COMPOSITA:
Eine Studie über Laufzeit-Architekturen für massiv parallele Systeme

Luc Bläser
HSR Hochschule für Technik Rapperswil
Jürg Gutknecht
ETH Zürich

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Vortrag KPS 2013
1 Oktober 2013

Motivation

- Heutige Betriebssysteme haben weitgehend gleiche Architektur
  - Gemeinsame UNIX-Gene (1960/70s)
- Anforderungen haben sich geändert
  - Nebenläufigkeit und Multi-Cores
  - Skalierung und Effizienz verbesserungsfähig
- Unser Ansatz: Neudesign von Grund auf
  - Hochgradig parallele Programmiersprachen
  - Möglichst effiziente Multi-Core Nutzung

Composita

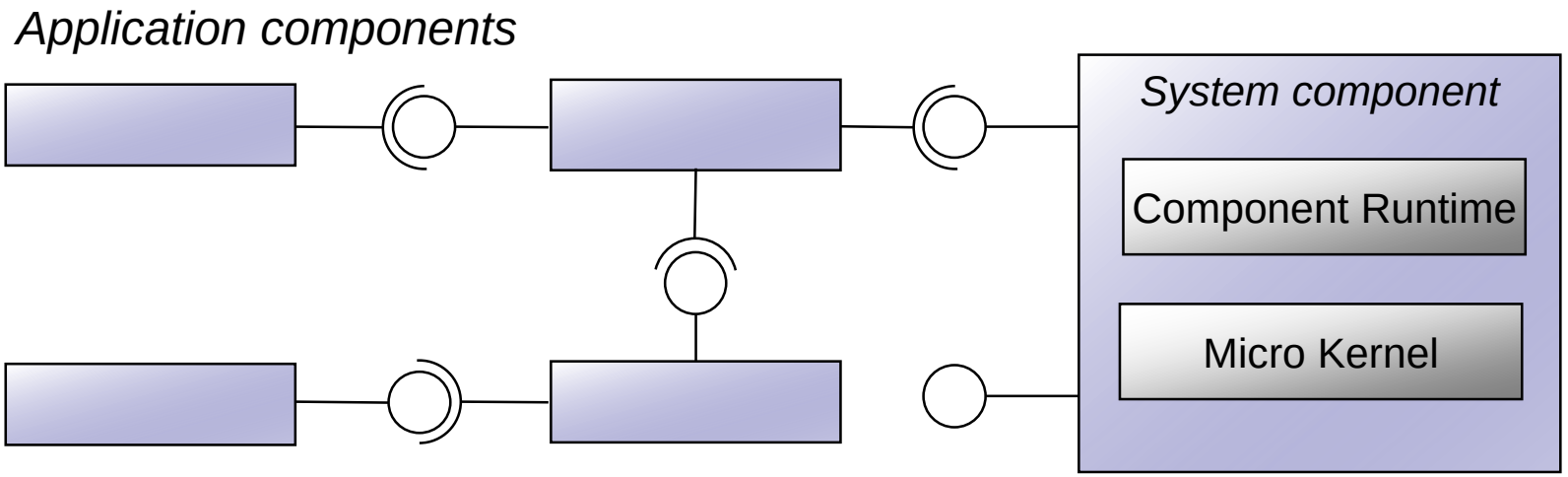
- Ultra-kompaktes Betriebssystem für massiv parallele Anwendungen
  - Fein granulare Call-Stacks
    - Millionen von Leichtgewichtsprozessen
  - Super schnelle Kontextwechsel
    - Fast so schnell wie Prozeduraufrufe
  - Software-instrumentierte Checkpoints
    - Ersatz für teure preemptive Kontextwechsel
  - Speicherverwaltung und Recycling
    - Mit hierarchischen Kompositionen statt Garbage Collection

Programmiersprache CL

- Zielsprache für das System
  - Fein-granulare Nebenläufigkeit
  - «Actor» Komponenten
  - Nachrichten-Kommunikation
  - Hierarchische Kompositionen
- Challenge für das System
  - Sehr viele Prozesse mit sehr viel Interaktionen

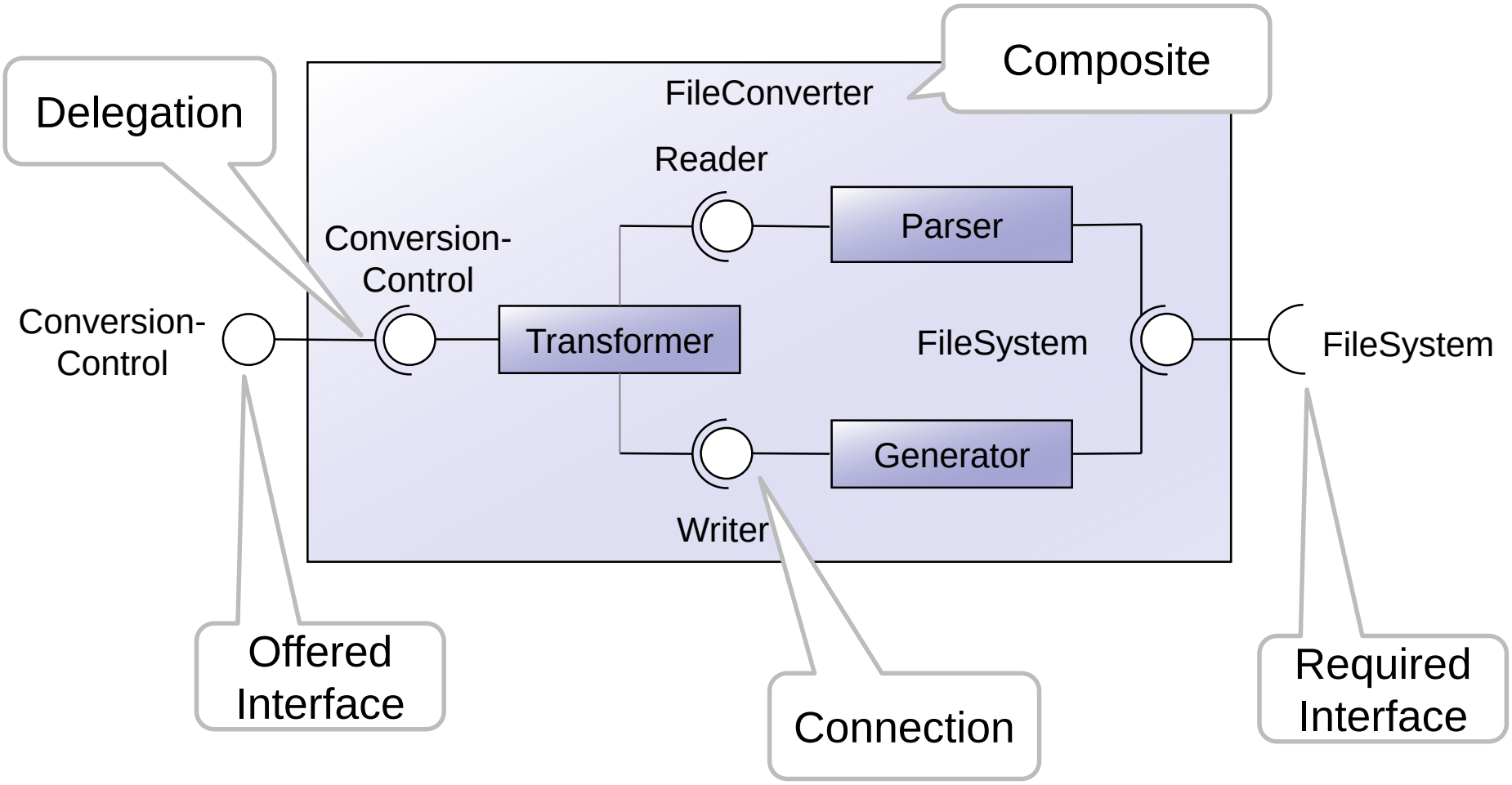
Composita: Übersicht

- System und Anwendungen als Komponenten
  - Mit Schnittstellen verbunden



  - Intel Multi-Core PC Machines
  - 256 KB Kernel-Grösse, Booten in ca. 1 Sekunde
  - Begrenzte #Driver: IDE Disks, Keyboard, Grafikkarte

Komponentenstrukturen

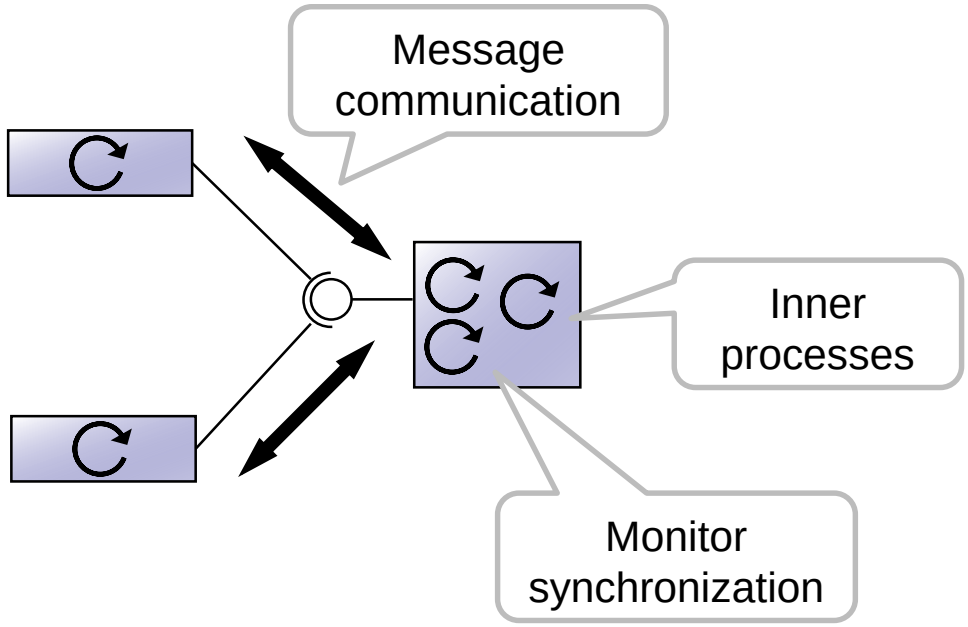


Komponentenmodell

- Composite kapselt innere Komponenten
  - Erzeugt, verbindet, löscht seine Sub-Komponenten
  - Beliebige Komponentenstruktur zur Laufzeit
- Offered und Required Interfaces
  - Kommunikations-Ports
  - Verbindbar falls gleicher Name
  - Nur umgebende Komponente setzt Verbindungen

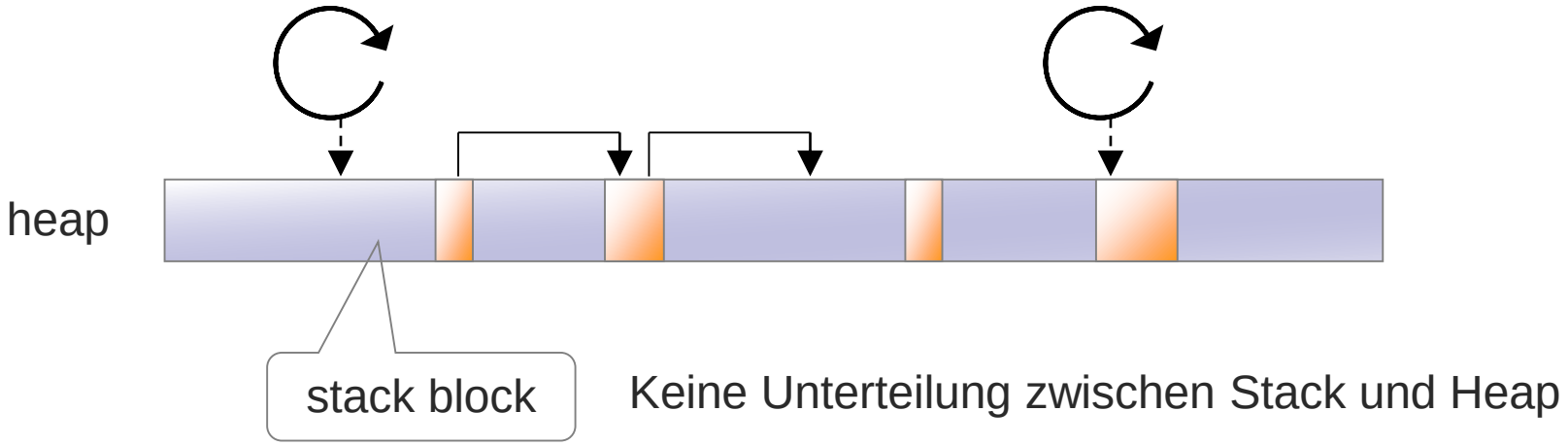
Nebenläufigkeitsmodell

- Prozesse innerhalb Komponente
  - Shared Memory in der Komponente
  - Monitor-Synchronisation: Exclusive, Shared Locks und generische Awaits
- Kommunikation zwischen Komponenten
  - Senden und Empfangen von Nachrichten
  - Formales Protokoll
  - Separate parallele Kommunikation zwischen je zwei Komponenten



Prozess-Stacks

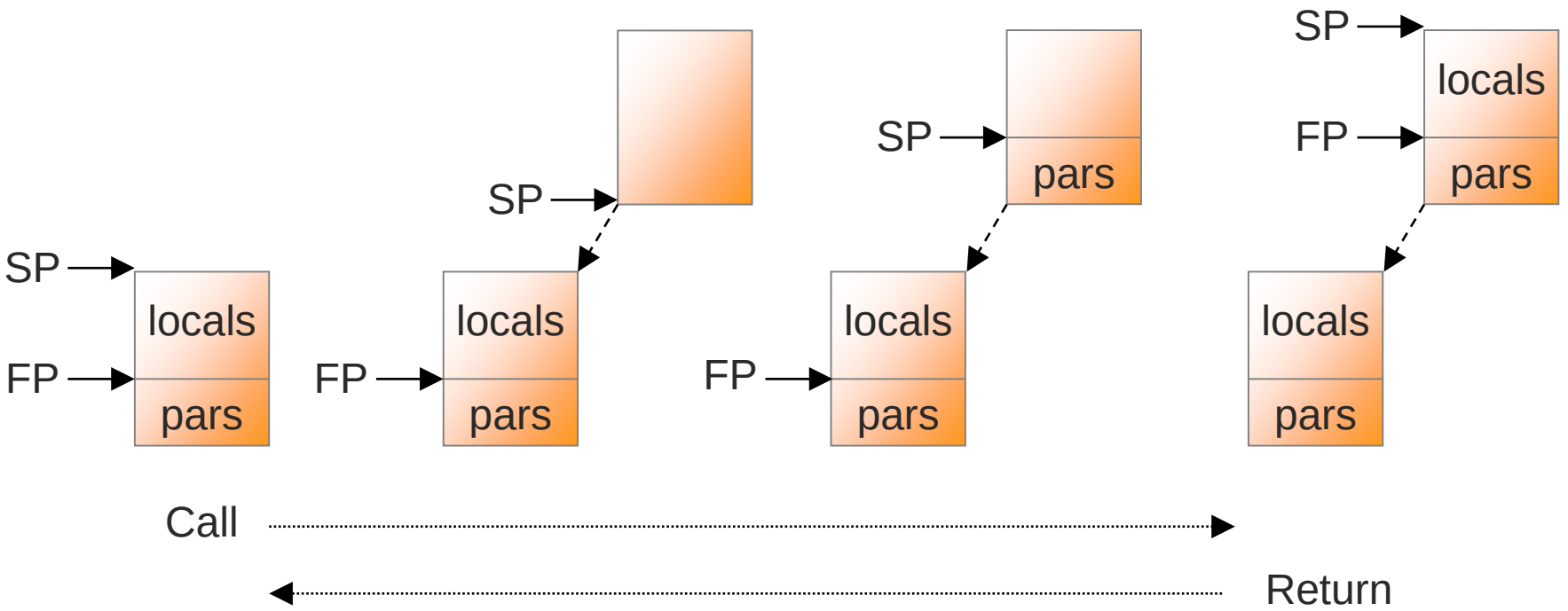
- Beliebig kleine Stacks
  - Nicht an Page-Granularität fixiert
- Stack = Liste von Heap-Blöcken
  - Dynamische Erweiterung und Verkürzung



- Stackgröße pro Prozedur durch Compiler bestimmt
  - Kommunikation statt Methoden: weniger Prozeduraufrufe

Stack-Erweiterung

- Erweiterung bei Prozeduraufruf, Reduktion bei Return
  - Compiler fügt Code bei Prozeduranfang und Ende ein



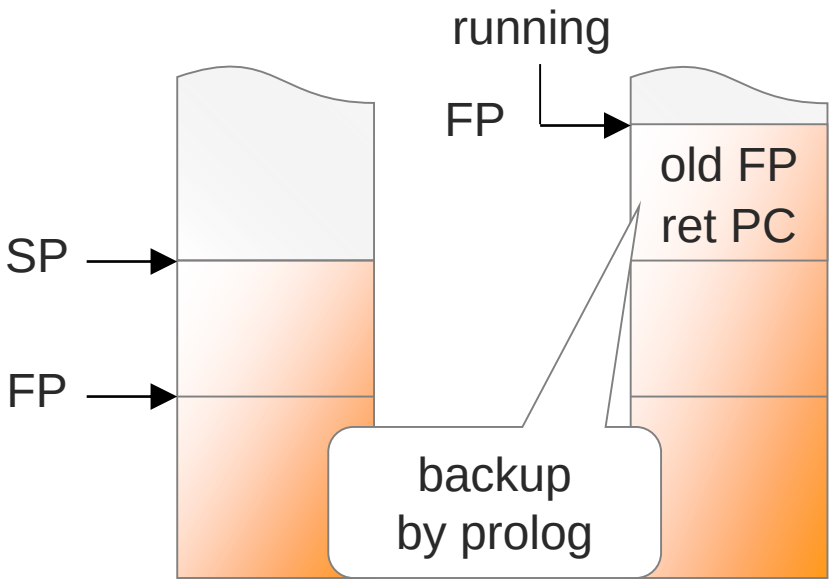
- Systemaufrufe und Interrupts
  - Auf Prozessor-assoziertem System-Stack („Run to Completion“)

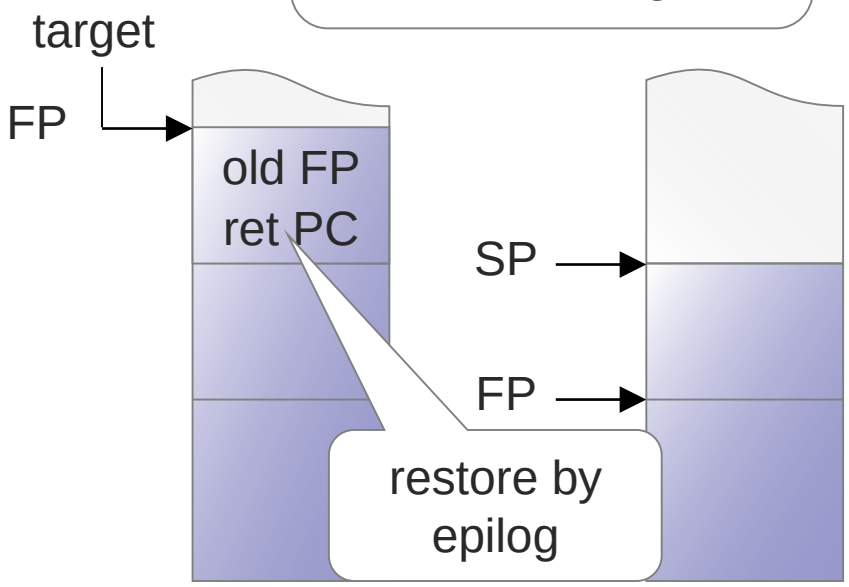
Synchrone Kontextwechsel

- Systemaufruf per normaler Prozeduraufruf
  - Kein SW-Interrupt
  - Keine Kernel-Protection wegen sicherer Sprache
- Direkter Switch zu Zielprozess

```
PROCEDURE Switch(target: Process);
BEGIN
  running := REGISTER.FP;
  REGISTER.FP := target
END Switch;
```

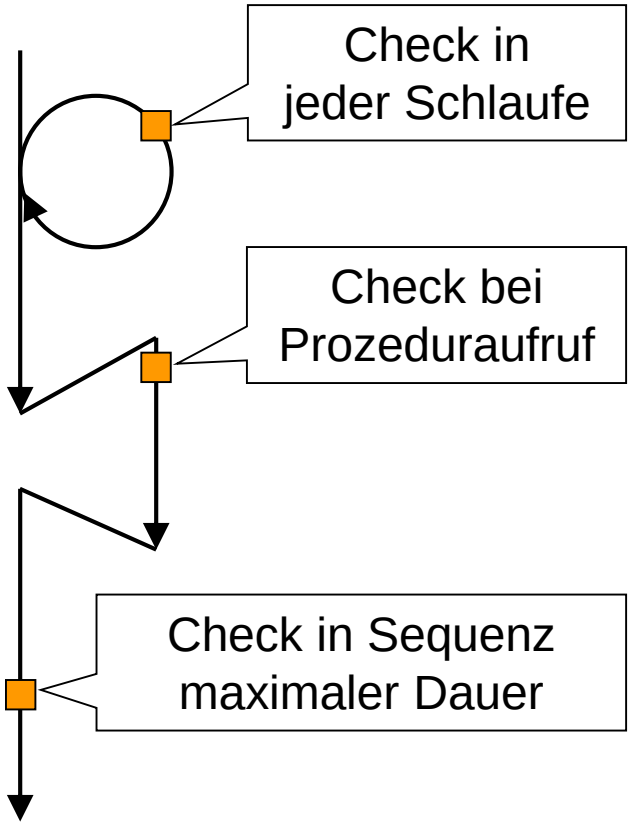
Zzgl. Synchronisation bei Manipulation der Warteschlangen





Alternative zu Preemption

- Compiler instrumentiert Checkpoints in Programmcode
  - Checks in Intervallen von garantierter maximaler Laufzeit
  - Checks initiieren Kontextwechsel nach Ablauf eines Zeitintervalls
  - Sichert nur die benötigten Register auf dem Stack
  - Stack-Platz schon vom Compiler reserviert



Register durch Timer-Interrupt gesetzt

```
IF Timeout THEN
 Switch(ready)
END
```

Aufruf sichert notwendige Register

Software Checkpoints

- Etwas schwächer als Preemption
  - Simuliert Preemption nach definierter maximaler Verzögerung
  - Kein kooperatives Multitasking nötig
- Reduziert Footprint pro Prozess
  - Kein unnötiger Backup-Platz pro Prozess
  - Sonst ein paar KB mit FPU, SSE.
  - Relevant für möglichst hohe Anzahl Prozesse

Checkpoint Kosten

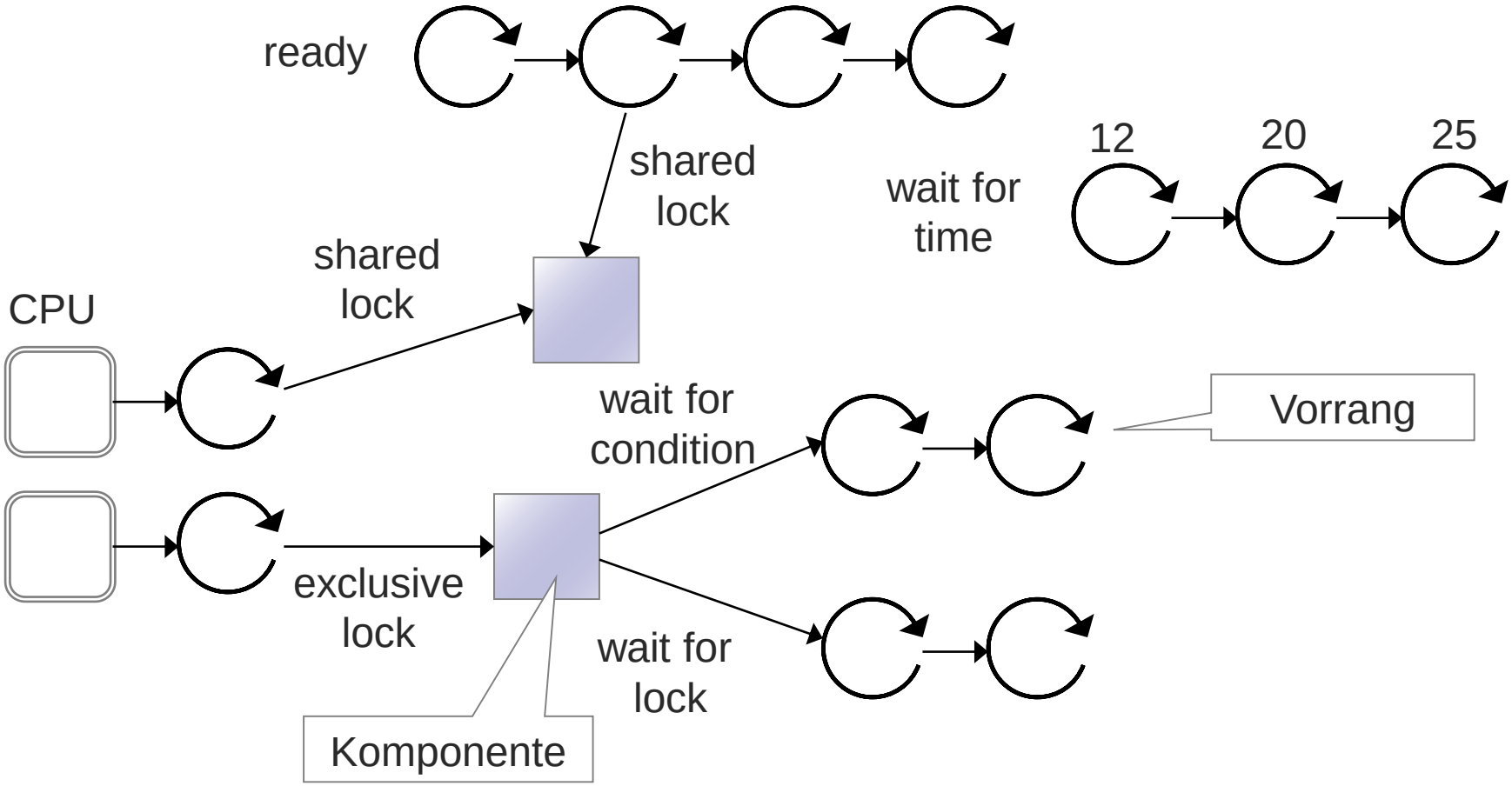
- Vernachlässigbar bei Szenarien mit häufigen Interaktionen
- Bis ca. 20% Overhead für Hot Loops

Program	**With checks**	**No checks**	**Overheads**
ProducerConsumer (1 producer, 1 consumer)	1011	1048	**-3.5%**
Eratosthenes (2..10000)	234	230	**1.9%**
TokenRing (1000 nodes)	266	273	**-2.7%**
SpinningProcesses (16 instances)	**181**	**151**	**19.6%**

In milliseconds, rounded to millisecond, average of 3 subsequent executions, Intel 2 Core i7 3520M, 2.9GHz, 8GB main memory.

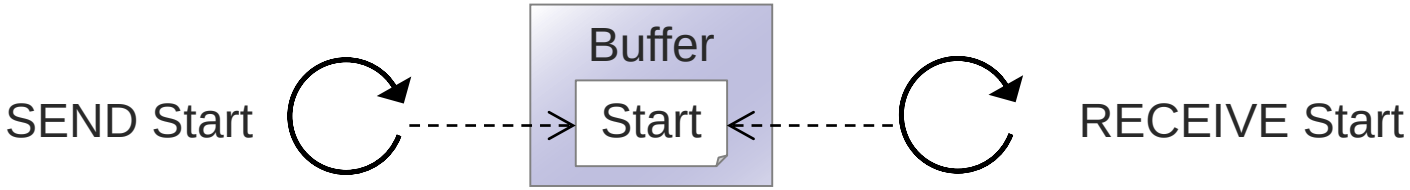
Prozess Scheduling

- Warten auf systemüberwachte Bedingung
  - Wechsel zu wartendem Prozess falls Bedingung erfüllt ist
  - Direkte Weitergabe des Locks (“Signal and Pass”)



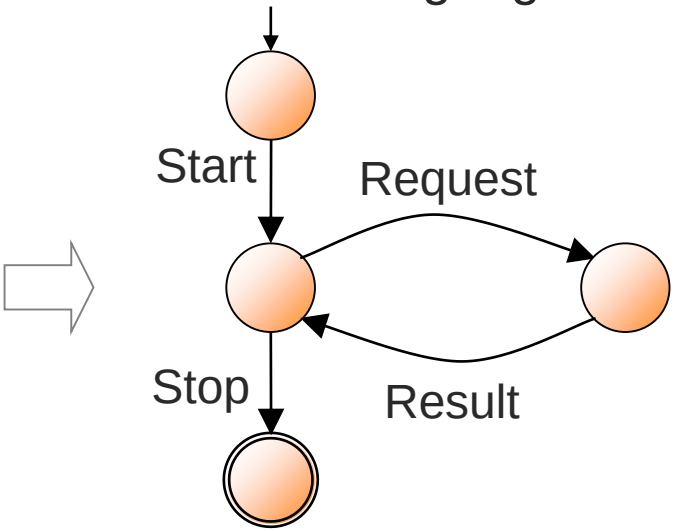
Kommunikationskanäle

- Nachrichtenaustausch zwischen zwei Prozessen
  - Automatisch synchronisierter interner Buffer
  - Maximale Nachrichtengröße in Protokoll bestimmt Buffergröße
  - CL benutzt kleine Buffer (max. 4 Nachrichten)



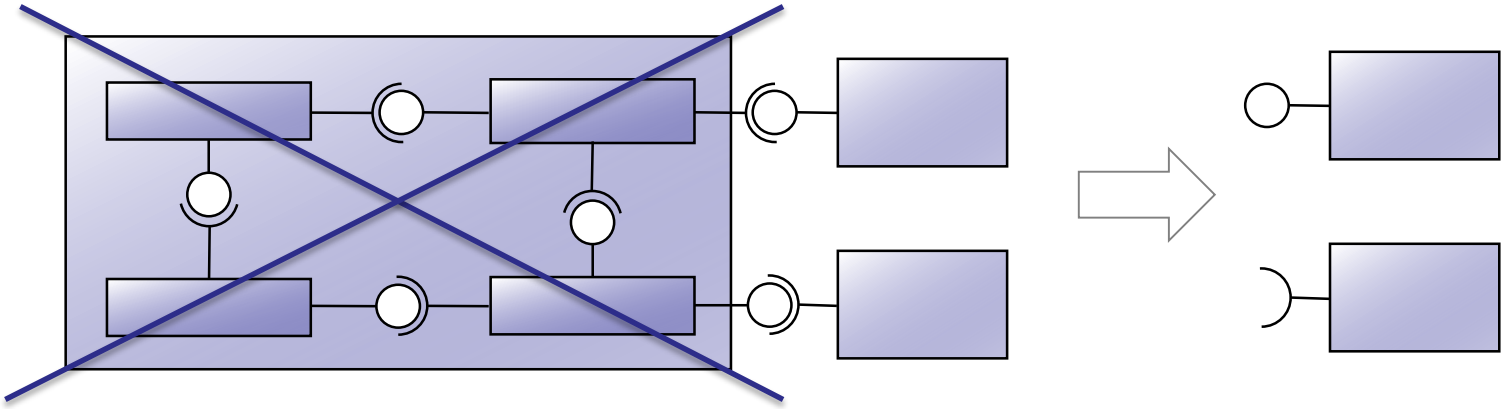
- Compiler generiert Zustandsmaschine für das Protokoll
  - Sender prüft das Protokoll mit Zustandsübergang

```
IN Start
{
  IN Request(id: TEXT)
  OUT Result(x: ANY)
}
IN Stop
```



Speicherverwaltung

- Drastisch vereinfacht wegen strikten Hierarchien
  - Klares Ownership wegen des Containments
  - Keine Erreichbarkeitserkennung per GC
- Deallokation von Komponente C
  1. Warten auf Beendigung der Kommunikationen mit C
  2. Verbindungen der Interfaces von C auflösen
  3. Rekursives Löschen der inneren Komponenten von C



Nebenläufige Skalierung

- Maximale Anzahl Threads / Leichtgewichtsprozesse

COMPOSITA	.NET (Win8)	JVM (Win8)
4,367,000	100,000	100,000

64 bit, 8GB main memory, .NET 4.5, Java 7

- Composita: Von #Prozesse abhängig
  - Verfügbarem Hauptspeicher
  - Speicher pro Komponente und dynamischer Stack

Nebenläufige Performance

Program (ms)	COMPOSITA	C#	Java
1 Producer - 1 Consumer	**1011**	5427	6020
1 Producer – 10 Consumer	**1327**	22324	26255
10 Producer – 10 Consumer	**10141**	40158	30513
Eratosthenes (2..1000)	**5**	31	31
Eratosthenes (2..10,000)	**235**	877	640
Eratosthenes (2..100,000)	**14594**	49216	38023
TokenRing (1000 nodes)	**266**	4500	4596
TokenRing (10,000 nodes)	**2822**	49,945	53,582
TokenRing (100,000 nodes)	**30,106**	518,956	1,163,088

In milliseconds, rounded to millisecond, average of 3 subsequent executions, Intel 2 Core i7 3520M, 2.9GHz, 8GB main memory. C# on .NET Framework 4.5, x64, with optimization compiler option, on Windows 8, Java 7 version 1.7.0.21, 64 bit server under Windows 8.

Schlussfolgerungen

- Moderne Laufzeitarchitektur für massive Parallelität
  - Frei von Design-Legacies
  - «Design from Scratch»
- Composita als Proof of Concept
  - Hohe Grad an Parallelität (Prozessen)
  - Sehr effiziente Prozess-Interaktionen
- Innovative Design-Konzepte
  - Micro-Stacks
  - Effiziente Kontextwechsel
  - Code-instrumentierte Preemption
  - Hierarchische Speicherstruktur