Parallel & Distributed Real-Time Systems

Lecture #15

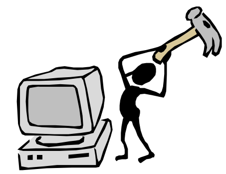
Risat Pathan

Department of Computer Science and Engineering
Chalmers University of Technology

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant techniques

What are the effects if the hardware or software is not fault-free in a real-time system?



CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant techniques

What causes component faults?

- Specification or design faults:
  - Incomplete or erroneous models
  - Lack of techniques for formal checking
- Component defects:
  - Manufacturing effects (in hardware or software)
  - Wear and tear due to component use
- Environmental effects:
  - High stress (temperature, G-forces, vibrations)
  - Electromagnetic or elementary-particle radiation

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant techniques

What types of (hardware) faults are there?

- Permanent faults:
  - Total failure of a component
  - Caused by, for example, short-circuits or melt-down
  - Remains until component is repaired or replaced
- Transient faults:
  - Temporary malfunctions of a component
  - Caused by magnetic or ionizing radiation, or power fluctuation
- Intermittent faults:
  - Repeated occurrences of transient faults
  - Caused by, for example, loose wires

Fault-tolerant techniques

What types of (software) faults are there?

- Permanent faults:
  - Total failure of a component
  - Caused by, for example, corrupted data structures
  - Remains until component is repaired or replaced
- Transient faults:
  - Temporary malfunctions of a component
  - Caused by data-dependent bugs in the program code
- Intermittent faults:
  - Repeated occurrences of transient faults
  - Caused by, for example, dangling-pointer problems

Fault-tolerant techniques

How are faults handled at run-time?

- Error detection:
  - Erroneous data or program behavior is detected
  - Watchdog mechanism, comparisons, diagnostic tests
- Error correction:
  - The originally-intended data/behavior is restored
  - Intelligent codes used for restoring corrupt data
  - Check-pointing used for restoring corrupt program flow
- Fault masking:
  - Effects of erroneous data or program behavior are ”hidden”
  - Voting mechanism

Fault-tolerant techniques

How are errors detected?

- Watchdog mechanism:
  - A monitor looks for signs that hardware or software is faulty
  - For example: time-outs, signature checking, or checksums
- Comparisons:
  - The output of redundant components are compared
  - A ”golden run” of intended behavior can be available
- Diagnostic tests:
  - Tests on hardware or software are (transparently) executed as part of the schedule

Fault-tolerant techniques

How is fault-tolerance obtained?

- Hardware redundancy:
  - Additional hardware components are used
- Software redundancy:
  - Different application software versions are used
- Time redundancy:
  - Schedule contains ample slack so tasks can be re-executed
- Information redundancy:
  - Data is coded so that errors can be detected and/or corrected

Fault-tolerant techniques

Hardware redundancy:

- Voting mechanism:
  - Majority voter (largest group must have majority of values)
  - k-plurality voter (largest group must have at least k values)
  - Median voter
- N-modular redundancy (NMR):
  - $2m+1$ units are needed to mask the effects of m faults
  - One or more voters can be used in parallel

This technique is very expensive, which means that it is only justified in the most critical applications.

Fault-tolerant techniques

Software redundancy:

- N-version programming:
  - Different versions of the program are run in parallel
  - Voting is used for fault masking
  - Software development is diversified using different languages and even different software development teams
- Recovery-block approach:
  - Different versions of the program are used, but only one version is run at a time
  - Acceptance test is used for determining validity of results

This technique is also very expensive, because of the development of independent program versions.

Fault-tolerant techniques

Time redundancy (backward error recovery):

- Retry:
  - The failed instruction is repeated
- Rollback:
  - Execution is re-started from the beginning of the program
  - Execution is re-started from a checkpoint where sufficient program state has been saved

This technique does not require additional hardware, which significantly reduces the weight, size, power-consumption and cost of the system.

Fault-tolerant techniques

Information redundancy (forward error recovery):

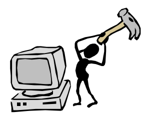
- Duplication:
  - Errors are detected by duplicating each data word
- Parity encoding:
  - Errors are detected/corrected by keeping the number of ones in the data word odd or even
- Checksum codes:
  - Errors are detected by adding the data words into sums
- Cyclic codes:
  - Errors are detected/corrected by interpreting the data bits as coefficients in a polynomial and deriving redundant bits through division of a generator polynomial

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

To extend real-time computing towards fault-tolerance, the following issues must be considered:

1. What is the fault model used?
   - What type of fault is assumed?
   - How and when are faults detected?
2. How should fault-tolerance be implemented?
   - Using temporal redundancy (re-execution)?
   - Using spatial redundancy (replicated tasks/processors)?
3. What scheduling policy should be used?
   - Extend existing policies (for example, RM or EDF)?
   - Suggest new policies?



CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

What fault model is used?

Type of fault:
- Transient, intermittent and/or permanent faults
- For transient/intermittent faults: is there a minimum interarrival time between two subsequent faults?

Error detection:
- Voting (after task execution)
- Checksums or signature checking (during task execution)
- Watchdogs or diagnostic testing (during task execution)

Note: the fault model assumed is a key part of the method used for validating the system. If the true system behavior differs from the assumed, any guarantees we have made may not be correct!

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

How is fault-tolerance implemented?

Temporal redundancy:
- Tasks are re-executed to provide replicas for voting decisions
- Tasks are re-executed to recover from a fault
- Re-execution may be from beginning or from check-point
- Re-executed task may be original or simplified version

Spatial redundancy:
- Replicas of tasks are distributed on multiple processors
- Identical or different implementations of tasks
- Voting decisions are made to detect errors or mask faults

Note: the choice of fault-tolerance mechanism should be made in conjunction with the choice of scheduling policy.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

What do existing scheduling policies offer?

Static scheduling:
- Simple to implement (unfortunately, supported by very few commercial real-time operating systems)
- High observability (facilitates monitoring, testing & debugging)
- Natural points in time for self-check & synchronization (facilitates implementation of task redundancy)

Dynamic scheduling:
- RM simple to implement (supported by most commercial real-time operating systems)
- RM and EDF are optimal scheduling policies
- RM and EDF comes with a solid analysis framework

Fault-tolerant scheduling

How do we extend existing techniques to FT?

Uniprocessor scheduling:

- Use RM, DM or EDF and use any surplus capacity (slack) to re-execute tasks that experience errors during their execution.
- The slack is reserved *a priori* and can be accounted for in a schedulability test. This allows for performance guarantees (under the assumed fault model)
- Or: re-executions can be modeled as aperiodic tasks. The slack is then extracted dynamically at run-time by dedicated aperiodic servers. This allows for statistical guarantees.

Fault-tolerant scheduling

How do we extend existing techniques to FT?

Multiprocessor scheduling:

- Generate a multiprocessor schedule that includes primary and backup (active or passive) tasks.
- Execute the primary tasks in the normal course of things.
- Execute the active backup tasks in parallel (on other processors) with the primary.
- Activate the passive backup tasks in case the execution of the primary fails.
- Schedule passive backups for multiple primaries during the same period (overloading), and de-allocate resources reserved for a passive backup if its primary completes successfully.

Fault-tolerant scheduling

Some existing approaches to fault-tolerant scheduling:

- Quick-recovery algorithm:
  - Replication strategy with dormant ghost clones
- Replication-constrained allocation:
  - Branch-and-bound framework with global backtracking stage
- Fault-tolerant First-Fit algorithm:
  - Modified bin-packing algorithm for RM and multiprocessors
- Fault-tolerant Rate-Monotonic algorithm:
  - Modified RM schedulability analysis that accounts for task re-execution

Fault-tolerant scheduling

Quick-recovery algorithm: (Krishna & Shin, 1986)

Each invocation of a periodic task is called a version.

Replicas of versions are called clones. A primary clone is executed in the normal course of things. A ghost clone is a passive backup which lies domant until it is activated to take the place of a corresponding primary whose processor has failed.

For reliability reasons, the system runs a certain number $n(i)$ of clones of version i in parallel.

A system is said to sustain up to N_{sust} failures if, despite the failure of up to N_{sust} processors in any sequence, the system is able to schedule tasks so that $n(i)$ clones of version i can be executed in parallel without deadlines being missed.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

Quick-recovery algorithm:

C1: Each version must have ghost clones scheduled on N_{sust} processors, and a ghost and a primary of the same version may not be scheduled on the same processor.

C2: Ghosts are conditionally transparent. That is:

a) two ghost clones may overlap in the schedule if none of their corresponding primary clones are scheduled on the same processor

b) primary clones may overlap ghosts on the same processor only if there is sufficient slack in the schedule to continue to meet the deadlines of all the primary and activated ghosts on that processor

C1 and C2 are necessary and sufficient conditions for up to N_{sust} processor failures to be sustained.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

Replication-constrained allocation: (Hou & Shin, 1994)

For reliability reasons, certain critical tasks must have N_{repl} replicas. The value of N_{repl} is common for all critical tasks.

The replicas can be created in one of two ways:

R1: 1 primary and N_{repl} - 1 active backups on separate processors

R2: 1 primary and N_{repl} - 1 active backups on one processor

Task deadlines decide whether R1 or R2 is used for replication:

a) if task deadline is loose enough to allow for execution of both the primary and the N_{repl} - 1 backups before the deadline, R2 is chosen

b) otherwise, R1 is chosen.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

Replication-constrained allocation:

A B&B algorithm is applied whose objective is to maximize the probability of no dynamic failure, P_{ND}, which is the probability that all tasks within one LCM period meet their deadlines even in the presence of processor or communication-link failures.

Note: When the degree of replication is increased, the reliability of the system is increased, whereas the schedulability is decreased. The probability of no dynamic failure reflects both reliability and schedulability with a bias towards schedulability.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

Replication-constrained allocation:

Task allocation is performed using a global backtracking phase:

1) Start with an initial degree of replication, $N_{repl} = 2$.
2) Replicate the critical tasks for the given value of N_{repl}.
3) Apply the B&B algorithm and obtain the maximum P_{ND}.
4) If P_{ND} exceeds a required level, increase the value of N_{repl} by one and go to Step 2.
   If P_{ND} equals the required level, finish with given N_{repl}
   If P_{ND} is lower than the required level, finish with N_{repl} -1

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

Rate-Monotonic-First-Fit (RMFF): (Dhall & Liu, 1978)

Algorithm:

- Let the processors be indexed as $\mu_1, \mu_2, \ldots$
- Assign the tasks in the order of increasing periods (that is, RM order).
- For each task τ_i, choose the lowest previously-used j such that τ_i, together with all tasks that have already been assigned to processor μ_j, can be feasibly scheduled according to the utilization-based RM-feasibility test.
- Processors are added if needed for RM-schedulability.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

FT-First-Fit: (Oh & Son, 1994)

Basic idea (a simple modification of RMFF):

- Let the processors be indexed as $\mu_1, \mu_2, \ldots$
- Assign the tasks in the order of increasing periods (RM order).
- For each replica ν of task τ_i, choose the lowest previously-used j such that ν, together with all task replicas already assigned to processor μ_j, can be feasibly scheduled according to the utilization-based RM-feasibility test.
- Processors are added if needed for RM-schedulability.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

FT-RMFF: (Bertossi, Mancini & Rossini, 1999)

Basic idea: (a refined modification of RMFF)

- Extend the RM response-time analysis with two separate tests: **NoFaultCTT** for schedulability in the absence of failures, and **OneFaultCTT** for schedulability in the presence of failures.
- Assign tasks to processors in RM order, but with every other task the backup corresponding to the recently-assigned primary.
- A backup task is made active or passive depending on the tightness of the primary's deadline.
- Depending on the type of task (primary, active/passive backup) certain combinations of the schedulability test NoFaultCTT and OneFaultCTT must be satisfied.

CHALMERS | UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling

A set of n tasks scheduled according to the RM policy always meet their deadlines if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{LL} = n\left(2^{1/n} - 1\right)$$

(Liu & Layland, 1973)

Note: a lower bound can be derived by letting $n \to \infty$.

$$\lim_{n \to \infty} n\left(2^{1/n} - 1\right) = \ln 2 \approx 0.693$$

Consequence: a task set whose utilization does not exceed ≈ 70% is always schedulable.

Fault-tolerant scheduling

A set of n tasks scheduled according to the RM policy always meet their deadlines even in the presence of a single fault (using same-priority re-execution) if

$$U \leq U_{PM} = 0.5$$

(Pandya & Malek, 1998)

Note: this bound is less pessimistic than the trivial bound:

$$\lim_{n \to \infty} n\left(2^{1/n} - 1\right)/2 \approx 0.346$$

FT-RMA: an example of caution

FT-RMA: (X, Y & Z, 1997)

Make sure there is enough slack in the RM schedule to allow for the re-execution of any task instance if a fault occurs during its execution.

The added slack is distributed throughout the schedule such that the amount of slack available over an interval of time is proportional to the length of that interval.

The ratio of slack available over an interval of time is constant and can be regarded as the utilization U_B of a backup task B.

FT-RMA: an example of caution

FT-RMA:

A recovery scheme ensures that the slack reserved in the schedule can be used for re-executing a task before its deadline, without causing other tasks to miss their deadlines.

When an error is detected at the end of the execution of some task τ_k, the system enters recovery mode. In this mode, τ_k will execute at its own priority.

FT-RMA: an example of caution

A set of n tasks scheduled according to the RM policy always meet their deadlines even in the presence of a single fault (using same-priority re-execution) if

$$U \leq U_{FT-RMA} = n\left(2^{1/n} - 1\right)\left(1 - U_B\right)$$

(X, Y & Z, 1997)

where

$$U_B = \max \frac{C_i}{T_i}$$

FT-RMA: an example of caution

A set of n tasks scheduled according to the RM policy always meet their deadlines even in the presence of a single fault (using same-priority re-execution) iff

$$\forall i : R_i = C_i + \max(H_i, L_i) + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \le D_i$$

where

H_i : overhead due to re-execution of higher-priority tasks

L_i : overhead due to re-execution of lower-priority tasks

FT-RMA: an example of caution

Embarrassing flaw #1: The lowest-priority task may miss its deadline if a fault occurs during its execution and it is re-executing.

Remedy: A task τ_r will re-execute at its own priority, except for the following case: During recovery mode, any instance of a task that has a priority higher than that of τ_r and a deadline greater than that of τ_r will be delayed until recovery is complete. (X, Y & Z, 1998)

FT-RMA: an example of caution

Embarrassing flaw #2: The lowest-priority task may miss its deadline if a fault occurs in a higher-priority task during its execution and it is re-executing.

This flaw was discovered while using formal techniques to model and analyze the correctness of existing real-time scheduling policies. (Sinha & Suri, 1999)

FT-RMA: an example of caution

Moral of the story:

Whenever possible, formally verify the implementation of a real-time system. This is particularly important in safety-critical applications!

Also make sure that you are knowledgeable regarding possibilities and limitations of the techniques used:

Task	C_i	T_i	U_i
τ_1	0.4	3.6	0.1111
τ_2	0.5	4.0	0.125
τ_3	0.9	4.5	0.2
τ_4	0.91	5.4	0.1685

This task set suffers from both flaws.

Note that its FT-RMA utilization is higher than the fundamental bound due to Pandya and Malek.

$$U_{FT-RMA} = 0.6046 > U_{PM} = 0.5$$

CHALMERS | UNIVERSITY OF GOTHENBURG

End of lecture #15