IT-Infrastruktur

WS 2014/15

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz D'':

v1.0, 2014/11/27

- Rechnerstrukturen, Teil 3

Dieser Foliensatz

Vorlesungsübersicht

Seminar

Wiss. Arbeiten

Datenformate und Wandlung

PC als Arbeitsplatz

Ergonomie und Arbeitsschutz

Folien D

Rechnerstrukturen

(Telekommunikation)

Infrastruktur-Technologie

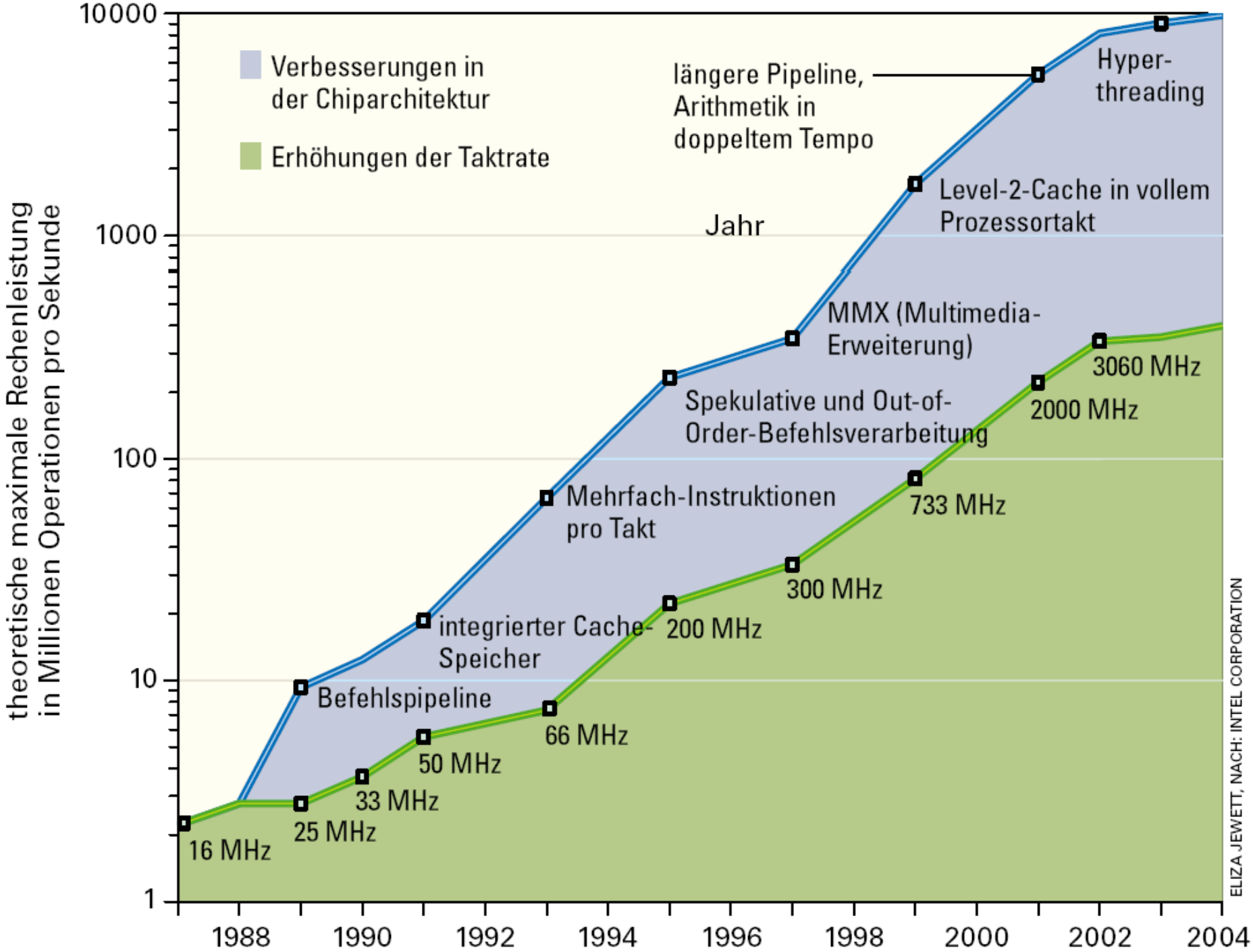
Zentrale / verteilte IT-Infrastrukturen

Mehrprozessor- und Multi-Core-Systeme

Ausgangssituation

- Uni-Prozessoren, ohne Pipeline
- Beschleunigen:
  - Prozessortakt (hat Grenzen)
  - Pipelining, skalar und superskalar (hat auch Grenzen)
  - mehr Leistung nur noch durch echte Parallelität erreichbar, also:

    mehr als eine CPU
    → **Multiprozessor- und Multi-Core-Systeme**


Verbesserungen in der Chiparchitektur
Erhöhungen der Taktrate
theoretische maximale Rechenleistung in Millionen Operationen pro Sekunde
10000
1000
100
10
1
Jahr
1988
1990
1992
1994
1996
1998
2000
2002
2004
längere Pipeline, Arithmetik in doppeltem Tempo
Hyper-threading
Level-2-Cache in vollem Prozessortakt
MMX (Multimedia-Erweiterung)
Spekulative und Out-of-Order-Befehlsverarbeitung
Mehrfach-Instruktionen pro Takt
integrierter Cache-Speicher
Befehlspipeline
16 MHz
25 MHz
33 MHz
50 MHz
66 MHz
200 MHz
300 MHz
733 MHz
2000 MHz
3060 MHz
ELIZA JEWETT, NACH: INTEL CORPORATION

Parallelisierung

- Einfaches Problem: zehn unabhängige Aufgaben parallel bearbeiten
  - z. B.: zehn separate Rechner einsetzen, perfekt parallelisierbar
- Schwierigeres Problem: eine komplexe Aufgabe parallel bearbeiten
  - wie aufteilen? Automatismus?

Parallele Architekturen

- Cluster: mehrere unabhängige Rechner, nur durch Netzwerk verbunden
- Multi-Prozessor: mehrere CPUs auf Hauptplatine
- Multi-Core: mehrere vollwertige CPUs-Kerne in einem CPU-Chip
- Hyper-Threading: mehrere logische CPUs in einem CPU-Chip (auch kombinierbar mit Multi-Core)

Hyper-Threading (HT)

- hardwareseitiges Multithreading (Intel)
- mehrere vollständige Registersätze und ein komplexes Steuerwerk
- parallel arbeitende Pipeline-Stufen (aber genauso viele Ausführungseinheiten wie in „normaler“ CPU)
- aus BS-Sicht: mehrere (virtuelle) CPUs
- mehrere parallele Befehls- und Datenströme (Threads) werden auf diese parallelen Stufen verteilt
- (erhöht die Anzahl *unabhängiger* Instruktionen in der Pipeline)

Multi-Core-System

- mehrere CPUs auf einem Chip
- alles mehrfach vorhanden (außer L2 Cache und höher sowie Bus)
- aus BS-Sicht: mehrere (echte) CPUs
- aktuell üblich: 2-/4-/6-/8-/12-/16-Core
- Beispiele:
  - AMD Opteron 16-Core
  - Intel Tera-Scale, Teraflops Research Chip (Polaris, 80 Cores)

Mehrprozessorsystem

- mehrere CPUs auf einem Mainboard
- weniger effiziente (ältere) Variante von Multi-Core-Systemen
- auch hier aus BS-Sicht: mehrere echte CPUs

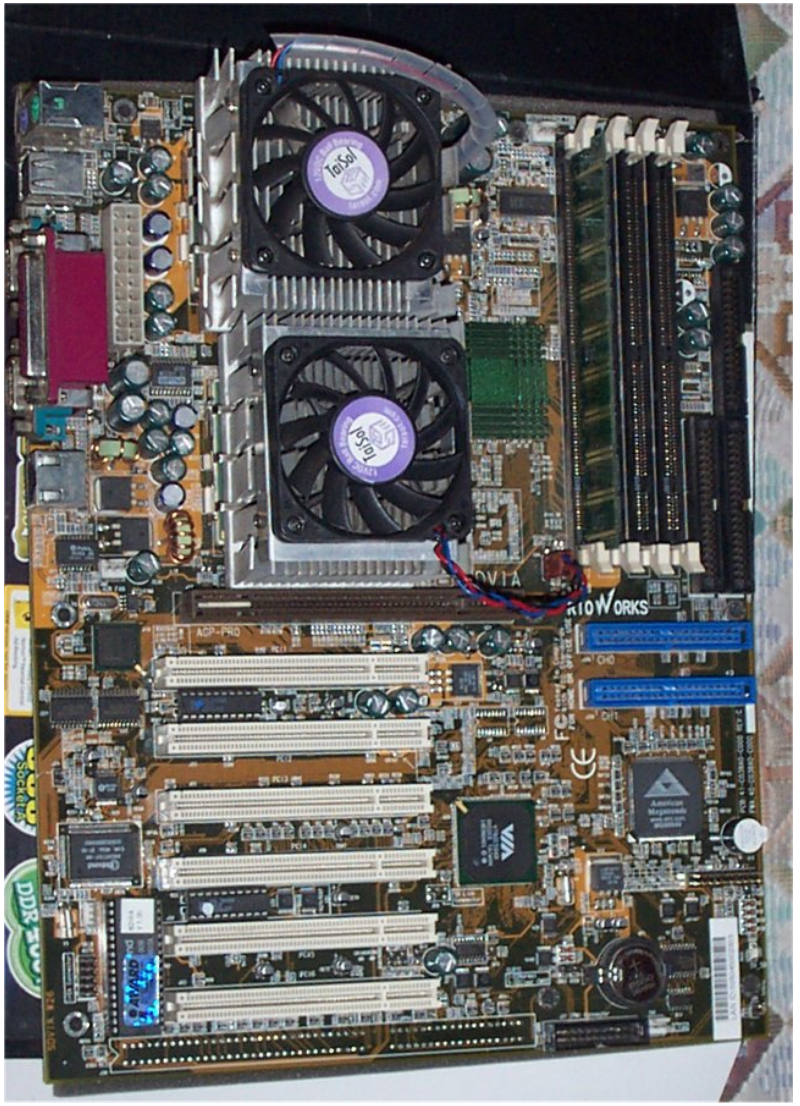


Bild: Wikipedia, http://de.wikipedia.org/wiki/Mehrprozessorsystem

Cluster

- mehrere Rechner mit je einer oder mehreren CPUs
- lokaler Speicher in jedem Rechner
- „lose gekoppeltes System“
- verteilte Anwendungen, die gleichzeitig auf mehreren Rechnern arbeiten
- Austausch zwischen Rechnern über Netzwerk

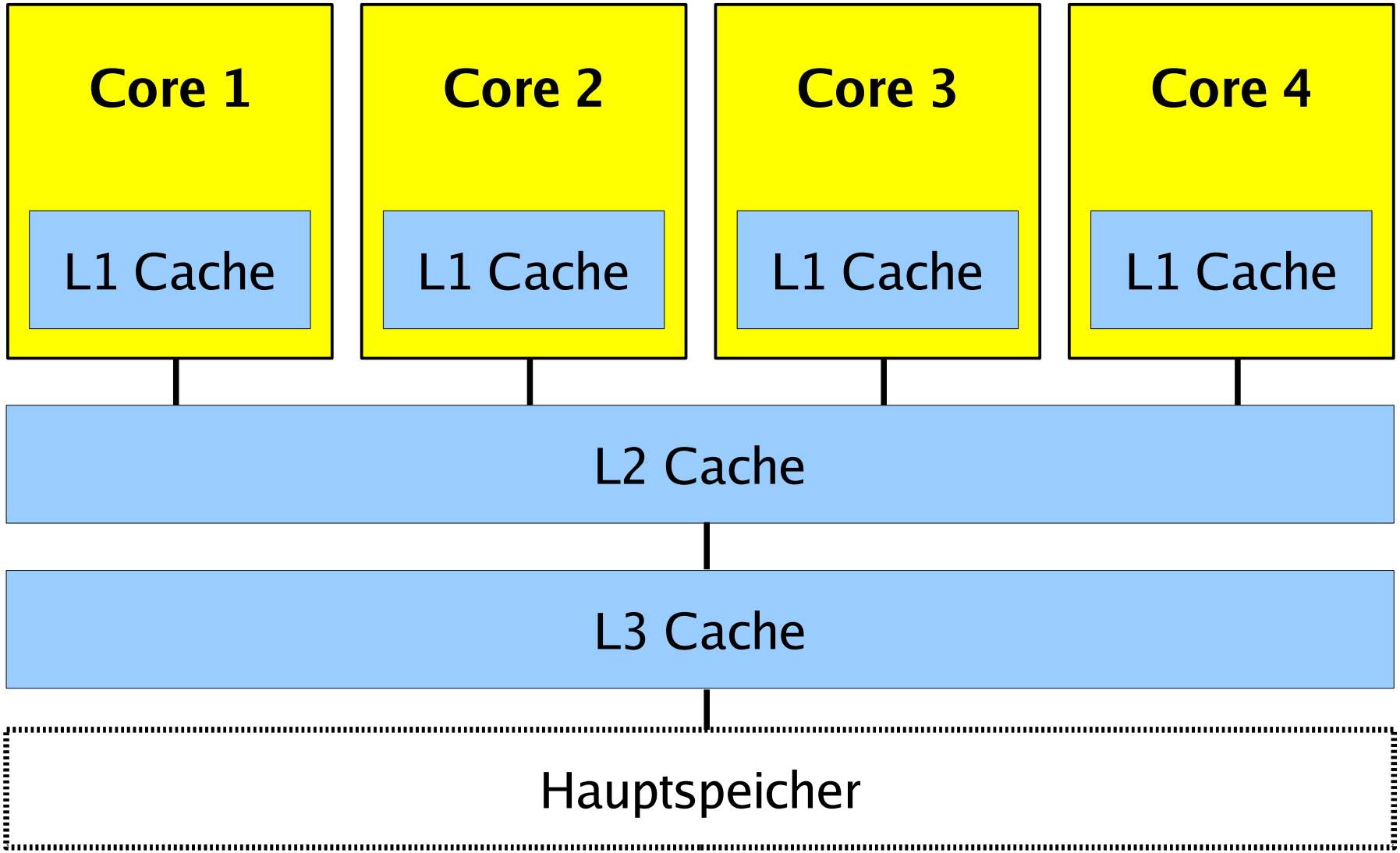
Cluster: MOSIX

- MOSIX2: http://www.mosix.org/
- Linux-basierter Cluster mit
  - automatischem Load-Balancing
  - Prozess-Migration
  - migrierbare Sockets
  - technisch: Virtualisierungsschicht
- am besten für Anwendungen mit wenig I/O geeignet

Speicher & HT, Multi-Core/-CPU

- Zugriff auf (gemeinsamen!) Hauptspeicher
  - Anbindung an RAM über gemeinsam genutzten Bus
  - Was tun bei parallelen Zugriffen auf den Hauptspeicher?
  - Parallel Write: Wer setzt sich durch?
- Cache
  - interner Cache: was tun, wenn mehrere (echte oder virtuelle) Kerne die gleiche Speicheradresse cachen?
  - Stichworte: Cache-Kohärenz, Cache-Konsistenz

Cache-Hierarchie bei Multi-Core



Cache-Kohärenz (1)

- konsistente Daten in allen Caches
- Beispiel:
  - CPU 1 liest Mem[x] und speichert Cache-Line im lokalen CPU-Cache
  - CPU 2 liest auch Mem[x] und speichert Cache-Line im lokalen CPU-Cache
  - CPU 1 schreibt Mem[x] und aktualisiert dabei auch den lokalen Cache
  - CPU 2 liest Mem[x] – was steht im lokalen Cache?

Cache-Kohärenz (2)

- Idee: Zu jedem Zeitpunkt ist ein bestimmter Wert Z für eine Speicherzelle Mem[x] gültig (der zuletzt geschriebene)
- Jeder Prozessor, der Mem[x] liest, sollte Z erhalten
- Unmittelbar nach Schreiben von Mem[x] muss man eine kurze Verzögerung akzeptieren, in der es „unterschiedliche Meinungen“ über Mem[x] gibt

Cache-Kohärenz (3): Protokolle

- Cache-Kohärenz-Protokolle garantieren Kohärenz
- zwei Ansätze
  - Verzeichnis: zentrale Liste mit dem Status aller Cache-Lines (in allen Caches)
    - Liste der CPUs mit Read-only-Kopie (Status *Shared*)
    - CPU mit exklusivem Schreibzugriff (Status *Exclusive*)
  - **Snooping:** Alle Cache Controller lauschen auf Speicherbus und erkennen Schreib- und Lese-zugriffe auf Cache-Line, die sie auch speichern

MESI Cache Coherence Protocol (1)

- Ziel: Verwalten, wo der aktuell(st)e Inhalt Mem[x] einer Speicherzelle x zu finden ist
- Für jede Cache-Line vier mögliche Zustände **M**, **E**, **S**, **I**:
  - **Modified:** Cache-Line nur im lokalen Cache, „*dirty*“: wurde verändert → Cache muss Daten ins RAM zurück schreiben, bevor weitere Lesezugriffe auf diese Adresse im RAM erlaubt sind. Nach dem Zurückschreiben Zustandsänderung in **Exclusive.**
  - **Exclusive:** Cache-Line nur im lokalen Cache, „*clean*“: identisch mit RAM. Kann jederzeit in Status **Shared** wechseln, wenn andere CPU den Wert lesen will. Auch Wechsel zu **Modified** möglich, wenn Wert überschrieben wird.

MESI Cache Coherence Protocol (2)

- (vier Zustände...)
  - **Shared:** Diese Cache-Line wird evtl. auch in anderen Caches vorrätig gehalten, „*clean*“: identisch mit RAM. Bei Schreibzugriff müssen alle Kopien (in anderen Caches) auf **Invalid** gesetzt werden.
  - **Invalid:** Diese Cache-Line ist veraltet (der Wert im Haupt-speicher hat sich geändert); nicht benutzen.

Zustandskombinationen (zwei Caches):

(Quelle: http://en.wikipedia.org/wiki/MESI_protocol)

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

MESI Cache Coherence Protocol (3)

- **Lesezugriff:** In allen Zuständen außer **invalid** erlaubt
- **Schreibzugriff:** nur im (lokalen!) Zustand **modified** oder **exclusive** erlaubt –
  Im Zustand **shared** müssen zuerst alle Kopien der Cache-Line auf **invalid** gesetzt werden.

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Nutzen?

- „Wer“ profitiert von mehreren Kernen/CPUs?
  - Takterhöhung beschleunigt jede Anwendung
  - Pipelining beschleunigt (automatisch) die meisten Anwendungen
  - Einsatz mehrerer Kerne / CPUs / HT:
    - zunächst gar keine Beschleunigung einer einzelnen Applikation
    - schlimmstes Szenario: Ein Kern durch Anwendung belegt, restliche Kerne untätig
    - Betriebssystem und Anwendungen müssen mehrere Kerne unterstützen

Multi-Thread-Support

- BS-Support: Scheduler muss das Verteilen von mehreren Prozessen/Threads auf mehrere Kerne unterstützen
- Anwendungs-Support: Anwendung muss „parallelisiert“ sein, also aus mehreren (relativ) unabhängigen Anwendungssträngen (Threads) bestehen

Typische Anwendungs-Designs (1)

- Master/Worker
  - Ein Master-Thread erhält oder erstellt Aufträge und verteilt diese an ...
  - ... mehrere Worker-Threads: Die rechnen / erledigen je einen konkreten Auftrag, geben das Ergebnis an den Master zurück und beenden sich
  - Beispiel: Apache Webserver

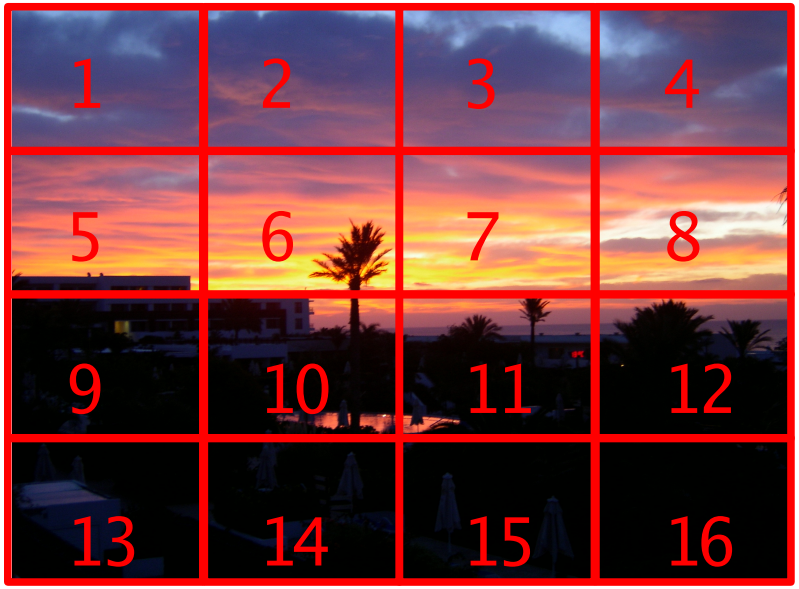
Bild: HP

```
[esser@vserver:~]$ pstree -p|grep apache
init(1)-+-apache2(19459)-+-apache2(1798)
        |                |-apache2(3400)
        |                |-apache2(3976)
        |                |-apache2(3977)
        |                |-apache2(3978)
        |                |-apache2(23603)
        |                |-apache2(26067)
        |                |-apache2(26238)
        |                |-apache2(5995)
        |                `-apache2(16194)
```

Typische Anwendungs-Designs (2)

- Örtliche Parallelisierung
- z. B. Bildverarbeitung:

→



  - einzelne Teilbereiche separat bearbeiten
  - regelmäßig „Randinformationen“ austauschen

Parallel programmieren (1)

- Threads in Standardsprache (C, C++, ...) von Hand erstellen
- Standardsprache mit Bibliothek um spezielle Parallelisierungsfunktionen erweitern (z. B. OpenMP, siehe https://computing.llnl.gov/tutorials/openMP/)
- spezielle parallele Programmiersprache nutzen
  - Occam, Erlang, Scala, Clojure, Fortress, ...
  - kurze Beschreibungen: z. B. unter http://pvs.uni-muenster.de/pvs/lehre/SS10/seminar/

Parallel programmieren (2)

- Beispiel in der Sprache Fortress:

```
for k<-1:5 do
  print k " "
  print k " "
end
```

erzeugt z. B. 4 1 4 1 5 2 5 2 3 3
und nicht 1 1 2 2 3 3 4 4 5 5

  - For-Schleife **implizit parallel**
  - Während Laufzeit des Programms werden neue Threads erzeugt, die Teile der Schleife berechnen
  - alternativ: neue Threads von Hand starten (für klassisches Modell, manuelle Parallelisierung)

Parallel programmieren (3)

- Klassisch / manuell

```
while (true) {
   req = read_request();          // Warten auf Arbeit
   T = new WorkerThread(req);     // neuen Thread ...
   T.start()                      // ... starten
}

Class WorkerThread extends Thread {
   ...
}
```

Parallel programmieren (4)

- Alternative „gute“ Nutzung eines Mehrkernsystems: Multi-User-Betrieb
- viele Anwender starten eigene Prozesse
- Nichts zu tun: Szenario ist schon parallelisiert

Ergänzende Literatur

- Parallelprogrammierung, in verschiedenen Hardware-Modellen: https://computing.llnl.gov/tutorials/parallel_comp/
- Kapitel 5 (Mehrprozessorsysteme) der Vorlesung Rechnerarchitektur, Univ. Dortmund, SS 2009, http://ls12-www.cs.tu-dortmund.de/de/patrec/teaching/SS09/rechnerarchitektur/