Chapter 6 – Checkpointing I
Failure During Program Execution

♦ Computers today are much faster, but applications are more complicated

♦ Applications which still take a long time -
  * (1) Database Updates
  * (2) Fluid-flow Simulation - weather and climate modeling
  * (3) Optimization - optimal deployment of resources by industry (e.g. - airlines)
  * (4) Astronomy - N-body simulations and modeling of universe
  * (5) Biochemistry - study of protein folding

♦ When execution time is very long - both probability of failure during execution and cost of failure become significant
Cost of Program Failures - Model

♦ Program takes $T$ hours to execute
♦ Transient failures - constant rate $\lambda$ failures per hour
  * Failure is instantaneous but all prior work is lost
♦ $E$ - expected total execution time including any computational work lost due to failures

♦ Case I: No failures during execution
  * Probability of case I: $e^{-\lambda T}$
  * Conditional expected total execution time = $T$

♦ Case II: A failure occurs at $\tau$ hours ($0 \leq \tau \leq T$) into the execution
  * Probability of case II: $\lambda e^{-\lambda \tau} d\tau$
  * Conditional execution time = $\tau + E$ : time $\tau$ wasted, program restarted and additional expected time $E$ needed to complete execution
Cost Model - Cont.

♦ Contribution of Case II:
\[
\int_{\tau=0}^{T} (\tau + E) \lambda e^{-\lambda \tau} \, d\tau = \frac{1}{\lambda} + E - e^{-\lambda T} \left\{ \frac{1}{\lambda T} + T + E \right\}
\]

♦ Combining both cases -
\[
E = Te^{-\lambda T} + \frac{1}{\lambda} + E - e^{-\lambda T} \left\{ \frac{1}{\lambda} + T + E \right\}
\]

♦ Solving for E:
\[
E = \frac{e^{\lambda T} - 1}{\lambda}
\]

♦ \( \eta \) - a dimensionless measure of the overhead
\[
\eta = \frac{E - T}{T} = \frac{E}{T} - 1 = \frac{e^{\lambda T} - 1}{\lambda T} - 1
\]

♦ \( \eta \) depends only on product \( \lambda T \)
  - expected number of failures during program execution time

♦ \( \eta \) increases very fast (exponentially) with \( \lambda T \)

♦ Preferably - not start from the beginning with every failure - checkpointing

\( \eta \)

\( \lambda T \)

Copyright 2007 Koren & Krishna, Morgan-Kaufman
Checkpointing - Definition

- A **checkpoint** is a snapshot of entire state of the process at the moment it was taken
  - all information needed to restart the process from that point
- Checkpoint saved on **stable storage** of sufficient reliability
- Most commonly used - **Disks**: can hold data even if power is interrupted (but no physical damage to disk); can hold enormous quantities of data very cheaply
- Checkpoints can be very large - tens or hundreds of megabytes
- **RAM** with a battery backup is also used as stable storage
- No medium is perfectly reliable - reliability must be sufficiently high for the application at hand
- Store data at a remote location to tolerate damage due to local conditions
Overhead and Latency of Checkpoint

- **Checkpoint Overhead**: increase in execution time of application due to taking a checkpoint
- **Checkpoint Latency**: time needed to save checkpoint
- In a simple system - overhead and latency are identical
- If part of checkpointing can be overlapped with application execution - overhead may be substantially smaller than latency
- **Example**: A process checkpoints by writing its state into an internal buffer - CPU can continue execution while the checkpoint is written from buffer to disk
Checkpointing Latency
Example

for (i=0; i<1000000; i++)
  if (f(i)<min) {min=f(i); imin=i;}
for (i=0; i<100; i++) {
  for (j=0; j<100; j++) {
    c[i][j] += i*j/min;
  }
}

Latency depends on checkpoint size - is program dependent and can change during execution
  ♦ few kilobytes or as large as several gigabytes
  ♦ 1st part: small checkpoint - only program counter and variables min and imin
  ♦ 2nd part: checkpoint must include c[i][j] computed so far

1st part - compute smallest value of f(i) for 0<i<1000000
2nd part - multiplication followed by division
Issues in Checkpointing

♦ At what level (kernel/user/application) should we check point?
♦ How transparent to user should checkpointing be?
♦ How many checkpoints should we have?
♦ At which points during the program execution should we checkpoint?
♦ How can we reduce checkpointing overhead?
♦ How do we checkpoint distributed systems with/without a central controller?
♦ How do we restart the computation at a different node if necessary
Checkpointing at the Kernel Level

♦ Transparent to user; no changes to program
♦ When system restarts after failure - kernel responsible for managing recovery operation
♦ Every OS takes checkpoints when process preempted
  * process state is recorded so that execution can resume from interrupted point without loss of computational work
♦ But, most OS have little or no checkpointing for fault tolerance

Checkpointing at the User Level

♦ A user-level library provided for checkpointing
  * Application programs are linked to this library
♦ Like kernel-level checkpointing, this approach generally requires no changes to application code
♦ Library also manages recovery from failure
Checkpointing at the Application Level

♦ Application responsible for all checkpointing functions
♦ Code for checkpointing & recovery part of application
♦ Provides greatest control over checkpointing process
♦ Disadvantage - expensive to implement and debug

Comparing Checkpointing Levels

♦ Information available to each level may be different
♦ Multiple threads - invisible at the kernel
♦ User & application levels do not have access to information held at kernel level
  * Cannot assign process identifying numbers - can be a problem
♦ User & application levels may not be allowed to checkpoint parts of file system
  * May have to store names and pointers to appropriate files
Optimal Checkpointing - Analytic Model

Boxes denote latency; shaded part = overhead

Latency - total checkpointing time

Overhead - part of checkpointing not done in parallel with application execution - CPU is busy checkpointing

Overhead has a greater impact on performance than latency

Latency $T_{lt} = t_2 - t_0 = t_5 - t_3 = t_8 - t_6$

Overhead $T_{ov} = t_1 - t_0 = t_4 - t_3 = t_7 - t_6$
♦ **Checkpoint** represents state of system at $t_0, t_3, t_6$

♦ If a failure occurs in $[t_3, t_5]$ - checkpoint is useless - system must roll back to previous checkpoint $t_0$

♦ $T_{r}$ - average recovery time - time spent in a faulty state plus time to recover to a functional state

♦ $E_{int}$ - amount of time between completions of two consecutive checkpoints

♦ $T_{ex}$ - amount of time spent executing application during this time

♦ $T$ - program execution time; $N$ uniformly placed checkpoints

♦ $T_{ex} = T/(N+1)$
Analytic Model – Calculating $E_{int}$

♦ **Case I:** No failures during $T_{ex}+T_{ov}$
  * $E_{int} = T_{ex}+T_{ov}$

♦ **Case II:** Failure occurs $\tau$ hours into $T_{ex}+T_{ov}$
  * We lose all work done after preceding checkpoint was taken = $T_{lt} - T_{ov} + \tau$
  * It takes an average of $T_r$ hours to recover
  * Total amount of additional time
    $= \tau + T_{lt} - T_{ov} + T_r$
  * Average value of $\tau$ = $(T_{ex}+T_{ov})/2$
  * Average additional time
    $= (T_{ex}+T_{ov})/2 + T_{lt} - T_{ov} + T_r$
Calculating $E_{int} - \text{First Order Approximation}$

♦ **Assumption** - at most one failure strikes the system between successive checkpoints

* Good approximation if $T_{ex} + T_{ov}$ is small compared to average time between failures $1/\lambda$

♦ **Contribution of case I:** 

$$ (T_{ex} + T_{ov}) e^{-\lambda(T_{ex}+T_{ov})} $$

♦ **Contribution of case II:**

$$ (1 - e^{-\lambda(T_{ex}+T_{ov})}) \left\{ T_{ex} + T_{ov} + \frac{T_{ex} + T_{ov}}{2} + T_{r} + T_{lt} - T_{ov} \right\} $$

♦ **Sum of both:**

$$ E_{int} \approx \frac{3}{2} T_{ex} + \frac{T_{ov}}{2} + T_{r} + T_{lt} - \left( \frac{T_{ex}}{2} + T_{r} + T_{lt} - \frac{T_{ov}}{2} \right) e^{-\lambda(T_{ex}+T_{ov})} $$

♦ \[
\frac{dE_{int}}{dT_{ov}} \gg \frac{dE_{int}}{dT_{lt}} - E_{int} \text{ more sensitive to } T_{ov} \text{ than to } T_{lt}
\]
Optimal Checkpoint Placement - Approximation

♦ In previous analysis – a given number \( N \) of equally spaced checkpoints and \( T_{\text{ex}} = T/(N+1) \)

♦ Optimal checkpoint placement problem – determine \( N \) (or \( T_{\text{ex}} \)) with the objective of minimizing the total execution time \((N+1)E_{\text{int}}\)
or equivalently, minimizing \( \eta = E_{\text{int}}/T_{\text{ex}} - 1 \)

♦ Using \( e^{-\lambda(T_{\text{ex}}+T_{\text{ov}})} \approx 1 - \lambda(T_{\text{ex}} + T_{\text{ov}}) \), we obtain

\[
\eta = \frac{\frac{3}{2}T_{\text{ex}} + \frac{T_{\text{ov}}}{2} + T_r + T_{\text{lt}} - \left(\frac{T_{\text{ex}}}{2} + T_r + T_{\text{lt}} - \frac{T_{\text{ov}}}{2}\right)(1 - \lambda(T_{\text{ex}} + T_{\text{ov}}))}{T_{\text{ex}}} - 1
\]

\[
= \frac{(T_{\text{ex}} + T_{\text{ov}}) \left[1 + \lambda\left(\frac{T_{\text{ex}}}{2} + T_r + T_{\text{lt}} - \frac{T_{\text{ov}}}{2}\right)\right]}{T_{\text{ex}}} - 1
\]

♦ and

\[
T_{\text{ex}}^{\text{opt}} = \sqrt{\frac{2T_{\text{ov}}}{\lambda} + 2T_{\text{ov}} \left(T_r + T_{\text{lt}} - \frac{T_{\text{ov}}}{2}\right)} \quad N_{\text{opt}} = \frac{T}{T_{\text{ex}}^{\text{opt}}} - 1
\]
Is Uniform Placement Optimal?

- Previously - we assumed that checkpoints are placed uniformly along the time axis
- Is this optimal?
- If the checkpointing cost is the same, irrespective of when the checkpoint is taken, the answer is “yes”
- If checkpoint size (and cost) vary from one part of the execution to the other, the answer is often “no”
Calculating $E_{\text{int}}$ – More Accurate Model

♦ More than one failure can occur between two checkpoints

* **Case I** remains the same

* **Case II**: Failure occurs $\tau$ hours into $T_{\text{ex}}+T_{\text{ov}}$

* We lose $\tau + T_{\text{lt}} - T_{\text{ov}} + T_{\text{tr}}$, after which computation resumes and takes an added average time of $E_{\text{int}}$

♦ **Contribution of Case II**:

\[
\int_{\tau=0}^{T_{\text{ex}}+T_{\text{ov}}} (\tau + T_{\text{r}} + T_{\text{lt}} - T_{\text{ov}} + E_{\text{int}})\lambda e^{-\lambda \tau} d\tau = E_{\text{int}} + T_{\text{r}} + T_{\text{lt}} - T_{\text{ov}} + \frac{1}{\lambda}
\]

\[
- \left( T_{\text{ex}} + T_{\text{r}} + T_{\text{lt}} + \frac{1}{\lambda} + E_{\text{int}} \right) e^{-\lambda(T_{\text{ex}}+T_{\text{ov}})}
\]

♦ **Adding the two cases**:

$E_{\text{int}} = (T_{\text{ex}}+T_{\text{ov}})e^{-\lambda(T_{\text{ex}}+T_{\text{ov}})} + E_{\text{int}} + T_{\text{r}} + T_{\text{lt}} - T_{\text{ov}} + \frac{1}{\lambda} - \left( T_{\text{ex}} + T_{\text{r}} + T_{\text{lt}} + \frac{1}{\lambda} + E_{\text{int}} \right) e^{-\lambda(T_{\text{ex}}+T_{\text{ov}})}$

♦ **The solution is**

\[
\mathcal{I}_{\mu}^{\mu} = \left( 1^\mu + 1^{\mu} - 1^{\mu_{\text{ov}}} + \frac{1}{\mu} \right) \left( \epsilon Y(1^{\text{ex}} + 1^{\mu_{\text{ov}}}) - 1 \right)
\]

* $E_{\text{int}}$ is more sensitive to $T_{\text{ov}}$ than to $T_{\text{lt}}
Optimal Checkpoint Placement - More Accurate Model

We are looking for $T_{ex}$ to minimize $\eta = \frac{E_{int}}{T_{ex}} - 1$

Using Calculus, the optimal $T_{ex}$ satisfies

$$e^{\lambda(T_{ex} + T_{ov})} = \frac{1}{1 - \lambda T_{ex}}$$

Optimal $T_{ex}$ does not depend on latency $T_{lt}$ or recovery time $T_{r}$

* Depends only on the overhead $T_{ov}$

And,

$$N_{opt} = \frac{T}{T_{ex}} - 1$$

Sequential Checkpointing

Application cannot be executed in parallel with checkpointing - $T_{lt} = T_{ov}$

$$\eta = \frac{(T_{r} + \frac{1}{\lambda})(e^\lambda(T_{ex} + T_{ov}) - 1)}{T_{ex}} - 1$$
Reducing Overhead - Buffering

- Processor writes checkpoint into main memory and then returns to executing application
- Direct memory access (DMA) is used to copy checkpoint from main memory to disk
  * DMA requires CPU involvement only at beginning and end of operation
- Refinement - copy on write buffering
- No need to copy portions of process state that are unchanged since last checkpoint
- If process does not update main memory pages too often - most of the work involved in copying pages to a buffer area can be avoided
Copy on Write Buffering

♦ Most memory systems provide memory protection bits (per page of physical main memory) indicating: (page) is read-write, read-only, or inaccessible

♦ When checkpoint is taken, protection bits of pages belonging to process are set to read-only

♦ Application continues running while checkpointed pages are transferred to disk

♦ If application attempts to update a page, an access violation is triggered

♦ System then buffers page, and permission is set to read-write

♦ Buffered page is later copied to disk

♦ This is an example of incremental checkpointing
Incremental Checkpointing

- Recording only changes in process state since the previous checkpoint was taken
- If these changes are few - less has to be saved per incremental checkpoint
- **Disadvantage**: Recovery is more complicated
- Not just loading latest checkpoint and resuming computation from there
- Need to build system state by examining a succession of incremental checkpoints
Reducing Checkpointing Overhead - Memory Exclusion

- Two types of variables that do not need to be checkpointed:
  * Those that have not been updated, and
  * Those that are “dead”

- A dead variable is one whose present value will never again be used by the program.

- Two kinds of dead variables:
  * Those that will never again be referenced by program, and
  * Those for which the next access will be a write

- The challenge is to accurately identify such variables.
Identifying Dead Variables

- The address space of a process has four segments: code, global data, heap, stack
  - Finding dead variables in code is easy: self-modifying code no longer used - code is read-only, no need to checkpoint
  - Stack segment equally easy: contents of addresses held in locations below stack pointer are obviously dead
    » virtual address space usually has stack segment at the top, growing downwards
  - Heap segment: many languages allow programmers to explicitly allocate and deallocate memory (e.g., malloc() and free() calls in C) - contents of free list are dead by definition
  - Some user-level checkpointing packages (e.g., libckpt) provide programmer with procedure calls (e.g., checkpoint_here()) that specify regions of memory that should be excluded from, or included in, future checkpoints
Reducing Latency

♦ Checkpoint compression - less written to disk
♦ How much is gained through compression depends on:
  * Extent of compression - application-dependent - can vary between 0 and 50%
  * Work required to execute the compression algorithm - done by CPU - adds to checkpointing overhead as well as latency

♦ In simple sequential checkpointing where $T_{lt} = T_{ov}$ - compression may be beneficial

♦ In more efficient systems where $T_{ov} << T_{lt}$ - usefulness of this approach is questionable and must be carefully assessed

♦ Another way of reducing latency is incremental checkpointing
CARER: Cache-Aided Rollback Error Recovery

- **CARER scheme**
  - Marks process footprint in main memory and cache as parts of checkpointed state
  - Reduces time required to take a checkpoint
  - Allows more frequent checkpoints
  - Reduces penalty of rollback upon failure

- Assuming memory and cache are less prone to failure than processor

- Checkpointing consists of storing processor's registers in main memory

- Includes processes' footprint in main memory + lines of cache marked as part of checkpoint
Checkpoint Bit For Each Cache Line

♦ Scheme requires hardware modification - an extra checkpoint bit associated with each cache line

♦ When bit is 1 - corresponding line is **unmodifiable**
  * Line is part of latest checkpoint
  * May not update without being forced to take a checkpoint immediately

♦ When bit is 0 - processor is free to modify word

♦ Process' footprint in memory + marked cache lines serve as both memory and part of checkpoint
  * Less freedom when deciding when to checkpoint

♦ Checkpointing is forced when
  * A line marked **unmodifiable** is to be updated
  * Anything in memory is to be updated
  * An **I/O** instruction is executed or an external interrupt occurs
Checkpointing and Roll Back

◆ Taking a checkpoint involves:
  * (a) Saving processor registers in memory
  * (b) Setting to 1 the checkpoint bit associated with each valid cache line

◆ Rolling back to previous checkpoint simple: restore registers, and mark invalid all cache lines with checkpoint bit = 0

◆ Cost:
  * A checkpoint bit for every cache line
  * Every write-back of a cache line into memory involves taking a checkpoint
Checkpointing in Distributed Systems

- **Distributed system**: processors and associated memories connected by an interconnection network
  - Each processor may have local disks
  - Can be a network file system accessible by all processors
- Processes connected by directional channels - point-to-point connections from one process to another
  - Assume channels are error-free and deliver messages in the order received
Process/Channel/System State

- The state of channel at $t$ is
  - set of messages carried by it up to time $t$
  - order in which they were received
- State of distributed system: aggregate states of individual processes and channels
- State is consistent if, for every message delivery there is a corresponding message-sending event
- A state violating this - a message delivered that had not yet been sent - violates causality
  - Such a message is called an orphan
- The converse - a system state reflecting sending of a message but not its receipt - is consistent
Consistent/Inconsistent States

♦ Example: 2 processes P and Q, each takes two checkpoints; Message m is sent by P to Q

♦ Checkpoint sets representing consistent system states:
  * \{CP_1, CQ_1\}: Neither checkpoint knows about m
  * \{CP_2, CQ_1\}: CP_2 indicates that m was sent; CQ_1 has no record of receiving m
  * \{CP_2, CQ_2\}: CP_2 indicates that m was sent; CQ_2 indicates that it was received

♦ \{CP_1, CQ_2\} is inconsistent:
  * CP_1 has no record of m being sent
  * CQ_2 records that m was received
  * m is an orphan message
Recovery Line

♦ Consistent set of checkpoints forms a recovery line
  - can roll system back to them and restart from there

♦ Example: \{CP1, CQ1\}
  * Rolling back P to CP1 undoes sending of m
  * Rolling back Q to CQ1 means: Q has no record of m
  * Restarting from CP1, CQ1, P will again send m

♦ Example: \{CP2, CQ1\}
  * Rolling back P to CP2 means: it will not retransmit m
  * Rolling back Q to CQ1: Q has no record of receiving m

♦ Recovery process has to be able to play back m to Q
  * Adding it to checkpoint of P, or
  * Have a separate message log which records everything received by Q
Useless Checkpoints

- Checkpoints can be useless
  - Will never form part of a recovery line
  - Taking them is a waste of time
- Example: CQ2 is a useless checkpoint
- CQ2 records receipt of m1, but not sending of m2
- \{CP_1, CQ_2\} not consistent
  - otherwise m1 would become an orphan
- \{CP_2, CQ_2\} not consistent
  - otherwise m2 would become an orphan
**The Domino Effect**

- A single failure can cause a sequence of rollbacks that send every process back to its starting point.
- Happens if checkpoints are not coordinated either directly (through message passing) or indirectly (by using synchronized clocks).
- **Example:** P suffers a transient failure
  - *Rolls back to checkpoint CP3*
  - *Q rolls back to CQ2 (so m6 will not be an orphan)*
  - *P rolls back to CP2 (so m5 will not be an orphan)*
  - *This continues until both processes have rolled back to their starting positions*
Disc. Messages

Suppose Q rolls back to CQ1 after receiving message m from P

All activity associated with having received m is lost

If P does not roll back to CP2 – the message was lost – not as severe as having orphan messages

m can be retransmitted

If Q sent an acknowledgment of that message to P before rolling back, then the acknowledgment would be an orphan message unless P rolls back to CP2
Livellock

- Another problem that can arise in distributed checkpointed systems
- \( Q \) sends \( P \) a message \( m_1 \); 
  \( P \) sends \( Q \) a message \( m_2 \)
- \( P \) fails before receiving \( m_1 \)
- \( Q \) rolls back to \( CQ_1 \) (otherwise \( m_2 \) is orphaned)
- \( P \) recovers, rolls back to \( CP_2 \), sends another copy of \( m_2 \), and then receives the copy of \( m_1 \) that was sent before all the rollbacks began
- Because \( Q \) has rolled back, this copy of \( m_1 \) is now orphaned, and \( P \) has to repeat its rollback
- This orphans the second copy of \( m_2 \) and \( Q \) must repeat its rollback
- This may continue indefinitely unless there is some outside intervention