Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Tobias Klaus, Florian Franzmann, Martin Hoffmann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
http://www4.cs.fau.de

4. Juni 2014

Annahmen

- C99
- x86 bzw. x86-64, d. h.
  - vorzeichenbehaftete Integer als Zweierkomplement implementiert
  - `char` hat 8 Bit
  - `short` hat 16 Bit
  - `int` hat 32 Bit
  - `long` hat 32 Bit auf x86 und 64 Bit auf x86-64

Frage 4

Zu was wird `-1L > 1U` auf x86-64 ausgewertet? Auf x86?

1. beides `0`
2. beides `1`
3. `0` auf x86-64, `1` auf x86
4. `1` auf x86-64, `0` auf x86

Erklärung

- auf x86-64 ist `int` kürzer als `long`

$\rightsquigarrow$ `unsigned int` wird zu `long` $\rightsquigarrow$ `-1L > 1L` $\Rightarrow$ `0`

- auf x86 entspricht `int` dem Datentyp `long`

$\rightsquigarrow$ `UINT_MAX > 1U` $\Rightarrow$ `1`

Frage 5

Zu was wird `SCHAR_MAX == CHAR_MAX` ausgewertet?

1. `0`
2. `1`
3. nicht definiert

Erklärung

- C99 schreibt nicht vor ob `char` vorzeichenbehaftet ist
- auf x86 und x86-64 ist `char` für gewöhnlich vorzeichenbehaftet

Frage 6

Zu was wird `UINT_MAX + 1` ausgewertet?

1. `0`
2. `1`
3. `INT_MIN`
4. `UINT_MIN`
5. nicht definiert

Erklärung

Der C-Standard garantiert, dass `UINT_MAX + 1 == 0`

Zwei Prinzipien für die Übung

KISS – Keep it Small and Simple!

- Kleine Softwaremodule mit geringer Kopplung
- *Eine* (C-)Funktion löst *eine* Aufgabe
- ☞ Bessere Wartbarkeit, Testbarkeit, Verifizierbarkeit

DRY – Don‘t repeat yourself!

- Code nicht unnötig duplizieren
- Oft benutzten (getesteten) Code wiederverwenden
- ☞ Einsatz von Bibliotheken

- Ein Beispiel: libmathe16

Verzeichnisstruktur in der Übung

- Quellverzeichnis (source)
- Hier liegen die Quelldateien
  - include ← Schnittstellenbeschreibungen (.h)
  - src ← Implementierung (.c)
  - tests ← Testfallimplementierungen (.c)
  - (cmake) ← (eigene CMake Erweiterungen)
- Binärverzeichnis (build)
- Hier landen ausschließlich(!) generierte Dateien
  - Objektdateien (.o)
  - Bibliotheken (.a)
  - Ausführbare Dateien

☞ „Out-of-Source Build“

☞ Beispiel

Buildsystem

DRY: Befehle (gcc/ar/...) nicht unnötig händisch wiederholen

- Stupides Wiederholen von Befehlen ist fehlerträchtig!
- Lösung: Buildsystem
  - Automatisiertes Bauen
  - Automatisches Auflösen von Abhängigkeiten
  - Viele existierende Lösungen: make, ANT, Maven, u.v.m.
- Wir nutzen *CMake*

Inhaltsverzeichnis

Was ist CMake?

- Ein Meta-Buildsystem!
  - Erzeugt Buildsystemdateien
    - *Makefiles* (GNU, NMake, ...)
    - *ninja*
    - Projektdateien (KDevelop, Eclipse, Visual Studio, Xcode)
  - Einfache, skriptähnliche Sprache
  - Plattform-/Betriebssystemunabhängig
  - Ermöglicht „Out-of-Source Builds“
- Weit verbreitet
  - KDE, MySQL, LLVM, u.v.m.

CMake in der Übung

- Konfigurationsdatei(en): CMakeLists.txt
- Separat in jedem Unterverzeichnis
  - Ausgehend vom Basisverzeichnis → `add_subdirectory(...)`
- Definition von sog. „Targets“
  - `add_executable(<Targetname> <Quelldatei1.c> <Quelldatei2.c>)`
  - `add_library(<Libraryname> <Quelldatei1.c> <Quelldatei2.c>)`
- Hinzubinden von Bibliotheken
  - `target_link_libraries(<Targetname> <Libraryname>)`
  - Abhängigkeiten werden automatisch erkannt
- Manuelle Festlegung von Abhängigkeiten
  - `add_dependency(<Targetname1> <Targename2>)`

Erzeugen von Makefiles

- *Außerhalb* des Quellverzeichnisses
- `% cmake <Pfad zum Quellverzeichnis>`
- `% make help` zeigt alle möglichen Targets
- `% ccmake <Buildverzeichnis>` zum Einstellen von Buildparametern

☞ Beispiel

Inhaltsverzeichnis

Debugger: gdb

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
    - Erlauben von core dumps (in der laufenden Shell): z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Programm sollte Debug-Symbole enthalten
  - `ccmake .` → `CMAKE_BUILD_TYPE: Debug`
- Aufruf des Debuggers mit `gdb <Programmname>`
- in der Übungsaufgabe: make cgdb, make gdbtui

Befehlsübersicht

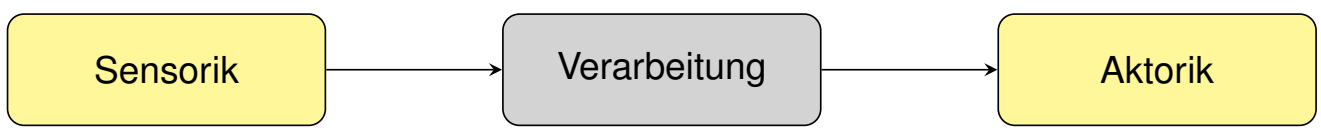
- Programmausführung beeinflussen
  - Breakpoints setzen:
    - `b [<Dateiname>:]<Funktionsname>`
    - `b <Dateiname>:<Zeilennummer>`
  - Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
  - Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
  - schrittweise Abarbeitung auf Ebene der Quellsprache mit
    - `s` (step: läuft in Funktionen hinein)
    - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
  - Breakpoints anzeigen: `info breakpoints`
  - Breakpoint löschen: `delete breakpoint#`

Befehlsübersicht

- Variableninhalte anzeigen/modifizieren
  - Anzeigen von Variablen mit: `p expr`
    - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
  - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
  - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
  - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
  - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
  - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
  - Anzeigen und Löschen analog zu den Breakpoints

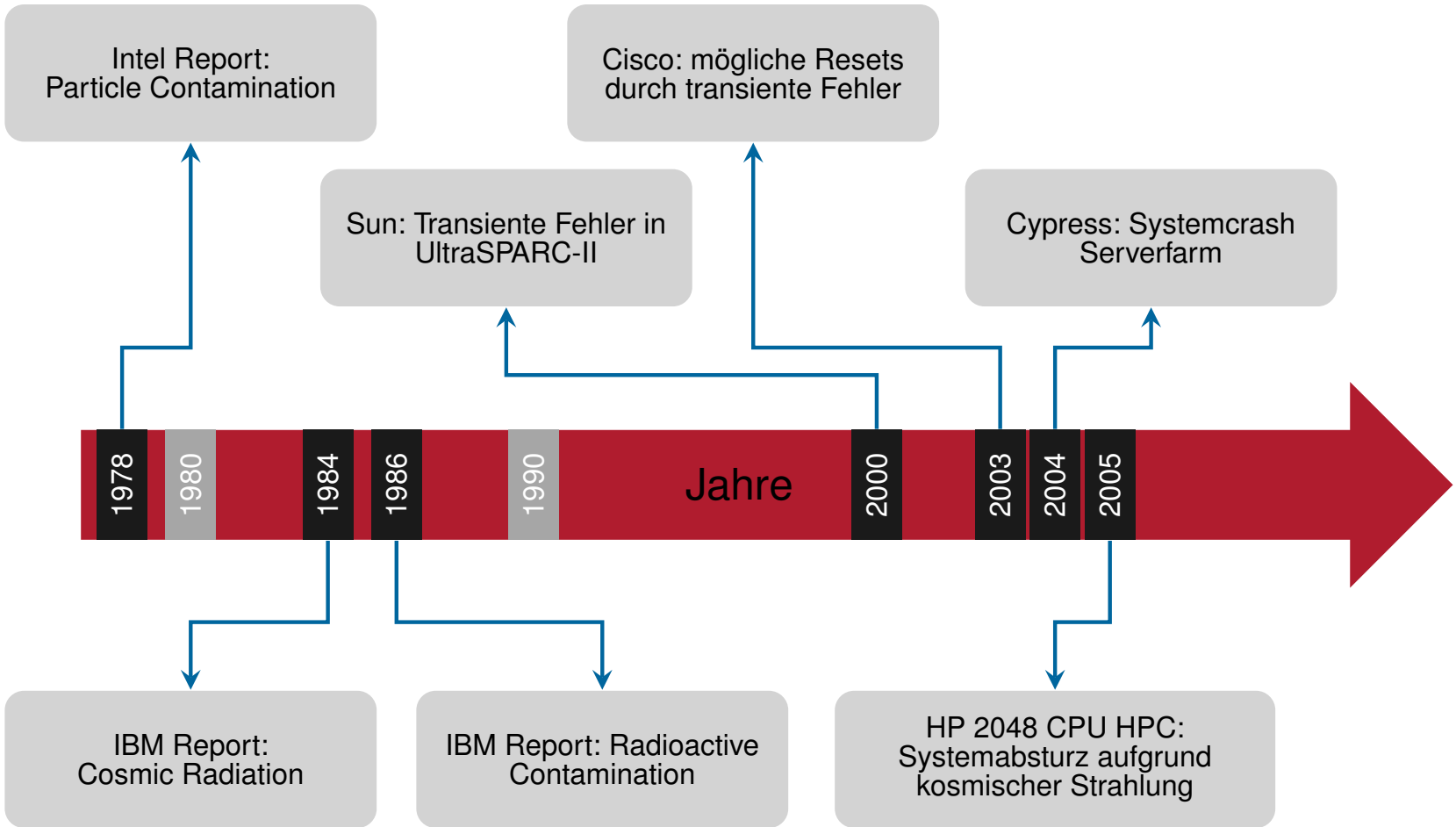
Inhaltsverzeichnis

Fehlertoleranz in eingebetteten Systemen



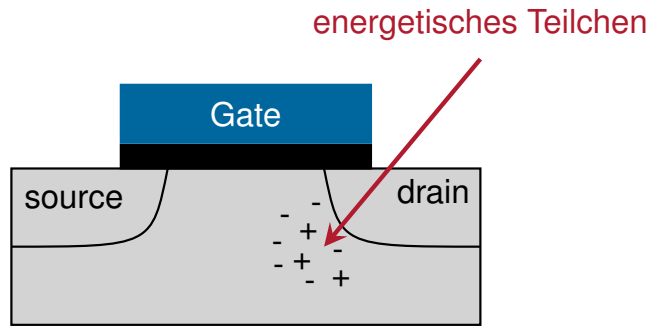
- Hohe Zuverlässigeits- und Sicherheitsanforderungen
  - Safety Integrity Level (SIL), ISO26262, . . .
  - Einsatz von Fehlertoleranztechniken
- aber auch hohe *Kostensensitivität*
  - Trend zu Multiapplikationssystemen
  - Einsatz von Mehrkernarchitekturen

Auswirkungen transienter Fehler



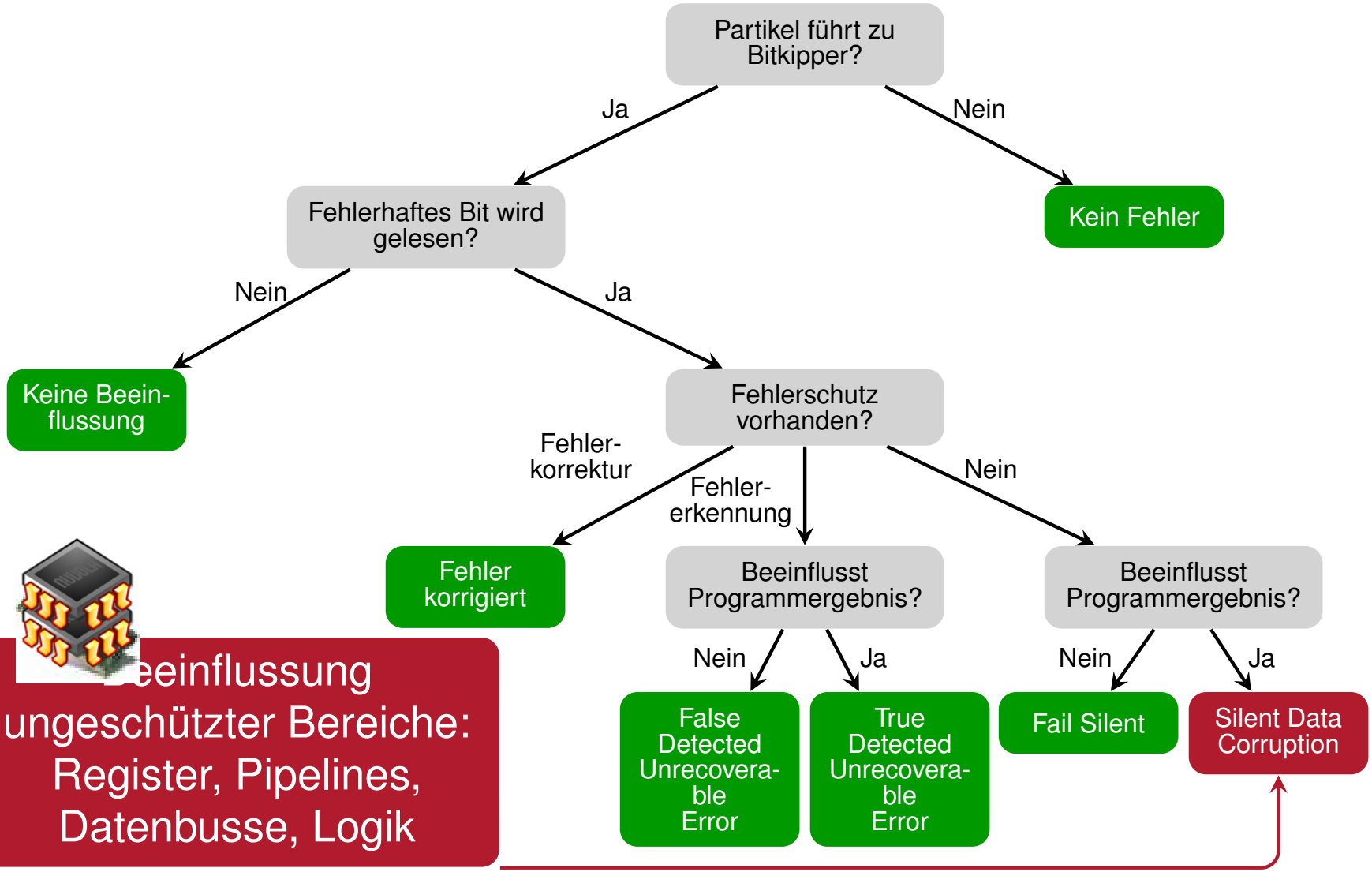
Fehlermodell - Transienter Hardwarefehler

- Steigende Fehleranfälligkeit durch sinkende Strukturgrößen
- Transienter Hardwarefehler (Single Event Upset, Soft-Error)
- Verursacht durch:
  - Ionisierende, elektromagnetische Strahlung
  - Spannungsschwankungen
  - Abgesenkte Versorgungsspannung
  - Rauschen, Übersprechen auf Leitungen



- Auswirkung transienter Fehler
  - Beeinflussung von:
    - Registerinhalten
    - Berechnungen der ALU
    - Daten oder Instruktionen auf dem CPU Bus
    - Daten auf dem Speicher- oder Peripheriebus

Auswirkungen transienter Fehler



Transiente Fehler

- Aktuelle Hardware kann bestimmte Zuverlässigkeitsanforderungen nicht mehr erfüllen!

International Roadmap for Semiconductors 2002:

> *Below* 100 *nm, single event upsets severely impact field-level product reliability, not only for memory, but for logic as well.*

Implications of microcontroller software on safety-critical automotive systems (Infineon 2008):

> *Probability of failures per hour for usual microcontroller core system is not reaching SIL3 requirements.*

- Chiphersteller empfehlen Einsatz von geeigneten Gegenmaßnahmen

N-fach modulare Hardware als Gegenmaßnahme

- Sicherheitskritische Systemkomponenten sind *mehrfach verbaut*:
- Einsatzgebiet: Maskierung von *Hardwarefehlern*
- Hardware NMR $\leadsto$ Toleranz *permanenter* HW Fehler/Ausfälle

Computers in Spaceflight: The NASA Experience[1]

> *(The Space Shuttle's) five general-purpose computers have reliability through redundancy, rather than the expensive quality control employed in the Apollo program. Four of the computers, each loaded with identical software, operate in what is termed the "redundant set" during critical mission phases such as ascent and descent. The fifth (. . .) is a backup. The four actuators that drive the hydraulics at each of the aerodynamic surfaces are also redundant, as are the pairs of computers that control each of the three main engines.*

[1] `http://history.nasa.gov/computers/Ch4-4.html`

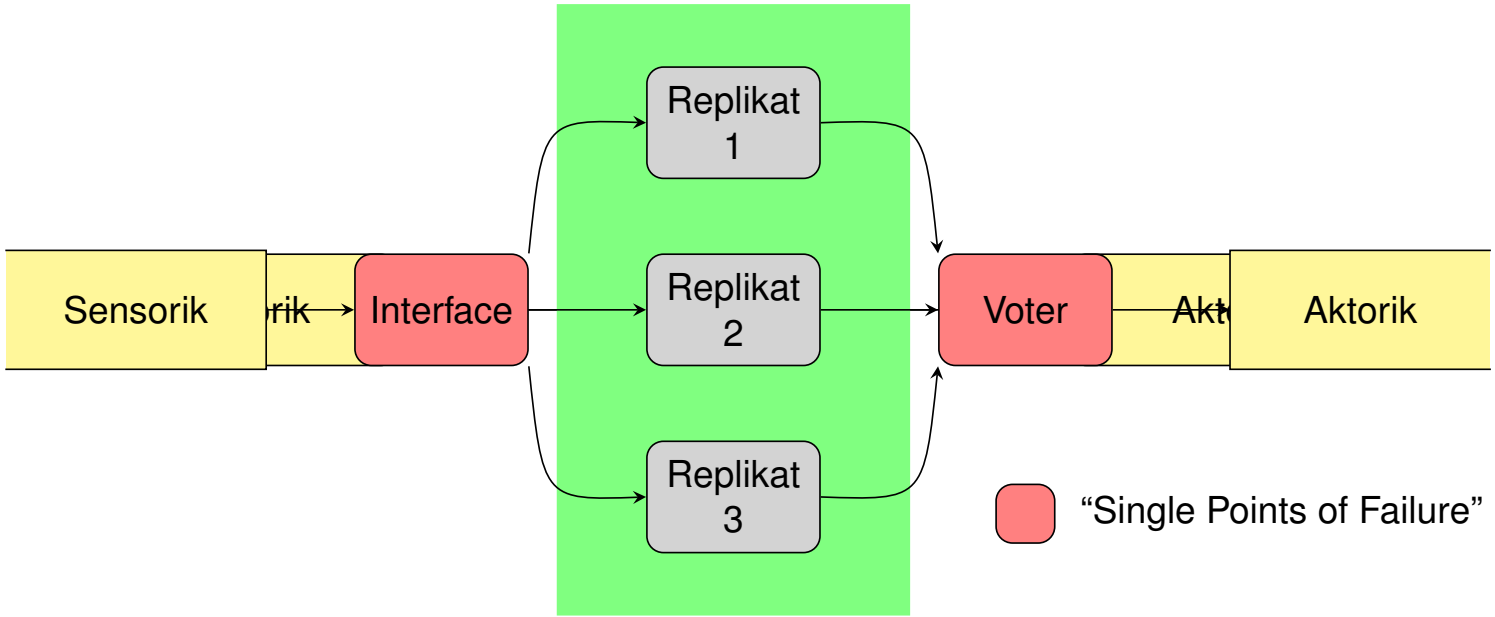
N-fach modulare Hardware (Forts.)

- “Klassische” unkomplizierte Lösung ($\rightsquigarrow$ *straightforward*)
- Vorteil:
  - Einfache, dennoch effektive Umsetzung
  - Toleranz transienter und *permanenter* Hardwarefehler
- Nachteile:
  - enormer Kostenaufwand im Sinne von:
    - Platz
    - Energie
    - Gewicht
    - Geld...
  - Keine Selektivität (Multiapplikation)
- Dennoch vorgeschrieben für hochsicherheitskritische Systeme:
  - Flugzeuge, Raumfahrzeuge, Atomkraftwerke, etc.

N-fach modulare Software

- Teure NMR-Hardware in vielen Systemen nicht umsetzbar
  - ⇝ Hohe Kostensensitivät im Automobilbereich
- Lösung: Softwarebasierte NMR
  - Mehrfache Ausführung sicherheitskritischer Softwarekomponenten
  - Vergleich der Ergebnisse (Voting)
  - Idealerweise unter Einhaltung der ursprünglichen Schnittstelle
- Aber: Nur Maskierung *transienter* Hardwarefehler

Klassische “Triple Modular Redundancy” (TMR)



- Schnittstelle sammelt Eingangsdaten (Replikationsdeterminismus)
- Verteilt Daten und aktiviert Replikate
- Mehrheitsentscheider (Voter) wählt Ergebnis
- Ergebnis wird an Aktor versendet

Redundanzbereich

Inhaltsverzeichnis

Aufgabe 2 – Vorgabe holen

- Vorherige Aufgabe in Unterordner `aufgabe1` verschieben
- Änderung ins Repository hochladen (`git add, commit push`)
- Vorgabe herunterladen:
  - `git remote add vorgabe gitosis@i4git:vezs_ss14_vorgabe`
  - `git pull vorgabe master`
- Einrichten der MySQL Zugangsdaten in: `~/.my.cnf`
  - Gruppennummer bei `user, database` setzen (z.B., `vezs1`)
  - Datenbankpasswort setzen

```
cat ~/.my.cnf
```

```
[client]
user=vezs1
database=vezs1
host=faui48d
password=YourMySQLPassword
```

Hands on: System bauen ...

- Am Anfang: `source aufgabe2/ecosenv.sh`
  - Einmal nach starten der Shell genügt → setzt Umgebungsvariablen
- Implementierung befindet sich in `app_plain.c`
- Build Ordner betreten und System kompilieren:
  - `cd <your_repo>/aufgabe2/build`
  - `cmake ..`
  - `make`
- Auflistung der existierenden Targets: `make help`
- Einrichten des Fail* Variantennamens
  - `ccmake .` → `FAILVARIANT`
  - Sichern und beenden mit Shortcut `c`, `g`

Hands on: ... und Fehler injizieren

1. Ausführung des *golden run*: `make fail-1-trace`
2. Importieren der Injektionsziele: `make fail-2-import`

→ ACHTUNG: Setzen von FAILVARIANT nicht vergessen!

→ ACHTUNG: Vorherige Ergebnisse werden gelöscht!

3. Starten des Servers zum Verteilen der Einzelexperimente: `make fail-3-server`
4. Parallelausführung der Clients: `make fail-4-client-parallel`

→ Hierfür eigenes Terminalfenster verwenden

5. Auswertung der Ergebnisse: `make fail-5-browse`

→ Browser nur lokal: `localhost:5000`

Fehlermodell

Einzelbitkipper im gesamten (tatsächlich verwendeten) RAM!
Register werden nicht injiziert!

Hands on: Untersuchung des Binaries

- Erzeugen des ISO images: `make iso`
- Jetzt gibt es eine .dis Datei (`objdump -CDS tt_scope.elf > tt_scope.dis`)
- Enthält Disassembly vermischt mit C-Code

Fragen?