HPR

Rechnerarchitektur (Disziplin)
Ingenieurswissenschaft: bestehende und zukünftige Rechenanlagen beschreiben, vergleichen, beurteilen, verbessern und entwerfen
Betrachtung der Eigenschaften des Ganzen (Rechenanlage), der Teile (Komponenten) und seiner Verbindungen (Globalstruktur / Infrastruktur)
Aufgabe der Kompromißfindung zwischen
Zielsetzungen (Einsatzgebiet, Anwendungsbereich)
Randbedingungen (Technologie, Größe, Geld, Energieverbrauch)
Gestaltungsgrundsätzen (Modularität, Fehlertoleranz)
Anforderungen (Kompatibilität, Standards, OS)
Unter Betracht: Desktop-PC vs. Server vs. ES

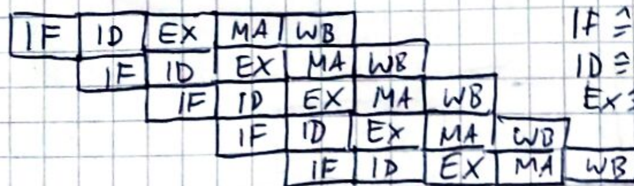
Beim Prozessorentwurf:
Moore's Law: ausnutzen der zur Verfügung stehenden Transistoren
Leistungsaufnahme: $P_{total} = P_{switching} + P_{static} + P_{shortcircuit} + P_{leakage}$
$P_{switching} = C_{eff} * f * (V_{dd})^2$
Steigerung der Rechenleistung nicht mehr durch Takterhöhung möglich (Dennard's Scaling)
=> Multi / Many core - Systeme
=> hierarchische Speicher / Cache - Strukturen auf einem Chip
=> NoC
=> adaptive Strukturen
=> heterogene Strukturen (Beschleuniger / FPGAs / GPUs)
Probleme mit der Verlustleistung (TDP) und dem Speicher
Memory Wall, Kapazität (Fest/ Flüchtiger Speicher)
Zugriffsgeschwindigkeiten
Zuverlässigkeit und Verfügbarkeit
Parallelität & Lokalität

Parallelismus auf Befehlebene

Pipelining: Zerlegen eines Maschinenbefehls in Teilschritte, die von dafür dedizierten Komponenten unabhängig und parallel abgearbeitet werden können
Die Stufen sind jeweils durch Pipelineregister getrennt

RISC ≙ Reduced Instruction Set Computers
einheitliches und festes Befehlsformat
Load / Store Architektur: Befehle arbeiten auf Registeroperanden
Lade / und Schreiboperationen greifen auf Speicher zu
Einzyklus Maschinenbefehle: effizientes Pipelining des Maschinenbefehlszyklus
einheitliches Zeitverhalten der Maschinenbefehle, wovon nur Lade / Speicherbefehle und Verzweigungen abweichen
Optimierende Compiler: Reduzierung der Maschinenbefehle im Code

5-stufige Pipeline für RISC



IF ≙ Instruction Fetch
ID ≙ Instruction Decode
EX ≙ Instruction Execute
MA ≙ Memory Access
WB ≙ Write Back

5 Befehle gleichzeitig

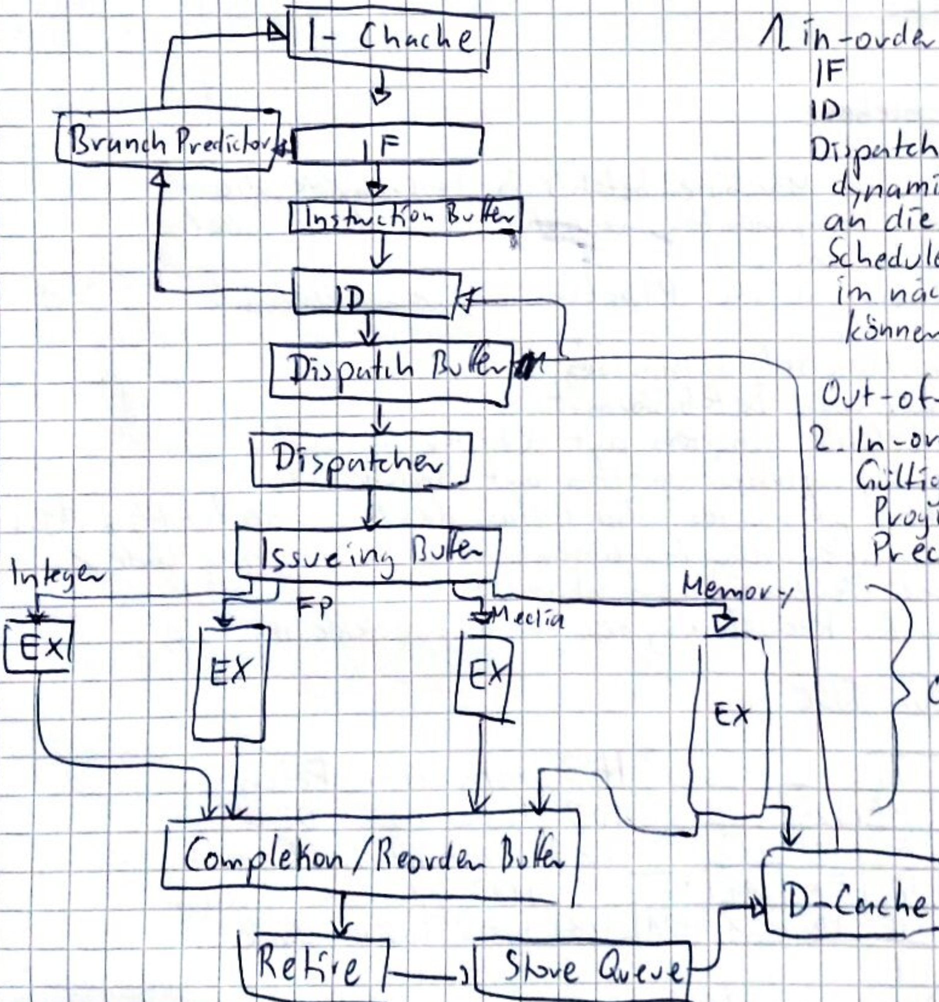
Pipeline-Konflikte ≙ Situationen, die es verhindern, dass die nächste Instruktion im Befehlsstrom im zugewiesenen Taktzyklus ausgeführt wird.
Erfordert Pipeline stall (anhalten) => Leistungseinbußen zum theoretischen Speedup
Bei „einfacher" Pipeline müssen auch alle Befehle nach dem Blockierten warten, die davor können weiter durchlaufen

1. Strukturkonflikte:
   Bedingt durch die Ressourcenbegrenzung.
   Beispiel: Gleichzeitiges mehrfaches Schreiben auf denselben Speicher nicht möglich
2. Datenkonflikte:
   Bedingt durch Datenabhängigkeiten (Echt-; Gegen- und Ausgabeabhängigkeit)
   RAW WAR WAW
3. Steuerkonflikte:
   Bedingt durch den Steuerfluss im Programm

Konfliktauflösung der Datenabhängigkeiten: Pipeline Bubbles (NOOP), Pipeline Stall } Leistungseinbußen

Ohne Einbußen: Forwarding
FU erkennt, dass eine Instruktion in MA oder WB in ein Eingaberegister der Instruktion in EX schreibt.
=> Pfad zwischen WB/MA nach EX
=> lediglich ein Takt einzubüßen

Superskalare Pipelines
Ziel: Erhöhung der Instructions per Cycle (IPC)
Superskalare RISC behalten folgendes bei:
- feste Wortlänge (32-bit, ..)
- Lade/Speicherarchitektur



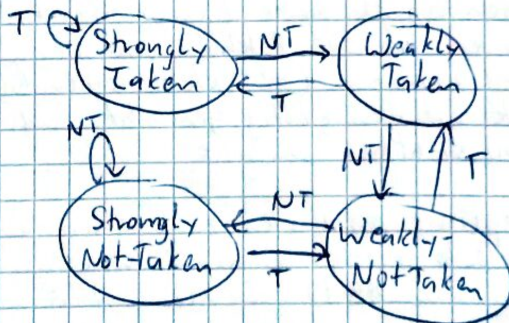
1. in-order-Abschnitt:
   IF
   ID
   Dispatch (Zuweisungsstufe)
   dynamische Zuordnung der Befehle an die Maschineneinheiten
   Scheduler bestimmt, wie viele Befehle im nächsten Takt zugeordnet werden können.

Out-of-order Abschnitt: EX
2. In-order Abschnitt:
   Gültigmachen der Ergebnisse gemäß Programmordnung
   Precise Interrupts / Spekulation

} Out of order

IF: holt mehrere Befehle aus dem Befehls-Cache in den Befehlsholpuffer
* entspricht der Zugriffsbandbreite (64 bit ...)
Welche Befehle geholt werden hängt von der Sprungvorhersage ab
Die Verzweigungseinheit überwacht die Ausführung von Sprüngen,
holt spekulativ Befehle, nutzt dafür dynamische Sprungvorhersage techniken,
unterstützt Rückrollen bei fälschlicherweise ausgeführten Befehlen
Der Befehlsholpuffer entkoppelt IF von ID
Spekulative Ausführung benötigt temporäre Speicherregister, damit eventuell fälschlich entstehende Ergebnisse nicht nach außen sichtbar sind. Pipeline muss Rückrollmechanismen unterstützen
Auftretende Exceptions dürfen nicht fälschlicherweise geworfen werden
Basisblock ≙ Abschnitt im Program, das sprungfrei ist
Bei korrekter Vorhersage: ausführen der weiteren Befehle (ohne Verzögerung)
Bei inkorrekter ": Rückrollen der Ergebnisse
Statisch: Für einen Befehl im Prozessor fest verdrahtet
Dynamisch: Wird zur Laufzeit entschieden, unter Berücksichtigung des Programmverhaltens
=> Mehr Hardwareaufwand
Realisierung durch BTAC (Branch-Target Adress Cache)
Speichert die Adresse der Verzweigung und des Sprungziels
Lookup-Tabelle, die die Sprungzieladresse für jeden Branch im Programmcode enthält
Diese wird anhand der Branch History table (BHT) aufgebaut. Hier kommen Prädiktoren zum Einsatz
Der Branch-prediction-Buffer (BPB) enthält anhand von BTAC und BHT die Sprungbits, die dann das Holen von weiteren Befehlen beeinflussen
Not Taken (NT) => PC += 4
Taken (T) => PC = BTAC
Nach Ausführung des Sprungbefehls wird die Vorhersage ~~[illegible]~~ verifiziert und (gegebenenfalls) aktualisiert
Aliasing: Zwei Branches können auf einen BPB Eintrag abgebildet werden => Korrelation der Sprünge. Kein Korrektheitsproblem
1-Bit Prädiktor:

2-Bit Prädiktor:



=> Sättigungszähler, saturiert bei 3 und 0
Inkrement für jeden genommenen Sprung,
Dekrement sonst

Bei Superskalaren Pipelines sehr aufwändig, eine gute Sprungvorhersage zu implementieren.
Zweistufige adaptive Prädiktoren
Gselect und gshare "
Hybrid- "
(m,n)-Korrelations "

ID: Dekodierung der im Befehlspuffer abgelegten Befehle
entspricht der Befehlbereitstellungsbandbreite
Bei CISC-Architekturen (IA-32)
Aufteilung in mehrere Schritte:
1. Bestimmung der Grenzen der geholten Befehle
2. Dekodierung
3. Generierung RISC-ähnlichen Befehle
4. Dispatch
Registerumbennenung
Dynamische Umbenennung der Operanden- und Zielregister
Abbildung von nach Außen sichtbaren auf interne, physikalische Register
Jedes Zielregister wird auf ein noch unbelegtes physikalisches Register abgebildet (# kann größer sein, als die externen Register)
=> Auflösung von Namenskonflikten
Schreiben der Befehle in ein Befehlsfenster
=> Befehle sind frei von Steuerflussabhängigkeiten (durch Branch-prediction)
und " " Namensabhängigkeiten (durch Umbenennung)

Dispatch: Zuordnung der im Befehlsfenster wartenden Befehle zu den Ausführungseinheiten
Dynamische Auflösung der Konflikte von echten Datenabhängigkeiten und Ressourcenkonflikten
Zuordnung bis zur maximalen Zuordnungsbandbreite pro Takt
Festhalten der ursprünglichen Programmreihenfolge
Protokollierung jedes Befehls und dessen Ausführungszustand
Vor den Ausführungseinheiten liegen die Umordnungspuffer
kann geteilt werden
Befehle müssen warten, wenn diese voll sind (=> Ressourcenkonflikt)

EX: Ausführung der im Opcode spezifizierten Operation und schreiben des Ergebnisses im Umbenannten Zielregister
Unterscheidung von Einzyklusoperationen und Mehrzyklusoperationen

Completion: Befehlsausführung beendet, Ergebnis wird gepuffert und eventuell forwarded.
Bereinigung der Reservierungstabellen
Aktualisierung des Zustands im Rückordnungspuffer
Es kann eine Exception angezeigt werden

Retire: Commitment <=> Befehl completion && Befehl hängt von keiner Spekulation ab && keine Exception vor oder während EX geworfen && alle Befehle vor diesem Befehl sind committed worden (=> Befehlsreihenfolge!)
=> Befehlsresultate werden gültig gemacht und aus den Schattenregistern sichtbar gemacht

Precise Interrupts: Exceptions dürfen erst dann gültig gemacht werden, wenn sie erfolgreich committed wurden.
Alle Resultate vor der PI werden gültig gemacht. Alle danach verworfen, die Berechnungen werden abgebrochen
Retten des Zustands
Einleiten der Behandlungsroutine
Wiederaufnehmen des unterbrochenen Ablaufs
Mehrere Alarme können gleichzeitig in einem Takt in verschiedenen Stufen entstehen

Warten bis in WB-Stufe, jede Instruktion führt ein Exception Status Register (ESR) mit sich. Diese " wird dann wie eine NOOP behandelt.
Problem, wenn in MA ein Speicherschreibzugriff erfolgt und dann in WB eine Ausnahme Verarbeitung beginnt
Muss Rückgängig gemacht werden
Mithilfe von Rückordnungspuffer, Completion und Retire - Phasen

Dynamische Methoden zur Erkennung und Auflösung von Datenkonflikten
Detaillierte Betrachtung der Zuordnungsphase (Dispatch)
Fallstudie: Tomasulo (dynamisches Scheduling)
Konfliktauflösung und Ablaufsteuerung verteilt
Jede Funktionseinheit verfügt über eine Reservierungstabelle (Umordnungspuffer, Reservation Station)
Übernimmt die Kontrolle über die Abarbeitung eines Maschinenbefehls, wenn dieser von der Dekodiereinheit zur Ausführung angestoßen wurde (issue)
ein von einer Funktionseinheit i produziertes Ergebnis wird direkt an die Funktionseinheit j weitergegeben, wenn diese das Ergebnis als Operand benötigt
Ergebnisbus:
Alle Funktionseinheiten, die auf einen Operanden warten, werden gleichzeitig bedient
Inhalt der Reservation Stations (RS):
Feld für die zwei möglichen Operanden (Src 1, Src 2)
Die Nummern (Tags) der RS derjenigen Funktionseinheiten, die die Quelloperanden für die auszuführende Operation liefern werden (RS 1, RS2)
Jeweils ein Flag, der anzeigt, ob der Operand verfügbar ist (vld 1, vld2)
Einen Namen (destination-tag) für das Ziel (dst)
=> Maschinenbefehle können angestoßen werden, wenn in der RS noch ein Eintrag frei ist. Andernfalls muss gewartet werden.
Dann wird für jeden angestoßenen Befehl die Inhalte seiner Quellregister und die dazugehörigen ready-Bits in die RS der Ausführungseinheit kopiert.
Phasen: 1. Issue: Holen der Befehle aus der Instruction-Queue
Dispatch Holen der Operanden
2. Ausführen: wenn die Operanden verfügbar sind
Beobachten des Ereignisbusses (RAW-Konflikte überwacht)
Out-of-order
3. Rückschreiben: Schreiben der Ergebnisse auf den Ereignisbus
Kennzeichnen des Umordnungspuffers als „verfügbar"

VLIW
Breites Befehlsformat mit mehreren Feldern, die Funktionseinheiten steuern können
Bis zu n beim Funktionseinheiten
Operationen auf RISC Architekturebene
Erzeugung zur Übersetzerzeit
Analyse von Kontrollfluss (Steuerfluss), Datenfluss, Datenabhängigkeiten, Schleifenparallelisierung, Software Pipelining, Scheduling, Loop unrolling
Software Pipelining:
Technik zur Reorganisierung von Schleifen
Jede Iteration in dem mit Software-Pipelining generierten Code enthält Befehle aus verschiedenen Iterationen der Schleife

Multiflow Trace
- globales Befehlsscheduling (über Basisblockgrenzen)
Trace selection: Auswahl des Pfads (trace) durch den Kontrollflussgraph anhand statistischer Aussagen oder profiling
Lange Befehlssequenz mit Kompensierungscode, falls die Vorhersage falsch war.
Packen von unabhängigen Befehlen in VLIW

1 für „kann mit dem folgenden Befehl parallel ausgeführt werden"
0 für „ nicht " " " " "
Am Ende immer 0

Vergleich HW schedule vs SW schedule

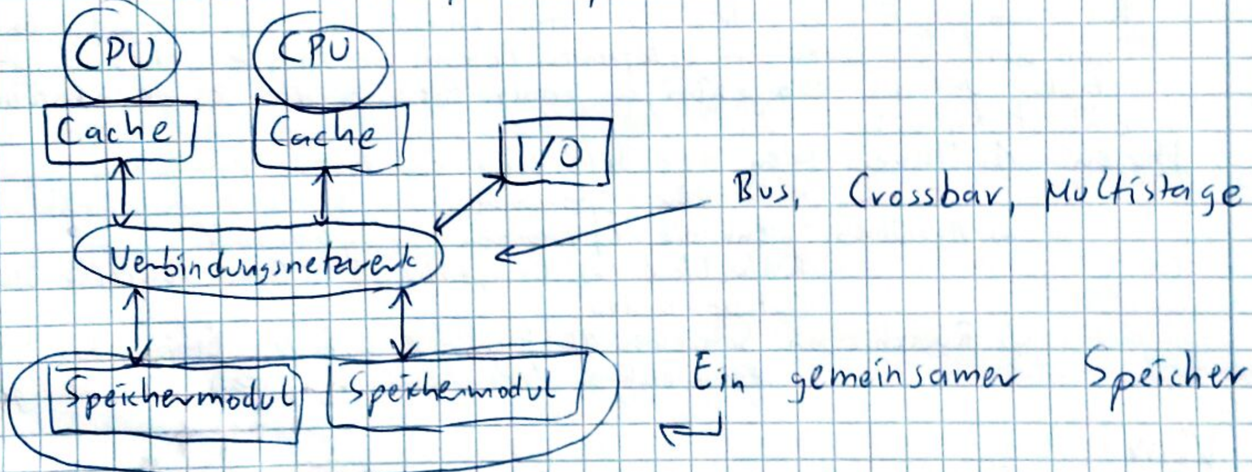
+ verbesserte Speicherdisambiguierung, da zur Laufzeit die Adressen bekannt sind
+ Sprungvorhersage
+ Precise Interrupts
+ Kein Kompensierungscode
+ Kompatibilität für mehrere Implementierungen

− hoher HW Aufwand
− wenig Spielraum für das Finden von Parallelisierungsmöglichkeit

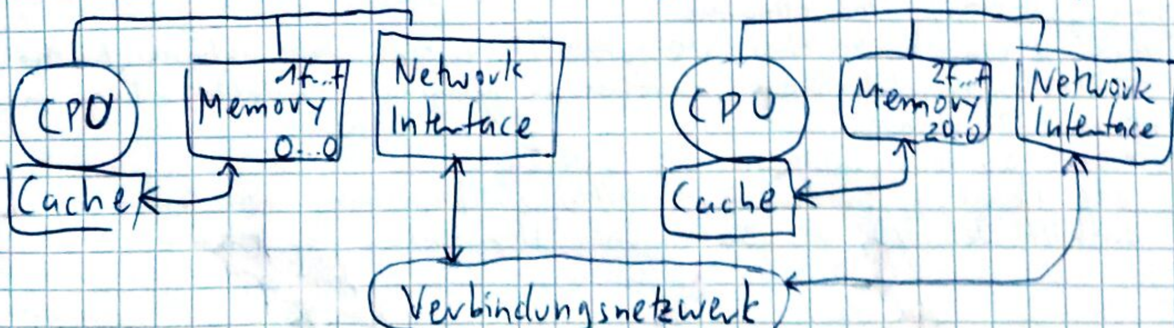
Parallelrechner ⇒ Kommunikation (Netz)

Parallele Architekturen:

UMA (Uniform Memory Access)



NUMA (Non-UMA)
CC_NUMA (Cache-Coherent NUMA)
Globaler Adressraum, Zugriff auf entfernten Speicher



NORMA (No Remote Memory Access)

Programmiermodell: Abstraktes Modell einer parallelen Maschine, auf der der Programmierer sein Programm formuliert
Spezifiziert, wie Teile parallel abgearbeitet werden
, wie Informationen ausgetauscht werden und
welche Synchronisationsoperationen verfügbar sind, um
die Aktivitäten zu koordinieren
Anwendungen werden auf Grundlage eines parallelen Programmier-
modells formuliert
Multiprogramming: Menge von unabhängigen parallelen Programmen
⇒ keine Koordination oder Kommunikation
Bei gemeinsamen Speicher:
Kommunikation und Koordination über geteilte Variablen und
Zeiger die gemeinsame Variablen Referenzieren
⇒ Verwendung conventioneller Speicheroperationen
Atomare Synchronisationsoperationen
Alternative / Andernfalls:
Message Passing
Kein gemeinsamer Adressbereich
Verwendung von send- und receive Operationen
Send (X, Q, t) X wird an Q geschickt, Label t
receive (Y, P, t) An Y wird gespeichert, was P mit Label t schickt

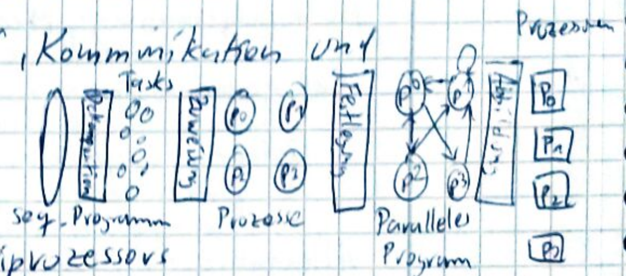
Datenparallelismus
Gleichzeitige Ausführung von Operationen auf getrennten Elementen einer Datenmenge (Array, Matrix, ...)
Typischerweise in GPUs, vermehrt auch in CPUs
Spezielle Form: SPMD (Single Program Multiple Data)
Ein Programmcode wird auf alle Datenelemente angewendet
Funktionsparallelismus
Unabhängige Funktionen werden verschiedenen Prozessen ausgeführt
Task-Level-parallelism
Beispiel: Streaming Anwendung
Können Funktionspipelines bilden

Ziele der Parallelprogrammierung
Speedup gegenüber sequenzieller Abarbeitung
Ausgewogene, effiziente Aufteilung der Arbeit auf die Rechenknoten
Minimierung des anfallenden Overheads (Kommunikation & Koordination)
Vorgehensweise:
Festlegung der parallel ausführbaren Aufgaben
Aufteilen auf Rechenknoten
für Berechnung
Datenzugriff
Ein/Ausgabe
Verwaltung des Overheads
Großteils Aufgabe des Programmierers
Mithilfe des Compilers, der Laufzeitumgebung und des Betriebssystems umsetzbar

Ocean: Simulation der Bewegung der Wasserströmung im Ozean
Ganze Welt modelliert
Wechselseitige Beziehungen bedingen komplexes Verhaltensmodell
Lösung komplexer Gleichungssysteme erfordern Numerische Lösungsverfahren (Jakobi / Gauss-Seidel-Verfahren)
Physikalisches Modell ist kontinuierlich in Raum & Zeit
Diskretisierung erforderlich
Kompromiss zwischen Genauigkeit der Samples gegen die Rechenzeit

Parallelisierungsschritte:
1. Aufteilung / Dekomposition der Aufgabe in Tasks (= kleinste Parallelisierungseinheit)
2. Zuweisung der Tasks an Prozesse
3. Festlegung / Orchestrierung von Datenzugriff, Kommunikation und Synchronisation zwischen den Prozessen
4. Abbilden der Prozesse auf Prozessoren

1-3 auch „Partitionierung" genannt



Prozess / Thread ≙ Virtualisierung eines Multiprozessors
nicht zwingend gleich # Prozessoren
Jeder Prozesse führt eine Teilmenge der Tasks aus

Prozessor ≙ physikalische Ressource
Führt einen oder mehrere Prozesse aus
Überbuchung je nach Anwendung sinnvoll
Maskierung von Mehrzyklusoperationen und Latenzen
HW - Beispiel: Hyperthreading
SW - " : OpenMP

Tasks können zur Laufzeit dynamisch erzeugt werden, ihre # stellt die Obergrenze für # Prozessen dar.
Tasks können in einen Prozess gruppiert werden
Asynchrone Methode für parallele Programme: Ignorieren der Datenabhängigkeiten
⇒ Ordnung der Tasks auf Prozessen kann nicht vorhergesagt werden
Ausführung ist nicht deterministisch

Architektur, Programmiermodel und Programmiersprache spielen eine Rolle bei der Festlegung von Tasks an Prozesse
Es bedarf Mechanismen für die Kommunikation, den Datenzugriff und die Orchestration und Synchronisation
Ziel: Reduzierung des Overheads und des Parallelisierungsaufwandes
Erhalten der Lokalität der Datenzugriffe
⇕
Festlegung von: Programmiermodell:
Shared-Memory / Nachrichtenorientiert
Datenparallelismus HW / SW

Pragmas; Shared Memory

create (p, proc, args) // Erzeuge Process p, der Ausführung von proc mit den Argumenten args startet
g_alloc (size) // Allokation eines size-großen gemeinsamen Speichers
lock (name) // Fordere wechselseitigen Zugriff an
unlock (name) // Freigeben des Locks

barrier (name, number) // Globale Synchronisation für number-viele Prozesse

wait_for_end(number) // warte, bis number-Prozess endet

while (!flag); or wait(flag) // Warte auf gesetztes Flag, entweder wiederhole Abfrage (spin) oder blockiere

flag = 1; or Signal(flag) // Setzt flag; weckt den Prozess auf, der wiederholt flag abfragt.

Message Passing

create(procedure) // erzeuge Prozess, der bei procedure startet

send (src_address, size, dest, tag) // Sende size Bytes von Adresse src_address an Prozess dest mit tag als Identifier

receive (buffer_address, size, src, tag) // empfange eine Nachricht mit der Kennung tag vom src-Prozess und lege size Bytes im Puffer bei buffer_address ab.

barrier (name, number) // Globale Synchronisation von number Prozessen

Gültigkeitsproblem bei Rechnern mit mehreren Prozessoren, die unabhängig voneinander auf denselben Speicher schreibend und lesend zugreifen können.
Mehrere lokale Kopien (in Caches) müssen in Einklang gebracht werden.
Eine Cache-Speicherverwaltung heißt Cache-kohärent, wenn ein Lesezugriff immer den Wert des zeitlich zuletzt durchgeführten Schreibzugriff auf das gelesene Wort liefert.
Dieses Problem tritt auch bei anderer HW auf, die Schreibrechte auf dem geteilten Speicher hat. Siehe DMA-Controller
Problem 1: Write through:
DMA schreibt in RAM. CPU hat veraltetes Datum in Cache, das als gültig eingetragen ist.
Problem 2: Copy-Back:
CPU ändert Wert einer globalen Variable, aktualisiert dies aber nur im Cache. DMA liest veraltetes Datum aus dem RAM.
Abhilfe für Problem 1: non-cachable-data-flag
markiere alle Variablen im RAM-Bereich, den sich CPU und DMA teilen als nicht-cachbar
Abhilfe für Problem 2: Cache-Flush
DMA signalisiert der CPU, dass sie alle mit „dirty" (also veränderten) Variablen in den RAM speichern soll.
Danach werden alle Cache Einträge ungültig gemacht.
Alternative: Cache-Clear:
Setze alle Einträge im Cache auf ungültig.

All diese Probleme treten auch auf, wenn mehrere Prozessoren sich einen Adressbereich teilen

Kohärenz	Konsistenz
Welcher Wert geliefert wird	Wann ein geschriebener Wert bei einem Lesezugriff geliefert wird

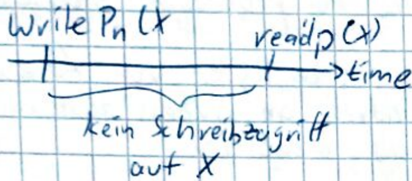
Ein Speichersystem ist kohärent bei Einhaltung der Programmordnung, kohärenter Speichersicht und Schreibserialisierung

Einhaltung der Programmordnung ≙ Lesezugriff nach Schreibzugriff liefert geschriebenen Wert

Kohärente Speichersicht ≙ genügend Zeit zwischen Schreib- und Lesezugriff && keine intervenierende Schreibzugriffe
=> Lesezugriff nach Schreibzugriff liefert ~~gelesen~~ geschriebenen Wert

Schreibserialisierung ≙ Schreibzugriffe auf dieselbe Speicherzelle werden von externem Sicht-Standpunkt aus serialisiert.

Kohärente Sicht des Speichers ≙ Einhaltung der Programmordnung, nur dass auch ein anderer Prozess P_n dazu erlaubt ist, zu schreiben und dennoch von P der geschriebene Wert gelesen wird.



Wann wird ein geschriebener Wert sichtbar?
- Man kann nicht fordern, dass ein Lesezugriff sofort einen geschriebenen Wert liest

Konsistenzmodell: Strategie, wann ein Prozessor die Schreiboperationen eines anderen Prozessors sieht.

Ein paralles Programm, das auf einem Multiprozessor läuft, kann mehrere Kopien eines Datums in mehreren Caches haben.

Migration: Daten können zu einem Cache migrieren und dort in einer Transparenten Weise verändert werden
Reduziert Latenz durch Wegfallen des Zugriffs auf ein entferntes Datum
Reduziert auch die erforderliche Bandbreite des gemeinsamen Speichers

Replikation: Mehrfache Migration in verschiedene Caches
Benötigt Möglichkeit für Blockierung beim Zugriff auf das gemeinsame Datum

Möglichkeiten um die Kohärenzanforderungen zu erfüllen
- Write-invalidate-Protokoll:
  - Sicherstellen, dass P_n exklusiven Zugriff hat
  - Dann schreiben
  - Dann allen anderen Prozessoren mitteilen, dass ihre Kopie ungültig ist.
- Write-Update-Protokoll:
  - Wie Write-invalidate, nur hier wird gefordert, das alle anderen Kopien ebenfalls verändert werden (spätestens beim Zugriff darauf)

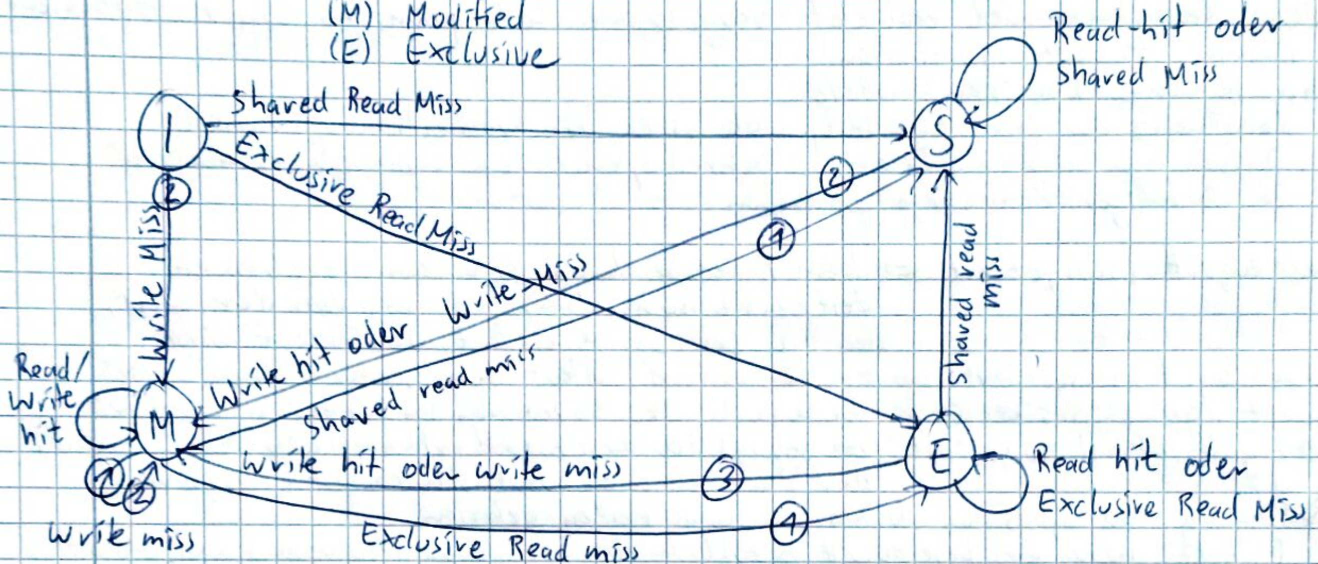
WI vs. WU:
- WI benötigt nur eine Invalidierung zu senden. Danach kann theoretisch beliebig oft auf denselben Datum geschrieben werden, ohne dass Overhead entsteht.
- Bei WU ist immer ein broadcast notwendig

HW-Umsetzung der Kohärenzprotokolle
Tabellen basiert (directory-based) halten den Zustand gemeinsamer Daten in Tabellen fest
Alternative: Bus-Schnüffeln

MESI - Kohärenzprotokoll

Jeder Cache verfügt über eine Schnüffel-Logik und Steuersignale
- Invalidate: Zum Invalidieren von Einträgen in anderen Caches
- Shared: Anzeige, ob ein zu ladender Block bereits in einem anderen Cache liegt
- Retry: Aufforderung an einen anderen Prozessor, den Ladevorgang abzubrechen und zu wiederholen, sobald das aktuelle Datum (fertig) geschrieben wurde.

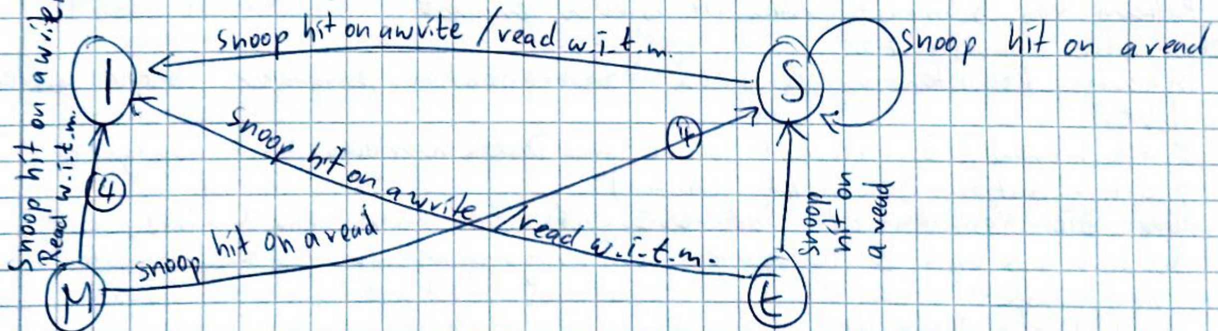
Jede Cache Zeile wird um 2 Bits erweitert, sie zeigen den Status an:
- (I) Invalid
- (S) Shared
- (M) Modified
- (E) Exclusive



① Cache-Zeile wird in den Hauptspeicher zurückkopiert (Line Flush)

② Cache-Zeile wird in den anderen Caches invalidiert (Line Clear)

③ wie 2, gilt jedoch nur für write miss



④ Retry-Signal wird aktiviert und danach wird die Cache Zeile in den RAM kopiert.

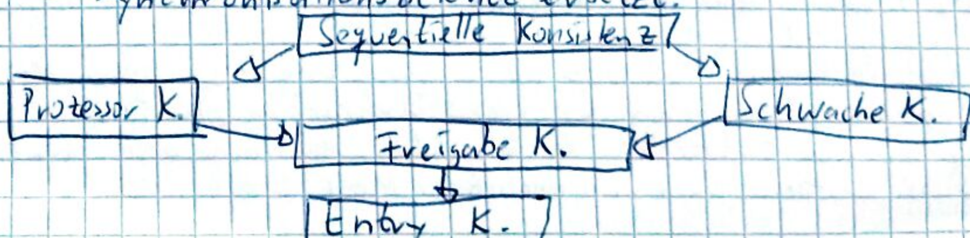
Dies ist alles nur möglich, wenn ein Bus den gemeinsamen Speicher, die Snoop-Controller und die Caches verbindet.

Bei Systemen ohne Bus werden Verzeichnisbasierte Kohärenzprotokolle verwendet.
Realisierbar in HW oder SW, Verwaltung de- oder zentral
Tabellen protokollieren für jeden Blockrahmen, ob dieser in den lokalen oder einem der entfernten Cachespeicher als Cache block übertragen worden ist.
Diese Zustände dann sind ähnlich zu MESI

Sequenzielle Konsistenz ≙ Multiprozessorsystem, dessen beliebige Berechnung stets das Ergebnis erzeugt, das ein Einkern-System bei sequenzieller Abarbeitung erzeugt hätte.
Alle Lese- und Schreibzugriffe werden in einer beliebigen, aber mit der Programmordnung konformen Reihenfolge am Speicher wirksam.
Entspricht einer überlappenden sequenziellen Ausführung anstelle einer parallelen Ausführung
Schreibzugriffe müssen atomar sein. D.h. der jeweilige Wert muss überall gleichzeitig wirksam sein.

Dieses Konsistenzmodell verbietet vorgezogene Ladeoperationen und nichtblockierende Caches. ⇒ Leistungseinbußen!
Abgeschwächte Konsistenzmodelle:
Konsistenz nur zum Lesezeitpunkt einer Synchronisationsoperation
Dazwischen dürfen Lese- und Schreiboperationen mehrerer Prozessoren in zufälliger Reihenfolge passieren

Lesezugriff „ausgeführt" ⟺ kein Prozessor kann zu einem gegebenen Zeitpunkt mehr dafür sorgen, dass der Wert, den der Lesezugriff liefert, verändert wird.
Schreibzugriff „ausgeführt" ⟺ ein Lesezugriff liefert den geschriebenen Wert
Zugriff „ausgeführt" ⟺ er ist bezüglich aller Prozessoren im System ausgeführt
Lesezugriff „global ausgeführt" ⟺ sowohl der Lesezugriff, als auch der Schreibzugriff, der den gelesenen Wert schrieb, sind ausgeführt
Bedingung der globalen Ausführung wird fallen gelassen
Es gilt: Bevor ein Lesezugriff bezüglich eines anderen Prozessors ausgeführt werden darf, müssen alle anderen vorhergehenden Lesezugriffe ausgeführt worden sein UND
Bevor ein Schreibzugriff irgendeines anderen Prozessors ausgeführt werden darf, müssen alle ~~anderen~~ vorhergehenden Zugriffe ausgeführt worden sein.
Konsequenz:
Prozessorkonsistenz führt nicht mehr zu sequenzieller Konsistenz
Puffern von Schreibzugriffen ist wieder erlaubt
Schwache Konsistenz
Bisherige Konsistenzmodelle lassen Synchronisation paralleler Threads außer Acht.
Konkurrierende Zugriffe auf gemeinsame Daten werden durch geeignete Synchronisation geschützt (Mutex)
Idee: Die Konsistenz des Speicherzugriffs wird nicht mehr zu jeder Zeit, sondern nur nach zu bestimmten, vom Programmierer gesetzten Synchronisationszeitpunkten.
⇒ Kritische Bereiche
Innerhalb dieser Bereich werden Inkonsistenzen zugelassen und konkurrierende Zugriffe werden unterbunden
Synchronisationszeitpunkte sind Ein- und Austritt des kritischen Bereiches, diese müssen sequenziell konsistent sein.
Hürdeneigenschaft der Synchronisationsbefehle muss die HW unterstützen
Ordnung der Speicherbefehle wird durch die sehr viel losere
" " Synchronisationsbefehle ersetzt.



Synchronisationsmechanismen
- Bereitstellung über Software befehle (=> Abbildung auf HW-Befehle)
- Geringe Blockierung bei kleineren Systemen
  - test_and_set, atomare Abarbeitung von read/modify/write ununterbrechbar, nur 1 Thread hat Zugriff gleichzeitig
  - Load-locked lädt einen Lock-Wert aus dem Zielregister und kann mit Hilfe von Store Conditional erkennen, ob ein weiterer Thread auf diesem Speicherwert arbeitet. Dann wird die Aktualisierung des Speichers abgebrochen.
- Große Blockierung
  - Spin-Locks
    - Sperrmechanismus mit ständigem Versuch der Akquise durch Prozessor
    - Verwendung bei kurzer Belegungsdauer und kurzer Set-Dauer
    - => Busy Waiting
    - Bei Mehrfachem Zugriff müssen alle "Verlierer" ihren Cache aktualisieren, da der "Gewinner" eine Invalidierung nach dem Schreiben sendet
- Barrier Synchronisation
  - " mehreren Prozesse, Anzahl Parametrisiert

Nachrichtenorientiertes Programmiermodell (Message Passing) bei verteiltem Speicher
- Synchrones Message passing
  - Sender & Empfänger blockieren, bis Nachricht ausgetauscht wurde
  - => Dead-lock Gefahr (RECV bleibt aus, wenn beide senden)
- Synchronisation und Kommunikation in einer Primitive
  - Blockierendes asynchrones SEND
    - Prozess erhält Kontrolle wieder, sobald die zu sendenden Daten im Sendepuffer liegen und nicht mehr verändert werden
  - Blockierendes asynchrones RECV
    - Kontrolle wird erst an den Empfangenden Prozess zurückgegeben, wenn die zu empfangenen Daten im Empfangspuffer liegen und von dort in den lokalen Bereich kopiert wurden
  - Nichtblockierendes asynchrones SEND/RECV
    - Kontrolle geht sofort zurück and den sendenden/empfangenden Prozess.
    - Datentransfer läuft im Hintergrund
  - Probe-Funktionen
    - prüfen, ob ein Datentransfer vollzogen wurde. Sender: ob es empfangen wurde, Empfänger: ob es da ist.
  - Sender-Initiiert
    - Request-to-send -> Receiver ready -> Message transfer (Sender wartet auf RECV)
  - Empfänger-initiiert
    - Request-to-receive -> Message transfer (Empfänger wartet auf SEND)
- HW-Unterstützung:
  - Verbindungsnetzwerk bietet rudimentäre Übertragungsprimitive an und entlastet den Prozessor von der Koordination
  - DMA übernimmt Datenschieben oder dedizierter Kommunikationsprozessor

Parallelisierungsstrategien
- Functional Decomposition
  - Software Pipeline
  - Master/Slave
- Domain Decomposition — Bei Datenabhängigkeiten Bsp. Aliasing (Bilder)
  - Disjunkte Gebiete / Überlappende Gebiete der Daten
  - Divide & Conquer

OpenMP

Compilerbasierter Ansatz für Parallelrechnung auf gemeinsamen (verteilten) Speicher
#pragm omp

Fork bei #pragma omp parallel
Implizite Barriere am Ende des Blocks ('}')
Nur der master-Thread führt alles danach weiter aus (bis zum nächsten Fork, if any)

#pragma omp parallel sections
jeder Thread im Team kriegt eine Anweisung aus dem Sektionsblock, wenn er idle ist.
#pragma omp section

#pragma omp parallel for /do [parameters]
jeder Thread im Team führt Schleifeniterationen aus. Die Daten werden hierbei aufgeteilt

[parameters] kann sein: private (x)
jeder Thread im Team hat seine eigene Kopie von x
Alle anderen Variablen sind implizit shared
→ firstprivate (x)
Wert wird anfangs vom gemeinsamen in den privaten Bereich kopiert
→ lastprivate (x)
Wert aus der letzten Iteration wird am Ende vom privaten in den gemeinsamen Bereich kopiert

Aufteilungsstrategien schedule (strategy (size))
static (size)
jede Iteration bekommt statisch für sie vorherbestimmte Datenpakete
nützlich wenn jede Iteration gleich lange dauert
dynamic (size)
Daten werden zur Laufzeit bestimmt, jeder fertige Thread erhält size viele Neue Daten
nützlich wenn die Iterationen verschieden lange dauern

Reduktionen
#pragma omp parallel for reduction (op : var1, var2)
für parallele Reduktionen wie z.B. Summe über ein Array von Zahlen

#pragma omp barrier
Warten, bis alle Threads diesen Befehl erreicht haben

#pragma omp atomic
eine Anweisung, die nur von einem Thread gleichzeitig ausgeführt werden darf ⇒ sequentiell

#pragma omp critical (name)
erzeugt eine kritische Sektion, die wie atomic behandelt wird. Kann an mehreren Stellen im Code logisch über den gleichen Bezeichner vereinigt werden

#pragma omp master
folgende Region wird nur vom Master-Thread ausgeführt

#pragma omp single wird nur einmal (von beliebigem Thread) ausgeführt

#pragma omp task
zur Funktionszerlegung, drückt einem freien Thread einen Task in die Hand

Seit OpenMP 4.0
Vektorisierung
Unterstützung von Beschleunigern

Message Passing Interface (MPI)
Bibliotheksbasierter Ansatz für Parallelisierung ohne gemeinsam (verteilten) Speicher (NORMA)

Includes → Initialisierung → Programm (OpenMP) & MPI calls → Finalize

MPI_Init / MPI_Finalize
MPI_Comm_Size / MPI_Comm_Rank
MPI_Send / MPI_Recv

Kommunikationsgruppen
Global: MPI_COMM_WORLD

Rang gesetzt für jeden Prozess, der MPI_Init aufruft
Wichtig für gerichtete Kommunikation und Steuerung der Programmausführung

Prozess A → Puffer von A → MPI → Puffer von B → Prozess B
Verschiedene Send-Methoden bsend / ssend / isend / send
Grundsätzlich: Blockierend oder nicht-blockierend
Unterschiedlichen Gründe für das Blockieren wählbar:
Warten auf Recv oder Warten bis Puffer wieder nutzbar

int MPI_Send(buff, count, dtype, tag, dest, comm)

buff: void* Adresse des Sendebuffers
count: int # zu sendende Objekte des Typs dtype
dtype: MPI_Datatype zu sendender Datentyp
dest: int Prozess ID des Empfängers
tag: int ID der Nachricht
comm: MPI_Comm Kommunikator

int MPI_Recv(buff, count, dtype, src, tag, comm, status)

buff: void* Adresse des Empfangspuffers
count: int # der zu empfangenden Objekte des Typs dtype
dtype: MPI_Datatype Datentyp der zu empfangenden Daten
src: int ID des Senderprozesses
tag: int ID der Nachricht
comm: MPI_Comm Kommunikator
status: MPI_Status Statusinformation

MPI_ANY_TAG oder MPI_ANY_SOURC als Wildcard

MPI_SendRecv (void* sendbuff, int sendcount, MPI-Datatype sendtype,
int dest, int sendtag, void* recvbuf, int recvcount,
int src, int recvtag, MPI_Comm comm, MPI_Status status)
entspricht MPI_Send und MPI_recv in parallelen Threads

MPI_SendRecv-Replace
wie MPI_SendRecv, allerdings wird der Inhalt des Sendepuffers durch den Inhalt des Empfangenen ersetzt.

Kollektive Operationen
Müssen von allen Prozessen in der Gruppe ausgeführt werden
Blockierende Operationen
Synchronisation: Barrier
Kommunikation: Broadcast / scatter / gather
Reduktion: reduction

int MPI_Barrier (~~comm~~ MPI_Comm comm)
Alle Prozesse in comm werden synchronisiert

int MPI_Bcast (void* buff, int count, MPI_Datatype dtype, int root, MPI_Comm comm)

root: Sender
alle Prozesse in comm erhalten den Inhalt von buff

int MPI_gather (void* sendbuff, int sendcount, MPI-Datatype send dtype,
void* recvbuff, int recvcount, MPI_Datatype recvdtype,
int root, MPI_Comm Comm, MPI_Status *status)

root empfängt von allen Prozessen in comm die Daten im Sendepuffer
Die Daten im Empfangspuffer werden nach Sender ID sortiert

int MPI_Scatter (siehe oben ↑)
root sendet jedem Prozess einen Teil seines Sendepuffers
Prozess k erhält sendcount Elemente im recvbuff, beginnend mit
Element sendbuff [k * sendcount]

Reduktion
Daten von unterschiedlichen Prozessoren werden mit Operator verknüpft
Das Ergebnis kann nur der Root verfügbar sein
oder Allen
oder Allen, aber das Reduktionsergebnis hängt von der Prozess ID ab.

int MPI_Reduce (void* sendbuff, void* recvbuff, int count, MPI_Datatype dtype,
MPI_Op op, int root, MPI_Comm comm)

Kombiniert die Element in sendbuff und liefert das Ergebnis zum Root-Prozess

=> Open MP bei gemeinsamem Speicher
Kommunikation implizit über geteilte Variablen
Compiler basierter Ansatz

MPI bei verteiltem Speicher
Kommunikation explizit über Nachrichten
Bibliotheksbasierter Ansatz

OpenCL

Zur Kommunikation über heterogene Rechensysteme hinweg um Beschleuniger ansprechen und nutzen zu können

Beispiel: GPU, FPGA, Xeon Phi, Intel MIC, DSPs, ASIP

Dedizierte Beschleuniger:
+ Schneller eigener Speicher
+ Beliebige Kombination möglich
− Teuerer/lange Datentransfer über PCIe o.Ä.
− Datenkonsistenz meist schwer und Aufgabe des Programmierers

Integrierte (on-die) Beschleuniger
+ Gemeinsamer Speicher
+ kein Datentransfer
− Fixe Kombination
− Die-Fläche geteilt

FPGA
+ quasi jede digitale Schaltung möglich
+ hohe Flexibilität
− i.d.R. aufwändige Konfiguration

GPU (oder sogar GP-GPU)
+ mittlerweile General Purpose
+ Große Bekanntheit durch CUDA
+ massiv-parallele Ausführung möglich
+ Threadwechsel in einem Takt → Maskierung von Speicherlatenzen
− ungeeignet bei Code mit Verzweigungen
− hoher Grad an Parallelität in Anwendung notwendig

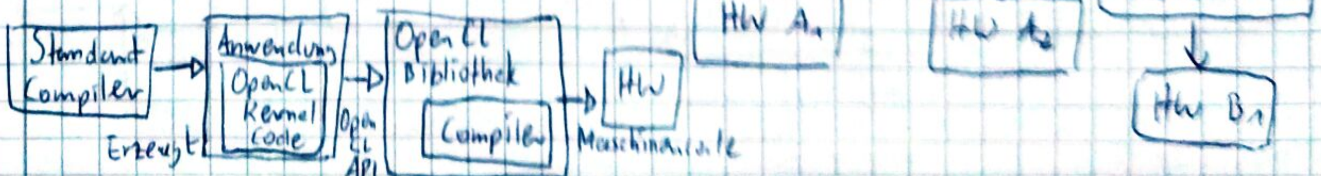
Xeon Phi / Intel MIC
+ > 60 Cores (Pentium P54c)
+ 512 bit SIMD
+ bidirectional ring interconnect
+ Gemeinsamer Speicher
+ x86
− umfangreiche Optimierung

OpenCL als Programmiermodell für all diese Architekturen

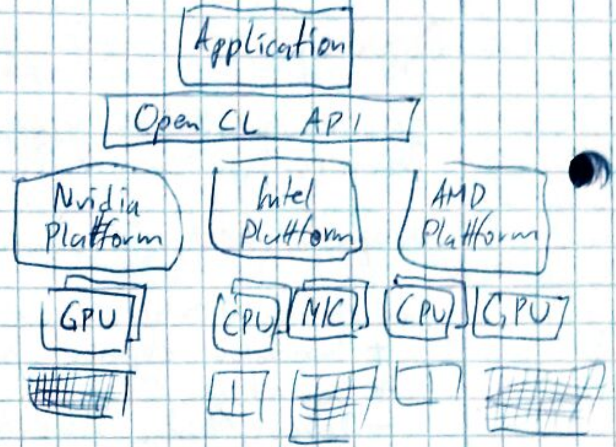
Grundsätzliche Struktur
Bibliotheksbasierter Ansatz
Eine Bibliothek für ausgewählte HW
Mehrere gleichzeitig nutzbar mit
ICD (Installable Client Driver)

Anwendung
OpenCL API
OpenCL Bibliothek A
ICD
OpenCL Bibliothek B
HW A_1
HW A_2
HW B_1

Kein spezieller Compiler notwendig
Compiler für Kernel ist in Bibliothek enthalten
Kernel wird als String mitgeführt und dann
zur Laufzeit für HW übersetzt



Organisation von Recheneinheiten



- Pro Hersteller eine Plattform
- Verschiedene Geräte pro " möglich
- " # Recheneinheiten pro Gerät
- Viele API Aufrufe notwendig; Wrapper oft genutzt
- Mit Plattform-ID können Details wie Hersteller, Name oder unterstützte OpenCL extensions abgefragt werden
- Analog mit Geräte-ID

OpenCL Kontext

- Kontext benötigt zur Interaktion mit Geräten und deren Speicher
- Gesendete Befehle werden in einer command queue gepuffert
  - Diese kann in-order oder out-of-order abgearbeitet werden
- Speichertransfer ebenfalls gepuffert (clCreateBuffer)
  - Angabe von Größe und write/read-only
  - clEnqueue[Read, Write]Buffer

Erzeugung von ausführbarem Kernel-Code

- CL_program ⊃ CL_kernel
- Cl_Kernel ∈ {binary, quellcode}
- binärcode ∈ {fertiger Maschinencode, Zwischeninterpretation}
- Ablauf der Erzeugung:

```
clCreateProgramWith[Source, Binary];
clBuildProgram();
clCreateKernel();     //extrahiere benötigten Kernel aus Programm
```

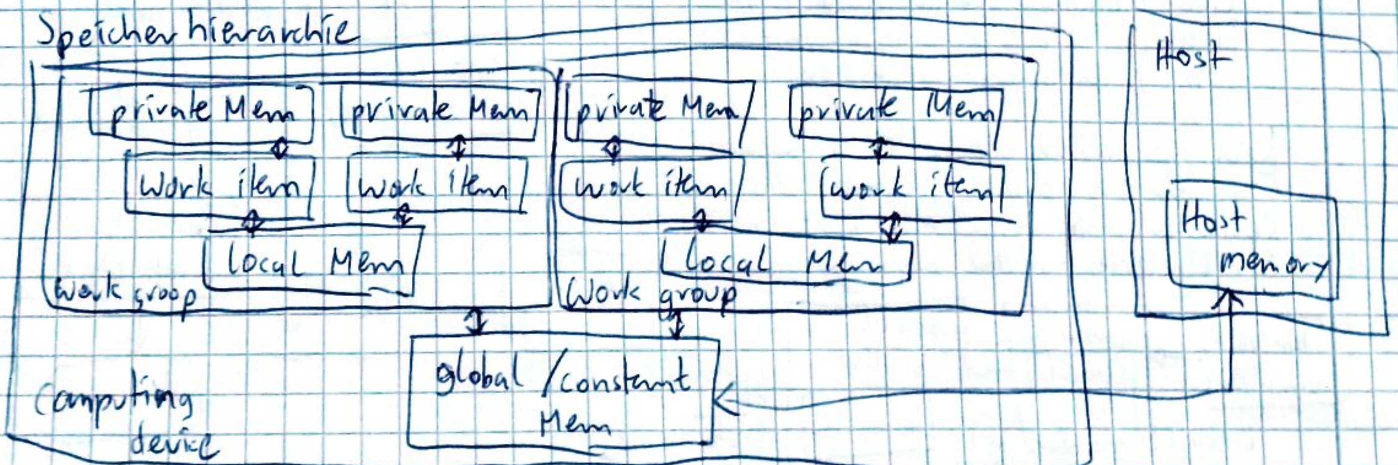
Parameter für eine Kernel-Funktion

```
cl_int clSetKernelArg(cl_kernel kernel, cl_uint arg_index,
                      size_t arg_size, const void* arg_value)
```

- kernel: valides Kernelobjekt
- arg_index: Position des Parameters
- arg_size: Größe des Parameters im Speicher
- arg_value: Adresse des " " "

clEnqueueNDRangeKernel trägt Kernelausführung in die command queue ein
- Parameter: command queue, kernel, #Dimensionen der Arbeitsgruppen, # work groups, # work items

Speicherhierarchie



OpenCL vs CUDA

+ Herstellerunabhängig
+ 1 Kernel nutzbar für verschiedene HW
– Abstraktion erschwert HW-spezifische Optimierung
– Aufwendige Einbindung in Quellcode

OpenCL auf GPU:
 1 Thread pro Work item
 Kerne in Gruppen organisiert
 Schnelle Threadwechsel bei langsamen Speicherzugriff

OpenCL auf CPU / Xeon Phi
 Ein Thread berechnet mehrere work items / groups
 Wichtig: Cache-Nutzung
 Ähnlich zu OpenMP

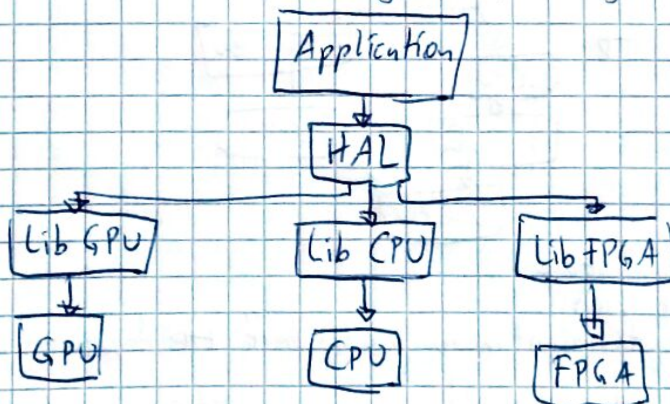
OpenCL auf FPGA
 Operationen werden in Verilog übersetzt
 Generierung möglichst vieler Pipelines mit SIMD Eigenschaft

Open ACC (Open Accelerators

von den Machern von OpenMP
für heterogene parallele Rechensysteme
Mehraufwand für Compiler

HAL (Hardware Abstraction Layer)

Beste Wahl der Recheneinheiten abhängig von Problem & Eingabegröße
=> statische Wahl nur in wenigen Fällen sinnvoll
Idee: dynamisches hin- und Herschalten zwischen Recheneinheiten je nach Problem und Eingabegröße
Aufwand für Entwickler: Datentransfers + Konsistenz erhalten
 Kann verschiedene Optimierungsziele verfolgen (Energie, Zeit, ...)
HAL abstrahiert HW von Implementierung
 Programmierer muss sich nicht mehr darum kümmern welche HW vorliegt
 Implementierungen können nachträglich hinzugefügt werden und ist getrennt von der Anwendungsentwicklung möglich
 => Portabilität durch Auslagerung der Implementierungen in Bibliotheken
 Was vorhanden ist, wird geladen, falls gebraucht



Expliziter Funktionsaufruf
 über HALadapt API
 direkt an das Laufzeitsystem weitergegeben, Funktionsparameter werden in einem internen Stack zwischengespeichert, somit sind zusätzliche Funktionen ausführbar.

Impliziter Funktionsaufruf
HALadapt muss den Funktionspointer überschreiben mit Hilfe von Registrierungsaufruf
Funktionspointe zeigt dann auf spezielle HALadapt Funktion
Funktionsparameter müssen gesichert werden, um zusätzliche Funktionen aufrufen zu können

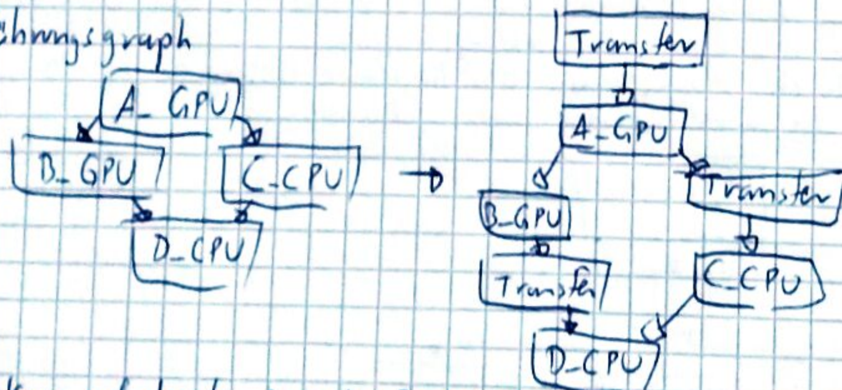
Call Stack
HAL organisiert Funktionsaufrufe in einem Call Stack, der abgearbeitet wird

Dynamische Laufzeitvorhersage
Kernels und Datentransfers werden zur Laufzeit evaluiert und die Ergebnisse in einer Datenbank gespeichert.
Dient zur Laufzeitvorhersage von zukünftigen Berechnungen und Transfers in Abhängigkeit der Problemgröße und -Art.
Datenbank besteht aus History-Structs
Messungen stehen in Vector-Structs
Sollte für ein Problem und dessen Größe noch keine Einträge geben, wird eine Messung angefordert
Möglichkeit zur Interpolation, falls zwei andere Einträge nah genug an der Abfrage liegen

Task-Modell
Heterogenität verlangt Trade-Off: Unterschiedliche Ausführungszeiten vs. Datentransfer
Mehrere aktive Kernel pro Task
Tasks können zu Graphen zusammengefasst werden
HAL adapt bietet automatische Abbildungsentscheidung
Verschiedene Optimierungsziele können verfolgt werden
Zusätzliche Heuristiken können als Plug-In hinzugefügt werden
z.B HEFT (heterogeneous earliest finish time) (List-Schedule-Algos

Automatische Datentransfers durch Versionskontrolle
Spezielle Datenstrukturen
Welche Daten befinden sich wo?
Ermöglicht Bestimmung des Systemzustands
Sind die Eingabedaten für zukünftige Ausführungen bekannt, können nötige Datentransfers im voraus berechnet werden

Ausführungsgraph



Reaktion auf konkurrierende Prozesse
Rechendauer verändert sich
Bisherige Recheneinheit evtl. nicht mehr vorteilhaft ↦ Anpassung der Klassifikation

Einbindung HAL per (OpenCL → HAL)-Wrapper, dieser wählt beste Bibliothek und bestes Gerät aus. Transparent für Anwendung