Fault Tolerant Computer Architecture

Introduction

Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

ACACES 2010

Goals & Expected Background

• Course goal: learn how to protect computer hardware from faults
  – Fundamental concepts
  – Concrete examples
  – Real systems
  – Current ideas from academic papers

• Expected background: basic processor architecture
  – Dynamically scheduled processor cores
  – Cache coherent shared memory systems
  » Could sneak by without this, but it would help
Course Topics

- Course loosely follows Synthesis Lecture
- Four aspects to fault tolerance
  
<table>
<thead>
<tr>
<th>Tuesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect Errors</td>
<td>Recover</td>
<td>Self-Repair</td>
</tr>
<tr>
<td>Determine that something went wrong</td>
<td>Resume execution from a safe point</td>
<td>Keep the problem from repeating</td>
</tr>
<tr>
<td>Diagnose Faults</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Figure out the cause of the problem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Course Topics – More Details

- Introduction (Monday)
  - Motivation
  - Goals and challenges
  - Faults and their causes
  - Error models
  - Trends
- Error detection (Tuesday)
- Error recovery (Thursday)
- Diagnosis and self-repair (Friday)
- Conclusions and the future (Friday)
Outline (of Introduction)

• Motivation, goals, and challenges
• Some examples of fault tolerant systems
• Faults

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!), cellphones, game consoles, etc.
"Stuff" Happens

- We wouldn’t need fault tolerance otherwise!

- **Physical problems**
  - Cosmic radiation
  - Melted wire

- **Design flaws**
  - Incorrect logic (e.g., Pentium's FDIV, AMD's quad-core TLB bug)

- **Software bugs**

- **Operator error**
  - Incorrect software installation
  - Accidental use of `rm -R *`

- **Malicious attacks**

---

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
  
  - **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
    - Requires **redundancy** (a.k.a. spares or backups)
    - Example: having a co-pilot on an airplane
    - **KEY:** no fault tolerance without redundancy of some kind
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 = \( \frac{MTTF}{MTTF + MTTR} \)
    – 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
- **Liveness**: won’t stop running
- **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
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 (Intel CoreDuo and a Mac)
  – Run them more slowly to avoid certain hardware faults
  – Use 2 different operating systems (Windows 7 and Snow Leopard)
    » No, neither one can be Vista
  – Use 2 versions of PowerPoint written by 2 independent groups

Fault Tolerance & Performance

• Most important not to degrade performance during fault-free operation
  – This is the common-case \( \Rightarrow \) 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)
Outline

• Motivation, goals, and challenges
• Some examples of fault tolerant systems
• Faults

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 \rightarrow 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

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 MTTReboot (also by me): 1 week
- Expected MTTRe-install: 6 months
- Expected probability of it successfully coming out of “hibernation” mode: 0 (std dev = 0, too)
Outline

• Motivation, goals, and challenges
• Some examples of fault tolerant systems
• Faults
  – Intro and terminology
  – Causes of faults
  – Error models

Why We Study Faults

• Know thy enemy!
• If we don't understand faults, it is much more difficult to design systems that can tolerate them
  – We at least have to know how to model them
Faults

• **Fault:** incorrect state of hardware resulting from physical defect, physical phenomenon, or design bug

• Faults introduced during manufacturing (at time=0)
  – Bad solder connection between chip pin and motherboard
  – Broken wire within chip

• Faults that occur during operation (after t=0)
  – Cosmic ray knocks charge off DRAM cell

• Faults introduced during system design (before t=0)
  – Pentium’s incorrect floating point division design

Errors

• **Error:** manifestation of a fault
  – Bit in main memory is a 0 instead of a 1 (due to cosmic ray)

• But not all faults lead to errors!
  – Trees falling in empty forests don’t make sounds

• Example of **masked** fault
  – Cosmic ray knocks charge off logic signal, but after it had been correctly latched in and saved
Failures

- **Failure**: system level effect of an error (user-visible)
  - System produces incorrect result of computation (e.g., 2+2=5)
  - System “hangs” (e.g., Blue Screen of Death)

- Not all errors lead to failures!

- Example of masked error
  - Bit flip in memory location that’s not accessed again

Physical Failures During Lifetime

- Three phases of system lifetime
  - Infant mortality
  - Normal lifetime
  - Wear-out period

- Physical failures follow famous “bathtub curve”
Fault $\rightarrow$ Error $\rightarrow$ Failure Example

- Cosmic ray knocks charge off of DRAM cell
  $\rightarrow$ Error: bit flip in memory
  $\rightarrow$ Failure: computation produces incorrect result

Duration of Faults/Errors

- **Transient (soft):** occurs once and disappears
  - E.g., Cosmic ray knocks charge off transistor $\rightarrow$ bit flip
  - Tend to be due to transient physical phenomena
  - Also known as Single Event Upset (SEU)

- **Intermittent:** occurs occasionally
  - E.g., Loose connection $\rightarrow$ occasionally open circuit

- **Permanent (hard):** occurs and doesn’t go away
  - E.g., Broken connection $\rightarrow$ always open circuit
  - Could occur at fabrication or due to in-field wearout
Types of Masking

• Logical
  – E.g., if a fault flips a bit from 0 to 1 and it is then ANDed with a bit that is 0, then the fault cannot manifest itself as an error

• Functional
  – E.g., incorrect data is produced by an instruction that gets squashed due to a branch misprediction
  – E.g., the destination register of a NOP is corrupted by a fault

Outline

• Motivation, goals, and challenges
• Some examples of fault tolerant systems
• Faults
  – Intro and terminology
  – Causes of faults
  – Error models
Physical Defects: Transient Phenomena

• Cosmic radiation
  – High energy particles that constantly bombard Earth
  – May have enough energy to disrupt charge on transistor ($Q_{\text{crit}}$)
  – Used to be only a problem for DRAM, but becoming a problem for SRAM and even for logic (as $Q_{\text{crit}}$ decreases)
  – Trends:
    » $Q_{\text{crit}}$ decreasing
    » Probability increasing that a cosmic ray that hits a transistor will disrupt its charge
    » Transistor size decreasing $\rightarrow$ smaller probability that a cosmic ray will hit a particular transistor
    » More transistors per system $\rightarrow$ greater probability of fault


Physical Defects: Transient Phenomena

• Alpha particle radiation
  – Similar to cosmic rays, but radiation comes from metal decay
  – Often, the metal housing of the computer is the source
  – Lead solder joints also a problem $\rightarrow$ want to use “old lead”
  – Trends (same as for cosmic radiation):
    » $Q_{\text{crit}}$ decreasing
    » Probability increasing that an alpha particle that hits a transistor will disrupt its charge
    » Transistor size decreasing $\rightarrow$ smaller probability that an alpha particle will hit a particular transistor
    » More transistors per system $\rightarrow$ greater probability of fault
Physical Defects: Transient Phenomena

• Electromagnetic Interference (EMI)
  – Electromagnetic waves from other sources (e.g., microwave oven, power lines, etc.) can cause transient disruptions
  – EMI can be created by the circuit itself! Called “crosstalk”
  – EMI can induce electrical current on wires and thus change the signals on wires

• There are other sources of transient faults, but they tend to be less significant

Physical Defects: Manufacturing Defects

• Manufacturing is not a perfect process, especially for microprocessors
  – It’s not easy to manufacture something with dimensions on the order of 32nm
  – Many stages of chip processing which have to be done perfectly and avoid contamination

• And testing doesn’t filter out all defective systems
  – Often impossible to test for every possible defect in a reasonable amount of time
  – Also, testing won’t detect defects that don’t manifest immediately

• Nanotechnology makes this problem even worse
Physical Defects: Manufacturing Defects

• Manufacturing flaws
  – Bad solder connection between chip and board
  – VLSI defects (e.g., broken wire, bad via, etc.)

• Trends
  – Flaws may decrease as manufacturing process matures
  – But flaws increase at start of each new process
  – Tougher to avoid VLSI defects as dimensions shrink

Physical Defects: Manufacturing Defects

• VLSI fabrication process variability
  – During fab, there’s some amount of variability in dimensions
    » Thickness of gate oxide dielectric
    » Length of channel
    » Area of via
    » Etc.
Physical Defects: Manufacturing Defects

• Variability can lead to undesirable behavior
  – Gate thickness falls below usable threshold → extra leakage current
  – Wire resistance is too high → signal too slow for clock

• Trend: variability rising as VLSI dimensions shrink
  – When dimensions are on the order of a handful of atoms, it doesn’t take much variability to cause significant problems

Go to David Brooks’s ACACES class on this topic!

Physical Defects: Operational Defects

• Permanent (hard) defects can occur during operation

• Electromigration
  – If current density is too large for wire, wire metal will “migrate” away and potentially lead to broken link
  – Exacerbated by thermal cycling due to hotter chips
  – Trend: getting worse as wires become smaller and chips become hotter
Physical Defects: Operational Defects

• Gate oxide breakdown
  – MOSFET transistor has gate oxide that insulates gate from channel
  – If this oxide breaks down, will get a short between gate and channel
  – Trend: getting worse as gate oxides become thinner (only a handful of atoms thick!)

Hardware Design Flaws: Logical Bugs

• Famous example: Intel Pentium floating point divide didn’t work in every single case due to bug in design → very costly recall
• Sun UltraSPARC III had design flaw in a special cache that meant that it couldn’t be used → loss in performance
• AMD’s quad-core Barcelona chip had design bug in TLB hardware → long, expensive delay in shipping chips
Hardware Design Flaws: Timing Bugs

- Logic is fine, but the timing analysis is flawed
- Example: clocking a processor at 3 GHz when there's a slow path that can run only at 2.8 GHz
- Timing analysis must consider critical path delay and environmental effects (operating temperature, EMI, cross-talk, etc.) to determine max operating speed
- This problem is exacerbated by process variability

Outline

- Motivation, goals, and challenges
- Some examples of fault tolerant systems
- Faults
  - Intro and terminology
  - Causes of faults
  - Fault/error models
Purpose of Fault/Error Modeling

- Model = abstraction of physical phenomenon
- Simple, tractable way to analyze effects of faults/errors
- Terminology: often more appropriate to use “error model”, but literature usually uses “fault model”
  - Thus I'll use “fault model”
- Example: “fail-stop” network switch ➔ if a fault occurs, the switch will just stop doing anything
  - Model reflects the behavior of many potential underlying faults
  - E.g., switch has short from power to ground, switch is on fire, etc.
  - Easier to work with this model than to consider all possible faults

Limitations of Fault Modeling

- Garbage In ➔ Garbage Out
- If model doesn’t reflect behavior of a type of fault, an analysis based on that model won’t handle that type of fault
- “Deflated car tire” fault model doesn’t handle case where brakes stop working
  - Having a spare tire to handle this fault isn’t helpful
- More computer-y example: Fail-stop fault model for network switch doesn’t handle case where switch starts routing packets incorrectly
  - And this fault model represents several realistic underlying faults
Using Fault Models

• Testing
  – Instead of testing for every possible fault, test for faults that are in assumed fault model

• System design
  – Instead of designing system to tolerate every possible fault, design system to tolerate faults that are in assumed fault model

Low-Level Hardware Fault Modeling

• Stuck-at-x fault model (x=0 or x=1)
  – Useful for modeling faults in state or on wires
  – Many real faults can be modeled this way (but not all!)
  – Most prevalent model, particularly for testing purposes
  – Easy to use and analyze

• Stuck-open and/or stuck-closed
  – Models faults in switches (i.e., transistors)
Low-Level Hardware Fault Modeling

• Coupling (bridging) faults
  – Signal on one wire is equal/inverse (coupled) to signal on other wire
  – Models effects of short circuits or cross-talk

• Fail-stop faults
  – Assumes that faulty component stops doing anything
  – Often used for modeling entire processor or network switch

Transition and Delay Faults

• Most fault models consider signal value on wire or transistor (e.g., stuck-at-1)
• Transition faults model the situations in which correct value is produced on given wire or transistor … but not at right time
• Delay faults model the situations in which paths (not just given nodes) exhibit incorrect timing behavior
• These fault models model many known physical phenomena, but their effects are generally much less tractable to analyze
Other Fault Models

• Can we develop fault models that aren’t so low-level?
  – Do we need to consider each transistor?
  – How about logic gate level?
  – Or even coarser granularity (e.g., ALU, cache line, processor core)

How Many Faults at Once?

• Many fault models include assumption that only one fault can occur at given time
  – Helps to make analysis more tractable
  – E.g., “single stuck-at fault” model

• Reasonable assumption if:
  – Faults are rare
  – System doesn’t require extreme reliability
  – Faults are detected and, if necessary, removed quickly

• The problem with latent faults
  – Fault occurs, but isn’t detected
  – Later, a “single” fault occurs, but this is now double fault scenario
  – If you only plan for single faults, then this situation is a problem
The Art of Choosing a Fault Model

• What is an appropriate hardware fault model for:
  – Mainframe system
  – Desktop processor
  – iPhone
  – Embedded controller for anti-lock brakes
  – Embedded controller for toaster oven

• Remember: only include those faults that matter for your situation
  – Do you care about transient errors in an iPhone?
  – Do you care about transistor wearout in a laptop?

Outline

• Motivation, goals, and challenges
• Some examples of fault tolerant systems
• Faults

• Preview of Coming Attractions
  – Tomorrow (Tuesday): Error detection
  – Thursday: Error recovery
  – Friday: Diagnosis & self-repair, future of fault tolerant architecture
Recommended Reading

• “Why Do Internet Services Fail, and What Can be Done About It?,” Oppenheimer et al. USITS 2003.

Fault Tolerant Computer Architecture

Error Detection

Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

ACACES 2010
Where We Are in the Course

Tuesday
Detect Errors
Determine that something went wrong

Thursday
Recover
Resume execution from a safe point

Diagnose Faults
Figure out the cause of the problem

Self-Repair
Keep the problem from repeating

Friday

Error Detection

• Detection is most important part of fault tolerance
  – Can’t tolerate a fault if you don’t know it happened!

• Outline of today’s class
  – General concepts in error detection
  – Microprocessor cores
  – Cache/memory
  – Multiprocessor-specific issues

• Note: some fault tolerance solutions integrate detection and recovery (e.g., TMR) → we will cover these solutions today
Error Detection in General

- Must have redundancy to detect errors
  - Fundamental requirement
  - But what kind of redundancy?
- 4 types of redundancy
  - Physical (spatial)
  - Temporal
  - Design
  - Information

Outline

- General concepts
  - Physical redundancy
  - Temporal redundancy
  - Design redundancy
  - Information redundancy
  - End-to-end vs. localized detection
- Microprocessor cores
- Cache/memory
- Multiprocessor-specific issues
Physical (Spatial) Redundancy

• Physically replicate a module
  – Most obvious approach

• Design issues
  – How many replicas are needed?
    » For error detection?
    » For error correction?
  – How are errors detected/corrected?
  – Is the redundancy “active” or “passive”?

• Canonical example: triple modular redundancy (TMR)
  – 3 replicas
  – Errors corrected by majority voter
  – Redundancy is passive (no special action taken if error detected)

Physical Redundancy: TMR

• Strengths
  – Tolerates error in any single module
  – Tolerates soft and hard errors
  – Simple design
  – Small performance penalty, even when faults occur

• Weaknesses
  – Can’t tolerate multiple faults
    » Can’t tolerate any faults after a single latent hard fault
  – Expensive hardware (3x cost)
  – Uses lots of power (approx 3x power of unprotected)
  – Single point of failure at voter
  – Can’t tolerate errors due to design faults … why not?
Physical Redundancy: NMR

• N-modular redundancy (N is an odd integer)
  – Why is N odd?
• Can tolerate more errors than TMR
  – Tolerates up to N/2 – ½ errors
• Cost = N*cost of module
• Still has single point of failure at voter!
  – But voter is simple and can be designed to be very robust
• One solution to single voter problem
  – “Restoring organ” = TMR with triplicated voter
  – How does this help?

Physical Redundancy: Boeing 777

• Boeing 777 requires near-perfect reliability
• Its main flight computer:
  – Has 3 identical units in TMR configuration
  – Each of these units has 3 processors in TMR configuration
  – The three processors in each unit are heterogeneous!
    » Intel 80486 (the x86 before the original Pentium)
    » Motorola 68040
    » AMD 29050

Called "triple-triple redundant 777 primary flight computer" in paper by Yeh, 1996.
Physical Redundancy: Active vs. Passive

• NMR is passive, since system doesn’t really do anything different when error is detected
• With active redundancy, the system detects the error, locates/diagnoses it, and reconfigures to tolerate it

• Standby sparing has at least two redundant modules
  – Operational module has internal error detection mechanism
  – If a hard error is detected, the system reconfigures to use spare
    » “Cold standby”: standby was inactive and must be warmed up
    » “Hot standby”: standby was active and is in correct state

More Active Redundancy

• Pair-and-spare
  – Like standby sparing, except each module is a pair
  – This pair compares outputs to detect errors
  – If error detected, a spare module (i.e., pair) is configured in

```
  module   compare   module
         ↓      ↓        ↓
    module   compare   module
         ↓      ↓        ↓
  ▼ module  ▼ module  ▼ module
    ▼ switch
```

(C) 2010 Daniel J. Sorin
Hybrid Physical Redundancy

• Combine passive and active redundancy
• Example: NMR with spares
  – Let’s say we have 5 replicas
  – Organize 3 into TMR scheme
  – Save other 2 for use as spares

After first hard fault, map in a spare
Hybrid Physical Redundancy

- Combine passive and active redundancy
- Example: NMR with spares
  - Let’s say we have 5 replicas
  - Organize 3 into a TMR scheme
  - Save other 2 for use as spares
  - After first hard fault, map in a spare
  - After second hard fault, map in other spare
  - Even after 2 hard faults, can tolerate a third fault
  - Thus, system can tolerate 3 faults that occur sequentially
  - Recall that 5MR can only tolerate 2 faults

Watchdog Timers

- So far, we’ve figured out how to detect when something is wrong … but how do we detect when we’re not doing anything at all?
- Watchdog timer monitors a module and triggers a recovery if the module doesn’t do anything in a given amount of time
  - E.g., put watchdog timer on microprocessor bus
- Who watches the watchdog?
  - If we assume single fault scenario, then this usually isn’t a problem
  - But what if watchdog has hard fault that causes it to never timeout and trigger a recovery?
Outline

- **General concepts**
  - Physical redundancy
  - Temporal redundancy
  - Design redundancy
  - Information redundancy
- **Microprocessor cores**
- **Cache/memory**
- **Multiprocessor-specific issues**

Temporal Redundancy

- Replicate the actions on a hardware module using the same module, but at a different time
- Effective for tolerating transient faults
  - Can this help at all for hard faults?
- Can be performed at various granularities
  - Perform same addition twice on same adder
  - Execute same instruction twice on same core
  - Execute same group of instructions twice on same core
Design Redundancy

- Use multiple different designs to guard against design fault (bug) in any of them
- Disadvantage: very costly to have multiple different designs
  - Also: doesn’t tolerate faults in design specification – only tolerates faults in implementations of that specification
- Examples
  - Boeing 777 uses processors from Motorola, Intel, and AMD
  - “N-version programming” – have N design teams develop N different versions of a piece of software

Information Redundancy

- For a given k-bit piece of information, add r check bits to it that make it possible to detect/correct errors in the original k-bit information
- Example: parity bit
  - By adding a single bit to a word of information, we can detect any single-bit error in it
- Example: checksum
  - Sender computes a number (checksum) that nearly uniquely identifies long stream of information that it sends, and it sends checksum along with information
  - Receiver uses same algorithm to compute checksum on information it receives (not including checksum). If receiver’s checksum matches sender’s checksum, then information is error-free
Basic Idea

- Start with k-bit data word
- Add r check bits
- Total = n-bit codeword \((n=k+r)\)
- Map \(2^k\) data words to \(2^n\) sized codeword space
  - Key idea: spread data words far apart from each other in codeword space
- Overhead = \(r/k\)
  - E.g., for (single-bit) parity, the overhead is \(1/k\)

Hamming Distance

- Hamming distance (HD): number of bits in which two words differ from each other
- E.g., 0010 and 1110 have a Hamming distance of 2
- For a group of codewords, the minimum HD between any two codewords determines the code’s ability to detect and/or correct errors
  - This is a fundamental rule, not just some ad-hoc reasoning
- Can detect up to t-bit errors if HD \(\geq t + 1\)
- Can correct up to t-bit errors if HD \(\geq 2t + 1\)
Hamming Distance Visual: HD=2

Can detect single errors

Hamming Distance Visual: HD=3

Can correct single errors
Can detect double & single errors
Arithmetic Codes

- Codes that are preserved by arithmetic operations
  - If $X$ and $Y$ are codewords, then $Z = F(X,Y)$ is a codeword
- Arithmetic codes let us detect errors in ALUs
- Two types of codes, where $f(X)$ is the encoding of $X$ and $C(X)$ is the check symbol computed from $X$
  - **Separable**: $f(X) = \text{concatenation of } X \text{ and } C(X)$ denoted $X, C(X)$
  - **Non-separable**: $f(X) \neq X, C(X)$
- **Why is separability a desirable feature?**
  - Think about hardware implementation issues
- Example (assume addition is performed modulo $M$)
  - AN code: $f(X) = A^X$
  - $A \cdot (X+Y \mod M) = (A^X + A^Y) \mod AM$

Self-Checking Circuits

- **What properties/invariants can we build into circuits such that codeword inputs do not lead to codeword outputs in the presence of faults?**
- **Self-testing** circuit: for every fault from a prescribed set, the circuit produces a non-codeword output in response to at least one codeword input
- **Fault-secure** circuit: for every fault from a prescribed set, the circuit never outputs an incorrect codeword in response to codeword inputs
- **Totally self-checking**: self-testing AND fault-secure
Implementing EDC/ECC in Hardware

• Where does EDC/ECC get used?
  – Disk, CD-ROM
  – Memory (DRAM, SRAM)
  – Buses
  – Network

• Tradeoff between EDC and ECC

• ECC: Forward error recovery
  – Often on critical path, so can slow down even fault-free system

• EDC: Backward error recovery
  – Detecting error leads to recovery (can be slow)

• So would you use ECC or EDC in your L1 cache?
  – How about in DRAM?

End-to-End vs. Localized Error Detection

• Can use any of these 4 types of redundancy to detect errors ... but at what granularity?

• Localized
  – In a system with N modules, each module has localized, dedicated error detection mechanism

• End-to-end
  – Check high-level invariants that are true of entire system (from “end to end”)

• Canonical example: communication between two endpoints with several switches in between them
  – Localized: check each switch and each link
  – End-to-end: sender adds checksum to message and destination checks it (with no checking in between endpoints)
The End-to-End Argument


• End-to-end detection is:
  – Necessary: some errors can’t be detected any other way
  – Cheaper: no need to use hardware/power to detect low-level faults
  – Not helpful for diagnosis: when error detected, no idea where it occurred

• Challenge? Finding the “ends”
  – What level is high enough to check? Microarch? Arch? Higher??
  – What invariants do we check at these levels?

More End-to-End Examples

• Detect bit flip in register?
  – No! Wait to see if it causes violation of expected control flow
  – No! Wait to see if it causes program crash

• Detect bit flip in coherence request?
  – No! Wait to see if it causes violation of memory consistency model
Outline of Today’s Class

• General concepts in error detection
• Microprocessor cores
• Cache/memory
• Multiprocessor-specific issues

Microprocessor Cores

• No longer assuming we’re protecting a black box
  – Take advantage of our knowledge of what the core does
• Some approaches we’ll look at next
  – Lockstepped redundant cores
  – Redundant multithreading (without lockstepping)
  – Runtime checking of invariants
  – Using software to detect hardware errors
**Tightly Lockstepped Cores**

- Classic example of physical redundancy
- Compare results of each instr (or group of instrs)
- Pros: logically simple, tight bound on detection latency
- Cons: >100% overhead for power, energy, hardware

---

**Redundant Multithreading**

- Two insights
  - Multithreaded cores provide inexpensive redundancy
  - Lockstepping is getting more challenging
- The idea: redundant multithreading (RMT)
- Loosely synchronized redundant threads
  - Simultaneous multithreaded (SMT) core
  - Active thread (A-thread) passes instruction results to redundant thread (R-thread)
  - R-thread holds architected state of thread
  - If results are same, that instruction can be committed by R-thread; else must re-try until results agree
RMT Design Space

Many design options for RMT
1. When to compare outputs/state of threads?
2. Should the redundant thread be complete or partial?
3. On what hardware should the threads run?

RMT: When to Compare

• As we compare more frequently ...
  + Lower error detection latency
  + Shorter “output commit problem”
    » More effort spent performing comparisons

• A comparison optimization
  – “Fingerprinting”: each thread creates a fixed-length fingerprint (i.e., checksum or signature) of its execution and the threads periodically compare fingerprints
  – Fingerprint is lossy hash of execution history
    » Greatly reduces comparison overhead
    » Slight probability of aliasing
RMT: Partial Redundancy

- Key idea: use a replica thread that isn’t complete (i.e., has fewer instructions than original thread)
- Explicitly trade off error detection coverage vs. costs (performance degradation, power, energy)

RMT: Where to Run the Threads

- Both on same multithreaded core
  - Low latency comparisons
  - Some contention between threads → lower performance
  - Vulnerable to hard faults in shared resources (incl. chipkill)
- On different cores of same chip
  - Slightly higher comparison latencies
  - No contention between threads
  - Less vulnerable but still susceptible to chipkills
- On different cores on different chips
  - Higher comparison latencies → lower performance
  - No contention between threads
  - Little vulnerability
  - Commercial example: HP NonStop Advanced Architecture
Dynamic Verification of Invariants

- **End-to-end error detection schemes**
  - At runtime, check high-level invariants instead of components
- **Example**:
  - Component: Check the logic for updating program counter (PC)
  - Invariant: Check that PC is always within certain expected range
- **So what invariants should we check?**
  - Ideally, end-to-end definition of correct behavior
  - But that might not always be possible or practical

Control Flow Checking

- **Goal**: check that dynamic control flow follows legal path
- **Assign each basic block address-independent ID**
  - ID computed from block contents
- **Embed IDs of legal successors in each block**
  - Most blocks have one or two legal successors
  - Pick correct ID at runtime

![Control Flow Diagram](image)
Dataflow Checking

![Diagram showing processor, basic block, execute, track, analyze, compare, dynamic dataflow signature, static dataflow signature, sub r4, r2, r3, mul r5, r3, r1, add r3, r5, r4.]

Argus [Meixner, Micro 2007]

- If core maintains 3 invariants, its behavior is correct
  - Control flow is correct
  - Dataflow is correct
  - Computations are correct
- Argus implementations combine checkers for each invariant to comprehensively detect errors in cores
DIVA [Austin, Micro 1999]

• Key idea: check complicated out-of-order core with simple, provably correct in-order checker core that has same ISA

• Heterogeneous physical redundancy
  – Checker is smaller than original core \(\rightarrow\) less expensive than 2MR
  – Can detect errors due to design bugs

• Challenge – how does checker core match throughput of core it is checking?

Anomaly Detection

• Can detect likely errors by detecting anomalous behaviors

• Examples of detectable anomalies
  – Unusual data values
  – Branch mispredictions for confident predictions
  – Exceptions
  – Page faults
  – Crashes
  – Unusually high amount of OS activity

• Generally inexpensive approach to detection
  – How complete can it be?
Using Software to Detect Errors in Core

• Compiler can augment program with redundant instructions and comparisons to detect errors
• Pros? Cons?

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Code with Redundancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>add r1, r2, r3</td>
</tr>
<tr>
<td>xor r4, r1, r5</td>
<td>add r11, r12, r13</td>
</tr>
<tr>
<td>store r4, 0($r6)</td>
<td>xor r4, r1, r5</td>
</tr>
<tr>
<td></td>
<td>xor r14, r11, r15</td>
</tr>
<tr>
<td></td>
<td>bne r4, r14, error</td>
</tr>
<tr>
<td></td>
<td>store r4, 0($r6)</td>
</tr>
<tr>
<td></td>
<td>// r1 = r2 + r3</td>
</tr>
<tr>
<td></td>
<td>// r4 = r1 XOR r5</td>
</tr>
<tr>
<td></td>
<td>// Mem[$r6] = r4</td>
</tr>
<tr>
<td></td>
<td>// r11 = r12 + r13</td>
</tr>
<tr>
<td></td>
<td>// r4 = r1 XOR r5</td>
</tr>
<tr>
<td></td>
<td>// r14 = r11 XOR r15</td>
</tr>
<tr>
<td></td>
<td>// if r4 !=r14, goto error</td>
</tr>
<tr>
<td></td>
<td>// Mem[$r6] = r4</td>
</tr>
</tbody>
</table>

Outline of Today’s Class

• General concepts in error detection
• Microprocessor cores
• Cache/memory
• Multiprocessor-specific issues
Cache/Memory Error Detection

• Except for L1 cache, ECC is standard solution
• At L1 cache, can use ECC or EDC
• If EDC on L1
  – Cheaper than ECC
  – Requires write-through L1 so that valid data in L2 if L1 detects error
• If ECC on L1
  – More expensive than EDC
  – Can use write-back L1
• Recent work has tried to get best of both worlds
  – Optimize for common case (error-free execution)
  – Keep EDC on L1 and the extra redundant bits required for correction in a separate small structure

Beyond EDC/ECC

• In-Cache Replication [Zhang, DSN 2003]
  – Replicate some cache blocks in otherwise unused cache frames
• Replica Cache [Zhang, ICS 2004]
  – Replicate some cache blocks in dedicated R-cache
• And more … but not going to discuss it here
Outline of Today’s Class

• General concepts in error detection
• Microprocessor cores
• Cache/memory
• Multiprocessor-specific issues

Multiprocessor Error Detection

• Multiprocessor consists of cores plus shared memory subsystem
  – Includes caches, memories, coherence controllers, interconnection network
• Could add localized checker to each structure
  – But difficult to show that sum of error detection schemes is sufficient → what about interactions between modules?
• End-to-end error detection is appealing
  – But what high-level invariants do we check?
Dynamic Verification of Memory Consistency

- Correctness of memory system defined by memory consistency model
  - If we can check that system obeys consistency model, then we can comprehensively detect errors in memory system
- Dynamic verification of memory consistency (DVMC)
  - Meixner and Sorin [ISCA 2005, DSN 2007]
  - One sub-component is dynamic verification of cache coherence

Where We Are in the Course

- Tuesday: Detect Errors
- Wednesday: Correctness of memory system defined by memory consistency model
- Thursday: Recover
- Thursday: Diagnose Faults
- Friday: Self-Repair
  - Keep the problem from repeating
Recommended Reading


Recommended Reading

Fault Tolerant Computer Architecture

Error Recovery

Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

ACACES 2010

Where We Are in the Course

- **Thursday:** Error recovery
- **Friday:** Diagnosis, reconfiguration & the future

Detect Errors
Determine that something went wrong

Recover
Resume execution from a safe point

Diagnose Faults
Figure out the cause of the problem

Self-Repair
Keep the problem from repeating
Error Recovery

- Error detection (Tuesday's class) provides safety but not necessarily liveness
- Now, what do we once we've detected an error???

Outline of Today’s Course

- Forward error recovery (FER) vs. backward error recovery (BER)
- BER in general
- Uniprocessor BER
- Multiprocessor BER
Recovering from Errors

- **Two basic approaches**
  - Forward Error Recovery (FER)
  - Backward Error Recovery (BER)
- **FER:** continue to go **forward** in presence of errors
  - Use redundancy to mask effects of errors
  - E.g., have a co-pilot that can seamlessly take over airplane
  - Tightly integrated with error detection (or subsumes it)
- **BER:** go **backward** to recover from errors
  - Use redundancy to enable recovery to saved good state of system
  - E.g., click “Undo” to recover to old copy of file

Forward Error Recovery

- **Uses sufficient physical or information redundancy to not only detect errors but also correct them**
  - FER is inseparable from error detection
  - Error detection is on critical path
- **Canonical examples**
  - Triple modular redundancy (TMR)
    - majority voter
- **Error correcting codes (ECC)**
  - Enough extra bits to detect and correct errors
Backward Error Recovery

- BER requires independent error detection mechanism
  - Can't recover from error you don't detect!
  - BER takes error detection off critical path (unlike FER)
- Canonical examples
  - Periodic checkpoint/recovery (+ error detection)
  - Logging of changes to system state (+ error detection)
- BER may not be suitable for real-time systems
  - Why not?

We'll spend most of this class period on BER

Very Rough Comparison: FER vs. BER

<table>
<thead>
<tr>
<th>Feature</th>
<th>FER</th>
<th>BER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error detection on critical path</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Fault-free performance</td>
<td>Some degradation</td>
<td>Little degradation</td>
</tr>
<tr>
<td>Performance if faults</td>
<td>No slowdown</td>
<td>Slow recovery</td>
</tr>
<tr>
<td>Hardware cost</td>
<td>Higher</td>
<td>Lower</td>
</tr>
<tr>
<td>Design complexity</td>
<td>Lower</td>
<td>Higher</td>
</tr>
</tbody>
</table>
Performance of FER vs. BER

Warning: do not take this graph too seriously. The relative heights of the curves and their shapes are gross estimates that do not correspond to any particular system.

Outline of Today’s Course

• Forward error recovery (FER) vs. backward error recovery (BER)
• BER in general
• Uniprocessor BER
• Multiprocessor BER
Backward Error Recovery (BER)

- If error detected, recover backwards & re-execute
  - Recover to previous state of system that we know is error-free
  - Assumes that error will be gone before resuming execution

- Some terminology:
  - Checkpointing: periodically saving state of system
  - Logging: saving changes made to system state
  - Recovery point: the point to which we recover in case of error
  - Validated checkpoint: error detection mechanism has found no errors in execution up to this checkpoint

- BER can be implemented in hardware or software
  - We'll focus on hardware implementations in this class

BER Abstraction

- Diagram showing the core and active state of the system, with checkpoints awaiting validation and the most recently validated checkpoint.
Key Idea: BER Takes Error Detection Off Critical Path

- Validate CP only when all cores agree it is error-free
  - Example: no outstanding coherence requests in checkpoint
- Nodes perform error detection, then coordinate
- Error detection can be off critical path and pipelined
  - Reason why we have checkpoints awaiting validation
- Can hide long fault detection latencies
  - (Number of outstanding checkpoints) x (checkpoint length)
  - Design tolerance to be longer than longest detection latency

Don’t slow down execution to validate checkpoints
I/O and the Outside World

- **Output commit problem** – Can’t send uncommitted data beyond sphere of recoverability
  - E.g., can’t tell printer to write check for $1M before we know that’s the right amount

- **Standard solution: wait to communicate with I/O**
  - Only send validated data to outside world
  - Problem: if it takes a long time for core1 to know that its most recent checkpoint is part of validated recovery line, then output will be delayed a long time

- **Input commit problem** – Input can’t be recovered
  - Solution: augment checkpointing with input logging

---

6 BER Issues

1) What state needs to be saved?
2) How do we save this state?
3) Where do we save it?
4) How often do we save it?
5) How do we recover the system to this state?
6) How do we resume execution after recovery?
(1) What State To Save

- Need to save all state that would be necessary if this were to become the recovery point
- In general, we only need to save the user-visible state
- For example, microprocessors:
  - Must save architectural state
  - Don't have to worry about micro-architectural state
  - We'll delve into this in more detail in a few slides

(2) How to Save State

- Two “flavors” of BER:
  - Checkpointing: Periodically stop system and save state
  - Logging: Log all changes to state
- Checkpointing
  - Only incurs performance overhead at periodic checkpoints
  - Can only recover at coarse granularity
  - Size of checkpoint is often fixed
- Logging
  - Finer granularity of rollback
  - Incurs overhead for logging many common operations
  - Amount of state logged is variable (but may have upper bound)
- Hybrid approaches are also used
  - Why might these be useful?
(3) Where to Save State

• Have to save state where it is “safe”
  – A fault in the recovery point state could make recovery impossible

• In core (can’t survive loss of processor chip)
  – Processor saves registers to shadow registers

• In cache (same as core, if on-chip cache)
  – Processor copies registers into cache

• In memory (memory can be made pretty safe)
  – Processor cores copy registers into memory
  – Write-through cache copies data into memory

• In disk (arguably the safest, but slow)
  – E.g., databases log updates to disks

• In tape (too slow except for rare backups)

(4) When to Save State

• Checkpointing
  – Can choose checkpoint interval

• Logging
  – Continuously saving state (every time it changes)

• For checkpointing, a larger checkpoint interval means
  – Less overhead due to checkpointing (since less frequent)
  – Coarser checkpoint granularity (can’t recover to arbitrary point)
(5) How to Recover State

- **Checkpointing**: Copy pre-fault recovery point checkpoint into architectural state
- **Logging**: Unroll log to undo changes since recovery point

- Tradeoff between these two depends on system

(6) How to Resume Execution

- Simply resuming execution after recovery may not be feasible
  - E.g., recovery due to hard fault in interconnection switch
- May need to reconfigure before resuming, to ensure forward progress
  - E.g., reconfiguring the routing in interconnect to avoid dead switch
- What if you can’t resume? Does BER still provide any benefits (in any metric)?
Outline of Today’s Course

- Forward error recovery (FER) vs. backward error recovery (BER)
- BER in general
  - Uniprocessor BER
  - Multiprocessor BER

Uniprocessor BER: What State To Save

- Assume disks are safe storage (common assumption)
- Checkpoint state = architectural state
  - Architectural registers (including program counter, etc.)
  - NOT micro-architectural state (e.g., branch predictor state)
    » Why not?
  - Memory (and caches)
**Uniprocessor BER: How/Where To Save State**

- **Architectural registers**
  - Copy them to shadow registers within processor
  - Or map them to memory and thus save them in cache (or memory)

- **Modified blocks in cache**
  - Use write-through cache to copy cache state to memory
  - Or periodically flush all modified blocks to memory
  - Why don’t we save unmodified blocks in cache?

- **Dirty pages in memory**
  - Periodically flush all dirty pages to disk
  - Why don’t we save clean pages?

**Uniprocessor BER: When To Save State**

- **If we save after every instruction**
  - Almost has to be done with logging (rather than checkpointing)
  - Enables finest granularity of recovery (but is this overkill?)
  - In OOO processor, must save precise state of system
    - Potentially high overhead
      - Logging takes time (but is it on critical path?)
      - Extra power consumption

- **If we save after every N instructions (N >> 1)**
  - Coarser granularity → recovery is likely to have to go back farther in time and potentially undo more error-free work
  - But if errors are rare, this penalty won’t matter much
  - Overhead might be reduced by checkpointing (instead of logging)
Uniprocessor BER: How To Recover State

• Not possible if fault is in recovery point state!

• Architectural registers
  – Copy them back from shadow registers
  – Load them back from where they were mapped in memory

• Cache
  – Don’t have to do anything – state is already saved on memory/disk
  – Will get re-loaded with state after resuming execution

• Memory
  – Don’t have to do anything – state is already saved on disk
  – Will get re-loaded with state after resuming execution

Uniprocessor BER: How To Resume Execution

• May not be possible if fault can’t be tolerated (even if we were able to recover from it)
  – E.g., hard fault in instruction fetch unit
  – Other examples?

• For transient errors, nothing needs to be done before resuming execution
Outline of Today’s Course

• Forward error recovery (FER) vs. backward error recovery (BER)
• BER in general
• Uniprocessor BER
  • Multiprocessor BER
    – Message passing systems
    – Shared memory systems

Multiprocessor (MP) BER

• Two types of MPs:
  – Shared memory MPs
  – Message passing MPs
• Message passing
  – Processors communicate via explicit messages
• Shared memory
  – Processors communicate via global shared memory
  – Loads and stores to shared memory used to transfer data
• Goal: create consistent checkpoints (via checkpointing or logging)
  – Consistent checkpoint = set of per-processor checkpoints that, in aggregate, constitutes a consistent system state
  – Recovery line = set of recovery points = consistent checkpoint
Msg Passing MP BER: What State to Save

• Uniprocessor state at each processor in MP
• Messages received
  – From other processes
  – From outside world (e.g., Internet)
  – State depends on whether communication is reliable or lossy
• Key: in a consistent checkpoint, it shouldn’t be possible for process P1 to have received message M from process P2 if P2’s checkpoint doesn’t yet include having sent M

Msg Passing MP BER: How/Where/When

• We’re going to start with checkpointing, since “logging” schemes are actually checkpointing schemes that have been augmented with logging (to achieve certain goals that we’ll get to later)
Msg Passing MP BER: Checkpointing

- Checkpointing schemes differ based on “When”
  - Uncoordinated
  - Coordinated

Uncoordinated Checkpointing

- Each process independently takes checkpoint
  - Doesn’t coordinate with other processes

- Pros
  - Easier to implement
  - No performance penalty for coordination

- Cons
  - May be tougher to recover
  - Tough/impossible to create consistent recovery line
    » Might end up with some inconsistent checkpoints
    » Could lead to cascading rollbacks (aka “domino effect”)
Determining the Recovery Line

- If checkpoints are taken independently, how do we figure out where the recovery line is (i.e., which set of checkpoints represents a consistent state)?
- Construct a **checkpoint dependency graph**
- A checkpoint $C_2$ depends on another checkpoint $C_1$ if $C_1$ includes the sending of a message whose reception is included in $C_2$
  - Thus, if $P_2$ recovers to pre-$C_2$ checkpoint, then $P_1$ must recover to pre-$C_1$ checkpoint

Constructing the Recovery Line

- Using the dependency graph, we can construct the recovery line either:
  - On-demand (when an error occurs)
  - Continuously
- Any checkpoints that are older than the recovery line can be discarded (since we would never want to recover to them)
  - This process is sometimes known as **garbage collection**
The Domino Effect

• If the most recent checkpoint is inconsistent (i.e., includes a message reception but not its sending), then system must recover to earlier checkpoint
  – But what if that one is also inconsistent?
  – And then the one before that one?
• In worst case, we would have to undo all work and recover to the beginning of execution

---

Coordinated Checkpointing

• To avoid cascading rollbacks, the processes can coordinate when they take their individual checkpoints
• Pros (besides no domino effect!)
  – Easier/faster recovery
  – No need to construct dependency graph
  – Can be more aggressive in garbage collection
• Cons
  – More complex to implement
  – Coordination incurs a performance penalty
Blocking 4-phase Coordination

- Algorithm for creating consistent checkpoint
  - Key idea: nothing in flight if we let all in-flight messages drain
  1) Centralized coordinator broadcasts TakeCheckpoint request to all processes to take checkpoint
  2) Each process then takes a checkpoint and sends acknowledgment to coordinator that it has completed
  3) Centralized controller waits for all acks and then broadcasts CheckpointDone message
  4) Each process resumes execution

More Optimized Coordination

- 4-phase algorithm is slow because it is blocking
- Some non-blocking algorithms are faster, but more complex
- Another alternative is to use synchronized clocks to facilitate coordination
  - Each process takes checkpoint every N clock cycles
  - If clocks are perfectly synchronized, this works, but that's tough to do
  - Better yet, as long as clock skew is less than the minimum communication latency between any two processes, then this works (because a message can't go backwards in time)
Logical Time Coordination

- **Logical time clocks have been used to coordinate checkpoints**
  - Each node has its own logical clock
  - Each node takes “independent” checkpoint every N logical cycles

- **Logical time is a time basis that respects causality**
  - If event A causes event B, then A must happen earlier in logical time than B
  - E.g., sending of a message happens earlier than reception

- **Many logical time bases/algorithms exist**
  - Loosely synchronized physical clocks (skew < min latency)
  - Token-passing among processes to advance logical time

- **Advantage of logical time coordination is that it is implicit and non-blocking**
  - Don’t have to stop to coordinate --- just look at local logical clock

Message Passing BER: Logging

- **Goals of logging**
  - Speed up output commit by removing dependencies between checkpoints
  - Solve the input commit problem

- **Three types of logging schemes**
  - Pessimistic
  - Optimistic
  - Causal (won’t cover this in class)
**Pessimistic Logging**

- Log every message reception before processing it (and integrating its effects into execution)
- If P1 detects error, P1 recovers to its most recent checkpoint and replays messages log (only those messages that arrived after checkpoint taken)
  - **KEY:** No need to recover any other process!
- Since there are no longer any dependencies between checkpoints on different processes, **output commit doesn’t require waiting to establish consistent recovery line**
- **Disadvantages:**
  - Logging is on critical path (degrades performance)
  - Logs may take up lots of storage space

---

**Optimistic Logging**

- **Take message logging off the critical path**
  - Let received messages affect the execution while they are being logged in parallel
  - Assumes that it is very rare for an error to occur between when message arrives and when it has been logged
    - “Window of vulnerability”
  - Tradeoff: better performance vs. not as reliable
Outline of Today’s Course

- Forward error recovery (FER) vs. backward error recovery (BER)
- BER in general
- Uniprocessor BER
  - Multiprocessor BER
    - Message passing systems
    - Shared memory systems

Shared Memory MP BER

- Must save uniprocessor state for all processors in MP
- Must also save state that corresponds to communication between the processors
  - Cache and memory state
  - Includes cache coherence state
- Same challenge as for message passing: how do we create consistent checkpoint despite in-flight messages (coherence transaction)?
  - We now consider only coordinated checkpointing
Shared Memory MP BER: Blocking CP

- Same blocking four-phase algorithm we just saw for message passing systems
- Examples
  - Multiprocessor CARER [FTCS '90, Ahmed, Frazier, and Marinos]
  - ReVive [ISCA '02, Prvulovic et al.]
- Disadvantage of blocking? Performance

Shared Memory MP BER: Nonblocking CP

- As with message passing, can also use logical time to coordinate checkpoints in shared memory system
- Example: SafetyNet [ISCA '02, Sorin et al.]
  - Used different logical time base depending on coherence protocol
Where We Are in the Course

- **Thursday: Error recovery**
- **Friday: Diagnosis, reconfiguration & the future**

**Detect Errors**
- Determine that something went wrong

**Thursday**
- **Recover**
  - Resume execution from a safe point

**Diagnose Faults**
- Figure out the cause of the problem

**Self-Repair**
- Keep the problem from repeating

---

Recommended Reading

Fault Tolerant Computer Architecture

Fault Diagnosis and Self-Repair
&
Course Wrap-Up

Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

ACACES 2010

Where We Are in the Course

• Friday: Diagnosis, self-repair & the future

Tuesday
Detect Errors
Determine that something went wrong

Thursday
Recover
Resume execution from a safe point

Friday
Diagnose Faults
Figure out the cause of the problem
Self-Repair
Keep the problem from repeating
Diagnosis & Self-Repair

- Error detection provides safety
- Error recovery MAY provide liveness
- But what if error is due to permanent fault?
  - Want to diagnose it and repair system to stop using fault part

Outline of Today’s Course

- Diagnosis
  - General concepts
  - Microprocessor cores
  - Multicore processors
- Self-repair
- Wrap-up and future of fault tolerant architecture
Why Diagnosis Matters

- **For systems with BER**
  - Problem: no guarantee of forward progress
    - If hard fault, then after recovery, might just exercise fault again
    - Want to diagnose faulty hardware and stop using it
- **For systems with FER**
  - Problem: hard fault can lead to violation of fault model
    - Example: TMR with a single hard fault
      - TMR fault model is single fault
      - Yet once a hard fault occurs any subsequent fault violates fault model and can cause failure
    - Want to diagnose faulty hardware and reconfigure to avoid possible failures

Diagnosis Granularity

- At what granularity should we diagnose faults?
- Finer is better, right? But ...
  - Finer granularity requires more effort (hardware, power)
  - Might be more information than is useful
  - Do I care if transistor 289,631,112 is faulty?
- Key: match diagnosis granularity to self-repair granularity
- Example: if self-repair granularity is entire core, then no need to diagnose where within a core the fault is
System Model Implications for Diagnosis

- System model determines how easy diagnosis is
- “Easy” system models
  - Systems with FER
  - Systems with BER and localized error detection schemes
- “Hard” system models
  - Systems with BER and end-to-end error detection schemes
  - All you know is that an error occurred somewhere

Focus of rest of this section is on hard system models

Well-Know Diagnosis Option: BIST

- BIST: built-in self-test

Circuit under test

BIST Unit

Generates specific set of test inputs

Compares test outputs to golden outputs of fault-free system

So BIST detects faults, but how does it diagnose?
Diagnosis With BIST

- Key: analyze test outputs to deduce fault location
- Example: storage array

Row1 = pass
Row2 = fail
Row3 = pass
Row4 = pass

Two Approaches to Diagnosis

1. Periodic BIST
   - Periodically test all or part of the system with predetermined test vectors
   - Advantage: can target specific faults and potentially get better coverage
   - Disadvantage: time spent doing BIST incurs performance and energy overhead

2. Using live execution
   - As system executes normally, diagnose faults
   - Advantage: little overhead
   - Disadvantage: may not uncover fault for a long time
Periodic BIST

• Several design decisions
  – How often to test?
  – How comprehensive is the test?
  – What fault models to test for?
• To reduce impact on performance, can try to test units when they're otherwise idle
  – E.g., test the FPU when no instruction is using it
• To reduce the hardware cost and improve flexibility, can use software
  – Add instructions to ISA that provide access to the BIST scan chain and enable the construction of test programs

Core Diagnosis During Live Execution

• Bower et al. [MICRO 2005]
• Assume system has scheme to detect errors at instruction granularity (e.g., RMT or DIVA)
• Track instruction resource usage while in-flight
• Maintain saturating error counters for core resources
  – Increment counter for each structure associated with a faulted instruction
  – Counter saturation indicates hard fault in associated structure
Core Diagnosis During Live Execution, cont’d

• Li et al. [DSN 2008]
• Assume error detection at very high level
  – E.g., detect error via detecting system crash
  – No diagnostic information
• Assume BER scheme
• If error detected on Core 1, recover to validated checkpoint on Core 1 and on Core 2
• Can compare the execution traces on Core 1 and Core 2 to infer where fault on Core 1 is

Cache/Memory Diagnosis

• Almost exclusively BIST-based diagnosis
• Could use ECC, but generally a mismatch of granularity
  – ECC granularity is much finer than self-repair granularity
Outline of Today’s Course

• Diagnosis
• Self-repair
• Wrap-up and future of fault tolerant architecture

Why Self-Repair Matters

• For exactly same reasons as diagnosis!
• In fact, self-repair is inseparable from diagnosis
• Can't repair if we don't know what to repair
Self-Repair Granularity

- Once again, self-repair tied to diagnosis
- (Same) Key: match diagnosis granularity to self-repair granularity
- (Same) Example: if self-repair granularity is entire core, then no need to diagnose where within a core the fault is

Self-Repair

- **Self-repair = physical redundancy + reconfiguration**
- Physical redundancy (spare parts) can be:
  - Hot or cold – tradeoffs?
  - Homogeneous or heterogeneous – tradeoffs?
Self-Repair for Superscalar Cores

- Superscalar cores have lots of redundancy
  - Exists for performance but can be used for fault tolerance
- Superscalar core has more than bare minimum number of many units:
  - ALUs
  - Physical registers
  - Reorder buffer (ROB) entries
  - Etc.
- Reconfiguration
  - Mark a faulty ALU as busy (and never mark it free)
  - Mark a faulty register as busy (and never mark it free)
  - Etc.

Self-Repair for Simple Cores

- Simple cores have little or no hardware redundancy
- Must use software to emulate functionality of faulty hardware
- Detouring [Meixner, DSN 2008]
  - Translate software so that it doesn’t touch faulty hardware
  - Can “Detour” around many faulty units
    » Multiplier, shifter, register, l-cache frame, etc.
Detouring Big Picture

Fault-Aware Compilation Tool-chain

- Linker: Map around I-Cache errors
- Register Allocation: Do not allocate faulty registers
- Instruction Selection: Emulate faulty functional units

Core-specific Binary

Faulty Core

List of Faults

Program

Self-Repair for Caches/Memory

- Storage has LOTS of redundancy
- Intra-chip granularity
  - Easy to deconfigure row or column of array
- Inter-chip granularity (for DRAM)
  - Chipkill memory designs allow for loss of entire chip
  - Key: spread each dataword and its ECC bits across many chips
  - The loss of any single chip can’t corrupt enough bits in any given dataword to thwart the ECC
Self-Repair for Multicores

• Multicore processors offer new opportunities
• Obvious solution: core shutdown
  – If fault in core, stop using that core
  – Pros: simple and we’ve got plenty of cores to spare (or will soon)
  – Cons: not efficient to discard entire core because it has one fault
• These drawbacks have led to a variety of more efficient solutions

Self-Repair for Multicores: Offload

• If unit X is faulty on Core C, then offload that functionality to another core
• Example:
  – Core 1 has a faulty multiplier (and Core 2 does not)
  – When Core 1 encounters a multiply, it sends it to Core 2 to be executed there
• Can augment this approach with scheduling
  – Schedule the thread that performs lots of multiplications on Core 2
Multicore Self-Repair: Core Cannibalization

- Pipe stage granularity [Romanescu, PACT 2008]

Chip relying on Core Shutdown

Core 1 Core 2 Core 3

Functional cores 0
Most logic fault-free, chip useless

Chip with Core Cannibalization

Core 1 Core 2 Core 3

Functional cores 2
Same faults, chip functional

Multicore Self-Repair: StageNet

- More flexible approach [Gupta et al., MICRO 2008]
Recommended Reading - Diagnosis

- “Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults,” Li et al. DSN 2008.

Recommended Reading – Self-Repair

Outline of Today’s Course

• Diagnosis
• Self-repair
• Wrap-up and future of fault tolerant architecture

Summary

• In this course, we have learned:
  – Why fault tolerance matters
  – Why faults occur
  – How to detect errors
  – How to recover from errors
  – How to diagnose hard faults
  – How to perform self-repair
Some Interesting Open Problems

• Error detection
  – Handling multiple-error scenarios
  – Errors in other processor models (GPU, NPU, etc.)
  – Errors in floating point units

• Error recovery
  – BER: Mitigating impact of output commit problem

• Diagnosis & self-repair
  – Handling faults in memory subsystem (not incl. storage itself)
  – Finding appropriate granularity
  – Dealing with much more numerous faults in future CMOS or nano

Relationship Between FT and Other Issues

• Security
  – Can we treat security breaches like faults?
  – What’s the fault model?

• Tolerating software bugs
  – Can we use hardware to help?

• Reducing vulnerability to faults
  – Can we design hardware such that faults are less likely to cause failures?
  – Would make FT easier

• Power/temperature management
  – Can we manage power, temp, and FT in integrated fashion?

• And many more …
But Does Anyone Use This Stuff?

- Industry hasn’t YET adopted much fault tolerance for commodity (non-mainframe) processors
  - EDC/ECC on storage and buses
  - Redundant lockstepped cores (not commodity)
- Will industry change its mind? If so, when?
  - IBM has had fault tolerance for years
  - Intel has recently done research in fault tolerance
- Is fault tolerance analogous to power/temperature?
  - That is, ignored for a long time and then suddenly critical

THANK YOU!