EDA122/DIT061 Fault-Tolerant Computer Systems

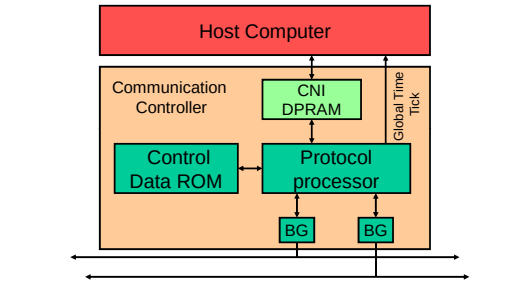
Welcome to Lecture 17

Time-triggered systems
Error detection
Course summary

Outline

- Clock synchronization in time-triggered systems
- Fault detection vs. error detection
- Error detection techniques (continued)
  - Generic principles
  - Hardware layer techniques
  - Software layer techniques
  - System layer techniques
- Wrap-up and course summary

Hardware Structure of the Communication Controller in a bus-based TTA node



CNI = Communication Network Interface

DPRAM = Dual port random access memory
BG = Bus guardian

Clock Synchronization

- Assumption: Each node sends at least one message in each TDMA communication cycle.
- The communication controller (CC) timestamps the reception of each correctly received message.
- The CC saves the actual arrival time and the expected arrival time, which allows it to calculate the skew between its own clock and the sender's clock.
- During a TDMA cycle, the CC can determine the skews between its own clock and all other clocks in the cluster.
- At regular intervals, the CC calculates the average of the clock skews and uses this to correct its own clock.
- The algorithm used for correcting the local clocks is called the Fault-Tolerant Average (FTA) algorithm.

The FTA algorithm

- Of an ensemble of *n* clocks, the FTA algorithm can tolerate *k* faulty clocks
- At a synchronization point, FTA calculates the values of the other clocks based on the latest clock skew information
- The *k* lowest and *k* highest clock values are discarded
- The average of the remaining clock values is used as a correction value

Outline

- Clock synchronization in time-triggered systems
- Fault detection vs. error detection
- Error detection techniques (continued)
  - Generic principles
  - Hardware layer techniques
  - Software layer techniques
  - System layer techniques
- Wrap-up and course summary

Fault Detection vs. Error Detection

- Terminology
  - Both the terms *fault detection* and *error detection* are used in the literature, see discussion in the beginning of Chapter 6.4 in the course book
- We distinguish between
  - Concurrent (on-line) error detection
    - Detection of errors during operation
    - Purpose is to mask or minimize adverse effects of errors
  - Non-concurrent (off-line) fault detection
    - Testing to find physical hardware faults while the system is off-line
    - Purpose is to identify faulty hardware units

Off-line fault detection techniques

- Functionality checking (p. 127 in the course book)
  - Examples:
    - Test of random access memory (RAM) by writing and reading back test patterns to all memory words
    - Test of CPU by running special test programs
- Loop back testing (p. 129 in the course book)
  - Example:
    - "echo" testing of communication paths

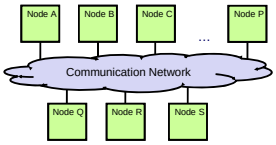
Outline

- Clock synchronization in time-triggered systems
- Fault detection vs. error detection
- Error detection techniques (continued)
  - Generic techniques
  - Hardware layer techniques
  - Software layer techniques
  - System layer techniques
- Wrap-up and course summary

Layered fault tolerance

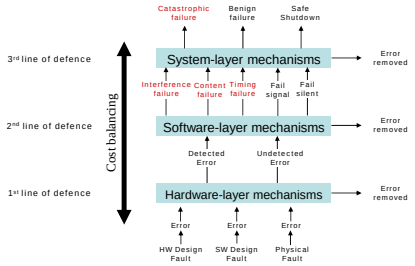
Fault tolerance can be implemented at three abstraction layers in a distributed system:

- **Hardware layer** – mechanisms implemented in hardware either *within one integrated circuit*, or by *replication of integrated circuits* within a node.
- **Software layer** – mechanisms implemented in software dealing with errors occurring within a node
- **System layer** – mechanisms implemented in software dealing with errors occurring in other nodes or the communication network.



Distributed system

Layered fault tolerance



Generic techniques for error detection (I)

- **Duplication and comparison** (pp. 128 and 138-141 in the course book)
  - Examples:
    - Comparison of redundant signals
    - Self-checking pair
- **Consistency checking** (p. 128 in the course book)
  - Uses a priori knowledge about information or system behaviour
  - Examples:
    - Hardware exceptions in CPUs
    - Range checking in software of constrained program variables

Generic techniques for error detection (II)

- **Information redundancy** (pp. 125 and 128 in the course book)
  - Use of error detecting and error correcting codes
  - Examples:
    - Cyclic redundancy check (CRC)
    - Hamming codes
- **Time redundancy** (p. 125 in the course book)
  - Examples:
    - Double time redundant execution with comparison
    - Triple time redundant execution with majority voting

Hardware techniques for error detection (examples)

- **Bus monitoring** (p. 130 in course book)
  - Checking the range of addresses generated by a CPU
  - Examples
    - Checking that the CPU use an even address when reading a 32 or 64-bit word.
    - Checking CPU memory accesses using a memory management unit (MMU).
- **Watchdog timers** (p. 130 in course book)
  - Hardware technique supported by software
- **Power supply monitoring** (pp. 130-131 in the course book)

CPU Exceptions

- Modern central processing units (CPUs) (or cores) are equipped with hardware implemented error detection mechanisms called *hardware exceptions*
- The number and type of hardware exceptions varies depending on the CPU design
- When a hardware exception is raised, the CPU stops the program execution and jumps to an exception routine
- The handling of exceptions is very similar to how a CPU responds to interrupt signals
- Some examples of common hardware exceptions is given in the next two slides

Examples of CPU exceptions (1)

Bus error: detects errors during read and write accesses to the main memory. This exception is raised (triggered) when the CPU attempts to access an address to which no memory or I/O device is connected.

Address error: detects when the CPU attempts to access memory using an address that is not aligned with the word size of the CPU.

Undefined instruction: detects if the CPU during an instruction fetch reads a value from memory (or the instruction cache) that doesn't correspond to a valid instruction. This error can occur if the program counter is erroneously loaded with an address pointing to a data area rather than a program code area.

Examples of CPU exceptions (2)

Privilege violation: detects if a user program attempts to execute an instruction which is allowed only for programs that execute in the superuser mode (privileged mode), such as the operating system or device drivers. User programs normally executes in user mode (normal mode).

Division by zero: detects if a program tries to divide a number with zero.

Spurious interrupt: detects if an interrupt is signalled but no interrupt vector is provided by the interrupting device. (The interrupt vector tells the CPU which device it was that raised the interrupt signal and thereby indicates which interrupt service routine that the CPU shall execute.)

Watchdog timers

- Watchdog timers are used to detect slow programs or programs that hang in infinite loops
- The principle is simple:
  - When a program starts to execute, either the program itself or the operating system starts a hardware timer.
  - The timer must be reset by the program within a given deadline, otherwise the timer will send an interrupt signal to the CPU.
  - The interrupt signal causes the CPU to take appropriate recovery actions, such as restarting the program or rebooting the node.
- Watchdog timers are common in embedded real-time systems.
- They can be used to "transform" *timing failures* into *signalled failures* or *silent failures*.

Initiating recovery via a watchdog timer

- Watchdog timers are often used in conjunction with other error detection mechanisms to simplify implementation of recovery
- This works as follows:
  - When an error is detected, the error detection mechanism stores an error code in a designated memory area (preferably a "crash-proof" memory)
  - The error detection mechanism then forces a program hang, which causes the watchdog timer to raise an interrupt.
  - The interrupt signal invokes a recovery routine that reads the error code and then selects and executes appropriate recovery actions.
  - Restart and recovery could be done for an entire node, a single program, or a group of programs.

Software techniques for error detection (examples)

- **Operating system assertions**
  Examples:
  - Integrity checks of data structures used by the operating system
  - Execution time monitoring of application and system processes
- **Compiler generated run-time assertions**
  Examples:
  - Value range overflow checking
  - Loop iteration bound overflow checking
  - Type checking of constrained variables
- When these mechanisms detect an error, they typically execute a TRAP or software interrupt instruction, causing the CPU to execute an exception handler that initiates appropriate recovery actions.

Examples of how acceptance tests/ software assertions can be constructed (from lecture 4)

- Satisfaction of requirements
  - Inversion of mathematical functions; e.g. squaring the result of a square-root operation to see if it equals the original operand
  - Checking sort functions; result should have elements in descending order
  - …
- Reasonable checks
  - Checking physical constraints; e.g. speed, pressure, etc
  - Checking sequence of application states
  - …

EDMs in a jet-engine controller

EDM	Description
WDM	A timer which must be reset periodically to prevent it from tripping, i.e. signalling that an error has occurred
Hardware exceptions	Hardware EDMs supported by the Motorola 68340 processor.
Software exceptions	Software checks generated automatically by MATRIXx or by the programmer using the *exception*-clause in the ADA-language. They detect erroneous execution, erroneous calculations and other errors.
Software assertions	Range checks on engine parameters.

S/W assertion	Failure condition	Possible cause	Effect when not detected
NH overspeed (HP shaft)	The measured speed of the compressor and high-pressure turbine is too high.	Overspeed of the HP shaft, failure in the input data.	There is a risk for engine disintegration.

Software exception	Description
MATH_ERROR	Raised when the predefined ADA exception NUMERIC_ERROR or CONSTRAINT_ERROR is raised. This happens if a numeric operation is undefined or when a variable is erroneously assigned.

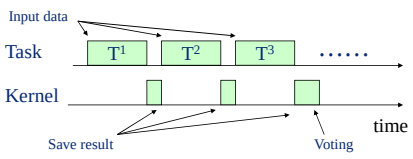
Hardware exception	No	Description
Illegal instruction	4	Occurs if the processor attempts to execute an unimplemented instruction.

Time redundancy

- Commonly used to detect or mask errors caused by transient physical faults, such as particle radiation-induced soft errors.
- **Error detection**
  - A program is run two times on the same processor
  - Errors are detected by comparing the outputs of the two runs
- **Error masking**
  - A program is run three times on the same processor
  - Errors are masked by majority voting
- **Important Assumption:**
  - Only one run is affected by an error
  - => Fault containment must be ensured between the runs.

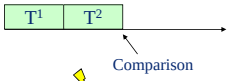
Temporal Error Masking

- A task is executed at least three times to produce three or more copies of a result
- A majority vote is performed on the copies to mask any faults
- Input data integrity ensured by end-to-end checksum

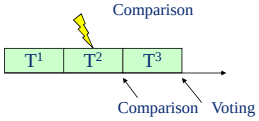


Temporal Error Masking

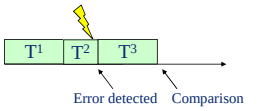
Fault free execution:



Error detected by the comparison between T^1 and T^2:



Error detected by other HW or SW EDM:



Overview of error detection mechanisms

Hardware layer
- **CPU hardware exceptions:** *Bus error, Address error, Undefined instruction, Privilege violation, Division by zero, Spurious interrupt, etc.*
- **Error detecting and correcting codes in main memory, caches, internal buffers**
- **Special hardware circuits (often connected to the non-maskable interrupt signal of a CPU):** *Power supply monitor, Network (bus) guardian.*
- **Watchdog timer (usually implemented as combination of HW and SW)**

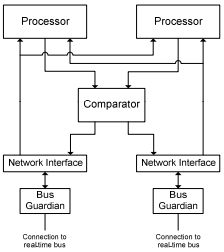
Software layer
- **Compiler:** *Type checking of constrained variables, Value range overflow, Loop iteration bound overflow*
- **OS:** *Processing time overflow, consistency checks on OS data*
- **Application:** *Time-redundant execution of tasks, application specific consistency checks*

System layer
- **End-to-end checksums**
- **Comparison of results produced by two nodes**
- **Voting on results produced by three or more nodes**

Combination of error detection techniques

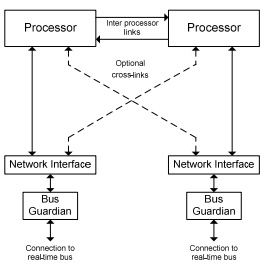
- Hardware, software and system layer techniques often combined to achieve ***high error detection coverage***.
- The next slide (from lecture 2) describes how ***duplication and comparison in hardware*** and ***end-to-end checksums*** are combined to ensure the ***fail silence property*** for a node in a distributed system.

HW architecture for a fail-silent node in a distributed system
(slide from Lecture 2)



- Processor failures are detected by ***duplication and comparison***
- The processors produce replicated messages that are compared by the comparator.
- The network interfaces receive messages from the comparator and send them to other nodes via two redundant real-time busses.
- The message content (payload) is protected by *end-to-end checksums* calculated by the content producer.
- The *end-to-end* checksums ensures that faults in the comparator and network interfaces are detectable by the service users (other nodes).

Self-checking node supporting software implemented message comparison



- The processors executes the same programs and exchange copies of outgoing messages via the inter processor links
- They compare the message copies and stops execution if the copies do not match.
- An error counter stores the number of mismatches that has occurred.
- The node is restarted after a mismatch only if the value of the error counter is below a predefined threshold
- The bus guardian protects the bus from erratic behavior (e.g., babbling idiot) of the network interfaces

Outline

- Clock synchronization in time-triggered systems
- Fault detection vs. error detection
- Error detection techniques (continued)
  - Generic techniques
  - Hardware layer techniques
  - Software layer techniques
  - System layer techniques
- Wrap-up and course summary

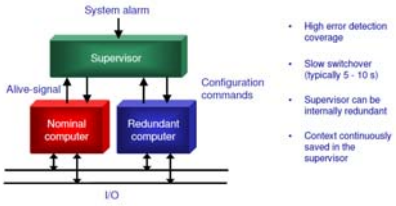
Highlights from guest lectures

- **Torbjörn Hult – Fault tolerant computers in space applications**
- **Jan Jacobson – Why standards for functional safety?**
- **Lars Holmlund – Fault tolerance in the Gripen Flight Control System**

Fault tolerant computers in space applications

RUAG

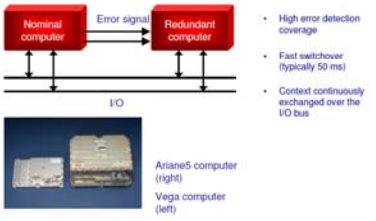
Computers in most satellites



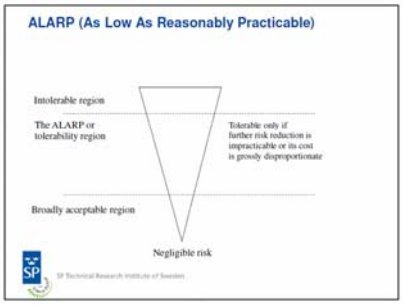
Fault tolerant computers in space applications

RUAG

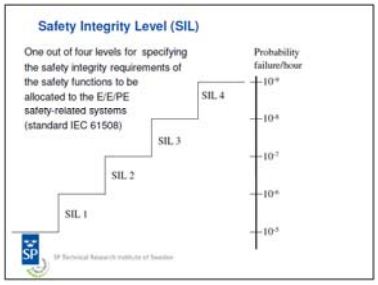
Ariane5 computers



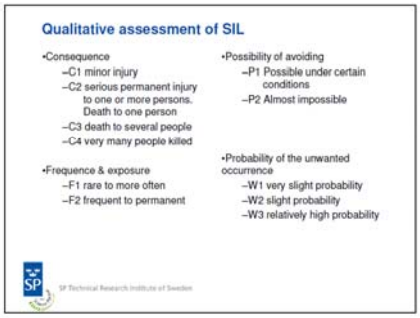
Why standards for functional safety?



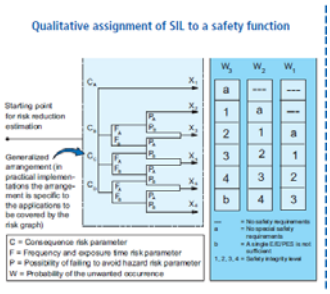
Why standards for functional safety?



Why standards for functional safety?



Why standards for functional safety?

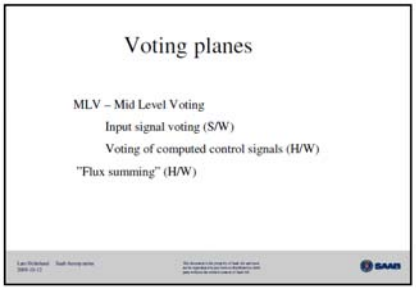


Lecture 17 EDA122/DIT061 Fault-Tolerant Computer Systems 37

JAS Gripen EFCS

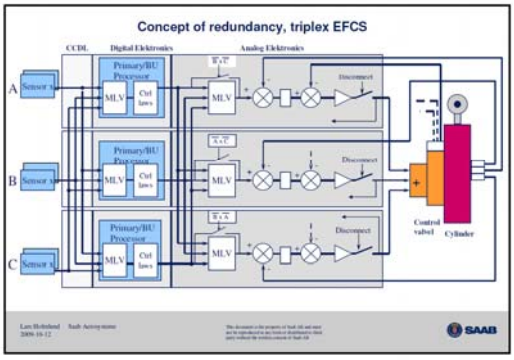
Lecture 17 EDA122/DIT061 Fault-Tolerant Computer Systems 38

JAS Gripen EFCS



Lecture 17 EDA122/DIT061 Fault-Tolerant Computer Systems 39

JAS Gripen EFCS



Lecture 17 EDA122/DIT061 Fault-Tolerant Computer Systems 40

Summary of study of failures in high-performance computing systems (I)

- Failure rates vary widely across systems, ranging from 20 to more than 1,000 failures per year, and depend mostly on system size and less on the type of hardware.
- Failure rates are roughly proportional to the number of processors in a system, indicating that failure rates are not growing significantly faster than linearly with system size.
- There is evidence of a correlation between the failure rate of a machine and the type and intensity of the workload running on it. This is in agreement with earlier work for other types of systems [2], [6], [19].
- The curve of the failure rate over the lifetime of an HPC system looks often very different from lifecycle curves reported in the literature for individual hardware or software components.
- Time between failure is not modeled well by an exponential distribution, which agrees with earlier findings for other types of systems [5], [27], [9], [15], [19]. We find that the time between failure at individual nodes, as well as at an entire system, is fit well by a gamma or Weibull distribution with decreasing hazard rate (Weibull shape parameter of 0.7-0.8).

Lecture 17 EDA122/DIT061 Fault-Tolerant Computer Systems 41

Summary of study of failures in high-performance computing systems (ii)

- Failures exhibit significant levels of temporal correlation at both short and long time lags. We find indication of autocorrelation for all types of failures; however, autocorrelation is particularly strong for hardware and software failures.
- We also find indication of spatial correlation, i.e., correlation between failures at different nodes during the same time interval. However, those are limited to failures with network root cause and not significant for other types of failures.
- Mean repair times vary widely across systems, ranging from one hour to more than a day. Repair times depend mostly on the type of the system, and are relatively insensitive to the size of a system.
- Repair times change significantly over the lifetime of a system. Both mean and median repair times drop by more than a third after the first year in operation.
- This might indicate that during the first year of operation, system administrators get significantly better at quickly identifying the root cause of a problem and fixing it.
- Repair times are extremely variable, even within one system, and are much better modeled by a lognormal distribution than an exponential distribution.

Lecture 17 EDA122/DIT061 Fault-Tolerant Computer Systems 42

Final remarks (I)

- **No system is perfect** – *single points of failure* cannot be avoided completely (cf. Fukushima nuclear disaster in Japan last year)
- **Redundancy is no panacea** – it may prevent system failures, but increases cost, failure rate and energy consumption
- **IT systems are physical artifacts** – the quality of their service depends on both the software and the hardware
  - Vertical thinking is needed – from transistors to user interfaces
- **IT-systems cannot be fully understood** – they are too complex
  - Billions of transistors, millions of lines of code lead to almost infinite numbers of fault, error and failure scenarios

Final remarks (II)

- Development of dependable IT systems requires **holistic thinking**
- We need to consider how different parts of a system interact:
  - Hardware, software, users, environment, organization, management …

Holism (from *holos*, a Greek word meaning *all*, *entire*, *total*) is the idea that all the properties of a given system (physical, biological, chemical, social, economic, mental, linguistic, etc.) cannot be determined or explained by its component parts alone. Instead, the system as a whole determines how the parts behave.

Questions?

Thank You!

Good Luck with The Exam!