Redundanz und Replikation

Fehlertoleranz auf Systemebene

Kurt Kanzenbach

Friedrich Alexander Universität Erlangen-Nürnberg

25. November 2014

Replikation

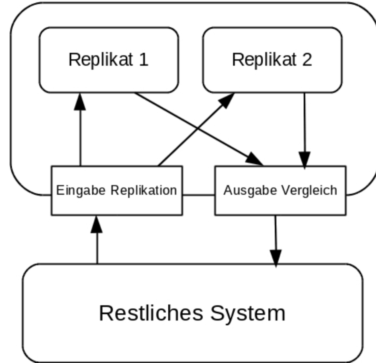

Abbildung: Replikation [1]

- Replikation: Koordinierter Einsatz von strukt. Redundanz
- Prinzip: Eingabe replizieren → Ausführung → Ausgabe vergleichen

Motivation

Abbildung: Redundante Hardware [2]

- Redundante/Fehlertolerante Hardware ist teuer
- $\Rightarrow$ Software Lösungen für Fehlertoleranz notwendig

Probleme bei Replikation

Replikdeterminismus

- Deterministisches Verhalten der Replikate
- Gleiche Eingaben $\Rightarrow$ gleiche Ausgaben?

Häufigkeit von Überprüfungen

- Wann wird repliziert?
- Wann wird verglichen?

Isolation

- Wie gut sind Replikate voreinander geschützt?

Agenda

1 Instruktions-Ebene
- SWIFT
- TRUMP

2 Prozess-Ebene

3 Redundante Virtuelle Maschinen

4 Vergleich

5 Zusammenfassung

Software-Implemented Fault Tolerance

- Compiler-Erweiterung
- Replikation von Instruktionen
- Duplizierte Instruktionen verwenden eigene Register
- Implementiert im Compiler vor Registervergabe
- Fehlererkennung: Vergleich von Registern

Software-Implemented Fault Tolerance - Ein- und Ausgabe

Eingabe

- `load`-Instruktionen
- Rückgabewerte von Funktionsaufrufen
- Annahme: Hauptspeicher/Caches mit ECC geschützt

Ausgabe

- `store`-Instruktionen
- Argumente für Funktionsaufrufe

Software-Implemented Fault Tolerance - Beispiel

```
ld r3 = [r4]

add r1 = r2, r3



st [r1] = r2
```

(a) Original Code

```
1: br faultDet, r4 != r4'
   ld r3 = [r4]
2: mov r3' = r3
   add r1 = r2, r3
3: add r1' = r2',r3'
4: br faultDet, r1 != r1'
5: br faultDet, r2 != r2'
   st [r1] = r2
```

(b) SWIFT Code

Abbildung: SWIFT Beispiel [3]

Software-Implemented Fault Tolerance - Recovery

- SWIFT-R = SWIFT + Recovery
- Dreifache Ausführung von Instruktionen
- Fehlermaskierung: durch Mehrheitsentscheid
- `br faultDet` ⇒ `majority(r, r', r")`

Software-Implemented Fault Tolerance - Bewertung

Vorteile

+ Universell anwendbar
+ Kurze Recovery-Zeiten

Nachteile

- Bandbreite
- Isolation
- Fehler in Stack Pointer/Opcodes

Triple Redundancy Using Multiplication

TRUMP

- Idee: Einsparung von einem Replikat
- 2. Replikat mit Prüfsumme versehen
- Basis: AN-Codes

AN-Codes

- A ist konstant
- Codewort: Original $\cdot A$
- Für TRUMP: $A = 3$

Triple Redundancy Using Multiplication - Beispiel

```
ld r3 = [r4]

add r1 = r2, r3



st [r1] = r2
```

(a) Original Code

```
1: call recovery, 3*r4 != r4t
   ld r3 = [r4]
2: mul r3t = 3, r3
   add r1 = r2, r3
3: add r1t = r2t, r3t
4: call recovery, 3*r1 != r1t
5: call recovery, 3*r2 != r2t
   st [r1] = r2
```

(b) TRUMP Code

Abbildung: TRUMP Beispiel [3]

Triple Redundancy Using Multiplication - Bewertung

Wie bei SWIFT, zusätzlich:

Vorteile

+ Niedrigerer Ressourcenverbrauch
+ Bessere Performance

Nachteile

- Nicht einsetzbar bei logischen Operationen
- Niedrigere Zuverlässigkeit ⇒ Hybrid-Techniken? [3]

Überblick

1 Instruktions-Ebene
- SWIFT
- TRUMP

2 Prozess-Ebene

3 Redundante Virtuelle Maschinen

4 Vergleich

5 Zusammenfassung

Process-level Recovery

- Zweite Möglichkeit: Prozessinkarnationen replizieren
- Process-level Recovery (PLR) nach Alex Shye et al. [4]
- SoR: Anwendung und Bibliotheken
- Erzeugung von redundanten Prozessinkarnationen pro Anwendung
- Isolation der Prozesse durch das Betriebsystem

Process-level Recovery - Prinzip

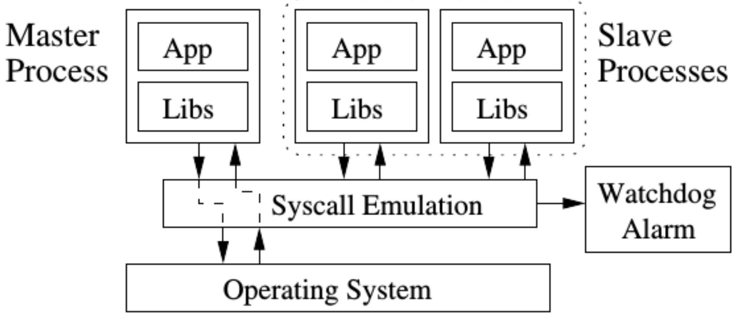

Abbildung: Process-level Recovery [4]

Process-level Recovery - Ein- und Ausgabe

Eingabe

- Lesende Systemaufrufe einmal ausführen
- Ergebnis replizieren

Ausgabe

- Schreibende Systemaufrufe einmal ausführen
- Ergebnis vergleichen

Process-level Recovery - Bewertung

Vorteile

+ Frequenz der Vergleiche geringer
+ Keine Modifikation von Anwendung/Betriebssystem
+ Isolation

Nachteile

- Höhere Fehlererkennungslatenz
- Fehler im PLR-Code/Betriebssystem
- Kritische Performance bei speicherintensiven Anwendungen

Überblick

1 Instruktions-Ebene
- SWIFT
- TRUMP

2 Prozess-Ebene

3 Redundante Virtuelle Maschinen

4 Vergleich

5 Zusammenfassung

Hypervisor-basierte Fehlertoleranz

- Dritte Möglichkeit: Virtuelle Maschinen replizieren
- Replikation von kompletten Systemen
- Hypervisor-basierte Fehlertoleranz nach Bressoud und Schneider [5]
- SoR: Virtuelle Maschinen

Hypervisor-basierte Fehlertoleranz - Prinzip

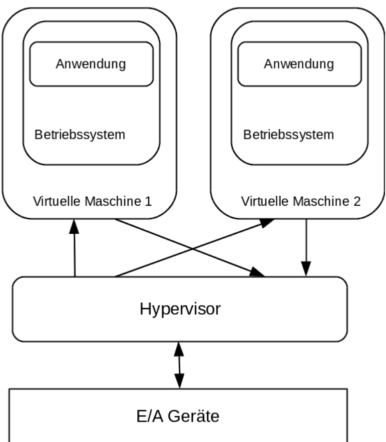

Abbildung: RVM-System mit zwei Virtuellen Maschinen

Hypervisor-basierte Fehlertoleranz - Protokoll

Instruktionen

- Deterministische an Hardware weitereichen
- **Nicht**-deterministische nur einmal ausführen

Interrupts

- Aufteilung der Ausführung in Epochen
- Interrupts puffern und zum gleichen Zeitpunkt weitereichen

Hypervisor-basierte Fehlertoleranz - Bewertung

Vorteile

+ Keine Modifikation von Anwendung/Betriebssystem
+ Isolation

Nachteile

- Hohe Anzahl an Ein- und Ausgaben
- Fehler im Hypervisor
- Performance

Überblick

1 Instruktions-Ebene
- SWIFT
- TRUMP

2 Prozess-Ebene

3 Redundante Virtuelle Maschinen

4 Vergleich

5 Zusammenfassung

Replikation auf unterschiedlichen Ebenen

- Replikation auf verschiedenen Ebenen hat Vor- und Nachteile
- Hohe Ebene:
  - $\uparrow$ Erkennungslatenz von Fehlern
  - $\uparrow$ Flexibilität
  - $\downarrow$ Häufigkeit von Vergleichen
- Niedrige Ebene:
  - $\downarrow$ Replikationskomplexität
  - $\downarrow$ Recovery Zeit

Gegenüberstellung

Ebene:	**Instruktion**	**Prozess**	**Virtualisierung**
Replikate	Instruktionen	Prozesse	Virtuelle Maschinen
SoR	Prozessor	Anwendung	System
Vergleiche	`stores`	Systemaufrufe	E/A Operationen
Replikation	`loads`	Systemaufrufe	Interrupts
Isolation	--	+	++
Komplexität	++	o	+
Zuverlässigkeit	o	+	?
Performance	--	+	--
Recovery	++	o	o

Tabelle: Vergleich von Replikation auf unterschiedlichen Ebenen

Zusammenfassung

- Software Techniken erfreuen sich Beliebtheit
- Grundlage: Redundanz und Replikation
- Prinzip: Eingabe replizieren $\rightarrow$ Ausführung $\rightarrow$ Ausgabe vergleichen
- Replikation auf unterschiedlichen Ebenen hat Vor- und Nachteile

Literatur I

- S. Mukherjee, *Archtitecture Design For Soft Errors*. Morgan Kaufmann Publishers Inc, 2008.
- "Woodcrest Cluster," http://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/woodcrest-cluster.jpg, 2014, online; accessed 15-Nov-2014.
- G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE Micro*, vol. 27, no. 1, pp. 36–47, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1109/MM.2007.4

Literatur II

A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 297–306. [Online]. Available: http://dx.doi.org/10.1109/DSN.2007.98

T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 80–107, Feb. 1996. [Online]. Available: http://doi.acm.org/10.1145/225535.225538