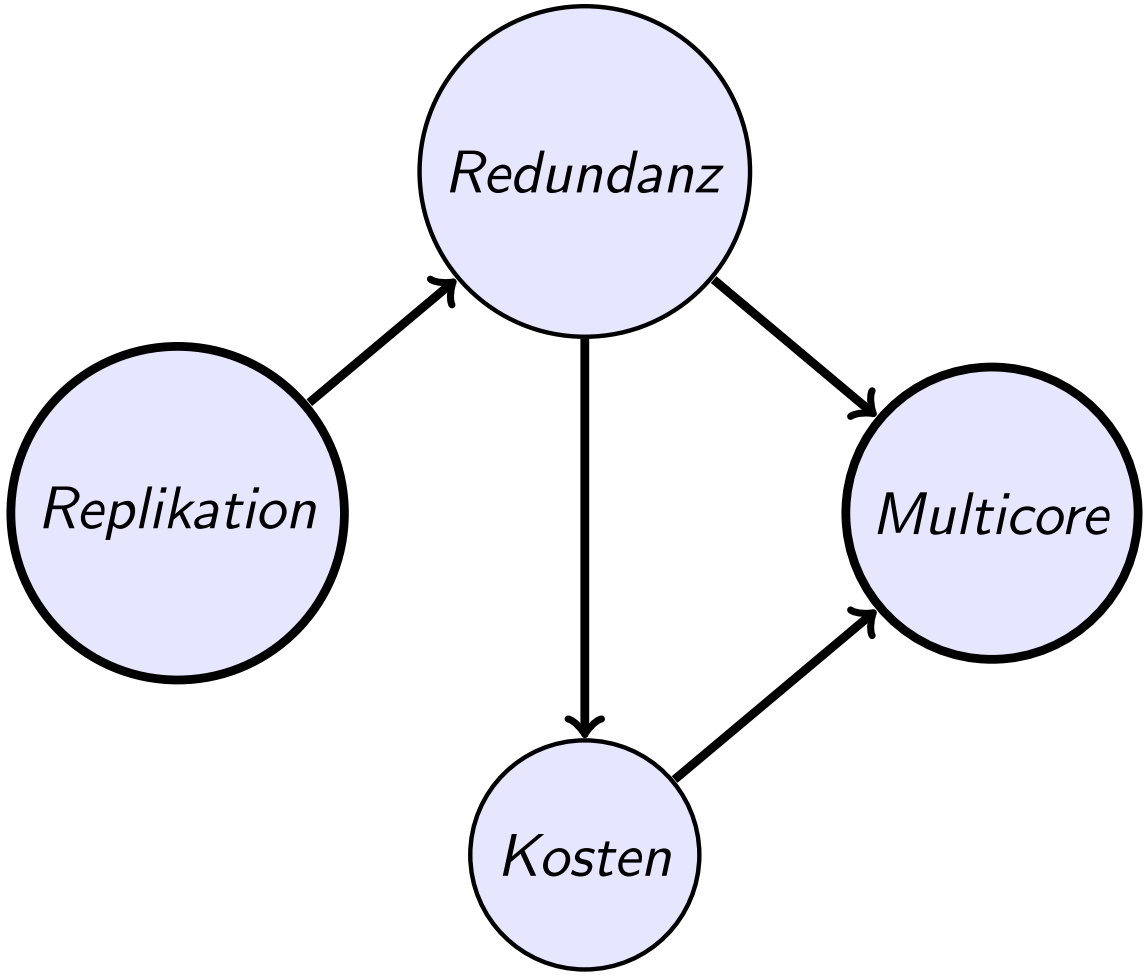
Fehlertoleranz in Mehrkernsystemen

Stefan Reif

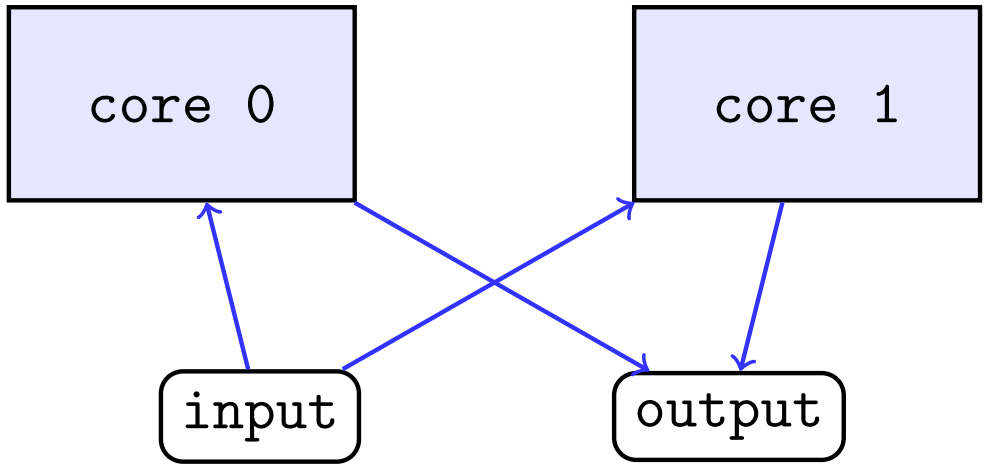
13.01.2015

Multicore-Prozessoren und Fehlertoleranz



$\Rightarrow$ Wie nutzt man Multicore-Prozessoren effizient für Fehlertoleranz?

Hardware-Lockstepping[1]



- SOR: Prozessorkern
- 2 Kerne arbeiten synchron
- Zusätzlich in Hardware: Eingabe duplizieren und Ausgabe vergleichen
- Fehler = Unterschiedliche Ausgabe $\Rightarrow$ Trap
- Nötig: Deterministische Prozessor-Kerne

Hardware-Lockstepping (synchron)

Bewertung von Lockstepping

- \+ relativ einfaches Konzept
- \+ bewährt, bekannt

} Software-Sichtweise

- – Synchronisierung der Prozessorkerne schwierig
- – ISA-Determinismus schwierig sicherzustellen
- – Kosten
  - Hardware-Aufwand
  - Technische Beschränkungen

} Hardware-Sichtweise

Hardware-Lockstepping (synchron)

Bewertung von Lockstepping

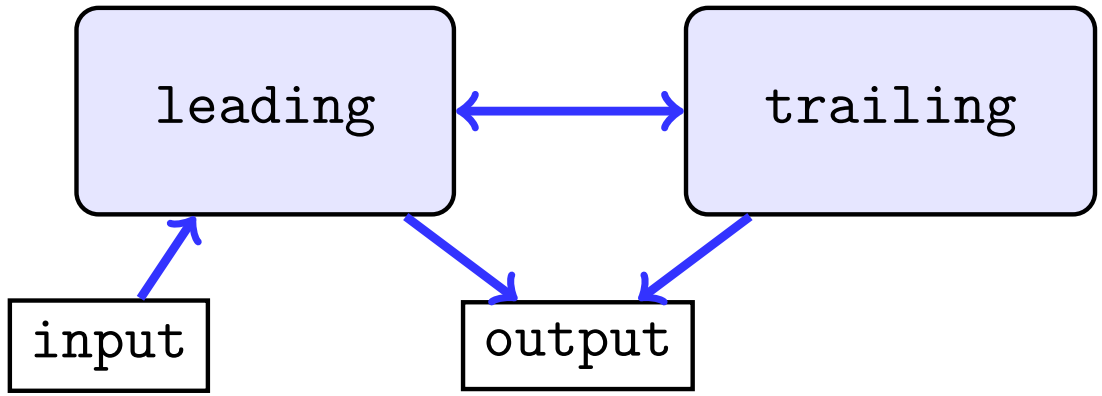
- \+ relativ einfaches Konzept
- \+ bewährt, bekannt
- – Synchronisierung der Prozessorkerne schwierig
- – ISA-Determinismus schwierig sicherzustellen
- – Kosten
  - Hardware-Aufwand
  - Technische Beschränkungen

Gesucht: Alternative

- Schwächere Synchronisierung
- Schwächerer Determinismus

} Hardware-Sichtweise

Asynchrones Multithreading



Konzept

- Zwei Threads: *leading* und *trailing*
- Bedingung: Beide Threads Äquivalent
- SOR: Lokaler Zustand der Threads
- Koordination: Lade- und Schreib-Operationen

SRT – Simultaneous and Redundand Threading

Umsetzung[2]

- Zwei Threads mit Simultaneous Multithreading (SMT) [1]
  - → Hardware-Unterstützung
  - → Feingranular geteilte Resourcen
- SOR: Prozessorzustand
- Input: Lade-Operationen
- Output: Schreib-Operationen

[1] alias *Hyperthreading*

SRT – Simultaneous and Redundand Threading (2)

2 Rollen: *leading* und *trailing* Thread

- Geteilte Resourcen $\Rightarrow$ Konflikt
- Zeitlicher Versatz der beiden Threads $\Rightarrow$ Konfliktauflösung
- Idealfall: Kein Konflikt um gemeinsame Hardware

Vorteile von SRT gegenüber Lockstepping

+ Weniger Vergleich-Operationen
+ Effizientere Hardware-Auslastung
+ Asynchrone Ausführung
+ Nichtdeterminismus in Hardware teilweise erlaubt

Hardware-Unterstützung

- Ziel: Bessere Performance
- SMT[2] $\rightarrow$ effiziente Kommunikation
- Lade-Operationen: Load-Value-Queue
- Schreib-Operationen: Store-Value-Queue

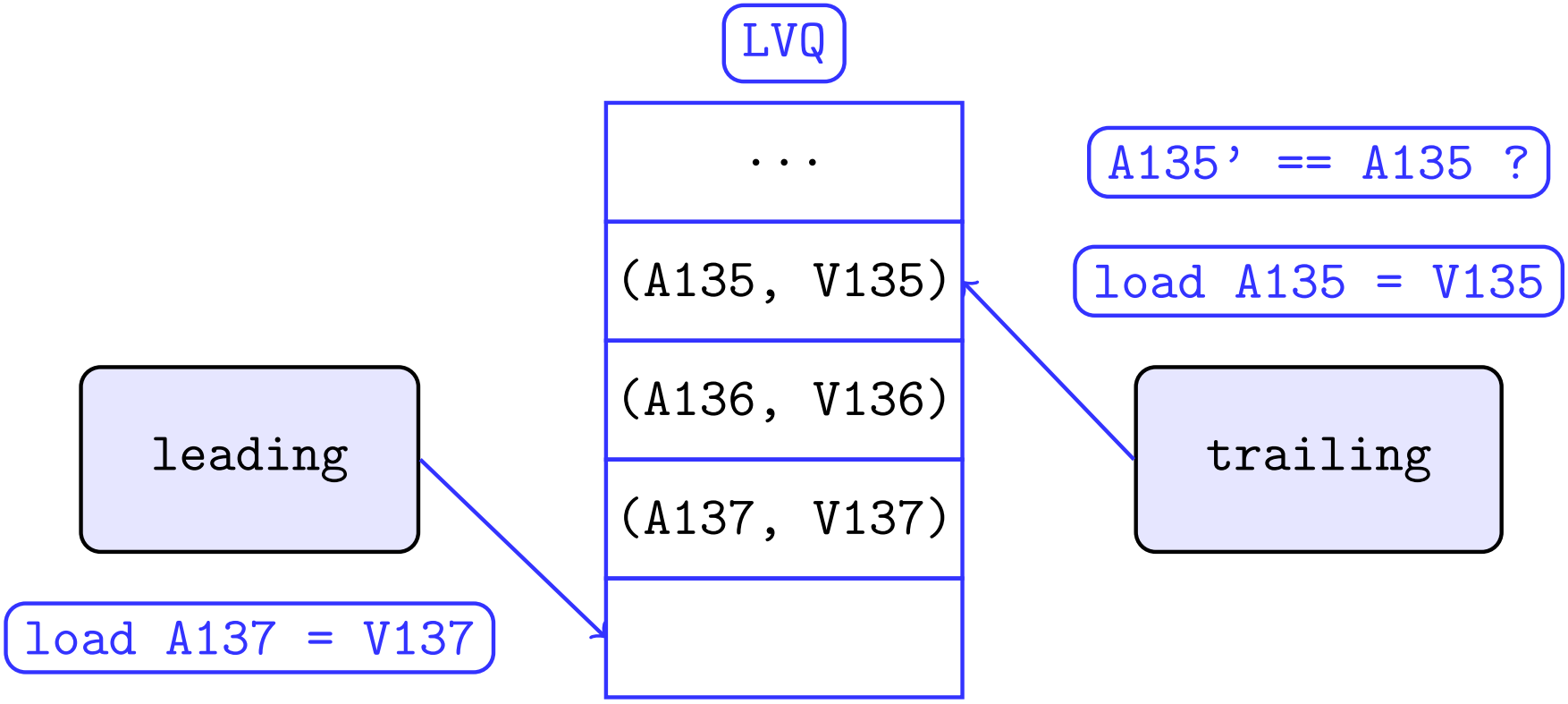
} Determinismus und Fehlererkennung

- Slack Fetch
- Branch Outcome Queue

} Effizienz

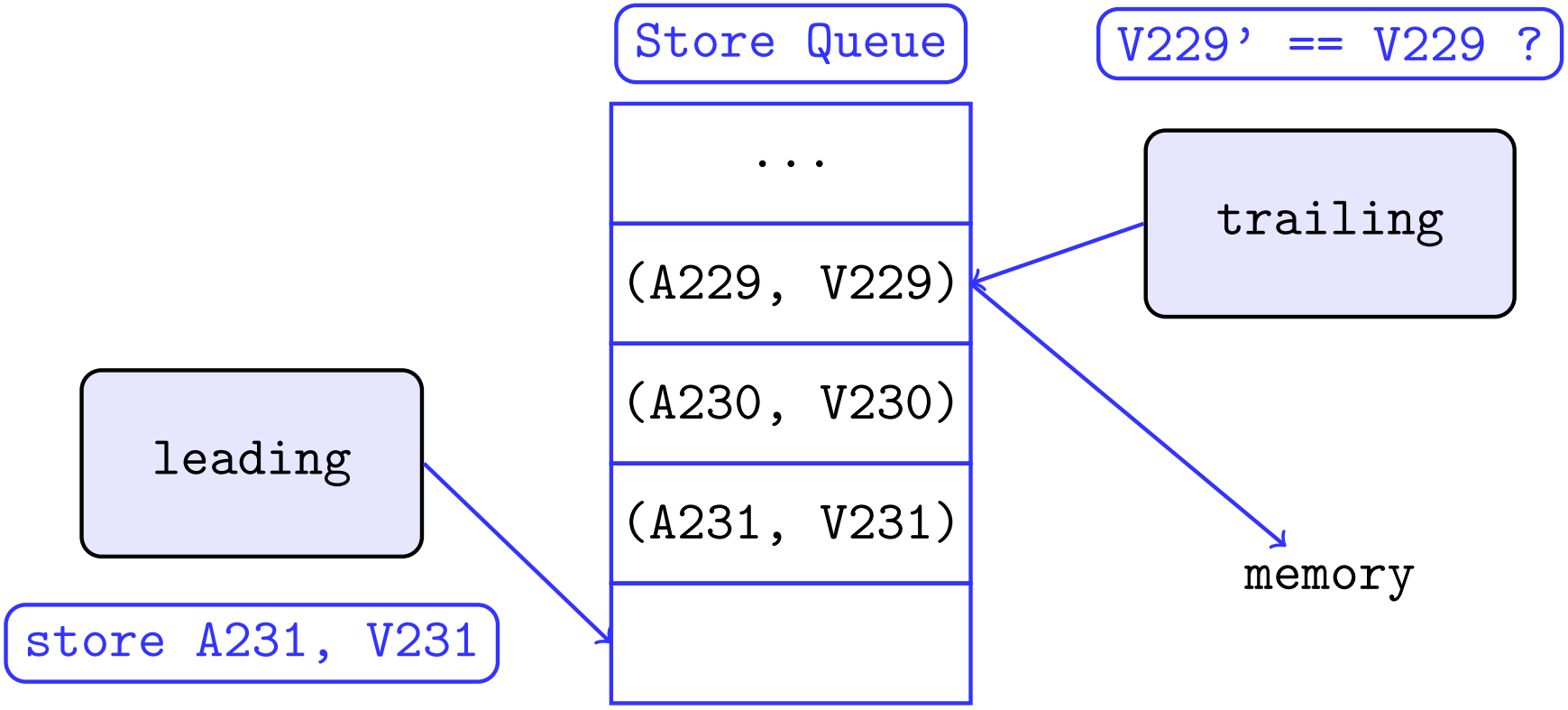
[2] alias *Hyperthreading*

Lade-Operationen



$\Rightarrow$ Kopie der Eingabe

Speicher-Operationen

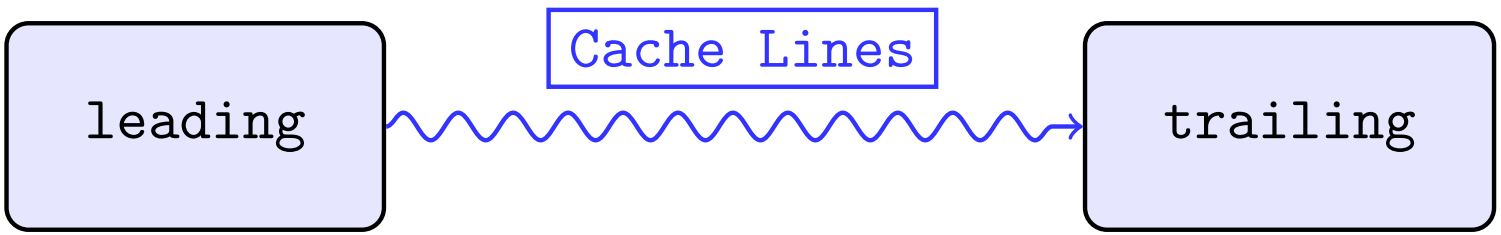


$\Rightarrow$ Vergleich der Ausgabe

Effiziente Implementierung

Slack Fetch

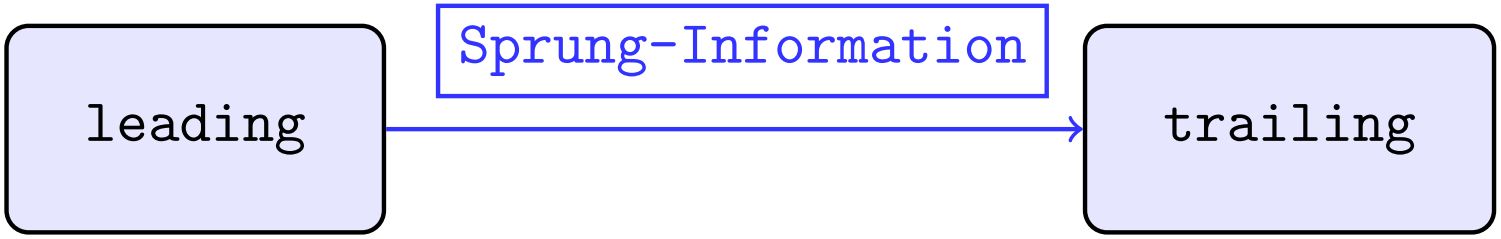
- Ziel: Trailing Thread hat möglichst wenige Cache-Misses
- Lösung: Zeitdifferenz an Cache-Zugriffen ausrichten
- Implementierung über Manipulation der Cache-Hardware



Effiziente Implementierung (2)

Branch Outcome Queue

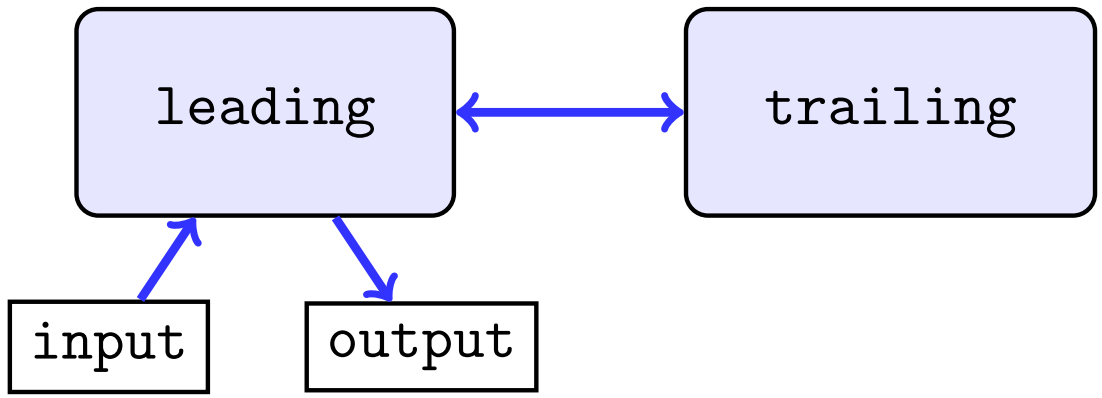
- Ziel: Trailing Thread kennt bei allen bedingten Sprüngen den Ausgang
- Lösung: Leading Thread kommuniziert Sprünge an Trailing Thread
- Implementierung über Hardware-Queue



SRT – Evaluation

+ Effizienter als Lockstepping
  - Weniger Hardware-Aufwand
  - 16% - 29% schneller[2]
+ Asynchrone ausführung
− Hardware-Unterstützung nötig
− Relativ viel Kommunikation nötig
→ Kann man SRT auch ohne Hardware-Unterstützung implementieren?

SRMT – Software-based Redundant Multithreading



Konzept[4]

- Ähnlich zu SRT, keine Hardwareunterstützung nötig
- Gemeinsamkeit: leading und trailing Thread
- Unterschied: Compiler-Transformationen
- SOR: Thread-Zustand

SRMT

Compiler-Transformationen

- Erzeugen von zwei Threads
  - → Unterschiedliche Versionen aller Funktionen
  - → Kommunikation
- Eingabe-Duplikation
- Ausgabe-Vergleich

Input-Duplikation

- leading Thread lädt alle Werte aus globalem Speicher
- Ergebnis wird an trailing Thread kommuniziert

# original code	# leading thread	# trailing thread
...	...	...
ld r1, [mem1]	ld r1, [mem1]	
	send r1- - - - - - - ▶	**receive r1**
// use r1	// use r1	// use r1

Ausgabe-Vergleich

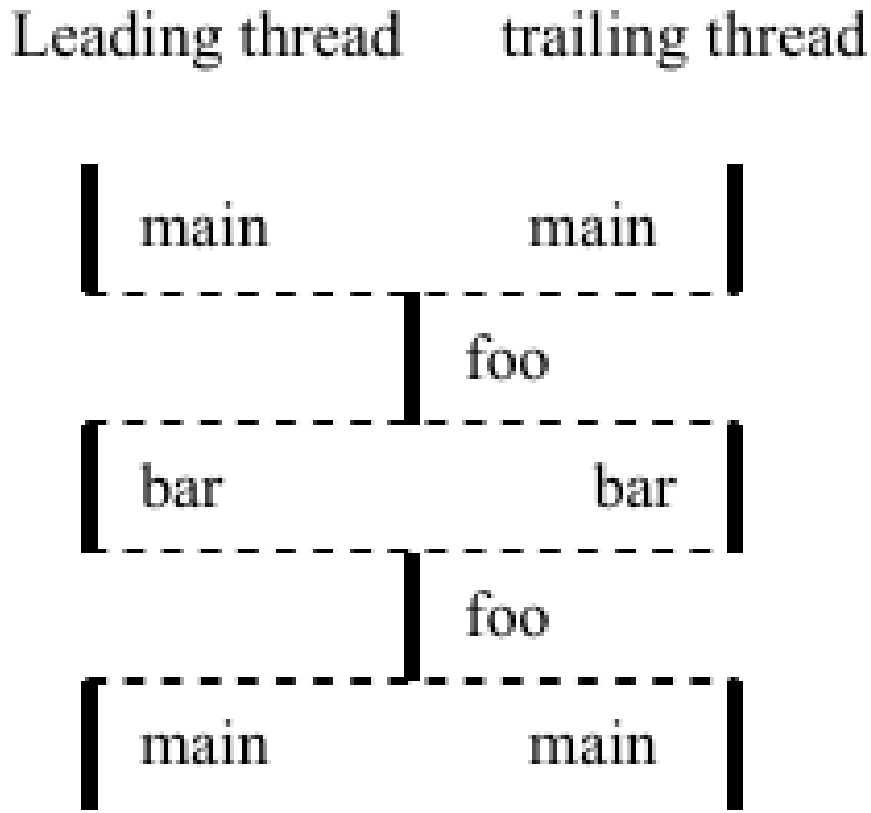
- leading Thread sendet Wert an trailing Thread
- trailing Thread sendet Bestätigung

st r2, [mem2]	send mem2 send r2 **wait**(ack) ◀- - - - - st r2, [mem2]	receive mem2' receive r2 check mem2 and mem2' check r2 and r2' **signal**(ack)

Extern Sichtbare Funktionen

```
// SRMT function
main() {
   …
   ret = foo();
   …
}
// SRMT function
bar() {
   …
}
// binary function
foo() {
   …
   ret = bar();
   …
}
```

(a)



(b)

Extern Sichtbare Funktionen (2)

- Problem: Bibliotheks-Funktion (nicht-SRMT) ruft SRMT-Funktion auf $\Rightarrow$ welche?
- Lösung: Dritte, extern sichtbare Variante der Funktion
- Externe Version führt beide anderen Thread-Versionen in zwei Threads aus
- Nur bei extern sichtbaren Funktionen nötig

$\Rightarrow$ Kombinationen mit nicht-SRMT-Code möglich

Optimierung der Kommunikation

Optimierte Software-Queue

- Nachrichtenpuffer + Verzögertes Senden
- Gepuffertes Versenden von Paketen

$\Rightarrow$ Bessere Cache-Effizienz

Kommunikation in Hardware

- Hardware-Unterstützung für Inter-Core-Kommunikation
- Wünschenswert aber oftmals nicht vorhanden

SRMT – Evaluation

Performance

- Software-Kommunikation: **186%** Overhead
- Hardware-Kommunikation: **19%** Overhead

$\rightarrow$ verglichen mit Programm ohne SRMT

Vor- und Nachteile

$+$ Hardware-Unabhängig

$+$ Abwägung Performance $\Leftrightarrow$ Fehlertoleranz möglich

$+$ Compiler erzwingt Replik-Determinismus

$-$ Kommunikation ist Flaschenhals

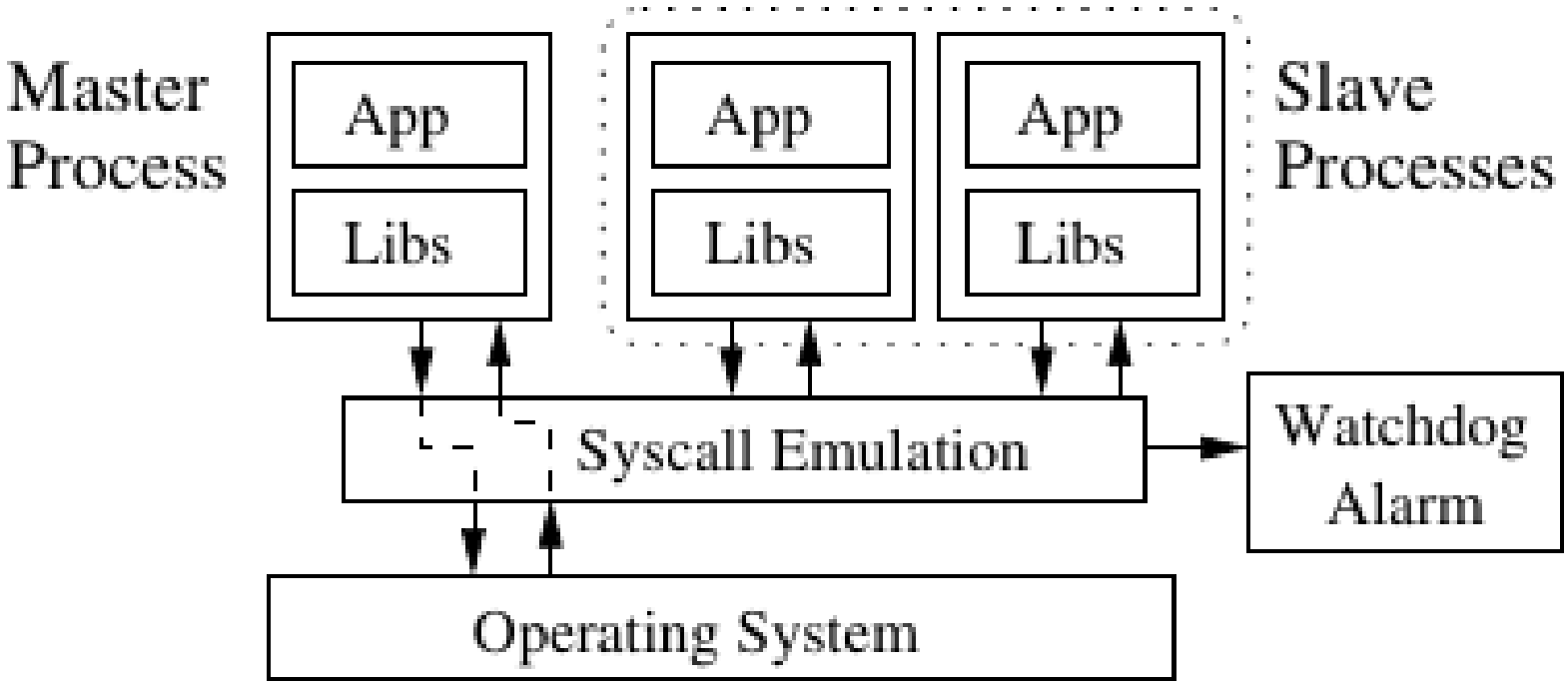
$\rightarrow$ Kann man mit weniger Vergleichen Fehlertoleranz erzielen?

PLR – Process Level Redundancy

Konzept[3]

- Mehrere redundante Prozesse
- SOR: Prozess ($\rightarrow$ Adressraum)
- Synchronisation bei Systemaufrufen
- Rein Software-basierter Ansatz

PLR-Architektur



PLR aus Sicht der Anwendung

- Transparent für Anwendung und Betriebssystem
- Deterministische Anwendung nötig
- Optimal bei wenig Input/Output

PLR – Evaluation

Performance

- 2 Prozesse → 16.9% Overhead
- 3 Prozesse → 41.1% Overhead

→ verglichen mit 1-Prozess-Ausführung des Programms

Vor- und Nachteile

+ Flexibilität
+ Portabilität
  → Hardware-Unabhängig
+ Ignorieren gutartiger Fehler
  → Sehr wenige Vergleiche
– Hoher Overhead bei Input/Output
– Hardware-Eigenschaften nicht effizient nutzbar

Bewertung

	LS	SRT	SRMT	PLR
Portabilität	(−−)	(−−)	(++)	(+)
Flexibilität	(−−)	(−−)	(+)	(++)
Performance	(−)	(++)	(+)	(0)
Anwendungs-Transparenz	(++)	(++)	(0)	(+)
Auswirkungen gutartiger Fehler	(−−)	(−)	(+)	(++)

Fazit

- Mehrkern-Prozessoren $\Rightarrow$ Strukturelle Redundanz
- Hardware-basiert $\rightarrow$ Lockstepping, SRT
- Software-basiert $\rightarrow$ SRMT, PLR
- Unterschiede bezüglich
  - Portabilität
  - Flexibilität
  - Performanz
  - Transparenz für Anwendung, Hardware
  - Auswirkungen gutartiger Fehler
- Einen besten Ansatz gibt es nicht


S. Mukherjee.
Architecture Design for Soft Errors.
Morgan Kaufman Publishers, 2008.


S. K. Reinhardt and S. S. Mukherjee.
Transient fault detection via simultaneous multithreading, volume 28.
ACM, 2000.


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 *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 297–306. IEEE, 2007.


C. Wang, H.-s. Kim, Y. Wu, and V. Ying.
Compiler-managed software-based redundant multi-threading for transient fault detection.

In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 244–258. IEEE, 2007.