Chapter 5 – Software Fault Tolerance

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

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

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 × n integer matrices C = A × B
    ∗ The naive approach takes O(n³) time
    ∗ Instead – pick at random an n-element vector of integers, R
    ∗ M1 = A × (B × R) and M2 = C × R
    ∗ If M1 ≠ M2 – an error has occurred
    ∗ If M1 = M2 – high probability of correctness
    ∗ May repeat by picking another vector
  ∗ Complexity - O(m n²); m is number of checks

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 by also increase false-alarm rate and decrease specificity

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
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

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

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

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

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

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
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

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 temperate of boiler
- Every second - \((p_i, t_i)\) measured - input to controller
- Implicit perturbation may consist of using \((p_{i-1}, t_{i-1})\) as an alternative to \((p_i, t_i)\)
- Explicit Perturbation - Reorder Inputs
- Example 2: add floating-point numbers \(a, b, c\) - compute \(a+b\), and then add \(c\)
  - \(a=2.2\times 10^{20}, b=5, c=-2.2\times 10^{20}\)
  - Depending on precision used, \(a+b\) may be \(2.2\times 10^{20}\) resulting in \(a+c=0\) 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
  - \(t[i] = f(t_0) + \sum_{i=0}^{n} \frac{(t[i-1] - f(t_0))}{(1)}\)

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 mask a large fraction of hardware faults - experimental studies by injecting faults

SIHFT - Example

- \(n\)-bit bus
- Bit \(i\) stuck-at-0
- If data sent has \(i\)th bit=1 - error
- Transformed program with \(k=2\) executed on same hardware - \(i\)th 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 Os in original program will turn into 1s - small probability of an undetected fault

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
Example – Program Transformation for k=2

• Result divided by k to ensure proper transformation of output

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

d = y;
```

(a) The original program

```c
#include <stdio.h>

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

(b) The transformed program

Floating-Point Variables

• Some simple choices for k no longer adequate
• Multiplying by k = -1 – only the sign bit will change
• Multiplying by k = 2^i – 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

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
• Some issues that exist for SIHFT exist for RESO
• Transformations of operands are limited to simple shifts which correspond to k = 2^i
• Avoiding an overflow is easier for RESO – the datapath can be extended to include extra bits

N-Version Programming

• N independent teams of programmers develop software to same specifications – N versions are run in parallel – output voted on
• If programs are developed independently – very unlikely that they will fail on same inputs
• Assumption – failures are statistically independent: probability of failure of an individual version = q
• Probability of no more than m failures out of N versions –

\[ p_{\text{ind}}(N, m, q) = \sum_{i=0}^{m} \binom{N}{i} q^i (1 - q)^{N-i} \]

Consistent Comparison Problem

• N-version programming is not simple to implement
• Even if all versions are correct – reaching a consensus is difficult
• Example:
  - V1, ..., VN – independently written versions for computing a quantity X and comparing it to some constant C
  - Xi – value of X computed by version Vi (i=1,...,N)
• The comparison with C is said to be consistent if either all Xi < c or all Xi ≥ c
Consistency Requirement

- Example:
  - A function of pressure and temperature, \( f(p, t) \), is calculated
  - Action A1 is taken if \( f(p, t) < C \)
  - Action A2 is taken if \( f(p, t) > C \)
  - Each version outputs action to be taken
  - Ideally all versions consistent - output same action
  - Versions are written independently - use different algorithms to compute \( f(p, t) \) - values will differ slightly
  - Example: \( C = 1.0000 \); \( N = 3 \)
  - All three versions operate correctly - output values:
    - \( X_1 \), \( X_2 \) < \( C \) - recommended action is A1
    - \( X_3 > C \) - recommended action is A2
  - Not consistent although all versions are correct

Consensus Comparison Problem

- If versions don't agree - they may be faulty or not
- Multiple failed versions can produce identical wrong outputs due to correlated fault - system will select wrong output
- Can bypass the problem by having versions decide on a consensus value of the variable
- Before checking if \( X \geq C \), the versions agree on a value of \( X \) to use
- This adds the requirement: specify order of comparisons for multiple comparisons
- Can reduce version diversity, increasing potential for correlated failures
- Can also degrade performance - versions that complete early would have to wait

Independent vs. Correlated Versions

- Correlated failures between versions can increase overall failure probability by orders of magnitude
- Example: \( N = 3 \), can tolerate up to one failed version for any input: \( q = 0.0001 \) - an incorrect output once every ten thousand runs
- If versions stochastically independent - failure probability of 3-version system
  \[ q^3 + 3q^2(1 - q) \approx 3 \times 10^{-8} \]
- Suppose versions are statistically dependent and there is one fault, causing system failure, common to two versions, exercised once every million runs
- Failure probability of 3-version system increases to over \( 10^{-5} \), more than 30 times the failure probability of uncorrelated system

Consistency Problem

- Theorem: Any algorithm which guarantees that any two \( n \)-bit integers which differ by less than \( 2^k \) will be mapped to the same \( m \)-bit output, \( m+k \leq n \), must be the trivial algorithm that maps every input to the same number
- Proof:
  - We start with \( k=1 \)
  - 0 and 1 differ by less than \( 2^1 \)
  - The algorithm will map both to the same number, say \( a \)
  - Similarly, 1 and 2 differ by less than \( 2^1 \) so they will also be mapped to \( a \)
  - Proceeding, we can show that 3, 4,... will all be mapped by this algorithm to \( a \)
  - Therefore this is the trivial algorithm that maps all integers to the same number, \( a \)
  - Exercise: Show that a similar result holds for real numbers that differ even slightly from one another

Another Approach - Confidence Signals

- Each version calculates \( |X-C| \); if \( |X-C| < \delta \) for some given \( \delta \), version announces low confidence in its output
- Voter gives lower weights to low confidence versions
- Problem: if a functional version has \( |X-C| < \delta \), high chance that this will also be true of other versions, whose outputs will be devalued by voter
- The frequency of this problem arising, and length of time it lasts, depend on nature of application
- In applications where calculation depends only on latest inputs and not on past values - consensus problem may occur infrequently and go away quickly

Version Correlation Model

- Input space divided to regions: different probability of input from region to cause a version to fail
- Example: Algorithm may have numerical instability in an input subspace - failure rate greater than average
- Assumption: Versions are stochastically independent in each given subspace Si
  - \( \text{Prob\{V1 fails | input from Si\}} = \text{Prob\{V1 fails\}} \times \text{Prob\{input from Si\}} \)
  - Unconditional probability of failure of a version
    \[ \text{Unconditional probability of failure of a version = \sum_i \text{Prob\{V1 fails | input from Si\}} \times \text{Prob\{input from Si\}} \}
  - Unconditional probability that both fail
    \[ \text{Unconditional probability that both fail = \sum_i \text{Prob\{V1 and V2 fail | input from Si\}} \times \text{Prob\{input from Si\}} \}
    \[ = \text{Prob\{V1 fails\}} \times \text{Prob\{V2 fails\}} \]
Version Correlation: Example 1

- Two input subspaces S1, S2 - probability 0.5 each
- Conditional failure probabilities:
  - Version S1 S2
  - V1 0.01 0.001
  - V2 0.02 0.003
- Unconditional failure probabilities:
  - P(V1 fails) = 0.01 * 0.5 + 0.001 * 0.5 = 0.0055
  - P(V2 fails) = 0.02 * 0.5 + 0.003 * 0.5 = 0.0115

If versions were independent, probability of both failing for same input = 0.0055 * 0.0115

The two versions are positively correlated: both are more prone to failure in S1 than in S2

Version Correlation: Example 2

- Conditional failure probabilities:
  - Version S1 S2
  - V1 0.010 0.001
  - V2 0.003 0.020
- Unconditional failure probabilities - same as Example 1
- Joint failure probability - P(V1 & V2 fail) = 0.01 * 0.02 * 0.5 + 0.001 * 0.003 * 0.5 = 0.000025

Much less than the previous joint probability or the product of individual probabilities

Tendencies to failure are negatively correlated:

- V1 is better in S1 than in S2, opposite for V2 - V1 and V2 make up for each other’s deficiencies
- Ideally - multiple versions negatively correlated
- In practice - positive correlation - since versions are solving the same problem

Causes of Version Correlation

- Common specifications - errors in specifications will propagate to software
- Intrinsic difficulty of problem - algorithms may be more difficult to implement for some inputs, causing faults triggered by same inputs
- Common algorithms - algorithm itself may contain instabilities in certain regions of input space - different versions have instabilities in same region
- Cultural factors - Programmers make similar mistakes in interpreting ambiguous specifications
- Common software and hardware platforms - if same hardware, operating system, and compiler are used - their faults can trigger a correlated failure

Achieving Version Independence - Incidental Diversity

- Forcing developers of different modules to work independently of one another
- Teams working on different modules are forbidden to directly communicate
- Questions regarding ambiguities in specifications or any other issue have to be addressed to some central authority who makes any necessary corrections and updates all teams
- Inspection of software carefully coordinated so that inspectors of one version do not leak information about another version

Achieving Version Independence - Methods for Forced Diversity

- Diverse specifications
- Diverse hardware and operating systems
- Diverse development tools and compilers
- Diverse programming languages
- Versions with differing capabilities

Diverse Specifications

- Most software failures due to requirements specification
- Diversity can begin at specification stage - specifications may be expressed in different formalisms
- Specification errors will not coincide across versions - each specification will trigger a different implementation fault profile

Diverse Hardware and Operating Systems

- Output depends on interaction between application software and its platform - OS and processor
- Both processors and operating systems are notorious for the bugs they contain
- A good idea to complement software design diversity with hardware and OS diversity - running each version on a different processor type and OS

Diverse Development Tools and Compilers

- May make possible “notational diversity” reducing extent of positive correlation between failures
- Diverse tools and compilers (may be faulty) for different versions may allow for greater reliability
Diverse Programming Languages
♦ Programming language affects software quality
♦ Examples:
  ∗ Assembler - more error-prone than a higher-level language
  ∗ Nature of errors different - in C programs - easy to overflow allocated memory - impossible in a language that strictly manages memory
  ∗ No faulty use of pointers in Fortran - has no pointers
  ∗ Lisp is a more natural language for some artificial intelligence (AI) algorithms than are C or Fortran
♦ Diverse programming languages may have diverse libraries and compilers - will have uncorrelated (or even better, negatively-correlated) failures

Choice of Programming Language
♦ Should all versions use best language for problem or some versions be in other less suited languages?
  ∗ If same language - lower individual fault rate but positively correlated failures
  ∗ If different languages - individual fault rates may be greater, but overall failure rate of N-version system may be smaller if less correlated failures
  ∗ Tradeoff difficult to resolve - no analytical model exists - extensive experimental work is necessary

Versions With Differing Capabilities
♦ Example: One rudimentary version providing less accurate but still acceptable output
  ∗ 2nd simpler, less fault-prone and more robust
  ∗ If the two do not agree - a 3rd version can help determine which is correct
  ∗ If 3rd very simple, formal methods may be used to prove correctness

Back-to-Back Testing
♦ Comparing intermediate variables or outputs for same input - identify non-coincident faults

Single Version vs. N Versions
♦ Assumption: developing N versions - N times as expensive as developing a single version
  ∗ Some parts of development process may be common, e.g. - if all versions use same specifications, only one set needs to be developed
  ∗ Management of an N-version project imposes additional overheads
  ∗ Costs can be reduced - identify most critical portions of code and only develop versions for these
  ∗ Given a total time and money budget - two choices:
    ∗ (a) develop a single version using the entire budget
    ∗ (b) develop N versions
  ∗ No good model exists to choose between the two

Experimental Results
♦ Few experimental studies of effectiveness of N-version programming
♦ Published results only for work in universities
  ∗ One study at the Universities of Virginia and California at Irvine
    ∗ 27 students wrote code for anti-missile application
    ∗ Some had no prior industrial experience while others over ten years
    ∗ All versions written in Pascal
    ∗ 93 correlated faults identified by standard statistical hypothesis-testing methods: if versions had been stochastically independent, we would expect no more than 5
    ∗ No correlation observed between quality of programs produced and experience of programmer

Recovery Block Approach
♦ N versions, one running - if it fails, execution is switched to a backup
  ∗ Example - primary + 3 secondary versions
  ∗ Primary executed - output passed to acceptance test
  ∗ If output is not accepted - system state is rolled back and secondary 1 starts, and so on
  ∗ If all fail - computation fails
  ∗ Success of recovery block approach depends on failure independence of different versions and quality of acceptance test
Distributed Recovery Blocks

- Two nodes carry identical copies of primary and secondary
- Node 1 executes the primary - in parallel, node 2 executes the secondary
- If node 1 fails the acceptance test, output of node 2 is used (provided that it passes the test)
- Output of node 2 can also be used if node 1 fails to produce an output within a prespecified time

Distributed Recovery Blocks - cont.

- Once primary fails, roles of primary and secondary are reversed
- Node 2 continues to execute the secondary copy, which is now treated as primary
- Execution by node 1 of primary is used as a backup
- This continues until execution by node 2 is flagged erroneous, then system toggles back to using execution by node 2 as a backup
- Rollback is not necessary - saves time - useful for real-time system with tight task deadlines
- Scheme can be extended to N versions (primary plus N-1 secondaries run in parallel on N processors)

Exception Handling

- Exception - something happened during execution that needs attention
- Control transferred to exception-handler-routine
- Example: y = a * b, if overflow - signal an exception
- Effective exception-handling can make a significant improvement to system fault tolerance
- Over half of code lines in many programs devoted to exception-handling
- Exceptions deal with
  - (a) domain or range failure
  - (b) out-of-ordinary event (not failure) needing special attention
  - (c) timing failure

Domain and Range Failure

- Domain failure - illegal input is used
- Example: if X, Y are real numbers and \( Y = \sqrt{X} \) is attempted with \( Y = -1 \), a domain failure occurs
- Range failure - program produces an output or carries out an operation that is seen to be incorrect in some way
- Examples include:
  - Encountering an end-of-file while reading data from file
  - Producing a result that violates an acceptance test
  - Trying to print a line that is too long
  - Generating an arithmetic overflow or underflow

Out-of-the-Ordinary Events

- Exceptions can be used to ensure special handling of rare, but perfectly normal, events
- Example - Reading the last item of a list from a file - may trigger an exception to notify invoker that this was the last item
- Timing Failures:
  - In real-time applications, tasks have deadlines
  - If deadlines are violated - can trigger an exception
  - Exception-handler decides what to do in response: for example - may switch to a backup routine

Requirements of Exception-Handlers

- (1) Should be easy to program and use
- Be modular and separable from rest of software
- Not be mixed with other lines of code in a routine - would be hard to understand, debug, and modify
- (2) Exception-handling should not impose a substantial overhead on normal functioning of system
- Exceptions be invoked only in exceptional circumstances
- Exception-handling not inflict a burden in the usual case with no exception conditions
- (3) Exception-handling must not compromise system state - not render it inconsistent
Software Reliability Models

- Software is often the major cause of system unreliability - accurately predicting software reliability is very important.
- Relatively young and often controversial area.
- Many analytical models, some with contradictory results.
- Not enough evidence to select the correct model.
- Although models attempt to provide numerical reliability, they should be used mainly for determining software quality.