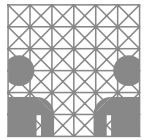


64-040 Modul InfB-RS: Rechnerstrukturen

https://tams.informatik.uni-hamburg.de/lectures/2016ws/vorlesung/rs

– Kapitel 17 –

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Wintersemester 2016/2017

Kapitel 17

Parallelarchitekturen

Motivation: ständig steigende Anforderungen

- Simulationen, Wettervorhersage, Gentechnologie...
- Datenbanken, Transaktionssysteme, Suchmaschinen...
- Softwareentwicklung, Schaltungsentwurf...

- Performance eines einzelnen Prozessors ist begrenzt

$\Rightarrow$ Verteilen eines Programms auf mehrere Prozessoren

Vielfältige Möglichkeiten

- wie viele und welche Prozessoren?
- Kommunikation zwischen den Prozessoren?
- Programmierung und Software/Tools?

Begriffe

- **Antwortzeit**: die Gesamtzeit zwischen Programmstart und -ende, inklusive I/O-Operationen, Unterbrechungen etc. („*wall clock time*“, „*response time*“, „*execution time*“)

  $$\text{performance} = \frac{1}{\text{execution time}}$$

- **Ausführungszeit**: reine CPU-Zeit

  Unix `time`-Befehl:

  ```
  597.07u 0.15s 9:57.61 99.9%
  ```

  ```
  597.07 user CPU time [sec.]
    0.15 system CPU time
  9:57.61 elapsed time
    99.9 CPU/elapsed [%]
  ```

- **Durchsatz**: Anzahl der bearbeiteten Programme / Zeit

- **Speedup**: $s = \frac{\text{performance x}}{\text{performance y}} = \frac{\text{execution time y}}{\text{execution time x}}$

Wie kann man Performance verbessern?

- Ausführungszeit = $\langle$*Anzahl der Befehle*$\rangle \cdot \langle$*Zeit pro Befehl*$\rangle$

- weniger Befehle
  - *gute* Algorithmen
  - bessere Compiler
  - mächtigere Befehle (CISC)
- weniger Zeit pro Befehl
  - bessere Technologie
  - Architektur: Pipelining, Caches...
  - einfachere Befehle (RISC)
- parallele Ausführung
  - superskalare Architekturen, SIMD, MIMD

Amdahl's Gesetz

Möglicher Speedup durch Beschleunigung einer Teilfunktion?

1. System berechnet Programm P,
   darin Funktion X mit Anteil $0 < f < 1$ der Gesamtzeit
2. System berechnet Programm P,
   Funktion X' ist schneller als X mit Speedup S_X

Amdahl's Gesetz

Gene Amdahl, Architekt der IBM S/360, 1967

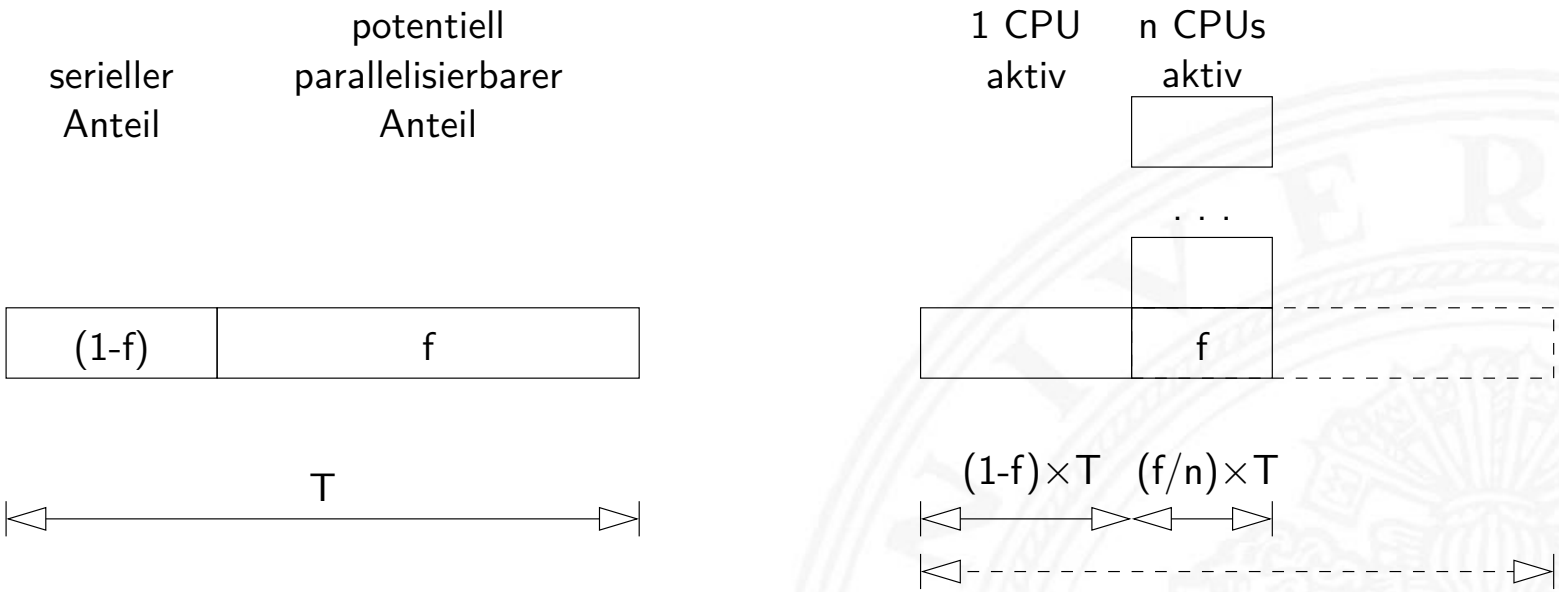
- Speedup

$$S_{gesamt} = \frac{1}{(1-f) + f/S_X}$$

Amdahl's Gesetz (cont.)

Speedup $$S_{gesamt} = \frac{1}{(1-f) + f/S_X}$$

- nur ein Teil f des Gesamtproblems wird beschleunigt



$\Rightarrow$ möglichst großer Anteil f

$\Rightarrow$ Optimierung lohnt nur für relevante Operationen

allgemeingültig: entsprechend auch für Projektplanung, Verkehr...

Amdahl's Gesetz (cont.)

- ursprüngliche Idee: Parallelrechner mit n-Prozessoren

Speedup $$S_{gesamt} = \frac{1}{(1-f) + k(n) + f/n}$$

n	# Prozessoren als Verbesserungsfaktor
f	Anteil parallelisierbarer Berechnung
$1-f$	Anteil nicht parallelisierbarer Berechnung
$k()$	Kommunikationsoverhead zwischen den Prozessoren

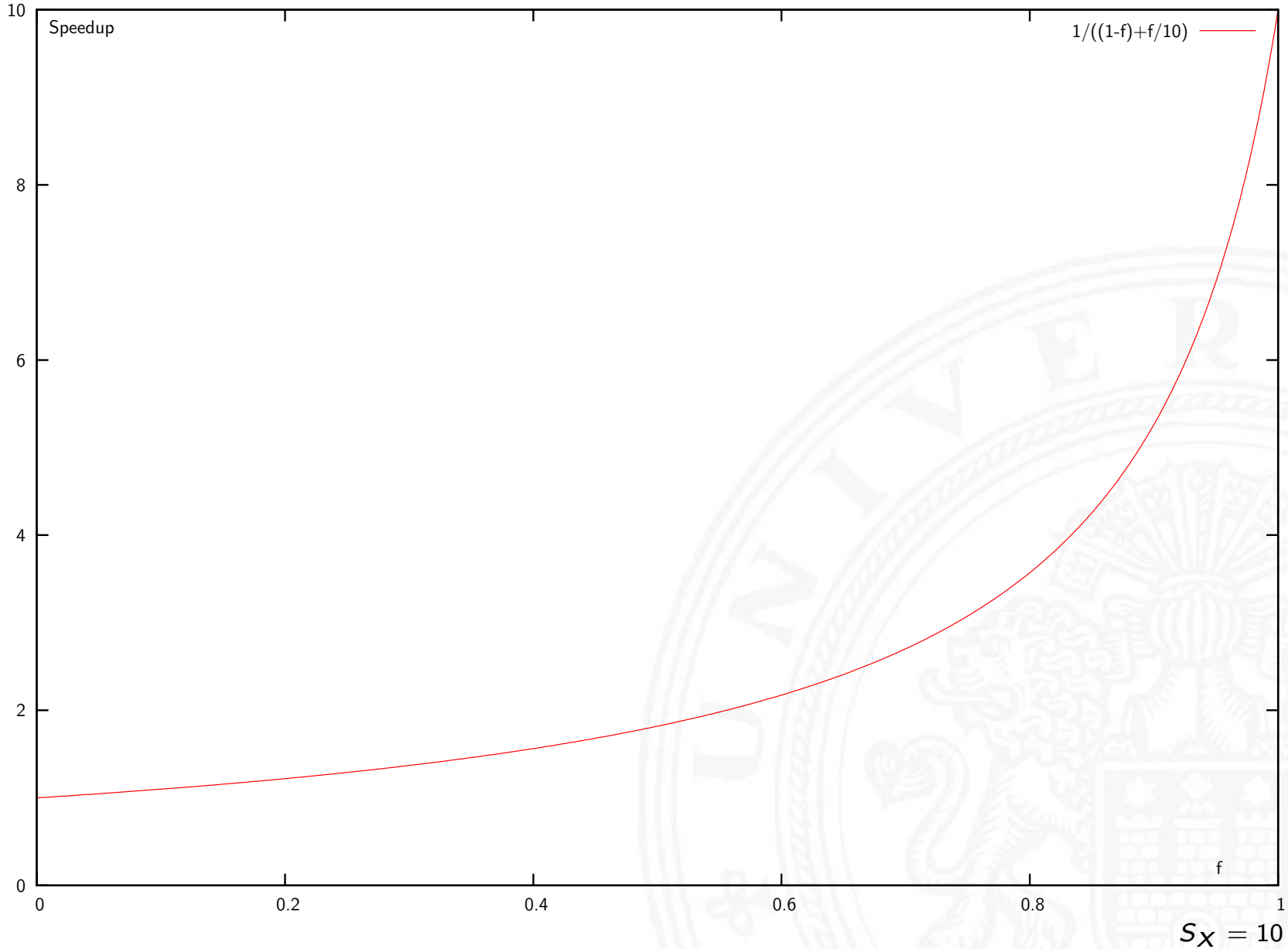
- Aufgaben verteilen
- Arbeit koordinieren
- Ergebnisse zusammensammeln

Amdahl's Gesetz: Beispiele

S_X	f	S_{gesamt}		
10	0,1	$1/(0,9 + 0,01)$	=	1,1
2	0,5	$1/(0,5 + 0,25)$	=	1,33
2	0,9	$1/(0,1 + 0,45)$	=	1,82
1,1	0,98	$1/(0,02 + 0,89)$	=	1,1
4	0,5	$1/(0,5 + 0,125)$	=	1,6
4 536	0,8	$1/(0,2 + 0,0\ldots)$	=	5,0
9 072	0,99	$1/(0,01 + 0,0\ldots)$	=	98,92

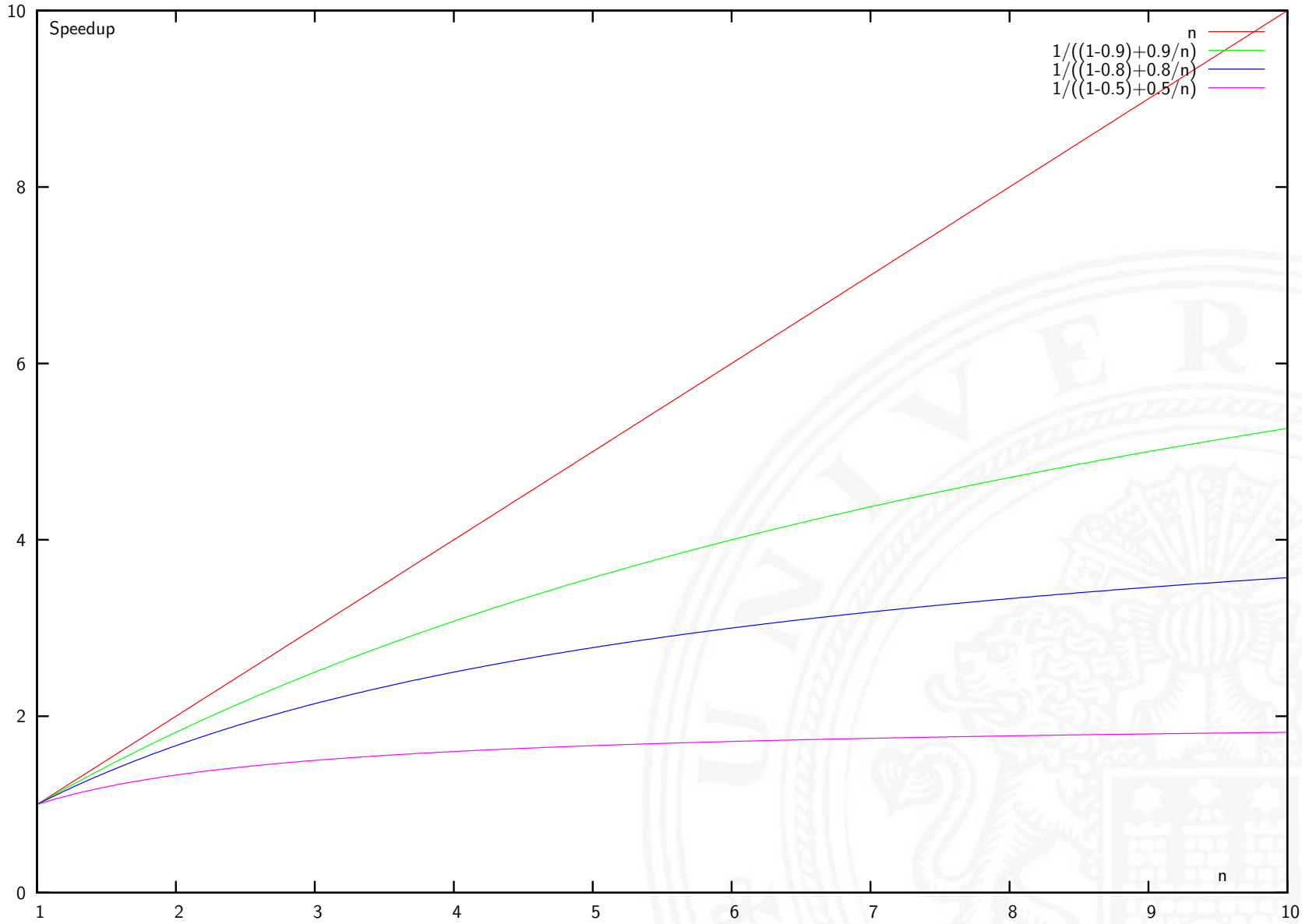
- Optimierung bringt nichts, wenn der nicht beschleunigte „serielle“ Anteil $(1 - f)$ eines Programms überwiegt
- n-Prozessoren (große S_X) wirken *nicht linear*
- die erreichbare Parallelität in Hochsprachen-Programmen (z.B. Java) ist gering, typisch $S_{gesamt} \leq 4$

Amdahl's Gesetz: Beispiele (cont.)

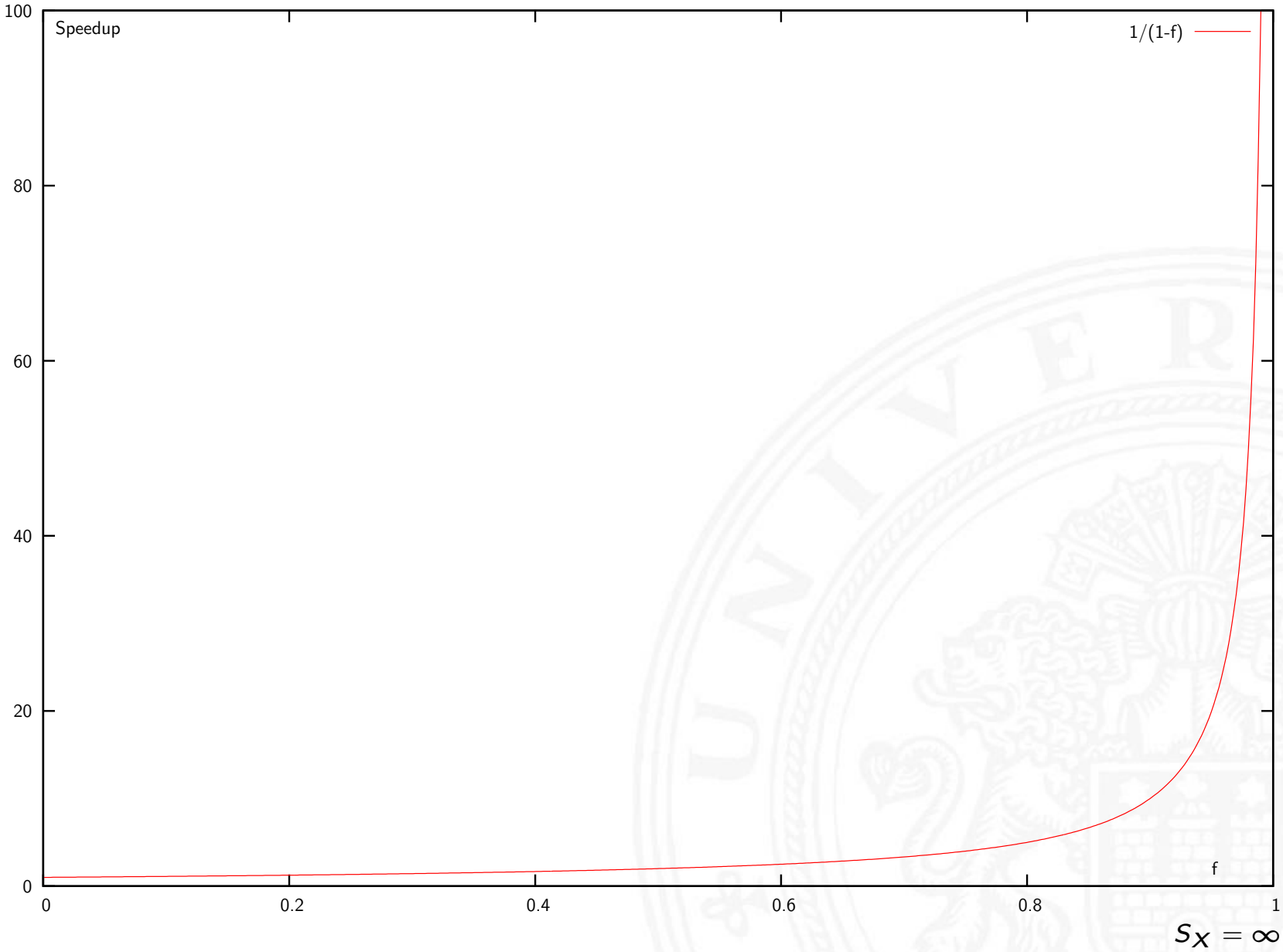


$S_X = 10$

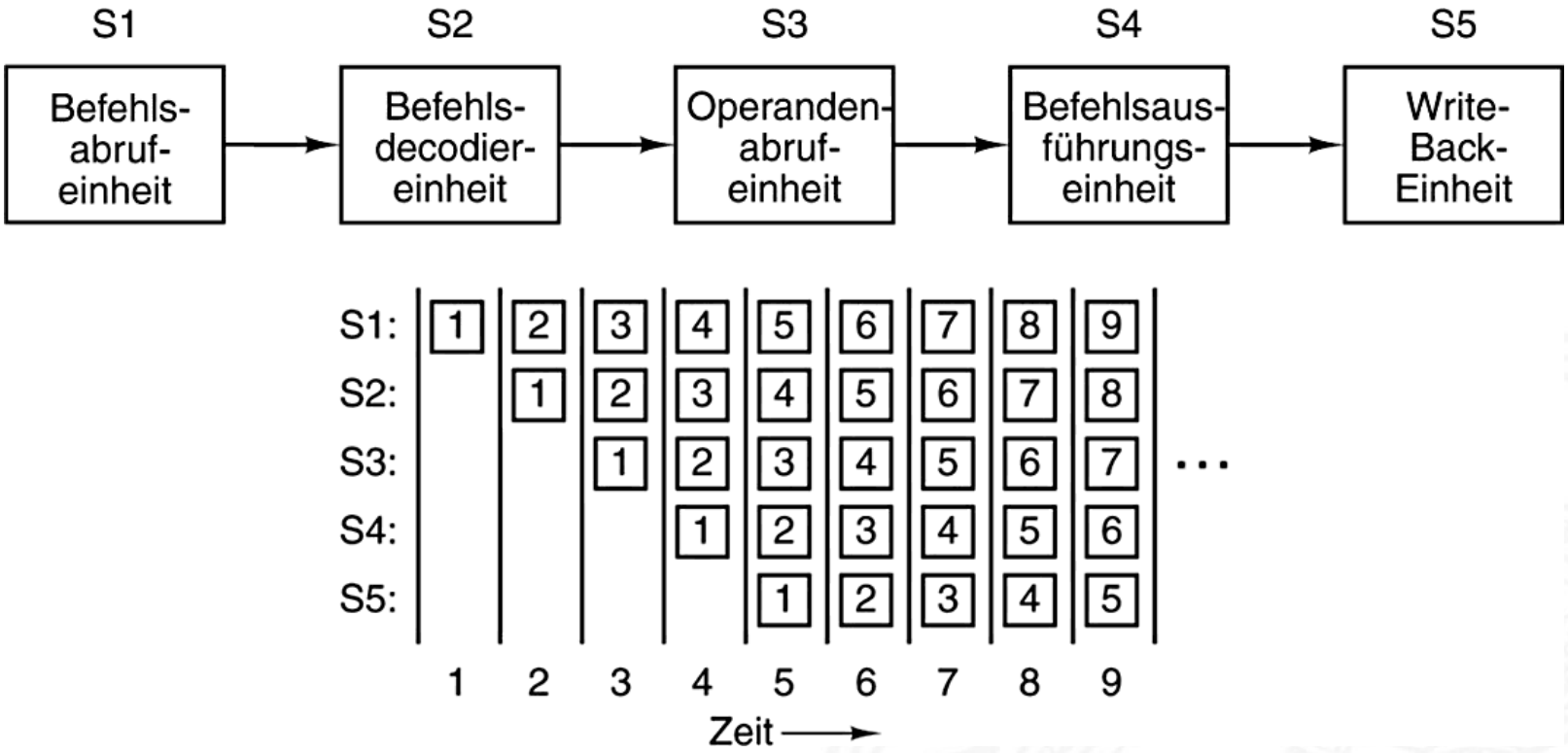
Amdahl's Gesetz: Beispiele (cont.)



Amdahl's Gesetz: Beispiele (cont.)



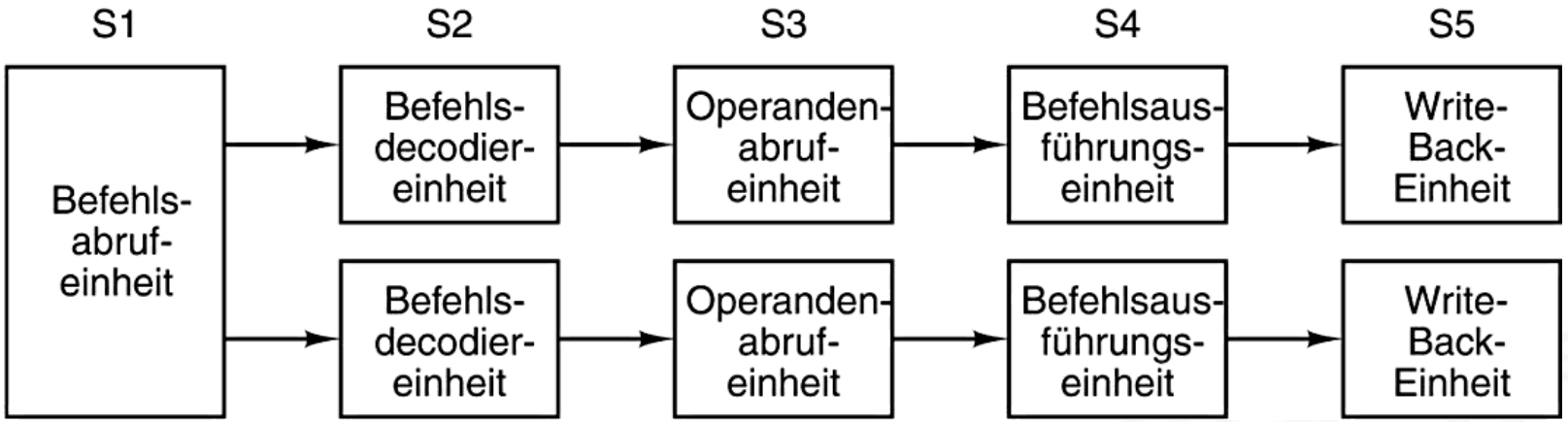
Befehlspipeline



[TA14]

- Befehl in kleinere, schnellere Schritte aufteilen ⇒ höherer Takt
- mehrere Instruktionen überlappt ausführen ⇒ höherer Durchsatz

Parallele Pipelines



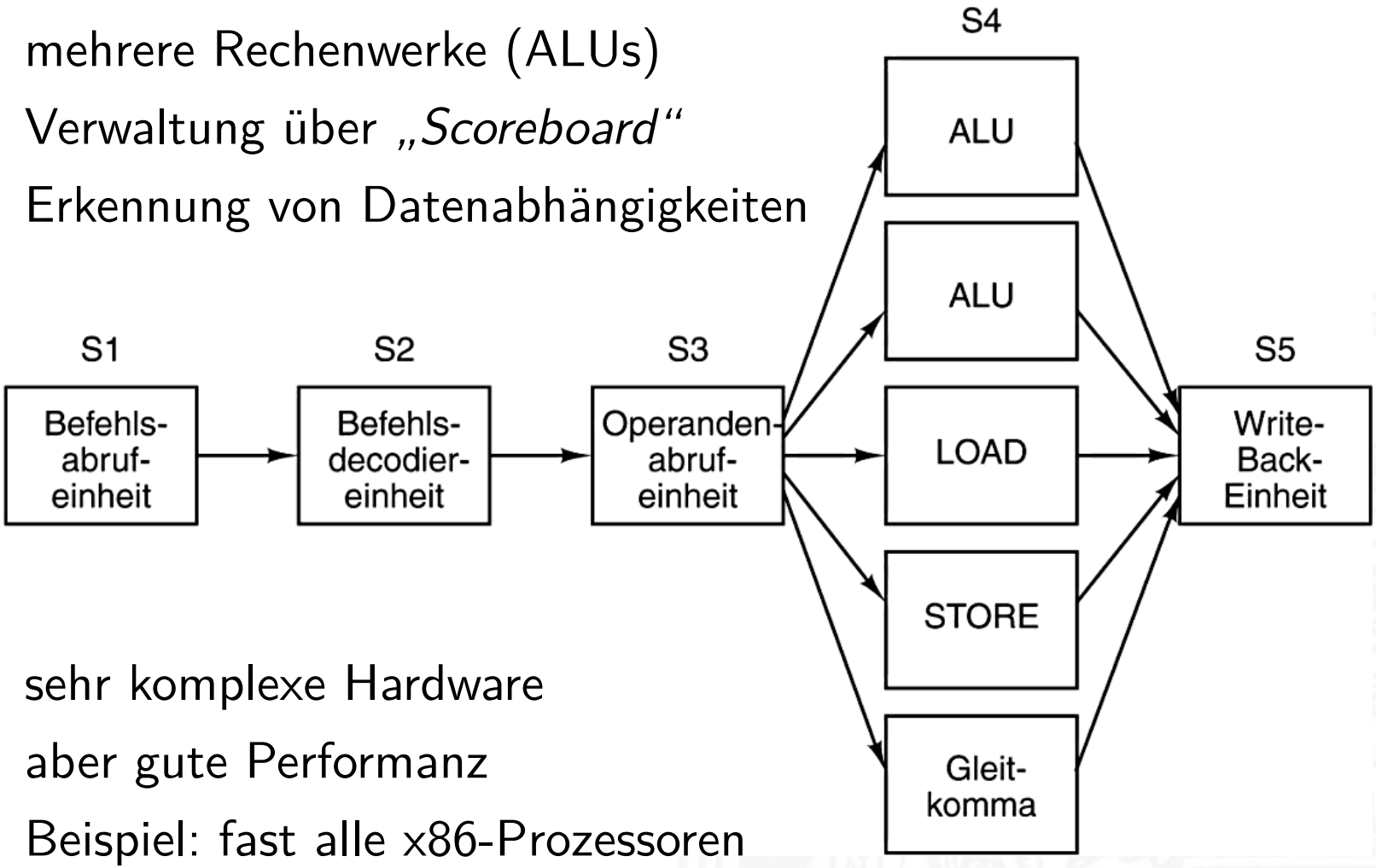
[TA14]

- im Bild jeweils zwei Operationen pro Pipelinestufe
- parallele („superskalare“) Ausführung

- komplexe Hardware (Daten- und Kontrollabhängigkeiten)
- Beispiel: Pentium

Superskalarer Prozessor

- mehrere Rechenwerke (ALUs)
- Verwaltung über „*Scoreboard*“
  Erkennung von Datenabhängigkeiten



- sehr komplexe Hardware
- aber gute Performanz
- Beispiel: fast alle x86-Prozessoren seit Pentium II

[TA14]

Superskalar

- Superskalare CPUs besitzen mehrere Recheneinheiten: 4. . . 10
- in jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet

$\Rightarrow$ ILP (**I**nstruction **L**evel **P**arallelism)

- Hardware verteilt initiierte Instruktionen auf Recheneinheiten
- pro Takt kann *mehr als eine* Instruktion initiiert werden
  Die Anzahl wird dynamisch von der Hardware bestimmt:
  0. . . „*Instruction Issue Bandwidth*“

\+ sehr effizient, alle modernen CPUs sind superskalar

– Abhängigkeiten zwischen Instruktionen sind der Engpass, das Problem der Hazards wird verschärft

Superskalar – Datenabhängigkeiten

Datenabhängigkeiten

- RAW – **R**ead **A**fter **W**rite
  Instruktion I_x darf Datum erst lesen, wenn I_{x-n} geschrieben hat
- WAR – **W**rite **A**fter **R**ead
  Instruktion I_x darf Datum erst schreiben, wenn I_{x-n} gelesen hat
- WAW – **W**rite **A**fter **W**rite
  Instruktion I_x darf Datum erst überschreiben, wenn I_{x-n} geschrieben hat

Datenabhängigkeiten superskalarer Prozessoren

- RAW: echte Abhängigkeit; Forwarding ist kaum möglich und in superskalaren Pipelines extrem aufwändig
- WAR, WAW: „*Register Renaming*“ als Lösung

Superskalar – Datenabhängigkeiten (cont.)

„Register Renaming“

- Hardware löst Datenabhängigkeiten innerhalb der Pipeline auf
- Zwei Registersätze sind vorhanden
  1. Architektur-Register: „logische Register“ der ISA
  2. viele Hardware-Register: „Rename Register“
     - dynamische Abbildung von ISA- auf Hardware-Register

- Beispiel
  - Originalcode

```
tmp  = a + b;
res1 = c + tmp;
tmp  = d + e;
res2 = tmp - f;
```

  nach Renaming

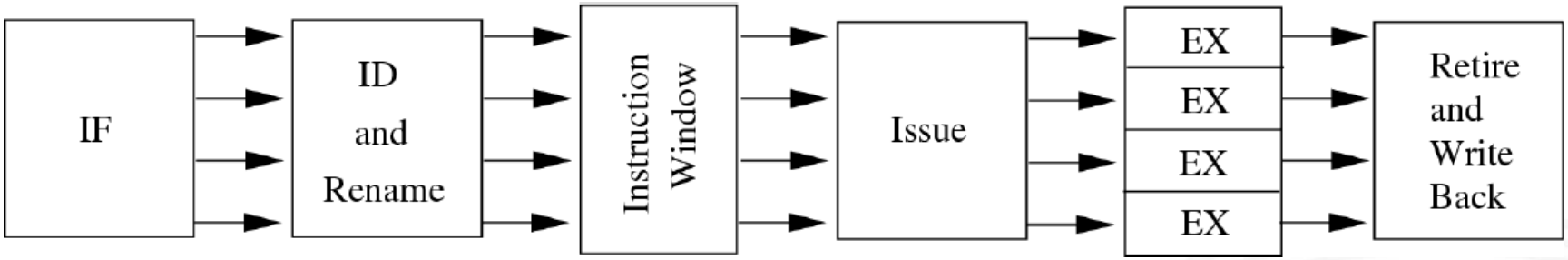
```
tmp1 = a + b;
res1 = c + tmp;
tmp2 = d + e;
res2 = tmp2 - f;
tmp  = tmp2;
```

  - Parallelisierung des modifizierten Codes

```
tmp1 = a + b;        tmp2 = d + e;
res1 = c + tmp1;     res2 = tmp2 - f;     tmp = tmp2;
```

Superskalar – Pipeline

Aufbau der superskalaren Pipeline

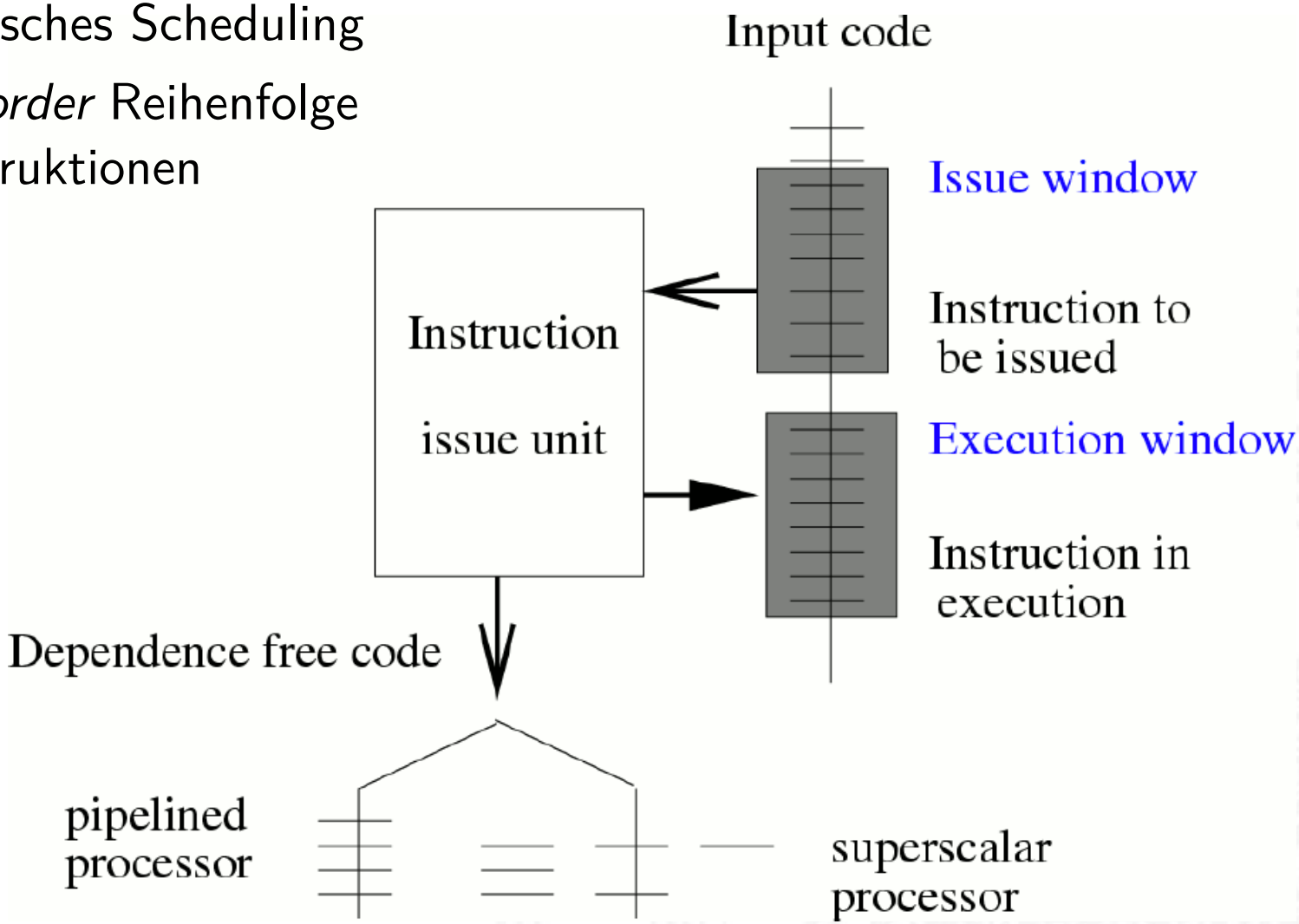


- lange Pipelines mit vielen Phasen: Fetch (Prefetch, Predecode), Decode / Register-Renaming, Issue, Dispatch, Execute, Retire (Commit, Complete / Reorder), Write-Back
- je nach Implementation unterschiedlich aufgeteilt
- entscheidend für superskalare Architektur sind die Schritte
  vor den ALUs: Issue, Dispatch $\Rightarrow$ *out-of-order* Ausführung
  nach -"- : Retire $\Rightarrow$ *in-order* Ergebnisse

Superskalar – Pipeline (cont.)

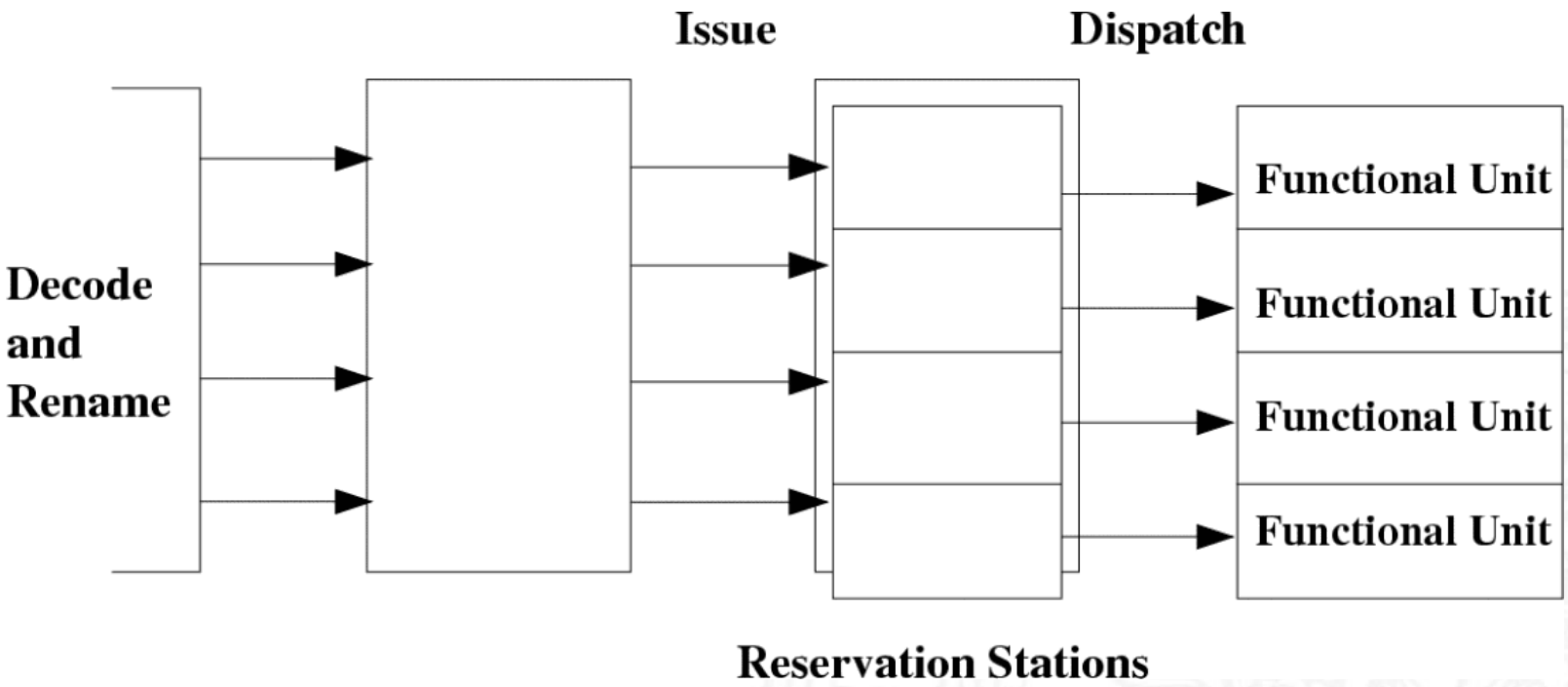
- Dynamisches Scheduling

⇒ *out-of-order* Reihenfolge der Instruktionen



Superskalar – Pipeline (cont.)

- Issue: globale Sicht
  Dispatch: getrennte Ausschnitte in „*Reservation Stations*“



Superskalar – Pipeline (cont.)

- Reservation Station für jede Funktionseinheit
  - speichert: initiierte Instruktionen die auf Recheneinheit warten
  - –"– zugehörige Operanden
  - –"– ggf. Zusatzinformation
  - Instruktion bleibt blockiert, bis alle Parameter bekannt sind und wird dann an die zugehörige ALU weitergeleitet
- ggf. „*Retire*“-Stufe
  - Reorder-Buffer: erzeugt wieder *in-order* Reihenfolge
  - commit: „richtig ausgeführte“ Instruktionen gültig machen
    abort: Instruktionen verwerfen, z.B. Sprungvorhersage falsch
- Dynamisches Scheduling: zuerst '67 in IBM 360 (R. Tomasulo)
  - Forwarding
  - Registerumbenennung und Reservation Stations

Superskalar – Probleme

Spezielle Probleme superskalarer Pipelines

- weitere Hazard-Möglichkeiten
  - die verschiedenen ALUs haben unterschiedliche Latenzzeiten
  - Befehle „warten“ in den Reservation Stations

  $\Rightarrow$ Datenabhängigkeiten können sich mit jedem Takt ändern
- Kontrollflussabhängigkeiten:
  Anzahl der Instruktionen zwischen bedingten Sprüngen
  limitiert Anzahl parallelisierbarer Instruktionen

$\Rightarrow$ *„Loop Unrolling“* besonders wichtig
+ optimiertes (dynamisches) Scheduling: Faktor 3 möglich

Superskalar – Probleme (cont.)

Softwareunterstützung für Pipelining superskalarer Prozessoren
„*Software Pipelining*“

- Codeoptimierungen beim Compilieren als Ersatz/Ergänzung zur Pipelineunterstützung durch Hardware
- Compiler hat „globalen“ Überblick

$\Rightarrow$ zusätzliche Optimierungsmöglichkeiten

Superskalar – Ausnahmebehandlung

Interrupts, System-Calls und Exceptions

- Pipeline kann normalen Ablauf nicht fortsetzen
- Ausnahmebehandlung ist wegen der Vielzahl paralleler Aktionen und den Abhängigkeiten innerhalb der Pipelines extrem aufwändig
  - einige Anweisungen können verworfen werden
  - andere Pipelineaktionen müssen vollendet werden
    benötigt *zusätzliche Zeit* bis zur Ausnahmebehandlung
  - wegen Register-Renaming muss *viel mehr Information* gerettet werden als nur die ISA-Register

Superskalar – Ausnahmebehandlung (cont.)

Prinzip der Interruptbehandlung

- keine neuen Instruktionen mehr initiieren
- warten bis Instruktionen des Reorder-Buffers abgeschlossen sind
- Verfahren ist von der „Art“ des Interrupt abhängig
  - Precise-Interrupt: Pipelineaktivitäten komplett Beenden
  - Imprecise-Interrupt: wird als verzögerter Sprung (Delayed-Branching) in Pipeline eingebracht
    Zusätzliche Register speichern Information über Instruktionen die in der Pipeline nicht abgearbeitet werden können (z.B. weil sie den Interrupt ausgelöst haben)
- Definition: Precise-Interrupt
  - Programmzähler (PC) zur auslösenden Instruktion ist bekannt
  - alle Instruktionen bis zur PC-Instr. wurden vollständig ausgeführt
  - keine Instruktion nach der PC-Instr. wurde ausgeführt
  - Ausführungszustand der PC-Instruktion ist bekannt

Beispiel: Pentium 4 / NetBurst Architektur

- superskalare Architektur (mehrere ALUs)
- CISC-Befehle werden dynamisch in „μOPs“ (1...3) umgesetzt
- Ausführung der μOPs mit „Out of Order“ Maschine, wenn
  - Operanden verfügbar sind
  - funktionelle Einheit (ALU) frei ist
- Ausführung wird durch „Reservation Stations“ kontrolliert
  - beobachtet die Datenabhängigkeiten zwischen μOPs
  - teilt Ressourcen zu
- „Trace“ Cache
  - ersetzt traditionellen Anweisungscache
  - speichert Anweisungen in decodierter Form: Folgen von μOPs
  - reduziert benötigte Rate für den Anweisungsdecoder
- „Double pumped“ ALUs (2 Operationen pro Taktzyklus)

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

- große Pipelinelänge $\Rightarrow$ sehr hohe Taktfrequenzen

Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

- umfangreiches Material von Intel unter:
  `ark.intel.com`, `techresearch.intel.com`

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

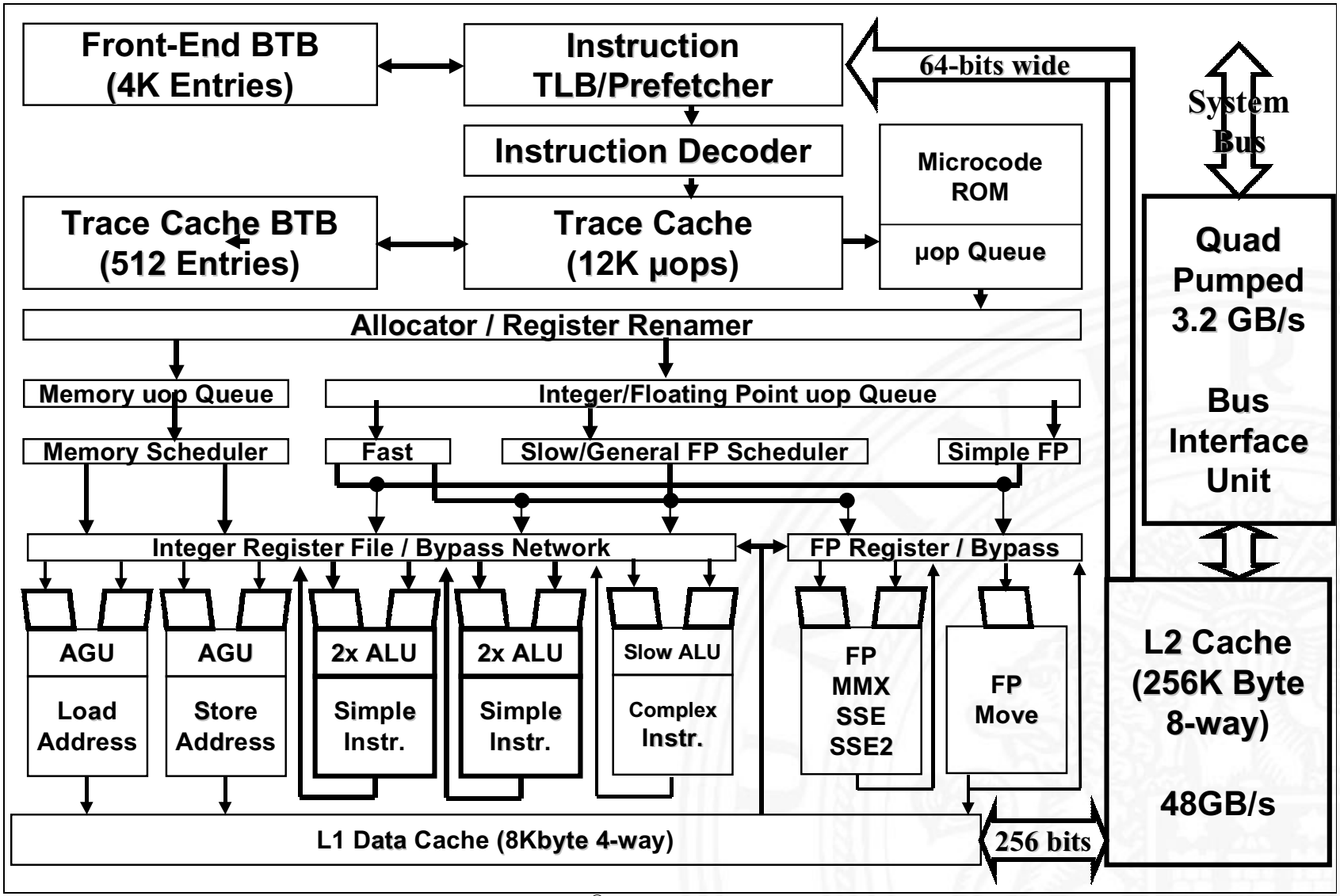
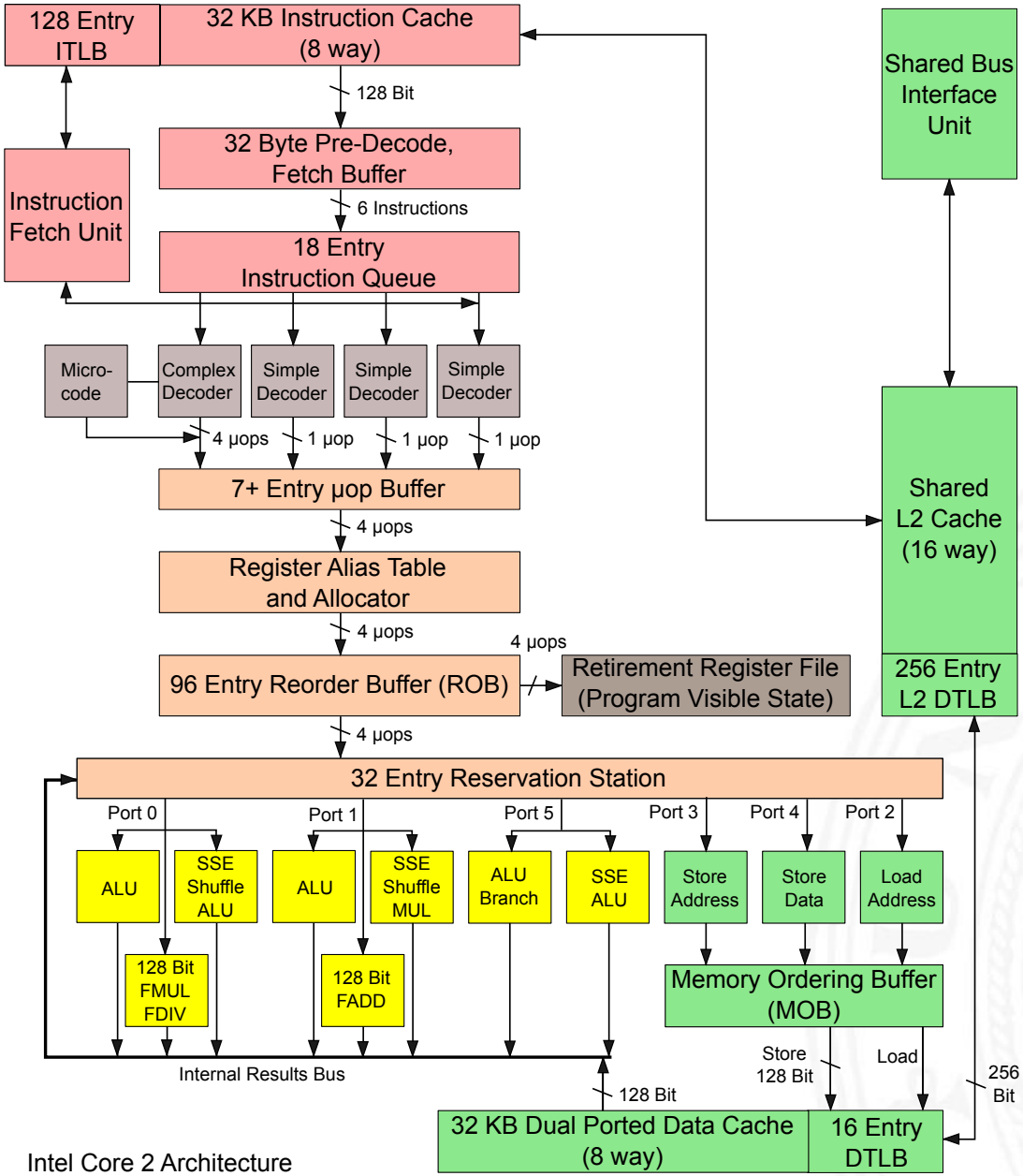


Figure 4: Pentium® 4 processor microarchitecture

Intel: Q1, 2001 [Intel]

Beispiel: Core 2 Architektur



Intel Core 2 Architecture

Parallelrechner mit mehreren Prozessoren

- Taktfrequenzen $> 10\,\text{GHz}$ nicht sinnvoll realisierbar
  - hoher Takt nur bei einfacher Hardware möglich
  - Stromverbrauch bei CMOS proportional zum Takt

$\Rightarrow$ mehrere Prozessoren
Datenaustausch: „*Shared-memory*“ oder Verbindungsnetzwerk

– Overhead durch Kommunikation

– Programmierung ist ungelöstes Problem

- aktueller Kompromiss: bus-basierte „SMPs“ mit 2. . . 16 CPUs

Flynn-Klassifikation

SISD *„Single Instruction, Single Data“*

- jeder klassische von-Neumann Rechner (z.B. PC)

SIMD *„Single Instruction, Multiple Data“*

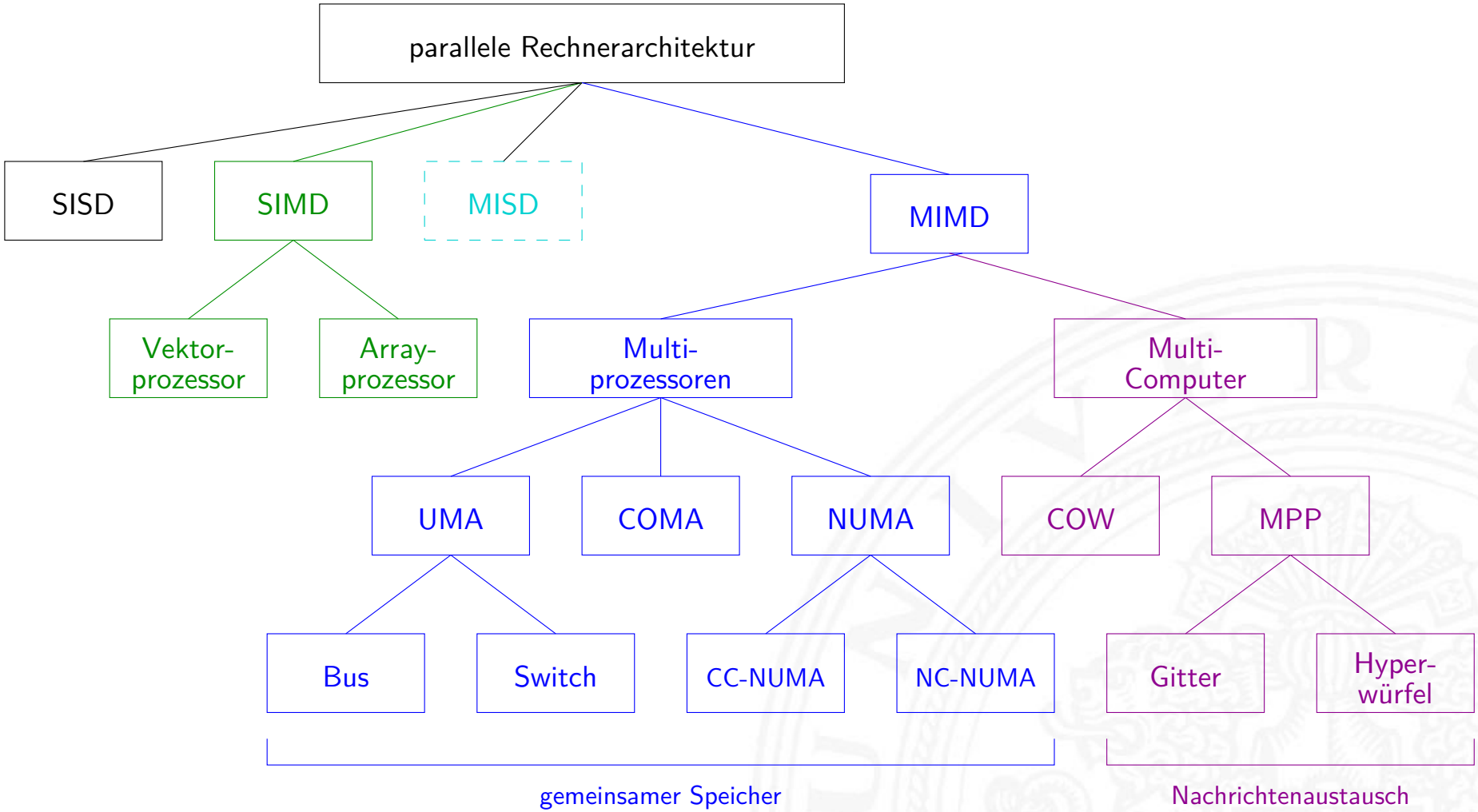
- Vektorrecher/Feldrechner
  z.B. Connection-Machine 2: 65 536 Prozessoren
- Erweiterungen in Befehlssätzen: superskalare Recheneinheiten werden direkt angesprochen
  z.B. x86 MMX, SSE, VLIW-Befehle: 2...8 fach parallel

MIMD *„Multiple Instruction, Multiple Data“*

- Multiprozessormaschinen
  z.B. Compute-Cluster, aber auch Multi-Core CPU

MISD *„Multiple Instruction, Single Data“* :-)

Detaillierte Klassifikation



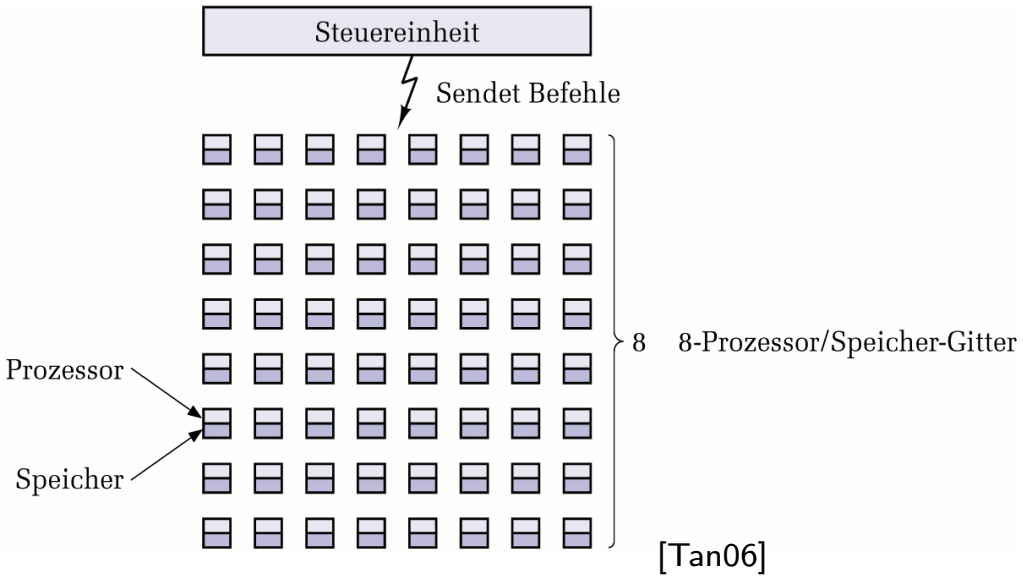
[TA14]

SIMD: Vektorrechner Cray-1 (1976)

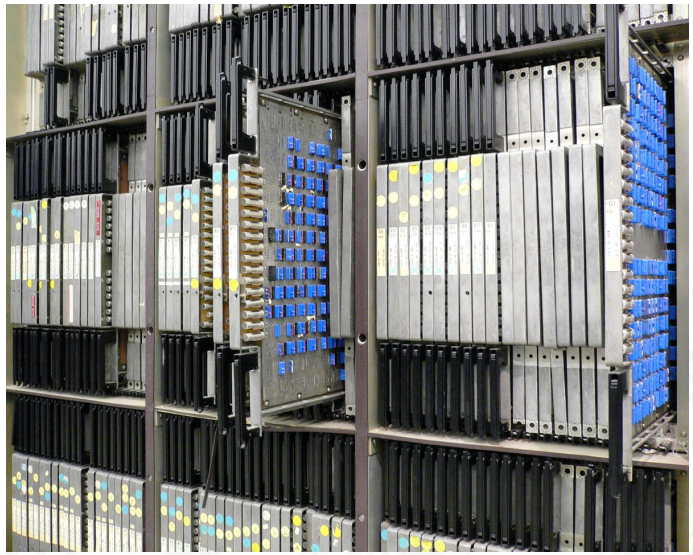
legendärer „Supercomputer“

- Prinzip: Anwendung eines Rechen-Befehls auf alle Elemente von Vektoren/Matrizen
- Adressberechnung mit „Stride“
- „Chaining“ von Vektorbefehlen
- schnelle skalare Befehle
- ECL-Technologie, Freon-Kühlung
- 1662 Platinen (Module), über Kabel verbunden
- 80 MHz Takt, 136 MFLOPS

SIMD: Feldrechner Illiac-IV (1964...1976)

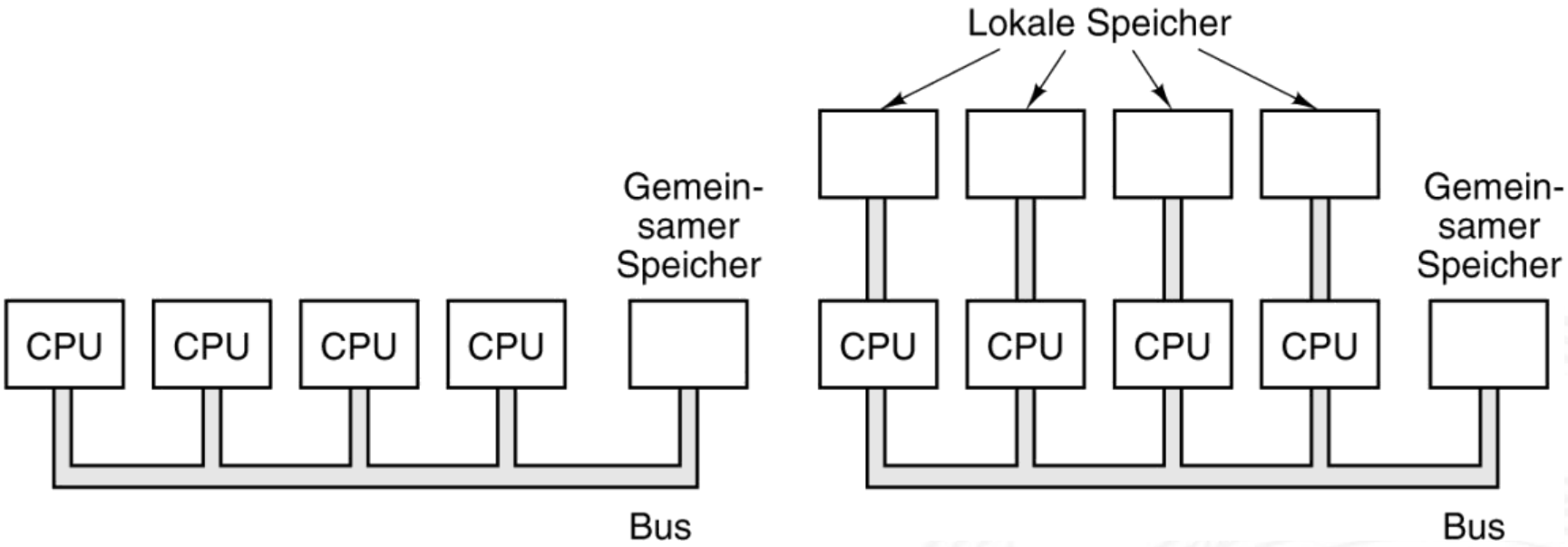


[Tan06]



- ein zentraler Steuerprozessor
- 64 Prozessoren/ALUs und Speicher, 8×8 Matrix
- Befehl wird parallel auf allen Rechenwerken ausgeführt
- aufwändige und teure Programmierung
- oft schlechte Auslastung (Parallelität Algorithmus vs. #CPUs)

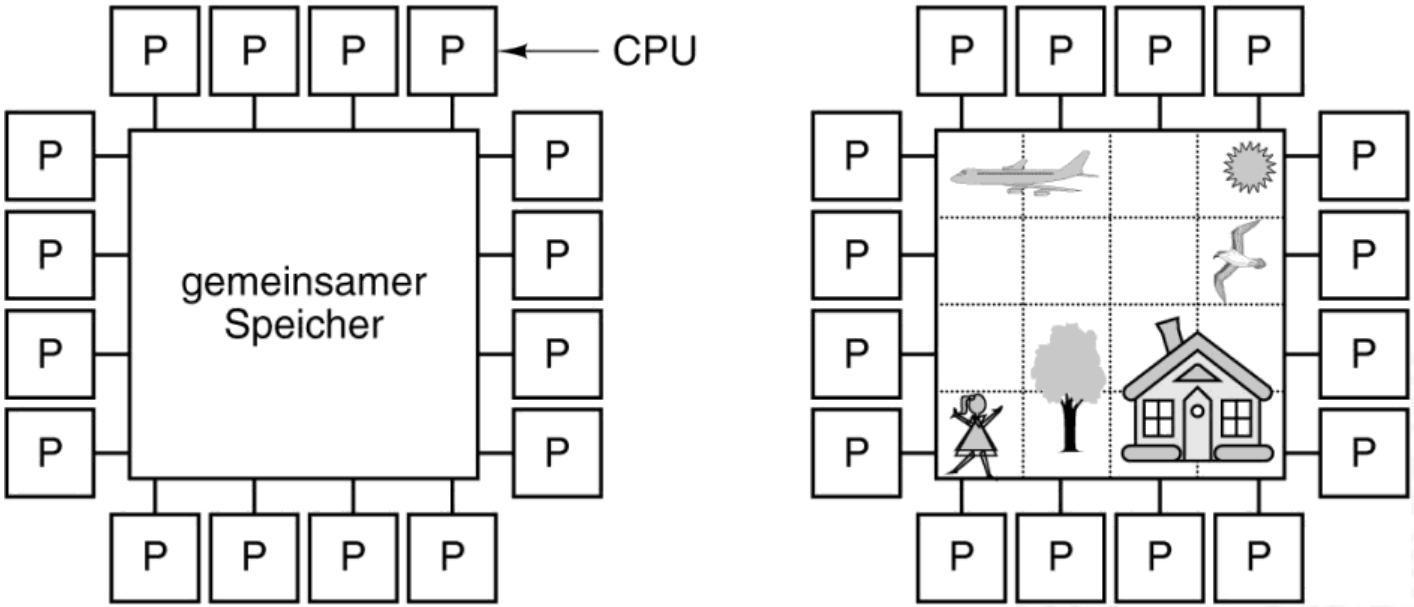
MIMD: Konzept



[TA14]

- mehrere Prozessoren, über Bus/Netzwerk verbunden
- gemeinsamer („shared“) oder lokaler Speicher
- unabhängige oder parallele Programme / Multithreading
- sehr flexibel, zunehmender Markterfolg

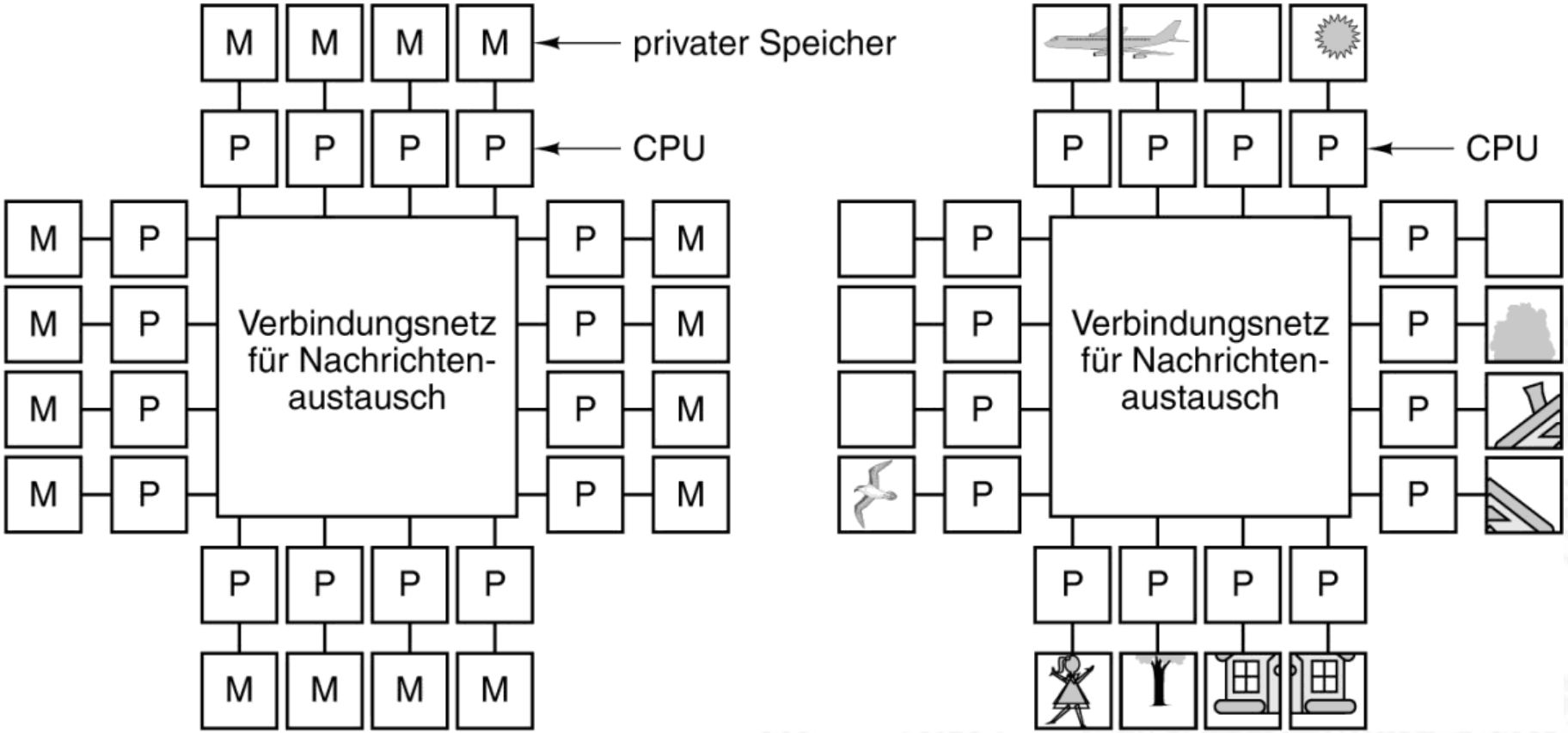
MIMD: Shared-Memory



[TA14]

- mehrere CPUs, aber gemeinsamer Speicher
- jede CPU bearbeitet nur eine Teilaufgabe
- CPUs kommunizieren über den gemeinsamen Speicher
- Zuordnung von Teilaufgaben/Speicherbereichen zu CPUs

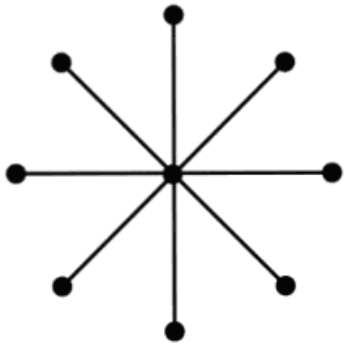
MIMD: Message-Passing



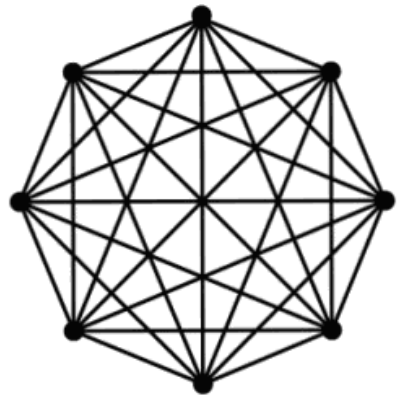
[TA14]

- jede CPU verfügt über eigenen (privaten) Speicher
- Kommunikation über ein Verbindungsnetzwerk
- Zugriff auf Daten anderer CPUs evtl. recht langsam

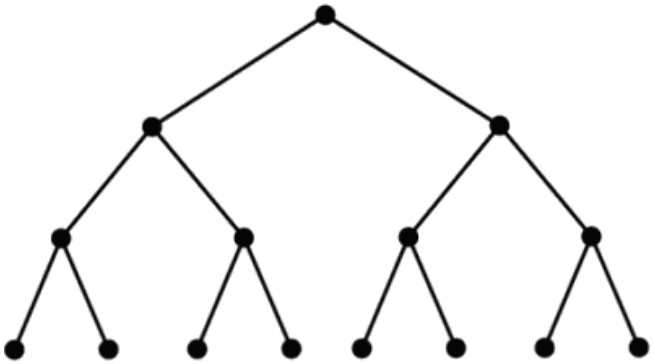
Verbindungs-Netzwerke



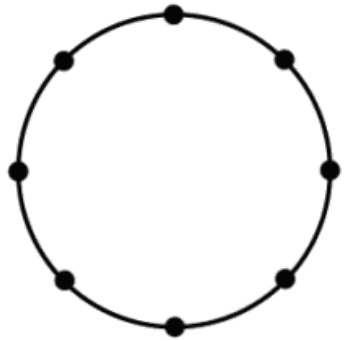
Stern



vollständige Vernetzung



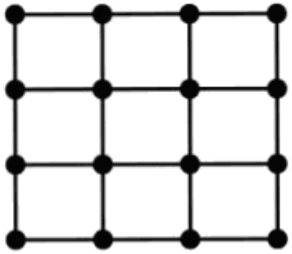
Baum



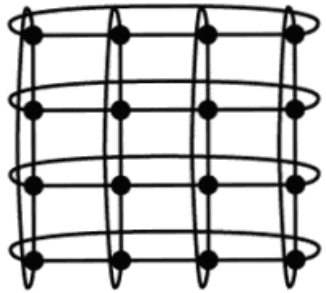
Ring

[TA14]

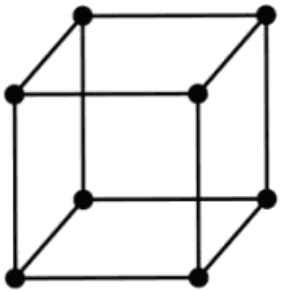
Verbindungs-Netzwerke (cont.)



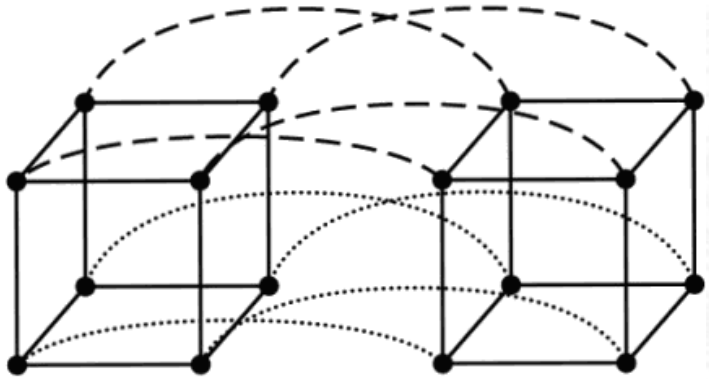
Gitter



doppelter Torus



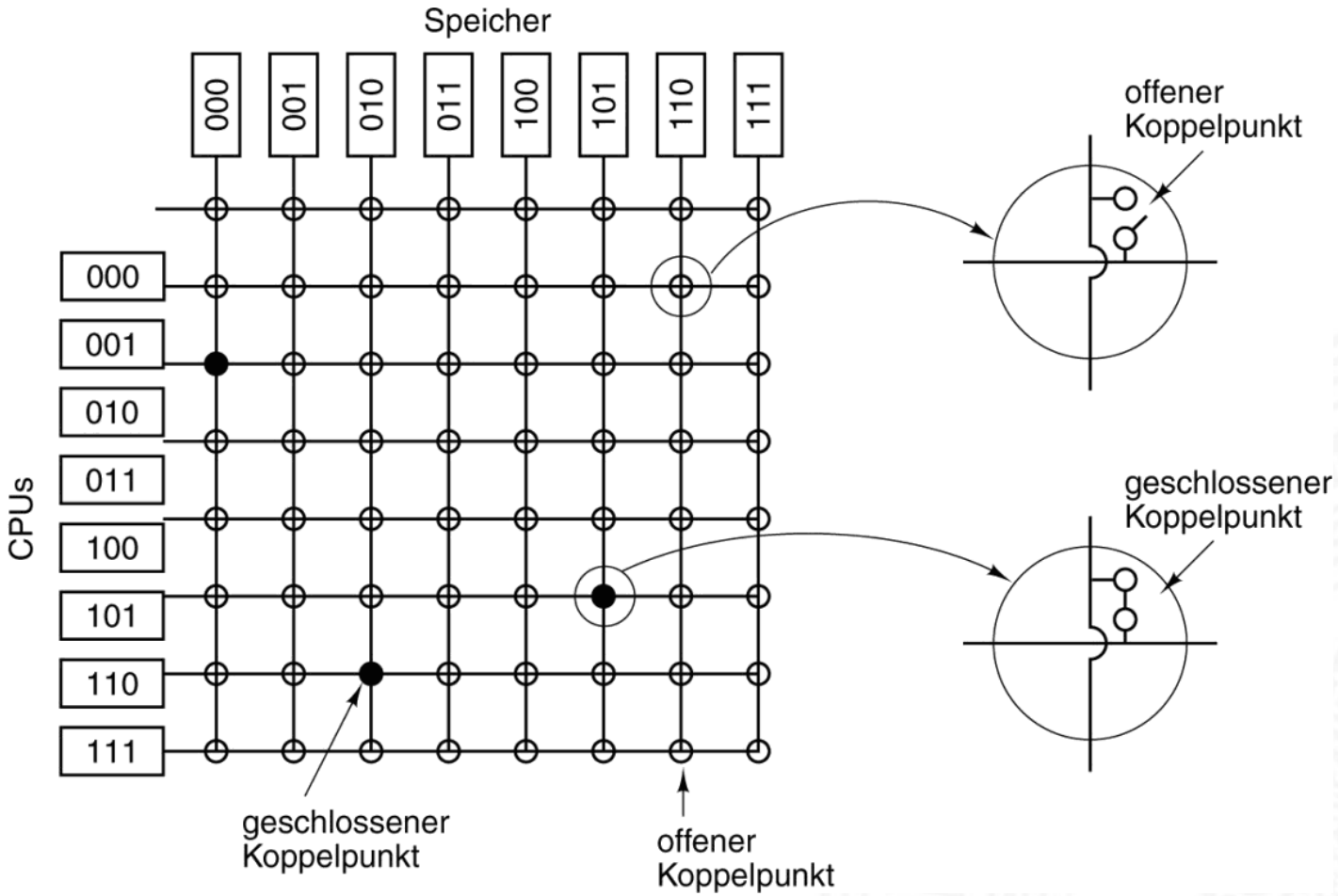
Würfel (Cube)



4-D-Hypercube

[TA14]

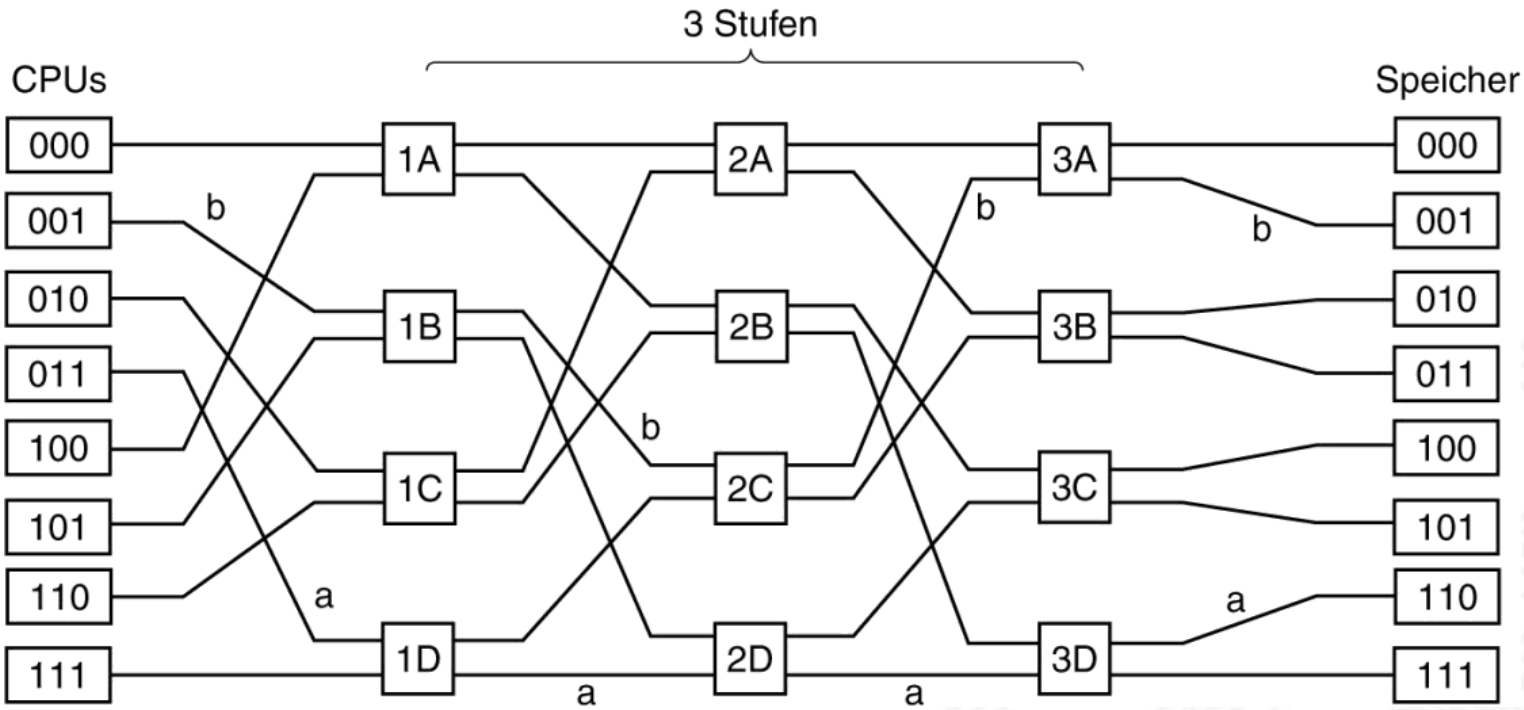
Kreuzschienenverteiler („*Crossbar Switch*“)



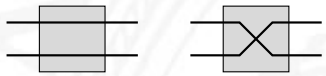
[TA14]

- jede CPU kann auf jeden Speicher zugreifen
- hoher Hardwareaufwand: $O(N^2)$ Schalter und Verbindungen
- Konflikte bei gleichzeitigem Zugriff auf einen Speicher

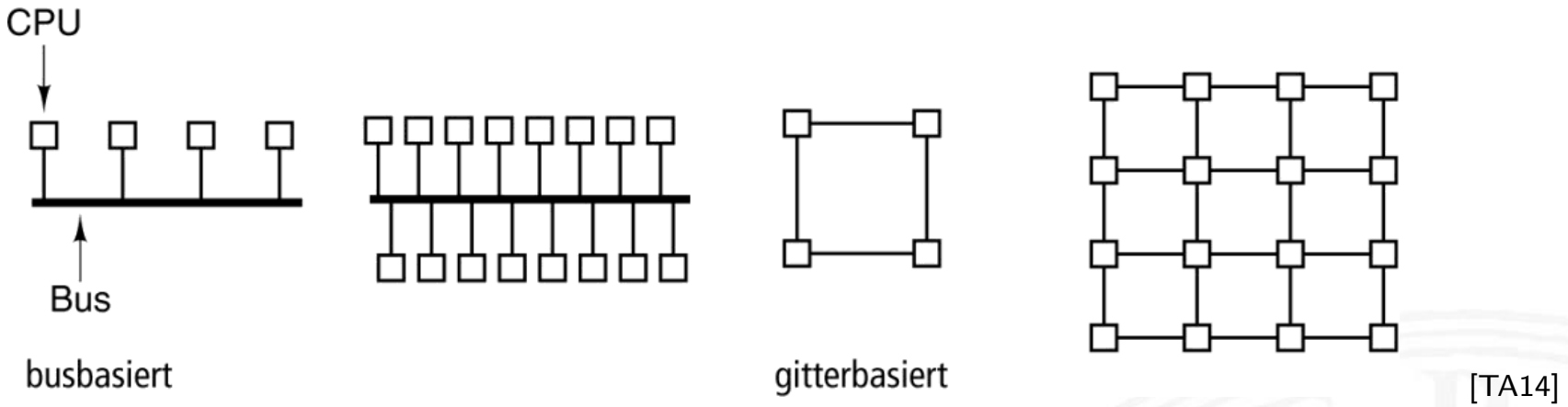
Omega-Netzwerk



[TA14]

- Schalter „gerade“ oder „gekreuzt“: 
- jede CPU kann auf jeden Speicher zugreifen
- aber nur bestimmte Muster, Hardwareaufwand $O(N \ln N)$

Skalierbarkeit



[TA14]

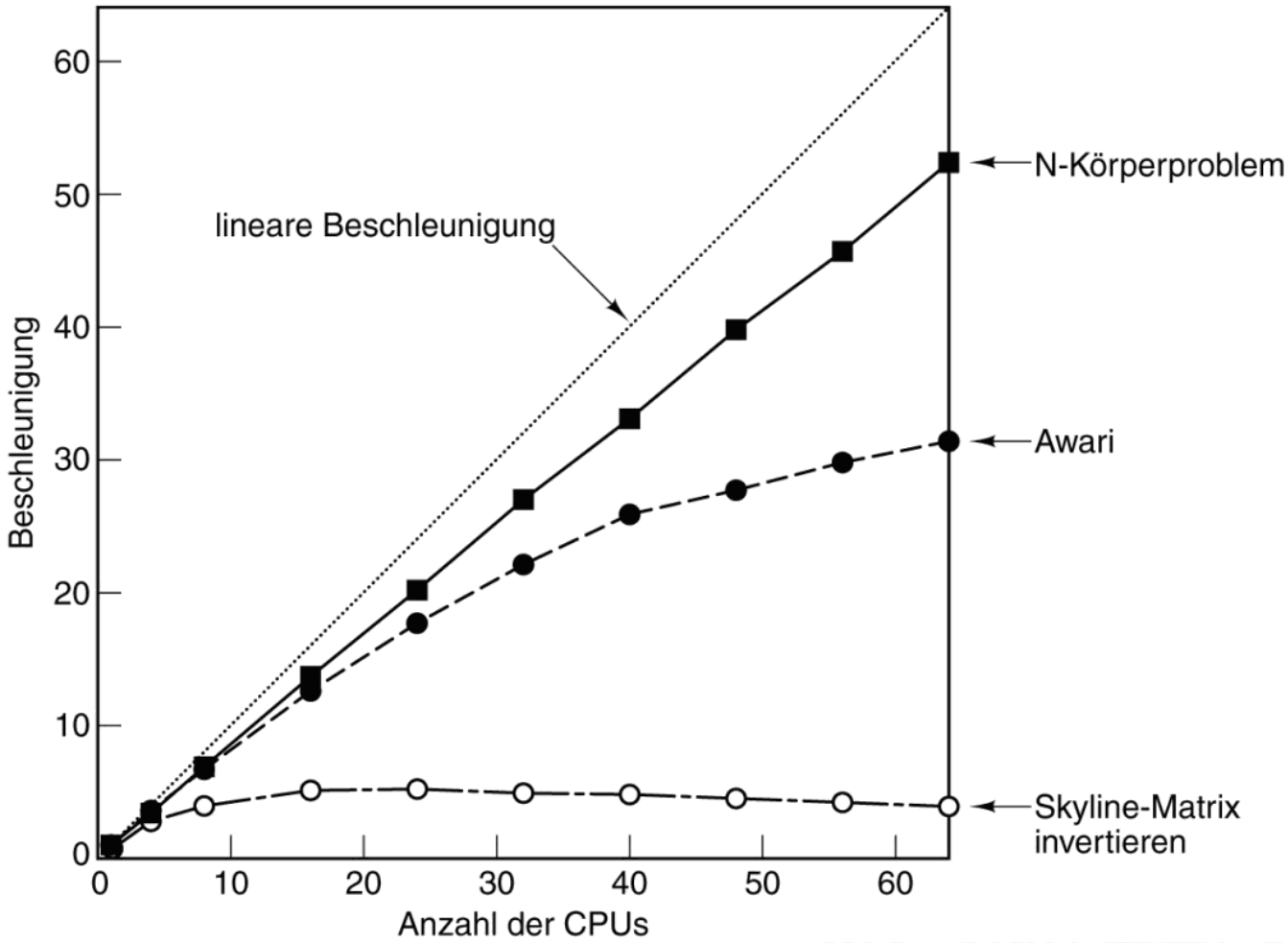
Wie viele CPUs kann man an ein System anschliessen?

- Bus: alle CPUs teilen sich die verfügbare Bandbreite

$\Rightarrow$ daher normalerweise nur 2...8 CPUs sinnvoll

- Gitter: verfügbare Bandbreite wächst mit Anzahl der CPUs

Speedup



[TA14]

- Maß für die Effizienz einer Architektur / eines Algorithmus'
- wegen Amdahl's Gesetz maximal linearer Zuwachs
- je nach Problem oft wesentlich schlechter

Parallelarchitekturen – Anmerkungen

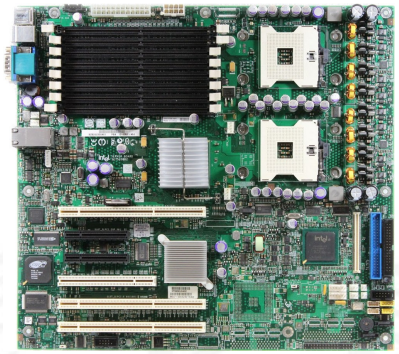
- Programmierung: ein ungelöstes Problem
  - Aufteilung eines Programms auf die CPUs/Rechenknoten?
  - insbesondere bei komplexen Kommunikationsnetzwerken
- Programme sind nur teilweise parallelisierbar
  - Parallelität einzelner Programme: kleiner 8
    gilt für Desktop-, Server-, Datenbankanwendungen etc.

  $\Rightarrow$ hochgradig parallele Rechner sind dann Verschwendung
- *Wohin mit den Transistoren aus „Moore's Law“?*

  $\Rightarrow$ SMP-/Mehrkern-CPUs (4...16 Proz.) sind technisch attraktiv
- Vektor-/Feld-Rechner für Numerik, Simulation ...
  - Grafikprozessoren (GPUs) sind Feld-Rechner
  - neben 3D-Grafik zunehmender Computing-Einsatz (OpenCL)
  - hohe Fließkomma-Rechenleistung (*single precision*)

SMP: *Symmetric Multiprocessing*

- mehrere Prozessoren nutzen gemeinsamen Hauptspeicher
- Zugriff über Verbindungsnetzwerk oder Bus
- geringer Kommunikationsoverhead



\+ Bus-basierte Systeme sind sehr kostengünstig

– aber schlecht skalierbar (Bus als Flaschenhals, s.o.)

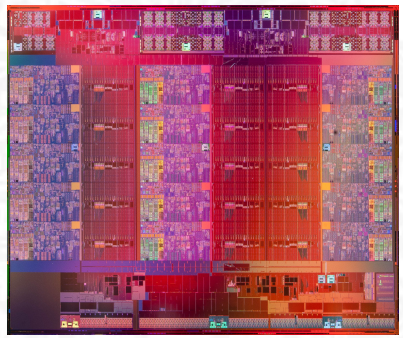
– Konsistenz der Daten
  - lokale Caches für gute Performanz notwendig
  - Hauptspeicher und Cache(s): Cache-Kohärenz MESI-Protokoll und „*Snooping*“

  → siehe Kapitel „18 Speicherhierarchie“
  - Registerinhalte: ? problematisch

– Prozesse wechseln CPUs: „*Hopping*“

- Multi-Core Prozessoren sind „SMP on-a-chip“

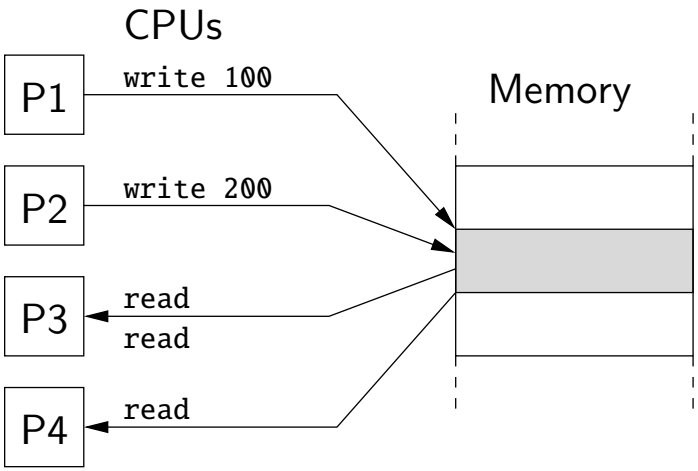
15-Kern Xeon E7 v2 [Intel]



SMP: Eigenschaften

Symmetric Multiprocessing

- alle CPUs gleichrangig, Zugriff auf Speicher und I/O
- Konsistenz: *Gleichzeitiger Zugriff auf eine Speicheradresse?*



$W1 \rightarrow 100$	$W1 \rightarrow 100$	$W2 \rightarrow 200$
$W2 \rightarrow 200$	$R3 \leftarrow 100$	$R4 \leftarrow 200$
$R3 \leftarrow 200$	$W2 \rightarrow 200$	$R3 \leftarrow 200$
$R3 \leftarrow 200$	$R3 \leftarrow 200$	$W1 \rightarrow 100$
$R4 \leftarrow 200$	$R4 \leftarrow 200$	$R3 \leftarrow 100$

$\Rightarrow$ *„Locking“* Mechanismen und Mutexe

- spez. Befehle, atomare Operationen, Semaphore etc.
- explizit im Code zu programmieren

SMP: Cache-Kohärenz

Cache für schnelle Prozessoren notwendig

- jede CPU hat eigene Cache (L1, L2 ...)
- aber gemeinsamer Hauptspeicher

Problem der *Cache-Kohärenz*

- Prozessor P_2 greift auf Daten zu, die im Cache von P_1 liegen
  - P_2 Lesezugriff: P_1 muss seinen Wert P_2 liefern
  - P_2 Schreibzugriff: P_1 muss Wert von P_2 übernehmen oder seinen Cache ungültig machen
  - *Was ist mit gleichzeitigen Zugriffen von P_1, P_2?*
- diverse Protokolle zur Cache-Kohärenz

  → siehe Kapitel „18 Speicherhierarchie“
  - z.B. MESI-Protokoll mit *„Snooping“*
    Modified, *Exclusive*, *Shared*, *Invalid*
  - Caches enthalten Wert, Tag und 2 bit MESI-Zustand

SMP: `volatile`

- MESI-Verfahren garantiert Cache-Kohärenz für Werte im Cache und im Hauptspeicher

Vorsicht: Was ist mit den Registern?

- Variablen in Registern werden von MESI nicht erkannt
- Compiler versucht, häufig benutzte Variablen soweit wie möglich in Registern zu halten
- globale/*shared*-Variablen niemals in Registern halten
- Java, C: Deklaration als `volatile`

SMP: Erreichbarer Speedup (bis 32 Threads)

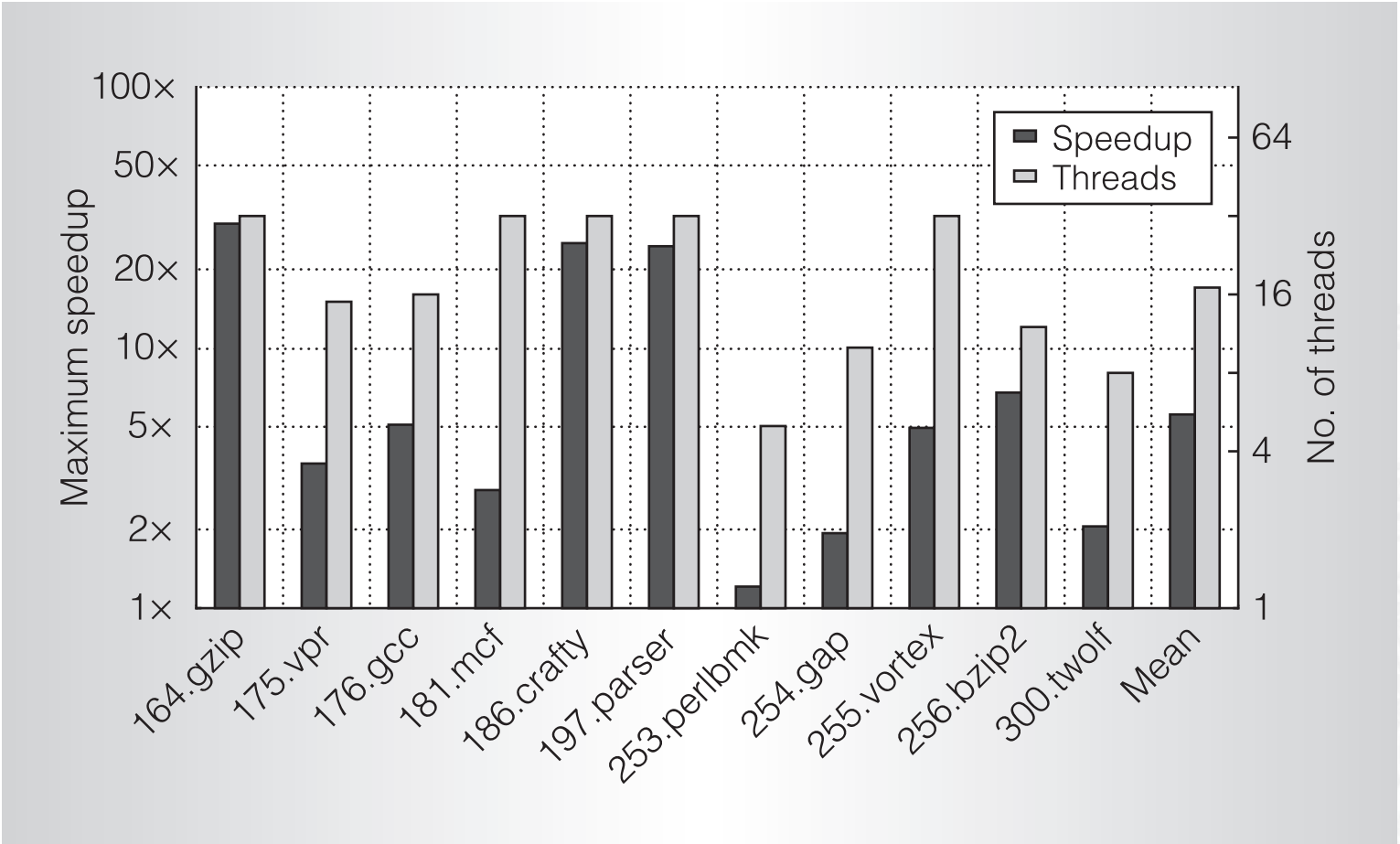


Figure 4. Maximum speedup achieved on up to 32 threads over single-threaded execution (black bars) and minimum number of threads at which the maximum speedup occurred (gray bars).

M.J .Bridges et.al.: *Revisiting the Sequential Programming Model for the Multicore Era*, IEEE Micro 2008 [Br+08]

Literatur

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–86894–238–5

[Tan06] A.S. Tanenbaum:
Computerarchitektur – Strukturen, Konzepte, Grundlagen.
5. Auflage, Pearson Studium, 2006. ISBN 3–8273–7151–1

[Intel] Intel Corp.; Santa Clara, CA.
`www.intel.com ark.intel.com`

[HP12] J.L. Hennessy, D.A. Patterson:
Computer architecture – A quantitative approach.
5th edition, Morgan Kaufmann Publishers Inc., 2012.
ISBN 978–0–12–383872–8

Literatur (cont.)

[PH16a] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface: ARM Edition*. Morgan Kaufmann Publishers Inc., 2016. ISBN 978–0–12–801733–3

[PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*. 5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0

[Br+08] M.J. Bridges [u. a.]: *Revisiting the Sequential Programming Model for the Multicore Era*. in: *IEEE Micro* 1 Vol. 28 (2008), S. 12–20.