Reliable Computing I

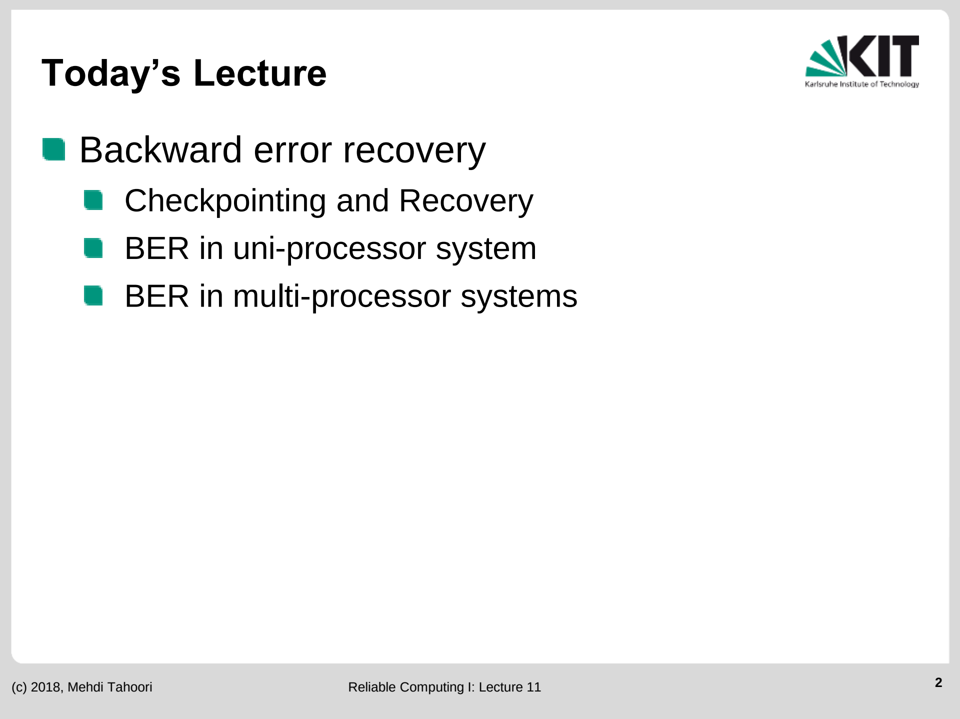
Lecture 11: Checkpointing and Recovery

Instructor: Mehdi Tahoori

INSTITUTE OF COMPUTER ENGINEERING (ITEC) – CHAIR FOR DEPENDABLE NANO COMPUTING (CDNC)

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

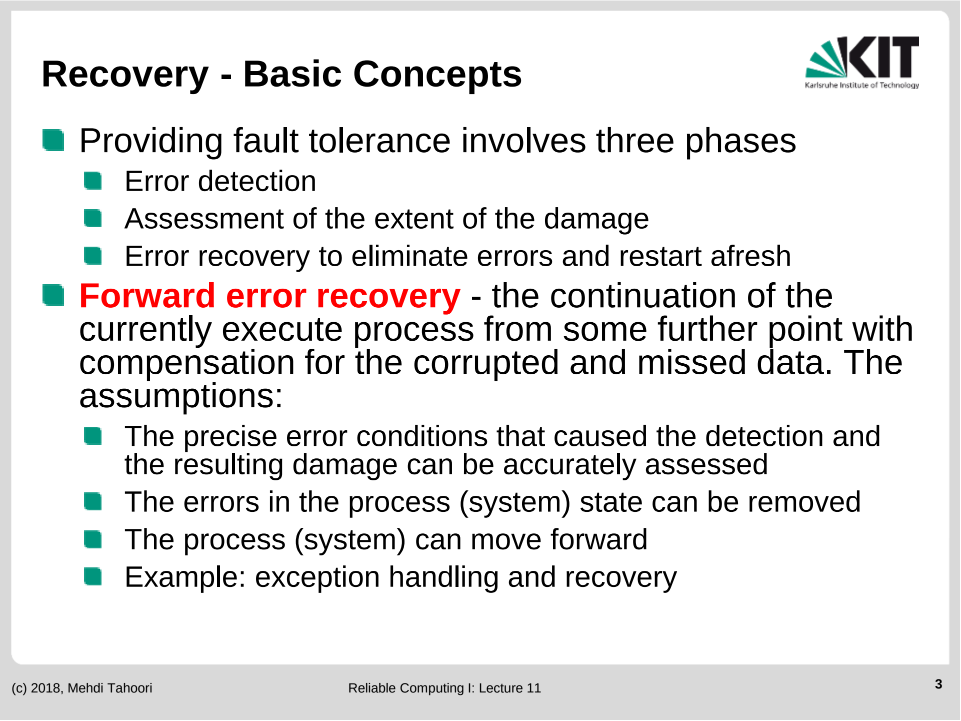
www.kit.edu



Today’s Lecture

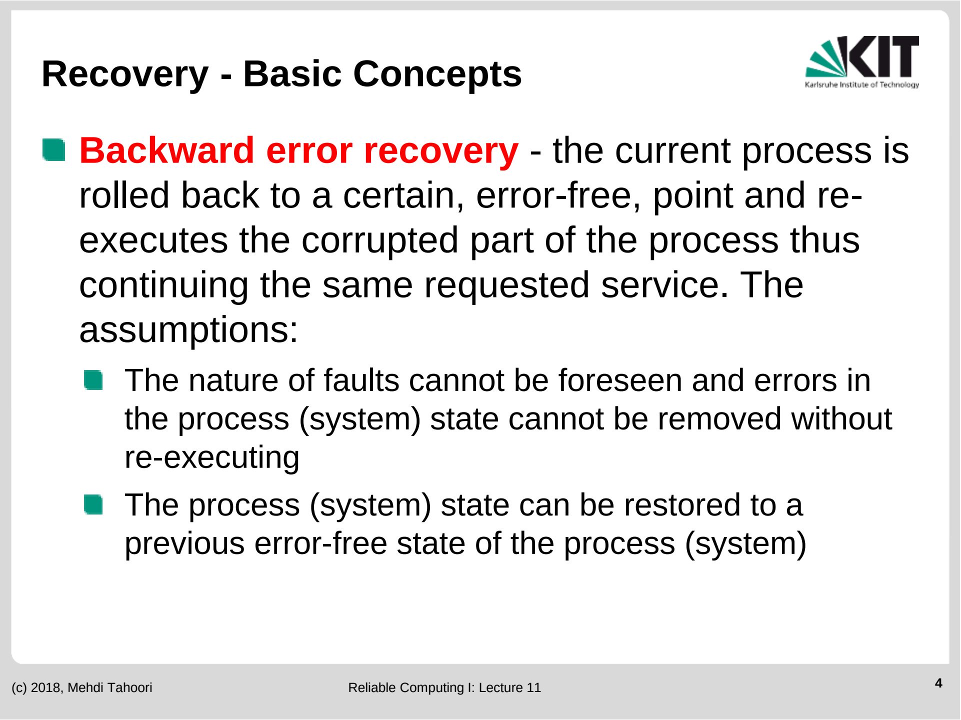
- Backward error recovery
  - Checkpointing and Recovery
  - BER in uni-processor system
  - BER in multi-processor systems

(c) 2018, Mehdi Tahoori   Reliable Computing I: Lecture 11   2



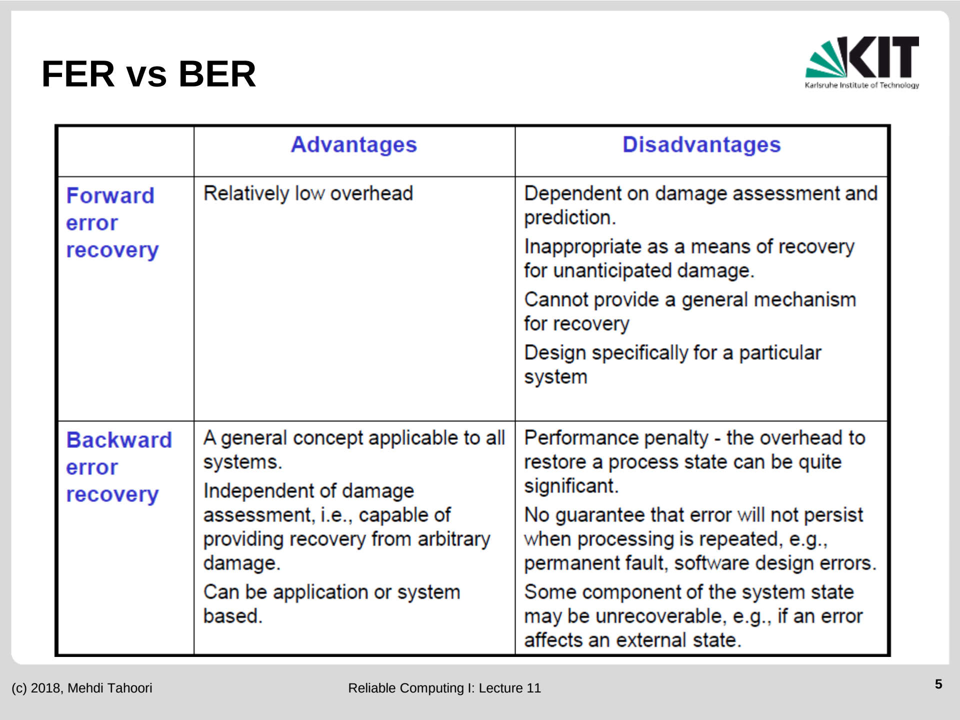
Recovery - Basic Concepts

- Providing fault tolerance involves three phases
  - Error detection
  - Assessment of the extent of the damage
  - Error recovery to eliminate errors and restart afresh
- **Forward error recovery** - the continuation of the currently execute process from some further point with compensation for the corrupted and missed data. The assumptions:
  - The precise error conditions that caused the detection and the resulting damage can be accurately assessed
  - The errors in the process (system) state can be removed
  - The process (system) can move forward
  - Example: exception handling and recovery



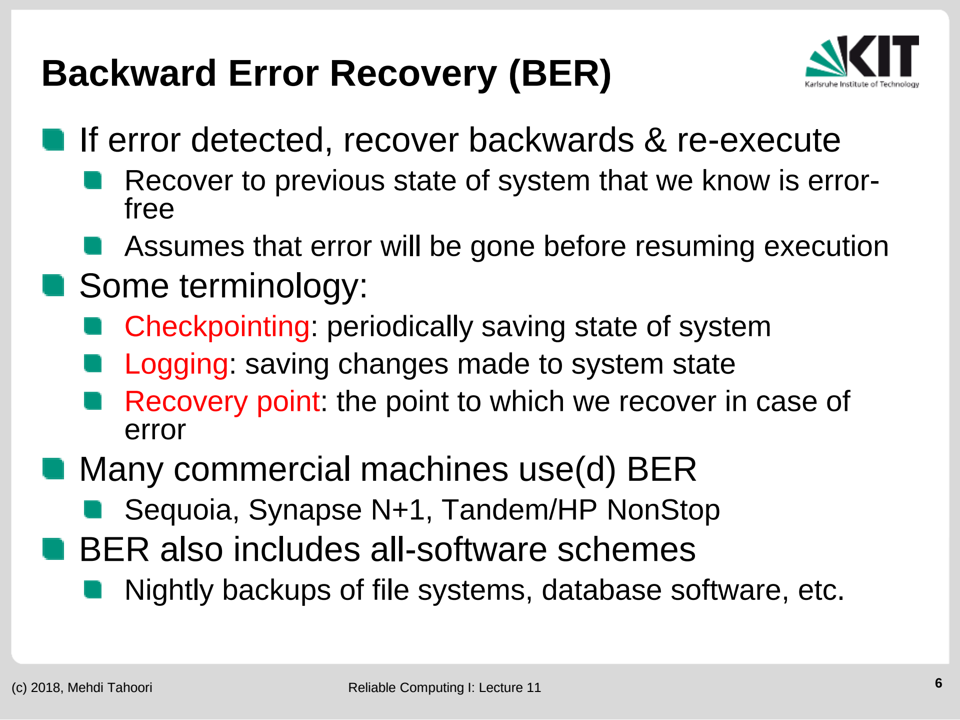
Recovery - Basic Concepts

- **Backward error recovery** - the current process is rolled back to a certain, error-free, point and re-executes the corrupted part of the process thus continuing the same requested service. The assumptions:
  - The nature of faults cannot be foreseen and errors in the process (system) state cannot be removed without re-executing
  - The process (system) state can be restored to a previous error-free state of the process (system)



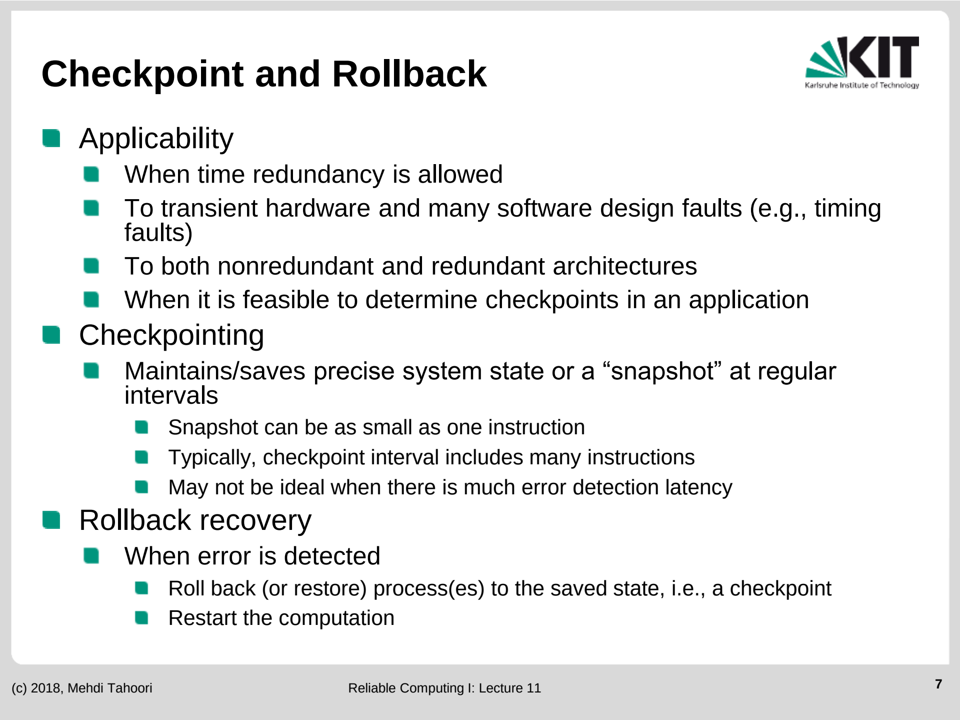
FER vs BER

	Advantages	Disadvantages
Forward error recovery	Relatively low overhead	Dependent on damage assessment and prediction. Inappropriate as a means of recovery for unanticipated damage. Cannot provide a general mechanism for recovery Design specifically for a particular system
Backward error recovery	A general concept applicable to all systems. Independent of damage assessment, i.e., capable of providing recovery from arbitrary damage. Can be application or system based.	Performance penalty - the overhead to restore a process state can be quite significant. No guarantee that error will not persist when processing is repeated, e.g., permanent fault, software design errors. Some component of the system state may be unrecoverable, e.g., if an error affects an external state.



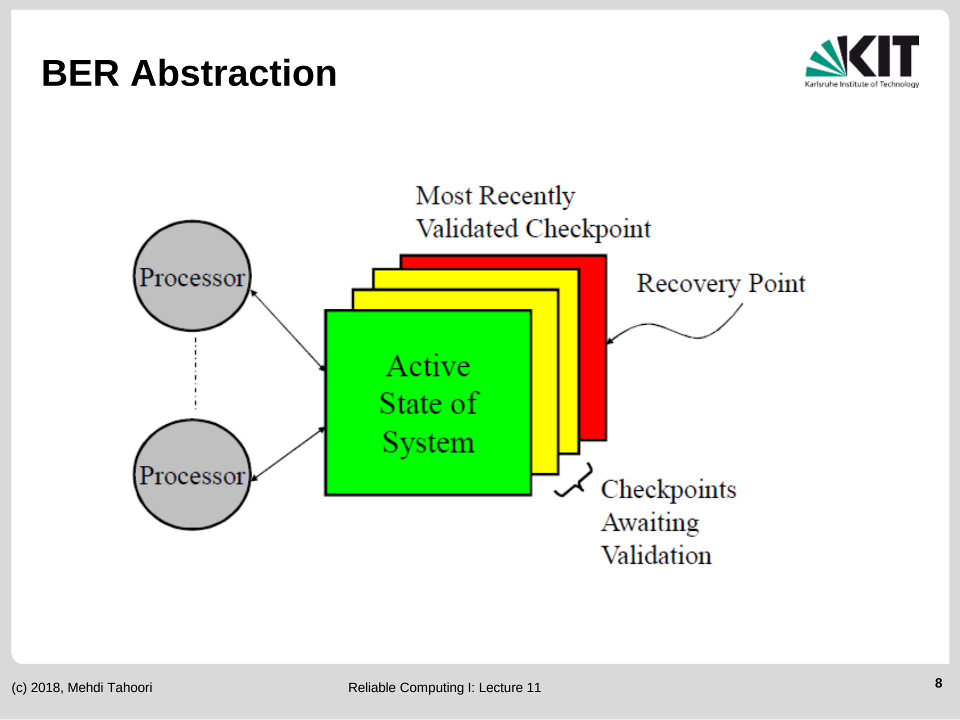
Backward Error Recovery (BER)

- 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.

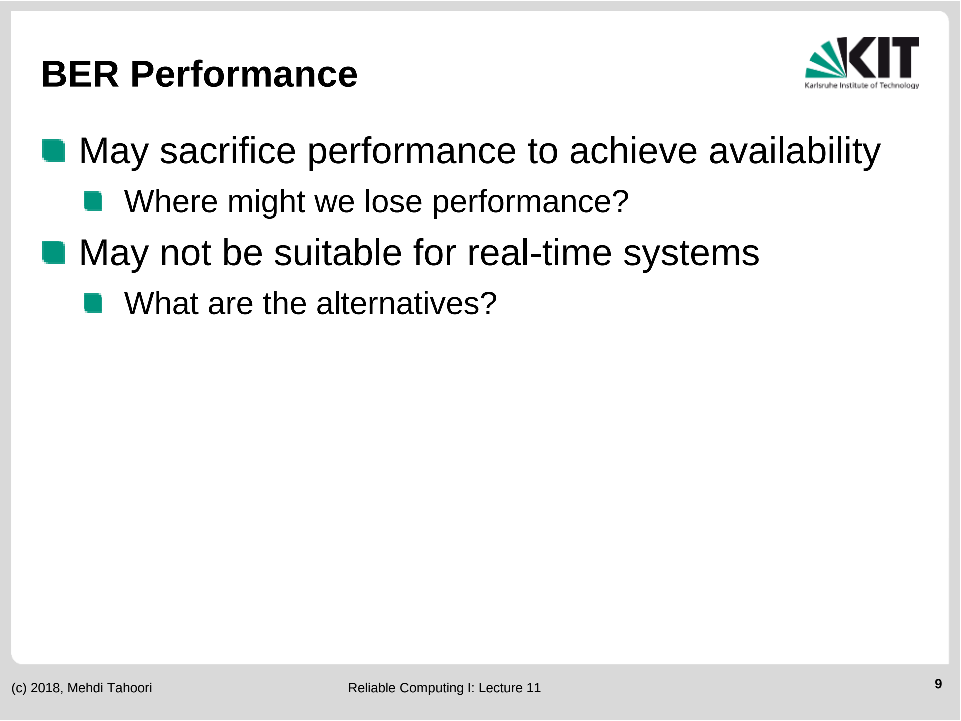


Checkpoint and Rollback

- Applicability
  - When time redundancy is allowed
  - To transient hardware and many software design faults (e.g., timing faults)
  - To both nonredundant and redundant architectures
  - When it is feasible to determine checkpoints in an application
- Checkpointing
  - Maintains/saves precise system state or a “snapshot” at regular intervals
    - Snapshot can be as small as one instruction
    - Typically, checkpoint interval includes many instructions
    - May not be ideal when there is much error detection latency
- Rollback recovery
  - When error is detected
    - Roll back (or restore) process(es) to the saved state, i.e., a checkpoint
    - Restart the computation

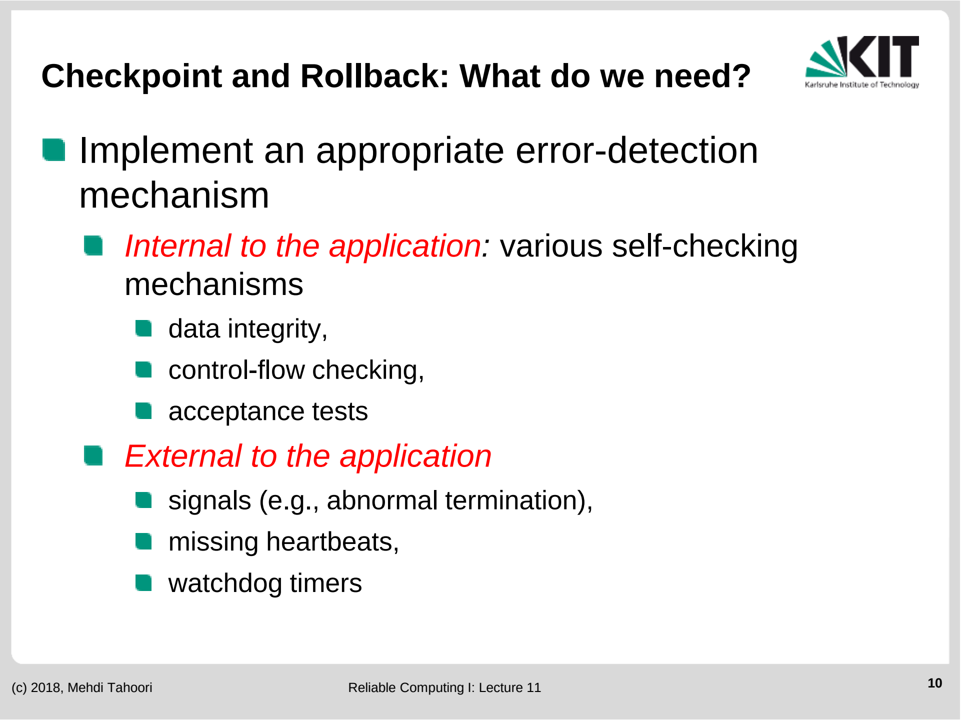


BER Abstraction



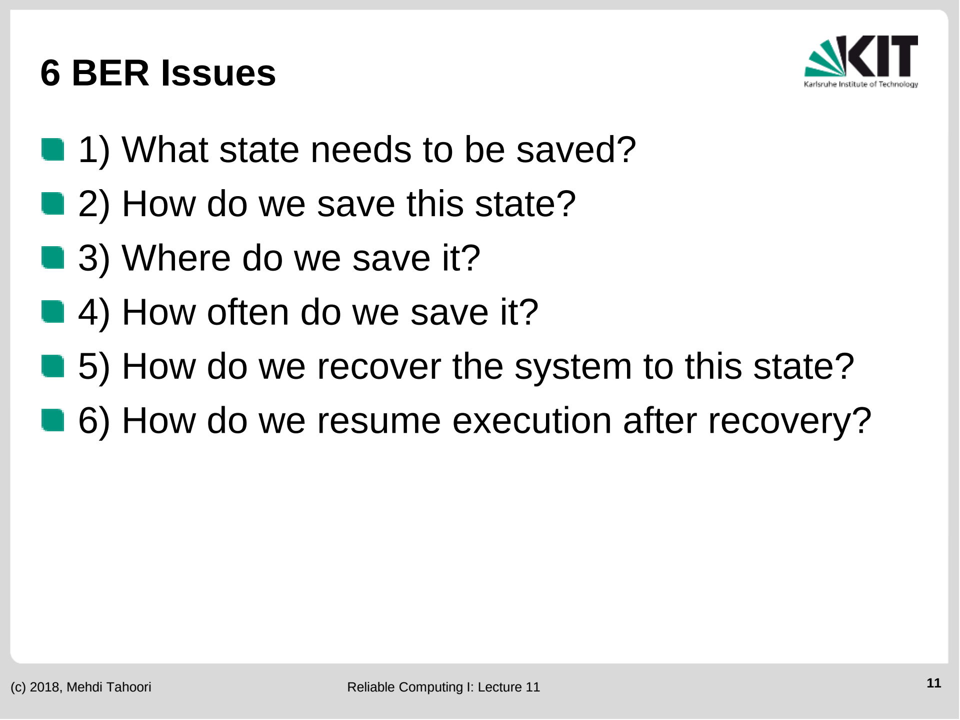
BER Performance

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



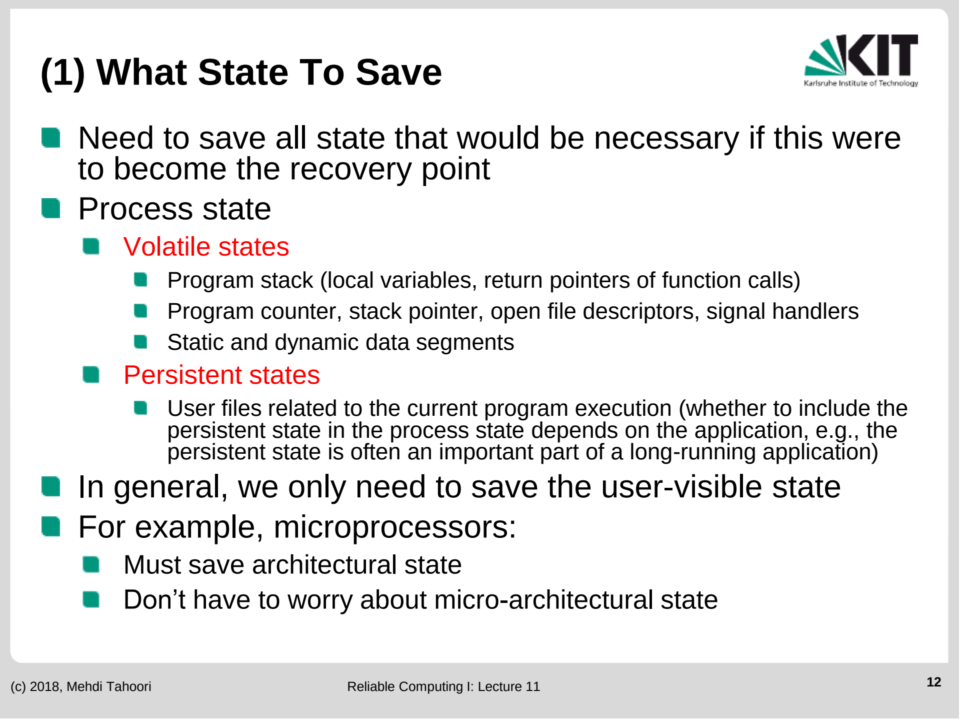
Checkpoint and Rollback: What do we need?

- Implement an appropriate error-detection mechanism
  - *Internal to the application:* various self-checking mechanisms
    - data integrity,
    - control-flow checking,
    - acceptance tests
  - *External to the application*
    - signals (e.g., abnormal termination),
    - missing heartbeats,
    - watchdog timers



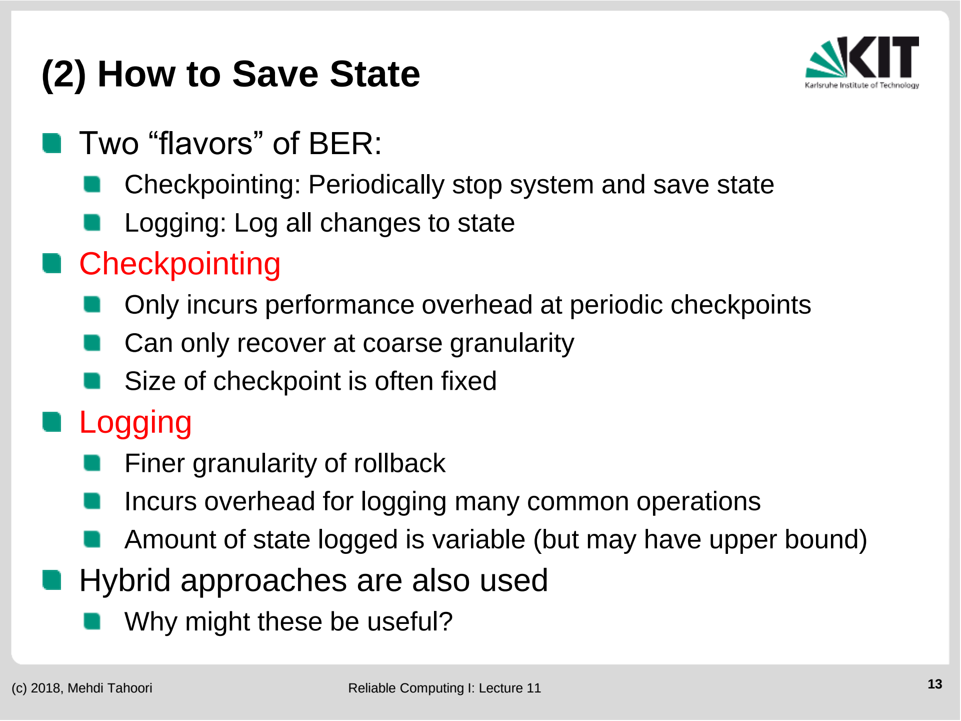
6 BER Issues

- 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

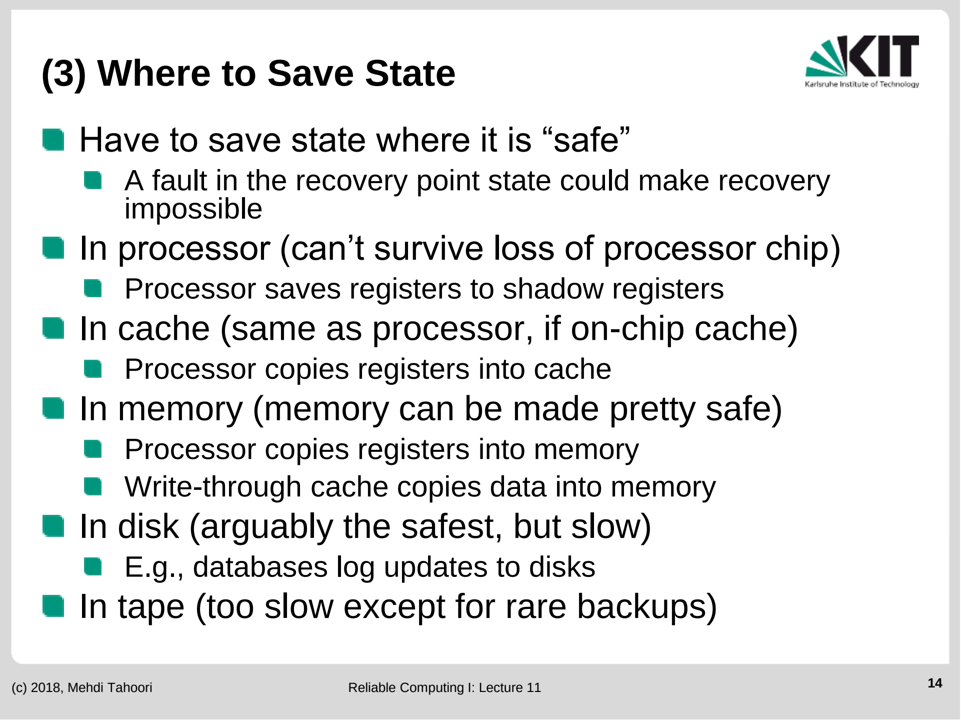
- Need to save all state that would be necessary if this were to become the recovery point
- Process state
  - Volatile states
    - Program stack (local variables, return pointers of function calls)
    - Program counter, stack pointer, open file descriptors, signal handlers
    - Static and dynamic data segments
  - Persistent states
    - User files related to the current program execution (whether to include the persistent state in the process state depends on the application, e.g., the persistent state is often an important part of a long-running application)
- 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



(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?

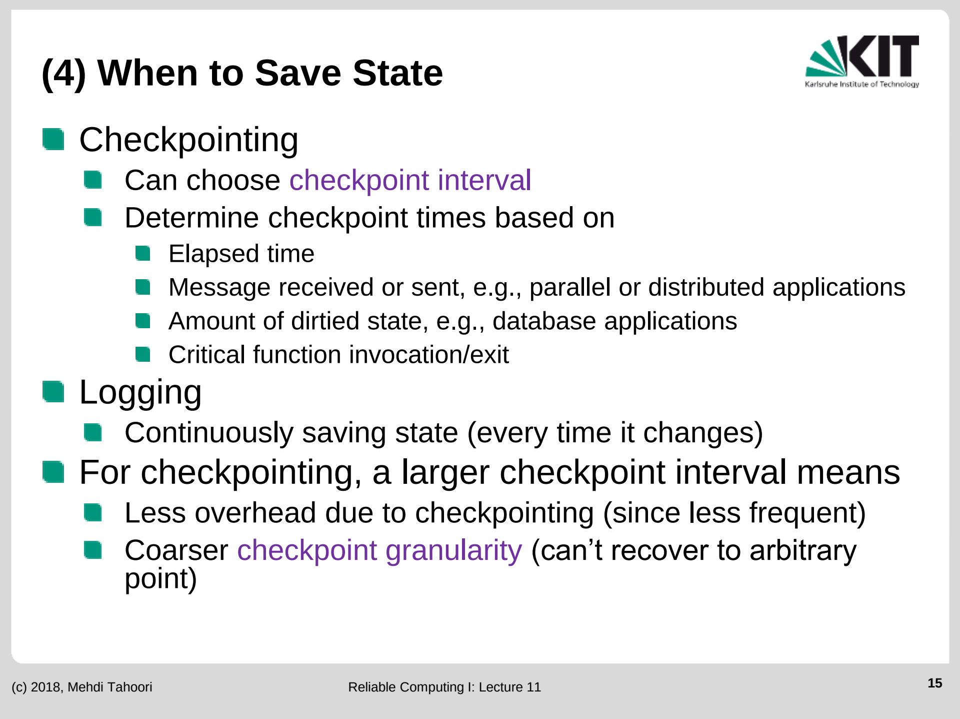
(c) 2018, Mehdi Tahoori Reliable Computing I: Lecture 11 13



(3) Where to Save State

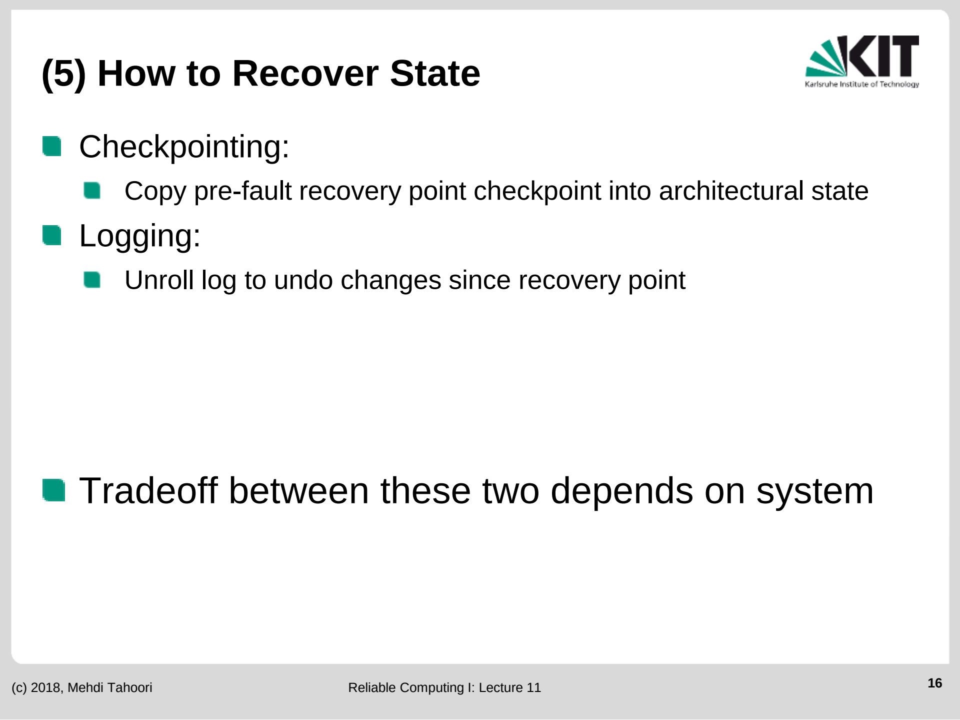
- Have to save state where it is “safe”
  - A fault in the recovery point state could make recovery impossible
- In processor (can’t survive loss of processor chip)
  - Processor saves registers to shadow registers
- In cache (same as processor, if on-chip cache)
  - Processor copies registers into cache
- 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)

(c) 2018, Mehdi Tahoori Reliable Computing I: Lecture 11 14



(4) When to Save State

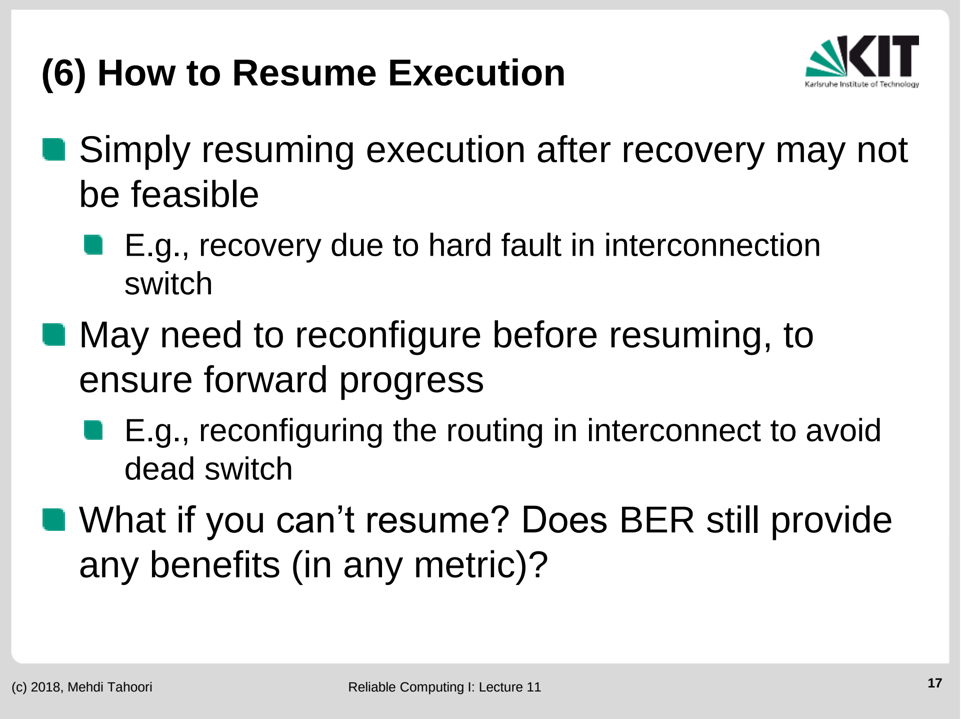
- Checkpointing
  - Can choose checkpoint interval
  - Determine checkpoint times based on
    - Elapsed time
    - Message received or sent, e.g., parallel or distributed applications
    - Amount of dirtied state, e.g., database applications
    - Critical function invocation/exit
- Logging
  - Continuously 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

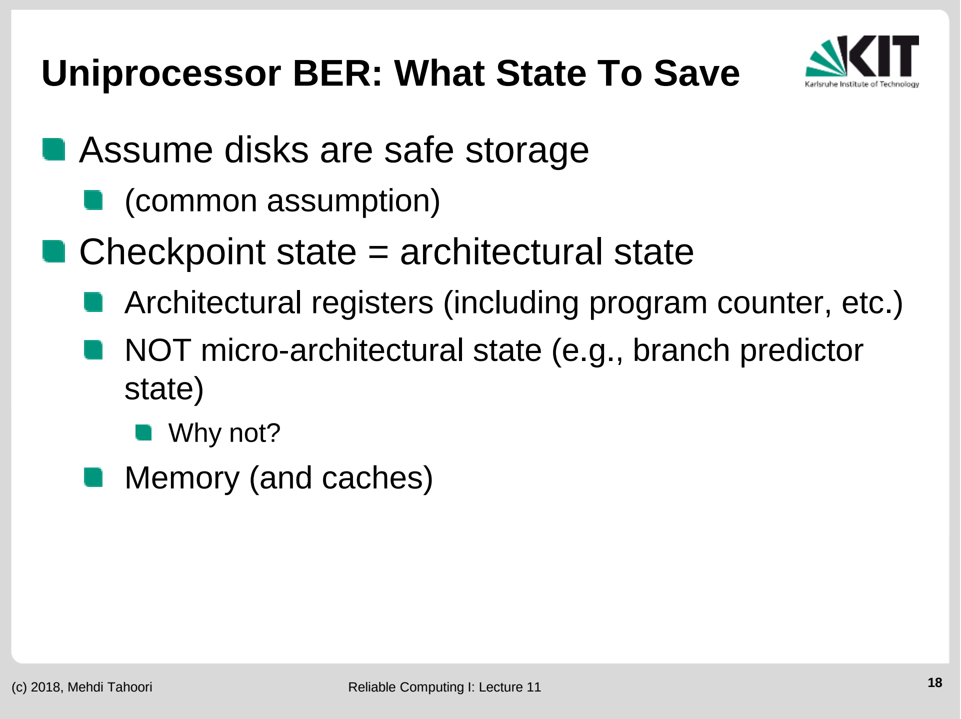
- Checkpointing:
  - Copy pre-fault recovery point checkpoint into architectural state
- Logging:
  - Unroll log to undo changes since recovery point

- Tradeoff between these two depends on system



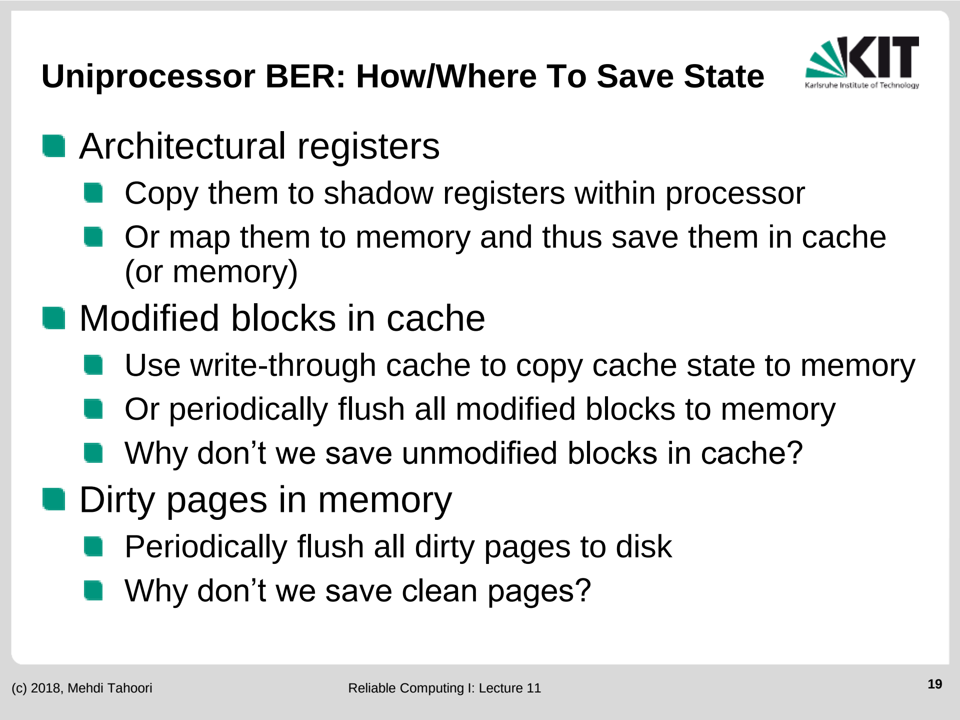
(6) How to Resume Execution

- 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)?



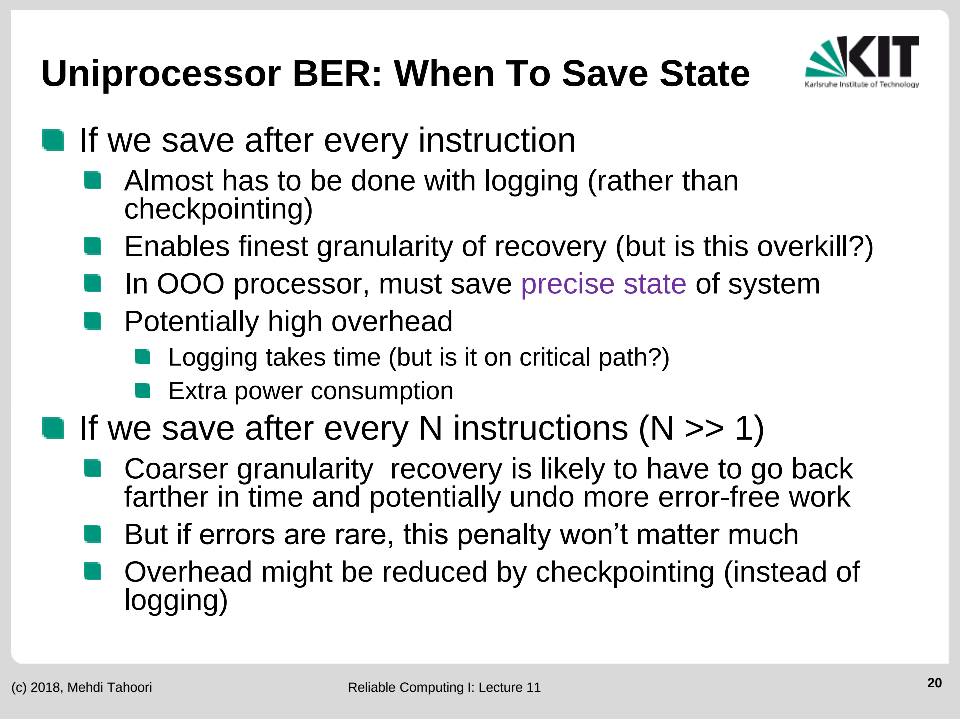
Uniprocessor BER: What State To Save

- 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)



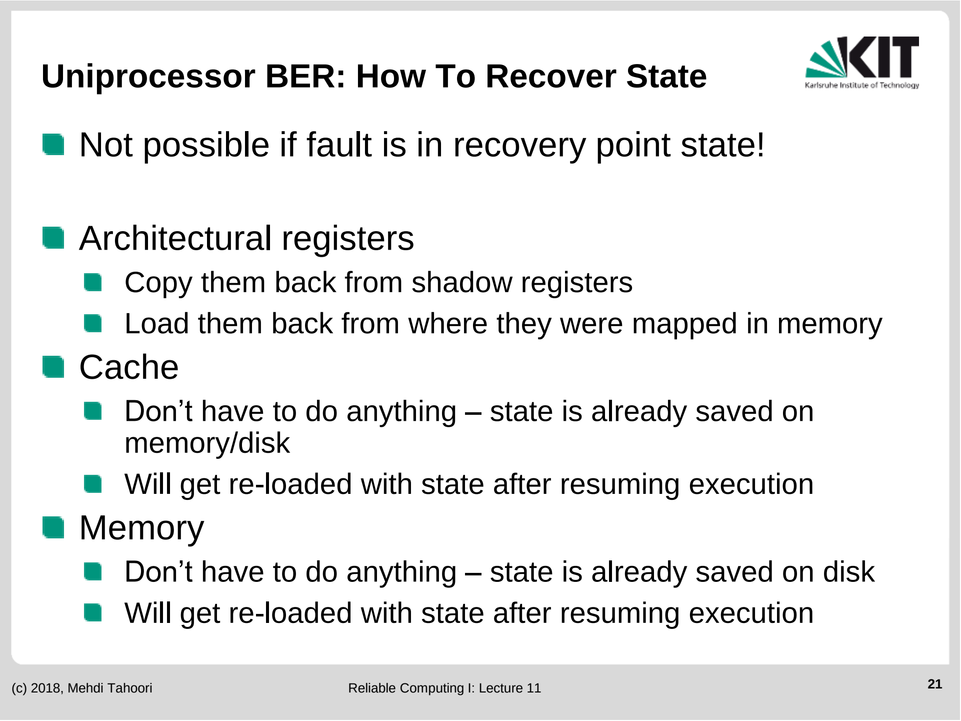
Uniprocessor BER: How/Where To Save State

- 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?



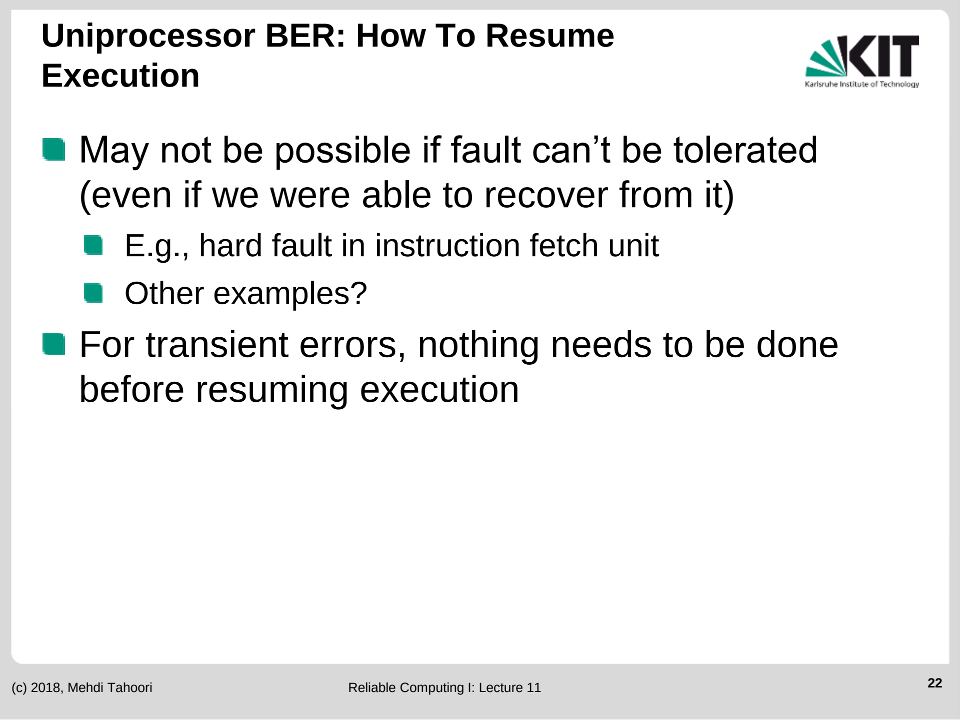
Uniprocessor BER: When To Save State

- 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)



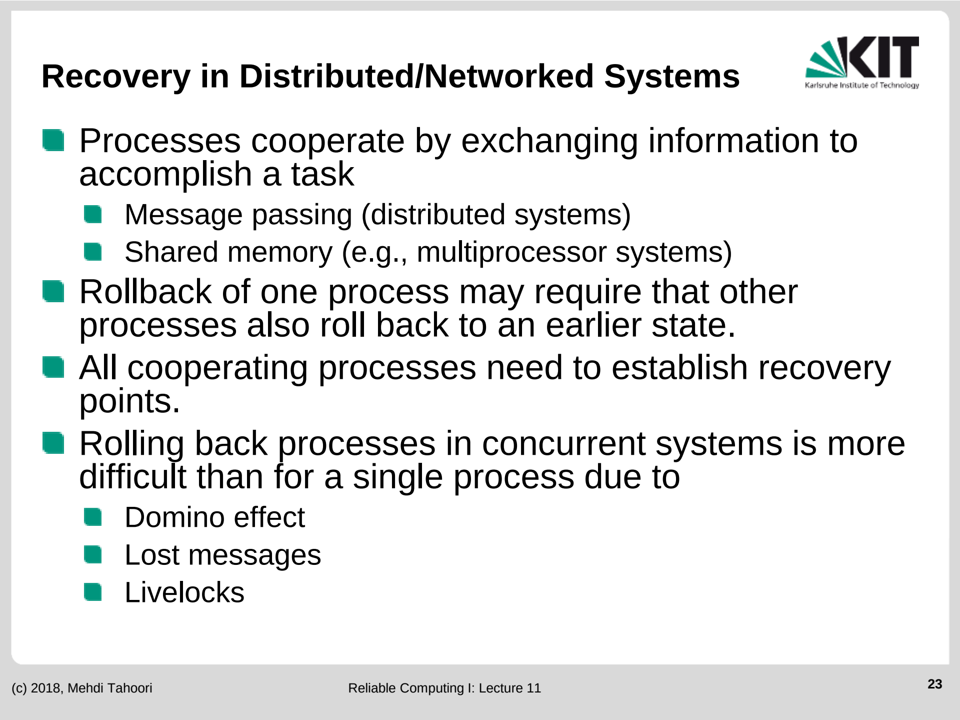
Uniprocessor BER: How To Recover State

- 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



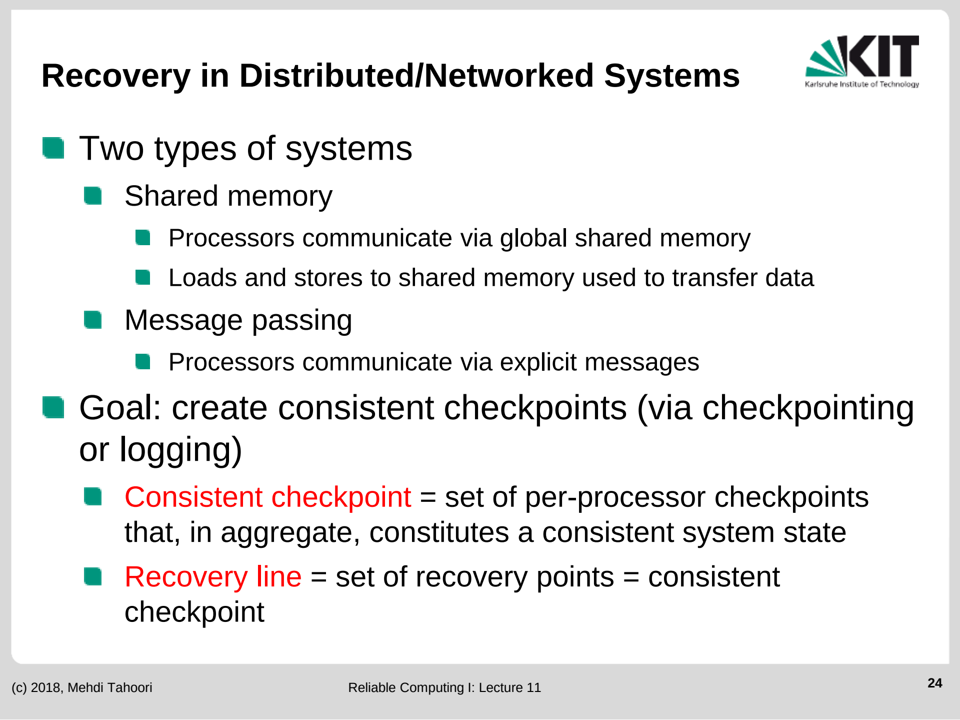
Uniprocessor BER: How To Resume Execution

- 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



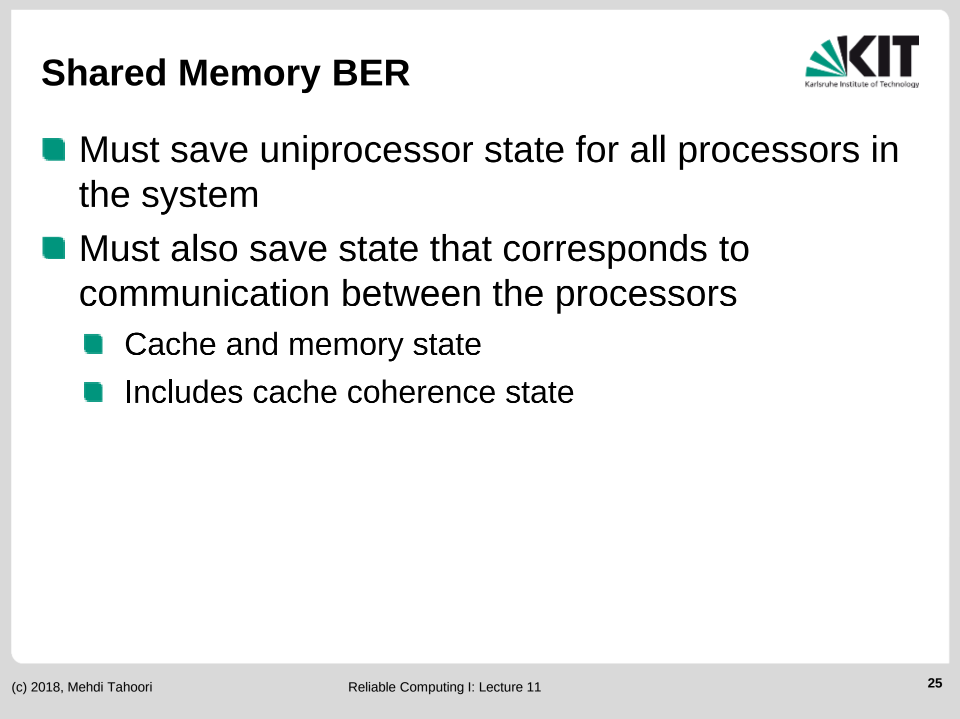
Recovery in Distributed/Networked Systems

- Processes cooperate by exchanging information to accomplish a task
  - Message passing (distributed systems)
  - Shared memory (e.g., multiprocessor systems)
- Rollback of one process may require that other processes also roll back to an earlier state.
- All cooperating processes need to establish recovery points.
- Rolling back processes in concurrent systems is more difficult than for a single process due to
  - Domino effect
  - Lost messages
  - Livelocks



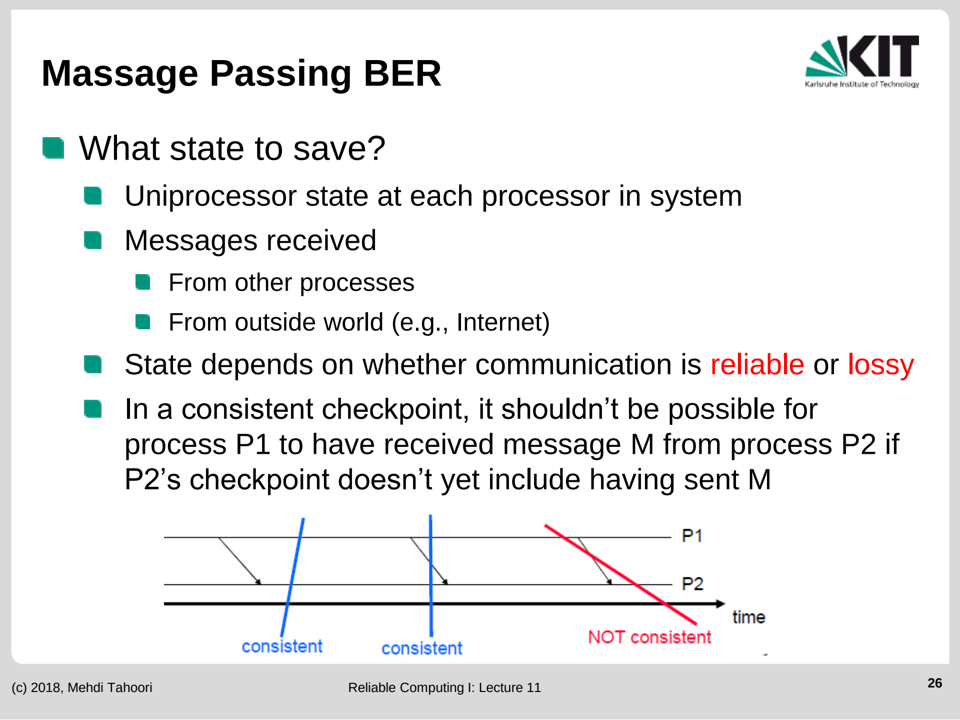
Recovery in Distributed/Networked Systems

- Two types of systems
  - 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



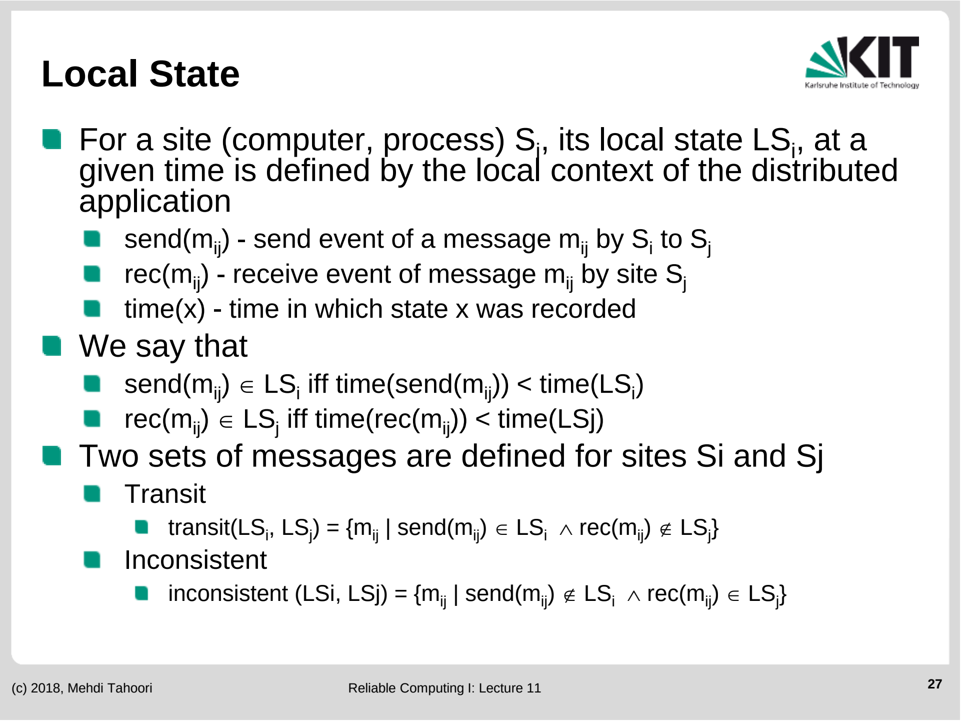
Shared Memory BER

- Must save uniprocessor state for all processors in the system
- Must also save state that corresponds to communication between the processors
  - Cache and memory state
  - Includes cache coherence state



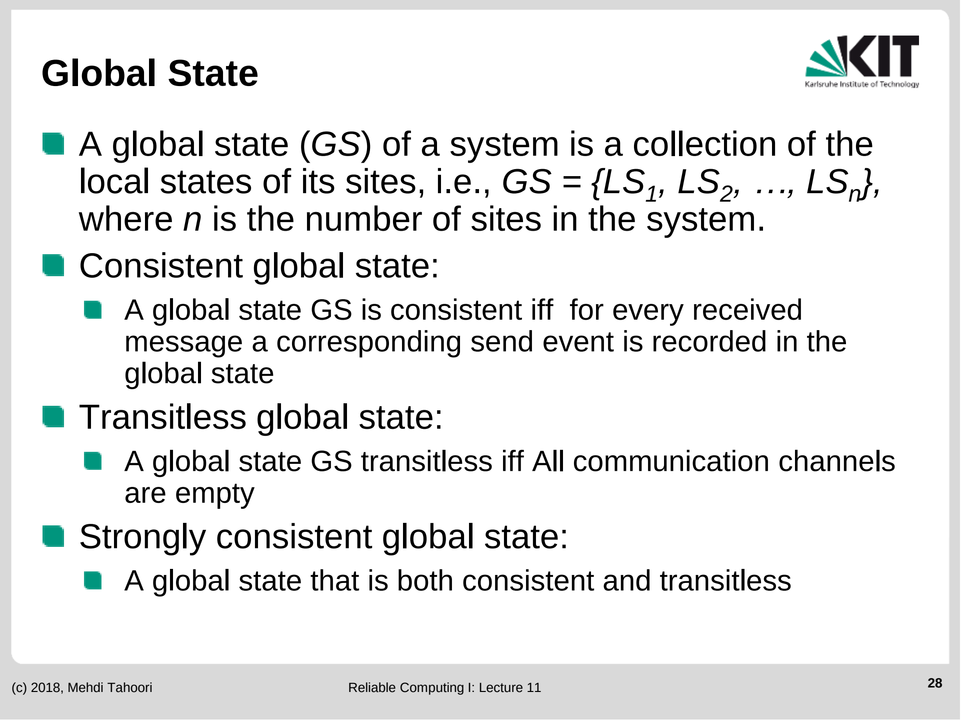
Massage Passing BER

- What state to save?
  - Uniprocessor state at each processor in system
  - Messages received
    - From other processes
    - From outside world (e.g., Internet)
  - State depends on whether communication is reliable or lossy
  - 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



Local State

- For a site (computer, process) S_i, its local state LS_i, at a given time is defined by the local context of the distributed application
  - send(m_{ij}) - send event of a message m_{ij} by S_i to S_j
  - rec(m_{ij}) - receive event of message m_{ij} by site S_j
  - time(x) - time in which state x was recorded
- We say that
  - send(m_{ij}) $\in LS_i$ iff time(send(m_{ij})) < time(LS_i)
  - rec(m_{ij}) $\in LS_j$ iff time(rec(m_{ij})) < time(LSj)
- Two sets of messages are defined for sites Si and Sj
  - Transit
    - $\text{transit}(LS_i, LS_j) = \{m_{ij} \mid \text{send}(m_{ij}) \in LS_i \wedge \text{rec}(m_{ij}) \notin LS_j\}$
  - Inconsistent
    - $\text{inconsistent (LSi, LSj)} = \{m_{ij} \mid \text{send}(m_{ij}) \notin LS_i \wedge \text{rec}(m_{ij}) \in LS_j\}$

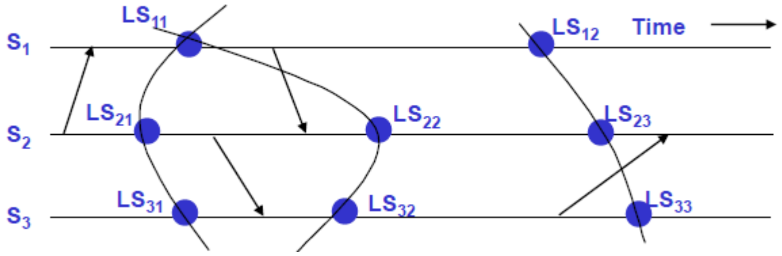


Global State

- A global state (*GS*) of a system is a collection of the local states of its sites, i.e., $GS = \{LS_1, LS_2, \ldots, LS_n\}$, where *n* is the number of sites in the system.
- Consistent global state:
  - A global state GS is consistent iff for every received message a corresponding send event is recorded in the global state
- Transitless global state:
  - A global state GS transitless iff All communication channels are empty
- Strongly consistent global state:
  - A global state that is both consistent and transitless

Local/Global State - Examples

- The global states:
  - $GS_1 = \{LS_{11}, LS_{21}, LS_{31}\}$ is a strongly consistent global state.
  - $GS_2 = \{LS_{12}, LS_{23}, LS_{33}\}$ is a consistent global state .
  - $GS_3 = \{LS_{11}, LS_{22}, LS_{32}\}$ is an inconsistent global state.



Uncoordinated Checkpointing

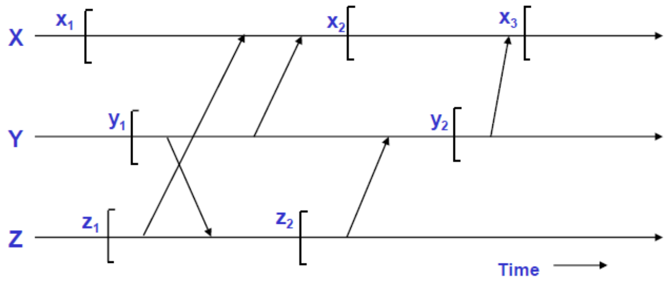
- Each process independently takes checkpoint
  - Doesn’t coordinate with other processes
- Pros
  - Easier to implement
  - No performance penalty for coordination
- Cons
  - May be tougher to recover
  - Tough/impossible to create consistent recovery line
    - Might end up with some inconsistent checkpoints
    - Could lead to cascading rollbacks (aka “domino effect”)

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

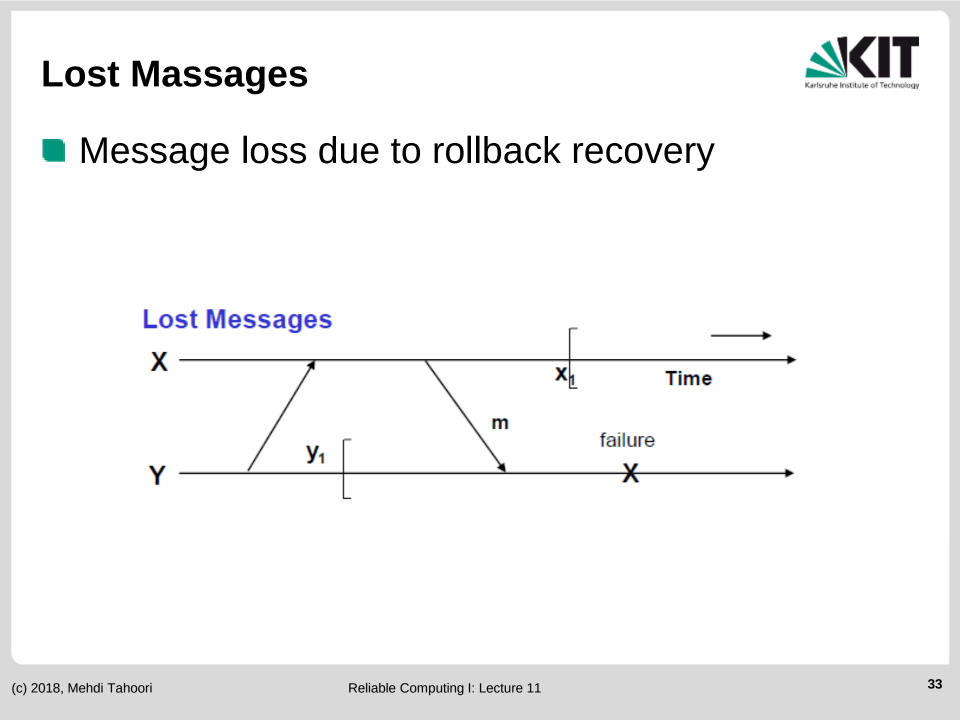
Domino Effect: Example

- Rollback of X does not affect other processes.
- Rollback of Z requires all three processes to roll back to their very first recovery points.



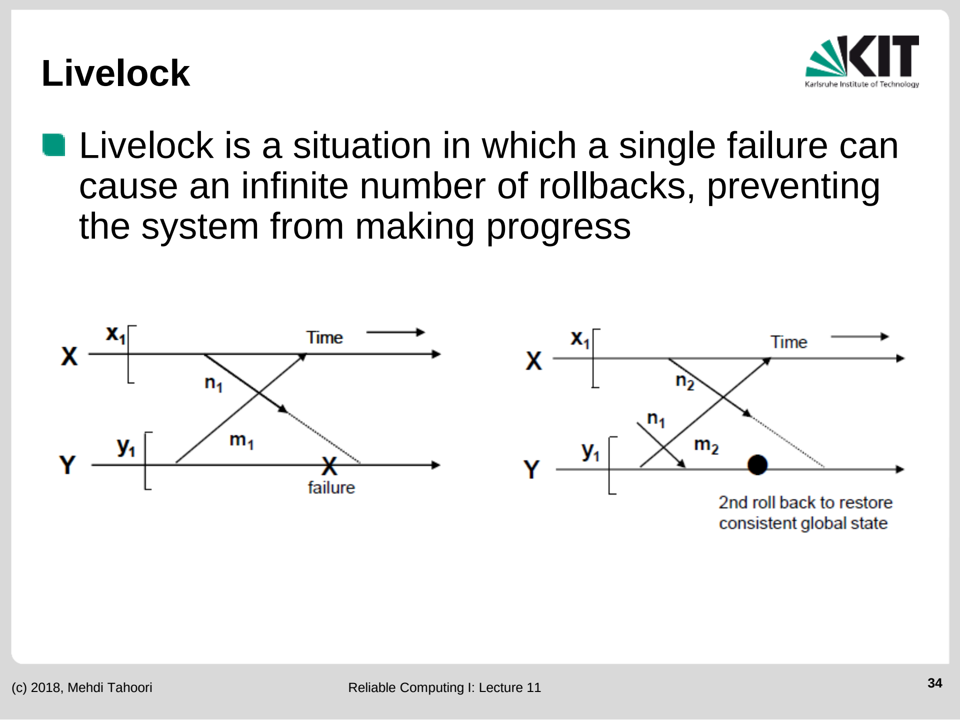
X, Y, Z - cooperating processes

[- recovery points



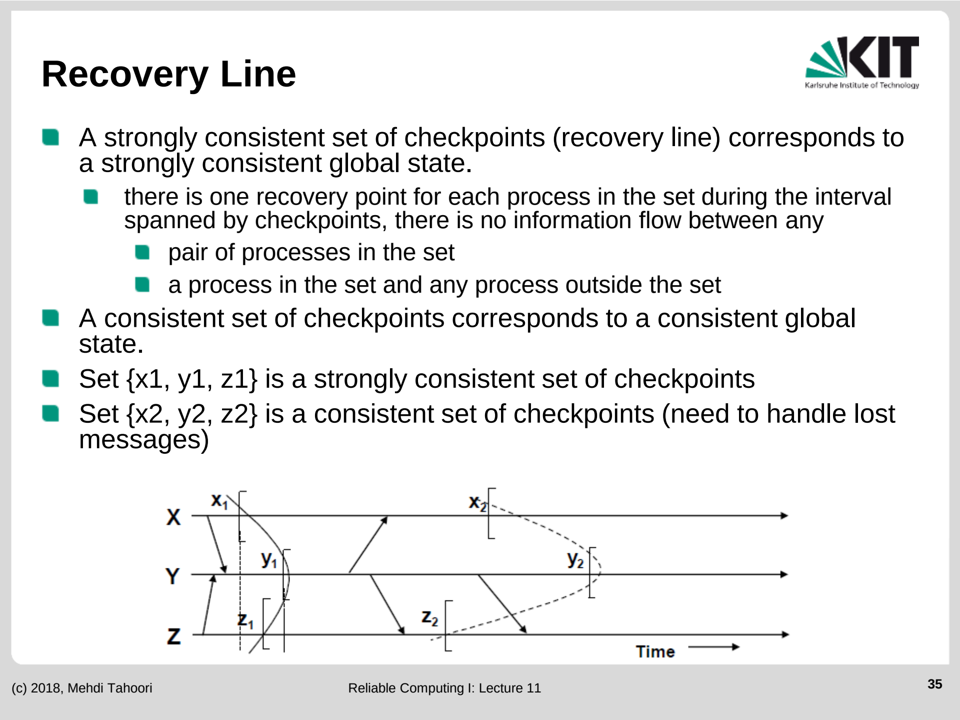
Lost Massages

- Message loss due to rollback recovery



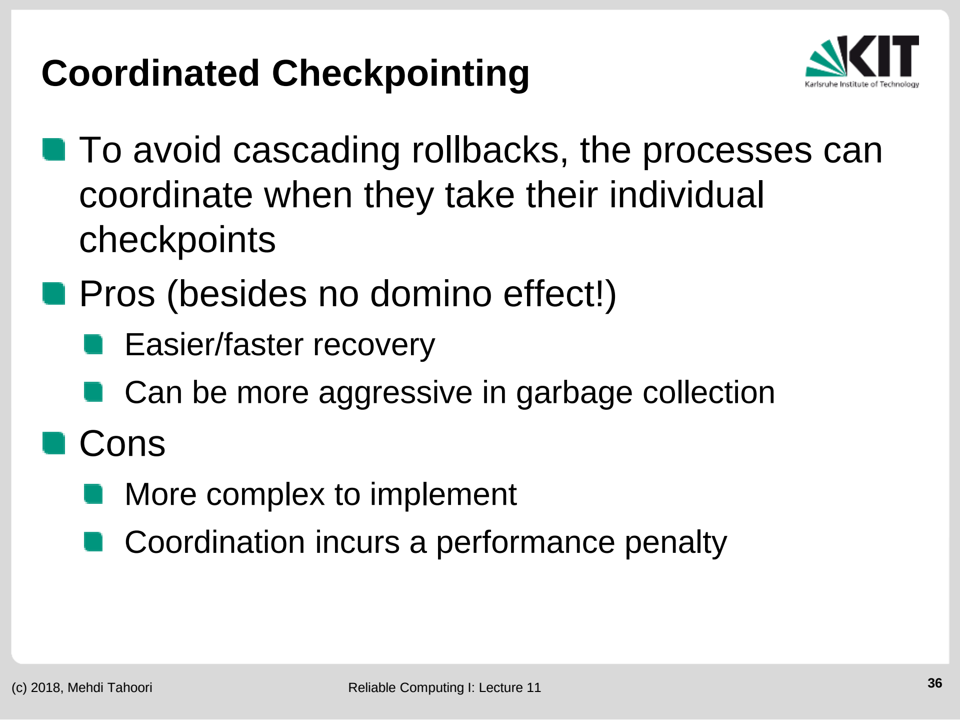
Livelock

- Livelock is a situation in which a single failure can cause an infinite number of rollbacks, preventing the system from making progress



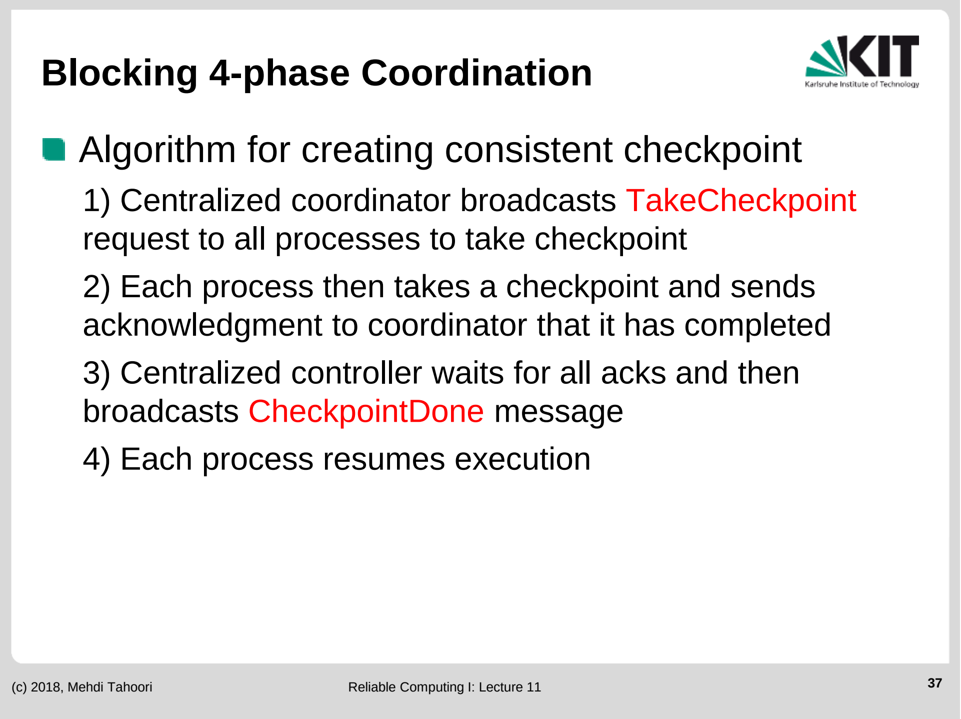
Recovery Line

- A strongly consistent set of checkpoints (recovery line) corresponds to a strongly consistent global state.
  - there is one recovery point for each process in the set during the interval spanned by checkpoints, there is no information flow between any
    - pair of processes in the set
    - a process in the set and any process outside the set
- A consistent set of checkpoints corresponds to a consistent global state.
- Set {x1, y1, z1} is a strongly consistent set of checkpoints
- Set {x2, y2, z2} is a consistent set of checkpoints (need to handle lost messages)



Coordinated Checkpointing

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



Blocking 4-phase Coordination

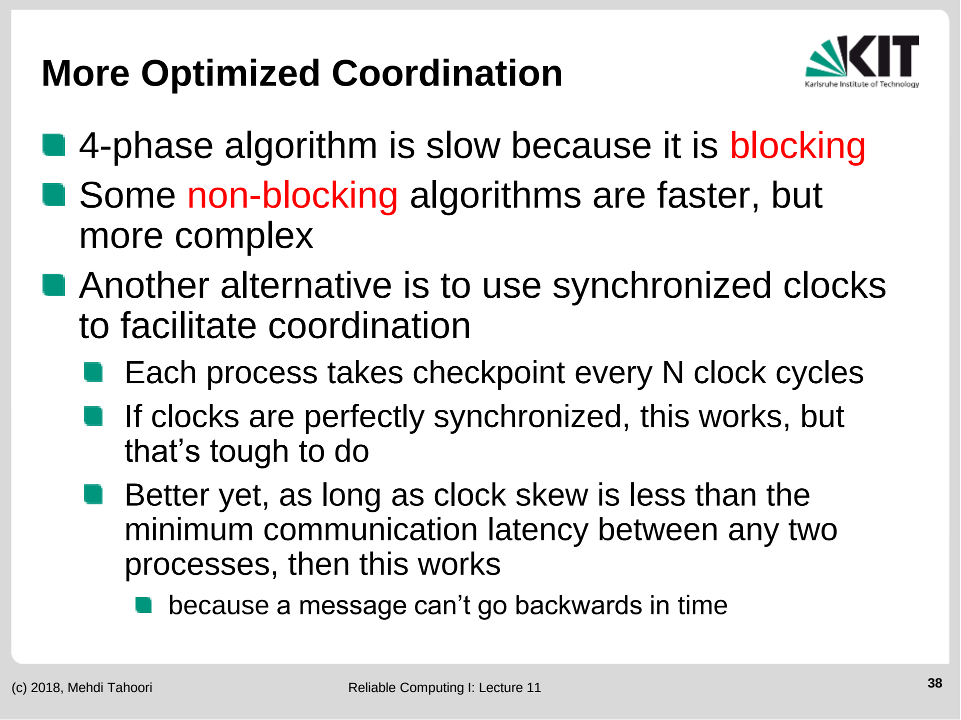
- Algorithm for creating consistent checkpoint

  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

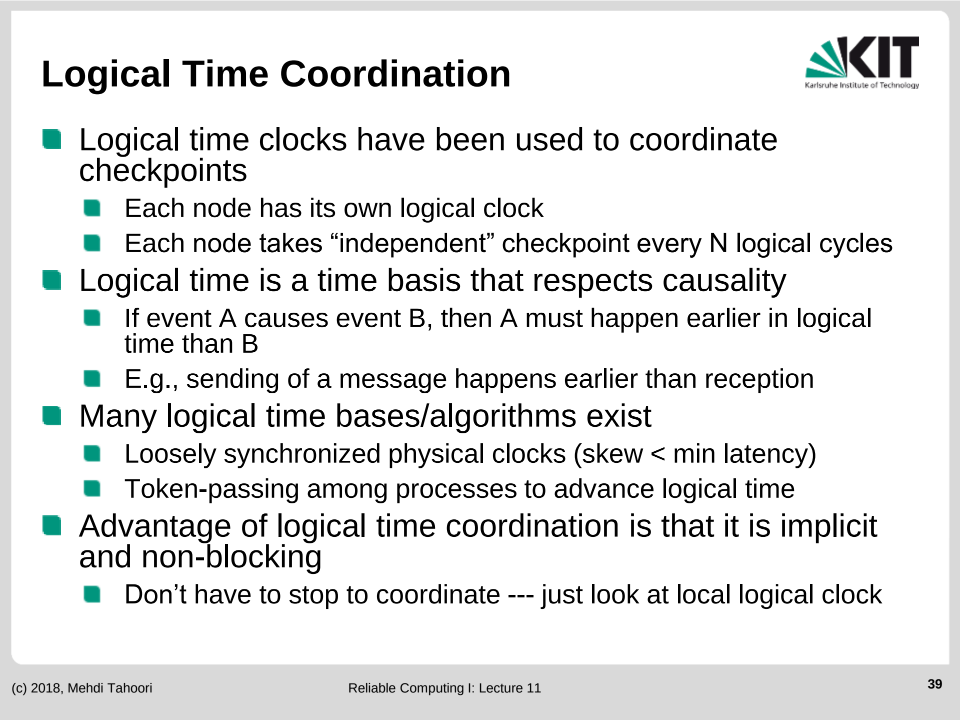
  3) Centralized controller waits for all acks and then broadcasts CheckpointDone message

  4) Each process resumes execution



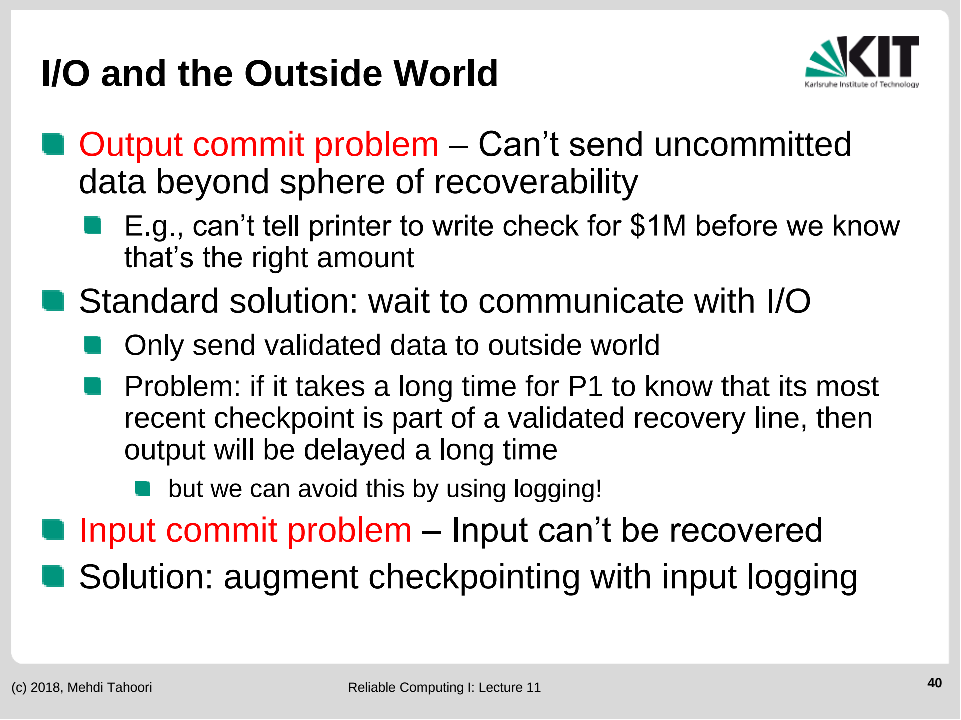
More Optimized Coordination

- 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



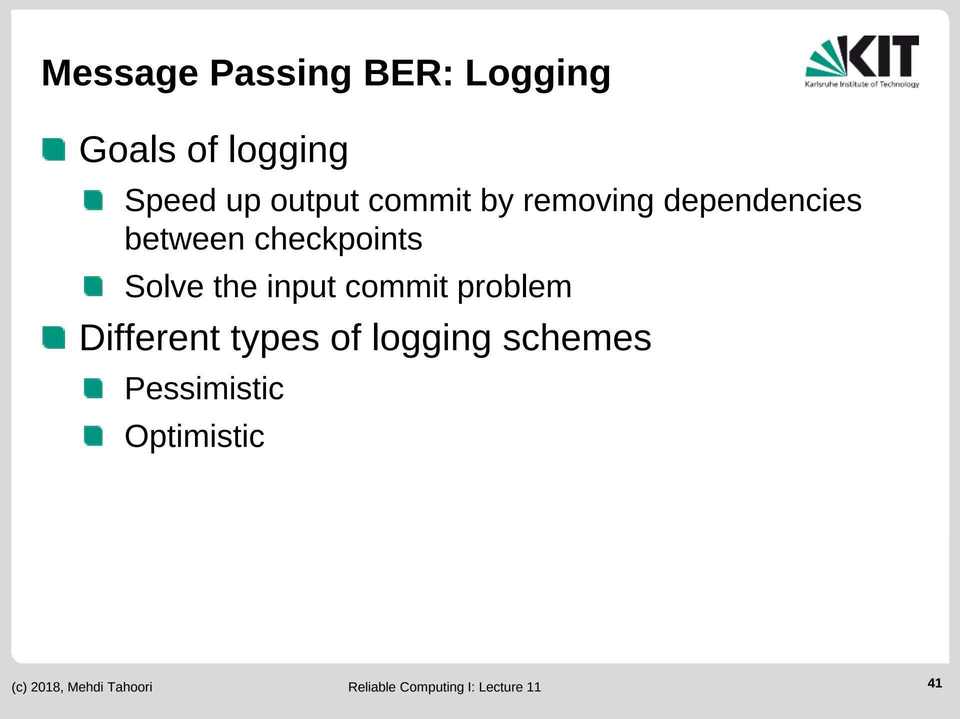
Logical Time Coordination

- 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



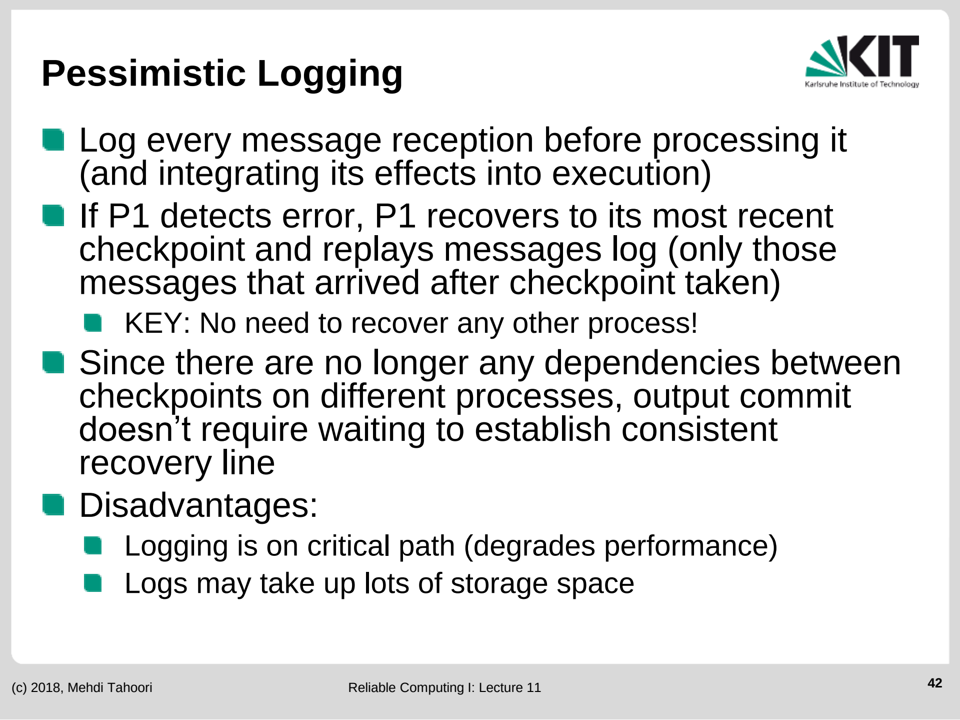
I/O and the Outside World

- 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



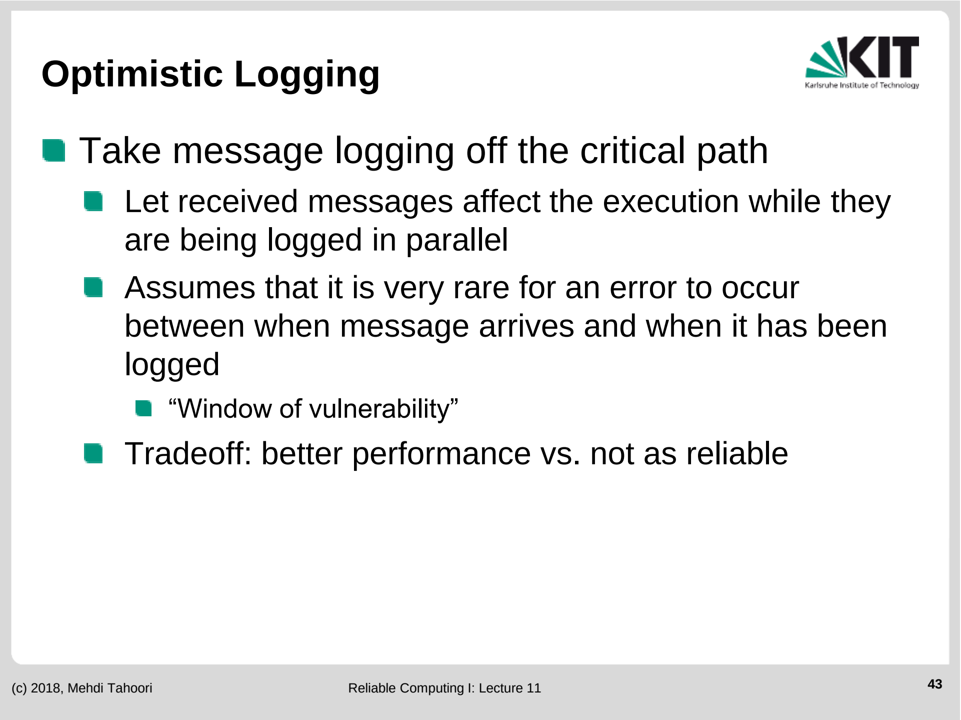
Message Passing BER: Logging

- Goals of logging
  - Speed up output commit by removing dependencies between checkpoints
  - Solve the input commit problem
- Different types of logging schemes
  - Pessimistic
  - Optimistic



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