

## Outline

- Basic Concepts
- Physical Redundancy
- Error Detecting/Correcting Codes
- Re-Execution Techniques
- **Backward Error Recovery Techniques**

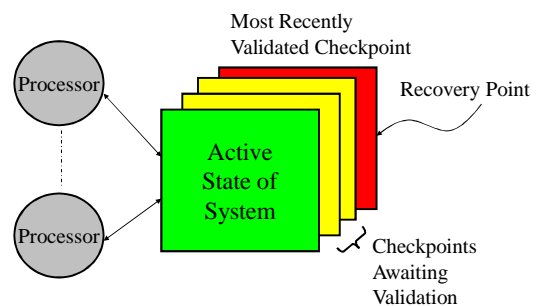
## Backward Error Recovery (BER)

- **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
- **Many commercial machines use(d) BER**
  - Sequoia, Synapse N+1, Tandem/HP NonStop
- **BER also includes all-software schemes**
  - Nightly backups of file systems, database software, etc.

## BER Performance

- **May sacrifice performance to achieve availability**
  - Where might we lose performance?
- **May not be suitable for real-time systems**
  - What are the alternatives?

## BER Abstraction



## 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 processor (can't survive loss of processor chip)**
  - Processor saves registers to shadow registers
- **In cache (same as processor, if on-chip cache)**
  - Processor copies registers into cache
- **In memory (memory can be made pretty safe)**
  - Processor copies 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)?**

#### 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?

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

73

## 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 ( $N \gg 1$ )**
  - Coarser granularity  $\rightarrow$  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)

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

74

## 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

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

75

## 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**

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

76

## Multiprocessor (MP) BER

- In this class, we consider two types of MPs:
  - Shared memory MPs (beyond scope of ECE 254 – take ECE 259)
  - **Message passing MPs (focus on this in ECE 254)**
- Shared memory
  - Processors communicate via global shared memory
  - Loads and stores to shared memory used to transfer data
- Message passing
  - Processors communicate via explicit messages
- 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

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

77

## 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 (see ECE 259 / CPS 221)
- The details of how this is done are beyond the scope of this course
- Those of you interested should refer to:
  - SafetyNet [ISCA '02, Sorin et al.]
  - Multiprocessor CARER [FTCS '90, Ahmed, Frazier, and Marinos]
  - ReVive [ISCA '02, Prvulovic et al.]

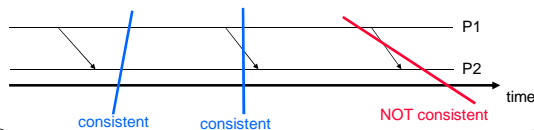
(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

78

## 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



(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

79

## Msg Passing MP BER: How/Where/When

- Refer to the Elnozahy survey paper for more detail on all of this
- 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)

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

80

## Msg Passing MP BER: Checkpointing

- Checkpointing schemes differ based on “When”
  - Uncoordinated
  - Coordinated
  - Communication-induced (won't cover in class)

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

81

## 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”)

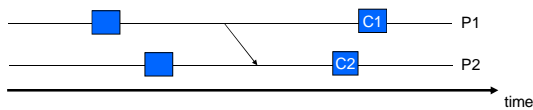
(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

82

## 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 C2 depends on another checkpoint C1 if C1 includes the sending of a message whose reception is included in C2
  - Thus, if P2 recovers to pre-C2 checkpoint, then P1 must recover to pre-C1 checkpoint



(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

83

## 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**

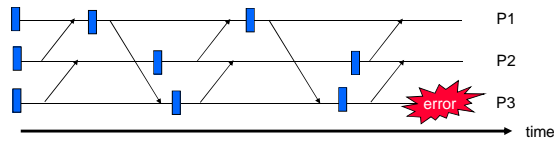
(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

84

## 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



(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

85

## 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

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

86

## Blocking 4-phase Coordination

- Algorithm for creating consistent checkpoint
  - 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

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

87

## 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)

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

88

## 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

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

89

## 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 P1 to know that its most recent checkpoint is part of a validated recovery line, then output will be delayed a long time ... but we can avoid this by using logging!
- **Input commit problem** – Input can't be recovered
- **Solution: augment checkpointing with input logging**

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

90

## 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)

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

91

## 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

(C) 2011 Daniel J. Sorin

ECE 254 / CPS 225

92

## 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

- **Basic Concepts**
- **Physical Redundancy**
- **Error Detecting/Correcting Codes**
- **Re-Execution Techniques**
- **Backward Error Recovery Techniques**