

CHALMERS

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

EDA421/DIT171 - Parallel and Distributed Real-Time Systems, Chalmers/GU, 2011/2012

- Erroneous data or program behavior is detected
- Watchdog mechanism, comparisons, diagnostic tests

Fault-tolerant techniques

• Error correction:

Updated November 29, 2011

CHALMERS

- 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

CHALMERS 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

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

CHALMERS

CHALMERS

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.

CHALMERS 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

CHALMERS

Fault-tolerant techniques

Time redundancy (backward error recovery):

- Retry:
 - The failed instruction is repeated

Lecture #12

- Rollback:
 - Execution is re-started from the beginning of the program
 - Execution is re-started from a <u>checkpoint</u> 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.

CHALMERS

Fault-tolerant techniques

Information redundancy (forward error recovery):

development of independent program versions.

- 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 multiplication and division of a <u>generator polynomial</u>

CHALMERS

Fault-tolerant scheduling

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

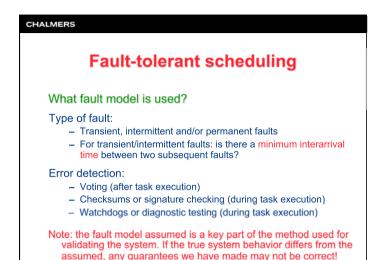
- 1. What is the fault model used?
 - What type of fault is assumed?
 - How and when are faults detected?

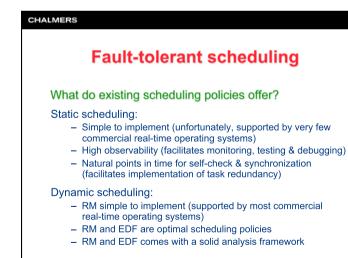
2. How should fault-tolerance be implemented?

- 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?



Lecture #12





CHALMERS 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

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.

EDA421/DIT171 - Parallel and Distributed Real-Time Systems, Chalmers/GU, 2011/2012

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

CHALMERS

Updated November 29, 2011

Fault-tolerant scheduling

Fault-tolerant scheduling

- Generate a multiprocessor schedule that includes primary and

- Activate the passive backup tasks in case the execution of the

same period (overloading), and de-allocate resources reserved

- Schedule passive backups for multiple primaries during the

for a passive backup if its primary completes successfully.

- Execute the primary tasks in the normal course of things.

- Execute the active backup tasks in parallel (on other

How do we extend existing techniques to FT?

backup (active or passive) tasks.

processors) with the primary.

Multiprocessor scheduling:

primary fails.

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

Each invocation of a periodic task is called a version.

Replicas of versions are called <u>clones</u>. A <u>primary clone</u> is executed in the normal course of things. A <u>ghost clone</u> 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 <u>sustain</u> 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

Fault-tolerant scheduling

Quick-recovery algorithm:

C1: Each version must have ghost clones scheduled on *Nsust* 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 *Nsust* processor failures to be sustained.

Fault-tolerant scheduling Replication-constrained allocation: (Hou & Shin, 1994) For reliability reasons, certain critical tasks must have Nrepl replicas. The value of Nrepl is common for all critical tasks.

The replicas can be created in one of two ways:

- R1: 1 primary and *Nrepl* 1 active backups on separate processors
- R2: 1 primary and Nrepl 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 Nrepl - 1 backups before the deadline, R2 is chosen b) otherwise, R1 is chosen.

CHALMERS

CHALMERS

Fault-tolerant scheduling

Replication-constrained allocation:

Task allocation is performed using a global backtracking phase:

- 1) Start with an initial degree of replication, Nrepl = 2.
- 2) Replicate the critical tasks for the given value of Nrepl.
- by one and go to Step 2.

CHALMERS

Fault-tolerant scheduling

Replication-constrained allocation:

Lecture #12

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

Fault-tolerant scheduling

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

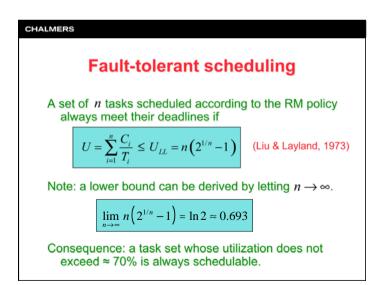
Algorithm:

- Let the processors be indexed as μ_1, μ_2, \dots
- 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_i , can be feasibly scheduled according to the utilization-based RM-feasibility test.
- Processors are added if needed for RM-schedulability.

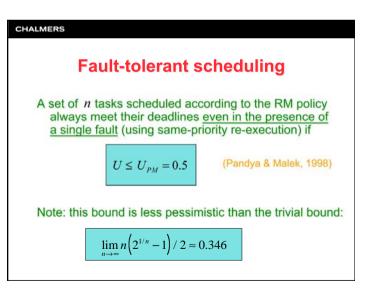
- 3) Apply the B&B algorithm and obtain the maximum PND.
- 4) If *PND* exceeds a required level, increase the value of *Nrepl*
 - If *PND* equals the required level, finish with given *Nrepl* If PND is lower than the required level, finish with Nrepl -1

CHALMERS Fault-tolerant scheduling FT-First-Fit: (Oh & Son, 1994) Basic idea (a simple modification of RMFF): - Let the processors be indexed as μ_1, μ_2, \dots - Assign the tasks in the order of increasing periods (RM order). - For each replica ν of task τ_i , choose the lowest previously-used *i* such that ν , together with all task replicas already

- used *j* such that *v*, together with all <u>task replicas</u> 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 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 <u>absence</u> of failures, and OneFaultCTT for schedulability in the <u>presence</u> 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.



FT-RMA: an example of caution

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

CHALMERS

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.

CHALMERS

FT-RMA: an example of caution

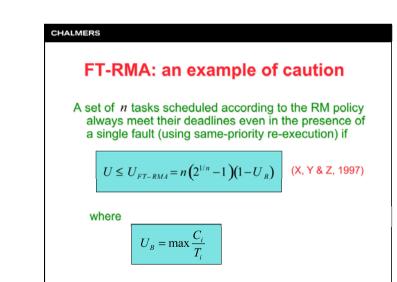
A set of *n* tasks scheduled according to the RM policy always meet their deadlines even in the presence of a single fault (using same-priority re-execution) iff

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

where

 ${\cal H}_i$: 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

FT-RMA:

A <u>recovery scheme</u> 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 R, the system enters <u>recovery mode</u>. In this mode, R will execute at its own priority.

CHALMERS

Lecture #12

its deadline if a fault occurs during its execution and it is re-executing.

FT-RMA: an example of caution

Remedy: A task P, 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 P_r and a deadline greater than that of P_r will be delayed until recovery is complete. (X, Y & Z, 1998)

EDA421/DIT171 - Parallel and Distributed Real-Time Systems, Chalmers/GU, 2011/2012

CHALMERS

Lecture #12

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)

CHALMERS

Updated November 29, 2011

CHALMERS

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	Ci	Ti	Ui
P1	0.4	3.6	0.1111
P ₂	0.5	4.0	0.125
Pз	0.9	4.5	0.2
P4	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.

 $U_{FT-RM4} = 0.6046 > U_{PM} = 0.5$