SW Dependability Methods

✧ Why **software** dependability methods?
✧ **Static SW** dependability methods
✧ **Worst Case Execution Analyse**
✧ **How does cache effect WCEA**
Software implements System functionality

- Software is playing an increasingly important role in system functionality.
- An exponential increase in On-Board software functionality.
- Increase in software complexity.
- Amount of software on-board increases, from few kbyte in early 80th to many Mbytes today.

- SOHO, 1995 2*64 KB
- Rosetta, 2003, 2*1MB
- ATV, 2006, 8MB
Software implements a large part of space systems functionality
- the System Dependability and Safety approach needs to be supported through correspondent **Software Dependability and Safety methods**
- **Software Dependability and Safety requirements** need to be derived from system Dependability and Safety recommendations

**System functional** Dependability and Safety needs to be specified through functional software requirements.
- Software Dependability and Safety is primarily to handle typical **software failures modes** (e.g. deadlock, task overrun, buffer overflow, division by zero).
- Software Dependability and Safety requirements need to be specified to ensure fault tolerance (e.g. through FDIR, watch-dog, exception handling, etc.) and operational contingency.

✓ **Functional Sw Dependability and Safety Requirements**: derived from System Dependability and Safety
✓ **Specific Sw Dependability and Safety Requirements**: defined by Sw Dependability and Safety
ECSS standard

Three branches:

ECSS M - Project Management
ECSS Q - Product Assurance
ECSS E - Engineering

Three levels:

1-Level: Strategy
2-Level: Objective and Function
3-Level: Methods, procedures, tools

http://www.ecss.nl/
SW Fault handling activities, ECSS Q80-03

SW Fault Prevention Methods

SW Fault Removal Methods

SW Fault Tolerance Methods

System engineering process related to SW

SW req. & architecture engineering process

SW design & implementation process

SW delivery & acceptance process

SW verification process

SW validation process
Methods identified in ECSS Q80-03 to support the assessment of software dependability and safety

- Software Failure Modes Effects and Criticality Analysis (SFMECA)
- Software Fault Tree Analysis (SFTA)
- Hardware-Software Interaction Analysis (HSIA)
- Software Hazard Analysis (HA)
- Software Common Cause Failure Analysis (SCCF)
- In service history - Re-use file

Those are all analysis activities which do not require the execution of the software
SW Dependability Methods, objective

- SW FMECA - Identify as early as possible the critical operations from the fault tolerance point of view:
  - SW Fault preventive method, potential failures are identified and their cause can be removed early in the development.
  - By making a systematic analysis of all SW functions during the architectural design phase, possible sources of errors can be identified, classified by criticality level.
- SFTA – Verify that the SW design/implementation does not contribute to System Feared Events
- HSIA – Verify that Software correctly interacts with HW and that all HW failure modes are considered
  - HW failure modes are taken into account in the software requirements definition.
  - design characteristics will not cause the software to overstress the HW, or adversely change failure severity consequences on failures occurrence.
# Dependability assessment methods applicable to life cycle phases

<table>
<thead>
<tr>
<th>Method</th>
<th>Requirements and Architecture phase</th>
<th>Design and Implementation phase</th>
<th>Verification Testing</th>
<th>Operations</th>
<th>Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software FMECA</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Less applicable</td>
<td>Less applicable</td>
</tr>
<tr>
<td>Software Fault Tree Analysis</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Less applicable</td>
<td>Less applicable</td>
</tr>
<tr>
<td>Hw-Sw Interaction Analysis</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Less applicable</td>
<td>Less applicable</td>
</tr>
<tr>
<td>Software Hazard Analysis</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Less applicable</td>
<td>Less applicable</td>
</tr>
<tr>
<td>Software Common Cause analysis</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Less applicable</td>
<td>Less applicable</td>
</tr>
<tr>
<td>In Service History</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Less applicable</td>
<td>Less applicable</td>
</tr>
<tr>
<td>Engineering analysis</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Most applicable</td>
<td>Less applicable</td>
<td>Less applicable</td>
</tr>
</tbody>
</table>
Design Constraints

A number of Dependability and Safety constraints force the adoption of Techniques and rules during design and implementation activities

- A number of Design & Coding Practices can be applied in order to
  - adopt specific architectural design choices to prevent or tolerate faults
  - implement specific functions to prevent faults
  - implement specific recovery actions to tolerate faults

<table>
<thead>
<tr>
<th>Design &amp; Coding Practices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defensive Programming</td>
</tr>
<tr>
<td>Assertion Programming</td>
</tr>
<tr>
<td>Recovery Blocks</td>
</tr>
<tr>
<td>Segregation/Partitioning</td>
</tr>
<tr>
<td>Watchdog</td>
</tr>
<tr>
<td>Alive flag</td>
</tr>
</tbody>
</table>
Fault Removal Techniques

Testing activities which require the execution of the software

<table>
<thead>
<tr>
<th>White Box Testing</th>
<th>Black Box Testing</th>
<th>Test Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Coverage</td>
<td>Back-to-Back Testing</td>
<td>Test Result Analysis</td>
</tr>
<tr>
<td>Branch Coverage</td>
<td>Interface Testing</td>
<td>Test Coverage Analysis</td>
</tr>
<tr>
<td>Path Coverage</td>
<td>Stress Testing</td>
<td>Test Witnessing</td>
</tr>
<tr>
<td>Basis Path Coverage</td>
<td>Statistical Testing</td>
<td>Fault Seeding</td>
</tr>
<tr>
<td>Multiple Condition Coverage</td>
<td>Monte-Carlo Simulation</td>
<td>Mutation Analysis</td>
</tr>
<tr>
<td>Linear Code Sequence and Jump Coverage</td>
<td></td>
<td>Sensitivity Analysis</td>
</tr>
<tr>
<td>Data Flow Coverage</td>
<td>Loop Testing</td>
<td>Regression Analysis</td>
</tr>
<tr>
<td>Loop Testing</td>
<td>Cause-Effect Graphing Technique</td>
<td></td>
</tr>
<tr>
<td>Fault Injection</td>
<td>Fault Injection</td>
<td></td>
</tr>
<tr>
<td>Run-Time Anomaly Detection</td>
<td>Run-Time Anomaly Detection</td>
<td></td>
</tr>
</tbody>
</table>

Test Data Selection

- Boundary Value Analysis
- Equivalence Partitioning

Regression Analysis
SW Dependability Methods

- Why software dependability methods?
- Static SW dependability methods
- Worst Case Execution Analyse
- How does cache effect WCEA
SW Worst Case Execution Analyse

- WCEA verifies performance requirements on a real time system
- Identifies and measure Worst Case Execution Timing (WCET)
- Results are used to assess performance and schedulability
- WCET, static or dynamic
  - Static analyse: find the longest feasible execution path, calculate execution time by support of processor model
    - + Real HW not needed
    - - Data driven systems difficult to simulate
  - Dynamic analyse: use sample execution times with worst case initial state and compute overall execution times
    - + Processor model not needed
    - - Difficult to find WC initial state
Cache processor

- Cache memory is used for high performance processor as speed gap between processor and memory
- Cache memory is relatively small and very fast
- Cache memory stores most recently accessed memory words, other schemes exist
- Instruction or data cache
- Useful terminology: read-hit, read-miss, write-hit, write-miss, cache conflict, cache thrashing
- Cache replacement policies: Least recently used (LRU)
LEON processor, architecture
LEON processor characteristics

- CMOS 0.18 µm technology
- LEON2-FT Sparc V8 with FPU
- PCI 2.2
- 86 MIPs / 23 MFlops at 100 MHz
- 700 mW at 100 MHz – 150 MIPs / W
- No Single Event Latch up below 70 MeV/mg/cm²

- Separate instruction and data cache (Harvard architecture)
- Set-associative caches: 1 - 4 sets, 1 - 64 kbytes/set. Random, LRR or LRU replacement
- Data cache snooping (DMA)
Cache impact on execution time

Cache misses and conflicts have several negative effects on program execution time:

- **Layout impact**: execution time depends on location in memory
- **Sequential impact**: execution time depends on actions taken earlier in program which influenced the state of cache
- **Concurrent impact**: execution time depends on actions taken by interrupts or higher-priority pre-empting task
Cache control mechanisms

- Freeze cache on interrupt or by program control – reduce concurrent impact of cache
- Lock cache – certain parts of cache will remain – reduce sequential and concurrent impact of cache
- Data cache write buffer
- Cache size is configurable – can be assigned specific memory areas
- Flush cache – clear cache content
- Etc.
Verification problems caused by cache

- To discover performance problems early - Need to predict SW execution times (e.g. for critical paths) at early stage in development.
  - Predictions may be based on measurements of existing similar SW and HW or estimated number machine instructions - Useful methods but cache adds uncertainty

- Performance verification of modules executed on real HW - First indication on prediction certainty
  - Measure execution time for test cases with different scenarios - Sequential and concurrent cache impacts varies for different test runs. Layout cache impacts as flight SW memory addresses are different

- Schedulability analysis – verification of real-time performance
  - Measure WCET for tasks, synchronization routines and kernel operations – cache adds uncertainty
Design and code patterns influencing cache performance

- **Cache killer pattern**
  - A program contains a structure that matches a specific pattern that makes the cache work poorly

- **Cache risk pattern**
  - A program contains a structure that under specific circumstances is a cache killer pattern but under other circumstances the cache works OK

- **Almost cache killer or cache risk**
  - Programs which becomes cache killer or cache risk during its evolution, e.g. in-flight patches
procedure P is
begin
    loop
        Pkg1.P1; -- call procedure P1 from package Pkg1
        Pkg2.P2;
        Pkg3.P3;
        Pkg4.P4;
        Pkg5.P5;
    end loop;
end P;

Assume that each package is placed in different 8KB areas and the cache is set for 8KB cache set.
procedure P is
Begin
  loop
    Pkg1.P1; -- call procedure P1 from package Pkg1
    Pkg2.P2;
    If Rare_Condition then
      Pkg3.P3;  -- call P3, but only rarely
      end if;
    Pkg4.P4;
    Pkg5.P5;
  end loop;
end P;

As long as Rare_Condition is false the loop calls only four packages and the I-cache works well.
Concurrent impact patterns

**task body** Low is
begin
  loop
    Pkg1.P1;
    Pkg2.P2;
    ...
    Pkg4.P4;
    Pkg5.P5;
  end loop;
end Low;

**task body** High is
begin
  ...
  <wait for something>;
  Pkg3.P3;
  <wait for something>;
  ...
end High;

Assume that task Low executes with no cache misses
Questions we need to answer:

Cache aware compilers and linkers are still in research state

- Can we and should we identify and avoid cache killer/risk structures?
- Is the cache becoming a SW design driver?

- What is the magnitude of cache killer/risk effect?
  - How much increases execution time?
  - How much performance margin is needed?

- What is your WCET with a cache memory?
- Do you have confidence in your Schedulability analysis?
- Is there a need for “performance failure tolerance”? 
Software Dependability Methods

Thank You for the attention!

Questions?

Maria Hernek, ESA/Estec
Maria.Hernek@esa.int