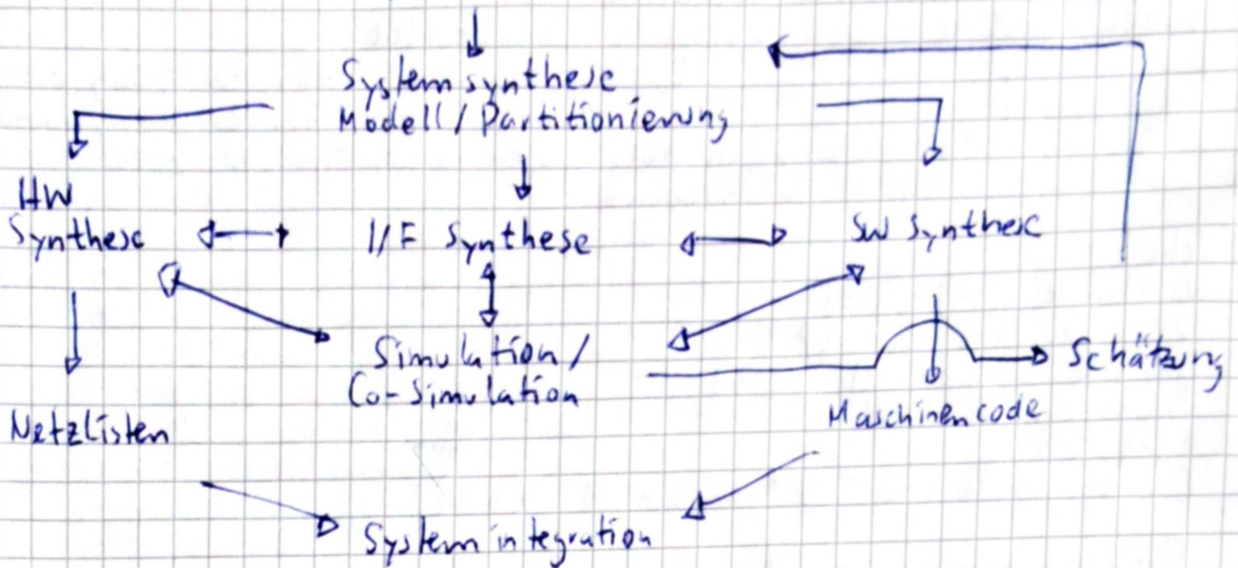
HSC



Nebenbedingungen können reine SW / reine HW Lösung ausschließen
=> Bedarf an spezialisierter HW mit Fähigkeit, Code auszuführen
ASIP
Nebenbedingungen / Constraints können sein:
Kosten, Chipfläche, Leistungsaufnahme, Time-to-Market, Echtzeitanforderungen

HW/SW Co-Design Spezifikationssprache
z.B. Language for instruction set architectures
+ Standard Vorgehen für HW / SW Zweig
+ Wiederverwendung von Komponenten
+ Leistungsfähige Werkzeuge für Übersetzung / Synthese vorhanden
- Übersetzung in andere Sprachen schwierig
- Verschiebung von Komponenten schwierig

Zielarchitekturen
Einfacher Prozessor: Steuerwerk, Rechenwerk, Speicherwerk, E/A-Werk
Bedingungsvektor: Flags wie Negative, Zero, overflow, ...
Von-Neumann Architektur: gemeinsamer Speicher für Daten & Code
Harvard-Architektur: getrennter Speicher für Code & Daten

Flynn'sche Taxonomie
SISD, MISD, SIMD, MIMD
CISC: Memory-to-memory Architektur
load / store sind eingebaute Instruktionen
=> kleinerer Code, komplexere HW
RISC: load-store Architektur
Benötigt mehr Transistoren für Speicherregister
allerdings weniger für die Instruktionen
pc = program counter (manchmal instruction pointer)
ir = instruction register
or = operant register
ac = accumulator
Pipeline: 5 Stufen, jede Instruktion muss jede Stufe durchlaufen
=> Latenz von 5 Takten
allerdings beendet dann jeden Takt eine Instruktion die Ausführung
=> erhöhter Durchsatz

Superskalare Pipelines:
dynamische Zuordnung der Befehle an mehrere Pipelines
out-of-order: Ausführungsreihenfolge ≠ Befehlsreihenfolge = Reihenfolge des Gültigwerdens der Ergebnisse
nutzen zeitliche - und räumliche Parallelität
↳ steigerbar durch mehr Ausführungseinheiten und hohe dynamische Zuweisungsbandbreite
↳ steigerbar durch mehr Stufen in der Pipeline oder höhere Taktfrequenz

Caches
Wortbreite (M) [bit]
Words per Line (wpl)
Line size = Wpl · M [bit]
Assoziativität (N)
Sets (S)
Cache-größe = S * N * Wpl * M (bit)

Ein Blockrahmen enthält mehrere Informationen:
Teile der ursprünglichen Hauptspeicheradresse
Die Nutzdaten selber
Flags: dirty: CPU hat lokales Datum modifiziert
invalid: lokale Kopie veraltet, bzw. inkonsistent mit RAM
Schreibzugriff / Cache-Strategie
Cache-Hit
Write-Back: Änderungen werden zuerst in den Cache geschrieben, Setzen vom Dirty-Bit
Schreiben in den RAM erst bei Verdrängung der dirty-Cache-Zeile (⇒ Konsistenzprobleme bei Multi-Core Systemen)
Write-Through: Schreiben in sowohl Cache, als auch RAM (⇒ hohe Speicherbus-Belastung)
Cache Miss
Write-Around: Ziel ist nicht in Cache, Daten werden nur in den Hauptspeicher geschrieben
Write-Allocate: Zuerst Kopie vom Datum aus dem RAM in den Cache holen, und dann entweder Write-Back oder Write-Through anwenden

Capacity-Miss ⇒ Cache vergrößern
Conflict-Miss ⇒ erhöhen der Assoziativität
Compulsory-Miss ⇒ Prefetcher, spekulatives Laden

Direct Mapped: RAM-Adresse mod Cache-Größe ⇒ Cache Zeile
einfach, schnell, allerdings oft ineffizient, conflict misses
Vollassoziativ: Jeder RAM-Inhalt kann überall stehen
⇒ gute hit-Rate, komplexe HW, hoher Overhead beim Lookup, alle Zeilen müssen geprüft werden
n-Way-associative: Speicheradresse wird auf ein Set gemapped, dort kann das Datum dann an beliebiger Stelle stehen

µP: kleiner, älterer, ausreichend starker Kern
bewährt, gut bekanntes Laufzeitverhalten
energieeffizient, kein Cache, keine virtuelle Speicherverwaltung
oft mit AD / DA-Wandler, Watchdog, Zähler-Zeitgeber, Erweiterungsbus, Interrupt-Handler, Echtzeitkanäle, ROM, RAM, EEPROM, Flash, .. PWM
Schneller Kontextwechsel durch Pointer-operation: Register ≙ RAM
oft eingesetzt bei kontrollflussdominantem Code / genutzt zur Steuerung

DSPs

bei Daten-dominantem Code & gut vorhersagbaren Sprungzielen
vorwiegend MAC-Operationen
Signalfilterung: Audio / Video / AD-Wandlung / DA-Wandlung
Spezialisiert für digitale Signalverarbeitung
oftmals hohe Parallelisierung möglich
Harvard-Architektur mit mehreren Datenbussen für parallelen Operandenzugriff
Hardware Multiplizierer für Gleitkomma
Zero-Overhead Schleifen: spezieller Zähler bei bekannter # Iterationen
Hardware lädt Start- und Zieladresse der Schleife + Zähler
je nach Zähler (= 0?) wird der Programmzähler gesetzt
Adressgeneratoren => Einsparung der Adressrechnung
circular-addressing für Filter
bitreverse-Adressierung für FFT

```
LDF 0, R0              // load Floats
LDF 0, R1
RPTS N-1               // Repeat N-1 often
MPYF3 *(AR0)++, *(AR1)++, R0    // multiply
|| ADDF3 R0, R1, R1             // and accumulate
ADDF3 R0, R1, R1                // last sum
```

Oft nur wenige MACs verfügbar
Heterogene Datenregister
Echtzeitfähigkeit
Im Allgemeinen effizienter als Implementierung in GPP => wichtig bei portablen Geräten
Blocked-Floating Point: einzelner Exponent für Gruppe von Zahlen
Sättigungsarithmetik statt Überlauf
Programmentwicklung: häufig C, falls keine Compiler für spezielle Konstrukte / Funktionen bereitstehen: Assembler per Hand
C Bibliotheken

GPUs

verstecken lange Speicherzugriffszeiten (=> Stalls) durch Ausführung anderer Fragmente, die ablaufbereit sind
Je mehr Kontexte vorhanden sind, desto besser ist die Möglichkeit, Latenzen zu verstecken

NoC

Router on chip
Paketorientierte Kommunikation statt Bus (Best-Effort)
Virtuelle Kanäle reduzieren Blockaden
Zeitmultiplexing vom physischen Link auf virtuelle Kanäle

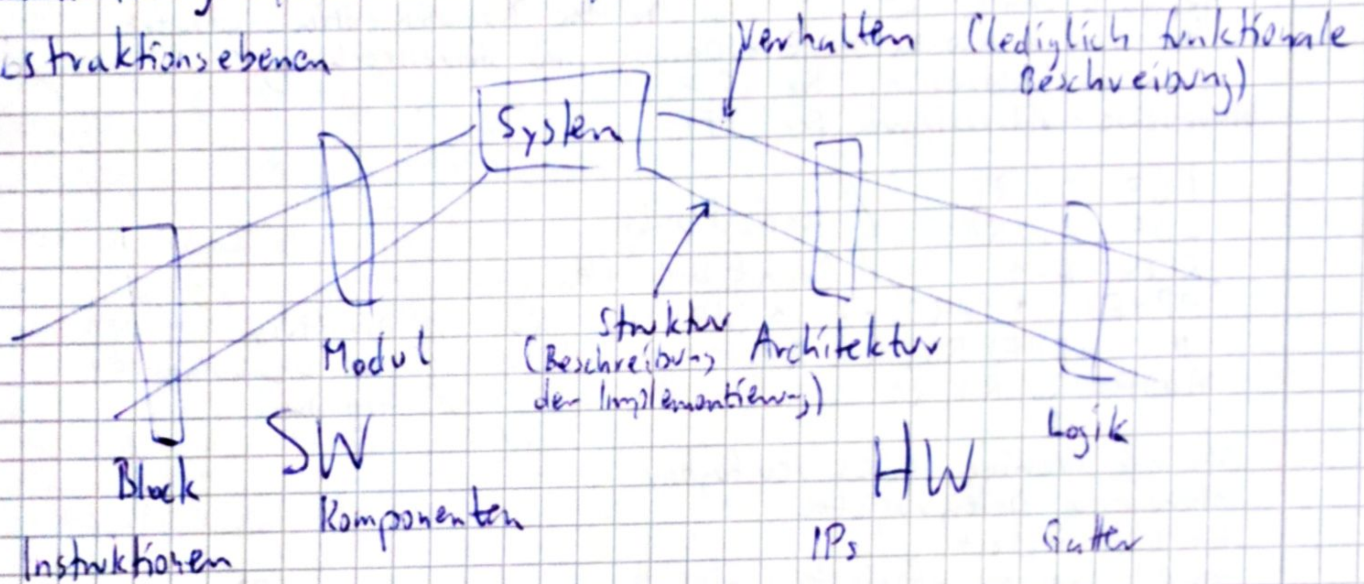
Leitungsorientiert:
Guaranteed Service
Overhead für Leitungsaufbau
Dann aber effiziente und garantierte Übertragung der Daten
sinnvoll bei großen Datenmengen

I/O Kohärenz: I/O-Geräte schnüffeln in CPU Cache
Volle Kohärenz: Auch die CPU schnüffelt in Geräte Cache

ASIP
Spezialisierung der ISA für eine Klasse von Anwendungen
" der Funktionseinheiten
" der Speicherarchitektur
" der Datentypen
HW-Unterstützung von Schleifen

Soft-IP-core-Blocks: HDL
Firm: HDL & Netzlisten (+ evtl. Floorplanning)
technologie-spezifische Syntheseinformationen } keine Laufzeitvorhersage möglich
abgebildet auf Bibliothek
Hard: fertig synthetisiertes Layout / Maskenbeschreibung } vorhersagbar

Abstraktionsebenen



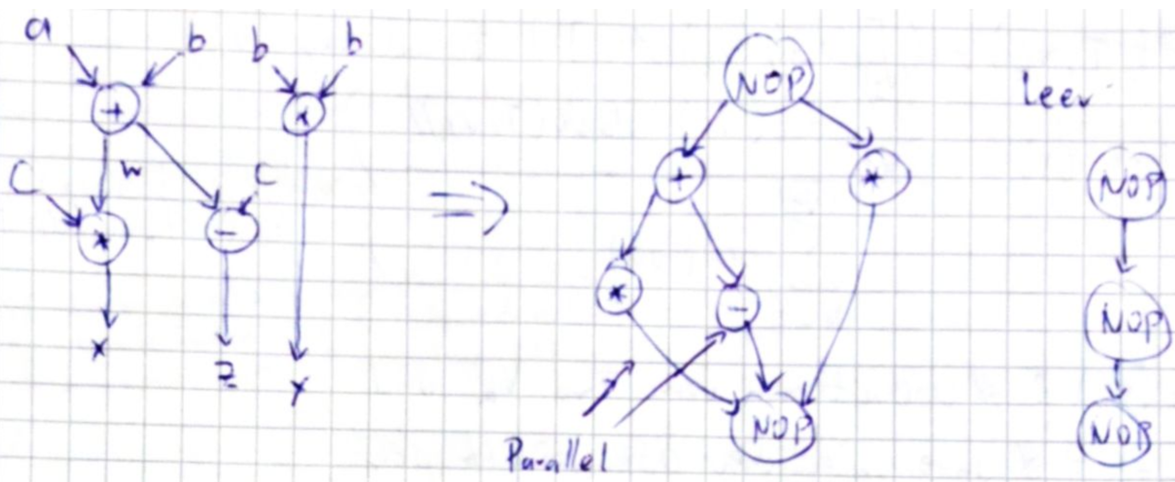
Syntheseaufgaben: Allokation (Auswahl von Komponenten)
Bindung (Zuweisung von Funktionen zu Komponenten)
Ablaufplanung (Festlegung der Ausführungsreihenfolge)

kürzere Entwurfszeiten
reduzierte Fehlerrate (beweisbar korrekte Ergebnisse von CAD Tools)
Exploration des Entwurfsraums
Ob die Constraints eingehalten werden (können) ist gerade zu Beginn des Designs (vor Fertigung ...) wichtig und kann kostspielige Fehler aufzeigen / vermeiden

Taskgraph: gerichtete Kantenmenge: Datenabhängigkeiten / Steuerabhängigkeiten
Knoten: Basisblock
Allokation: Welche HW wird zur Verfügung gestellt?
Bindung: Abbildung der Tasks auf die HW
Scheduling: Festlegung der Ausführungszeiten der Tasks in Abhängigkeit von ihrer Bindung & deren Datenabhängigkeiten

Pareto-Punkte: werden von keinem anderen Entwurfspunkt des Entwurfsraums in allen Eigenschaften unterboten

DFG: - keine explizite Ausführungsreihenfolge der Operationen
- Kommutativität & Assoziativität => verschiedene DFGs
- keine Modellierung von Kontrollstrukturen
CFG: - Mögliche parallele Ausführung nicht darstellbar
Hierarchische Sequenzgraphen
Einheiten modellieren Datenfluss
Hierarchie den Kontrollfluss
Alle Routinen beginnen ~~mit~~ und enden mit einem NOP-Knoten



Schätzung der Entwurfsqualität

Analyse, Simulation, Emulation, Rapid Prototyping (mit FPGAs)
nach Implementierung: Messung
Exaktheit, Treue & Zeitaufwand abhängig von der Granularität des Modells

Metriken & Exaktheit der Schätzung

E(D) ≙ abgeschätzte Metrik von Implementierung D
M(D) ≙ Messung der Metrik von Implementierung D
Exaktheit $A = 1 - \frac{|E(D) - M(D)|}{|M(D)|}$

Treue $F = \frac{2}{n \cdot (n-1)} \cdot \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mu_{i,j}$

Paarweise Korrektheit von Schätzung & Messung

Hardware Performanz
Taktperiode T
Latenz L
Ausführungszeit $T_{ex} = L \cdot T$
Durchsatz $R = 1/T_{ex}$ (≙ zeitliche Häufigkeit des Berechnungsergebnisses)
Tradeoff: viel HW ⇒ teuer, aber schnell
weniger HW ⇒ mehr Taktzyklen (⇒ höhere Latenz) aber günstig
Multi-cycle-operations als Trade-off "Treffen in der Mitte"
Pipelining mit P gleich langen Stufen:
$R = P/T_{ex}$
$\left.\begin{array}{l} T = 80ns \\ L = 5 \end{array}\right\} \Rightarrow T_{ex} = 400ns \Rightarrow R = 5/400ns = 0{,}0125$

Schätzung der Taktperiode
maximale Operatorenverzögerung ⇒ deutliche Unterbelastung schnellerer Operationen
minimiere Slack: suche im Intervall $[T_{min}, T_{max}]$ nach der Taktperiode mit maximaler Taktauslastung (minimaler Slack). Scheduling oft als Nachfolgeschritt zur Bestimmung der Gesamtausführungszeit T_{exec}.
ILP-Suche
Modellierung eines Latenzminimierungsproblems als ILP für diskrete Werte der Taktperiode zur Minimierung von T_{exec}
Taktschlupf: Anteil einer Taktperiode, in der eine HW Einheit nicht benutzt wird

$$slack(T, v_k) = (\lceil delay(v_k) / T \rceil) \cdot T - delay(v_k)$$

$$avgslack(T) = \frac{\sum_{k=1}^{|V_T|} (occ(v_k) \cdot slack(T, v_k))}{\sum_{k=1}^{|V_T|} occ(v_k)}$$ } mittlerer Schlupf in Taktperiode T

$occ(v_k) \hat{=}$ # Operationen vom Typ v_k und

$|V_T| \hat{=}$ # unterschiedlicher Operationstypen

geringerer mittlerer Schlupf => geringere Ausführungszeit für eine feste Anzahl Resourcen

Taktauslastung $util(T) = 1 - \frac{avgslack(T)}{T}$

Bei FSMs ist der Kontrollpfad kritischer als der Datenpfad:
Abhilfe: Pipelining des Kontrollpfades, Einführen von control register & state-register

Kostenmetriken
- Silizium fläche
- # Transistoren
- # Logic blocks (FPGAs)
- Packaging
- # pins
- Kontrolllogik
- Typ, # Controller

Software-Performance

$T_{exec} = I_c * CPI * T_{clock} = (I_c * CPI) / f$

- $I_c \hat{=}$ Instruction count
- $CPI \hat{=}$ Cycles per Instruction
- $T_{clock} \hat{=}$ Zeit der Taktperiode
- $f \hat{=}$ Frequenz, $1/T_{clock}$

MIPS-Rate $= I_c / (T_{exec} \cdot 10^6) = f / (CPI \cdot 10^6)$
Maß für Komplexität eines Algorithmus bzw dessen Umsetzung
Nur T_{exec} ist wirklich aussagekräftig

WCET kann nicht durch Profiling bestimmt werden
-> Programmanalyse notwendig
=> Prozessormodell notwendig
Welche Instruktionsreihenfolge führt im Worst Case zu welcher Ausführungszeit? Problem wächst exponentiell mit der Code Länge

Annahmen: WCET geschätzt ≥ WCET
Kein OS (Preemption), keine Interrupts, keine Rekursion, keine Pointeroperationen, beschränkte Schleifen, ein skalarer Prozessor

Aufstellen von Flussgleichungen: # Eintritte in einen Knoten = # Austritte von dort

Ein Programm besteht aus N Grundblöcken
jeder Grundblock B_i hat eine WCET c_i und wird x_i mal durchlaufen

$$\Rightarrow WCET = \sum_{i=1}^{N} c_i \cdot x_i$$

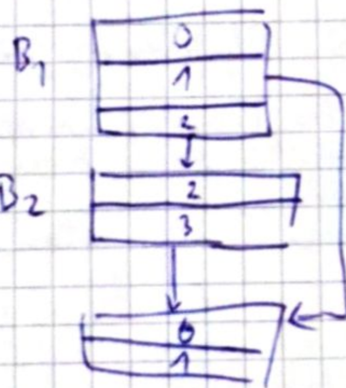
c_i kann geschätzt werden, da die Instruktionen in B_i bekannt sind
x_i herleitbar durch strukturelle Constraints und funktionale Constraints
Anschauung am Assembly code, nicht am Sourcecode

Modellierung von Caches
Warum überhaupt Caches? 90% der Ausführungszeit stammt oft aus ~10% des Codes. Dies zu cachen ist sinnvoll
Jeder Grundblock B_i besteht aus n_i - l Blöcken, d.h. jeder $B_{i,j}$ hat das gleiche x_i
Jede Ausführung eines l-Blocks führt entweder zu einem Hit oder Miss

$$x_i = x_{i,j} = x_{i,j}^{HIT} + x_{i,j}^{MISS} \quad , j = 1 ... n_i$$

$$\Rightarrow WCET = \sum_{i=1}^{N} \sum_{j=1}^{n_i} \left(c_{i,j}^{HIT} \cdot x_{i,j}^{HIT} + c_{i,j}^{MISS} \cdot x_{i,j}^{MISS} \right)$$

CFG



Konflikt bei 1,1 und 3,1
1,2 und 3,2

sonst keine, höchstens einen compulsory Miss

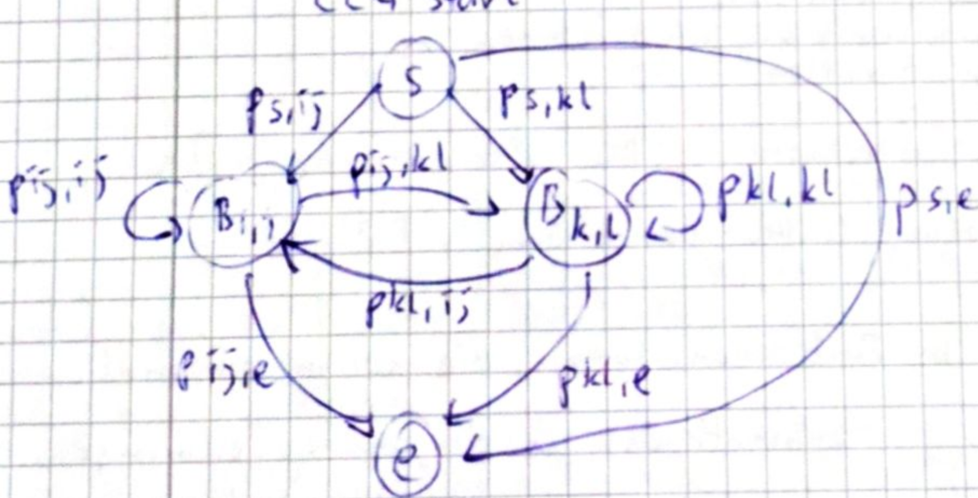
$\Rightarrow$ bspw. $x_{2,2}^{miss} \leq 1$

1,3 und 2,1 sind nicht im Konflikt, 1,3 lädt automatisch 2,1 mit in den Cache

$\Rightarrow$ bspw. $x_{1,3}^{miss} + x_{2,1}^{miss} \leq 1$

1,1 und 3,1 sind im Konflikt
ILP-Beschränkungen aus Konfliktgraph ablesbar

CCG Start



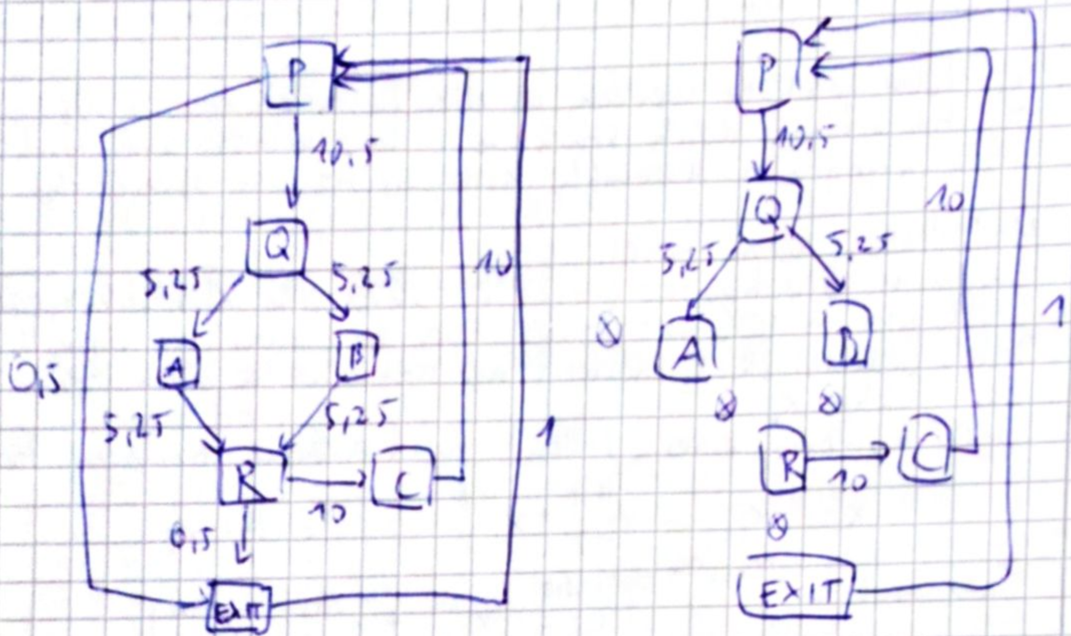
für alle Paare in Konflikt

$p_{x,y} \hat{=}$ Häufigkeit des Übergangs von x nach y

Ende CCG

$$\Rightarrow x_i = x_{ij} = p_{s,ij} + p_{kl,ij} + p_{ij,ij} = p_{ij,kl} + p_{ij,e} + p_{kl,e}$$

Profiling: zählen, wie oft welcher Block durchlaufen wurde
Tracing: protokollieren der Ausführungsreihenfolge
→ erlaubt das Berechnen der Ausführungskosten von einem Programm



Maximal aufspannender Baum

⊗: Platzierung der Zähler

Tracing
Witness-Menge zur Rekonstruktion der Programmausführungsreihenfolge
Lokalisierung von Performance-Bottlenecks möglich, falls Tracing zeigt, dass oft der Kontext gewechselt werden muss
Effiziente Methode: nur bei Basisblöcken einen Trace setzen, die Ziel einer Kontrollverzweigung sind

Teilprobleme der HW/SW Partitionierung
1. Festlegung der Abstraktionsebene der Spezifikation
2. Wahl der Granularität
3. Allokation der Systemkomponenten
4. Auswahl möglicher & realistischer Metriken & Schätzungen
5. Entwurf einer geeigneten Zielfunktion
6. Auswahl geeigneter Partitionierungsalgorithmen
7. Entwurfsablauf & Designintegration

Systemebene: Hierarchische Taskgraphen (CDFG, DFG, Sequenzgraphen, Petrinetze
Operationsebene/Strukturebene: Gatter-Beschreibung, IP Blöcke

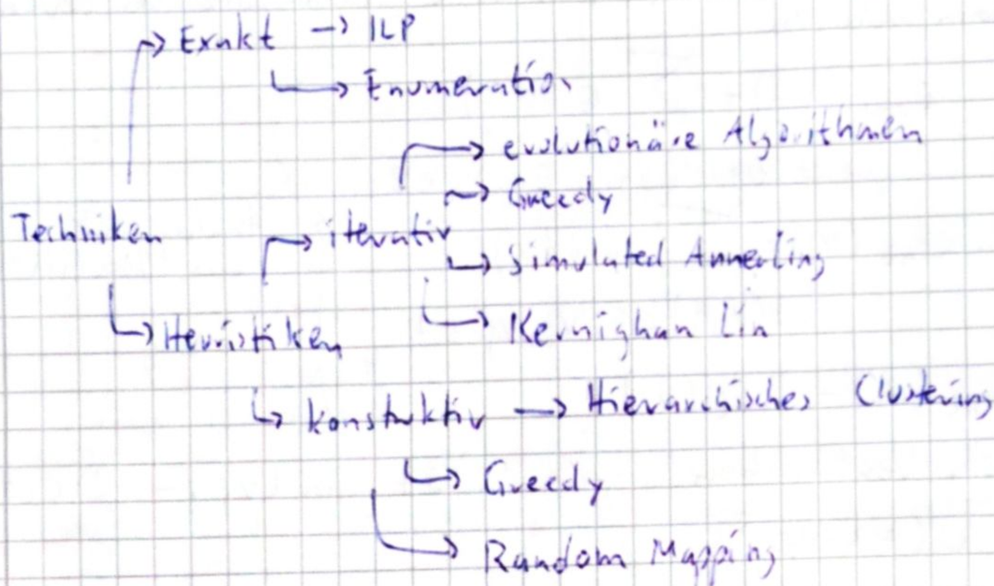
Komplexitätsbetrachtung
verteile n Objekte auf m Partitionen, sodass die Kosten minimal sind: NP-vollständig.
naiv: $\frac{1}{k!} \cdot \frac{n!}{(p!)^k}$ Kombinationen. k disjunkte Teilmengen mit Größe p
n = # Knoten

Beispiel für eine Kostenfunktion:

$$f(C,L,P) = k_1 \cdot h_c(C,\bar{C}) + k_2 \cdot h_L(L,\bar{L}) + k_3 \cdot h_P(P,\bar{P})$$

k_i ≙ Gewichtung h_x ≙ (Abweichung) von Attribut x zum Constraint
Überschreitung

Klassifikation von Partitionierungsalgorithmen



konstruktive Verfahren generieren eine Lösung (aus dem "Nichts")
iterative Verfahren gehen von einer Startlösung aus und verbessern diese so lange, bis ein Abbruchkriterium erreicht wird.
Heterogene Verfahren sind eine Kombination der beiden ?

Hierarchisches Clustering
Nutzt Nähefunktion, um Gruppierung von Objekten möglichst gewinnbringend durchzuführen
Nähefunktion oft schwer zu definieren
Doppelkanten bei Zusammenführen entfernen, die „bessere" überlebt
Kanten von Gruppe-zu-Gruppe entfernen
Multistage-Variante
Nutze in jeder Iteration eine andere Metrik, um beispielsweise Balance zu halten in dem Partitionsbaum
Schneiden problematisch wenn der Baum nicht ausbalanciert ist
=> Kommunikations-Bottleneck
Metriken:
Datenabhängigkeiten => Reduktion des Kommunikationsaufwandes über Partitionen hinweg
Sharing von Operatoren => Weniger HW nötig, Scheduler nötig
Kontrollfluss => weniger Kontrollflussübergaben über Partitionen hinweg
nutze Verzweigungswahrscheinlichkeit als Nähefunktion
Bedingte Eignung für Partition eines gesamten Systems
Skaliert gut bei großer Knotenanzahl
Erkennt keine Globalen Extrema
Dient als Startlösung für iterative Verfahren
Reduziert die Komplexität für große Knotenmengen

Tabu-Search
Lokale Evaluierung der umliegenden Designentscheidungen
wähle lokale beste Lösung, die nicht zu einer betrachteten Lösung führt
Tabu-Liste speichert Übergänge verhindert Zyklen & Umkehrung
Variation der Lösung konkret: veränderung der lösungsbeschreibenden Parameter => in diesem Fall: verschieben eines Knotens in eine andere Partition. Verbot der Umkehr => Verbot, Variation der Parameter Rückgängig zu machen => kurz- oder mittelfristig unmöglich eine Partition erneut zu besuchen

Kernighan-Lin Algorithmus
Anwendung bei Bipartitionen
Min-Cut Heuristik. Vertausche paarweise Knoten, die bei Vertauschung den größten Gain bringen

Externe Kosten $c_{ext}(v_i) = \sum_{e=(v_i,v_j)\in E,\ v_j \in B} c(e_{ij})$

Interne Kosten: $c_{int}(v_i) = \sum_{e=(v_i,v_j)\in E,\ v_i, v_j \in A} c(e_{ij})$

Analog für Kosten aus Partition B

$gain(v_i) = c_{ext}(v_i) - c_{int}(v_i)$

Bei gegenseitigem Vertauschen: $gain(v_i) + gain(v_j) - 2 \cdot c(e_{ij})$

Komplexität: $O(\#\text{Knoten}^3)$

Algo läuft solange, bis alle Knoten ein mal vertauscht wurden => auch schlechtere Lösungen werden akzeptiert. Am Ende werden alle Vertauschungen ein mal durchgeführt, bis die best gemessene Aufteilung erreicht ist. Dann wird terminiert.

Fiduccia Mattheyses Algorithmus
Betrachtet nicht nur balancierte Lösungen
Arbeitet auf Hyperkanten
Bewegt auch einzelne Knoten
Komplexität $O(\#\text{Hyperkanten}) \neq O(n^3)$
Cutset: Menge der Hyperkanten, die geschnitten werden
$gain(i) = \#\text{alter Cut} - \#\text{neuer Cut}$
Ausgewählte Zelle zur Verschiebung, die nicht das Balance-Kriterium verletzt, nennt sich Basiszelle. Sie muss den gain maximieren
Kritisches Netz ≙ Netz, das eine Zelle enthält, die bei Verschiebung den # cut ändert
=> nur kritische Netze tragen zur Änderung des Cutsets bei
Balance-Kriterium:

$$\frac{|A|}{|A|+|B|} \approx 0{,}3 - 0{,}7$$

Alternativ: Ratio-Cut

$$\text{minimiere } R_{V_A V_B} = \frac{cut(V_A, V_B)}{|V_A| \cdot |V_B|}$$

Gibt es mehrere Partitionierungen mit gleichem Gain $g(i)$, so liefert eine besser ausbalancierte Lösungen einen größeren Nenner
=> zu bevorzugen

1. Berechne Gain für jede Zelle
2. Finde Basiszelle mit maximalem gain
3. Markiere Basiszelle & aktualisiere gain der betroffenen kritischen Netze
4. Wenn es noch unmarkierte Zellen gibt, merke Reihenfolge und gehe zu 3.
5. Suche Sequenz der Verschiebung, die den gain maximieren. Verfahre mit Ratio-Cut. Falls gain ≤ 0 => terminiere
6. Führe Verschiebungen tatsächlich durch, hebe Markierungen auf, gehe zu 1

Simulated Annealing

```
S := start_state (S)
T := start_temp (S)
repeat
   while not stationary_state (S,T) do
      Snew := modify_state (S,T)
      if rand(0,1) ≤ e^(-(c(Snew) - c(S)) / T) then
         S := Snew
      end
      T := update (T)
   until convergence
end
```

$$\text{if } rand(0,1) \leq e^{\frac{-(c(S_{new}) - c(S))}{T}}$$

Laufzeit von convergence, update (T) und stationary_state (S,T) abhängig
- exponentiell, linear, polynomisch möglich
- für $t \to +\infty$: globales Optimum findbar
- generell: je länger, desto besser die Lösung

Gleichgewicht angenommen nach bestimmter # Iterationen oder wenn sich keine Besserung mehr ergibt

Genetische Algorithmen

Population: Menge von Lösungen
- # Anzahl Nachkommen, Kreuzungswahrscheinlichkeit, Mutationswahrscheinlichkeit
- Prinzip: Survival of the Fittest
  - Wähle die besten nach Qualitätsmaß aus $q(Ind)$
- Skalierung notwendig, damit nicht die besten die gesamte Population übernehmen

$$q_s(Ind) = \frac{c \cdot q_{avg} - q_{avg}}{q_{max} - q_{avg}} \cdot \left(q(Ind) - q_{avg} \right) + q_{avg}$$

Differenzmaß für die mittlere Qualität der Population, welches über einen Kontrollparameter c (≈ 2) beeinflusst werden kann

Selektion, Kreuzung, Mutation

Bei Mehrzieloptimierung: verringere Komplexität des Problems:
- Zielfunktion bildet auf niedrigere Dimension ab
- betrachte nur Pareto-Fronten
  - dominierende Punkte: Fitness = # dominierte Punkte
  - dominierte Punkte: Fitness = # dominierte insgesamt + Fitness der dominierenden Pu[nkte]
- Clustering: Fasse Pareto-Punkte entlang einer Front zusammen

HW-SW Partitionierung mit Greedy Algorithmen

```
P = {{}, O}    // HW only
Repeat
   Pold = P
   Foreach (Oi ∈ PHW)
      TryMove (P, Oi)
   End for
Until (P == Pold)
End
```

} Kann nicht aus lokalem Extremum ausbrechen