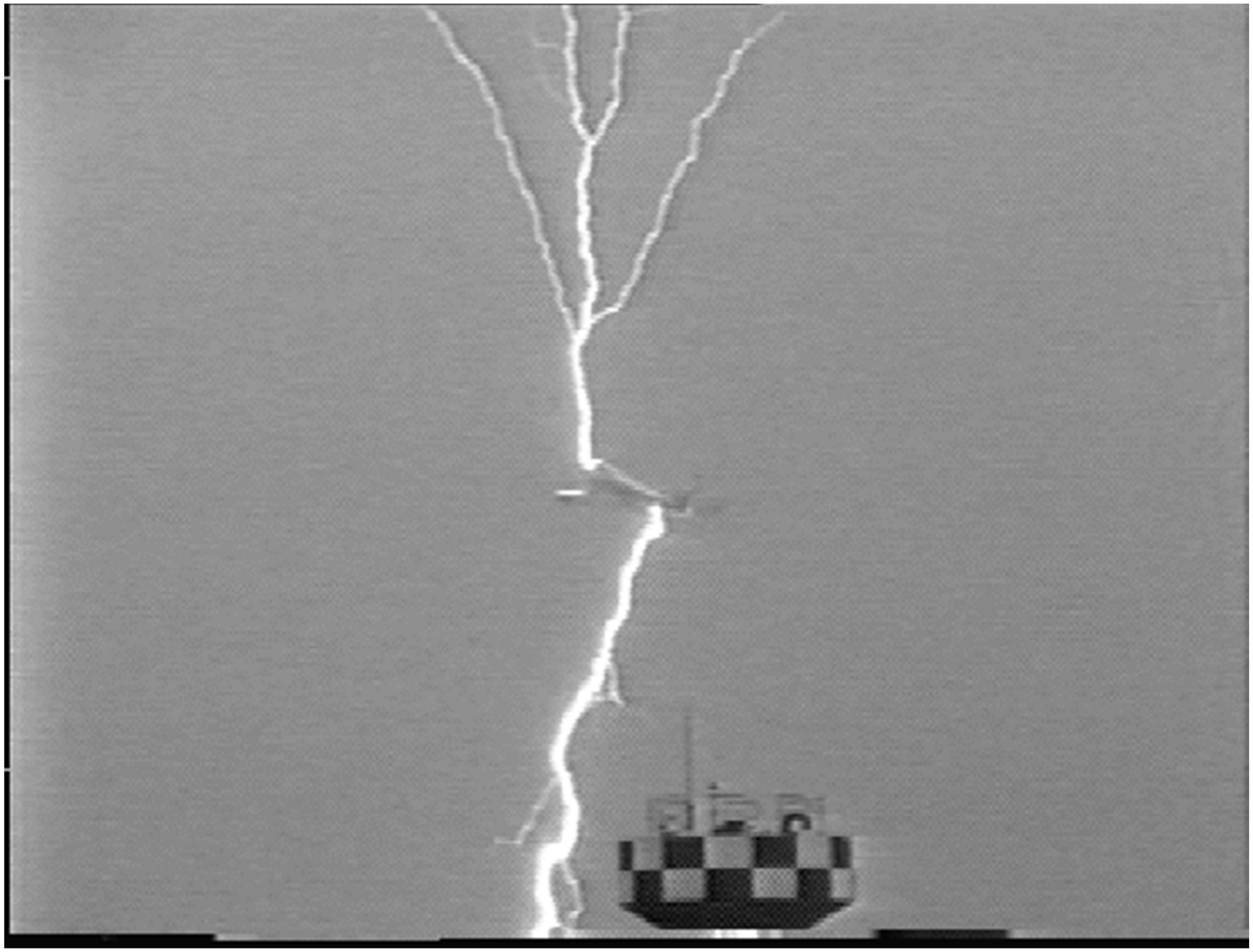
This will reset the built-in microcomputer.

NOTE: Your preset adjustments — such as preset channels or sound adjustments — will also be erased.

⏻/I/ATT
(Standby/On/ATT)
SEL (Select)

(This one is for real! Circa 1992)

Each Airplane Gets Hit By Lightning About Once/Year



http://www.crh.noaa.gov/pub/ltg/plane_japan.html

also: plane_fast_lightning_animated.gif

Tandem Environmental Outages

◆ Extended Power Loss	80%
◆ Earthquake	5%
◆ Flood	4%
◆ Fire	3%
◆ Lightning	3%
◆ Halon Activation	2%
◆ Air Conditioning	2%

- Total MTBF about 20 years
- MTBAoG* about 100 years
  - Roadside highway equipment will be more exposed than this

* (AoG= “Act Of God”)

IBM 3090 Fault Tolerance Features

Reliability

- Low intrinsic failure rate technology
- Extensive component burn-in during manufacture
- Dual processor controller that incorporates switchover
- Dual 3370 Direct Access Storage units support switchover
- Multiple consoles for monitoring processor activity and for backup
- LSI Packaging vastly reduces number of circuit connections
- Internal machine power and temperature monitoring
- Chip sparing in memory replaces de

Availability

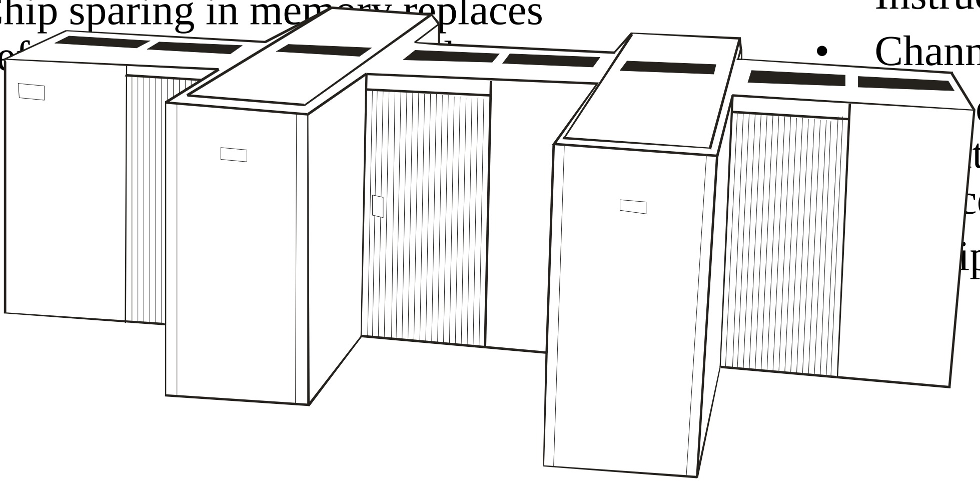
- Two or tour central processors
- Automatic error detection and correction in central and expanded storage
- Single bit error correction and double bit error detection in central storage
- Double bit error correction and triple bit error detection in expanded storage
- Storage deallocation in 4K-byte increments under system program control
- Ability to vary channels off line in one channel increments
- Instruction retry
- Channel command retry

detection and fault isolation

ts provide improved recovery and

ceability

ipath I/O controllers and units



More IBM 3090 Fault Tolerance

Data Integrity

- Key controlled storage protection (store and fetch)
- Critical address storage protection
- Storage error checking and correction
- Processor cache error handling
- Parity and other internal error checking
- Segment protection (S/370 mode)
- Page protection (S/370 mode)
- Clear reset of registers and main storage
- Automatic Remote Support authorization
- Block multiplexer channel command retry
- Extensive I/O recovery by hardware and control programs

Serviceability

- Automatic fault isolation (analysis routines) concurrent with operation
- Automatic remote support capability - auto call to IBM if authorized by customer
- Automatic customer engineer and parts dispatching
- Trade facilities
- Error logout recording
- Microcode update distribution via remote support facilities
- Remote service console capability
- Automatic validation tests after repair
- Customer problem analysis facilities

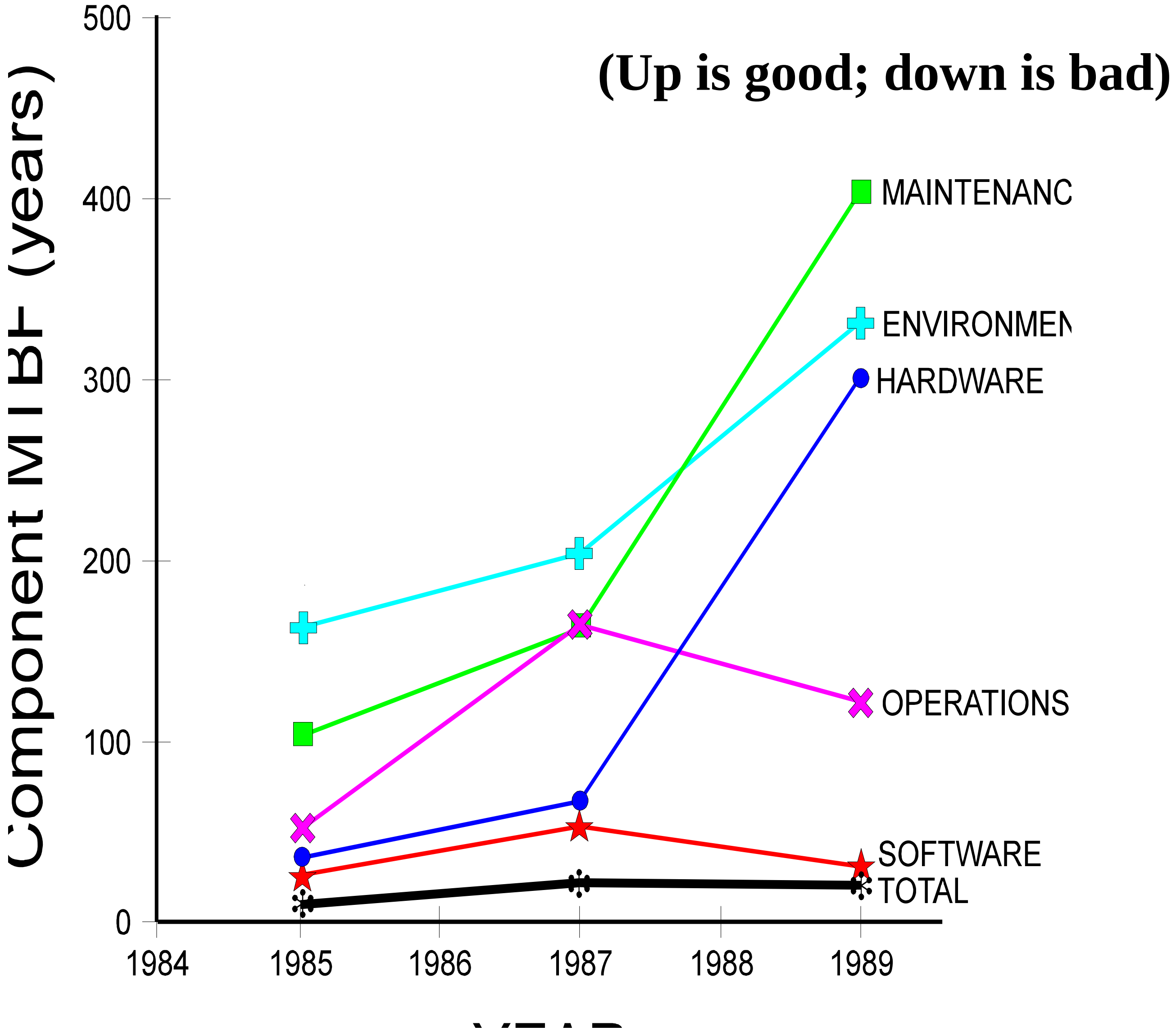
IBM 308X/3090 Detection & Isolation

- **Hundreds of Thousands of isolation domains**
  - 25% of IBM 3090 circuits for testability -- only covers 90% of all errors
- **System assumes that only 25% of faults are permanent**
  - If less than two weeks between events, assume same intermittent source
  - Call service if 24 errors in 2 hours
- **(Tandem also has 90% FRU diagnosis accuracy)**
- **Why doesn’t your laptop have memory parity?**
- **Does your web server have all these dependability features?**

Approximate Consumer PC Hardware ED/FI

This space intentionally blank

Tandem Causes of System Failures



Typical Software Defect Rates

- **Typical "good" software has from 6 to 30 defects per KSLOC**
  - (Hopefully most of these have been found and fixed before release.)
  - Best we know how to do is about 0.1 defects/KSLOC for Space Shuttle

- **Let's say a car component has 1 million lines of code.**
  - That's 6,000 to 30,000 defects, BUT
    - We don't know severity of each one
    - We don't know how often they'll be activated (usage profile)

  - So, we have no idea how to convert these defects into a component failure rate!
    - You can go by operational experience…
    - … but that doesn't address the risk of novel operating situations

  - AND, there is still the issue of requirements defects or gaps

Embedded Design Time Fault Detection

- **Follow rigorous design process**
  - Good software maturity level
  - Clear and complete specifications
  - FMEA, FTA, safety cases, other failure analysis techniques

- **Follow rule sets for more reliable software**
  - No dynamic memory allocation (avoids “memory leaks”)
  - Follow guidelines such as MISRA C standard (*http://www.misra.org.uk/*)
  - Target and track software complexity to manage risk

- **Extensive testing**
  - Development testing
  - Phased deployment with instrumentation and customer feedback
  - Adopt new technology slowly (let others shake out the obvious bugs)

Embedded Runtime Fault Detection Techniques

- **Watchdog timer (hung system; real-time deadline monitoring)**
  - Software resets timer periodically
  - If timer expires before being reset it restarts the system
- **Compute faster than is required**
  - Perform periodic calculations 2x to 10x faster than needed
    - Send network messages more often than needed and don’t bother to retry
    - Compute control loops faster than system time constants
  - Let inertia of system damp out spurious commands
    - Works well on continuous system with linear and reversible side effects
- **Rely upon user to detect faults**
  - User can re-try operation; BUT, not effective on unattended systems
- **Periodic system reboot & self-test to combat accumulated faults**
- **Avoid connection to Internet to avoid malicious faults**

Critical System Fault Detection

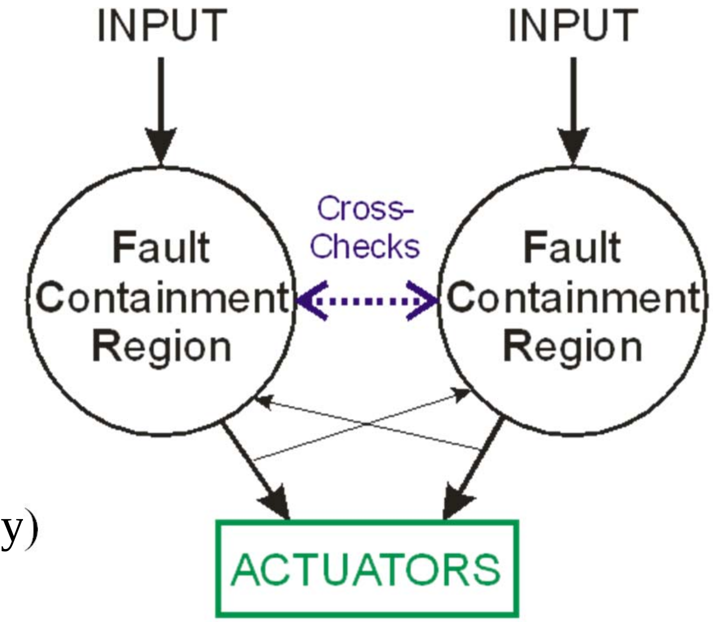
- **Concurrent detection: detects & reports error during computation**
  - Parity on memory, data buses, ALU, etc.
  - Software flow checking
    - Did software hit all required checkpoints?
  - Multi-version comparison to detect transient hardware faults
    - Compute multiple times and compare

- **Pre-emptive detection**
  - Idea is to *activate* latent faults before they can affect real computations
  - Execute self-test (perhaps periodic reboot just to exercise self-test)
    - Hardware tests
    - Data consistency checks
  - Exercise system to activate faults
    - Memory scrubbing daemon that reads all memory locations periodically

- **A single CPU can’t detect and mitigate all of its own faults**
  - Getting a safe/ultra-dependable system requires duplex computing path

2-of-2 Systems: Achieving "Fail-Fast/-Silent"

- **A technique to provide high-reliability safety systems**
  - Reliability theory often requires "fail-fast/fail-silent" behavior
  - Get that via cross-checking pairs of components
  - Note: doesn't improve reliability; it improves *safety* by killing off faulty components quickly



- **Assumptions:**
  - System is safe or otherwise "OK" if component fails fast/silent
    - Might be accomplished via shutdown
  - Faults occur independently and randomly
    - This means assuming "perfect" software
    - Failed common components (e.g., power supply) cause fail-fast/-silent behavior
  - Comparison circuits don't have a common failure mode
    - Each component checks the other's output independently
    - Both outputs needed to actuate system

Post-Modern Reliability Theory

- **Pre-WWII: mechanical reliability / "weakest link"**
- **"Modern" reliability: hardware dominates / "random failures"**

- **But, software & people matter! ("post-modern" reliability theory)**
  - Several schools of thought; not a mature area yet
  - Still mostly ignores people as a component in the system

1) Assume software never fails
   - Traditional aerospace approach; bring lots of $$$$ and cross your fingers

2) Assume software fails randomly just like electronics
   - May work on large server farms with staggered system reboots
   - Doesn't work with correlated failures -- "packet from Hell" or date rollover

3) Use software diversity analogy to create M of N software redundancy
   - Might work at algorithm level
   - Questionable for general software
   - Pretty clearly does NOT work for operating systems, C libraries, etc.

4) Your Ph.D. thesis topic goes here: ________________

Conclusions

- **Parallel and serial reliability equations**
  - There is genuine math behind using redundant components

- **Historically, goals of 100% unattainable for:**
  - Fault detection/isolation
  - Availability
  - Design correctness
  - Isolation from environmental problems
  - Procedural design correctness & following procedures

- **The biggest risk items are people & software**
  - But we're not very good at understanding software reliability
  - We understand people reliability, but it's not very good

Historical Perspective: Apollo 11 Lunar Landing

[Rocket engine burning during descent to Lunar Landing]

- **102:38:26 Armstrong:**
  (With the slightest touch of urgency) **Program Alarm.**
- **102:38:28 Duke: It's looking good to us. Over.**
- **102:38:30 Armstrong: (To Houston) It's a 1202.**
- **102:38:32 Aldrin: 1202. (Pause)**
  [Altitude 33,500 feet.]

The 1202 program alarm is being produced by data overflow in the computer. It is not an alarm that they had seen during simulations but, as Neil [Armstrong] explained during a post-flight press conference “In simulations we have a large number of failures and we are usually spring-loaded to the abort position. And in this case in the real flight, we are spring-loaded to the land position.”

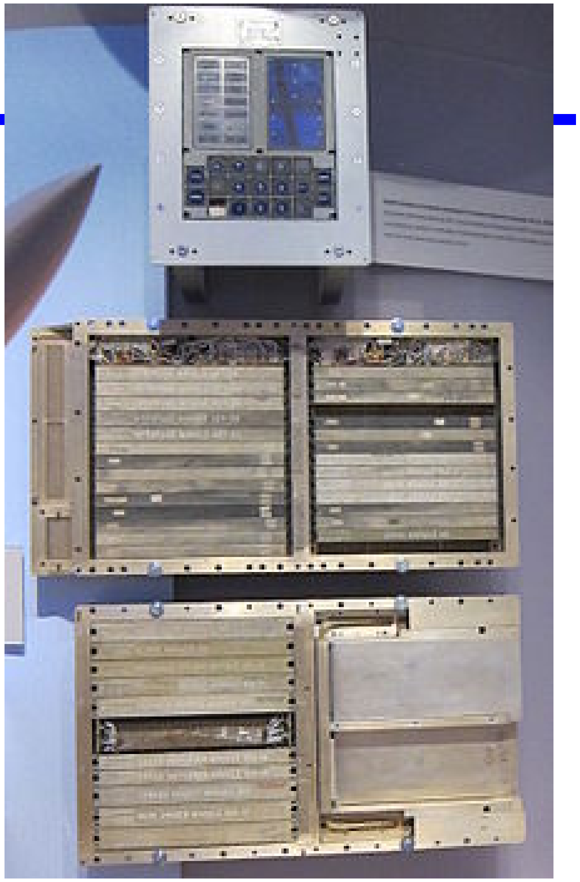
In Houston, Steve Bales, the control room's expert in the LM guidance systems, has determined that the landing will not be jeopardized by the overflow. The overflow consists of an unexpected flow of data concerning radar pointing. The computer has been programmed to recognize this data as being of secondary importance and will ignore it while it does more important computations.

[Apollo mission logs]

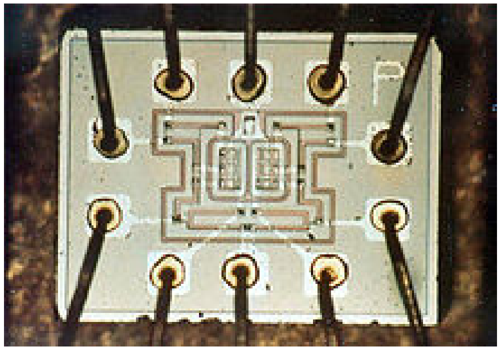
Video of Apollo 11 Landing

- **At the time all we had was audio – no live TV during the landing**
  - 11 min. clip; HBO mini-series, but accurate

- **Things to note:**
  - Collins is in the command module
  - Armstrong & Aldrin in Eagle ➔ Lunar Lander
  - 1201 & 1202 alarms light up the "abort mission" warning light
    - Computer/human interface was just a bunch of digits on a display panel
    - Total of five of these alarms (three shown in the HBO version of events)
  - At zero seconds of fuel remaining they're supposed to abort
    - Jettison lower half/landing stage and return to orbit
    - Q: for Apollo 11, how many seconds of fuel were left when they landed?



[Wikipedia]



Dual NOR Gate

A significant portion of the budget for maintaining elevators is spent on the time mechanics spend driving between repair jobs. This is a prime motivation for doing preventive maintenance and adding redundancy to increase reliability, thus eliminating the need for unscheduled repair visits. Because mechanics have to visit each elevator once a month to do adjustments and safety checks, it is highly advantageous to also do any repairs during that same visit. Note that most elevators are repaired under a maintenance contract, so any cost savings increase profitability for the repair service vendor, and any excess costs due to repeated mechanic visits must similarly be absorbed by the service vendor, reducing profits.

For this problem assume:

- Elevators are built by the factory with a single door motor shared by both doors – the independent door motors in the project assignment cost $500 to add, and this cost is absorbed by the maintenance company as a long-term investment in improving reliability. (If the single motor version fails the car traps passengers.)
- An emergency trip to an elevator by a mechanic costs $200 due to disruption in schedule and overtime pay.
- A non-emergency trip can be scheduled for the next business day when only one of two doors fails. A non-emergency trip costs $150.
- A mechanic visits every elevator every 30 days (= 1 “month” for this problem) for preventive maintenance and there is no extra cost for making a repair as well on such a visit.
- The failure rate of an elevator door motor is 237 failures per million operating hours (lambda = $237 * 10^{-6}$ per hour), and is the same per motor for single and dual motor elevator doors. Assume operation 20 hours/day, 30 days per month (and assume that the lambda is with respect to general elevator use, not just for the times the door motor is actually spinning).

If the elevator were unmodified (i.e., for an elevator with only a single door motor), how many months on average would it take per elevator to incur $500 in costs from emergency trips, assuming one emergency trip were required for each and every door motor failure. This computes the break-even point for investing in the second motor. (Answer should be in # of 30-day “months”.)

20 hours/day * 30 days = 600 hours/month * 237 * 10 $^{-6}$ = 0.142 failures/month.
$500 = 2.5 emergency trips. 2.5 / 0.142 = 17.6 months to break even

A new design is proposed that involves adding both a second motor and a cross-coupling device so that both doors will open even if one motor has failed. It is hoped that this will enable deferring motor maintenance until the next regularly scheduled maintenance visit, reducing costs. However, if elevator doors completely fail before they are repaired it will cause serious problems with customer relations. The cross-coupling device has a failure rate lambda of 24 per million operating hours (ten times better than the motors). Given that both doors are working after each maintenance visit, what is the probability that the cross-coupled door mechanism will still be able to operate the doors at the time of the next maintenance visit? (This is asking for a probability value that the doors won't fail completely between maintenance visits; carry all computations to 6 significant digits.)

This is a parallel + serial system, and mission time is 20 * 30 = 600 hours.

$R(600)_{motor} = e^{-\text{lamba*t}} = e^{-600*237e\text{-}6} = 0.867448$

$R(600)_{coupler} = e^{-\text{lamba*t}} = e^{-600*24e\text{-}6} = 0.985703$

Reliability of pair of motors is $1\text{-}[(1\text{-}R(600)_{motor})\text{^}2] = 1\text{-}[(1\text{-}0.867448)\text{^}2] = 0.982430$

Reliability of serial system (pair of motors + cross-coupler) = $R(600)_{motor\_pair} * R(600)_{coupler} =$
$0.982430 * 0.985703 = 0.968384$

what this means in practice is that you'll get a complete door failure on every elevator about once every 31.6 months if you don't repair motors as they fail instead of waiting for the next monthly maintenance visit.