FAULT TOLERANT SYSTEMS

http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems

Part 14 - Software Fault Tolerance I
Chapter 4 – Software Fault Tolerance

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Causes of Software Errors

- Designing and writing software is very difficult - essential and accidental causes of software errors
- Essential difficulties
  - Understanding a complex application and operating environment
  - Constructing a structure comprising an extremely large number of states, with very complex state-transition rules
  - Software is subject to frequent modifications - new features are added to adapt to changing application needs
  - Hardware and operating system platforms can change with time - the software has to adjust appropriately
  - Software is often used to paper over incompatibilities between interacting system components
- Accidental difficulties - Human mistakes
- Cost considerations - use of Commercial Off-the-Shelf (COTS) software - not designed for high-reliability applications

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Techniques to Reduce Error Rate

- Software almost inevitably contains defects/bugs
  - Do everything possible to reduce the fault rate
  - Use fault-tolerance techniques to deal with software faults
- Formal proof that the software is correct - not practical for large pieces of software
- Acceptance tests - used in wrappers and in recovery blocks - important fault-tolerant mechanisms
- Example: If a thermometer reads -40°C on a midsummer day - suspect malfunction
- Timing Checks: Set a watchdog timer to the expected run time ; if timer goes off, assume a hardware or software failure
  - can be used in parallel with other acceptance tests

Part.14.3

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Acceptance tests

- Verification of Output:
- Sometimes, acceptance test suggested naturally
  - Sorting; Square root; Factorization of large numbers; Solution of equations
- Probabilistic checks:
- Example: multiply $n \times n$ integer matrices $C = A \times B$
- The naive approach takes $O(n^3)$ time
- Instead - pick at random an n-element vector of integers, R
- $M_1 = A \times (B \times R)$ and $M_2 = C \times R$
- If $M_1 \neq M_2$ - an error has occurred
- If $M_1 = M_2$ - high probability of correctness
- May repeat by picking another vector
- Complexity - $O(m\, n^2)$; m is number of checks

Part.14.4

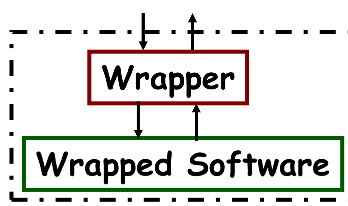
Copyright 2007 Koren & Krishna, Morgan-Kaufman

Range Checks

- Set acceptable bounds for output
  - if output outside bounds - declare a fault
- Bounds - either preset or simple function of inputs
  - probability of faulty test software should be low
- Example: remote-sensing satellite taking thermal imagery of earth
  - Bounds on temperature range
  - Bounds on spatial differences - excessive differences between temperature in adjacent areas indicate failure
- Every test must balance sensitivity and specificity
- Sensitivity - conditional probability that test fails, given output is erroneous
- Specificity - conditional probability that it is indeed an error given acceptance test flags an error
- Narrower bounds - increase sensitivity but also increase false-alarm rate and decrease specificity

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Single Version Fault Tolerance – Wrappers



- Robustness-enhancing interfaces for software modules
- Examples: operating system kernel, middleware, applications software
- Inputs are intercepted by the wrapper, which either passes them or signals an exception
- Similarly, outputs are filtered by the wrapper
- Example: using COTS software for high-reliability applications
- COTS components are wrapped to reduce their failure rate - prevent inputs
  - (1) outside specified range or
  - (2) known to cause failures
- Outputs pass a similar acceptance test

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Example 1: Dealing with Buffer Overflow

- C language does not perform range checking for arrays - can cause accidental or malicious damage
- Write a large string into a small buffer: buffer overflow - memory outside buffer is overwritten
- If accidental – can cause a memory fault
- If malicious - overwriting portions of program stack or heap - a well-known hacking technique
- Stack-smashing attack:
  * A process with root privileges stores its return address in stack
  * Malicious program overwrites this return address
  * Control flow is redirected to a memory location where the hacker stored the attacking code
  * Attacking code now has root privileges and can destroy the system

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Wrapper to Protect against Buffer Overflow

- All malloc calls from the wrapped program are intercepted by wrapper
- Wrapper keeps track of the starting position of allocated memory and size
- Writes are intercepted, to verify that they fall within allocated bounds
- If not, wrapper does not allow the write to proceed and instead flags an overflow error

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Example 2: Checking correctness of scheduler

- Wrapper around task scheduler in a fault-tolerant, real-time system
- Such schedulers may use Earliest Deadline First (EDF) - execute task with earliest deadline among tasks ready to run
  - Subject to preemptability constraints (tasks in certain parts of execution may not be preemptable)
- A wrapper verifies correctness of scheduling algorithm:
  - The ready task with earliest deadline was picked
  - Any arriving task with an earlier deadline preempts the executing task (if latter is preemptable)
- Wrapper needs information about the tasks, their deadlines, and their preemptability

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Example 3: Using software with known bugs

- Found through testing or field reports that software fails for a certain set of inputs, S
- Wrapper intercepts inputs and checks if in set S
- If not, forward to software module for execution
- If yes, return a suitable exception to system
- Alternatively, redirect input to some alternative, custom-written, code that handles these inputs

Example 4: Checking for correct output

- Wrapper includes an acceptance test to filter output
- If the output passes test, it is forwarded outside
- If not, exception is raised, and system has to deal with a suspicious output

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Factors in Successful Wrapping

- **Quality of acceptance tests:**
  - Application-dependent - has direct impact on ability of wrapper to stop faulty outputs
- **Availability of necessary information from wrapped component:**
  - If wrapped component is a "black box," (observes only the response to given input), wrapper will be somewhat limited
  - Example: a scheduler wrapper is impossible without information about status of tasks waiting to run
- **Extent to which wrapped software module has been tested:**
  - Extensive testing identifies inputs for which the software fails

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Single Version Fault Tolerance: Software Rejuvenation

- Example: Rebooting a PC
- As a process executes
  - it acquires memory and file-locks without properly releasing them
  - memory space tends to become increasingly fragmented
- The process can become faulty and stop executing
- To head this off, proactively halt the process, clean up its internal state, and then restart it
- Rejuvenation can be time-based or prediction-based
- Time-Based Rejuvenation - periodically
- Rejuvenation period - balance benefits against cost

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Time-Based Rejuvenation - Analytical Model

- $\tilde{N}(t)$ - expected number of errors in interval [0,t]
- C_e - Cost of failure
- C_r - Cost of each rejuvenation
- P - Rejuvenation period
- Adding costs due to rejuvenation and failure - overall expected cost of rejuvenation over one rejuvenation period

$$C_{rejuv}(P) = \tilde{N}(P)C_e + C_r$$

- Rate of rejuvenation cost

$$C_{rate}(P) = \frac{C_{rejuv}(P)}{P} = \frac{\tilde{N}(P)C_e + C_r}{P}$$

Part.14.13

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Analytical Model - Examples

- Constant failure rate over time - $\overline{N}(P) = \lambda P$

$$C_{rate}(P) = \lambda C_e + C_r/P$$

  - Optimal P is P* = ∞ - no software rejuvenation
  - Rejuvenation heads off the increased rate in failure as software ages - no aging in this case

- If $\tilde{N}(P) = \lambda P^n,\ n > 1$

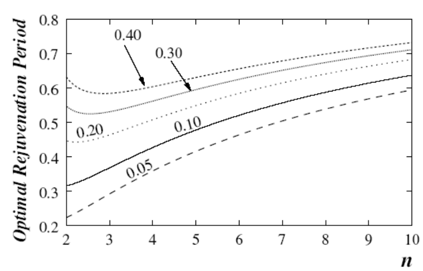
$$C_{rate}(P) = \lambda P^{n-1} C_e + C_r/P$$

  - Optimal P:

$$P^* = \left(\frac{C_r}{(n-1)\lambda C_e}\right)^{1/n}$$

- If n=2:

$$P^* = \sqrt{\frac{C_r}{\lambda C_e}}$$

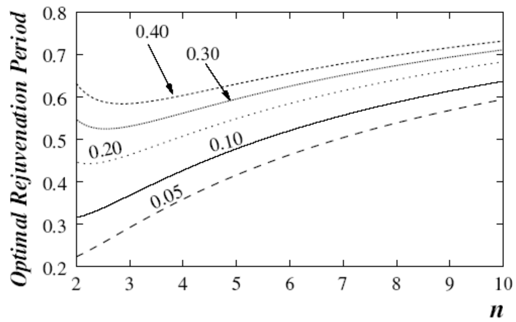


$\lambda = 1$; Time units are arbitrary;
Curve labels indicate C_r/C_e.

Part.14.14

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Optimal Rejuvenation Period



$\lambda = 1$; Time units are arbitrary;
Curve labels indicate C_r/C_e.

- Estimating the parameters C_e, C_r, $\tilde{N}(t)$ -
  - Can be done experimentally - running simulations on the software
  - System can be made adaptive - some default initial values and adjusting the values as more statistics are gathered

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Prediction-Based Rejuvenation

- Monitoring system characteristics - amount of memory allocated, number of file locks held, etc. - predicting when system will fail
- Example - a process consumes memory at a certain rate, the system estimates when it will run out of memory, rejuvenation can take place just before predicted crash
- The software that implements prediction-based rejuvenation must have access to enough state information to make such predictions
- If prediction software is part of operating system - such information is easy to collect
- If it is a package that runs atop operating system with no special privileges - constrained to using interfaces provided by OS

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Example - Unix Operating System

- Unix provides the following utilities for collecting status information -
- **vmstat** - provides information about processor utilization, memory and paging activity, traps, and I/O
- **iostat** - outputs percentage CPU utilization at user and system levels, as well as a report on usage of each I/O device
- **netstat** - indicates network connections, routing tables and a table of all network interfaces
- **nfsstat** - provides information about network file server kernel statistics

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Least Squares Failure Time Prediction

- Once status information is collected - trends must be identified and failure time predicted
- Example: tracking allocation of memory to a process
- Points $\mu(t_1), \mu(t_2), \cdots, \mu(t_k)$ are given - $\mu(t_i)$ is the allocated memory at time t_i ($t_1 < t_2 < \cdots < t_k$)
- We can do a least square fit of a polynomial

$$f(t) = m_n t^n + m_{n-1} t^{n-1} + \cdots + m_1 t + m_0$$

so that $\sum_{i=1}^{k} [\mu(t_i) - f(t_i)]^2$ is minimized

- A straight line **f(t)=mt+c** is the simplest such fit
- This polynomial can be used to extrapolate into the future and predict when the process will run out of memory

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Weighted Least Squares

- In standard least-squares fit, each observed point $\mu(t_i)$ has equal weight in determining the fit
- **Variation**: select weights $w_1, w_2, \ldots, w_k$ - then find coefficients of f(t) to minimize

$$\sum_{i=1}^{k} w_i \left[\mu(t_i) - f(t_i)\right]^2$$

- Weights allow greater emphasis to certain points
- **Example:** $w_1 < w_2 < \cdots < w_k$ - more recent data influences the fit more than older data
- Curve-fitting vulnerable to impact of a few unusually high or low points - distorting fit and prediction
- Techniques are available to make the fit more robust by reducing the impact of these points

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Combined Approach

- Prediction-based rejuvenation with a timer reset on rejuvenation
- If timer goes off - rejuvenation is done regardless of when next failure is predicted to happen

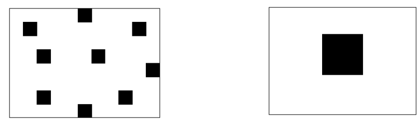
Rejuvenation Level

- Either application or node level - depending on where resources have degraded or become exhausted
- Rejuvenation at the application level - suspending an individual application, cleaning up its state (by garbage collection, re-initialization of data structures, etc.), and then restarting
- Rejuvenation at the node level - rebooting node - affects all applications running on that node

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Single Version Fault Tolerance: Data Diversity

- **Input space** of a program can be divided into fault and non-fault regions - program fails if and only if an input from the fault region is applied
- Consider an unrealistic input space of 2 dimensions
- In both cases - Fault regions occupy a third of input area



- Perturb input slightly - new input may fall in a non-faulty region
- **Data diversity:**
  - One copy of software: use acceptance test -recompute with perturbed inputs and recheck output
  - Massive redundancy: apply slightly different input sets to different versions and vote

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Explicit vs. Implicit Perturbation

- **Explicit** - add a small deviation term to a selected subset of inputs
- **Implicit** - gather inputs to program such that we can expect them to be slightly different
- **Example 1**: software control of industrial process - inputs are pressure and temperature of boiler
- Every second - (p_i, t_i) measured - input to controller
- Measurement in time i not much different from $i-1$
- **Implicit perturbation** may consist of using (p_{i-1}, t_{i-1}) as an alternative to (p_i, t_i)
- If (p_i, t_i) is in fault region - (p_{i-1}, t_{i-1}) may not be

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Explicit Perturbation - Reorder Inputs

- Example 2: add floating-point numbers a,b,c - compute a+b, and then add c
- a=2.2E+20, b=5, c=-2.2E+20
- Depending on precision used, a+b may be 2.2E+20 resulting in a+b+c=0
- Change order of inputs to a,c,b - then a+c=0 and a+c+b=5
- Example 2 - an example of exact re-expression
  - output can be used as is (if passes acceptance test or vote)
- Example 1 – an example of inexact re-expression - likely to have $f(p_i, t_i) \neq f(p_{i-1}, t_{i-1})$
  - Use raw output as a degraded but acceptable alternative, or attempt to correct before use, e.g., Taylor expansion

$$f(t) = f(t_0) + \sum_{n=0}^{\infty} \frac{(t - t_0)^n f^{(n)}(t_0)}{n!}$$

Part.14.23

Copyright 2007 Koren & Krishna, Morgan-Kaufman

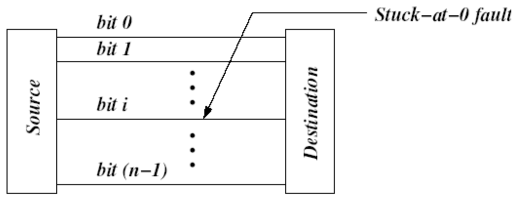
Software Implemented Hardware Fault Tolerance (SIHFT)

- Data diversity combined with time redundancy for Software Implemented Hardware Fault Tolerance (SIHFT)
- Can deal with permanent hardware failures
- Each input multiplied by a constant, k, and a program is constructed so that output is multiplied by k
- If it is not – a hardware error is detected
- Finding an appropriate value of k:
  - Ensure that it is possible to find suitable data types so that arithmetic overflow or underflow does not happen
  - Select k such that it is able to detect a large fraction of hardware faults - experimental studies by injecting faults

Part.14.24

Copyright 2007 Koren & Krishna, Morgan-Kaufman

SIHFT - Example



- n-bit bus
- Bit i stuck-at-0
- If data sent has ith bit=1 – error
- Transformed program with k=2 executed on same hardware - ith bit will use line (i+1) of bus - not affected by fault
- The two programs will yield different results - indicating the presence of a fault
- If both bits i and (i-1) of data are 0 – fault not detected - probability of 0.25 under uniform probability assumption
- If k=-1 is used (every variable and constant in program undergoes a two's complement operation) - almost all 0s in original program will turn into 1s - small probability of an undetected fault

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Overflow

- Risk of overflow exists even for small values of k
- Even k=-1 can generate an overflow if original variable is equal to the largest negative integer that can be represented using two's complement (for a 32-bit integer this is -2^{31})
- Possible precautions:
  - Scaling up the type of integer used for that variable.
  - Performing range analysis to determine which variables must be scaled up to avoid overflows

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Example – Program Transformation for k=2

- Result divided by k to ensure proper transformation of output

```
i = 0;
x = 3;
y = 1;
while (i < 5) {
    y = y * (x + i);
    i = i + 2;
}
z = y;
```

(a) The original program

```
i = 0;
x = 6;
y = 2;
while (i < 10) {
    y = y * (x + i)/2;
    i = i + 4;
}
z := y;
```

(b) The transformed program

Part.14.27

Copyright 2007 Koren & Krishna, Morgan-Kaufman

Floating-Point Variables

- Some simple choices for k no longer adequate
- Multiplying by k=-1 - only the sign bit will change (assuming the IEEE standard representation of floating-point numbers)
- Multiplying by $k = 2^l$ - only exponent field will change
- Both significand and exponent field must be multiplied, possibly by two different values of k
- To select value(s) of k such that SIHFT will detect a large fraction of hardware faults – either simulation or fault-injection studies of the program must be performed for each k

Part.14.28

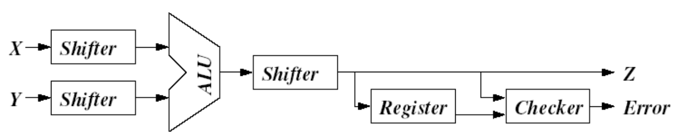
Copyright 2007 Koren & Krishna, Morgan-Kaufman

Recomputing with Shifted Operands (RESO)

- Similar to SIHFT - but hardware is modified
- Each unit that executes either an arithmetic or a logic operation is modified
- It first executes operation on original operands and then re-executes same operation on transformed operands
- Same issues that exist for SIHFT exist for RESO
- Transformations of operands are limited to simple shifts which correspond to $k = 2^l$ with an integer l
- Avoiding an overflow is easier for RESO – the datapath can be extended to include extra bits

Copyright 2007 Koren & Krishna, Morgan-Kaufman

RESO - Example



- An ALU modified to support the RESO technique
- Example – addition
- First step: The two original operands X and Y are added and the result Z stored in register
- Second step: The two operands are shifted by l bit positions and then added
- Third step: The result of second addition is shifted by same number of bit positions, but in opposite direction, and compared with contents of register, using checker circuit

Copyright 2007 Koren & Krishna, Morgan-Kaufman