Rechnerstrukturen am Karlsruher Institut für Technologie

Maximilian Heß, Carolin Beer

August 2019

Inhaltsverzeichnis

Zusammenfassung der Vorlesung „Rechnerstrukturen“ aus dem Sommersemester 2019.[1]

[1] https://capp.itec.kit.edu/teaching/rs/

1 Grundlagen

1.1 Einführung

Entwurf einer Rechneranlage: Ingenieurmäßige Aufgabe der Kompromissfindung zwischen Zielsetzung, Randbedingungen, Gestaltungsgrundsätzen und Anforderungen.

1.2 Entwurf von Rechneranlagen - Entwurfsfragen

- **Heutige Kennzahlen**
  - Höchstleistungsrechner: Exa-FLOPs (10^{18})
  - Intel Skylake-Prozessoren: 8 Mrd. Transistoren
  - Strukturbreite: 10nm
- **Prozessortypen**
  - Multicore/Manycore
  - Anwendungsspezifisch, bspw. Google TPUs
- **Zielsetzung**
  - Einsatzgebiet
    - **Desktop Computing**
      - PCs bis Workstations ($1000 - $10.000)
      - Günstiges Preis-/Leistungsverhältnis
      - Ausgewogene Rechenleistung für ein breites Spektrum von (interaktiven) Anwendungen
    - **Server**
      - Rechen- und datenintensive Anwendungen
      - Hohe Anforderungen an die Verfügbarkeit und Zuverlässigkeit
      - Skalierbarkeit
      - Große Dateisysteme und Ein-/Ausgabesysteme
    - **Eingebettete Systeme**
      - Mikroprozessorsysteme, eingebettet in Geräte und daher nicht unbedingt sichtbar
      - Sind auf spezielle Aufgaben zugeschnitten (hohe Leistungsfähigkeit, Spezialprozessoren)
      - Breites Preis-/Leistungsspektrum
      - Echtzeitanforderungen
      - Abwägung der Anforderungen an Rechenleistung, Speicherbedarf, Kosten, Energieverbrauch, etc.
  - Anwendungsbereich
    - Technisch-wissenschaftlicher Bereich: Hohe Anforderungen an die Rechenleistung, insbesondere Gleitkommaverarbeitung
    - Kommerzieller Bereich: Datenbanken, WEB, Suchmaschinen, Optimierung von Geschäftsprozessen, etc.
    - Eingebettete Systeme: Verarbeitung digitaler Medien, Automatisierung, Telekommunikation, etc.
  - Rechenleistung
    - Ermittlung über Benchmarks
    - Maßzahlen für die Operationsleistung: $MIPS$ oder $MFLOPS$
    - $MFLOPS = \frac{Anzahl\ ausgefuehrter\ Gleitkommainstruktionen}{10^6 \cdot Ausfuehrungszeit}$
  - Verfügbarkeit
  - Zuverlässigkeit
    - Bei Ausfällen von Komponenten muss ein betriebsfähiger Kern bereit sein
    - Verwendung redundanter Komponenten
    - Bewertung der Ausfallwahrscheinlichkeit mittels stochastischer Verfahren
    - Definition Verfügbarkeit: Wahrscheinlichkeit, ein System zu einem beliebigen Zeitpunkt fehlerfrei anzutreffen
- **Randbedingungen**
  - Technologische Entwicklung: Mikrominiaturisierung setzt sich fort, beispielsweise Verkleinerung der Strukturbreiten sowie Erhöhung der Integrationsdichte (Anzahl der Transistoren verdoppelt sich alle 18 Monate)
  - Größe
  - Geld

  - Energieverbrauch, Leistungsaufnahme
    * Mobile Geräte
      · Verfügbare Energiemenge durch Batterien und Akkumulatoren ist begrenzt → möglichst lange mit der vorhandenen Energie auskommen
      · Vermeiden von Überhitzungen
    * Green IT: Niedriger Energieverbrauch, ökologische Produktion, einfaches Recycling
  - Umwelt

- Gestaltungsgrundsätze: Modularität, Sparsamkeit, Fehlertoleranz, etc.
- Anforderungen: Kompatibilität, Betriebssystemanforderungen, Standards, etc.

1.2.1 Trends in der Rechnerarchitektur: Herausforderungen

Weltweite Forschungsaktivitäten bzgl. ExaScale-Rechner

- Verlustleistung: Überträgt man heutige (Stand 2010) Höchstleistungsrechner in den Exascale-Bereich, hätte man eine Verlustleistung von etwa 40 GW (diese kann allerdings höchstens 20-40 MW betragen)
- Hauptspeicher (DRAM), permanenter Speicher: Kapazität und Zugriffsgeschwindigkeit muss mit der Rechengeschwindigkeit mithalten
- Zuverlässigkeit und Verfügbarkeit
- Parallelität und Lokalität

1.3 Effizienter Entwurf - Grundlagen

- Mobile Geräte: Verfügbare Energiemenge durch Batterien und Akkus begrenzt → vorhandene Energie möglichst lange ausnutzen sowie Vermeidung von Überhitzung
- HPC: Hohe Temperaturen begrenzen die Verarbeitungsgeschwindigkeit und die beeinflussen die Zuverlässigkeit
- **CMOS-Schaltung**
  - MOSFET: Je nach Spannung am Gate und dem daraus resultierenden Feld im Kanal können Ladungsträger den Kanal passieren oder nicht. Extrem niedrige Stromaufnahme im Ruhezustand
  - Leistungsaufnahme: $P_{total} = P_{switching} + P_{shortcircuit} + P_{static} + P_{leakage}$
    * Statischer Leistungsverbrauch: P_{static} sowie Leistungsverbauch ($P_{leakage}$ bei Kriechströmen
    * Verbrauch bei Zustandsänderung: $P_{switching} = C_{eff} * V_{dd}^2 * f$ sowie Verbrauch während des Übergangs am Ausgang, wenn sich die Eingänge ändern $P_{shortciruit} = I_{mean} * V_{dd}$
    * Höhere Frequenzen erfordern steilere Taktflaken → schnelleres Laden von C_{eff} → höhere Versorgungsspannung
    * $P \sim fundP \sim V_{dd}^2$ aber da $f \sim V_{dd}^2 \Rightarrow P \sim V_{dd}^3$ und $P \sim f^3$
    * Energieverbrauch pro Aufgabe ist unabhängig von Frequenz, bzw. sogar negativ korreliert wegen statischer Ströme
  - Schaltwahrscheinlichkeit
    * $\mathbb{P}_{Schalt} = 2 \cdot \mathbb{P}_{Ausgang(1)} \cdot (1 - \mathbb{P}_{Ausgang}(1))$
    * $\mathbb{P}_{Ausgang}(1) = \mathbb{P}_{Eingang1}(0) \cdot \mathbb{P}_{Eingang2}(1) + \mathbb{P}_{Eingang1}(1) \cdot \mathbb{P}_{Eingang2}(0)$

1.3.1 Kosten von Prozessoren

- **Ziel**
  - Leistungsaufnahme senken ohne Verarbeitungsgeschwindigkeit zu beeinträchtigen
  - Optimierung der Systemarchitektur
  - Spezialisierte Prozessorkerne/Multicore-CPUs, die Parallelverarbeitung erlauben, verwenden
- **Kosten** Proportional zu $\sqrt{A_{Wafer}}$, Chipfläche wird durch Entwickler beeinflusst
  - $C_{Die} = \dfrac{C_{Wafer}}{\frac{\texttt{\#Dies}}{Wafer} * Ausbeute}$
  - $\texttt{\#Dies} = \dfrac{\pi * r_{Wafer}^2}{A_{Die}} - \dfrac{2\pi * r_{Wafer}}{\sqrt{2 * A_{Die}}}$
  - $\texttt{Ausbeute} = \dfrac{\texttt{Ausbeute}_{Wafer}}{(1 + \texttt{Defekte})^N}$

1.4 Einführung in den Entwurf eingebetteter Systeme

1.4.1 Die Hardware-Beschreibungssprache VHDL

- Standardisierte Hardware-Beschreibungssprache: Die verschiedenen Schaltungsbeschreibungen des gesamten Entwurfsablaufs können dargestellt werden - von der algorithmischen Spezifikation bis hin zu realisierungsnahen Strukturen
- Eingesetzt zum ASIC- und FPGA-Entwurf
- Bei synchroner Zuweisung werden Änderungen abhängig von einem Takt und den Eingangssignalen geändert. Bei asynchroner Zuweisung unabhängig und unmittelbar (nach einer gewissen Schaltzeit)
- Automatischer Synthesewerkzeuge: Flexibilität, weniger fehleranfällig. Dafür Randbedingungen schwer einzuhalten und Ergebnisse oft schlechter als bei manuellem Entwurf
- Chip-Entwurf mit VHDL (Top-Down)
  - Grundlage des Entwurfs ist die Spezifikation der Schaltung: Gewünschtes Verhalten; Schnittstellen; Vorgaben bzwgl. Geschwindigkeit, Kosten, Fläche oder Leistungsverbrauch
  - Entwurfsschritte: Verhaltensspezifikation, High-Level-Synthese, RT-Beschreibung, Logik-Synthese, Gatterbeschreibung, Layout-Synthese, Geometriegebschreibung, Fertigung
  - Hauptbestandteile:
    - ENTITY: Schnittstellen
    - ARCHITECTURE: Verhalten
    - CONFIGURATION: Weist COMPONENT-Instanzen zu ENTITY und ARCHITECTURE zu.
  - Signale zur Kommunikation zwischen Instanzen untereinander oder mit Schnittstellen der äußeren Hülle (Entity)
  - Schnittstellendefinition eines MODULS

```
1  ENTITY blinklicht IS
2      PORT(
3          clk: IN Std_Logic;
4          reset: IN Std_Logic;
5          led: OUT Std_Logic
6      );
7  END blicklicht
```

  - Schema einer ARCHITECTURE

```
1  ARCHITECTURE Structure OF blinklicht IS
2      COMPONENT Counter
3      GENERIC(countMax : positive);
4      PORT(
5          clk: IN Std_Logic;
6          out: OUT Std_Logic
7      );
8      END COMPONENT
9      COMPONENT DCM
10     PORT(
11         clkin_in: IN Std_Logic;
12         rst_in: Std_Logic;
13         clkdv_out: OUT Std_Logic
14     );
15     END COMPONENT
16
17     Signal clk_int: Std_Logic
18
19     BEGIN
20         Inst:DCM: DCM
21         PORT MAP(
22             clkin_in => clk,
23             rst_in => reset,
24             clkdv_out => clk_int,
25         );
26         Inst_counter: counter
27         GENERIC MAP (countMax => 25000000)
28         PORT MAP(
29             clk => clkIn,
30             out => led
31         );
32     END Structure
```

1.5 Bewertung der Leistungsfähigkeit eines Rechners

- **Definitionen**
  - Wall-clock time, response time, elapsed time: Globale Latenzzeit für die Ausführung einer Aufgabe
  - CPU time (Vgl Unix time Kommando)
    - CPU time: Zeit, in der die CPU arbeitet
    - User CPU time: Zeit, inder die CPU ein Programm ausführt
    - System CPU time: Zeit, in der die CPU Betriebssystemaufgaben ausführt, die von einem Programm angefordert werden
- **Einfache Hardwaremaße**
  - Clock cycles per instruction: $CPI = \frac{\text{CPU time}}{\text{instruction count} \cdot \text{machine cycle time}} = \frac{T_{exe}}{IC \cdot T_C}$
  - Instructions per cycle: $IPC = \frac{1}{CPI}$
  - Millions of instructions per second: $MIPS = \frac{\text{Anzahl der ausgeführten Instruktionen}}{10^6 \text{ Ausführungszeit}}$, Floatingpointoperationen (MFLOPS) analog. Niedrigere MIPS-Werte resultieren bei gleicher Laufzeit in kompakterem Code, niedrigerer Schaltfrequenz und damit geringerem Energieverbrauch
  - Probleme: Angaben meist theoretische Maximalwerte bzw. abhängig vom konkreten Programm

1.5.1 Laufzeitmessungen bestehender Programme

- Bewertung der Leistungsfähigkeit mit Hilfe von einem Programm oder einer Programmsammlung und Messen der Ausführungszeit. Problem: Zugriff auf die Maschine notwendig und abhängig von der Güte des Compilers
- Kernels: Rechenintensive Teile reale Programme, vorwiegend numerische Algorithmen (beispielsweise LINPACK Softwarepaket zum Lösen linearer Gleichungen)
- Standardisierung zur Verbesserung der Vergleichbarkeit von Rechnern (inklusive OS und Compiler), z.B. SPEC CPU Benchmarks
  - SPEC: Non-profit Organisation, die Benchmarks zur Leistungsbewertung von Hardware und Software entwickelt. Mit dem Erwerb einer Lizenz verpflichten sich die jeweiligen Unternehmen, immer die kompletten Ergebnisse zu veröffentlichen. [1]
  - $SPEC_{ratio} = \frac{\texttt{Referenzzeit}_x}{\texttt{Laufzeit}_x \texttt{ auf Testsystem}}$ für Benchmark x. Der Endwert wird als geometrisches Mittel über alle Benchmarks berechnet, aufgrund der normalisierung unabhängig von der Referenzmaschine.

1.5.2 Messung während des Betriebs von Anlagen

- Monitore: Aufzeichnungselemente, welche die Verkehrsverhältnisse während des normalen Betriebs beobachten oder untersuchen
- Hardware-Monitor: Unabhängige physikalische Geräte, welche die Betriebsverhältnisse nicht beeinflussen
- Software-Monitor: Eingebaut ins OS, beeinträchtigen die normalen Betriebsverhältnisse
- Aufzeichnungstechniken: Kontinuierlich/spärlich, Gesamttracing, Realzeitauswertung, Post Processing
- Beispiel: Performance Counter
  - Misst verschiedene geschwindigkeitsrelevante Vorgänge, die ein Prozessor ausführt. Dabei beeinträchtigt er die Abarbeitung anderer Aufgaben im Prozessor nicht. Heutige Prozessoren erreichen bei den aktuellen Prozessortakten eine Auflösung im Mikro- bis Nanosekundenbereich[2]
  - Metriken zur Darstellung bestimmter Messgrößen. Beispielsweise Anzahl ausgeführter Befehle, Cache-Treffer oder Fehlzugriffe

1.5.3 Modelltheoretische Verfahren

- Unabhängig von Existenz eines Rechners
- Modellbildung: Annahmen über die Struktur und Betrieb eines Rechners sowie Analyse der relevanten Merkmale des Systems
- Ziel: Beziehungen zwischen Systemparametern aufdecken und Leistungsgrößen ermitteln
- Stellt für Analyse relevante Systemkomponenten und deren Datenverkehr dar
- Analytische Methoden
  - Versucht mathematische Beziehungen zwischen Leistungsgrößen herzuleiten. Oft minimaler Aufwand aber auch nur minimal aussagekräftig (beispielsweise Warteschlangenmodelle, Petrinetze, Diagnosegraphen, Netzwerkflussmodelle)

[1] https://de.wikipedia.org/wiki/Standard_Performance_Evaluation_Corporation
[2] https://de.wikipedia.org/wiki/Performance_Counter_(Mikroprozessor)

  - Gesetz von Little: Für ein stabiles Warteschlangensystem gilt

    $$k = \lambda t$$

    mit k als durchschnittliche Anzahl an Kunden im System, λ als durchschnittliche Ankunftsrate und t als durchschnittlicher Verweildauer.

- Simulationen
  - Modellierung zur evaluation neuer Ideen und der Exploration des Entwurfsraums. Mögliche Zielgrößen: Rechenleistung, Leistungsaufnahme, Zuverlässigkeit
  - Kompromiss zwischen Genauigkeit, kosten, Flexibilität und Komplexität
  - Taxonomie
    * User-Level Simulatoren: Simulation der Mikroarchitektur eines Prozessors ohne Berücksichtigung von Systemressourcen
    * Full-System Simulatoren modellieren ein vollständiges Computersystem, einschließlich CPU, IO, Disk und Netzwerk
    * Functional Simulatoren modellieren nur Funktionalitöt eines Prozessors, aber keine Berücksichtigung der Mikroarchitektur
    * Cycle-accurate Simulatoren erfassen parametrisierbare Details von Mikroarchitekturblöcken um Funktionalität und Zeitverhalten zu emulieren
  - Prozessimulatoren benötigen Liste von Befehlen die ausgeführt werden müssen
    * Trace-driven Simulation: Durchführung des Benchmarks auf anderem Prozessor/ Simulator, dabei die Befehle *tracen*. Diese Trace kann dann für einen Zyklengenauen Simulator verwendet werden. Jede Instruktion wird aud Basis des Mikroarchitekturmodells simuliert
    * Execution-driven Simulation: Simulator muss Zeitverhalten und Funktionalität genau reproduzieren. Daher aufwendig in der Entwicklung aber dafür genauer und flexibler

1.6 Zuverlässigkeit und Fehlertoleranz

- Taxonomie
  - Zuverlässigkeit: Wird während gewisser Zeitdauer bei zulässigen Betriebsbedingungen die spezifizierte Funktion erbracht?
  - Fehlertoleranz: Ist das System trotz begrenzter Anzahl fehlerhafter Subsysteme in der Lage die spezifizierte Funktion zu erbringen?
  - Sicherheit: Nichtvorhandensein einer Gefahr unter anzunehmenden Betriebsbedingungen. Verfügbarkeit, Überlebenswahrscheinlichkeit, ...
  - Vertraulichkeit: Datenschutz und Zugangssicherheit

- Fehler
  - Zustände
    * Fehlerfrei, sicher
    * Fehlerhaft, sicher/in Gefahr
    * Schaden/ Funktionsausfall
  - Ursachen
    * Entwurf (Spezifikation/Implementierung/Dokumentation)
    * Herstellung
    * Betrieb: Störung (externer Einfluss), Verschleiß, zufällig physikalischer Fehler, Bedienung, Wartung
  - Dauer: Temporär oder Permanent

- **Struktur-Funktions-Modell**
  - Gerichteter Graph, dessen Knoten die Komponenten und dessen Kanten die Funktionen eines Systems repräsentieren. Eine Kante $K_i \longrightarrow K_j$ bedeutet, dass K_i eine Funktion erbringt, die von K_j genutzt wird
  - eine Komponentenmenge ist ein System, wenn die erbrachten Funktionen durch eine äußere Spezifikation festgelegt werden
  - Binäres Fehlermodell gibt für jede Komponente und das Gesamtsystem an ob sie fehlerfrei sind: $Z : (S \cup \{S\}) \rightarrow \{\texttt{wahr, falsch}\}$. Die Systemfunktion gibt in Abhängigkeit der Komponenten die Funktion des Systems an.
  - Zuverlässigkeitsblockdiagramm: Zeigt mögliche „Verbindungspfade“ zwischen den Komponenten, von denen mindestens einer in Takt sein muss. Reihenschaltung von AND-Bausteinen, Parallelschaltung von OR-Bausteinen. Äquivalent zur Systemfunktion.
  - Negierter Systemfunktionsbaum (alle Blätter werden negiert, alle Verknüpfungen werden vertauscht ($AND \leftrightarrow OR$)) gibt an, wie sich Fehler des Systems auf Fehler der Komponenten zurückführen lassen.
  - **Fehlerbereiche** Komponenten, welche zeitgleich fehlerhaft sein können ohne zu Systemfehlern zu führen.
    * Einzelfehlerbereich: Menge von Komponenten, die exakt denselben Fehlerbereichen angehören
    * Perfektionskern: Das Komplement der Vereinigung aller Fehlerbereiche $\rightarrow$ Komponenten, welche nicht ausfallen dürfen

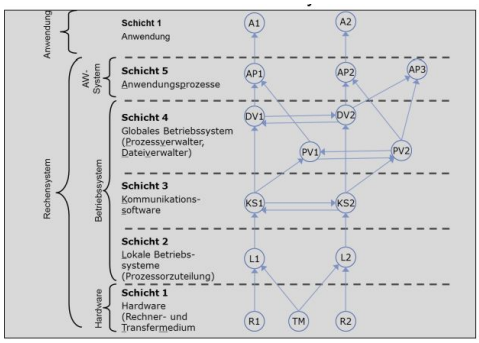

Abbildung 1.1: Schichtenmodell eines Zweirechensystems. Funktionszuordnungen sind nur von unteren auf höhere Schichten möglich.

- **Ausfallverhalten**
  - Teilausfall: Es fallen nicht alle Funktionen aus
  - Unterlassungsausfall (Fail-silent-System): Keine Ausgabe fehlerhafter Ergebnisse. Wenn ein Ergebnis ausgegeben wird, ist es korrekt
  - Anhalteausfall (Fail-stop-System): Keinerlei Ergebnisausgabe mehr
  - Haftausfall: Komponente gibt ständig denselben Ergebniswert aus
  - Binärstellenausfall: Ein Fehler verfälscht eine/mehrere Binärstellen
  - Unkritische Ausfälle (Fail-safe-System)
- **Ausfallverhalten**
  - Fehlereingrenzung und Vermeidung von Folgefehlern
  - Vertikal: Niedrigere Schichten prüfen Funktionsaufrufe vor Ausführung, Fehlerkorrekturcode in der Hardware, Plausibilitäts-/Konsistenzprüfungen
  - Horizontal: Isolierung fehlerhafter lokaler Knoten. Problem vor allem bei globalen Schichten.
- Um Anforderungen zu erfüllen sind Fehlertoleranz (Redundanz) und Fehlervermeidung wichtig.
- Zuverlässigkeitskenngrößen als Verteilungsfunktionen von Lebensdauer, Fehlerbehandlungsdauer und Sicherheit statistisch betrachten
  - $F_L(t) = \frac{N_f(t)}{N} = \frac{N - N_s(t)}{N} = 1 - R(t) = 1 - e^{-\lambda t}$ - Wahrscheinlichkeit, dass ein funktionierende System fehlerhaft wird mit Anzahl Komponenten N und f failed oder s survived. Fehlerwarscheinlichkeit exponentialverteilt mit Parameter λ
  - $R(t) = \frac{N_s(t)}{N}$ - Überlebenswahrscheinlichkeit
  - $N_f(t) = N - N_s(t)$
  - $z(t) = \frac{f_L(t)}{R(t)}$ als Ausfallrate
  - Ausfallverhalten („Badewannenkurve“): Frühphase (hohe Ausfallwahrscheinlichkeit durch Fertigungsfehler oder defekte Bauteile), Betriebsphase, Spätphase (hohe Ausfallwahrscheinlichkeit durch Verschleiß)
  - $V = \frac{E(L)}{E(L) + E(B)}$ als Verfügbarkeit mit Lebensdauer L und Behandlungsdauer B
  - $\phi(S) = \sum_{(K_1,\ldots,K_n) \in f^{-1}(\texttt{wahr})} \phi(\wedge_{i=1}^{n} K_i)$ Funktionswahrscheinlichkeit
  - Weitere Kenngrößen: Gefährdungwahrscheinlichkeit, Sicherheitswahrscheinlichkeit, mittlere Sicherheitsdauer
- **Redundanz**
  - Statische Redundanz: Vorhandensein redundanter Systeme während des gesamten Einsatzzeitraums (n-von-m-System)
  - Statische Redundanz/Zuverlässigkeit von n-von-m-Systemen (3-aus-5 vs. 2-aus-3): Das 3-aus-5-System zeigt in der Anfangszeit eine höhere Funktionswahrscheinlichkeit als das 2-aus-3-System. Nach einiger Zeit (Schnittpunkt der beiden Graphen im Bild) kehrt sich dieser Effekt jedoch um. Anfangs sind die einzelnen Funktionswahrscheinlichkeiten noch hoch und bei einem 3-aus-5-System können bis zu 2 Komponenten ausfallen. Wenn die Funktionswahrscheinlichkeit einen bestimmten Punkt unterschreitet, ist ein 2-aus-3-System sicherer, da nur 2 Komponenten zum Betrieb notwendig sind
  - Dynamische Redundanz: Vorhandensein redundanter Systeme, die erst nach Auftreten eines Fehlers aktiviert werden. Kann zuvor ungenutzt oder fremdgenutzt sein. Auch gegenseitige Redundanz möglich.

1.7 Grundlagen der Parallelverarbeitung

- **Formen**
  - Nebenläufigkeit: Objekte werden vollständig gleichzeitig abgearbeitet
  - Pipelining: Bearbeitung von Objekten wird in sequentielle Teilschritte zerlegt. Pipelinephasen dürfen für verschiedene Objekte überlappen.
- **Ebenen**
  - Programmebene: Vom OS organisisert, vollständig unabhängige Einheiten die parallel verarbeitet werden
  - Prozess/Taskebene: Programm wird in parallel ausführbare Prozesse zerlegt, jeweils eigener Adressraum. Benötigt Synchronisation und Kommunikation, Primitive durch OS implementiert
  - Blockebene: Threads, teilen sich Adressraum, Synchronisation über Mutex usw., dafür geringerer Aufwand für Erzeugung/ Beendigung/ Wechsel im Vergleich zum Prozess
  - Anweisungs-/Befehlsebene: Parallele Ausführung elementarer Anweisungen, Optimierende Compiler, Umordnen und Parallelisieren von Befehlen, Superskalar-Technologie
  - Suboperationsebene: Aufbrechen elementarer Anweisungen in parallel auszuführende Suboperationen durch Compiler, bspw. überlappte Ausführung bei Vektorrechner in Vektorpipeline
- Körnigkeit der Parallelität: Verhältnis Rechenaufwand zu Synchronisationsaufwand.

1.8 Klassifikation von Rechnerarchitekturen

- **Flynn'sche Taxonomie** Zahl Befehls- und Datenströme
  - Single Instruction, Single Data (SISD): Uniprozessor
  - Single Instruction, Multiple Data (SIMD): Vektorrechner, Feldrechner
  - Multiple Instruction, Single Data (MISD): Zuordnung Umstritten („darf es nicht geben“ vs. „Großrechner oder Supercomputer“)
  - Multiple Instruction, Multiple Data (MIMD): Multiprozessor

2 Parallelismus auf Machinenbefehlsebene

2.1 Pipelining

- RISC (Reduced Instruction Set Computers): Einfache, einzyklische Maschinenbefehle; Load/Store-Architektur; Einzyklus-Maschinenbefehle schaffen einheitliches Zeitverhalten, optimierende Compiler
- Zerlegung der Ausführung einer Maschinenoperation in Teilphasen, die dann von hintereinander geschalteten Verarbeitungseinheiten taktsynchron ausgeführt werden, wobei jede Einheit genau eine spezielle Teiloperation ausführt. Die Stufen einer Pipeline werden durch Pipeline-Register getrennt
- Stufen einer Standard-RISC-Pipeline (DLX-Pipeline): `Instruction Fetch (IF)`, `Instruction Decode (ID)`, `Execution (EX)`, `Memory Access (MA)` und `Writeback (WB)`, wobei alle Stufen unterschiedliche Ressourcen benutzen
- Idealerweise wird mit jedem Takt ein Befehl beendet, Zykluszeit abhängig von der langsamsten Pipelinestufe
- Gleitkommeverarbeitung und Integer-Division: Einführung spezieller Rechenwerke, um die Berechnung innerhalb eines Schrittes ausführen zu können
- **Verfeinerung der Pipeline-Stufen („Superpipelining“)**
  - Weitere Unterteilung der Pipeline-Stufen
  - Weniger Logik-Ebenen pro Pipeline-Stufe
  - Erhöhung der Taktrate
  - Führt aber auch zu einer Erhöhung der Ausführungszeit pro Instruktion
- Es kann zu Pipeline-Konflikten kommen, die die Instruktionsausführung im nächsten Taktzyklus verhindern. Behebung durch Anhalten der Pipeline bis zur betroffenen Instruktion (Pipeline stall) oder einfügen eines Leerzyklus (Pipeline Bubble).
- **Konfliktursachen**
  - Strukturkonflikte durch gemeinsame Ressourcen
  - Datenkonflikte aufgrund von Datenabhängigkeiten
  - Steuerkonflikte bei Instruktionen, die den Befehlszähler (Speicheradresse des auszuführenden Befehls) verändern

2.2 Nebenläufigkeit

2.2.1 Superskalartechnik

- Mehrfachzuweisung: Pro Takt können mehrere Befehle den Ausführungseinheiten zugeordnet und die gleiche Anzahl von Befehlsausführungen pro Takt beendet werden. Neben Pipelining übliche Parallelisierungsmethode für Mikroprozessoren.
- RISC-Eigenschaften bleiben weitestgehend erhalten: Lade-/Speicher-Architektur sowie festes Befehlsformat
- Entwurfsziel: Erhöhung des IPC (Instruction per Cycle)
  - **1. In-order-Abschnitt**
    - Befehle werden entsprechend ihrer Programmordnung bearbeitet
    - Umfasst: Befehlsholphase (IF), Dekodierphase (ID) und Zuordnungsphase (Dispatch)
    - Dynamische Zuordnung der Befehle an die Ausführungseinheiten. Der Scheduler bestimmt die Anzahl der Befehle, die im nächsten Takt zugeordnet werden können
  - **Out-of-order-Abschnitt**
    - Ausführungsphase
  - **2. In-order-Abschnitt**
    - Gültigmachen der Ergebnisse entsprechend der ursprünglichen Programmordnung (Retire)
    - Erhalten der korrekten Programmsemantik (Ausnahmeverarbeitung, Spekulation)

FYI: Datenabhängigkeiten

- Echte Datenabhängigkeiten (Read-after-Write): Ein Operand wurde verändert und kurz darauf gelesen. Da der erste Befehl den Operanden evtl. noch nicht fertiggeschrieben hat (Pipeline-Stufe „store“ ist weit hinten), würde der zweite Befehl falsche Daten verwenden. Ein „Shortcut“ im Datenweg der Pipeline kann den Hazard vermeiden. Bei problematischeren Situationen, wenn beispielsweise ein Rechenergebnis zur Adressierung verwendet wird oder bei berechneten und bedingten Sprüngen, ist ein Anhalten der Pipeline aber unumgänglich.

- Gegenabhängigkeit (Write-after-Read): Ein Operand wird gelesen und kurz danach überschrieben. Da das Schreiben bereits vor dem Lesen vollendet sein könnte, könnte der Lese-Befehl die neu geschriebenen Werte erhalten. In der normalen Pipeline eher kein Problem.

- Ausgabeabhängigkeit (Write-after-Write): Zwei Befehle schreiben auf denselben Operanden. Der zweite könnte vor dem ersten Befehl beendet werden und somit den Operanden mit einem falschen Wert belassen.

FYI: Spekulative Ausführung In modernen Prozessoren werden Maschinenbefehle in mehreren Verarbeitungsschritten innerhalb einer Verarbeitungskette (Pipeline) ausgeführt. Um die Leistungsfähigkeit des Prozessors zu maximieren, wird, nachdem ein Befehl in die Pipeline geladen wurde und z. B. im nächsten Schritt mit der Analyse des Befehls fortgefahren werden soll, gleichzeitig mit dem Laden des nächsten Befehles begonnen. Es befinden sich also (meistens) eine ganze Reihe von Befehlen zur sequentiellen Abarbeitung in der Pipeline. Wird jetzt am Ende der Pipeline festgestellt, dass ein bedingter Sprung ausgeführt wird, so sind alle in der Pipeline anstehenden und teilabgearbeiteten Befehle ungültig. Der Prozessor löscht jetzt die Pipeline und lädt diese dann von der neuen Programmcodeadresse neu. Je mehr Stufen die Pipeline hat, desto mehr schon berechnete Zwischenergebnisse müssen verworfen werden und um so mehr Takte wird die Pipeline nur partiell genutzt. Das reduziert die Abarbeitungsgeschwindigkeit von Programmen und reduziert die Energieeffizienz.[1]

- Ziel: Möglichst frühes Erkennen eines Sprungbefehls und Erkennen seiner Sprungzieladresse, damit die Befehle am Sprungziel möglichst ohne NOPs in die Pipeline gegeben werden können

- Beinhaltet die Vorhersage, ob ein Sprung ausgeführt wird und berechnet die Zieladresse des Sprungs

- **Statische Sprungvorhersage**
  - Vorhersage wird beim Compilieren eingebaut und ändert sich während des Programmablaufs nicht. Genauigkeit etwa bei 55 bis 80 % (Quelle: Wikipedia)
  - Geht bei Schleifen davon aus, dass Sprünge häufig ausgeführt werden, während dies bei Auswahlverfahren seltener vorkommt
  - Sprungvorhersagetechniken
    * `Stall/Freeze`: Wird während der ID-Phase ein Sprungbefehl festgestellt, wird die Pipeline angehalten bis in der EX-Phase bekannt ist, ob der Sprung ausgeführt wird
    * `Predict taken`: Geht immer davon aus, dass ein Sprung ausgeführt wird (verwendet bei Schleifen)
    * `Predict not taken`: Geht immer davon aus, dass ein Sprung nicht ausgeführt wird (verwendet bei Auswahlverfahren)

- **Dynamische Sprungvorhersage**
  - Sprungvorhersage wird zur Laufzeit von der CPU ausgeführt. Genauigkeit bei etwa 98% (Quelle: Wikipedia)
  - Sprungvorhersagetechniken
    * Der `Branch History Table` protokolliert die letzten Sprünge in einer Hashtabelle
    * `1-Bit-Prädikator`: Zu jedem Sprung wird ein Bit gespeichert. Ist es gesetzt, dann wird ein gespeicherter Sprung genommen. Bei Falschannahme wird dessen Bit invertiert. Problem: Alternierende Sprünge werden nicht berücksichtigt → `n-Bit-Prädikator`
    * `2-Bit-Prädikator`: Speichert vier Zustände und setzt das Korrektheitsbit erst nach 2 Fehlschlägen neu. Zustände sind `Predict strongly taken (11)`, `Predict weakly taken (10)`, `Predict weakly not taken (01)` und `Predict stronly not taken (00)`. In der Praxis bringen Prädikatoren mit mehr als 2 Bit kaum Vorteile.
  - Sprungzielvorhersagetechniken
    * Erweitert die Sprungvorhersage um eine Sprungzielvorhersage. Somit kann der Programmzähler sofort auf dieses Sprungziel gesetzt werden und die dortigen Instruktionen können in die Pipeline laden werden
    * Sprungzielcache: `Branch Target Address Cache` (Tabelle: Adresse der Verzweigung → Sprungzieladresse) und `Branch Target Buffer` (Direct-mapped-Cache) speichern die Adresse der Verzweigung und das entsprechende Sprungziel
    * `Branch Prediction Buffer`: Paralleler Zugriff auf den Befehlsspeicher und den BPB in der Befehlsholphase. Falls die Instruktion eine Verzweigung ist, bestimmt die Vorhersage die nächste zu holende Instruktion und berechnet die Adresse des Befehls. Nach Ausführung der Verzweigung wird die Sprungsvorhersage verifiziert und der Eintrag im BPB ggf. aktualisiert

---

[1] `https://de.wikipedia.org/wiki/Sprungvorhersage#.C3.9Cbersicht`

Superskalare Prozessorpipeline

- **Befehlsholphase (IF-Phase)**
  - Befehlsbereitstellung
    - Holen mehrerer Befehle aus dem Befehlscache in der Befehlsholpuffer (Anzahl entspricht typischerweise der Zuordnungsbreite)
    - Welche Befehle geholt werden hängt von der Sprungvorhersage ab
  - Verzweigungseinheit
    - Überwacht die Ausführung von Sprungbefehlen
    - Spekulatives Holen von Befehlen und Spekulation über den weiteren Programmverlauf (Verwendung hierzu der Vorgeschichte)
    - Gewährleistet im Falle einer Fehlspekulation die Abänderung der Tabellen sowie das Rückholen der fälschlicherweise ausgeführten Befehle
  - **Sprungvorhersage**
    - Statisch: Vorhersagerichtung für einen Befehl immer gleich. Für superskalare Prozessorarchitekturen zu unflexibel
    - Dynamisch: Verzweigungsrichtung in Abh. der Vorgeschichte, berücksichtigt Programmberhalten. Dafür hoher Hardwareaufwand. Lernen durch Abgleich mit Realität und Ändern zukünftiger Voraussagen. Mit zwei Bits feingranularer möglich.
  - Befehlsholpuffer: Entkoppelt die IF-Phase von der ID-Phase
- **Dekodierphase (ID-Phase)**
  - Dekodierung der im Befehlspuffer abgelegten Befehle. Die Anzahl entspricht typischerweise der Befehlsbereitstellungsbandbreite
  - Bei CISC-Architekturen (z.B. IA-32): Mehrere Schritte zur Dokodierung notwendig. Bestimmung der Grenzen der geholten Befehle sowie Generierung einer Folge von RISC-ähnlichen Befehlen. Ermöglicht effizientes Pipelining und superskalare Verarbeitung
  - Registerumbenennung: Dynamische Umbenennung der Operanden- und Resultatsregister. Zur Laufzeit wird für jeden Befehl das jeweils spezifizierte Zielregister auf ein unbelegtes physikalisches Register abgebildet. Automatische Auflösung von Namensabhängigkeitskonflikten (Write-after-Read, Write-after-Write, Read-after-Write)
  - Befehlsfenster (instruction window): Durch das Schreiben der Befehle in ein Befehlsfenster sind diese durch die Sprungvorhersage frei von Steuerflussabhängigkeiten und aufgrund der Registerumbenennung frei von Namensabhängigkeiten
- **Zuordnungsphase (Dispatch)**
  - Zuführung der im Befehlsfenster wartenden Befehle zu den Ausführungseinheiten sowie dynamischer Auflösung von echten Datenabhängigkeiten (Befehle die den Output eines anderen Befehls benötigen) und Ressourcenkonflikten
  - Zuordnung bis zur maximalen Zuordnungsbandbreite pro Takt
  - Zweistufige Zuweisung: Jeder Ausführungseinheit ist ein Umordnungspuffer (den sie sich ggf. mit anderen Ausführungseinheiten teilt) vorgelagert. Zuordnung eines Befehls an einen Umordnungspruffer kann nur erfolgen, wenn dieser einen freien Platz hat, ansonsten müssen die nachfolgenden Befehle warten (Auflösung von Ressourcenkonflikten)
  - Rückordnungspuffer (Reorder buffer): Festhalten der ursprünglichen Befehlsanordnung sowie Protokollierung der Ausführungszustände der Befehle in den folgenden Phasen
- **Befehlsausführung**
  - Ausführung der im Opcode spezifizierten Operation und Speichern des Ergebnisses im Zielregister (Umbenennungsregister)
  - Completion: Eine Instruktion beendet ihre Ausführung, unabhängig von der Programmordnung, wenn das Ergebnis bereitsteht. Danach: Bereinigung der Reservierungstabellen und Aktualisieren des Rückordnungspuffer
- **Rückordnungsstufe (Retire)**
  - Commitment: Nach Vervollständigung beenden die Befehle ihr Bearbeitung (Commitment) und werden in der Programmreihenfolge gültig gemacht. Ggf. werden Ergebnisse aus Umbenennungsregistern gültig gemacht durch zurückschreiben. Bedingungen:
    - Die Befehlsausführung ist vollständig
    - Alle Befehle, die in der Programmordnung vor dem Befehl stehen, haben bereits ihre Bearbeitung beendet oder beenden diese im selben Takt
    - Der Befehl hängt von keiner Spekulation ab
    - Vor oder während der Bearbeitung ist keine Unterbrechung aufgetreten
  - Bei Aufreten einer Unterbrechung: Precise interrups!
    - Alle Resultate, die in der Programmausführung vor dem Befehl stehen, werden gültig gemacht; die Ergebnisse aller nachfolgenden werden verworfen
    - Das Ergebnis des Befehls, der die Unterbrechung verursacht hat, wird in Abhängigkeit der Unterbrechung und der Architektur gültig gemacht oder verworfen
    - Komplexe Hardware notwendig

Dynamische Methoden zur Erkennung und Auflösung von Datenkonflikten am Beispiel Tomasulo (IBM 360/91)

- Ziel: Fortsetzung der Ausführung von Befehlen, auch wenn Datenabhängigkeiten vorliegen
- **Vorgehen zum Verhindern von Konflikten**
  - Read-after-Write: Der Prozessor verfolgt, wann Operanden zur Verfügung stehen
  - Write-after-Read und Write-after-Write: Nutzung von *Reservierungstabelle/Reservation Stations*, die Registerinhalte zwischenspeichert und so vor vorzeitigem Überschreiben schützt
- **Funktionsweise**[2]
  - Issue: Der Befehl an der aktuellen Position in der Operation Queue wird dekodiert und entsprechend seiner auszuführenden Operation in eine passende Reservation Station eingetragen. Operanden werden direkt aus der Registerdatei übernommen, wenn sie gültig sind. Dieser Vorgang wird als Registerumbenennung bezeichnet. Steht ein Operand noch nicht zur Verfügung, wird stattdessen die Adresse der RS eingetragen, die den Wert gerade berechnet. Ist keine passende RS frei, verbleibt der Befeht in der Operation Queue und die Zuweisung wird im nächsten Takt erneut versucht
  - Execute: Sobald alle Operanden in der Reservation Station zur Verfügung stehen, wird die Operation an die FU weiter gegeben und ausgeführt. Andernfalls wird der Common Data Bus/Ergebnisbusses auf eingehende Werte beobachtet und fehlende Operanden übernommen, wenn die Adresse der Quell-RS mit der benötigten Adresse übereinstimmt
  - Write Result: Sobald das Ergebnis der Operation berechnet wurde, wird es mitsamt der Adresse der ausgeführten RS auf den Common Data Bus gelegt und somit für die RS sichtbar, welche auf das Ergebnis warten

Zusammenfassung Superskalartechnik

- Aus einem sequentiellen Befehlsstrom werden Befehle zur Ausführung angestoßen
- Die Zuordnung erfolgt dynamisch durch die Hardware
- Es kann mehr als ein Befehl zugewiesen werden. Die Anzahl der zugewiesenen Befehle pro Takt wird dynamisch von der Hardware bestimmt und liegt zwischen Null und der maximalen Zuordnungsbreite
- Komplexe Hardwarelogik für dynamische Zuweisung notwendig
- Mehrere, voneinander unabhängige Funktionsanweisungen verfügbar
- Mikroarchitektur, da der Befehlssatz nicht verändert wird. Technisch gesehen „nur“ eine Erweiterung der Pipeline
- **FYI: Formen**[3]
  - Superskalare Prozessoren mit statischem Scheduling: Die Anzahl der pro CPU-Zyklus parallel ausführbaren Befehle ist nicht vorgegeben, sondern wird durch die CPU dynamisch bestimmt. Da es sich um statisches Scheduling handelt, wird die Reihenfolge der Befehle vom Compiler vorgegeben
  - Superskalare Prozessoren mit dynamischem Scheduling: Die CPU bestimmt sowohl, welche Befehle parallel ausgeführt werden, als auch die Reihenfolge, in der dies geschieht (Out-of-order execution)
  - VLIW-Prozessoren: Die Architekturen benutzen deutlich längere Befehle, in denen die parallel auszuführenden Befehle vorgegeben werden

2.2.2 Very Long Instruction Word (VLIW)

- Ziel: Beschleunigen der Abarbeitung durch Parallelität auf Befehlsebene
- Breites Befehlsformat, das in mehrere Felder aufgeteilt ist, aus denen die Funktionseinheiten (EX) gesteuert werden
- Eine VLIW-Architektur mit `n` unabhängigen Funktionseinheiten kann bis zu `n` Operationen gleichzeitig ausführen
- Operationen ähnlich zu Befehlssätzen in RISC-Architektur
- Steuerung der parallelen Abarbeitung zur Übersetzungszeit (automatisch parallelisierender Compiler). Der Compiler gruppiert die Befehle, die parallel ausgeführt werden können
- Die Gruppengröße ist abhängig von der Anzahl der Ausführungseinheiten
- Vorteil gegenüber superskalaren Prozessoren: Weniger Hardware-Logik notwendig → mehr Platz auf dem Chip für zusätzliche Funktionalität bei beispielsweise mehr Ausführungseinheiten
- Vorteile: Einfacher Kontrollpfad sowie Ausnutzungsmöglichkeiten der Compilertechnik (z.B. Softwarepipeling, Schleifenparallelisierung, etc.)
- Nachteil: Portierung des Codes auf andere Prozessoren eventuell schwierig

Statische Steuerung der parallelen Abarbeitung

- Zusätzliche Aufgaben für den Compiler: Code-Generierung (Kontrollflussanalyse, Datenflussanalyse, Datenabhängigkeitsanalyse), Schleifenparallelisierung, Scheduling (Beispiel auf Folie 8-33)
- Software-Pipelining: Technik zur Reorganisation von Schleifen. Jede Iteration im generierten Code enthält Befehle aus verschiedenen Iterationen der ursprünglichen Stufe

[2] https://de.wikipedia.org/wiki/Tomasulo-Algorithmus#Funktionsweise
[3] https://de.wikipedia.org/wiki/Superskalarit%C3%A4t

2.2.3 Multithreading (Mehrfädigkeit)

- Entwurfsziel: Reduzieren der Untätigkeits- oder Latanzzeiten, die bei Speicherzugriffen (insbesondere Cache-Fehlzugriffe) entstehen
- Ziel: Parallele Ausführung mehrerer Kontrollfäden
- **Ansätze**
  - Interleaved Multithreading (cycle-by-cycle): In jedem Zyklus wird ein Befehl aus einem anderen Kontrollfaden geholt und ausgeführt. Problem: Wenn Thread ohne Wartezeiten ausgeführt werden könnte, ist er mit dieser Technik deutlich langsamer
  - Blocked Multithreading: Die Befehle eines Threads werden solange ausgeführt bis ein Ereignis eintritt, das eine lange Wartezeit nach sich zieht. Bearbeitung daher nicht so stark verlangsamt, die Pipeline muss jedoch bei Wechsel geleert und neugestartet werden, das ist nur bei langen Wartezeiten sinnvoll
  - Simultaneous Multithreading: Mehrfach superskalar, die Ausführungseinheiten werden über eine Zuordnungseinheit aus mehreren Befehlspuffern versorgt. Jeder Befehlspuffer stellt einen anderen Befehlsstrom dar und hat einen eigenen Registersatz zugeordnet. Trade-off zwischen Geschwindigkeit pro Thread und Durchsatz an Threads.

3 Multiprozessoren - Parallelismus auf Prozess-/Blockebene

3.1 Allgemeine Grundlagen

3.1.1 Parallele Architekturmodelle

- **Multiprozessor mit gemeinsamem Speicher: Uniform Memory Access (UMA)**
  - Gleichberechtigter Zugriff der Prozessoren auf die Betriebsmittel
  - Gemeinsamer Speicher mit globalem Adressraum, Austausch von Daten über gemeinsamen Speicher durch LS-Operationen
  - Beispiele: Symmetrischer Multiprozessor (SMP), Multicore-Prozessor
- **Multiprozessor mit verteiltem Speicher: No Remote Memory Access (NORMA)**
  - Physikalisch verteilter Speicher
  - Jeder Knoten mit einem privaten Adressraum
  - Kommunikation durch Nachrichtenaustausch über ein Interconnect Network
  - Beispiel: Cluster
- **Multiprozessor mit verteiltem gemeinsamen Speicher: Non-Uniform Memory Access (NUMA)**
  - Globaler Adressraum über mehreren, exklusiv jeweils einem Prozessor zugeordneten Speichereinheiten
  - Speicher physikalisch verteilt
  - Zugriff auf entfernten Speicher über LS-Operationen (über ein Interconnect Network)
  - Beispiel: Cache-Coherent Non-Uniform Memory Access oder Distributed shared Memory (DSM)

3.2 Parallele Programmiermodelle

- Programmiermodell: Abstraktion einer parallelen Maschine, die spezifiziert, wie Teile des Programms parallel abgearbeitet werden und wie Informationen ausgetauscht werden können
- **Parallele Programmierung**
  - Aufteilung der Arbeit (work partitioning): Identifizieren der Teilaufgaben, die parallel ausgeführt und auf mehrere Prozessoren verteilt werden können (Programmsegmente müssen unabhängig von einander sein)
  - Koordination (coordination): Koordination/Synchronisierung/Kommunikation (zwischen) den/der Prozesse
  - Sychronisation und Koordination: Austausch von Informationen über gemeinsamen Speicher oder über explizite Nachrichten. Zusätzlicher Zeitaufwand hat Auswirkung auf die Ausführungszeit des parallelen Programms
  - : Parallelismus: Daten (z.B. Matrixmultiplikation) oder Funktion (unabhängige Funktionen)
  - **Gemeinsamer Speicher (Shared Memory)**
    - Verwendung konventioneller Speicheroperationen für die Kommunikation über gemeinsame Adressen
    - Atomare Synchronisationsoperationen
  - **Nachrichten (Message Passing)**
    - Kein gemeinsamer Adressbereich, Kommunikation der Prozesse (Threads) mit Hilfe von Nachrichten
    - Kommunikationsarchitektur: Verwendung von korrespondierenden Send- und Receive-Operationen
  - Beispiel: OpenMP für gemeinsamen Speicher, MPI für verteilten Speicher
- Durchzuführende Schritte beim Parallelisierungsprozess: Dekomposition, Zuweisung, Festlegung, Abbildung

3.3 Quantitative Maßzahlen

3.3.1 Definitionen

- $P(1)$ bzw. $P(n)$: Anzahl der auszuführenden Operationen/Tasks auf einem Einprozessor- bzw. Multiprozessorsystem
- $T(1)$ bzw. $T(n)$: Ausführungszeit auf einem Einprozessor- bzw. Multiprozessorsystem in Schritten oder Takten

3.3.2 Parallelitätsprofil

- Misst die entstehende Parallelität in einem Programm oder bei der Ausführung auf einem Parallelrechner und liefert so eine Vorstellung von der inhärenten Parallelität eines Algorithmus/Programms
- Grafische Darstellung: XY-Diagramm mit Anzahl der parallelen Aktivitäten in zeitlicher Abhängigkeit → Perioden von Berechnungs- Kommunikations- und Untätigkeitszeiten sind erkennbar
- Der Parallelitätsgrad $PG(t)$ gibt die Anzahl der Tasks an, die zu einem Zeitpunkt parallel bearbeitet werden können
- Leistungsangaben zu Multiprozessorsystemen werden mit Leistungsangaben zu Einprozessorsystemen in Beziehung gesetzt
- **Parallelindex I (Mittlerer Grad des Parallelismus)**
  - Durchschnittliche Parallelität pro Zeiteinheit
  - Kontinuierlich: $I = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} PG(t)dt$
  - Diskret: $\left(\sum_{i=1}^{m} i \cdot t_i\right) / \left(\sum_{t=1}^{m} t_i\right)$
- $S(n) = \frac{T(1)}{T(n)}$ gibt den Speedup an, die Effizienz berechnet sich zu $E(n) = \frac{S(n)}{n}$
- $U(n) = \frac{I(n)}{n}$ als Auslastung

3.3.3 Skalierbarkeit eines Parallelrechners

- Das Hinzufügen von weiteren Verarbeitungselementen führt zu einer kürzeren Gesamtausführungszeit, ohne dass das Programm geändert werden muss
- Bei fester Problemgröße und steigender Prozessorzahl wird eine Sättigung eintreten

3.3.4 Gesetz von Amdahl

- Ein Programm kann nie vollständig parallel ausgeführt werden, da einige Teile wie Prozess-Initialisierung oder Speicherverwaltung nur einmalig auf einem Prozessor ablaufen oder der Ablauf von bestimmten Ergebnissen abhängig ist→ Zerlegung des Programmlaufs in Abschnitte, die entweder vollständig sequentiell oder vollständig parallel ablaufen und fasst sie zu jeweils einer Gruppe zusammen[1]
- Gesamtausführungzeit (Gesetz von Amdahl): $T(n) = T(1) \cdot \frac{1-a}{n} + T(1) \cdot a$, a: Anteil des Programms, der nur sequentiell ausgeführt werden kann
- Superlinearen Speedup kann es nicht geben, da jeder paralle Algorithmus einen sequentiellen Teil hat, der die Beschleunigung begrenzt: $1 \leq S(n) \leq n$. Tritt jedoch in der Realität dennocg auf: Ursache ist beispielsweise, dass bei einem Mehrprozessorsystem die Daten vollständig in lokale Caches und Hauptspeicher passen (kein Verlust durch häufige Seitenwechsel)

3.4 Verbindungsnetzwerke

3.4.1 Verbindungsstrukturen

- **Verbindungsnetzwerke in Multiprozessoren**
  - Ermöglichen die Kommunikation und Kooperation zwischen den Verarbeitungselementen
  - Einsatz: Chip-Multiprozessor (NoC, beispielsweise Verbinden des gemeinsamen L2-Cache), Multiprozessor mit gemeinsamem oder verteilten Speicher
- **Charakterisierung von Verbindungsnetzwerken**
  - Latenz
    - Übertragungszeit einer Nachricht $T_{msg} = t_{\text{message startup time}} + t_{\text{message transfer time}}$, Vorraussetzung: Das Netzwerk ist konfliktfrei
    - Kanalverzögerung (channel delay): Dauer für die Belegung eines Kommunikationskanals durch eine Nachricht
    - Schaltverzögerung: Zeit, die benötigt wird um einen Weg zwischen zwei Knoten aufzubauen. Pfadberechnung oder Wegfindung
    - Blockierungszeit: Wird verursacht, wenn zu einem Zeitpunkt mehr als eine Nachricht auf eine Netzwerkressource zugreifen will
  - Durchsatz/Übertragungsbandbreite
  - Bisektionsbandbreite: Maximale Bandbreite, die das Netzwerk über die Bisektionslinie (teilt das Netzwerk in zwei Hälften) transportiert werden kann
  - Diameter/Durchmesser: Maximale Distanz zwischen zwei Prozessoren/Knoten
  - Verbindungsgrad: Anzahl der Direktverbindungen pro Knoten
  - Mittlere Distanz: Durchschnittlicher Abstand zwischen zwei Knoten

[1] https://de.wikipedia.org/wiki/Amdahlsches_Gesetz#Beschreibung

  - Komplexität/Erweiterbarkeit/Skalierbarkeit
  - Ausfalltoleranz/Redundanz
  - **Art des Transfers**
    * Durchschalte- oder Leistungsvermittlung: Schalten einer Direktverbindung zwischen zwei Knoten, blockierungsfrei, kurze Latenz, gut geeignet für lange Nachrichten
    * Paketvermittlung: Datenpakete fester Länge werden entsprechend einem Wegfindealgorithmus übertragen, geeignet für kurze Nachrichten

- **Verbindungsnetzwerk (IN)**
  - Knoten: über Netzwerkschnittstelle verbunden
  - Switch/Schaltelement: setzt Verbindungen
  - Link: Verbindungen, synchron mit gemeinsamer Taktquelle oder asynchron über Handshake
  - Nachricht: Uni-/Multi-/Broadcast, Fixed length möglich

- **Datentransfer**
  - Durchschalte- oder Leistungsvermittlung/Circuit Switching: Es wird eine direkte Verbindung zwischen Knoten geschaltet, keine Unterbrechung, keine Routing-Information im Paket nötig. Dafür blockieren der Leitung
  - Paketvermittlung/Packet switching: Pakete mit fester Länge werden geroutet, bedarfsorientiertes Blocken der Leitungen. Store and forward von Paketen oder Cut-through Switching

- End-to-end packet latency model: Betrachte Gesamtlatenz. $\texttt{E2E latency} = \texttt{Sender Overhead + Time of flight+Transmission time+}$
- Effektive Bandbreite: $EB = \frac{\texttt{Paketgröße}}{max\texttt{Sender OH, Empfänger OH, Übertragungszeit}}$

- **Verbindungsnetze**
  - Statische Verbindungsnetze
    * Nach Aufbau des Verbindungsnetzes bleiben die Strukturen fest
    * Vorhersagbare Kommunikationsmuster zwischen Knoten
    * Vollständige Verbindung: Alle Knoten sind miteinander verbunden. Höchste Leistungsfähigkeit aber nicht praktikabel bei hoher Prozessorzahl (Aufwand steigt qudratisch mit der Prozessorzahl)
    * Gitterstrukturen: Verbinden benachbarte Knoten. Disjunkte Bereiche können parallel genutzt werden, allerdings oft mehrere Schritte notwendig (bei unbenachbarten Knoten)
    * Unidirektionaler Ring: Nachrichten werden vom Quellknoten zum Zielknoten geschickt. Bei Ausfall bricht die Kommunikation zusammen
    * Bidirektionaler Ring: Übertragung kann die kürzeren der beiden Wege wählen. Bei einer Unterbrechung bleibt die Kommunikation bestehen
    * Chordaler Ring: Hinzufügen weiterer Direktverbindungen zur Erhöhung der Fehlertoleranz
    * Baumstruktur: Teilweise mit steigender Anzahl an Kommunikationskanälen in Richtung Wurzel
    * Diverse Kubusstrukturen, beispielsweise Hyperkubus: Binäre Bezeichnung der Knoten ermöglicht Routenbestimmung (benachbarte Knoten unterscheiden sich um genau eine Stelle, Start- und Zieladresse werden per XOR verknüpft). Allerdings schlechte Skalierbarkeit, da bei jeder Erweiterung der Knotengrad steigt und alle Knoten erweitert werden müssen
      · Anzahl der Knoten eines K-nären n-Kubus: $N = K^n$
      · Knoten-/Verbindungsgrad: $2n$
  - Dynamische Verbindungsnetze
    * Geeignet für dynamische Anwendungen mit variablen und nicht regulären Kommunikationsmustern
    * Bus: Wird von allen an den Bus angeschlossenen Prozessoren gemeinsam genutzt. Verwendung von Cache-Speichern (mit Cache-Kohärenz-Protokollen zur Synchronisiation) zur Reduzierung des Busverkehrs. Synchronisation durch Split-Phase Busprotokollen erforderlich
    * Kreuzschienenverteiler (Crossbar)
      · Hardwaresystem, das in einer Menge an Prozessoren zwischen allen disjunkten Paaren von Prozessoren blockierungsfreie Direktverbindungen zur Kommunikation aufbauen kann. Schaltelemente an allen Kreuzungspunkten → hoher Hardwareaufwand
      · Schalterelemente (2x2-Kreuzschienenverteiler) bestehen aus Zweierschaltern mit je zwei Ein- und Ausgängen, die entweder durch- oder über kreuz schalten können
    * Omega-Netz: Besteht aus $\frac{N \cdot log_k(N)}{k}$ Crossbars bei N Knoten mit Switch-Grad k. Nicht blockierungsfrei
    * Mischpermutationnetz: Kreisverschiebung der Adressbits
    * Allgemeine Beispiele für dynamische Verbindungsnetzwerke: Bus, Kreuzschiene, Schaltnetzwerk
    * Kreuzpermutation: Abbildung auf invertierte ID. Beispielsweise $001 \longrightarrow 100$
    * Mischpermutation: Addieren von $\forall 0..\frac{n-1}{2} : 2n$. Beispielsweise $000 \longrightarrow 000, 001 \longrightarrow 010$. Die obere Hälfte invertiert

3.5 Multiprozessoren mit gemeinsamem Speicher

- Gültigkeitsproblem: Paralleler Zugriff mehrere Prozessoren auf den Hauptspeicher → mehrere Kopien des gleichen Speicherwortes müssen miteinander in Einklang gebracht werden
- Inklusionseigenschaft bei Caches: Der Inhalt des Caches auf der höchsten Stufe ist auch in den Caches niedrigerer Stufen enthalten
- **Cache-Kohärenz-Problem**
  - 1. Fall (IO-Problem): System mit einem Mikroprozessor und weiteren Komponenten mit Master-Funktion (ohne Cache)
    - Zusätzlicher Master (beispielsweise DMA-Controller) kann Kontrolle über den Bus übernehmen und prozessorunabhängig auf den Hauptspeicher zugreifen
    - Problem: Prozessor oder DMA-Controller lesen eventuell veraltete Daten (stale data)
    - Lösungen des Kohärenzproblems
      - Non-Cacheable Data: Der gemeinsam genutzte Speicherbereich wird nicht gecacht bzw. mit „non-cacheable gekennzeichnet“
      - Cache-Clear/Cache-Flush: Nach einem DMA-Vorgang wird der Cache automatisch neu geladen
  - 2. Fall: Speichergekoppeltes Multiprozessorsystem
    - Mehrere Prozessoren mit jeweils eigenem Cache-Speicher sind über einen Systembus an einen gemeinsamen Hauptspeicher angebunden
- **Kohärenz**
  - Vereinfachte Definition „Cache-Kohärenz“: Ein Speichersystem ist kohärent, wenn jeder Lesezugriff auf ein Datum den aktuellen, geschriebenen Wert dieses Datums liefert
  - Ein paralleles Programm, das auf einem Multiprozessor läuft, kann mehrere Kopien eines Datums in mehreren Caches haben
  - Möglichkeiten, die Kohärenzanforderungen zu erfüllen
    - Write-invalidate-Protokoll: Sicherstellen, dass ein Prozessor exklusiven Zugriff auf ein Datum hat, bevor er schreiben darf. Vor dem Verändern einer Kopie in einem Cache-Speicher müssen alle Kopien in anderen Cache-Speichern für ungültig erklärt werden
    - Write-update-Protokoll: Beim Verändern einer Kopie in einem Cache-Speicher müssen alle Kopien in anderen Cache-Speichern ebenfalls verändert werden, wobei die Aktualisierung auch verzögert (allerdings spätestens beim Zugriff) erfolgen kann
- **Kohärenzprotokolle**
  - Bus-Snooping: Caches sind an einem gemeinsamen Bus und beobachten diesen, um feststellen zu können, ob sie eine Kopie eines Blocks enthalten, der benötigt wird
  - MESI-Kohärenzprotokoll (Hardware)
    - Jeder Cache verfügt über Snoop-Logik und Steuersignale: Invalidate-Signal (zum Invalidieren von Einträgen in anderen Caches), Shared-Signal (zeigt an, ob ein zu ladender Block bereits als Kopie besteht) und Retry-Signal (Aufforderung für den Prozessor das Laden eines Blocks abzubrechen. Nachdem ein anderer Prozessor aus dem Cache in den Hauptspeicher zurückgeschrieben hat wird das Laden wieder aufgenommen)
    - Erweiterung jeder Cache-Zeile um zwei Steuerbits zum Anzeigen der Zustände:
      - Invalid (I): Die Cache-Zeile ist ungültig. Mittels Shared-Signal zeigen die anderen Steuerungen an, ob sie diesen Block gespeichert haben (shared read hit/miss). Davon abhängig erfolgt der Übergang in S oder E. Bei einem Write-Miss erfolgt der Übergang in M
      - Shared (S): Der Speicherblock existiert sowohl lokal als auch in anderen Caches. Bei einem Schreibzugriff übergang in Zustand M und Ausgeben des Invalidate-Signals
      - Exclusive (E): Der Speicherblock existiert exklusiv nur in der Zeile dieses Caches. Lese- und Schreibzugriffe möglich, ohne den Bus benutzen zu müssen. Nach Schreibzugriff, Wechsel in E
      - Modified (M): Der Speicherblock existiert nur lokal und ist nach dem Laden verändert worden. Bei einem externen Zugriff durch einen anderen Prozessor muss dieser in den Hauptspeicher zurückkopiert werden. Der externe Prozessor wird über das Retry-Signal informiert, dass zunächst zurückgeschrieben werden muss
    - Zustandsgraphen und Beispiel in den Folien[2]
  - MOESI-Kohärenzprotokoll: Erweitert das MESI-Protokoll um den zusätzlichen Zustand `Owned` (`O`). Es vermeidet das Zurückschreiben von modifizierten Cache-Lines, wenn andere CPUs diese lesen wollen. Stattdessen wird der aktuelle Wert bei jeder Veränderung zwischen den Caches direkt propagiert[3] (praktisch, wenn ein Prozessor einen Wert in seinen Cache läd, der bereits in einem anderen Cache existiert)
  - Distributed Shared Memory
    - Multiprozessor mit verteiltem, gemeinsamem Speicher ohne Möglichkeit, die Broadcasteigenschaft des Busses zu nutzen
    - Verzeichnisbasierte, tabellenartige Cache-Kohärenzprotokolle, die in Hardware oder Software implementiert sein können

[2] VL-07 Folie 2-40 bis 2-47

[3] `https://de.wikipedia.org/wiki/MOESI`

    - Die Tabelle protokolliert für jeden Block im loken Speicher, ob dieser in den lokalen oder einen entfernten Cache-Speicher übertragen worden ist und hält die Zustände als Kopien
    - Zustände werden ähnlich denen des MESI-Protokolls definiert

- **Speicherkonsistenz**
  - Sequentielle Konsistenz: Ein Multiprozessorsystem heißt sequentiell konsistent, wenn das Ergebnis einer beliebigen Berechnung dasselbe ist, als wenn die Operation sequentiell auf einem Einprozessorsystem ausgeführt worden ist
  - Atomar Schreibzugriffe
  - Programmierer geht von sequentieller Konsistenz aus → sehr starke Leistungseinbußen bzgl. der Implementierung und der Leistung. Verbietet vorgezogene Ladeoperationen und nicht-blockierende Caches
  - Schwache Konsistenz (Speicherkonsistenzmodell)
    - Konkurrierende Zugriffe auf gemeinsame Daten werden durch Sychronisationen geschützt (beispielsweise Semaphoren oder Mutecies)
    - Idee: Konsistenz des Speicherzugriffs wird nicht mehr zu allen Zeiten gewährleistet sondern nur zu definierten Synchronisationspunkten. Einführung „kritischer Bereiche“ innerhalb denen keine Synchronisation gewährleistet sein muss
    - Voraussetzung: Hardware oder softwareseitige Unterscheidung der Synchronisationsbefehlen von den LS-Befehlen und eine sequentiell konsistente Implementierung der Synchronisationsbefehle
    - Atomare, nicht-unterbrechbare LS-Befehle auf den Speicher. Synchronisation auf Benutzerebene. Häufige Implementierung: Spin Locks oder Barriers

3.6 Multiprozessoren mit verteiltem Speicher

3.6.1 Programmiermodell

- Nachrichtenorientiert (message passing)
- **Synchrones Message Passing**
  - Sender und Empfänger blockieren bis die Nachricht beim Empfänger angekommen ist und in den gewünschten lokalen Speicher kopiert worden ist. Deadlock-Gefahr!
  - Drei-Phasen-Protokoll: `Request to send`, `Receiver ready`, `Message transfer`. Kann Sender- oder Empfer-initiiert worden sein
- Blockierendes, asynchrones Message Passing: Die Daten werden vor dem Senden in einen Puffer kopiert, so dass sie vom Hauptprogramm nicht mehr verändert werden können. Dieses blockiert so lange. Receive analog
- Nicht-blockierendes, asynchrones Message Passing: Die Kontrolle wird sofort an den Hauptprozess zurückgegeben, der Transfer läuft im Hintergrund. Zusätzliche Probe-Funktionen um zu prüfen, ob die Daten zum Empfänger kopiert worden sind
- **Hardwareunterstützung für Message passing Protokolle**
  - Großer Software-Overhead, daher Kopieren via DMA
  - Hardwareunterstützung soll die Latenz verringern und den Prozessor entlasten
  - Im Allgemeinen bietet das Verbindungsnetzwerke Übertragungsprimitive an
  - Hardwaresupport durch Kommunikationsprozessor

3.6.2 Fallstudien

- **IBM Blue Gene/L**
  - 2^16 Knoten in bis zu 64 Racks, die jeweils auch als Einzelsystem organisiert werden können
  - Knoten über ein fünfstufiges Netzwerk verbunden. Unterste Ebene: 64x32x32 3D-Torus
- **JUGENE: Juelicher BlueGene/P**
  - Im Juni 2010 Nummer 5 der Top500
  - Gesamtleistung mit der letzten Ausgbaustufe bei über einem PF/s
  - Aufbau: System mit jeweils 72 Racks mit jeweils 32 NodeCards, die etwa 13,9 TF erreichen. NodeCards bestehen aus jeweils 32 ComputerCards mit je einem Quadcore-Chip (Leistung etwa 13,6GF/s)
  - Verbindungsnetze
    - 3D-Torus, der alle 73728 ComputeNodes verbindet mit virtual cut-through hardware routing. 425MB/s auf allen node links (5,1GB/s insgesamt)
    - Collective Network (binärbaumartig organisiert) mit 850MB/s, das alle Nodes verbindet
    - Low Latency Global Barrier and Interrupt (sternförmig) für extrem niedrige Latenzen
    - 10 GBit Ethernet für externe Kommunikation
    - Control Network: GBit Ethernet für Managementaufgaben (Boot, Monitoring, Diagnose)
- **SuperMUC im Leibnitz Rechenzentrum Garching**
  - Distributed Memory Architecture mit Sandy-Bridge-Prozessoren mit jeweils acht multithreaded Cores

- Virtual Interface Architecture (VIA) und InfiniBand
  - Standardisiertes benutzerlevel Interface, das komplett flexibel in Hardware oder auf dem Hostprozessor implementiert sein kann
  - Gegensatz zur klassichen Berkeley Socket API ist der Kernel nicht mehr in jeden Transfer eingebunden, was massiv Performance spart (Kernel als Flaschenhals)

4 Vektorverarbeitung

4.1 Einführung

- Motivation: Gerade in HPC-Anwendungen fallen oft viele gleichartige Daten an, die auf ähnliche Weise verarbeitet werden sollen, so zum Beispiel bei Simulationen in der Meteorologie und Geologie (Single Instruction Multiple Data)[1]
- Verarbeitungsbeispiel: $Y = a \cdot X + Y$. X und Y sind zwei Vektoren gleicher Länge und a ist eine skalare Größe. Dieses Problem wird auf Skalarprozessoren durch eine Schleife gelöst[2], dadurch hoher Aufwand durch viele Schleifendurchläufe, Pipeline-Konflikte oder Mehrzyklusoperationen
- Verarbeitung in einem Rechenwerk mit pipeline-artig aufgebauten Funktionseinheiten
- **Code-Beispiel[3]: Berechnung von $Y = a \cdot X + Y$**

```
1  ; MIPS
2          L.D       F0, a           ; Skalar a laden
3          DADDIU    R4, Rx, #512    ; letzte Adresse 512/8 = 64
4  Loop:   L.D       F2, 0(Rx)       ; X(i) laden
5          MUL.D     F2, F2, F0      ; a * X(i)
6          L.D       F4, 0(Ry)       ; Y(i) laden
7          ADD.D     F4, F4, F2      ; a * X(i) + Y(i)
8          S.D       0(Ry), F4       ; Y(i) speichern
9          DADDIU    Rx, Rx, #8      ; Index (i) von X inkrementieren
10         DADDIU    Ry, Ry, #8      ; Index (i) von Y inkrementieren
11         DSUBU     R20, R4, Rx     ; Rand berechnen
12         BNEZ      R20, Loop       ; wenn 0, dann fertig
13
14 ; VMIPS
15         L.D       F0, a           ; Skalar a laden
16         LV        V1, Rx          ; Vector X laden
17         MULVS.D   V2, V1, F0      ; Vector-Skalar Multiplikation
18         LV        V3, Ry          ; Vector Y laden
19         ADDV.D    V4, V2, V3      ; Vektor Addition
20         SV        Ry, V4          ; Resultat speichern
```

- **Aufbau eines Vektorprozessors**
  - Einheit um Vektoren direkt aus Hauptspeicher zu laden
  - Vektoreinheit: Ein Satz Vektorregister
  - Mindestens eine Skalareinheit zur Ausführung von Befehlen, die nicht auf ganze Vektoren angewendet werden sollen (Skalarbefehle)
  - Vektoreinheiten und Skalareinheiten können parallel arbeiten → Vektorbefehle und Skalarbefehle können parallel ausgeführt werden
- **Pipeling eines Vektorprozessors**
  - Berechnung eines Ergebnisses pro Takt bei ununterbrochener Arbeit und einer Füllzeit/Einschwingzeit
  - Beispiel einer Gleitkommaoperation
    1. Laden eines Paars von Gleitkommazahlen aus Vektorregister
    2. Vergleich der Exponenten und Verschieben einer Mantisse
    3. Addition der ausgerichteten Mantisse
    4. Normalisieren des Ergebnisses und Schreiben in Zielregister
  - Verkettung von Pipelines: Erweiterung auf eine Folge von Vektoroperationen durch Verkettung der (spezialisierten) Pipelines. So können die Ergebnisse einer Pipline sofort der nächsten Pipeline zur Verfügung gestellt werden. Beispielsweise können Addition, Schieben und UND-Verknüofung unmittelbar hintereinander ausgeführt werden
  - Multifunktions- oder spezialisierte Pipelines zur Realisierung arithemtisch-logischer Verknüpfungen
    * Verwendung eines Multifunktionspipeline oder eine Anzahl von spezialisierten Pipelines
    * Multifunkspipeline: Der Aufbau erfordert eine höhere Stufenzahl, wobei aktuell nicht benötigte Stufen oder Pipelines übersprungen werden können
    * Spezialisierte Pipelines: Jeweils zur Durchführungs spezieller Funktionen benutzt. Hardware und Steuerung relativ einfach, allerdings werden mehrere unabhängige Pipelibes benötigt, um alle Funktionen abzubilden
- **Parallelität in einem Vektorrechner**

[1] https://de.wikipedia.org/wiki/Vektorprozessor#Funktionsweise_und_Anwendungsfelder
[2] https://de.wikipedia.org/wiki/Vektorprozessor#MIPS-Architekturbeispiel
[3] https://de.wikipedia.org/wiki/Vektorprozessor#MIPS-Architekturbeispiel

  - Vektor-Pipeline-Parllelität durch die Stufenzahl der Vektor-Pipeline
  - Mehrere Vektor-Pipelines in einer Vektoreinheit
  - Vervielfachen der Pipelines: Pro Takt wird nicht nur ein Operandenpaar sondern ein Vielfaches davon verarbeitet
  - Mehrere Vektoreinheiten, die parallel zueinander arbeiten (vgl. speichergekoppelter Multiprozessor)

- **Parallelarbeit in der Software**
  - Vektorisierung der innersten Schleife mittels vektorisierendem Compiler
  - Vektorverbundbefehle zur Verkettung von Vektorbefehlen (beispielsweise Vector-Multiply-Add)
  - Die Verteilung einer Berechnung auf mehrere, gleiche Pipelines geschieht durch den Compiler (Vektorisierung der innersten Schleife)

- **Memory Interleaving (Speicherverschränkung)**
  - Anpassen der Zugriffsgeschwindigkeit an die Verarbeitungsgeschwindigkeit der CPU
  - Der Speicher wird in gleich große, voneinander unabhängige Bereiche (Module, Speicherbänke) unterteilt, die zeitlich verschränkt gelesen oder beschrieben werden können. Aufeinander folgende Speicherworte werden zyklisch in aufeinander folgenden Speicherbänken abgespeichert[4]
  - Hat der Speicher eine geringere Taktrate als der Prozessor, verringern sich durch den abwechselnden Zugriff die Wartezeiten für Speicheroperationen → mehr Zeit für langsamere Speicherbausteine

4.2 Vektorverarbeitung

4.2.1 Eigenschaften

- **Vektor Stride**
  - Problem: Die Elemente eines Vektors liegen nicht immer so wie sie gebraucht werden in aufeinander folgend Speicherzellen. Beispielsweise ändert sich die Reihenfolge bei Matrizenmultiplikationen (Spaltenelemente × Zeilenelemente)
  - Stride-Wert bezeichnet den Abstand zwischen den Elementen. Dieser kann sich allerdings zur Laufzeit ändern oder ist dann erst bekannt. Lösung: Ablegen in Allzweckregister
  - In heutigen Vektorrechnern werden die Zugriffe von jedem Prozessor auf über mehrere Hundert Speicherbänke verteilt

- **Bedingte Ausführung**
  - Problem: Programme mit if-Anweisungen in Schleifen können nicht vektorisiert werden (Kontrollflussabhängigkeiten)
  - Lösung: Bedingt ausgeführte Anweisungen → Umwandlung von Kontrollflussabhängigkeiten in Datenabhängigkeiten
  - Bedingt ausgeführte Anweisungen
    * Vektor-Maskierungssteuerung: Verwendet einen zusätzlichen, boolschen Vektor, um die Ausführung eines Vektorbefehls zu steuern

```
1 LV        V1,Ra        ; load vector A into V1
2 LV        V2,Rb        ; load vector B
3 L.D       F0,#0        ; load FP zero into F0
4 SNEVS.D   V1,F0        ; sets VM(i) to 1 if V1(i)!=F0
5 SUBV.D    V1,V1,V2     ; subtract under vector mask
6 CVM                    ; set the vector mask to all 1s
7 SV        Ra,V1        ; store the result in A
```

    * Vektor-Mask-Register: Jede ausgeführte Vektorinstruktion arbeitet nur auf den Vektorelementen, deren Einträge eine 1 haben

- **Dünn besetzte Matrizen**
  - Elemente eines Vektors werden in einer komprimierten Form im Speicher abgelegt
  - Indexvektoren zeigen die Elemente an, die nicht 0 sind

```
1 ; Berechnet die Summe der duenn besetzten Vektoren A und C.
2 ; Die Indexvektoren K und M zeigen jeweils die Elemente von A und C an,
3 ; die nicht 0 sind
4 do   100 i = 1,n
5 100  A(K(i)) = A(K(i)) + C(M(i))
```

[4] https://de.wikipedia.org/wiki/Speicherverschränkung

– SCATTER-GATHER-Operationen mit Index-Vektoren: Unterstützen den Transport zwischen gepackter (SCATTER) und normaler (GATHER) Darstellung dünn besetzter Matrizen. Probleme für vektorisierenden Compiler: Konservative Annahmen bzgl. Speicherreferenzen

```
1 LV      Vk,Rk        ; load K
2 LVI     Va,(Ra+Vk)   ; load A(K(I))
3 LV      Vm,Rm        ; load M
4 LVI     Vc,(Rc+Vm)   ; load C(M(I))
5 ADDV.D  Va,Va,Vc     ; add them
6 SVI     (Ra+Vk),Va   ; store A(K(I))
```

- Beispiel: Realisierung von C-Code in Assembler

```
1 unsigned char i;
2 unsigned char a[64], b[64], c[64];
3 for (i = 0; i < 64; i++) {
4     c[i] = a[i];
5     if (a[i] == 0xff) {
6         c[i] = b[i];
7     }
8 }
```

```
1  ; Initialisierung
2  MOV     R1, 64
3  MTC1    VLR, R1      ; Vektorlaenge = 64
4  LV      V1, Ra
5  LV      V2, Rb
6  MOV     R2, 0
7
8  ; Berechnungen
9  ADDVS   V3, V1, R2   ; c[]=a[]
10 MOV     R1, 0xff
11 SEQVS.I V1, R1       ; Vergleich mittels 'equals'
12                      ; Setze Maske gleich 1 bei true, anderenfalls 0
13 SUBV.I  V3, V3, V3   ; Setze Werte=0, die in der Maske gesetzt sind
14 ADDV.I  V3, V3, V2
15
16 ; Bereinigen und speichern
17 CVM                  ; Vektormaske loeschen, sonst werden nicht alle
18                      ; Werte zurueckgeschrieben
19 SV      Rc, V3
```

4.2.2 SIMD-Verarbeitung in Mikroprozessoren am Beispiel Intel MMX Technologie

- Erweiterung für die IA-32-Prozessorarchitektur, die es erlaubt, größere Datenmengen parallelisiert und somit schneller zu verarbeiten[5]
- MMX-Register, die auf FPU-Register abgebildet werden (keine neuen Register eingeführt)
- Verarbeitung: (Mehrfach unterteilbares) 64 Bit Datenformat als Basis. Zwei solche Quellen können über eine Funktion verknüpft werden
- Saturation Arithemtik: Algorithmen in der grafischen Datenverarbeitung vermeiden Über-/Unterlauf bei Addition und Subtraktion nicht-vorzeichenbehafteter Pixel. Übergang zum größten oder kleinsten darstellbaren Wert. Keine Überprüfung der Werte oder Ausnahmebehandlung

[5] https://de.wikipedia.org/wiki/Multi_Media_Extension

5 Appendix: Formelsammlung und Definitionen

5.1 Grundlagen

5.1.1 Entwurfsfragen

- Kosten des Dies: $cost_{die} = \frac{cost_{wafer}}{\text{Dies pro Wafer} \cdot yield_{die}}$
- Anzahl der Dies pro Wafer: $dpw = \frac{\pi \cdot \left(\frac{1}{2} \cdot d_{Wafer}\right)^2}{A_{die}} - \frac{\pi \cdot d_{Wafer}}{\sqrt{2 \cdot A_{die}}}$ (theoretisches Maximum abzüglich Verschnitt)
- Die Yield (Ausbeute): $yield_{die} = yield_{Wafer} \cdot \left(1 + \frac{\text{Defekte pro Flaecheneinheit} \cdot A_{die}}{\alpha}\right)^{-\alpha}$
- Gesamtkosten eines IC: $cost_{IC} = \frac{cost_{die} + cost_{die-test} + cost_{packaging}}{yield_{final}}$

5.1.2 Energieeffizienter Entwurf

- **Prozessorleistung**
  - $P \sim U^2 \cdot f$
- **Wahrscheinlichkeiten**
  - $\mathbb{P}_A(1) = 1 - \mathbb{P}_{\neg A}(1)$
  - $\mathbb{P}_A(1) = 1 - \mathbb{P}_A(0)$
  - $\mathbb{P}_{A \wedge B}(1) = \mathbb{P}_A(1) \cdot \mathbb{P}_B(1)$
  - $\mathbb{P}_{A \vee B}(1) = 1 - \mathbb{P}_{A \wedge B}(0) = 1 - \mathbb{P}_A(0) \cdot \mathbb{P}_B(0)$
- **Schaltungswahrscheinlichkeit**
  - $\mathbb{P}_{Ausgang}(1) = \mathbb{P}_{Eingang1}(0) \cdot \mathbb{P}_{Eingang2}(1) + \mathbb{P}_{Eingang1}(1) \cdot \mathbb{P}_{Eingang2}(0)$
- **CMOS**
  - Leistungsaufnahme: $P_{total} = P_{switching} + P_{shortcircuit} + P_{static} + P_{leakage}$
  - $P_{switching} = C_{eff} \cdot V_{dd}^2 \cdot f$ (wesentlicher Anteil am Leistungsverbrauch)
  - $P_{shortcircuit} = I_{mean} \cdot V_{dd}$
- **Durchsatz**
  - Maximaler Durchsatz: $D_{max} = \frac{1}{Bedienzeit} = \frac{1}{t_{Zugriff} + t_{\text{Übertragung}}}$
  - Übertragungszeit: $t_{\text{Übertragung}} = \frac{\text{Größe der Übertragung}}{Bandbreite}$

5.1.3 Bewertung der Leistungsfähigkeit eines Rechners

- Gesamtausführungzeit (Gesetz von Amdahl): $T(n) = T(1) \cdot a + T(n) \cdot \frac{1-a}{n}$, a: Anteil des Programms, der nur sequentiell ausgeführt werden kann
- Für $n \to \infty$: $S(n) = \frac{1}{a}$
- Beschleunigung: $S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \cdot \frac{1-a}{n} + T(1) \cdot a} = \frac{1}{\frac{1-a}{n} + 1}$
- Effiziens: $E(n) = \frac{\text{Beschleunigung}}{\text{Anzahl der Kerne}} = \frac{S(n)}{n}$, $E \leq 1$
- Parallelindex: $I(n) = \frac{\text{Anzahl an Operationen}}{\text{Parallelgesamtausfuehrungszeit}} = \frac{P(n)}{T(n)}$
- Auslastung: $U(n) = \frac{I(n)}{n}$
- Mehraufwand für die Parallelisierung: $R(n) = \frac{P(n)}{P(1)}$
- $MFLOPS = \frac{Anzahl\ ausgefuehrter\ Gleitkommainstruktionen}{10^6 \cdot Ausfuehrungszeit}$
- IC (instruction count): Anzahl der ausgeführten Befehle
- C (cyles): Anzahl der ausgeführten Zyklen
- Ausführungszeit: $T_{exe} = \frac{\text{Anzahl an Zyklen}}{Prozessortakt} = \frac{C}{f}$
- Clock cycles per instruction (CPI, mittlere Anzahl Taktzyklen pro Befehl): $CPI = \frac{\text{CPU time}}{\text{instruction count} \cdot \text{machine cycle time}} = \frac{T_{exe}}{IC \cdot TC}$

- Instructions per cycle (IPC): $IPC = \frac{1}{CPI}$
- Taktfrequenz: $f = \frac{IC \cdot IPC}{T}$, niedrigerer Takt bedeutet niedrigere Leistungsaufnahme, da $P \sim f$
- $SPEC_{ratio} = \frac{Referenzsystem}{Testsystem}$, quantifiziert die Geschwindigkeit eines Rechensystems
- $SPEC_{rate}$ quantifiziert den Durchsatz eines Rechensystems
- Gesetz von Little: Mittlere Anzahl an Aufträgen = Durchsatz · Antwortzeit

5.1.4 Zuverlässigkeit und Fehlertoleranz

- Allgemeine Formel zur Berechnung der Zuverlässigkeit: $\phi_m^n = \sum_{k=n}^{m} \binom{m}{k} \cdot \phi(K)^k \cdot \big(1 - \phi(K)\big)^{m-k}$
- Zuverlässigkeitsberechnung mit Mehrheitsentscheider V: $\phi_m^n = \phi(V) \cdot \sum_{k=n}^{m} \binom{m}{k} \cdot \phi(K)^k \cdot \big(1 - \phi(K)\big)^{m-k}$
- Punktverfügbarkeit: $V = \frac{\text{Mittlere Funktionszeit}}{\text{Mittlere Funktionszeit+Mittlere Reparaturzeit}} = \frac{MTTF}{MTTF+MTTR}$
- **Berechnen von λ mit gegebener mittlerer Lebensdauer**
  - Gegeben: Überlebenswahrscheinlichkeit $R(S,t)$ eines Gesamtsystems, Überlebenswahrscheinlichkeit $R(t)$ einer Komponente sowie mittlere Lebensdauer
  - Berechne $\int_0^\infty R(S,t)dt =$ mittlere Lebensdauer

5.2 Verbindungsstrukturen

- Übertragungszeit = Startzeit + Transferzeit: $T_{msg} = t_s + t_w$
- Bisektionslinie: Teilt das Netzwerk in zwei gleiche Hälften
- Bisektionsbandbreite: Maximale Anzahl vom Megabytes pro Sekunde, die das Netzwerk über die Bisektionslinie transportieren kann
- Diameter (Durchmesser): Maximale Distanz zweier Prozessoren oder Anzahl der Verbindugen, die durchlaufen werden müssen oder maximale Pfadlänge zwichen zwei Knoten
- Verbindungsgrad eines Knoten: Anzahl der Direktverbindungen zu anderen Knoten
- Mittlere Distanz zwischen zwei Knoten: Anzahl der Links auf dem kürzesten Pfad zwischen den beiden Knoten