19.

Performance & Effizienz

Warum?

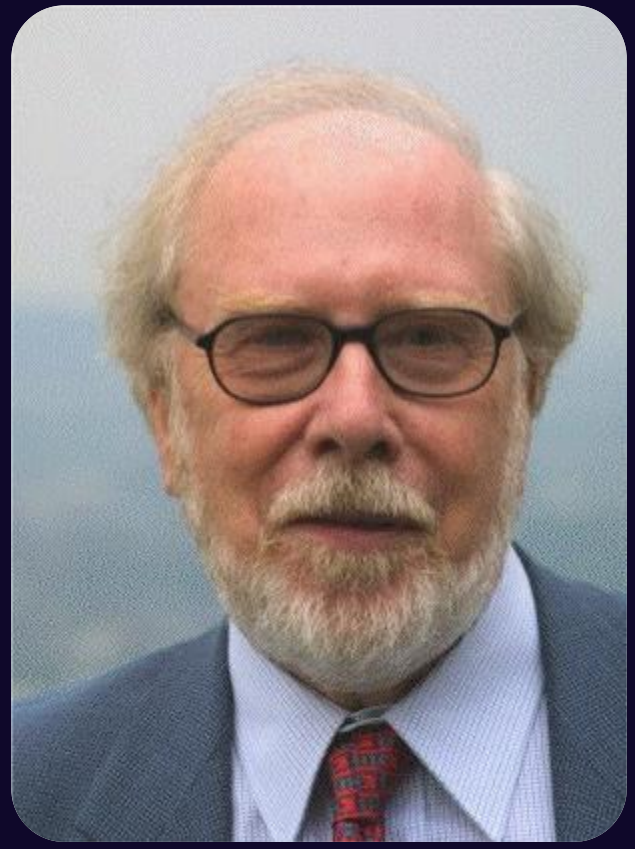
Warum *nicht*?

- Computer werden exponentiell schneller
- CPUs kosten weniger als Menschen
- „Flaschenhals ist sowieso IO“

> Warum?

“Software is getting slower more rapidly than hardware becomes faster.”

— Niklaus Wirth

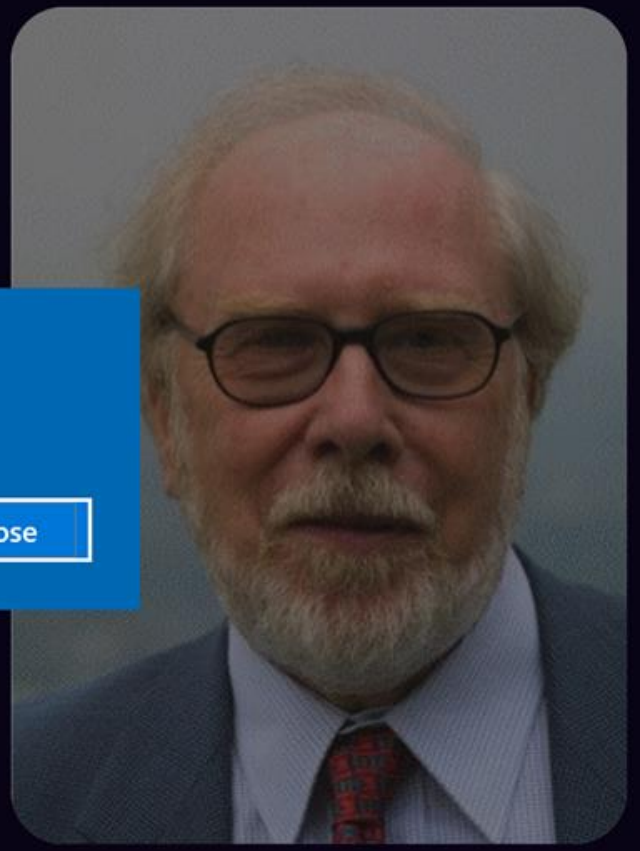


“Software is
than hardwa

Your battery is running low (8%)

You might want to plug in your PC.

Close



10:39
14.06.2016

- ≈ 8500€
- 10 Kerne
- 3,2 Ghz (3,7 Ghz Turbo)
- 22nm-Fertigung

$$\frac{Lichtgeschwindigkeit}{3{,}7\ Ghz} = 81mm$$

Struktur

(A) Begriffe & Vorgehen

(B) Die CPU

(C) Compiler & Programmiersprachen

A.

Begriffe & Vorgehen

Möglichst wenig Arbeit für Lösung

Möglichst viel Arbeit pro Zeit

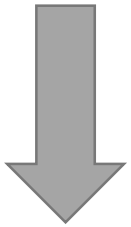
Effizienz

vs.

Leistung/Performance

(in Anlehnung an Chandler Carruth)

Algorithmenentwurf
(asymptotische Laufzeit, Effizienz)



Effizienz weiter verbessern
(konstante Faktoren, Aufwand verringern)

Leistung verbessern
(konstante Faktoren, „Lampen leuchten lassen“)

Makro-Optimierung

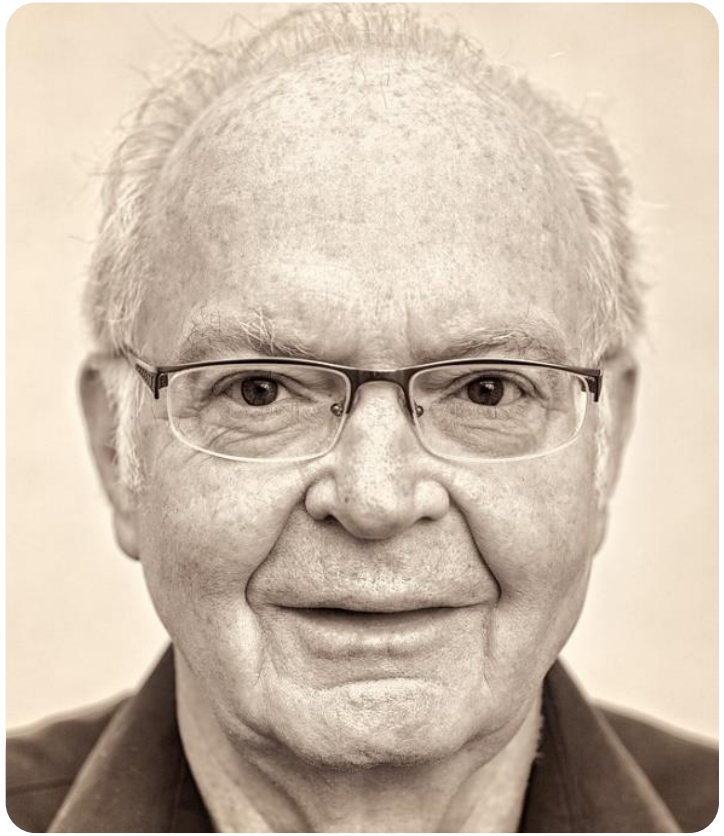
Zuerst!

Danach!

Mikro-Optimierung

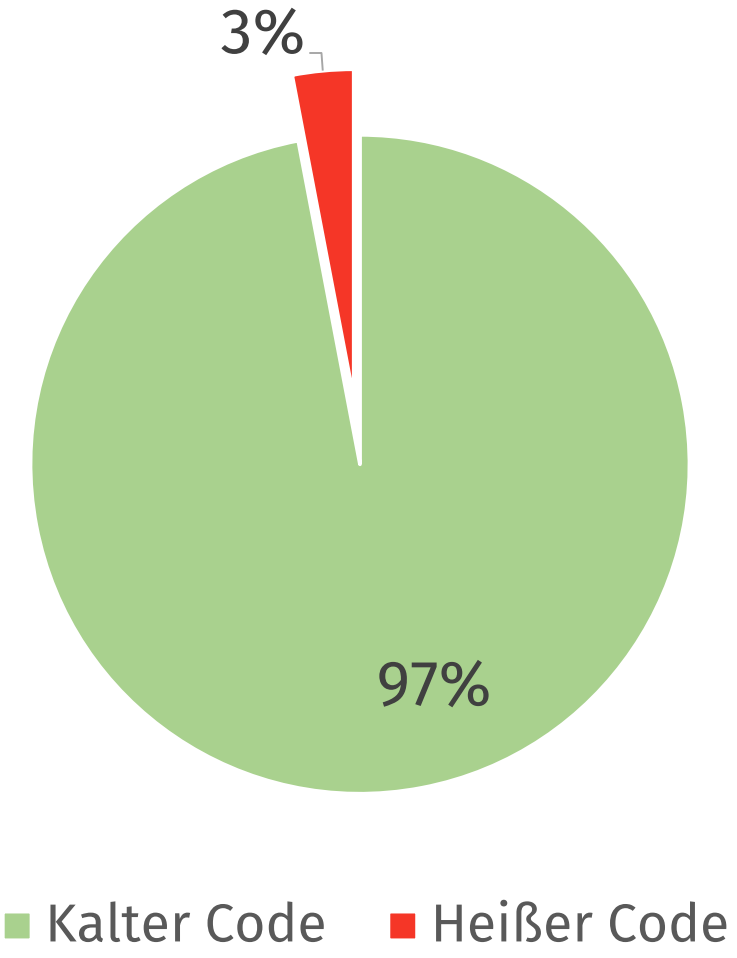
“Premature optimization is the root of all evil.”

— Donald Knuth [2]



Was/Wann optimieren?

- Nur ≈ 3% „kritisch“
- Geschwindigkeit für viele Programme irrelevant



(insb. Mikro-) **Optimierung zunächst ignorieren!**

Aber wie dann?

“You must *measure* everything!

We all have intuition. And intuition of programmers is as good as the intuition about the other gender. It’s always wrong.”

— Andrei Alexandrescu [3]

1. **Korrektheit** (algorithmische Komplexität im Hinterkopf)
2. Kritische Stellen *ausfindig machen*
3. Kritische Stellen *optimieren*

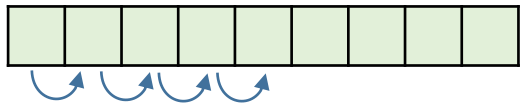
B.

Hallo CPU!

Erster Test!

Gegeben: Mehrere zufällige Zahlen
Aufgabe: Zahlen in unsere Datenstruktur einfügen
Bedingung: Datenstruktur muss immer sortiert bleiben

Array



Suche ≈ **n**
Einfügen ≈ **2n**

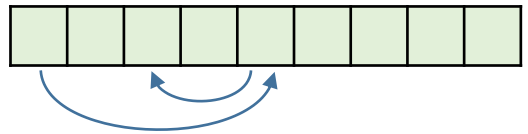
vs.

Linked List

Suche ≈ **n**
Einfügen ≈ **1**

vs.

Array mit bin. Suche



Suche ≈ **log n**
Einfügen ≈ **2n**

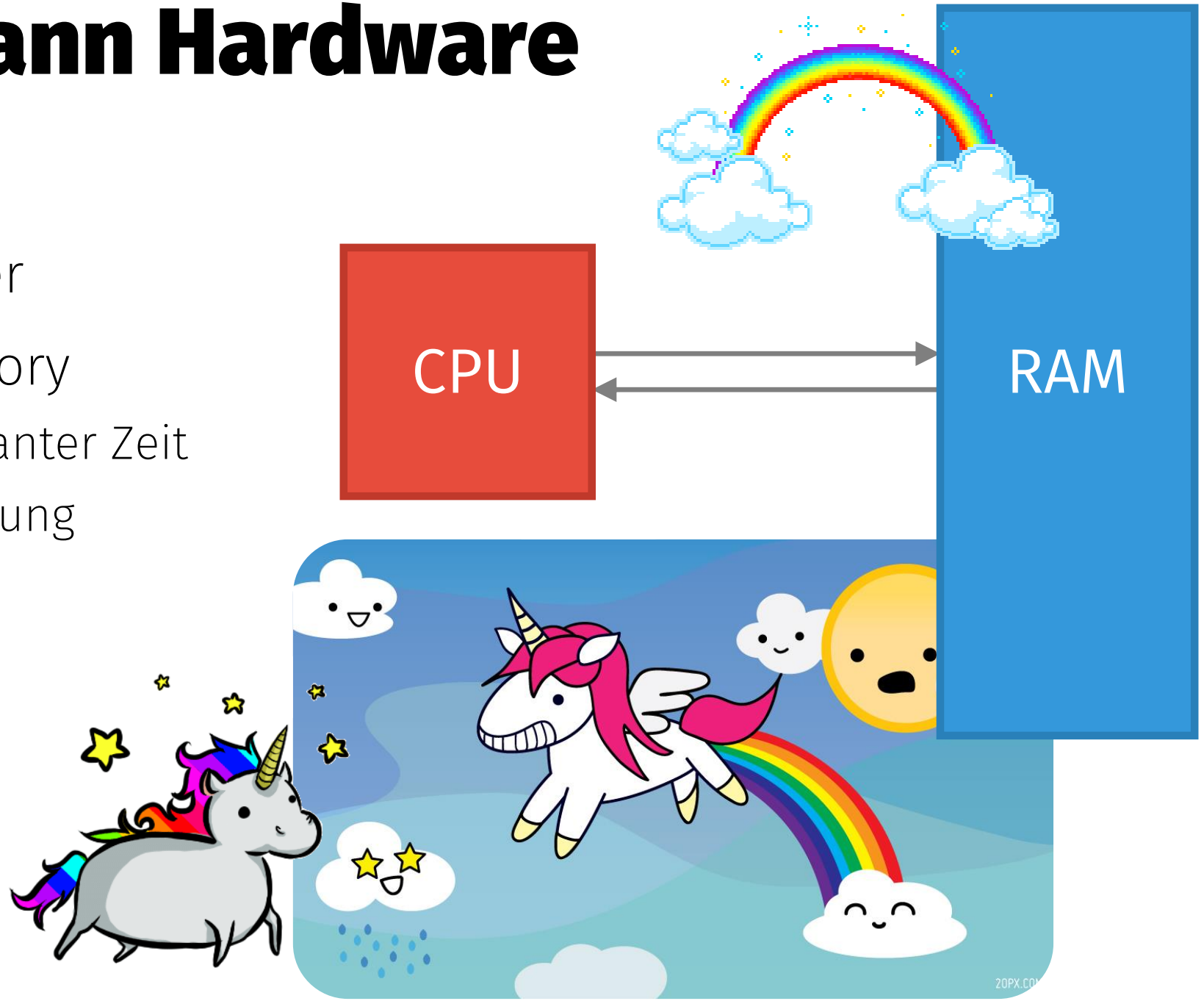
Warum anders in der Praxis?!

Von-Neumann Hardware

<u>Unsere Hardware:</u>

- Eine CPU, ein Speicher
- *Random* Access Memory
  - Jeder Zugriff in konstanter Zeit
  - Zugriff ohne Verzögerung
- Regenbögen
- Einhörner

Alles Quatsch!



Speicherhierarchie

	≈Größe	Zugriffsz	Vergleich
CPU-Register	500B	0.3 ns	≈ 5s (Buch von großem Schreibtisch nehmen)
L1-Cache	128 KiB	1 ns	≈ 15s (Schließfächer vor unserem Vorlesungsraum)
L2-Cache	1 MiB	7 ns	
L3-Cache	8 MiB	15 ns	
RAM	8 GiB	150 ns	≈ 35 min (Bibliothek am Neumarkt)
SSD	500 GB	150 µs	≈ 25 Tage (Venedig)
HDD	2 TB	10 ms	≈ 5 Jahre

Bringt nur was, wenn gleiche Adresse mehrmals gelesen wird?

Prefetcher

Wenn auf langsamen Speicher zugreifen

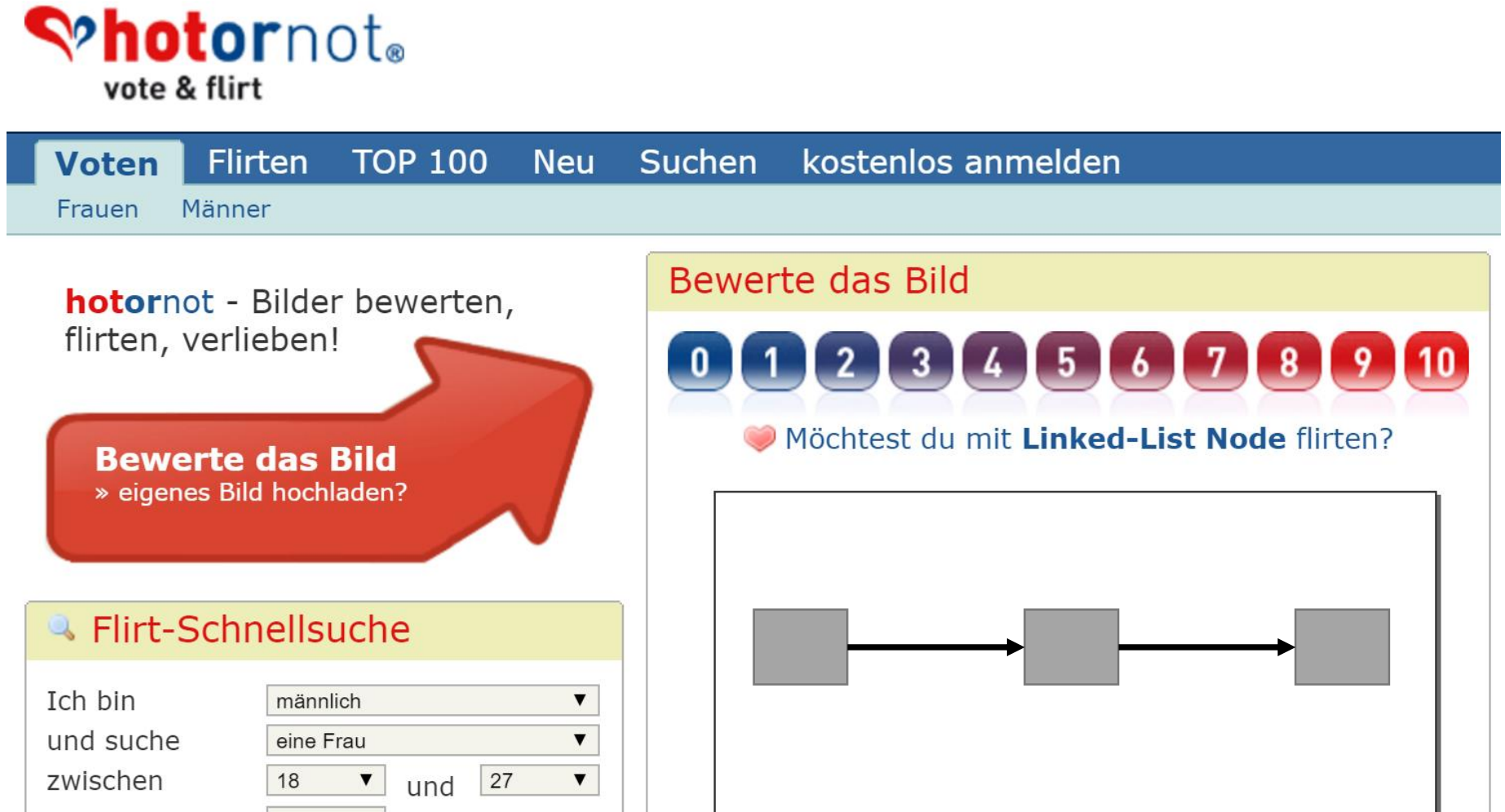
Mehr laden als gebraucht

- Immer 64 Byte laden (*Cache-Line*)
- Komplexere Techniken zum richtigen Prefetchen
  - Speicherzugriffsmusteranalyse
  - Basierend auf Branch-Prediction
- Linear durch Speicher: Quasi nie warten

Caching

Daten im Cache → „hot“

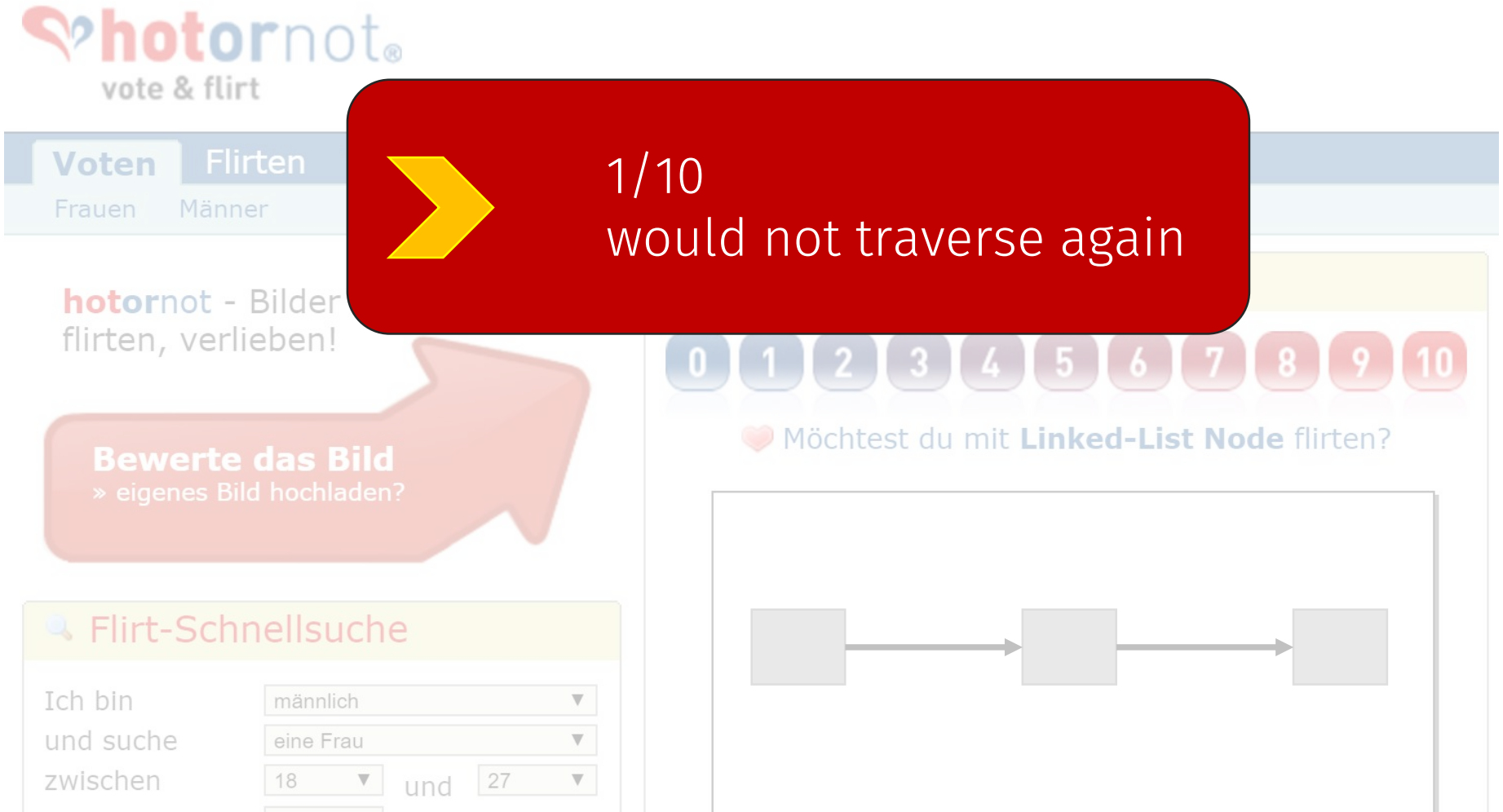
Daten nicht im Cache → „cold“



Caching

Daten im Cache → „hot“

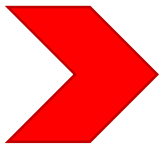
Daten nicht im Cache → „cold“



1/10
would not traverse again

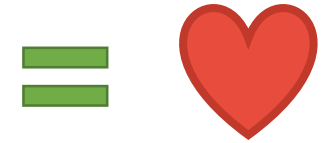
Probleme mit Linked List

- Jedes **`advance()`** ist ein *Cache-Miss*
- Nodes zufällig im Speicher verteilt
- Die „Vorteile“ werden fast nie genutzt



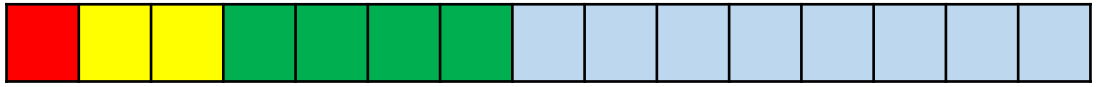
Finger weg von Linked List!

Java: `ArrayList`
C++: `vector`
Rust: `Vec`

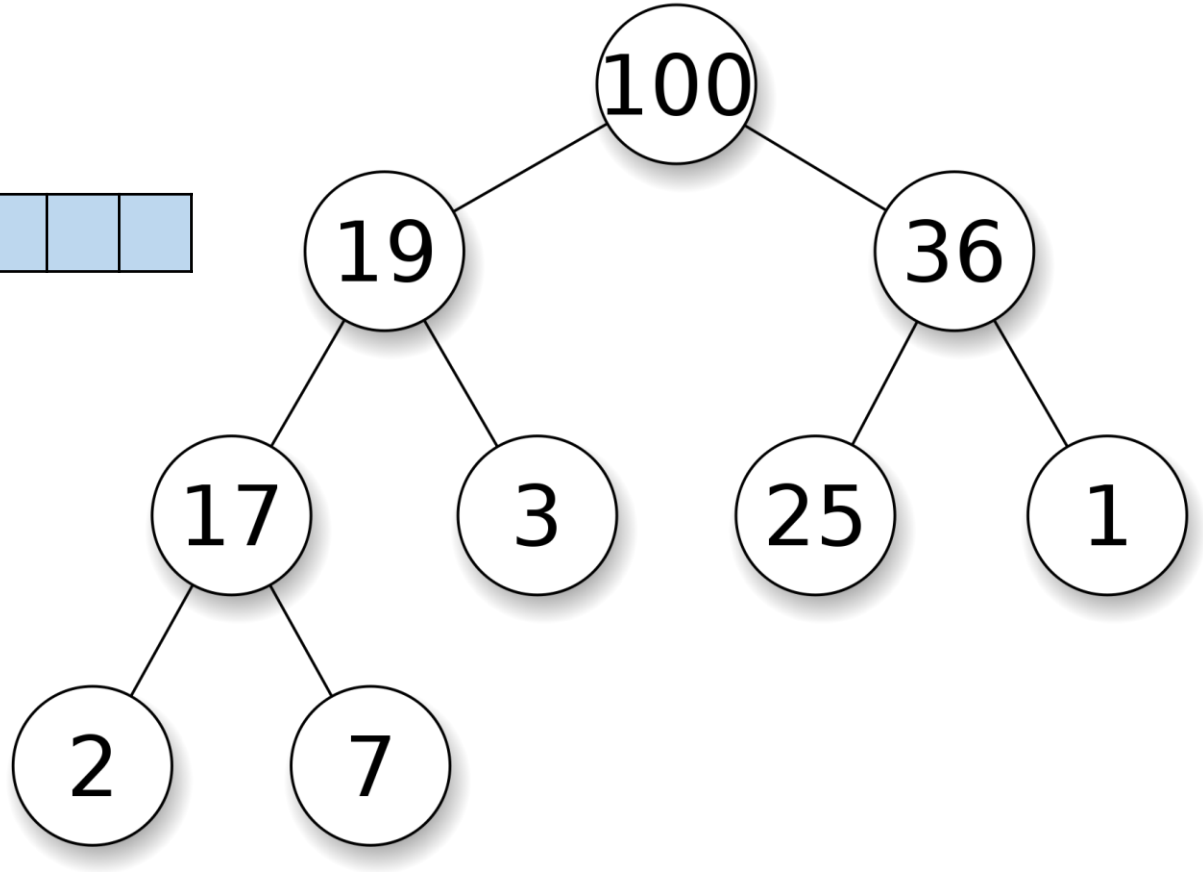
 `reserve()` 

InfoA: Binärer Heap

- In Array gespeichert

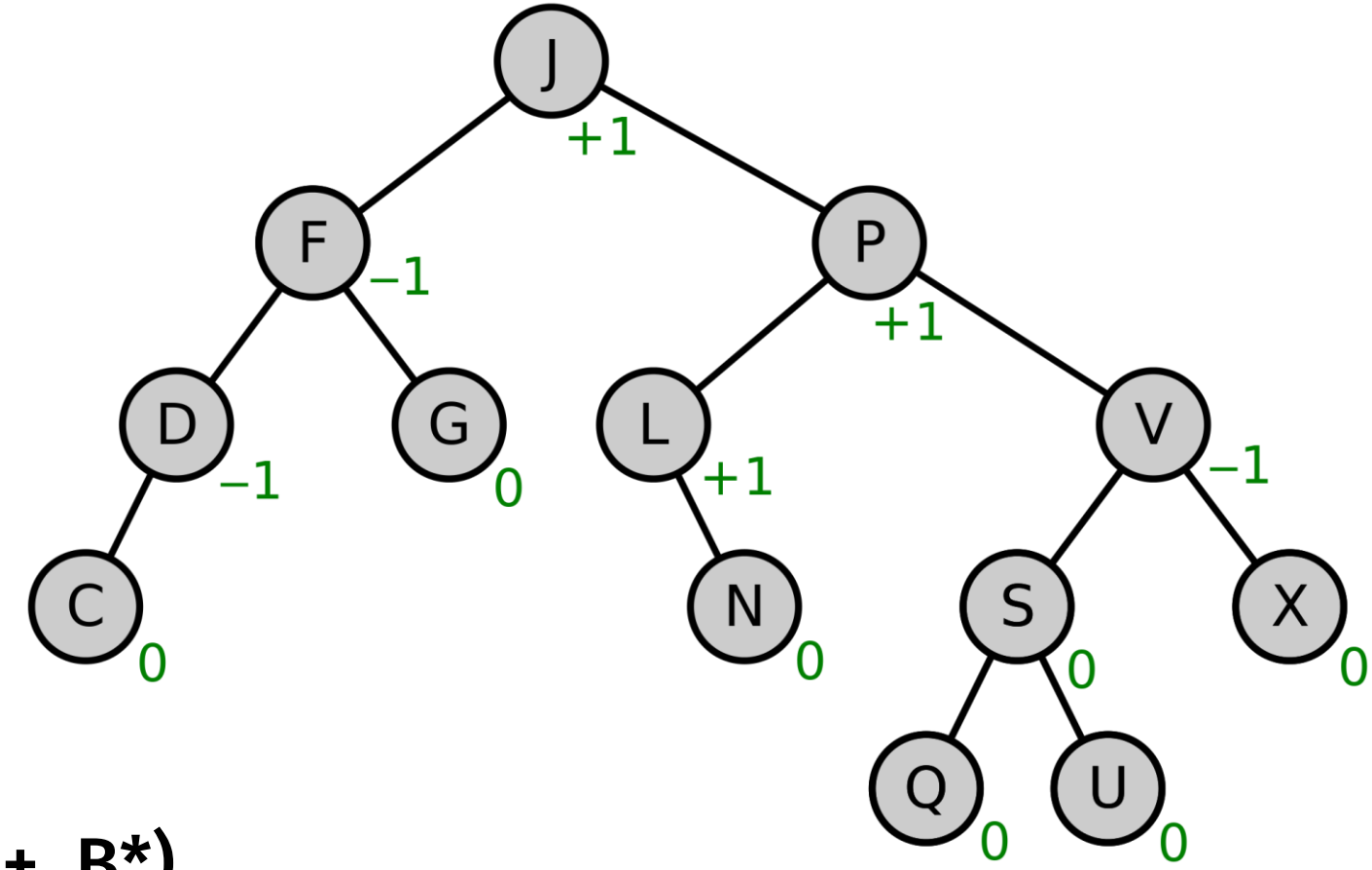


- Kinder manchmal weit weg `(2i + 1)`



InfoA: AVL-Baum

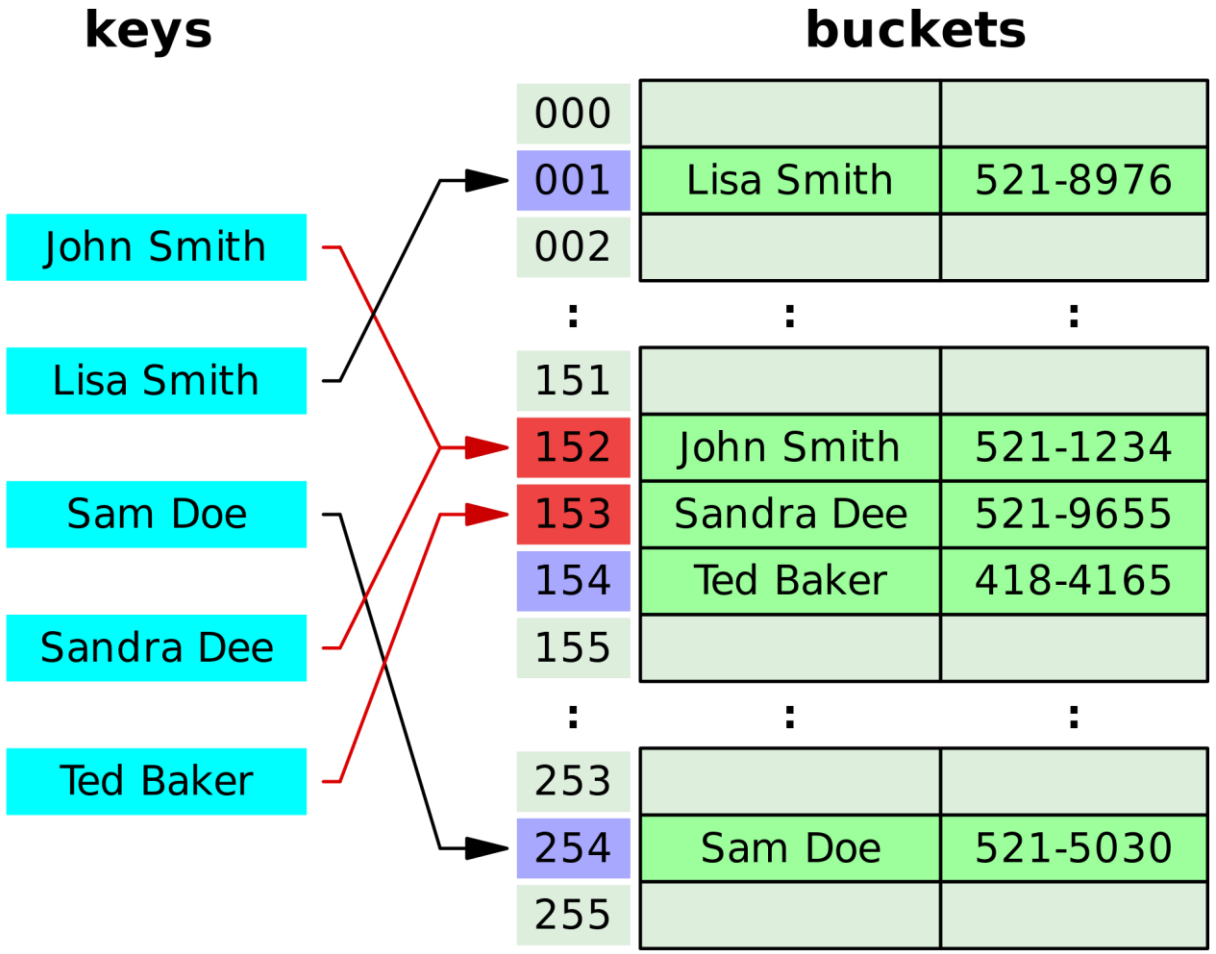
- Jeder Knoten extra allokiert
- Absteigen immer Cache-Miss

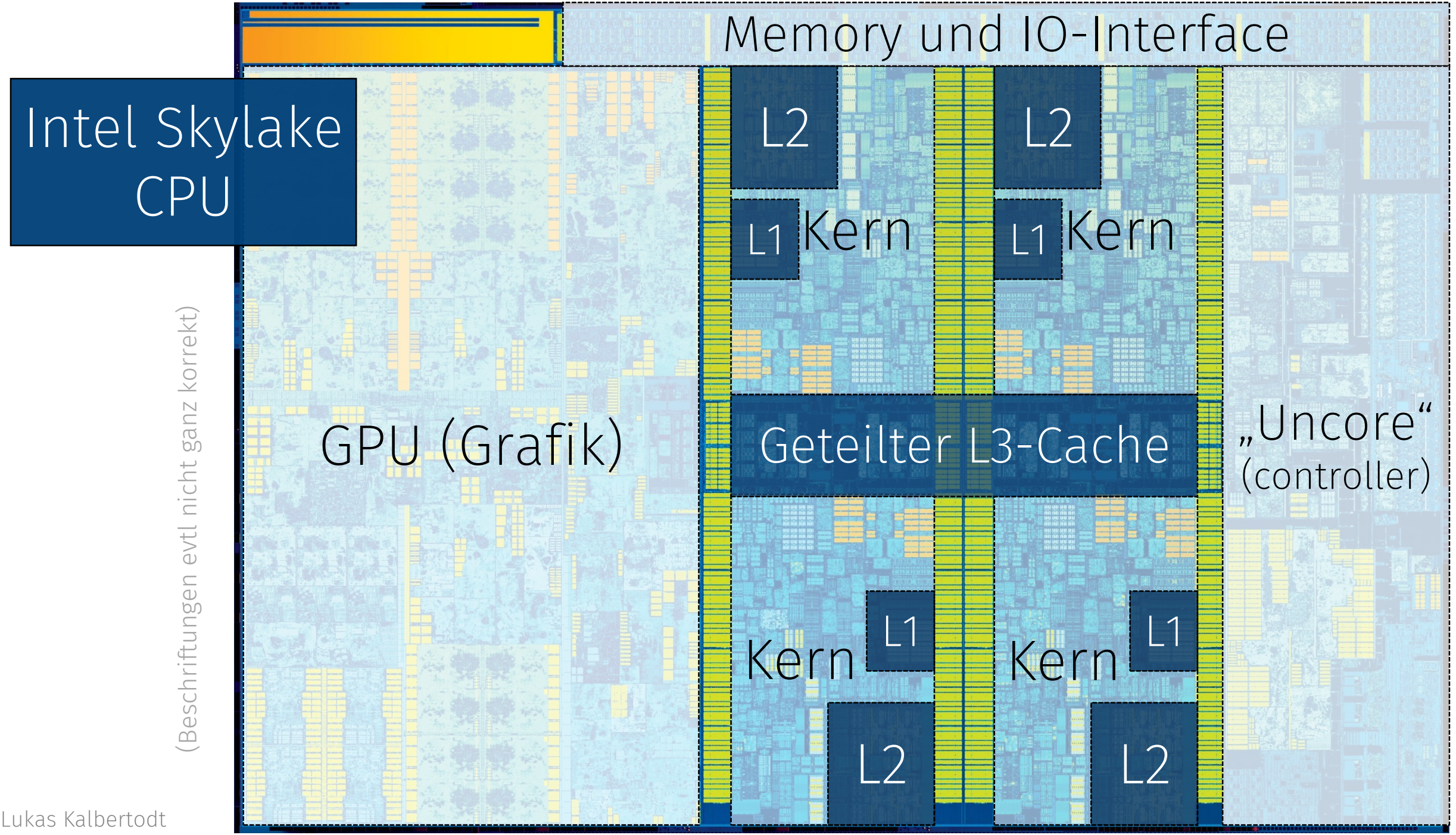


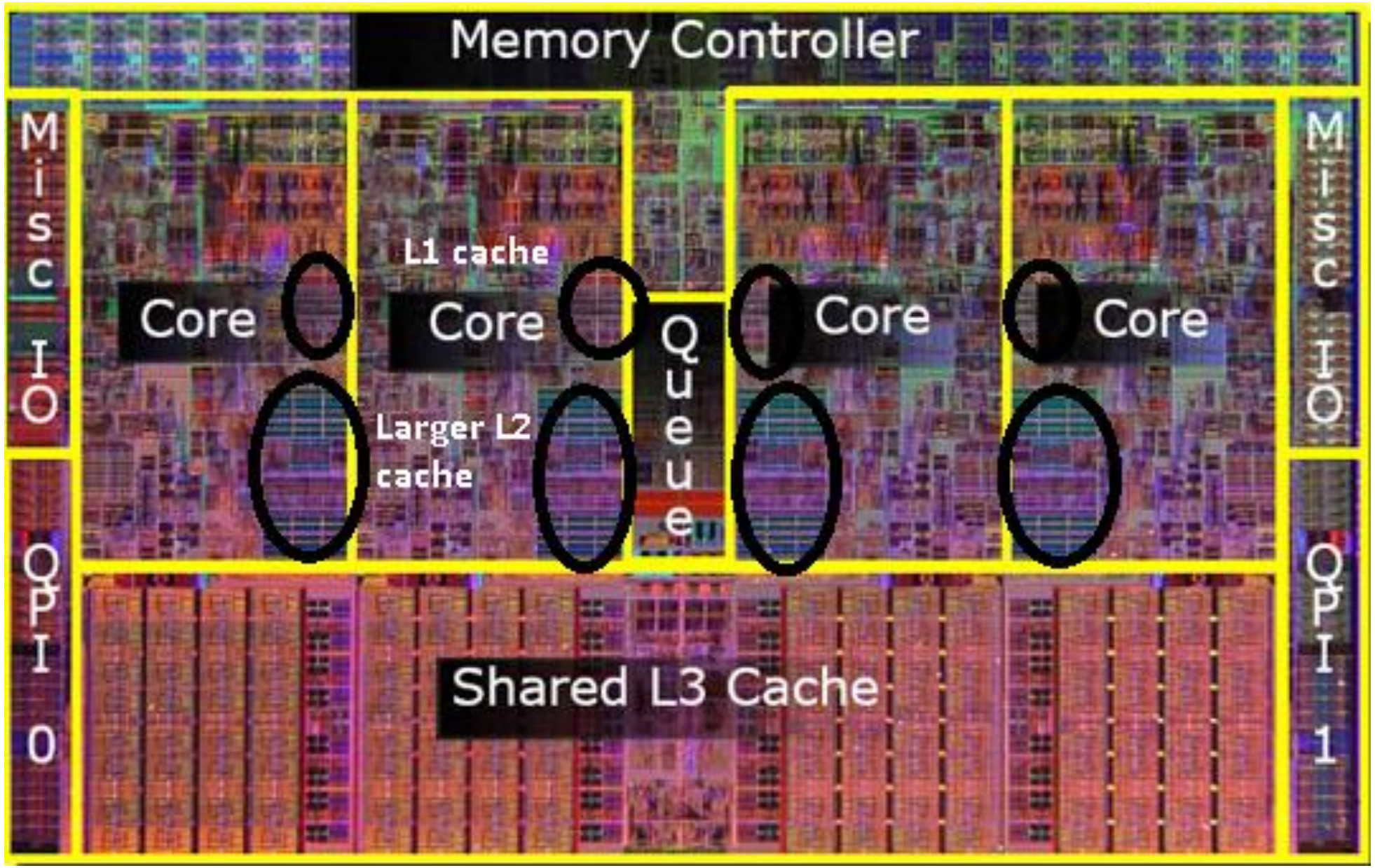
Besser: **B-Baum (B+, B*)**

InfoA: Hash-Tabelle

- Hängt von Strategie ab!
- Offenes Hashing:
  - Evtl. Linked List
- Geschlossenes Hashing:
  - Linear: Sehr lokal
  - Quadratisch: Große Sprünge



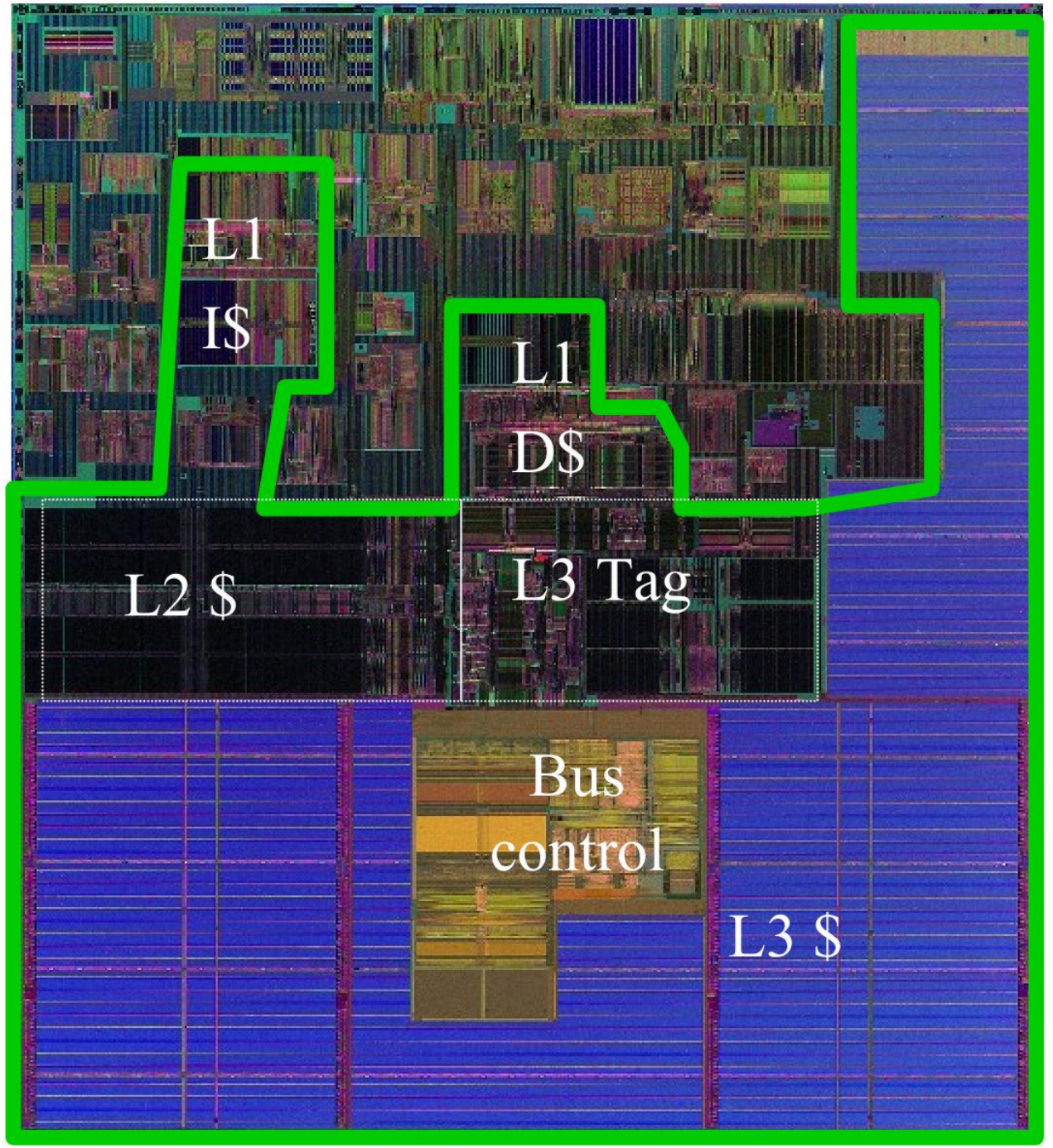

Intel Skylake CPU
Memory und IO-Interface
L2
L2
L1
Kern
L1
Kern
GPU (Grafik)
Geteilter L3-Cache
„Uncore“
(controller)
Kern
L1
Kern
L1
L2
L2
(Beschriftungen evtl nicht ganz korrekt)


Memory Controller
Misc IO
Misc IO
L1 cache
Core
Core
Core
Core
Queue
Larger L2 cache
QPI 0
QPI 1
Shared L3 Cache

Quelle

Intel Itanium II

- 211 Millionen Transistoren
- Davon 85% für Caches
- Insgesamt nur 1% der Fläche zur eigentlichen *Berechnung*!



Riesiger Aufwand, um Warten zu vermeiden!

Quelle

●●●

Branch Prediction & icache

- Instructions auch Daten
- Instruction Cache (*icache*)
- Sprünge:
  - *unbedingt*: Prefetcher fetcht :-)
  - *bedingt*: Problematisch…
  - *dynamisch*: :-(
- Vorhersage bedingter Sprünge
  - Erfolg von ≈ 98%

```
func:
   push rax
   mov rcx, rdi
   mov esi, 2
.LBB1_1:
   mov rdx, rsi
   imul rdx, rdx
   mov al, 1
   cmp rdx, rcx
   ja .LBB1_5
   test rsi, rsi
   je .LBB1_6
   xor edx, edx
   mov rax, rcx
   div rsi
   inc rsi
   test rdx, rdx
   jne .LBB1_1
   xor eax, eax
.LBB1_5:
   pop rcx
   ret
.LBB1_6:
   lea rdi, [rip + ploc8314]
   call _ZN4core9panicking
```

Pipelining

- Ausführung einer Instruktion in mehreren Teilen
- 5 bis 20 Pipeline-Stages
- Pro Kern parallele Berechnungen
- CPU ordnet Anweisungen um!

Ziel: Pipeline immer voll → Wahrscheinlichen Zweig direkt ausführen

CPUs sind...

... batshit crazy!

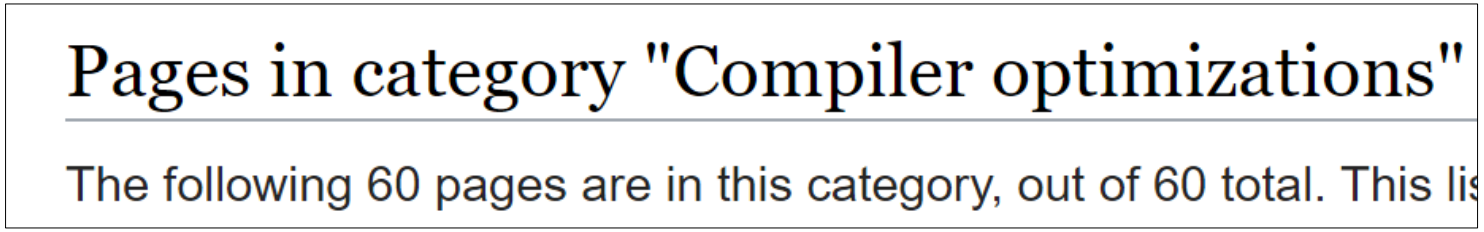
- Cache Verwaltung
  - Prefetcher (+ Zugriffsmusteranalysen)
  - Synchronisation von L1 & L2 Caches zwischen Kernen
  - Store-Buffer
  - Instruction Cache
- Branch Predictor

Hallo Compiler!

Compileroptimierungen

- *(Partial) Loop Unrolling*
- *Constant Fold*
- *Loop nest optimization:* Cache-freundliche verschachtelte Schleifen
- *Inlining*: Funktionsrumpf an Stelle des Aufrufs kopieren
- *Common Subexpression Elimination*
- *Dead Code Elimination*
  - Teilweise mit tiefen Analysen (Beispiel: Binäre Suche)

Und **viele** mehr…


Pages in category "Compiler optimizations"

The following 60 pages are in this category, out of 60 total. This lis

Compiler sind...

...auch batshit crazy!

Es wird nicht ansatzweise der Code ausgeführt, den man geschrieben hat!

Langsame vs. Schnelle Sprachen?

„Low Level“ **nicht** äquivalent zu „schnell“!

Viel Kontrolle	➔	Programmierer kann schlau sein	➔	Schnell
Eingeschränkt	➔	Compiler kann schlau sein	➔	Schnell

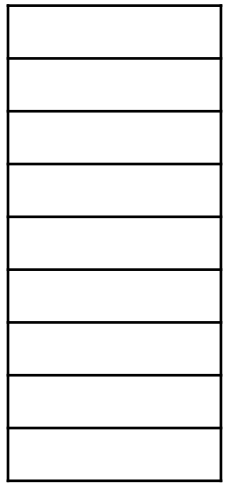
Gründe für *langsame* Sprachen:

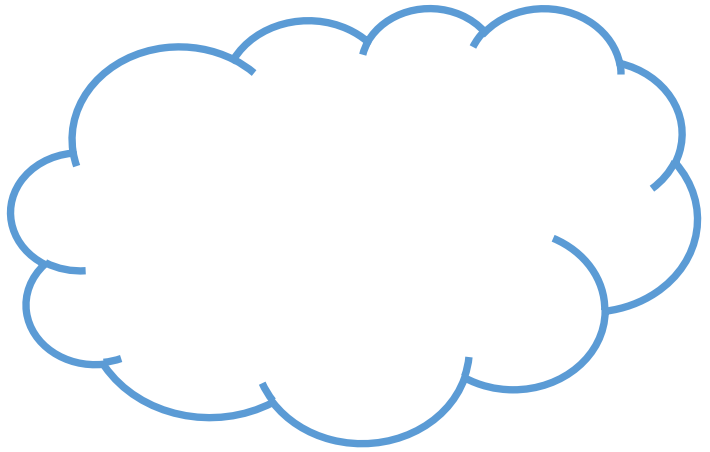
- Alles boxen (auf dem Heap)
- Dynamic Dispatch
- Dynamisches Typsystem

Heap

Stack

- dicht
- „hot“
- schnell





Heap

- verteilt
- oft „cold“
- Allokation kostet Zeit

- Manchmal ist Heap nötig (dynamische Größe…)
- Java: Alle nicht-primitiven Objekte auf dem Heap

Dynamic Dispatch

- „Dynamisches Binden“
- Realisierung per `vtable`
- CPU springt an dynamische Speicheradresse
  - icache-Miss
  - Pipelining kommt zum stehen
  - Inlining nicht möglich!
- Optimierung: Devirtualization
  - JVM besonders gut

Dynamisches Typsystem

„Nicht zu wissen, was einem
auf dem Heap erwartet“

- Benötigt Heap-Allokation
- Bei jeder Operation:
  - Typ prüfen
  - Mögliche Fehler-Branches erstellen

Fazit

1. Erst programmieren, dann messen, dann optimieren!
   - Von Makro- zu Mikro-Optimierung
2. An die *CPU* denken:
   - Kompakte Datenstrukturen benutzen
   - Regelmäßige, lineare Speicherzugriffe
3. An den *Compiler* denken:
   - Erst idiomatischen Code schreiben!
   - Optimierungen überprüfen (Assembly angucken)

Quelle

Referenzen

[1] Chandler Carruth, „Efficiency with Algorithms, Performance with Data Structures”, https://www.youtube.com/watch?v=fHNmRkzxHWs

[2] Donald Knuth, „Computer Programming as an Art”, 1974, S. 671

[3] Andrei Alexandrescu, “Writing Quick Code in C++, quickly”, https://www.youtube.com/watch?v=ea5DiCg8HOY