7 Time Redundancy
Application of Time Redundancy

Time redundancy is used at all abstraction levels:

1. Transistor level and gate level
   - For fault detection

2. Gate level and RT-level
   - Shadow registers for fault detection and check-pointing

3. Finite State Machines
   - Control Flow Monitoring for error detection
   - Shadow registers for restoring

4. Processor level
   - Self-checking design
   - Shadow registers for check-pointing

5. Software and System level
Checkpointing

- Principle used in many time redundant schemes

1. Store intermediate results in course of computation
   ➔ Called checkpoints

2. Perform self-checking computing

3. If an error occurs, restore the closest checkpoint in the past (Rollback)
   ➔ Tolerates transient faults
Error is detected

Checkpoint 0  Checkpoint 1  Checkpoint 2

Checkpoint ≈ System State
Checkpoint Storage

- The checkpoint is saved on storage of sufficient reliability
  - ECC protected memory or shadow registers
  - Battery backup memory
  - Non-volatile memory (Disks, etc.)

- No medium is perfectly reliable
  - Reliability must be sufficiently high for the considered application
Overhead and Latency of a Checkpoint

- **Checkpoint Overhead:**
  Increase in execution time of application due to taking a checkpoint

- **Checkpoint Latency:**
  Time until checkpoint is safely stored. May overlap with application execution.

- In a simple system - overhead and latency are identical
Selection of Checkpoint-Interval

- If too many checkpoints are used, the system performance decreases
  - The normal flow of the program is stopped at each checkpoint in order to save state information

- If the checkpoints are too far apart, the recovery time will increase
  - The program will have to be reexecuted from the last checkpoint
Derive expected computation time given failure rate and checkpoint overhead

Assumptions
- Checkpoint latency = 0
- Const. checkpoint overhead $T_{OV} = t_1 - t_0 = t_3 - t_2 = t_5 - t_4$
- Const. computation window $T_C$
- Const. time for recovery $T_R$
- Const. failure rate $\lambda$
Analytical Model (II)

- **First order approximation:** At most one failure happens between two subsequent checkpoints

- **Case 1:** No failure during one interval
  - Length of one interval: $E_{NF} = T_C + T_{OV}$
  - Probability that no failure occurs during that interval:
    \[ P_{NF} = e^{-\lambda(T_C + T_{OV})} \]
Case 2: One failure during one interval

- Probability that (at least) one failure occurs during that interval:
  \[ P_F = 1 - e^{-\lambda(T_C + T_{OV})} \]

- Assume failure occurs at \( \tau \) time units, expected value of \( \tau \):
  \[ E_\tau = \frac{T_C + T_{OV}}{2} \]

- Additional time due to failure: \( \tau + T_R \) with expected value:
  \[ E_\tau + T_R = \frac{T_C + T_{OV}}{2} + T_R \]

- Expected duration of one inter-checkpoint interval (case 2):
  \[ E_F = (T_C + T_{OV}) + \frac{T_C + T_{OV}}{2} + T_R = \frac{3}{2}(T_C + T_{OV}) + T_R \]
Analytical Model (IV)

- Expected duration of one interval between two checkpoints:

\[
E_{INT} = P_{NF}E_{NF} + P_FE_F
\]

\[
= e^{-\lambda(T_C+T_{OV})}(T_C + T_{OV}) + (1 - e^{-\lambda(T_C+T_{OV})}) \left( \frac{3}{2}(T_C + T_{OV}) + T_R \right)
\]

\[
= \frac{3}{2}(T_C + T_{OV}) + T_R - e^{-\lambda(T_C+T_{OV})} \left( \frac{1}{2}(T_C + T_{OV}) + T_R \right)
\]
Example

- $T_C = 10$, $T_{OV} = T_R = 1$

Time units

Expected duration of intercheckpoint interval

MTTF
Extensions of Time Redundancy

Tolerating permanent faults

- If an error occurs, restore the closest checkpoint in the past
- Use a different algorithm or different structures for recomputation

Minimize hardware costs

- Perform error-detection by repeated computation
  - With different algorithms
  - With different structures
N-Version Programming

- Similar to the NMR scheme used against hardware faults
- Standard NMR does not provide protection against design faults
  - Each of the N copies is of identical design
- In N-version programming:
  N independently written programs for a given function are run simultaneously or in sequence
- Results are obtained by voting upon the outputs of the individual implementations
7 Time Redundancy

7.1 Transistor and Gate Level
7.2 Gate Level and RT-Level
7.3 Finite State Machine Level
7.4 Processor Level
7.5 Software Level
Self-Testing Multiplexer-Based Circuits

Basic idea:

- Implement Boolean function as irredundant network of multiplexers
  - Input variables become \textit{SELECT} inputs of multiplexers
  - Synthesis of multiplexer network by ROBDDs (Reduced Ordered Binary Decision Diagrams)

- Fault-free circuit: Inversion of the inputs of the multiplexers results in inversion of the output
  - For stuck-at faults, the output does not change

- Perform original and inverted computation in consecutive time steps
Based on **Shannon expansion**:

\[
F(x_1,\ldots,x_n) = \overline{x}_i \cdot F\bigg|_{x_i=0} + x_i \cdot F\bigg|_{x_i=1}
\]

\[F\bigg|_{x_i=1}\text{: Co-factor}\]

- Shannon expansion is performed recursively on the sub-terms for every variable
Example (1)

- Multiplexer-based implementation of

\[ y = c + abc \]
Example (2)

- Example of sensitized path in a Shannon circuit
Time Redundancy: Transistor/Gate Level


- Idea: Complement the function by complementing the inputs of the MUXes

![Diagram of MUXes with time delays](image)
Multiplexer Faults

We have three different categories of multiplexer faults:

1. Stuck-at-0/1 on the output
   ➔ All paths can be sensitized, hence these faults are testable

2. Stuck-at-0/1 on the selector input
   ROBDD: All Multiplexers are irredundant
   ➔ Multiplexer selector is controllable, hence these faults are testable

3. Stuck-at-0/1 on the data inputs
   ➔ See 1.

➔ The circuit is self-testing
CMOS Implementations

Hardware-based Fault Tolerance
7 Time Redundancy

7.1 Transistor and Gate Level
7.2 Gate Level and RT-Level
7.3 Finite State Machine Level
7.4 Processor Level
7.5 Software Level
Gate Level and RT-Level

Idea:
Duplicate content of a register in one or more shadow registers

- Perform computation twice, by restoring from shadow register
  - Requires two shadow registers
  - Allows to detect soft-errors of arbitrary length

- Delay sampling with shadow register by small $t_d < t_{clk}$
  - Only one shadow register required
  - Allows to detect soft-errors and delay faults with length $t_d$
Time Redundancy: RAZOR Error Detection

- Sample result of computation at $t = t_{clk}$ and $t = t_{clk} + t_d$
- $t_d < t_{clk}$
- Can detect delay and transient faults
- Checkpointing at the Processor Level for Recovery
Processor Roll-back Using RAZOR

- Upon error detection by RAZOR, locally recover and flush pipeline

- Builds on existing branch prediction framework
  - To flush instruction pipeline

- Multiple cycle penalty for timing failure

- Scalable design as all communication is local
7 Time Redundancy

7.1 Transistor and Gate Level
7.2 Gate Level and RT-Level
7.3 Finite State Machine Level
7.4 Processor Level
7.5 Software Level
Control-Flow Monitoring Using Signatures

- Employ a Watchdog (a simple FSM) to monitor behavior of a functional unit

- Signature Monitoring
  - Pre-computed signature embedded in the Watchdog FSM
Use control flow signatures for fault tolerance in FSMs

The signature model:

- A basic block of states $S_i \subseteq S$ is assigned a unique signature $ID_i$.
- Invariant:
  Global register GR must contain ID of the current block at exit
- Difference value for incoming edge $(j,i)$ where $j$ is the parent node for $i$, $D_{j,i} = ID_j \oplus ID_i$
- Check for the expected signature $ID_i$ at node $i$
Example

- Suppose control flow passes through edge \((a, b)\)
  
  At block \(a\), \(GR = ID_a\)

- At block \(b\), we need to check:
  \[
  GR = GR \oplus D_{a,b}
  \]

  \[
  if \ (GR \neq ID_b) \ then \{ \text{error} \} \]

- If error is detected, restore previous state from shadow register
Synthesized Assertions

- Define assertions based on temporal logic (e.g. property specification language, PSL):
  - always
  - never
  - next
  - eventually!
  - until
  - Before

- Synthesize these assertion into monitoring FSM

- Manual process, but assertions may be derived from design verification process
Example

- always !\((a \& b)\)
- never \((a \& b)\)
- always \((req \rightarrow \text{next gnt})\):
  Every request is followed by a grant in the next cycle
7 Time Redundancy

7.1 Transistor and Gate Level
7.2 Gate Level and RT-Level
7.3 Finite State Machine Level
7.4 Processor Level
7.5 Software Level
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

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

- Checkpointing overhead consists of storing processor's registers in main memory
CARER: Checkpoint Bit For Each Cache Line

- Scheme requires hardware modification: 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 and Roll Back

- 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

- Taking a checkpoint involves:
  1. Saving processor registers in memory
  2. Setting the checkpoint bit associated with each valid cache line to 1

- Rolling back to previous checkpoint:
  Restore registers, and invalidate all cache lines with checkpoint bit = 0
7 Time Redundancy

7.1 Transistor and Gate Level
7.2 Gate Level and RT-Level
7.3 Finite State Machine Level
7.4 Processor Level
7.5 Software Level
Control-flow Monitoring Using Signatures: Software Approaches

- Partition the application into blocks, either in the assembly language or in the high-level language
- Instrument program by inserting checking code at the beginning and/or the end of the blocks
  - Eliminates the need for a hardware watchdog unit
- Two classes of approaches
  - non-preemptive signature checking
  - preemptive signature checking
Checkpointing at the OS Kernel Level

- Transparent to user; no changes to program
- Kernel takes checkpoints when process preempted
- Kernel responsible for managing recovery operation
  - Restore process space
  - Restore file-handles, communication sockets, etc.
- Process state is recorded so that execution can resume without loss of computational work
  - Memory subsystems require a form of “copy-on-write”
  - Distributed systems require synchronized checkpointing
Checkpointing at the User Level

- Application responsible for all checkpointing functions
- Code for checkpointing & recovery part of application

Checkpointing can also be provided by a library:

- A user-level library provided for checkpointing
- Like kernel-level checkpointing, this approach generally requires no changes to application code
- Library also manages recovery from failure