

# Skalierbare Betriebssystemkerne für Multiprozessorsysteme

Christian Brunner

[christian.brunner@mb.stud.uni-erlangen.de](mailto:christian.brunner@mb.stud.uni-erlangen.de)

# Motivation

Leistungszuwachs früher:

Taktrate steigern

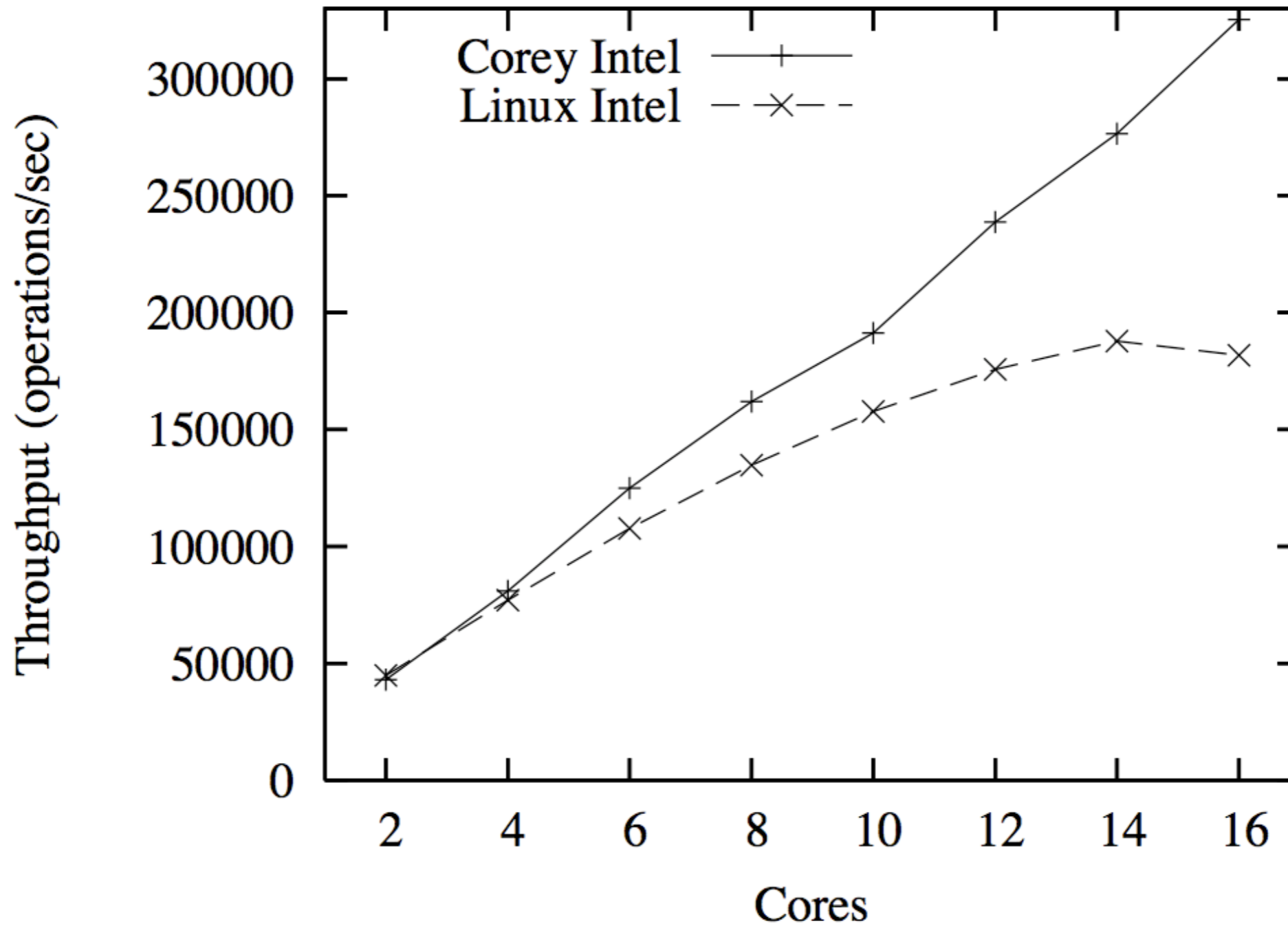
- ➡ Betriebssystem und Applikationen profitieren automatisch
- ➡ keine neuen Probleme

Leistungszuwachs heute:

Anzahl der Prozessorkerne pro System steigern

- ➡ Davon kann man nicht automatisch profitieren
- ➡ Zudem entstehen neue Probleme

# Motivation



# Agenda

- Grundlagen
  - ▶ Blockierende Synchronisation
  - ▶ Nicht-blockierende Synchronisation
- Skalierbare Betriebssystemkerne
  - ▶ Synthesis
  - ▶ Corey
- Fazit

# Grundlagen: Sperren

Synchronisation in gängigen Systemen

- Interrupts deaktivieren (nur im Einprozessorfall)
- Spinlocks (nur im Mehrprozessorfall)
- Semaphoren

# Grundlagen: Sperren

## Vorteile:

- leicht zu realisieren
- Effizienz bei wenig Parallelität

## Nachteile:

- Overhead, auch wenn es keine Konkurrenzsituation gibt
- Serialisierung der Ausführung, wenn alle Kerne auf den gleichen kritischen Abschnitt zugreifen wollen
- Gefahr von Deadlocks und unbeschränkter Prioritätenumkehr

# Grundlagen: nicht blockierende Synchronisation

Einteilung nach Herlihy:

1. wartefrei

jeder Faden endet in endlicher Zeit

2. nicht blockierend

mindestens ein Faden endet in endlicher Zeit

Wartefreiheit schließt Verhungern aus

➡ teuer zu realisieren

➡ Verhungern ist in Betriebssystemkernen tolerierbar

# Grundlagen: nicht blockierende Synchronisation

nicht blockierende Synchronisation behebt die Nachteile von Sperren

Realisierung: z.B. „Compare-and-Swap“ (CAS)

## Beispiel CAS2:

```
CAS2 (compare1, compare2, update1, update2, mem_addr1, mem_addr2) {  
    if (*mem_addr1 == compare1 && *mem_addr2 == compare2) {  
        *mem_addr1 = update1;  
        *mem_addr2 = update2;  
        return SUCCESS;  
    }  
    else  
        return FAIL;  
}
```



# Skalierbare Betriebssysteme

## Synthesis (1991)

- Ausschließlich nicht blockierende Synchronisation des Kernes
- keine Sperren

## Corey (2008)

- nicht blockierende Synchronisation und Sperren
- hohe Kontrolle der Anwendung über Systemressourcen

# Synthesis

- komplett nicht blockierend entwickelt
- Ziel: Nachteile von Sperren komplett umgehen:
  - Overhead
  - Flaschenhalse in Konkurrenzsituationen
  - Deadlocks
  - unbeschränkte Prioritätenumkehr

# Synthesis: Grundlagen

Herlihy's Ansatz, Datenstrukturen (DS) in wartefreie umzuwandeln:

- DS kopieren und manipulieren
- Zeiger, der auf die alte DS zeigt, auf die neue DS umschreiben

Für Synthesis nicht geeignet, da zu viel Overhead

# Synthesis: Grundlagen

## Lösungen:

- Information in ein oder zwei Wörter und mit CAS/CAS2 manipulieren
- nicht blockierende LIFO, FIFO oder verkettete Listen
- kodierter Systemstatus
- kritische Abschnitte reduzieren bzw. kürzen
- kritische Abschnitte aufteilen

# Synthesis: Quajects

## Quajects:

- Systembausteine von Synthesis
- Codefragmente mit Datenstrukturen

## Beispiele:

### Abstrakte Datenstrukturen

➡ Stapel, Warteschlange, verkettete Liste

### Betriebssystemabstraktionen

➡ Thread, Speichersegment, E/A-Geräte

# Synthesis: Stapel

```
void Push (elem) {
    retry:
        old_SP = SP;
        new_SP = old_SP - 1;
        old_val = *new_SP;
        if (CAS2(old_SP,old_val,new_SP,elem,&SP,new_SP) == FAIL)
            goto retry;
}
elem Pop () {
    retry:
        old_SP = SP;
        new_SP = old_SP+1;
        elem = *old_SP;
        if (CAS(old_SP,new_SP,&SP) == FAIL)
            goto retry;
        return elem;
}
```

# Synthesis: Scheduling

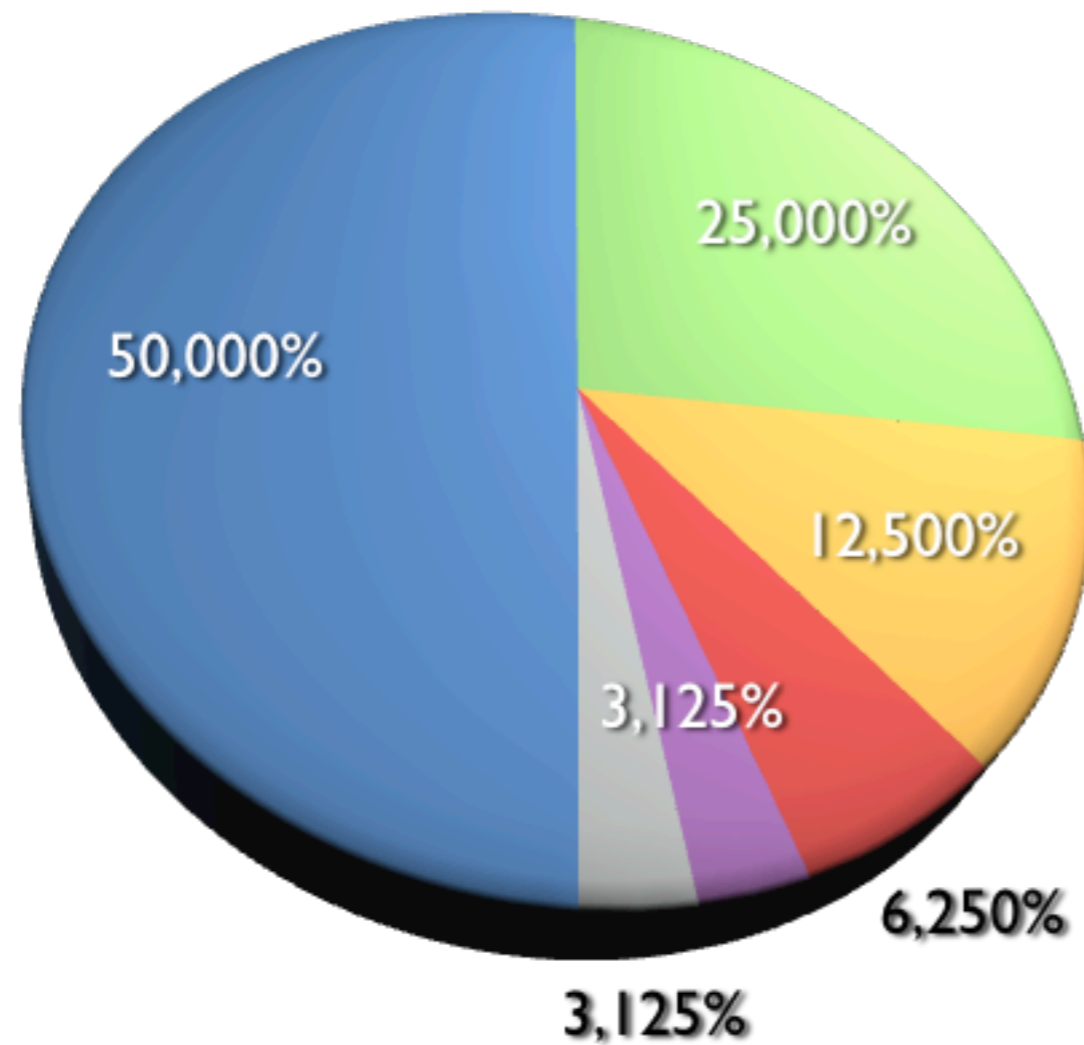
Jeder Faden ist durch einen Thread Table Entry (TTE) beschrieben

- Ein- / Auslagerungsroutine
- Fadenkontext

Feste Prioritäten

Zugewiesene Prozessorzeit pro Priorität fällt exponentiell

# Synthesis: Scheduling



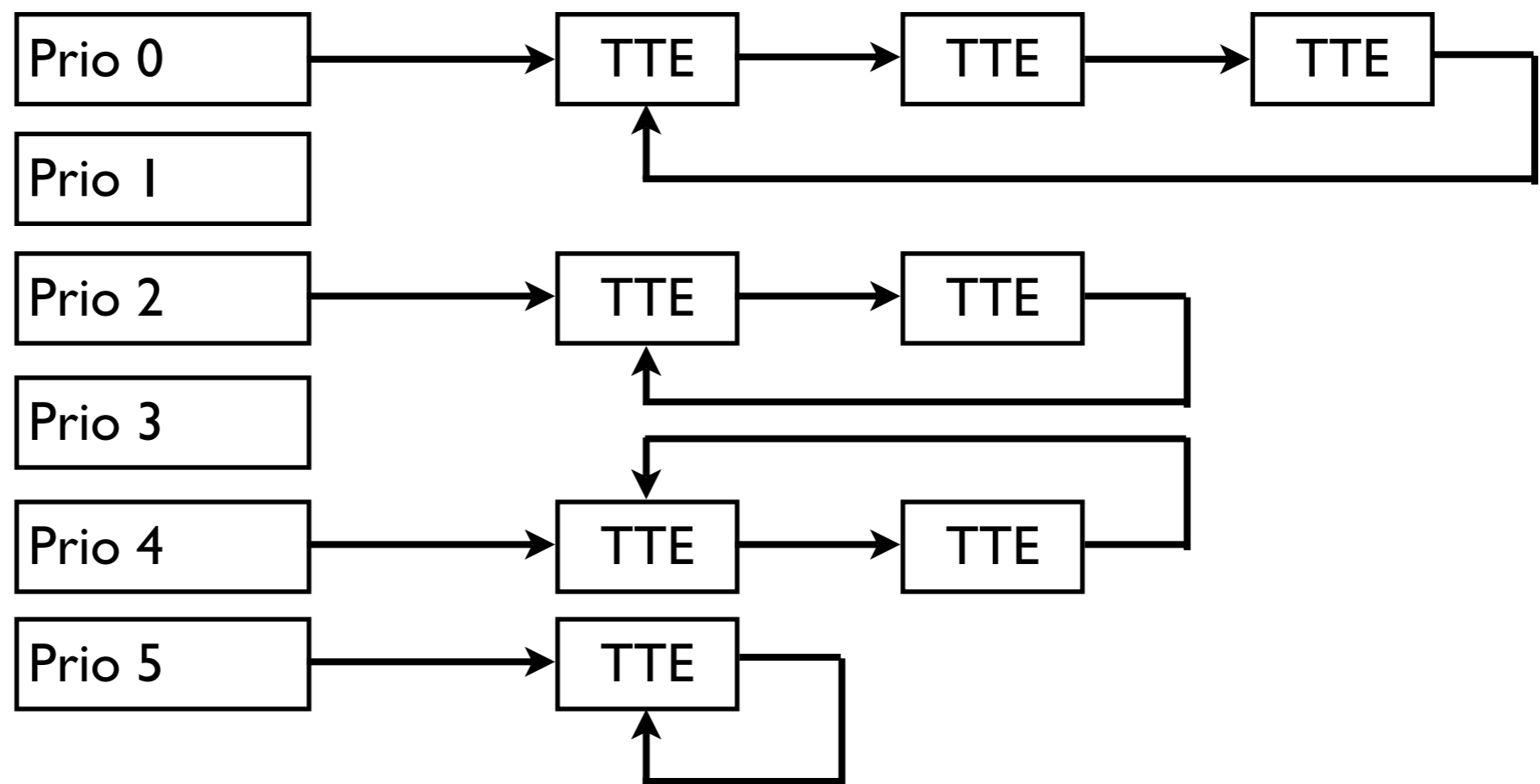
- Priorität 0
- Priorität 1
- Priorität 2
- Priorität 3
- Priorität 4
- Priorität 5

$$t = \frac{1}{2^{\text{prio}+1}}$$



# Synthesis: Scheduling

0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4
0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	5
0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4
0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	5



# Corey

- nicht ausschließlich blockierende Synchronisation verwendet
- Im Vordergrund: Konfigurierbarkeit des Kerns durch Applikation
- Basiert auf dem Prinzip des Exokernels
  - möglichst wenige Abstraktionen
  - Schutz der Hardware
- Weitergehende Funktionalität durch Bibliotheksbetriebssystem
- Alle Kerneldatenstrukturen sind zunächst kernlokal

# Corey: Abstraktionen

## **Freigaben (shares)**

Freigabe von Kernelressourcen zwischen Kernen durch Applikation bestimmt

## **Adressbäume (address trees)**

Freigabe von Daten zwischen Kernen kann nur von der Applikation ausgehen

## **Kernelkerne (kernel cores)**

Auslagerung der Kernels auf einen eigenen Kern

Systemaufrufe funktionieren dann über IPC mit gemeinsamen Speicher

# Corey: Bibliotheksbetriebssystem

bietet Funktionalität, die der Coreykern nicht zur Verfügung stellt

- Corey bietet Hardwarezugriff z.B. auf Netzwerkschnittstellen
- Protokollstapel für die Kommunikation muss das Bibliotheksbetriebssystem bereitstellen
- `cfork`: orientiert sich an `UNIX-fork`
  - Ausführung eines Kindprozess auf einem anderen Kern
  - Kind- und Elternprozess teilen im Normalfall wenig Daten
  - Kindprozess sieht Wurzelprozessbaum des Elterprozesses mit „copy-on-write“ Markierung

# Corey: Kernverwaltung

Optionale Abgabe der Kontrolle über Prozessorkerne an die Applikation

- garantierte Verfügbarkeit von Prozessorzeit
- keine Kontextwechsel
- Verwirklichung einer eigenen Schedulingstrategie durch das Bibliotheksbetriebssystem

Spezielles Szenario: Mindestens so viele Kerne wie Prozesse

- Scheduling uninteressant
- überhaupt keine Kontextwechsel mehr

# Corey: Geräteverwaltung

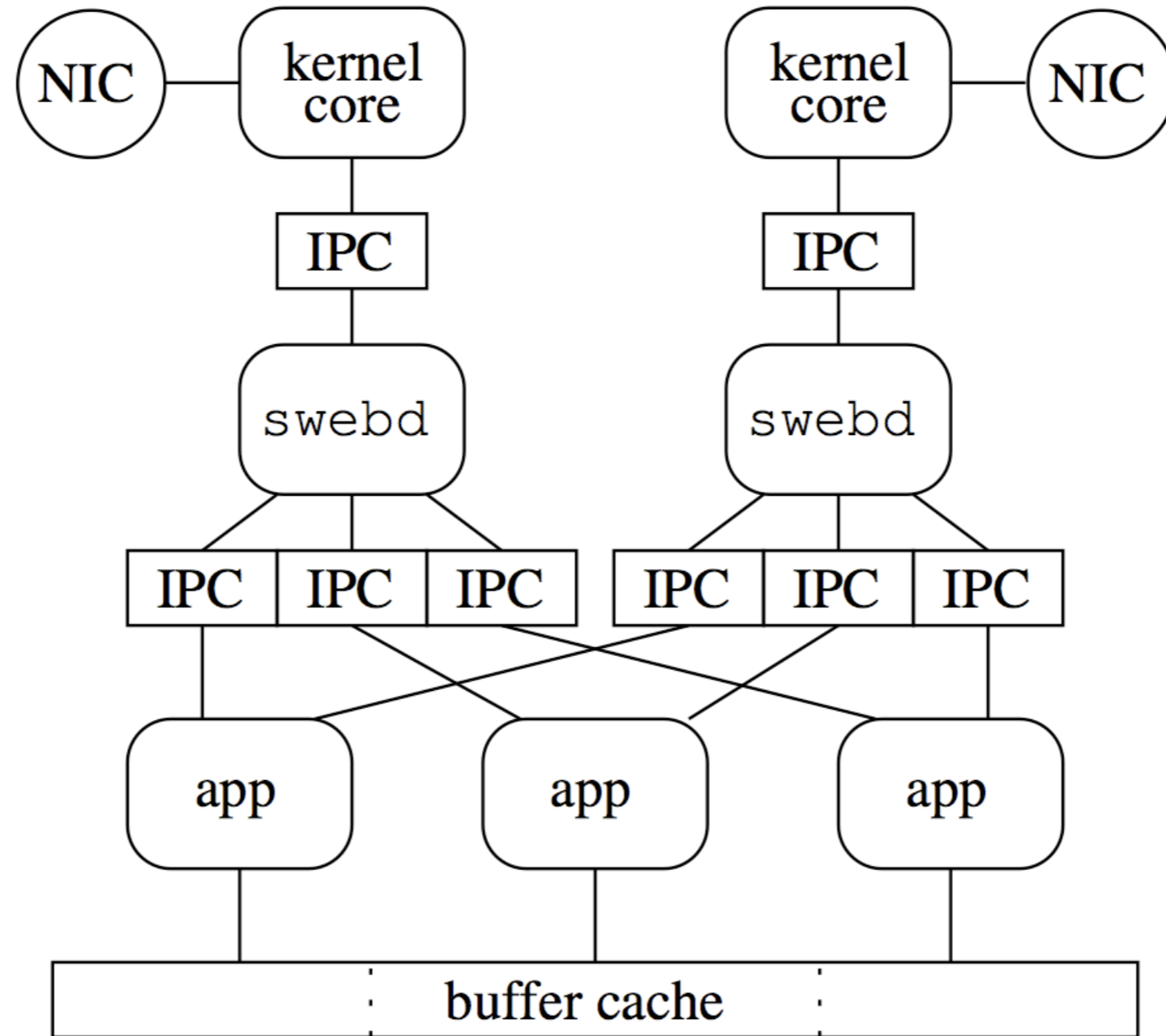
- Bereitstellung einer Liste von Geräteobjekten durch den Kern
- exklusiver Zugriff gewährleistet
- Kommunikation mit dem Geräte über Segmentfreigaben
- Virtualisierung weiterer Geräte, wenn nötig

# Corey: Benchmarks

## Benchmarks zur Leistungsanalyse

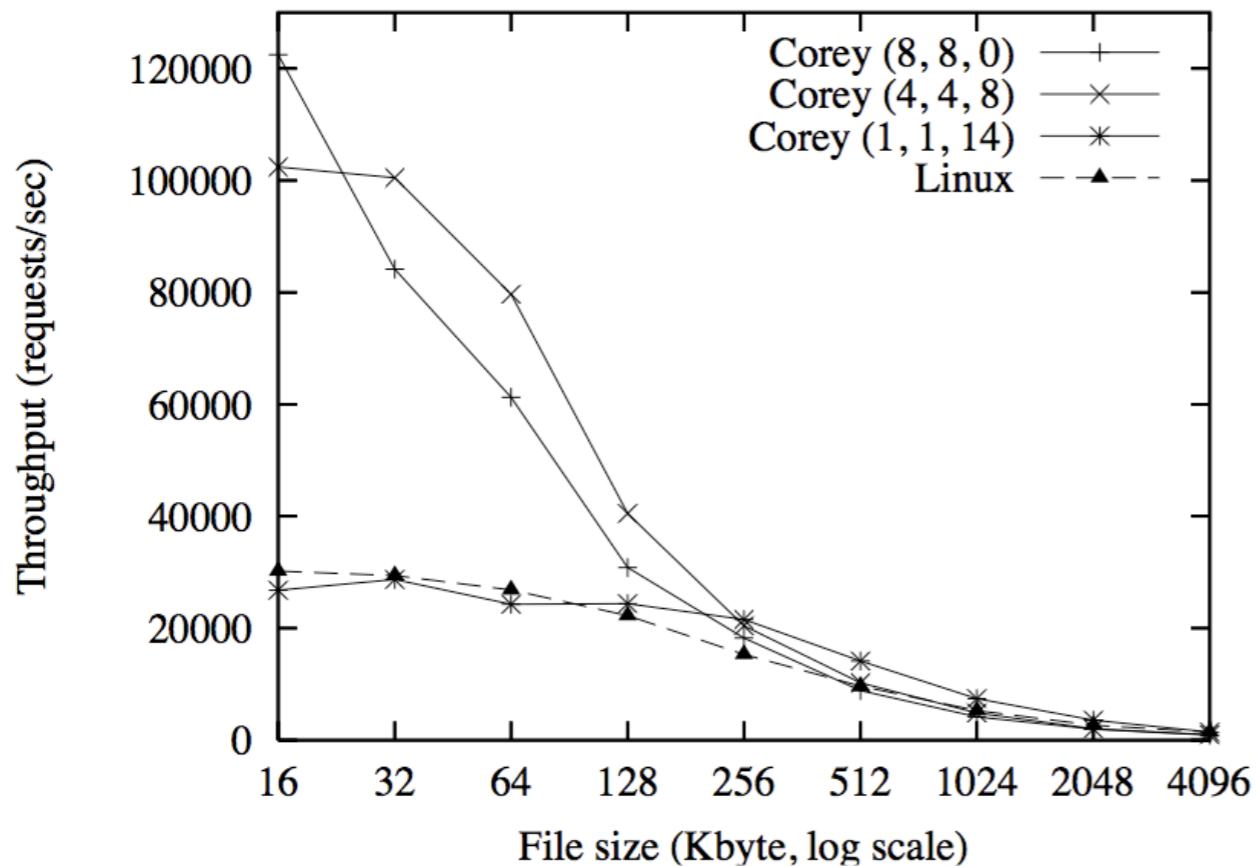
- Webserver
- TCP-Benchmark
- MapReduce

# Corey: Webserver

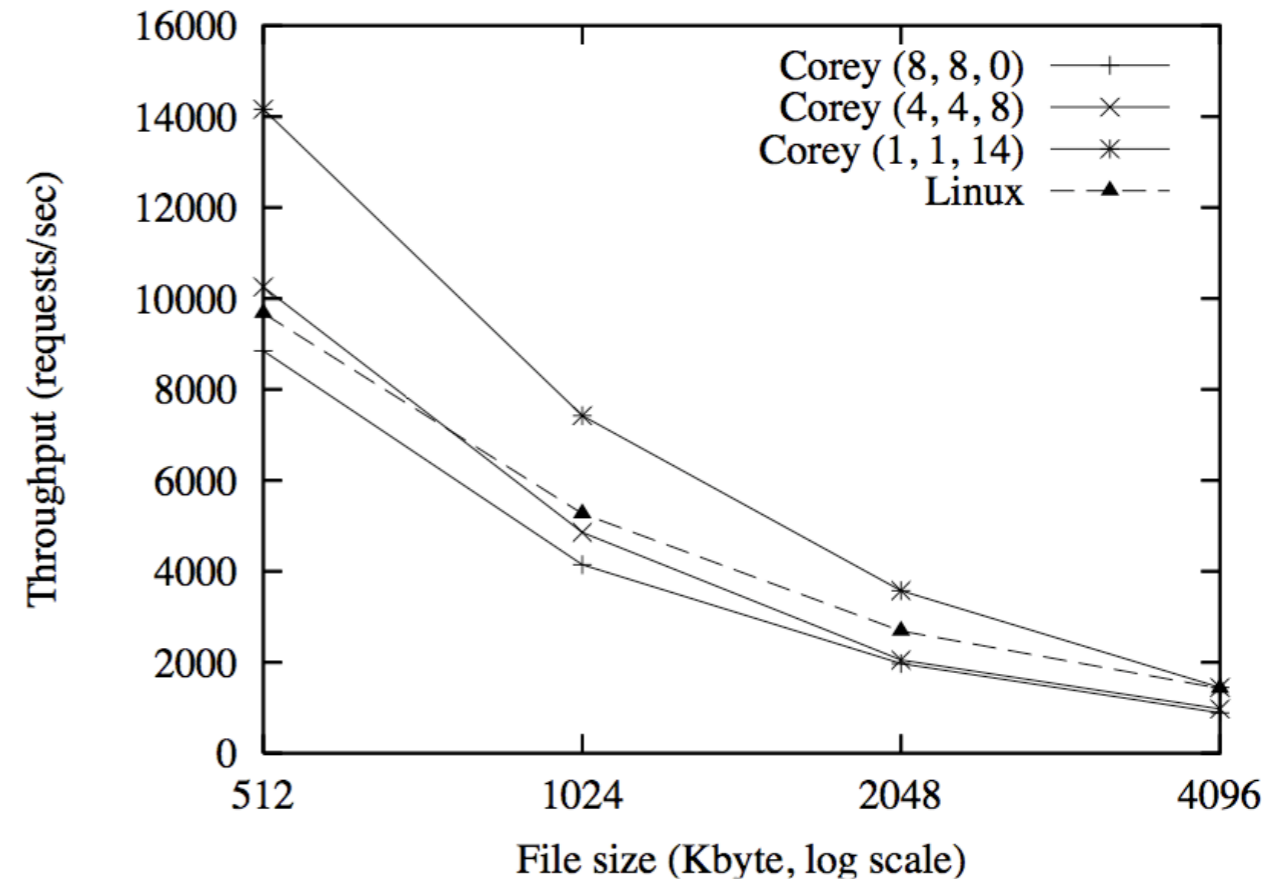




# Corey: Webserver



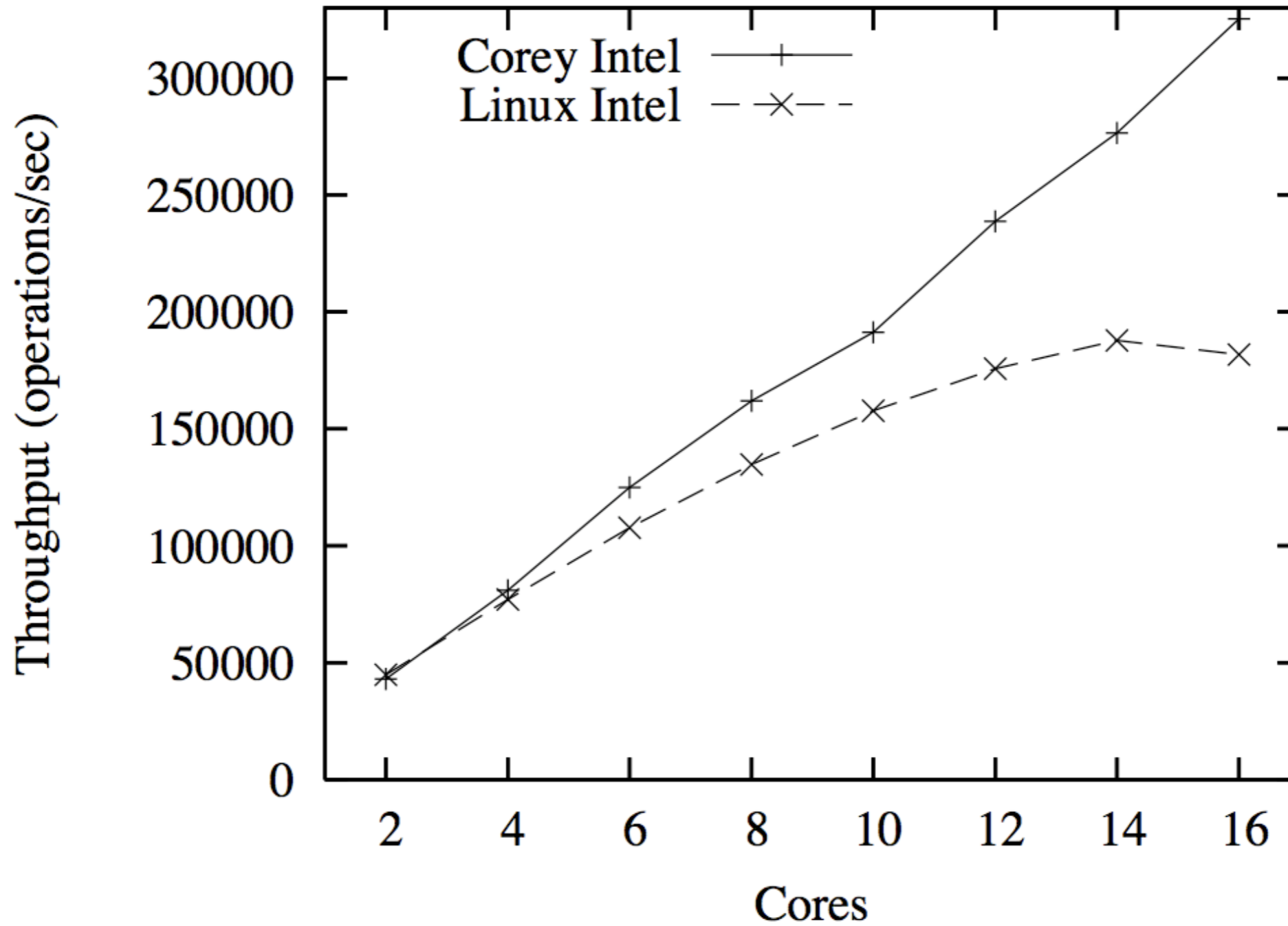
(a) All results.



(b) Larger file size results.

Corey (8,8,0)  $\triangleq$  8 Swed-Kerne, 8 Kernelkerne, 0 App-Kerne

# Corey:TCP-Benchmark



# Corey:TCP-Benchmark

Vorraussetzung: Pro Kern eine Netzwerkschnittstelle

Grund für die gute Skalierbarkeit von Corey

- Jeder Kern hat eine eigene Schnittstelle
  - ➡ Die Kerne beeinflussen sich nicht

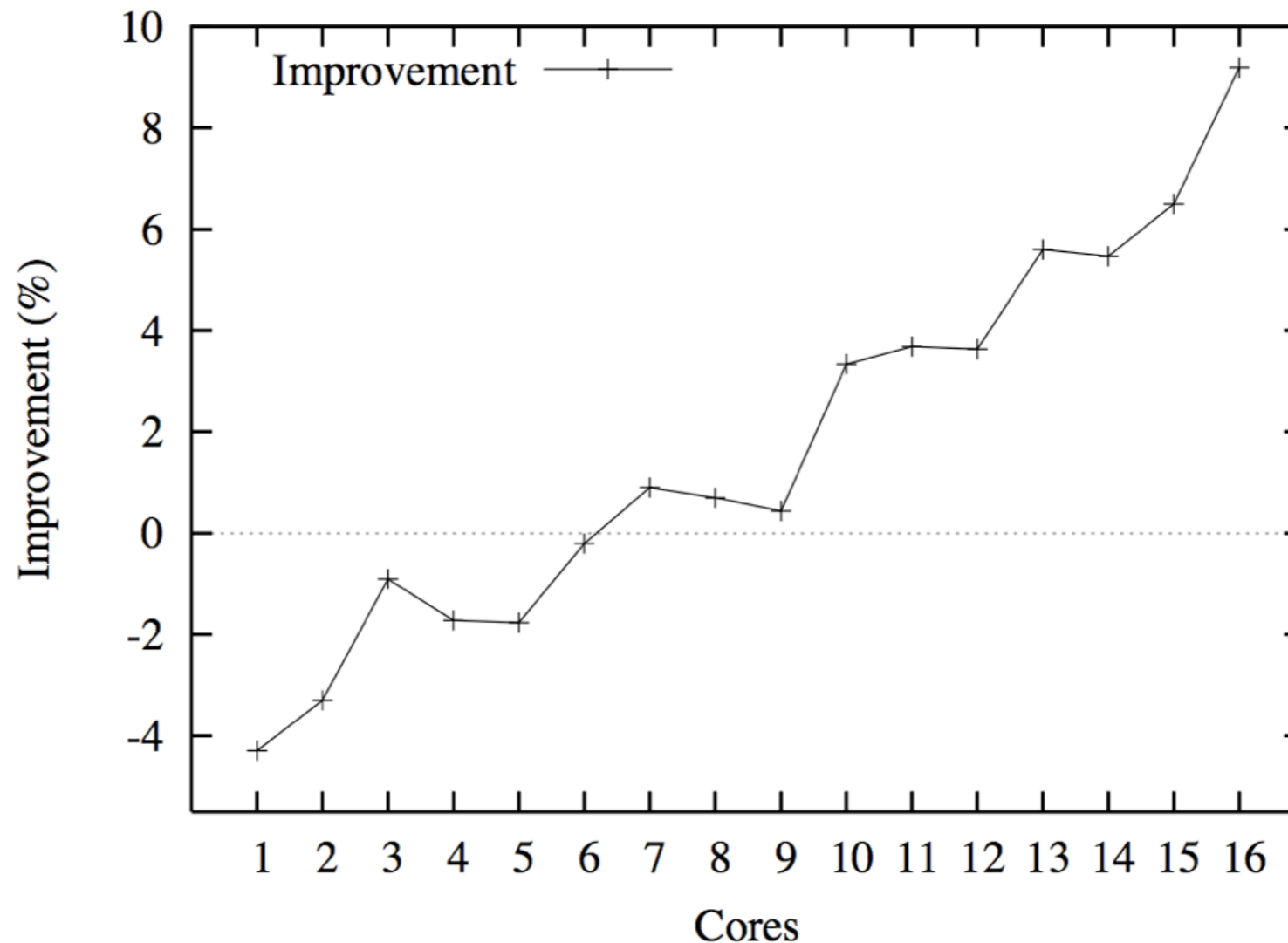
Grund für die schlechte Skalierbarkeit von Linux

- Globale Spinlocks beim Erstellen und Freigeben von Sockets  
`inode_lock` und `dcache_lock`

# Corey: MapReduce

MapReduce: Framework für parallele Berechnungen über große Datenmengen.

Konkretes Beispiel: wordcount über eine 1GB große Datei



# Corey: MapReduce

Grund für den Vorsprung von Corey bei vielen Kernen:

Linux:

- 11% der map-Phase und
- 17% der reduce-Phase

in `smp_invalidate_interrupt`

➡ TLB-Einträge werden geleert

➡ „remote TLB shutdown“ durch gemeinsamen Speicher

Corey verhindert das durch kernlokale Ressourcen

# Fazit

- Neue Prozessoren: mehr Kerne statt mehr GHz
  - ➡ Skalierbarkeit durch immer mehr Kerne pro Prozessor immer wichtiger
  - ➡ Was helfen z.B. 100 Kerne, wenn (wie beim TCP-Benchmark) mehr Kerne ab einer gewissen Anzahl sogar einen Leistungsrückgang bedeuten
- „Neue“ Konzepte sind gefragt, z.B.
  - nicht blockierende Betriebssystemkerne
  - höhere Kontrolle der Applikation über Ressourcen
- Problem: Es gibt nicht DEN richtigen Weg

**Danke für die Aufmerksamkeit!**