RS

$$\text{Kosten des Dies} = \frac{\text{Kosten des Wafers}}{\text{Dies pro Wafer} \times \text{Ausbeute}}$$

$$\text{Dies pro Wafer} = \frac{\pi \times (1/2 \times \text{Durchmesser des Wafers})^2}{\text{Fläche des Dies}} \quad \} \text{Gesamtfläche}$$

$$- \frac{\pi \times \text{Durchmesser des Wafers}}{\sqrt{2 \times \text{Fläche des Dies}}} \quad \} \text{Verschnitt}$$

$$\text{Ausbeute} = \underbrace{\text{Wafer Ausbeute}}_{1 - P(\text{Wafer komplett defekt})} \times 1/(1 + \underbrace{\text{Defekte pro Flächeneinheit}}_{\sim 0{,}016 \text{ – } \sim 0{,}057 \text{ je nach Technologie}} \times \text{Die-Fläche})^N$$

N = Maß für die Komplexität, 40nm ≈ N = 11 – 15

Hardware – Entwurf

Y-Diagramm, Top-Down-Entwurf, Bottom-up-Entwurf

Automatische Synthese, VHDL

Verhalten → HLS → RTL → Logiksynthes → Gatterbeschreibung → Layout-Synthese → Geometriebeschreibung → Fertigung → Chip

Bewertung der Leistungsfähigkeit

Antwortzeit / Ausführungszeit / Latenz: Zeit zwischen Start und Beendigung einer Aufgabe, interessant für den Nutzer

"A ist n-mal schneller als B", wenn:

$$\frac{\text{Ausführungszeit (B)}}{\text{Ausführungszeit (A)}} = n$$

Sicht des Rechenzentrumleiters:
"A ist n-schneller als B"

$$\frac{\#\text{Jobs (B)}}{\#\text{Jobs (A)}} = n$$

Dies betrachtet throughput / Durchsatz

Spielt eine Rolle für die Bewertung von multi-core Systemen

CPU-time := Zeit, in der die CPU arbeitet
User CPU-time := Zeit, in der die CPU ein Nutzer-Programm ausführt
System CPU-time := Zeit, die die CPU benötigt, um Betriebssystem-Aufrufe abzuarbeiten, die durch das Nutzerprogramm angefordert wurden

Gleichung für die Leistung einer CPU, feste Taktrate

$$\Rightarrow T_{exe} = \underbrace{IC}_{\text{instruction count}} \times \underbrace{CPI}_{\text{cycles per instruction}} \times \underbrace{TC}_{\text{Time per cycle}}$$

$\#$ instructions

$\Rightarrow CPI = T_{exe} / (IC \times TC)$ ↯ heutzutage ungenau wegen Komplexität der CPU

$\Rightarrow IPC = 1/CPI$

Probleme:

$$MIPS = \frac{\#\text{ausgeführte Instruktionen}}{10^6 \times \text{Ausführungszeit}}$$

$$MFLOPS = \frac{\#\text{ausgeführten Gleitkommaoperationen}}{10^6 \times \text{Ausführungszeit}}$$

→ Abhängigkeit von ISA & ausgeführter Befehlssequenz
⇒ Vergleich von Rechnern mit unterschiedlichen ISA schwierig
⇒ Unterschiedliche MIPS / MFLOPS bei verschiedenen Programmen
Ansonsten: niedriger Aufwand, bewerten aber nur spezielle Aspekte, beschreiben hypothetische Maximalleistung

Benchmarks (Programm / Programmsammlung zur Leistungsbewertung)
müssen compiliert werden
Güte von Compiler & Betriebssoftware fließen in das Ergebnis ein
Kernel := Rechenintensiver Teil des Programms
oft numerische Aufgabe
Anforderung an Benchmarks:
gute Portierbarkeit
repräsentativ für typische Nutzung der Rechner
Geometrisches Mittel $\neq \sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}}$
vorteil: unabhängig von Referenzmaschine:

$$\frac{\text{Geometrisches Mittel } (X_i)}{\text{Geometrisches Mittel } (Y_i)} = \text{Geometrisches Mittel } \left(\frac{X_i}{Y_i}\right)$$

SPEC = Standard Performance Evaluation Corporation

HW-Monitore
keine Beeinflussung der Laufzeit durch unabhängige physikalische Geräte

SW-Monitore
Einbau in das OS => Beeinträchtigung der normalen Verhältnisse

Modelltheoretische Verfahren
- unabhängig von der Existenz eines Rechners
- Abstraktion auf das Wesentliche

Ziele: Aufdecken von Zusammenhängen von Systemparametern
Ermitteln von Leistungsgrößen (Auslastung der Prozessoren, mittlere Antwortszeit, Queuelängen, ...)

Analytische Methoden
mathematischer Ansatz, um eben genannte Ziele zu erreichen
meist wenig Aufwand, aber auch wenig aussagekräftig
- Warteschlangenmodelle (Gesetz von Little $k = \lambda \cdot t$, bzw $Q = \lambda \cdot w$
- Petrinetze k = Ø # Aufträge Q = Ø Länge Queue
- Diagnosegraphen λ = Durchsatz w = Wartezeit
- Netzwerkflussmodelle t = Antwortzeit

Simulation
- Zentrales Mittel zur Evaluierung neuer Ideen
und zum Erkunden des Entwurfraums
Genauigkeit einstellbar
Wesentliche Modellierung der Zielmaschine in ihren Eigenschaften
User-Level-Simulatoren (keine Systemressourcen)
Full-System-Simulatoren (I/O, disk, ...)
Funktionale " (keine µ-Arch. nur Funktion)
Zyklen-Genaue " (alle Details der µ-Arch)
Für Benchmarks:
Trace-Driven-" (schreibt alle ausgeführten Befehle in eine Trace-Datei, ist eingabe für den Zyklen-genauen-Simulator)
Execution-Driven-" (erhält Benchmark Binary als Input)

Zuverlässigkeit & Fehlertoleranz

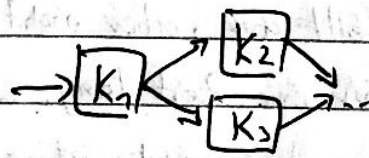
= Dependability
Fähigkeit des Systems, während einer Zeit t die vorgegebene Funktion unter gegebenen Constraints korrekt auszuführen

= fault tolerance
Fähigkeit des Systems, auch mit einer Anzahl fehlerhafter Subsysteme Zuverlässigkeit zu erfüllen.

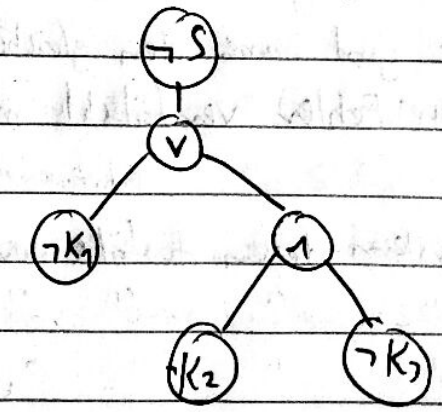
Kenngrößen der Dependability:
Verfügbarkeit, Überlebenswk'eit
Fehlerursache → Fehlerzustand → Fehlerwirkung
Fehlertoleranz soll Zuverlässigkeit erhöhen durch Vermeidung / Behebung / Maskierung der Fehlerzustände bevor sie Wirkung erlangen

Struktur-Funktions-Modell
Binäres Fehlermodell
$z(S) \rightarrow \{\text{wahr}, \text{falsch}\}$ bildet Komponenten auf Fehlerfreiheit ab.
$z(S,t) \rightarrow \bigwedge_{t_0 \leq t} z(S,t_0)$ ≙ System seit Beginn an fehlerfrei

Nicht-Redundantes-System F
$z(F) = \bigwedge_{K \in F} z(K)$
Systemfunktion $f(K_1, \ldots, K_n)$ gibt an, wie sich die Funktion des Gesamtsystems aus seinen Komponenten herleiten lässt
$S = K_1 \wedge (K_2 \vee K_3)$
$z(S) = z(K_1) \wedge (z(K_2) \vee z(K_3))$



Fehlerbaum ≙ $\neg S \Rightarrow \neg K_1 \vee (\neg K_2 \wedge \neg K_3)$



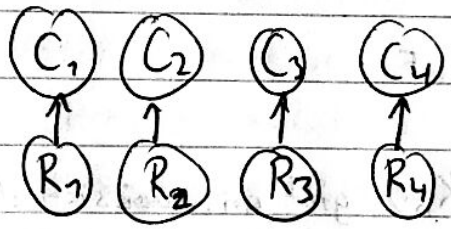
Fehlerbereich $B \subset S \hat{=}$ Menge von Komponenten, die zugleich fehlerhaft sein darf, ohne dass S fehlerhaft wird.

$B_1 = \{K_2\}$ und $B_2 = \{K_3\}$

Einzelfehlerbereich: sei für S eine Menge von Fehlerbereichen definiert, genannt Γ. Alle Komponenten, die genau den gleichen Fehlerbereich angehören seien eine Menge, genannt Einzelfehlerbereich

Perfektionskern

Das Komplement der Vereinigung aller Fehlerbereiche



$\Gamma = \{B_1, B_2\}$

$B_1 = \{R_1, R_2, C_1, C_2\}$

$B_2 = \{R_2, R_3, C_2, C_3\}$

Einzelfehlerbereiche $E_1 = \{R_1, C_1\}$, $E_2 = \{R_2, C_2\}$, $E_3 = \{R_3, E_3\}$

Perfektionskern: $P_1 = \{R_4, C_4\}$

Teilausfall: min. 1 Komponente fällt aus, aber nicht alle.

Unterlassungsausfall: Kein Ergebnis für eine Zeit lang, wenn es dann aber ausgegeben wird, ist es korrekt

Anhalteausfall: Komponente gibt nie wieder ein Ergebnis aus

Haftausfall: Komponente gibt immer den gleichen (falschen) Wert aus

Binärstellenausfall: Ein Fehler verfälscht mind. 1 Stelle des Ergebnisses

Fail stop-System: System, dessen Ausfälle nur Anhalteausfälle sind

Fail Silent-System: " " " " Unterlassungsausfälle "

Fail Safe-System: " " " " unkritische Ausfälle "

Fehlereingrenzung

Vertikal: Schichten prüfen die Funktionsautark vor Ausführung
unzulässiger Befehlscode wirft Fehlermeldung
Fehlerkorrekturcode im RAM
Plausibilität und Konsistens der Ergebnisse prüfen

Horizontal
räumliche, elektrische, thermische Isolation von Knoten
Redundanz

Vorgehensweise zur Erfüllung der Anforderungen:

Vermeidung { Sorgfalt beim Entwurf, Perfektionierung, Verwendung zuverlässiger Komponenten

Toleranz { Redundanz

Fehlervorgabe besteht aus: Fehlermodell + Menge zu tolerierender Fehler

Menge zu tolerierender Fehler ist formuliert bzgl. einer Fehlerbereichsannahme. Diese legt fest, wie viele Einzelfehlerbereiche gleichzeitig fehlerhaft werden können und welche Fehlfunktionen zu behandeln sind.

Behandlung braucht Zeitintervall

Zeitredundanz $t_z \triangleq$ Zeitintervall in dem keine weiteren Fehler auftreten

Fehlerbehandlungsdauer $t_F \triangleq$ Zeit, die nötig ist, um einen Fehler zu behandeln. $t_F \leq t_z$

Diese soll transparent, korrekt, schnell, günstig und unabhängig vom Rechensystem sein

Zuverlässigkeit / Sicherheit ✱ quantifizierbar mittels stochastischer Modelle:

Negative Zufallsvariablen:

Lebensdauer L, Dichte $f_L(t)$

Fehlerbehandlungsdauer B, Dichte $f_B(t)$

Sicherheitsdauer D, Dichte $f_D(t)$

Verteilungsfunktionen $F_x(t) = \int_0^t f_x(s)\,ds$ mit $x = L, B$ & D

$F_L(t)$ bezeichnet W'keit, dass ein zu Beginn fehlerfreies System in einem Zeitintervall $[0, t)$ fehlerhaft wird

$$F_L(t) = \frac{N_f(t)}{N}$$

$N_f(t)$: ✱ Komponenten, die bei t fehlerhaft sind

N: ✱ Komponenten

Reliability: $R(t) = \frac{N_s(t)}{N}$ $N_s(t)$: ✱ Komponenten, die bis t überleben

$$N_f(t) = N - N_s(t)$$

$$R(t) = 1 - F_L(t)$$

Fehler w'keit: $f_L(t) = \frac{d}{dt} F_L(t) = -\frac{d}{dt} R(t) = -\frac{1}{N} \times \frac{d}{dt} N_s(t)$

$\Rightarrow F_L(t) = 0$ und $\lim_{N \to \infty} F_L(t) = 1$

$\Rightarrow R(0) = 1$ und $\lim_{n \to \infty} R(t) = 0$

Ausfallrate

✱ erwarteter ausgefallener Komponenten zum Zeitpunkt

$$t = f_L(t) \times N$$

Zum Zeitpunkt t verbleiben dann $N_s(t) = R(t) \times N$

Ausfallrate ist dann:

$$z(t) = \frac{f_L(t)}{R(t)} = \frac{1}{R(t)} \times \frac{d}{dt} F_L(t) = \frac{1}{R(t)} \times \frac{d}{dt} R(t)$$

Ist nun die Ausfallrate bekannt, so ergibt sich die Fehler w'keit aus der Anfangswertaufgabe

$$\frac{d}{dt} F_L(t) = f_L(t) = z(t) \times R(t) = z(t) \times (1 - F_L(t))$$

mit der Anfangsbedingung $F_L(0) = 0$

Die Anfangswertaufgabe hat die Lösung

$$F_L(t) = 1 - e^{-\int_0^t z(s)\,ds}$$

Bei einer konstanten Ausfallrate (Software) $z(t) = \lambda$, ist die Fehler w'keit folglich: $F_L(t) = 1 - e^{-\lambda t}$

Verfügbarkeit $\hat{=}$ W'keit, ein System zu einem beliebigen Zeitpunkt fehlerfrei anzutreffen. Zeitlicher Anteil der Benutzbarkeit des Systems:

$$V = \frac{E(L)}{E(L) + E(B)}$$ $E(B)$: erwarteter Reparaturzeit

Falls Fehler $\Rightarrow \neg$ Sicherheit:

Gefährdungs w'keit $F_D(t)$

Sicherheits w'keit $S(t) = 1 - F_D(t)$

Mittlere Sicherheitsdauer $E(D) = \int_0^\infty S(t)\,dt$

$\hat{=}$ Erwartungswert der Dauer, bis ein gefährdender Zustand eintritt

Funktions w'keit ϱ

$$\varrho(S) = \sum_{(K_1, \ldots K_n) \in f^{-1}(\text{wahr})} \varrho\left(\bigwedge_{i=1}^{n} K_i\right)$$

Nichtfunktions w'keit

$$\varrho(\neg K) = 1 - \varrho(K)$$

Seriensystem $\varrho(S) = \prod_{K \in \Lambda} \varrho(K)$

Parallelsystem $\varrho(S) = \sum_{\emptyset \neq A \in \Lambda} (-1)^{1+\#A} \cdot \varrho\left(\bigwedge_{K \in A} K\right)$

$S = K_1 \vee K_2 \Rightarrow e(S) = e(K_1 \vee K_2)$

$= e(K_1) + e(K_2) - e(K_1 \wedge K_2)$

Zuverlässigkeitsverbesserung $\Phi_{S_1 \to S_2} = \frac{e(\neg S_1)}{e(\neg S_2)} = \frac{1 - e(S_1)}{1 - e(S_2)}$

Redundanz

Dynamisch: mit Umschalter, falls das primäre System ausfällt

Die ungenutzte Komponente kann hierbei anderweitig genutzt werden:

Ungenutzte Redundanz: hält nicht

fremdgenutzte " : erbringt andere Funktionen, hat nichts mit dem im Notfall zu ersetzenden System zu tun und wird verdrängt, falls Redundanz nötig ist

gegenseitige Redundanz: erbringt unterstützende Funktionen zu anderen Komponenten ⇒ graceful degradation

Statisch: immer an, Komparator notwendig

Überstimmen von n fehlerhaften Komponenten möglich, falls $n+1$ fehlerfreie vorhanden sind

Parallelverarbeitung

Nebenläufigkeit ≙ Objekte können (und werden) komplett gleichzeitig abgearbeitet

Pipelining ≙ Bearbeitung eines Objekts in Schritte zerlegt, diese können dann überlappt angeordnet sein

Ebenen der Parallelität

Programmebene: OS verwaltet Programme, diese haben

Keine (kaum) Datenabhängigkeiten und Synchronisation

Prozessebene
- Programm wird in Prozesse geteilt, hat eigene Datenbereiche, besitzt Kommunikations- und Synchronisationsmerkmale

Blockebene
- Leichtgewichtige Prozesse (threads)
- Sequenziell ausgeführte Befehlsströme, die sich einen Bereich im RAM teilen für Kommunikation
- Synchronisation über locks

Anweisungs- und Befehlsebene
- Parallele Ausführung mehrerer Anweisungen
- VLIW / EPIC (Compiler ordnen Befehle um usw.)

Suboperationsebene
- Elementare Anweisung wird durch Compiler oder Maschine in Suboperationen aufgebrochen, die parallel ausgeführt wird
- Vektoroperationen

Granularität ergibt sich aus dem Verhältnis $\frac{\text{Rechenaufwand}}{\text{Komm. / Sync. Aufwand}}$
- Programm-, Prozess- und Blockebene oft "grob"
- Anweisungsebene eher "fein" ← selten "mittel"

Klassifikation eines Rechensystems
- früher nach Anordnung der HW
- heute * Befehls- oder Datenströme (MIMD, SIMD, SISD)
- Prinzip der Virtualität:
  - darunterliegende HW für den Nutzer transparent

Pipelining

RISC

einfache Maschinenbefehle

Load- & store - Architektur

Einzyklus - Maschinenbefehle ⇒ einheitliche Zeitverhalten

Ausführungszeit eines Befehls ≙ Zeit zum Durchlaufen der Pipeline

k - Stufen : k - Takte Ausführungszeit, k Befehle gleichzeitig in der Pipeline

Annahme: Ideale Verhältnisse

Laufzeit $T = n + k - 1$ n = # Befehle im Programm

Beschleunigung $S = \frac{n \cdot k}{n+k-1}$

Die Ausführungszeit einer einzelnen Instruktion bleibt unverändert und

Pipeline - Konflikte

Einfache Lösung: Pipeline-stall / No-op ⇒ Leistung ⇩!

Strukturkonflikte (Ressourcenkonflikte)

nicht alle Kombinationen von Befehlen kann die Pipeline ungehemmt ausführen. Bsp: 2 Schreiboperationen auf eine Registerdatei mit nur einem Schreibeingang

Datenkonflikte

bedingt durch Datenabhängigkeiten, Operand ist noch nicht verfügbar

Steuerkonflikte

bedingt durch Verzweigungen im Quellcode

Einschränkung skalarer Pipelines:

CPI >= 1 oder IPC ≥ 1 (Lösung: parallele Pipelines)

Pipeline stalls bremsen nachfolgende Instruktionen (Lösung 000 ")

Lange Latenzzeiten einer Instruktion (Lösung: diversifizierte ")

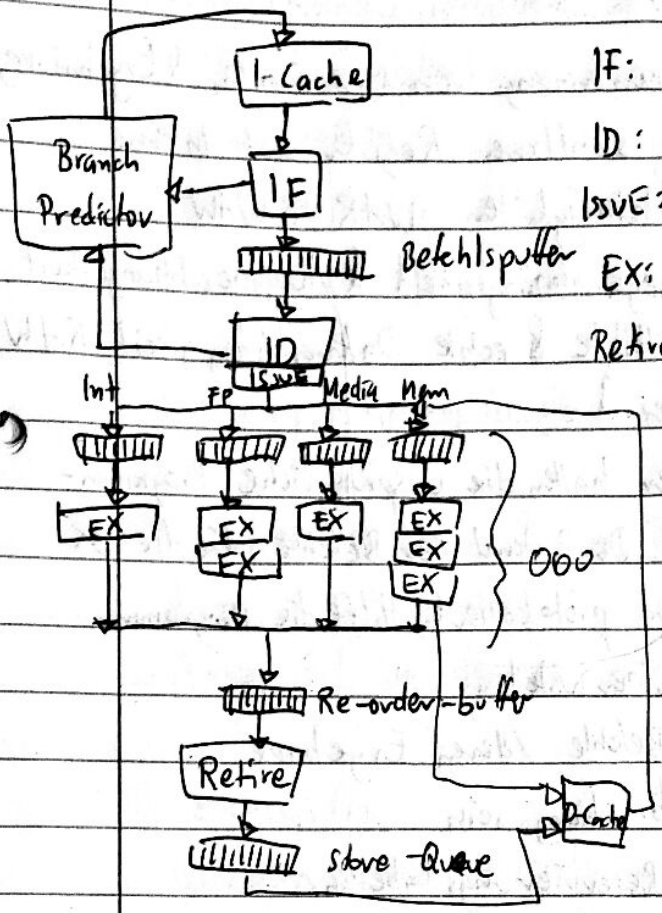
Nebenläufigkeit

dynamisch: super-skalare Pipeline

statisch: VLIW/EPIC

Superskalarität durch Mehrfachzuweisungsmethoden

$n > 1$ Befehle werden pro Takt geholt und beendet



IF: Instruction Fetch

ID: Instruction Decode w/ reg. renaming

Issue: Zuordnungseinheit

EX: Execution Units

Retire: Rückordnungseinheit

IF → Instruction Buffer

→ ID → Dispatch Buffer

→ Dispatch → Issuing Buffer

→ EX → Completion Buffer

→ Complete → Store Buffer

→ Retire

Branch-Predictors sorgt für spekulatives Holen von Befehlen. Seine Wahl ist zustandsabhängig & basiert auf Historie. Lauscht, welche Instruktionsadressen einen Sprungbefehl beherbergen und wie diese letzten Endes ausfielen. Die Tabellen zur Vorhersage werden dementsprechend aktualisiert.

Verwerfen & Rückrollen bei falscher Vorhersage nötig

BTAC (Branch-Target-Address Cache)

speichert Adresse der Verzweigung und letztes Ziel

BHT (Branch-History-Table)

Enthält Information über "taken" und "not-taken"

1 bit / 2 bit Prädiktoren

Hysteresemethode, springe bei TT von SNT zu ST, anstatt WT

bei NTNT von ST zu SNT, anstatt WNT

(m,n) Korrelations-Prädiktoren / 2 stufige adaptive Prädiktoren / Gselect und gshare-Prädiktoren / Hybridprädiktoren

ID-Phase: dynamische Umbenennung von Operanden & Ergebnisregister
Abbildung der nach außen sichtbaren Register auf Interne
⇒ Auflösung von Namensabhängigkeiten WAR & WAW
Gegenabhängigkeit & Ausgabeabhängigkeit
Dispatch löst Ressourcenkonflikte & echte Datenabhängigkeit RAW
Umordnungspuffer
Rückordnungspuffer halten die ursprüngliche Programmordnung fest. Der Zustand der Befehle, die die EX durchlaufen wird protokolliert. Hilft die Programmordnung wiederherzustellen
Completion: sammelt fertige Befehle / deren Ergebnisse
kann eine Unterbrechung sein
Bereinigung der Reservierungstabellen
Retire: Gültigmachen der Ergebnisse
⇒ sichtbar nach außen
Befehlsausführung ist vollständig, alle in der Programmordnung davor auch, keine Exception in der Zwischenzeit & korrekte Spekulation

Tomasulo
Jede FU hat eine Reservierungstabelle, die die Kontrolle über die Abarbeitung eines Befehls übernimmt
Ergebnisse werden über einen Ergebnisbus an alle FUs herangeführt

Empty	inFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
0	1	add	2	R3	1	0	R4	1	0

VLIW

breites Befehlsformat, mehrere Befehle pro Takt ausführbar
parallelisierende Compiler analysieren Kontroll- und Datenfluß & Datenabhängigkeiten und packen möglichst viele Befehle in ein Wort

Loop-Unrolling

```
for (int i=0; i<N; i++)        for (int i=0; i<N-3; i+=4)
    B[i] = A[i] +C      =>         B[i] = A[i]+C
                                   B[i+1] = A[i+1]+C
                                   B[i+2] = A[i+2]+C
                                   B[i+3] = A[i+3]+C
```

⇒ Prolog + Epilog in der Pipeline

Execution packets: Jeder Befehl, der auf '1' endet, kann parallel mit dem nächsten Befehl ausgeführt werden. Dieses Bit ist manuell setzbar

Multithreading

Reduzierung der Untätigkeitszeit und Überbrückung der Latenzzeiten
Ziel: Parallele Ausführung mehrerer Kontrollfäden, deren Kontext sicherbar und pausierbar ist ⇒ mehrere Registersätze & Programmzeiger

Cycle-by-Cycle Interleaving
Jeden Takt ist ein Kontextwechsel möglich

Block-Interleaving
Fäden werden gewechselt, sobald eine Latenz entsteht (Speicherzugriff o.Ä.)

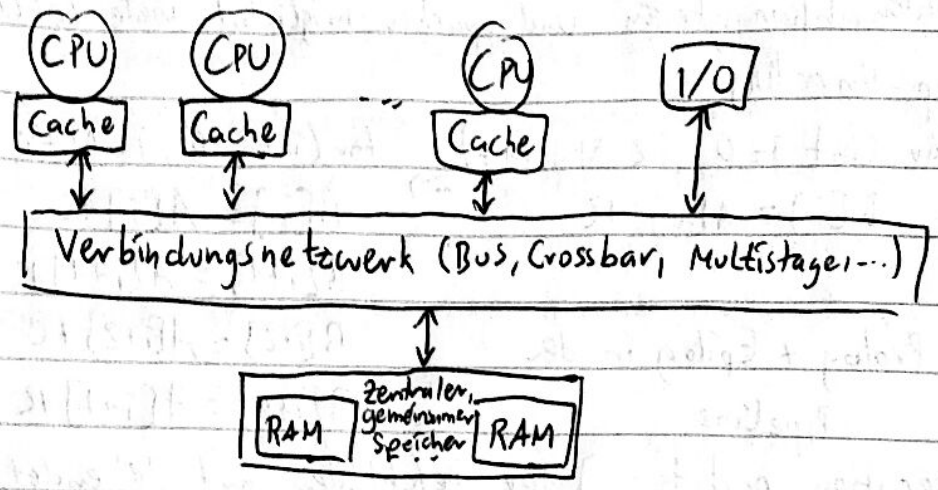
Simultaneous Multithreading
Mehrfach superskalarer Prozessor
Zuordnungseinheit erhält Befehle aus mehreren Befehlsströmen (Threads)

Multiprozessoren & Parallelismus auf Prozess-/Blockebene

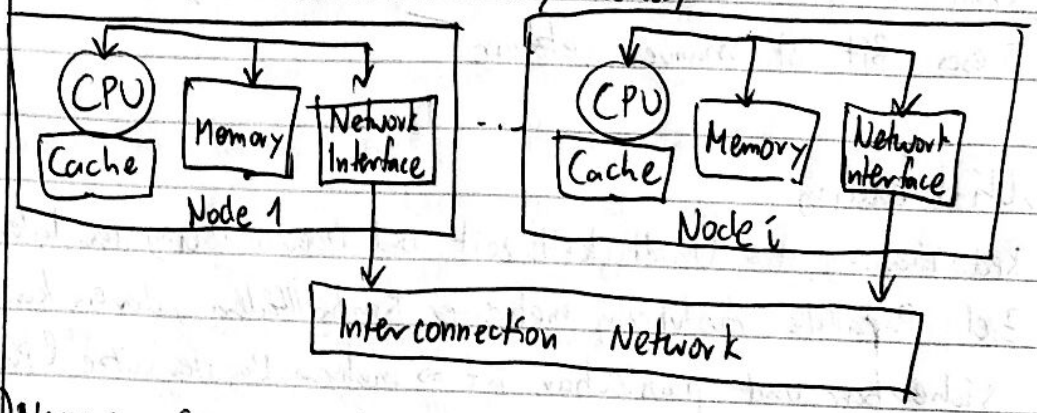
Architekturmodelle:

UMA Unified Memory Access

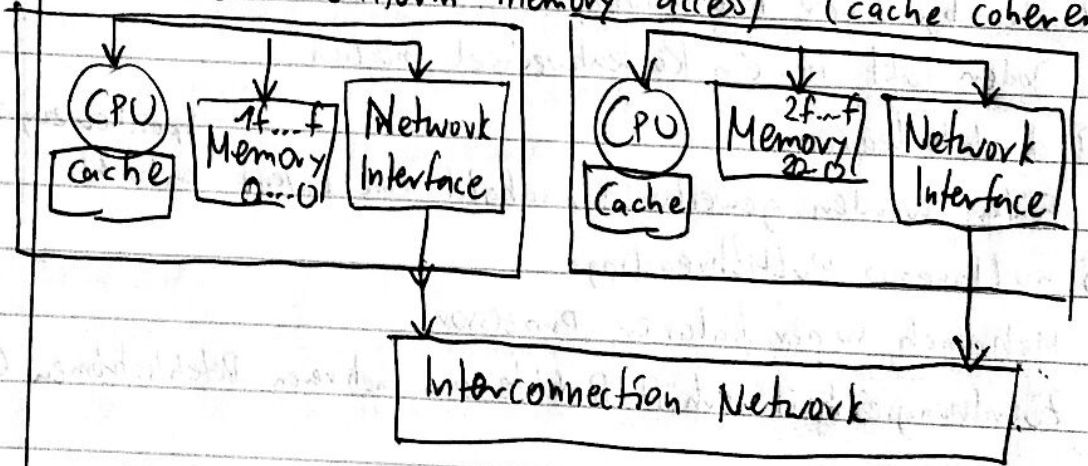
Symmetrischer Multiprozessor (SMP), Multicore



Norma (No remote Memory access)



(cc-)NUMA (Non-uniform memory access) (cache coherent)



Globaler Adressraum mit remote Zugriff

Formen des Parallelismus

Datenparallelismus ≙ Berechnungen von verschiedenen Datenelementen sind unabhängig (Matrixmultiplikation, ...)
SPMD (Single Program Multiple Data)

Funktionsparallelismus
Ein Programm lässt sich in unabhängige Teile zerlegen, die dann parallel ausgeführt werden können

Overhead: Synchronisation & Kommunikation
erhöht Ausführungszeit! Via geteiltem Speicher oder MPI

Kommunikation über gemeinsamen Speicher bedarf Schlossmechanismen: Mutex Lock / Unlock
Weitere Keywords shared / private / barrier / atomic / ...

MPI, bei keinem gemeinsamen Adressbereich
send() und receive() - Methoden

Befehle für MPI:
create(procedure) startet thread @ procedure
send(src_addr, size, dest, tag) Sende size Bytes von src_addr an dest mit tag
recv(buffer_addr, size, src, tag) Lege size viele Bytes in buffer_addr, wenn von src Nachricht tag kommt
barrier(name, number) Globale Synchronisation von number-Prozessen

Für shared memory
create(p, proc, args)
g_alloc(size)
lock(name)
unlock(name)
barrier(name, number)
wait_for_end(number)
wait_for_flag(flag)
set_flag(flag)

Quantitative Maßzahlen

Parallelitätsprofil, misst die Parallelität in einem Programm

Parallelitätsindex I

$$I = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} PG(t)\,dt \qquad I = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=n}^{m} t_i}$$

$$\frac{\#\text{Instruktionen}}{\#\text{Zeitschritte}}$$

$P(1) \hat{=}$ # auszuführender Operationen auf einem 1-CPU System

$P(n) \hat{=}$ # " " " " n- " "

$T(1) \hat{=}$ Ausführungszeit auf einem 1-CPU System

$T(n) \hat{=}$ " " " n- " "

Annahme: $P(1) = T(1)$ $\quad n \geq 2$

$T(n) \leq P(n)$

Speedup $S(n) = \frac{T(1)}{T(n)}$ $\quad 1 \leq S(n) \leq n$

Effizienz $E(n) = \frac{S(n)}{n}$ $\quad \frac{1}{n} \leq E(n) \leq 1$

Algorithmen unabhängige Definition

Vergleiche Parallelen Algo mit bestem bekannten sequenziellen Algo

⇒ absolute Effizienz & absoluter Speedup

Algorithmen abhängige Definition

Nutze den parallelen Algo so, als wäre er sequenziell und messe Laufzeit auf einem 1-CPU System

⇒ relative Effizienz & relativer Speedup

Mehraufwand $R(n) = \frac{P(n)}{P(1)}$ für die Parallelisierung $\quad 1 \leq R(n)$

Auslastung $U(n) = \frac{I(n)}{n} = R(n) \cdot E(n) = \frac{P(n)}{n \cdot T(n)}$

$\Rightarrow 1 \leq S(n) \leq I(n) \leq n$

$\frac{1}{n} \leq E(n) \leq U(n) \leq 1$

Zahlenbeispiel

$T(1) = P(1) = 1000$

$P(4) = 1200 \quad T(4) = 400$

$\Rightarrow S(4) = \frac{1000}{400} = 2,5$

$U(4) = \frac{1200}{4 \cdot 400} = 0,75$

$E(4) = \frac{2,5}{4} = 0,625$

$I(4) \frac{1200}{400} = 3$

$R(4) = \frac{1200}{1000} = 1,2$

Gesetz von Amdahl

Ein sequenzieller Teil in einem parallelen Programm begrenzt den möglichen Speedup signifikant

$S(n) = \frac{1}{1-a + a}$ für $n \to +\infty : S(n) = 1/a$

mit a = Anteil sequenziell

$a = 1/10 \Rightarrow S(n) \leq 10$

Superlinearer Speedup durch Cache-Hits möglich, die nicht auf einem 1-CPU-System möglich gewesen wären

Multithreading führt Deadlocks / Lifelocks ein, sowie Sättigungserscheinungen / Bottlenecks

Caches

Aktualisierungsstrategien

Cache-Zugriff	Write-Through (W/o write Alloc)	Write-Through (W/ write-Alloc)	Copy-Back (W/ write-Alloc)
Read-Hit	Cache-Datum → CPU	siehe links	siehe links
Read Miss	HS-Block, Tag → Cache HS-Datum → CPU, V=1	"	Cache-Zeile → HS V=1 HS-Block, Tag → Cache D=0 HS-Datum → CPU
Write-Hit	CPU-Datum → Cache, HS	"	CPU-Datum → Cache D=1
Write-Miss	CPU-Datum → HS	HS-Block, Tag → Cache, 1 → V CPU-Datum → Cache, HS	Cache-Zeile → HS HS-Block, Tag → Cache V=1, D=1 CPU-Datum → Cache

Gültigkeitsproblem

Mehrere Kopien liegen in verschiedenen Caches vor, jede CPU darf darauf schreiben
↑ des gleichen Datums

Cache-coherent ⇒ jeder Lesezugriff auf ein Datum liefert das zeitlich zuletzt geschriebene Datum auf dem gelesenen Datum

Problem mit DMA (zusätzlicher Master am Bus):

Write-through ⇒ CPU liest veraltetes Datum

Copy-back ⇒ DMA " " "

Lösung: non-cacheable-Data: immer cache-miss für CPU

(bei write-through) Cache-Clear: setze die cache-Einträge ungültig

(bei copy-back) Cache-Flush: alle "dirty"-Einträge werden zurückgeschrieben, danach alle ungültig

Kohärenz: definiert, welcher Wert bei einem Lesezugriff geliefert wird

Konsistenz: definiert, wann ein geschriebener Wert bei einem Lesezugriff geliefert wird

Write serialization ≙ 2+ Schreibzugriffe werden serialisiert und werden von allen Prozessoren in der gleichen Reihenfolge gesehen

Write-Invalidate: vor dem Verändern einer Kopie müssen alle anderen Kopien ungültig gemacht werden

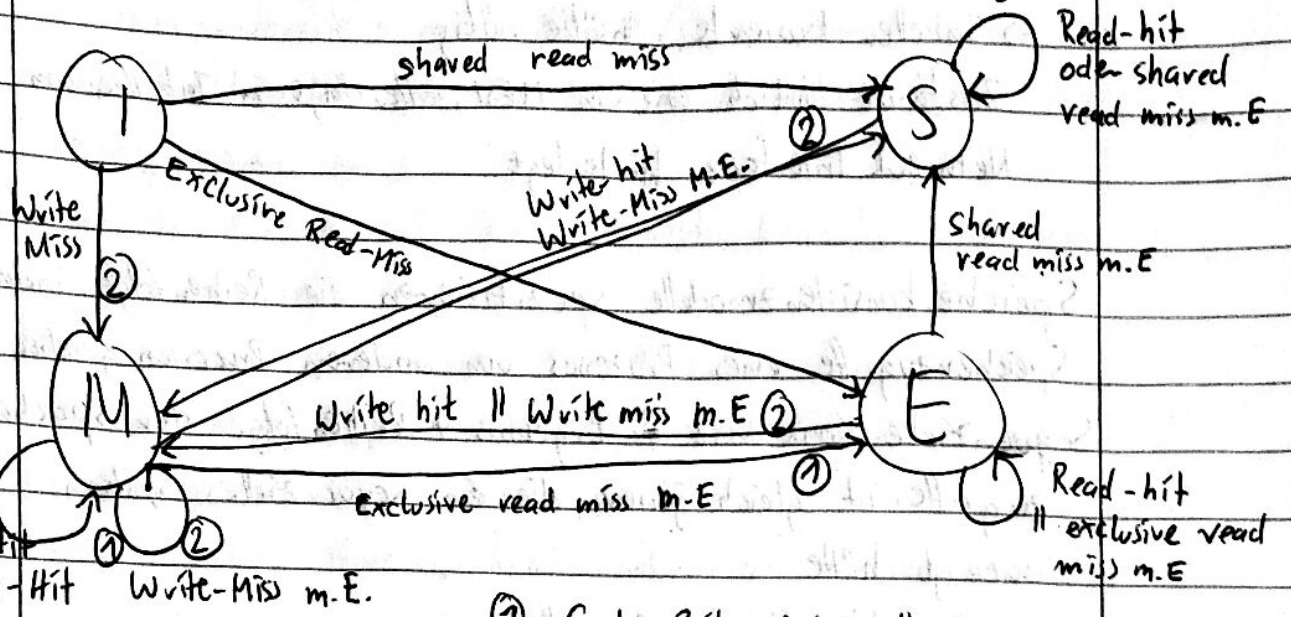
Write-Update: Nach dem Verändern müssen alle Kopien aktualisiert werden

Mehrere Broadcasts vs. 1 Invalidierung

Hardware Lösung: Tabellen-basierte Protokolle / Schnüffeln

MESI

Zustandsübergänge durch lokale Lese- und Schreibzugriffe

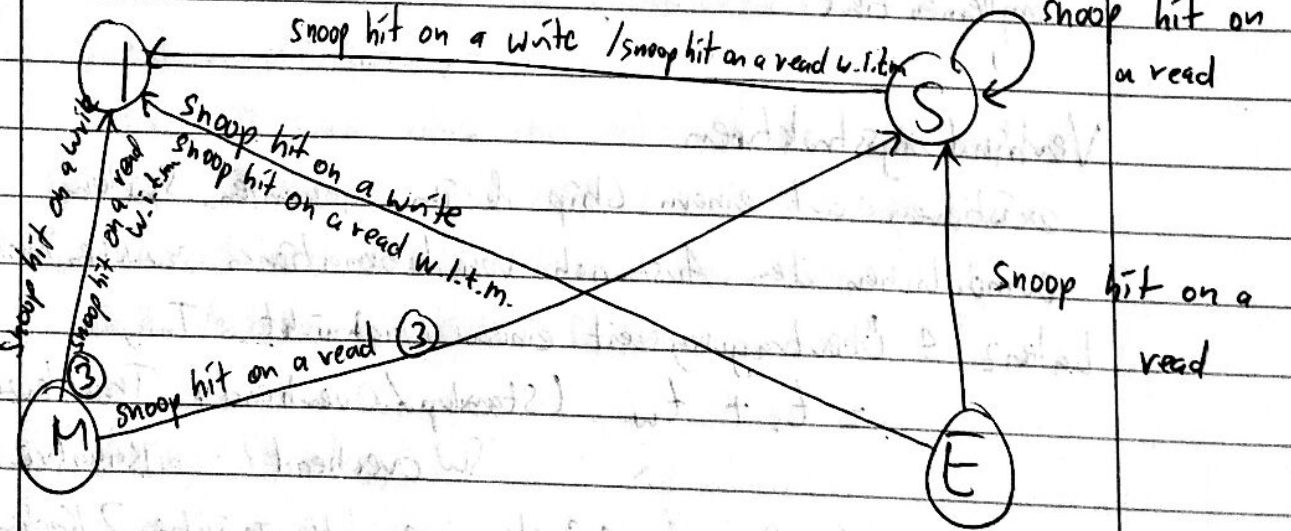


① Cache-Zeile wird in Hauptspeicher zurückkopiert

m.E. = gültige Cache-Zeile ② Kopien in anderen Caches werden invalidiert

wird ersetzt

Zustandsübergänge, die durch Schnüffeln ausgelöst werden



③ Retry Signal + zurückschreiben in Hauptspeicher

Bei distributed shared memory (DSM) gibt es keine Möglichkeit, zu schnüffeln
=> Tabellen-basierte Ansätze nötig
Zustände ähnlich wie in MESI, allerdings in Tabellen am Network Interface hinterlegt

Speicherkonsistenzmodelle spezifizieren die Reihenfolge, in der Speicherzugriffe eines Prozesses von anderen Prozessen gesehen werden
Sequenzielle Konsistenz ≙ Ergebnis + Reihenfolge der Speicherzugriffe ist gleich jener, die ein sequenzielles System erzeugt hätte
Abgeschwächte Konsistenzmodelle
Da sequenzielle Konsistenz sehr hinsichtlich Leistungseinbußen ist, wird Konsistenz nur noch zu Synchronisationszeitpunkten gefordert => mutex, lock, barrier, atomic, ...
Hürden eigenschaft durch spezielle Befehle in Hardware implementiert

Verbindungsstrukturen
existieren auf einem Chip & im gesamten System
ermöglichen den Austausch von Informationen zwischen Knoten
Latenz ≙ Übertragungszeit einer Nachricht: T_{msg}
$= t_s + t_w$ (Startup/Overhead + Transferzeit)
SW overhead Kanalverzögerung
Routing-Verzögerung ≙ Zeit, einen Weg zwischen 2 Knoten zu finden
Blockierungszeit ≙ Zeit, die anfällt, wenn gleichzeitig mehrere Nachrichten auf das Netz zugreifen

Durchsatz / Übertragungsbandbreite = max. Übertragungsleistung des Verbindungsnetzes (in MB/s oder Mb/s)

Bisektionsbandbreite = max. # MB/s, die das Netzwerk über die Bisektionslinie, die das NW in 2 gleiche Hälften teilt, transportieren kann

Diameter r = max. Distanz (# Verbindungen), die zwischen 2 Knoten liegen

Verbindungsgrad eines Knotens P = # Verbindungen von P

Mittlere Distanz d_a zwischen 2 Knoten = # Links auf dem kürzesten Pfad " " "

P/d_a = max. # neuer Nachrichten, die von jedem Knoten in einem Zyklus in das NW eingebracht werden können

Charakteristiken:

Komplexität / Kosten messen den Aufwand für die Implementierung in HW ~ # Schaltelemente, ~ # Links

Erweiterbarkeit: begrenzt / stufig / nur bei Vervielfachung der Knoten?

Skalierbarkeit: Beibehaltung wesentlicher Eigenschaften bei Erhöhung der Knotenanzahl

Ausfalltoleranz / Redundanz

S und D sind über ihr NI mit dem IN verbunden

Source Destination Network Interface Interconnect

Switch = Schaltelement (kann Puffer haben)

Link ≙ Verbindung zwischen Switches

Switching Strategy

circuit switching (Durchschalte- oder Leitungsvermittlung)

NI baut Verbindung von S nach D auf, diese bleibt bestehen, bis das Paket versendet wurde. Keine Routinginformationen, blockiert aber den gesamten Pfad.

Packet switching

Pakete haben Routinginformationen, die von Switches gelesen und zur Wegefindung genutzt werden. Ressourcen nur so lange belegt, wie sie genutzt werden, allerdings entsteht an jedem Switch ein Routing-Overhead.

Mehrere parallele Verbindungen möglich, Konflikte nicht ausgeschlossen

⇒ Flusskontrolle

Routing-Algorithmus abhängig von der Topologie:

Beispiel: 4x4 Gitter → zuerst X-Richtung, dann y-Richtung

Latenz- & Bandbreitenmodelle: End-2-end packet latency model

Annahme: Paket bereit im Sendepuffer

Zeitmessung von da an, bis das Paket im Empfängerpuffer liegt

Zuerst muss das Paket in einen Umschlag (+ Error detection bits, header, trailer) gepackt werden.

Gesamtzeit = Sender OH + Time of flight + Switching time + transmission time + routing time + Receiver OH

Sender OH = Bauen des Envelopes und ablegen im Puffer des NI

Time of flight = bei circuit switching: bestimmt durch # Links, # Switches und Zykluszeit dieser (Topologie)

bei packet switching: hinzukommt Overhead durch Wegefindung

Transmission time = Zusätzliche Zeit, die benötigt wird, alle Bits einer Nachricht nach Ankommen des ersten Bits beim Empfänger zu übertragen

Hängt von der Linkbandbreite ab:

$(N_{Pk} \times f)$ Für ein Paket: $(N_p + N_E) / (N_{pn} - f)$

Routing time: bei circuit-switching: Zeit, um den Weg aufzubauen
" packet- ": Zeit, den Weg in jedem Schaltelement aufzubauen

Switching-time: wird von der switching-strategy bestimmt:
Store-and-forward: ein Paket wird vollständig übertragen, bevor es weitergeleitet wird (zwischen zwei switches)
Cut-through: Übertragung eines Pakets zwischen Schaltelementen in überlappter Weise

Receiver-Oh: Overhead bei Ablegen der Verwaltungsinformationen und Weiterleiten des Pakets aus dem Empfangspuffer

Payload > Packetsize ? ⇒ Aufteilen des Payloads auf mehrere Pakete
bei Packet-switching können alle Paket nacheinander gesendet werden ⇒ Pipeline: Zusammensetzen, senden, receiver Oh

End-to-end packet latency = Sender Oh + Time of flight +
Receiver Oh + Transmission time +
Routing time + (N-1) · (max (Sender Oh,
Transmission time, receiver Oh))

Nacheinander folgende Pakete: Langsamste Stufe der Pipeline bestimmt den Durchsatz

effektive Bandbreite = $\frac{\text{Paketgröße}}{\max(\text{Sender Oh, Empfänger Oh, Übertragungszeit})}$

Bei circuit-switching ist die Routing time:

$L \cdot R$ + time of flight $= L \cdot R + L = L(R+1)$

L = # Links

R = # Taktzyklen für die Routingentscheidung

End-to-end packet latency @ circuit switching:
Sender OH + L + N + L(R+1) + Receiver OH
= Sender OH + L(R+2) + N + Receiver OH
allerdings keine Beachtung der möglicherweise belegten Leitung

End-to-end packet latency @ packet switching @ store-and-forward
= Sender OH + N(L+1) + L·R + Receiver OH

End-to-end packet latency @ packet switching @ cut-through
= Sender OH + L + N + L·R + Receiver OH

Wormhole Routing-Modus:
bei keinen Blockierungen identisch zu cut-through
andernfalls bleibt der Kopf der Nachricht stehen. ⇒ alle
nachziehenden Teile der Nachricht bleiben auch stehen.
Es werden nur Phits festgehalten, die Wegeinformationen
enthalten ⇒ Flit ≤ Packet Länge

Topologien: statisch / dynamisch
statisch: vollvermaschung ($n^2 \times L$!!.)
Kette (r = N-1, keine Redundanz)
Gitter
Torus
Ring (unidirektional, bidirektional)
Chordaler Ring (Ring mit Redundanz)
Baum (Binär, Fat-Tree
K-ärer n-Kubus (n = Dimension, k = #Knoten, die einen Zyklus in einer Dimension bilden ⇒ $N = k^N$ Knoten)
Knotengrad 2n

Hyperkubus Komplexität = $(N \cdot \log_2 N)/2$

Diameter = $\log_2 N$

Verdopplung von n nötig, um 2 skalieren

e-Cube-Routing: Knotennummern in binär ⇒ benachbarte Knoten unterscheiden sich an der Stelle, an der die ~~Richtung~~ Dimension gewechselt werden muss

Dynamische Verbindungsnetze (geeignet für variablen Kommunikations-Bus: 1 Nachricht pro Zeiteinheit muss sein)

Alle empfangen sie (w · f = Bandbreite)

Reduzierung des Bus-Verkehrs durch Caches

Split-Phase Busprotokolle: geben den Bus nach einer Speicheranfrage (diese erzeugt Latenz bis zur Antwort) wieder frei. Wenn die Antwort dann da ist, erfragt der Speicher den Bus, um zu antworten.

⇒ Speedup bei verschränktem Speicher + Pipelining

Crossbar: Vollständig vernetztes Netzwerk aus Weichen

Alle N! Permutationen möglich, und unter Umständen blockierungsfreie Kommunikation zwischen allen CPUs

Kosten: N^2 Kreuzungen

2x2 Kreuzschienenverteiler: Kann durchschalten / vertauschen

Mehrstufige Verbindungsnetze: Schalternetzwerke / Permutationsnetzwerke

p Eingänge können gleichzeitig auf p Ausgänge geschaltet werden

$M(a_n, a_{n-1}, \ldots, a_2, a_1) = (a_{n-1}, \ldots, a_2, a_1, a_n)$

Perfect Shuffle: ersten ans Ende, Rest aufrutschen

Kreuzpermutation (Butterfly)

$K(a_n, a_{n-1}, \ldots a_2, a_1) = (a_1, a_{n-1}, \ldots, a_2, a_n)$

Vertausche äußere Beide

Tauschpermutation

$T(a_n, a_{n-1}, \dots, a_2, a_1) = (a_n, a_{n-1}, \dots a_2, \overline{a_1})$

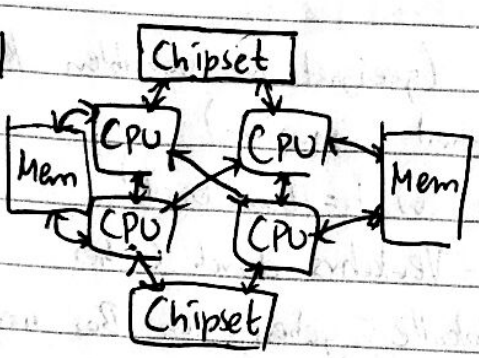
Wesentliche Eigenschaften:

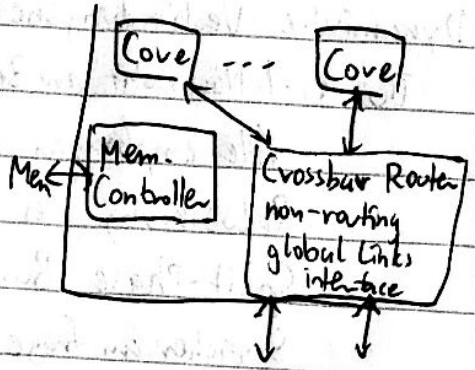
Mehrere Direktverbindungen parallel möglich

Schichtungen ähnlich wie in einem Protokoll-Stack

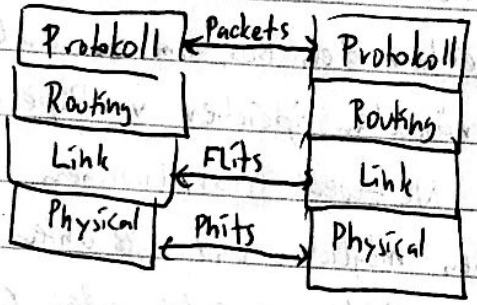
Header enthält Steuerinformation

QPI





Layers:



n Flits, Menge an Regeln zum Datenaustausch

← In der Firmware implementiert

1 Flit = 80b Flow control

1 Phit = 20b

72bit Payload
8bit CRC

QPI-Pout

Besteht aus 84 Leitungen ← Differenzpaar

↳ Paar von Drähten (Lane)

20 Lanes transmit, 20 Lanes receive 1 Lane clk

Logischer Wert der Leitung wird per Polarität eines Widerstands abgetastet

MultiLane-Distribution

Nachrichten: Fehlerkontrolle, Flusskontrolle, Cache Kohärenz

↑ Um den Empfänger nicht zu überfluten

Creditsystem

Multiprozessor mit verteilten Speicher

Nachrichtenorientiertes Kommunikationsmodell

Synchrones Message-Passing

Sender blockiert, bis die Nachricht beim Sender angekommen ist

Empfänger " , " " " angekommen ist

Deadlock-Gefahr

Synchronisation & Kommunikation gleichzeitig

=> Leistungseinbußen nicht unwahrscheinlich

Asynchrones Message-Passing

Blockiere erst bei RECV(), Berechnungen nach SEND() möglich, auch wenn noch keine Quittierung vorhanden ist

Nichtblockierendes RECV() muss mit probe-Funktion realisiert werden => Polling

Sender-initiiertes Message-Passing:

Request-to-send →

← Receiver ready

data →

Empfänger initiiertes Message-Passing:

← Request-to-receive

data →

DMA kann CPU entlasten, indem er das Message-Passing übernimmt

ebenfalls möglich durch dedizierte Kommunikationsprozessoren

Vektorprozessoren

SIMD-Verarbeitung

Verarbeitung von Vektoren in einem Rechenwerk mit Pipeline-artig aufgebauten FUs, die Vektoroperationen bereitstellen

```
L.D      F0,a      ; load scalar a
LV       V1,Rx     ; load vector 1
MULVS.D  V2,V1,F0  ; vector-scalar-multiply
LV       V3,Ry     ; load vector 2
ADDV.D   V4,V3,V2  ; add V4 = V3+V2
SV       Ry,V4     ; store Ry in V4
```

Skalareinheit zur Verarbeitung von arithmetischen Operationen auf Skalaren. Dies ist parallel zur Vektorverarbeitung möglich

Wenn die Pipeline gefüllt ist, ist mit jedem Takt ein Ergebnis zu erwarten.

Die Taktdauer ist durch die längste Teilverarbeitungszeit + Transferzeit zur nächsten Stufe gegeben

Beispielhafte Gleitkommaoperation: Addition

1. Laden der beiden Gk-Zahlen
2. Vergleichen der Exponenten
3. Ggf. schieben der Mantisse
4. Addition der Mantissen
5. Normalisieren des Ergebnisses
6. Schreiben des Ergebnisses

Ein Vektorprozessor ist eine Verkettung mehrerer

dieser spezialisierten Pipelines ⇒ stream processor
Daher kann es auch so genannte Multi-Funktions-Pipelines geben
diese haben mehr Stufen, für general-purpose, die
bei Nichtbedarf übersprungen werden können
Spezialisierte Pipelines führen eine Operation aus
Für wichtige Verknüpfungen von " benötigt man eine
Verkettung mehrerer dieser.
Vektor-Pipeline - Parallelität ist durch die Stufenanzahl
der Vektor-Pipeline gegeben
Pro Vektorrechner mehrere Vektor-Pipelines gegeben
Pro Takt kann ein Operandenpaar in mehrere Pipelines
gegeben werden

Parallelitätsebenen in Software
Vektorisierende Compiler können "innerste Schleifen"
vektorisieren
Spezialbefehle zum Ansteuern verketteter Vektor-Pipelines
Weitestgehend ähnliche Ideen wie bei parallelen CPUs
Nutzen von verschränkten Speichern ⇒ Zugriffszeiten auf
GRAM anpassen an GPU Takt
n Speicherbänke haben eine n-fache Verschränkung
Zugriffe auf die Speicher erfolgt zeitlich verschränkt
Zuteilung: $A_i \rightarrow M_j \Leftrightarrow j \equiv i \bmod n$

$A_0 \rightarrow M_0$

$A_n \rightarrow M_0$

$A_{2n} \rightarrow M_0$

$A_1 \rightarrow M_1$

$A_{n+1} \rightarrow M_1$

⋮

Nach einer gewissen Anlaufzeit werden in jedem Takt n Speicherworte geliefert

Vektor Stride
≙ Schrittweite im Speicher zwischen den nötigen Daten
Beispiel: Matrix im Speicher, Dimension = 3
↳ Stride zwischen den Zeilen: 1
Stride " " Spalten: 3
Vektorprozessoren können / müssen Strides > 1 verarbeiten können bei Zugriff auf nicht-sequenziellen Speicherzellen
Problem: Stridewert ist erst zur Laufzeit bekannt
Lösung: Ablegen des Stride-Wertes in Allzweckregister & Nötigkeit der Fähigkeit der Ladeeinheit, diesen berücksichtigen zu können
Falls Zugriffsfrequenz > Speichergeschwindigkeit:
nutze verschränkte Speicher

if-Anweisungen können nicht parallelisiert werden
Lösung: Bedingt ausgeführte Anweisungen durch Umwandlung von Kontrollflussabhängigkeiten in Datenabhängigkeiten
durch Vektor-Maskierungssteuerung
diese steuert die Ausführung eines Vektorbefehls
Vektor-Mask-Register
Eintrag '1' ⇒ Vektoroperation verändert Zielvektorregister an Stelle der '1'
Eintrag '0' ⇒ Stelle der '0' im Zielvektorregister bleibt unverändert

Dünn besetzte Matrizen / Vektoren werden mit Hilfe von Indexvektoren gespeichert. Diese enthalten Informationen über die Stellen, die nicht '0' sind.

Scatter / Gather Operationen unterstützen diese gepackte Darstellung:

Gather: holt den Vektor, dessen Elemente an den Adressen liegen, die sich aus Addition eines Basiswertes und den Inhalten des Index-Vektors ergibt.

Scatter: Speichert die gepackte Darstellung

```
LV    Vk, Rk        ; load K
LVI   Va, (Ra+Vk)   ; load A(K(i))
LV    Vm, Rm        ; load M
LVI   Vc, (Rc+Vm)   ; load C(M(i))
ADDV.D Va, Va, Vc   ; A += C
SVI   (Ra+Vk), Va   ; store A(K(i))
```

Problem für vektorisierende Compiler

erkennen nicht, wenn zwei Elemente innerhalb einer Iteration auf dieselbe Speicherzelle zeigen

Lösung: Verwendung von Software-Hash-Tabellen