Signalverarbeitungsalgorithmen

von der Theorie zur Implementierung

Markus Püschel
Electrical and Computer Engineering
Carnegie Mellon University

(13. – 16. Dezember 2004, TU Wien)

Idee der Vorlesung

Diese Vorlesung:

Signalverarbeitung
(Fouriertransformation, Filter)

Algorithmen

Implementierung

Computerarchitektur

Überblick über die Vorlesung

- **Teil I: Was ist Signalverarbeitung?**
  - Beispiele
  - Was sind die Bausteine der Signalverarbeitung?
  - Warum sind schnelle Implementierungen wichtig?
- **Teil II: Algorithmentheorie**
  - Lineare Transformationen
  - Diskrete Fouriertransformation (DFT)
  - FIR Filter
- **Teil III: Implementierung**
  - Probleme
  - Algorithmus, Implementierung und Mikroarchitektur: Beispiel DFT
  - Selbstadaptive Software: FFTW und SPIRAL
- **Abschließende Bemerkungen**

Teil I: Was ist Signalverarbeitung?

Definitionen

- **Definition: Signalverarbeitung**
  - [Die Disziplin die sich befasst mit] der Darstellung, Transformation und Manipulation von Signalen und der Information die sie enthalten (Oppenheim, Schafer 1999)
- **Definition: Signal**
  - (In der Signalverarbeitung) Eine Funktion über einem Indexbereich

$$s : \; I \rightarrow \mathbb{K}, \qquad i \mapsto s(i)$$

Typische Beispiele:

$\mathbb{K} = \mathbb{R}, \mathbb{C}, GF(2)$
(reelle, complexe, Bit-signale)
$I = \mathbb{R}, \mathbb{Z}, \{0, \ldots, n-1\}$
(kontinuierliche, diskrete, endliche Signale)

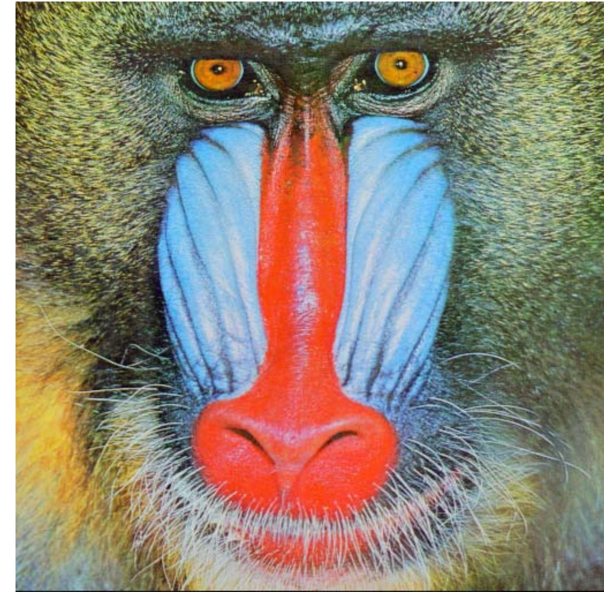
Beispiele

- **Multimedia**
  - Sprache (1-D), Bild (2-D), Video (3-D)
  - Qualitätsverbesserung, Kompression, Übertragung
- **Biometrie**
- **Medizinische/Biobildverarbeitung**
- **Computersehen**
- **Kommunikation**

Multimedia: Beispiel Bildkompression

Lena

Pepper

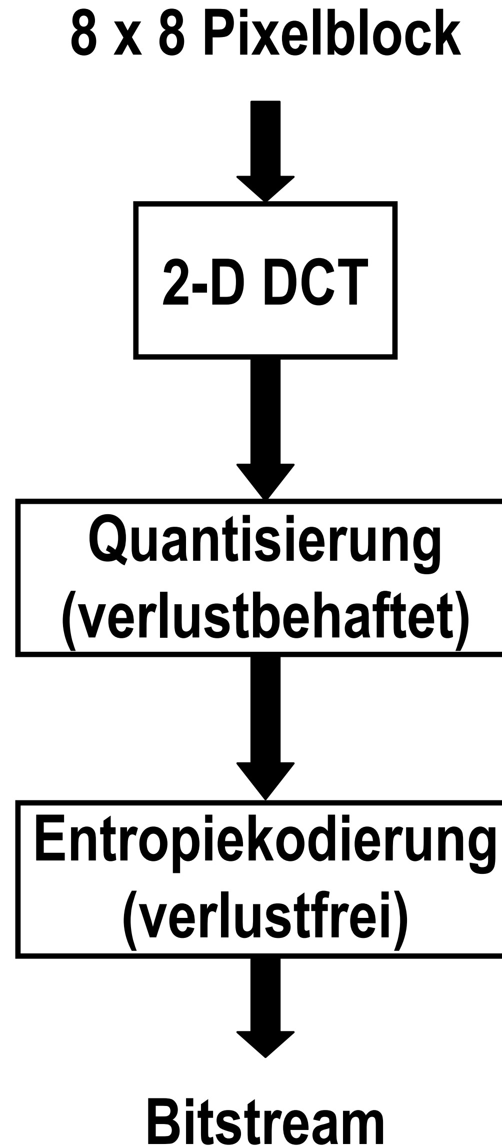


Baboon

$512 \times 512 \times 3$ bytes = 768KB

With JPEG, ~32KB

JPEG: Wie funktioniert's?



JPEG versus JPEG2000

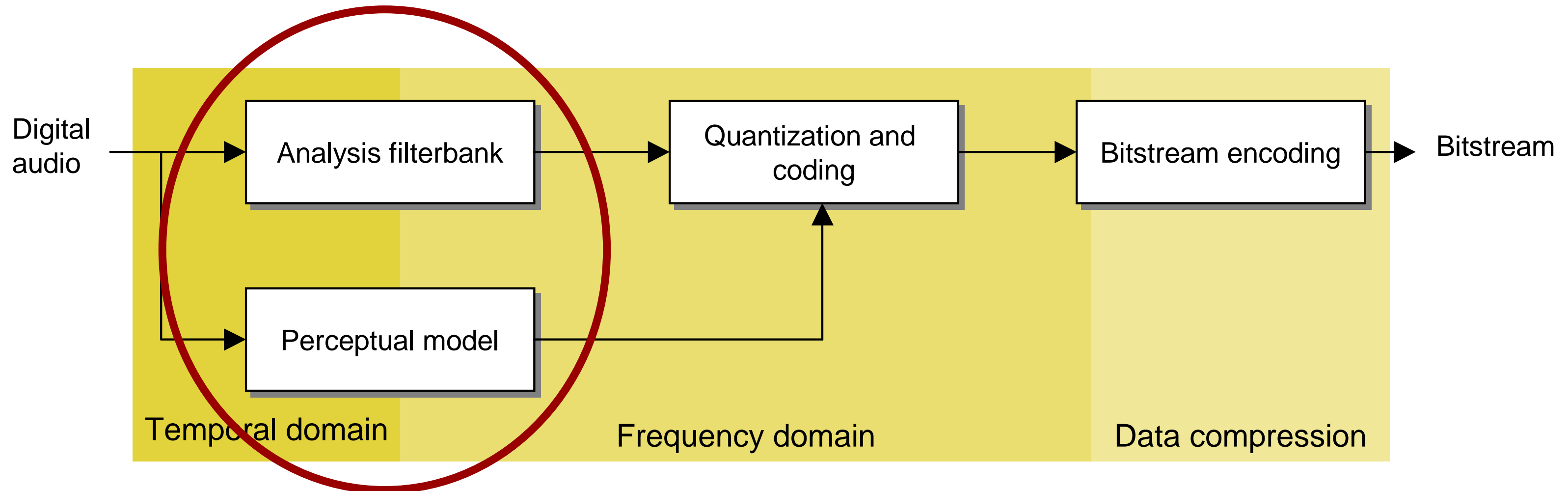
original: 3MB

JPEG: 19KB (DCT basiert)

JPEG2000: 19KB (wavelet basiert)

Multimediakodierung

- MPEG-I bis MPEG-IV
- Enthält Standards für Audio, Bild und Video
- Beispiel: MPEG-II, layer III audio = MP3



DFT, MDCT, DCT

Beispiel: Biometrie

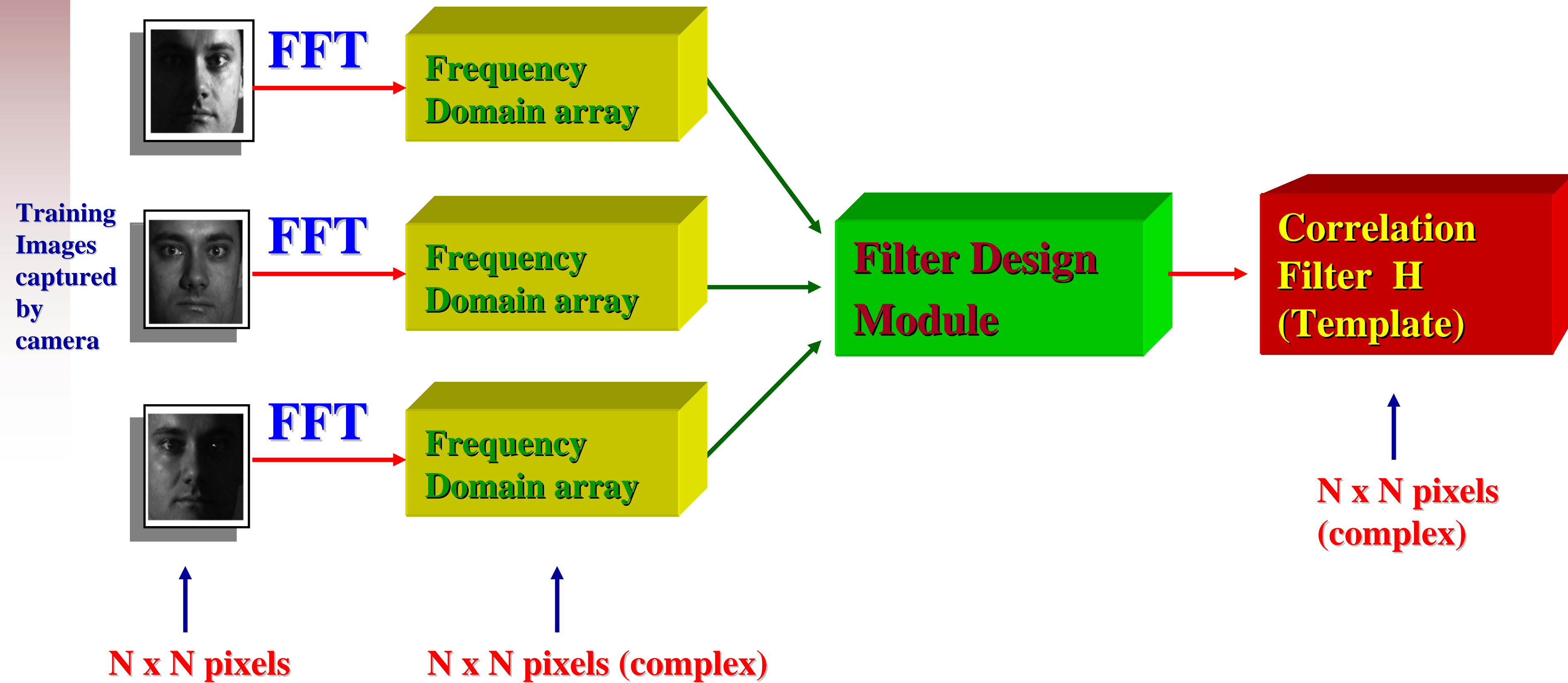
Facial Expression

Fingerprints

Illumination

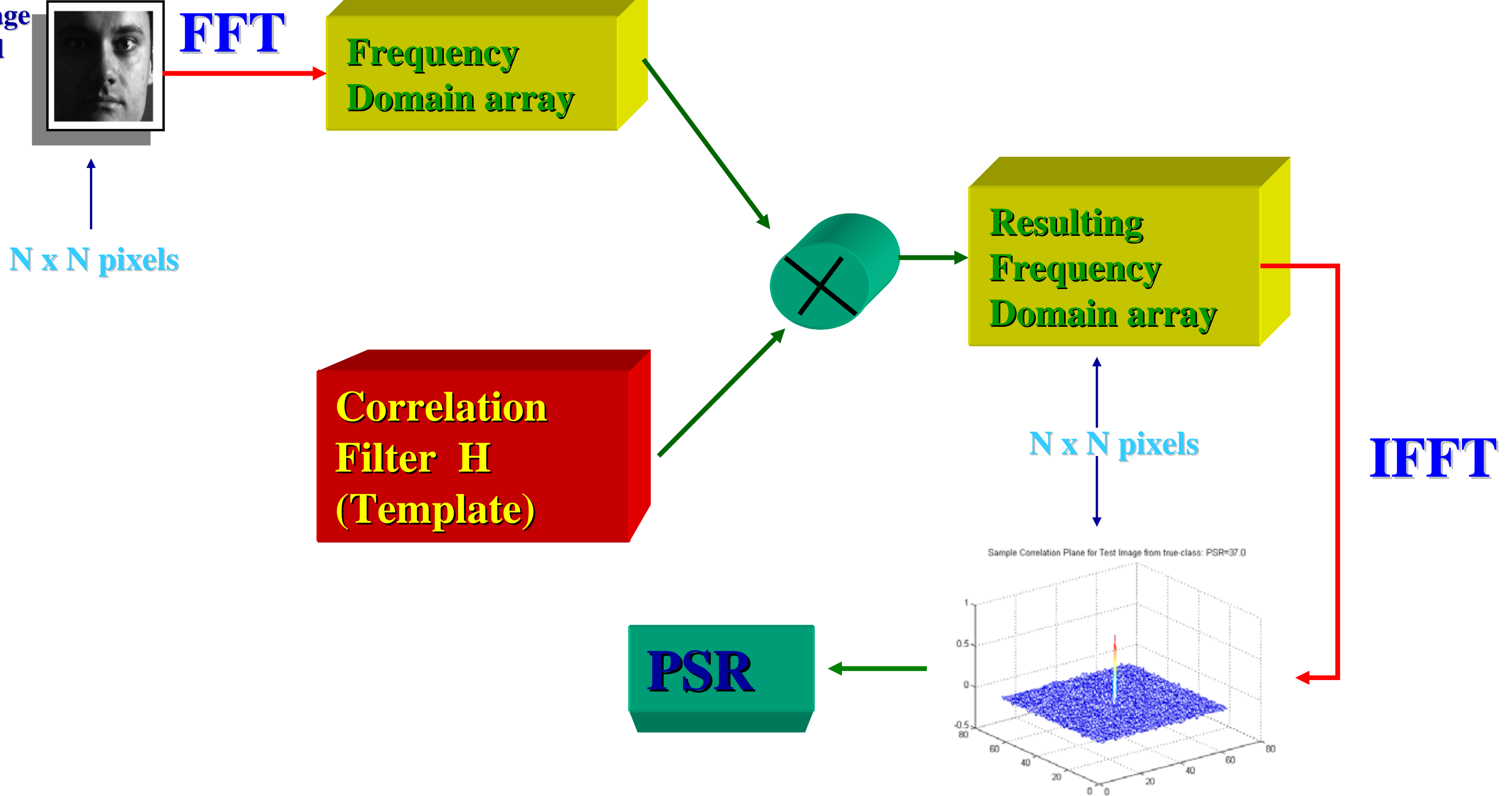
Folie: Bhagavatula/Savvides

Wie's Funktioniert: Registrierung



Folie: Bhagavatula/Savvides

Wie’s Funktioniert: Identifikation



Folie: Bhagavatula/Savvides

Beispiel: Cardiac MRI

long-axis

Superior Vena Ca

Pulmonary Vein

Right Atrium

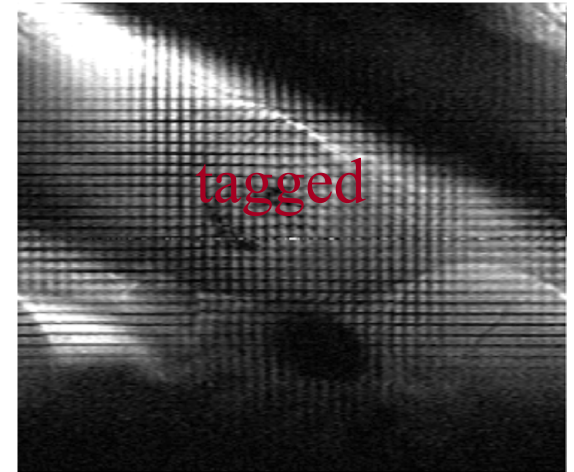
Tricuspid Valve

Inferior Vena Cava

transversal slices

short-axis

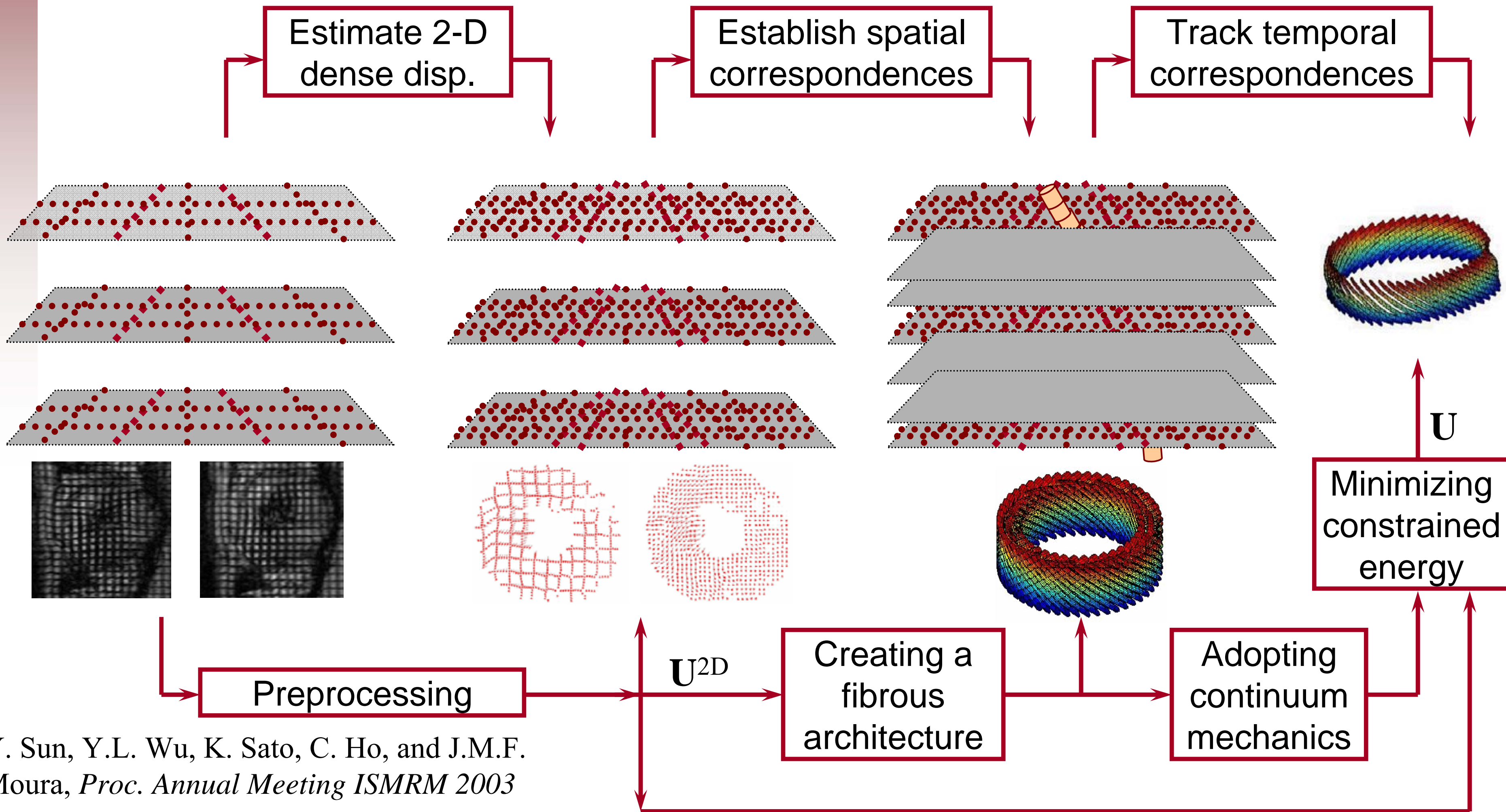
longitudinal slices



Ziel: 3-D-Movie von 2-D Daten

Folie: Hsien/Moura

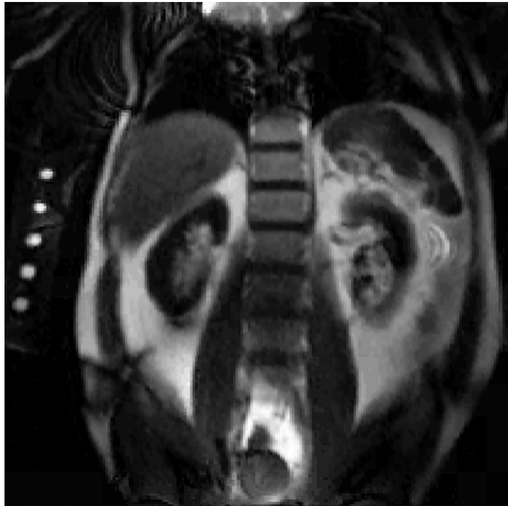
3-D Motion Estimation Procedure



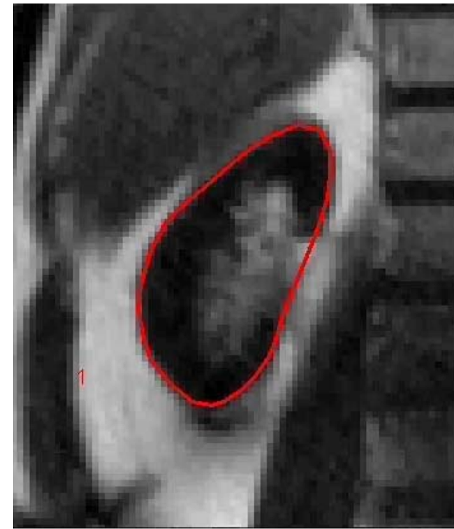
Y. Sun, Y.L. Wu, K. Sato, C. Ho, and J.M.F. Moura, *Proc. Annual Meeting ISMRM 2003*

Folie: Hsien/Moura

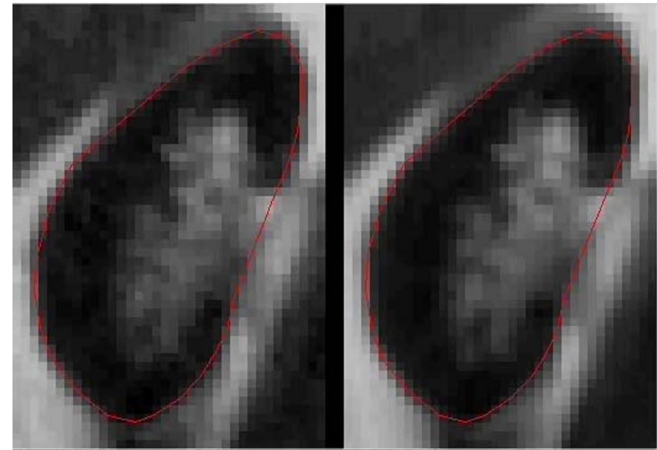
Beispiel: MRI



MRI



Tracking der Nieren

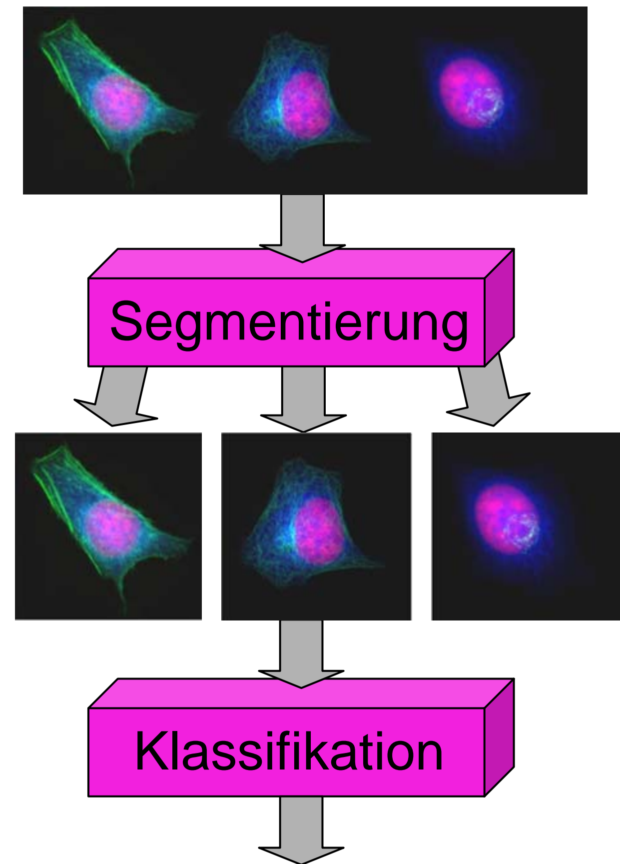


Herausrechnen der Bewegung

Folie: Sun/Moura

Beispiel: Biobildverarbeitung

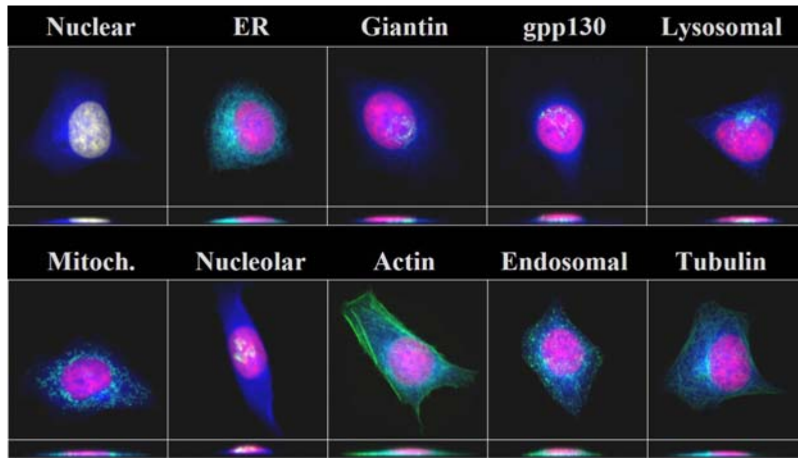
- **Ziel: Schnelle Identifikation von Proteinen mittels ihrer Verteilung in der Zelle**
- **Signalverarbeitung**
  - Segmentierung
  - Klassifikation (Wavelets, Frames)



Das ist Tubulin!

Folie: Kovacevic/Murphy

Bilder



Folie: Kovacevic/Murphy

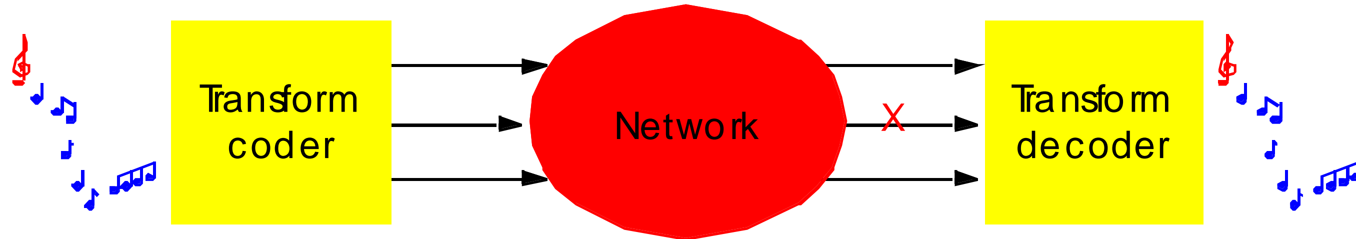
Beispiel: Computersehen

Suberbowl 2001 (Kanade et al.)

Plot: Kanade

Beispiel: Kommunikation

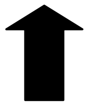
- Ziel: Robustheit gegen Verluste



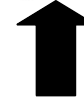
Folie: Kovacevic

Foto-zur-Oma Problem

- Ziel: sende ein digitales Foto nach Italien
- Verfügbar: FedEx or regulärer Post “Kanal”
  - FedEx 99% verlässlich, Kosten $39.99
  - Postal 80% verlässlich, Kosten $3.40
- 1 Floppy pro Umschlag
- Foto braucht 2 Floppies (CDs sind noch nicht erfunden)



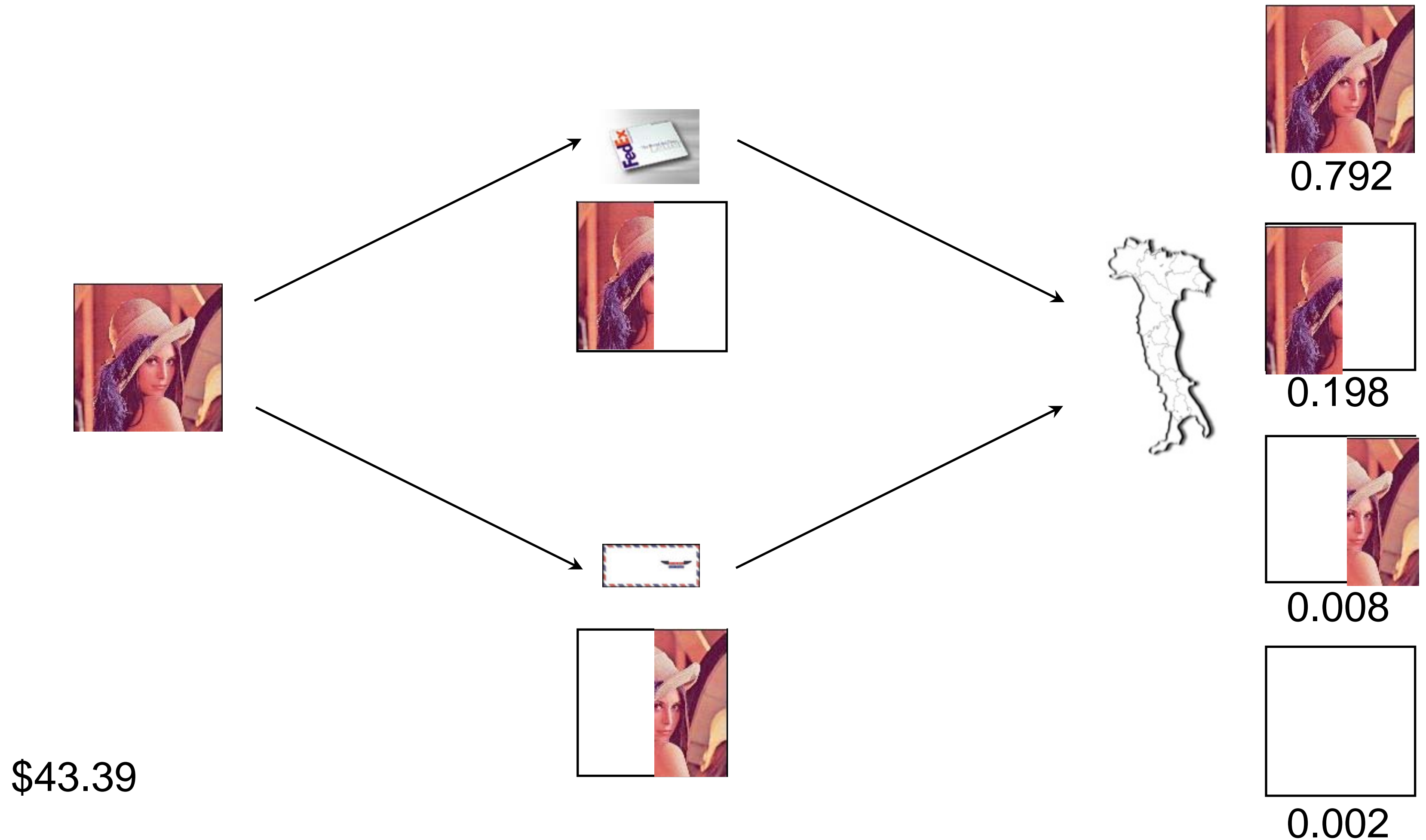
neue Freundin



Oma wohnt in Italien

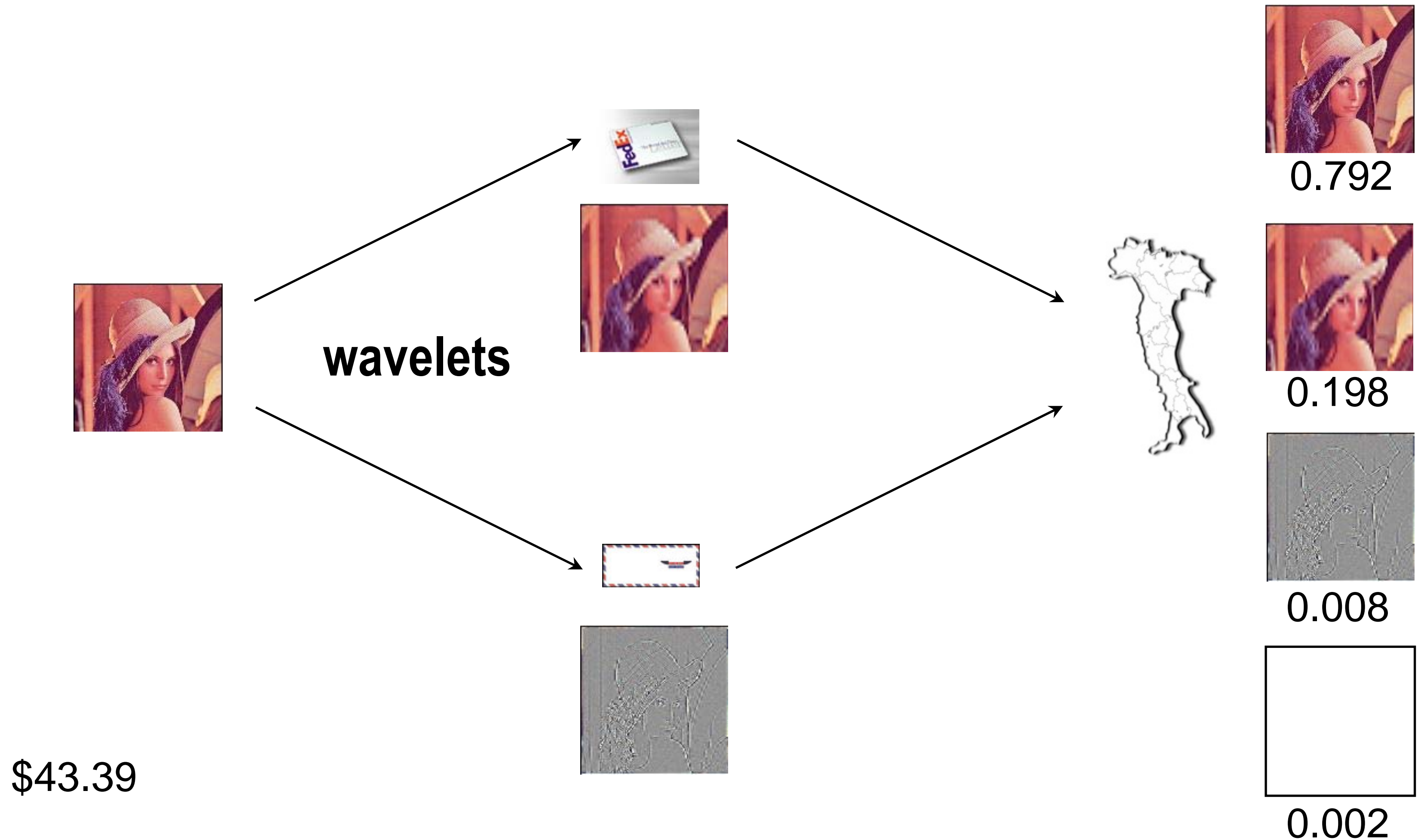
Folie: Kovacevic

Heterogener Kanal (dumme Lösung)



Folie: Kovacevic

Heterogener Kanal (kluge Lösung)



Folie: Kovacevic

Zusammenfassung: Hauptbausteine der Signalverarbeitung

Filter:
FIR, IIR, Korrelation, Filterbänke

Signaltransformationen:
DFT, DCT, wavelets, frames

Lineare Algebra:
Vektorsumme,
Matrix-Vektorprodukt,
…
Singulärwertzerlegung,
Matrixinversion,
…

Kodierung:
Huffman, arithmetisch,
Viterbi, LDPC

Implementierung

- **Unendliche Anforderungen an die Geschwindigkeit**
  - Sehr große Datensätze
  - Echtzeit
- **Vielfalt der Plattformen**
  - Hardware: ASIC, FPGA
  - Software
    - Ein- versus Mehrprozessorsystemen
    - Workstation versus eingebetteter Prozessor
    - Gleitpunkt versus Fixpunktarithmetik
  - Kombinierte Hardware/Software Plattformen
- **Problem: Implementierung schwierig, aufwendig, veraltet schnell**

In dieser Vorlesung: Einprozessor Workstations

Teil II: Algorithmentheorie

In dieser Vorlesung

- Signaltransformationen
  (hauptsächlich diskrete Fouriertransformation)

- FIR Filter

Signaltransformationen

- Mathematisch: Matrix-Vektor Multiplikation

$$x \mapsto y = M \cdot x$$

Eingabevektor (Signal) → x
Ausgabevektor (Signal) → y
Transformation = Matrix → M

- Wird oft so geschrieben:

$$y_k = \sum_{\ell=0}^{n-1} m_{k,\ell} x_\ell, \qquad M = [m_{k,\ell}]_{0 \le k,\ell < n}$$

- Idee: Stelle Signal in einer anderen Basis dar für bessere Verarbeitung

Transformationen: Beispiele

$$
\begin{aligned}
\mathbf{DFT}_n &= [e^{-2k\ell\pi i/n}]_{0\le k,\ell<n} \\
\mathbf{DCT\text{-}2}_n &= \left[\cos(k(2\ell+1)\pi/2n)\right]_{0\le k,\ell<n}, \\
\mathbf{DCT\text{-}3}_n &= \mathbf{DCT\text{-}2}_n^T \quad \text{(transpose)}, \\
\mathbf{DCT\text{-}4}_n &= \left[\cos((2k+1)(2\ell+1)\pi/4n)\right]_{0\le k,\ell<n}, \\
\mathbf{IMDCT}_n &= \left[\cos((2k+1)(2\ell+1+n)\pi/4n)\right]_{0\le k<2n, 0\le \ell<n}, \\
\mathbf{RDFT}_n &= [r_{k\ell}]_{0\le k,\ell<n}, \quad r_{k\ell} = \begin{cases} \cos\frac{2\pi k\ell}{n}, & k \le \lfloor\frac{n}{2}\rfloor \\ -\sin\frac{2\pi k\ell}{n}, & k > \lfloor\frac{n}{2}\rfloor \end{cases}, \\
\mathbf{WHT}_n &= \begin{bmatrix} \mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2} \\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2} \end{bmatrix}, \quad \mathbf{WHT}_2 = \mathbf{DFT}_2, \\
\mathbf{DHT} &= \left[\cos(2k\ell\pi/n) + \sin(2k\ell\pi/n)\right]_{0\le k,\ell<n}.
\end{aligned}
$$

Schnelle Algorithmen

- Transformation per Definition: $O(n^2)$
- Transformationen haben schnelle Algorithmen: $O(n \log(n))$
- Beispiel: FFT (fast Fourier transform)
- Diese existieren aufgrund von “Symmetrien” in den Matrizen

Die Geschichte schneller Algorithmen …

- **…beginnt mit der FFT**

- **FFT (Cooley-Tukey 1965) wird oft bezeichnet als der Anfang der digitalen Signalverarbeitung**

- **Geschichte:**
  - ~1805: FFT entdeckt von Gauss (nach [1])
    (Fourier publizierte das Konzept Fourieranalyse in 1807!)
  - 1965: Neu entdeckt von Cooley-Tukey
  - 2002: James W. Cooley bekommt die IEEE Jack S. Kilby Signal Processing Medal
    "For pioneering the Fast Fourier Transform (FFT) algorithm."

[1]: Heideman, Johnson, Burrus: "Gauss and the History of the Fast Fourier Transform" Arch. Hist. Sc. 34(3) 1985

Wie wurden/werden Algorithmen Gefunden?

- **Laaange die Matrix angucken bis man was sieht**
- **Beispiel: Cooley-Tukey FFT**
- **Anderes Beispiel:** *G. Bi "Fast Algorithms for the Type-III DCT of Composite Sequence Lengths" IEEE Trans. SP 47(7) 1999*

Darstellung von Algorithmen

- **Darstellung von Algorithmen: 2 Schulen**
  - Folge von Summationen
  - Strukturierte Matrixfaktorisierung: $M = M_1 \cdot M_2 \ldots M_k$

- **Beispiel vorige Folie:**

$$\mathrm{DCT}_n = K_m^n \left(\bigoplus_{0 \le i < k} \mathrm{DCT}_m(r_i) \right) (\mathrm{DCT}_k \otimes I_m) B_{n,k}$$

Beispiel C.-T. FFT: Was ist besser?

Das?:

$$\mathrm{DFT}_n = L^n_{n_2}(I_{n_1} \otimes \mathrm{DFT}_{n_2}) T^n_{n_1} (\mathrm{DFT}_{n_1} \otimes I_{n_2})$$

Oder das?: $k = n_1 k_1 + k_2,\ j = n_2 j_1 + j_2$

$$y_{n_2 j_1 + j_2} = \sum_{k_1=0}^{n_1-1} \left(\omega_n^{j_2 k_1}\right) \left(\sum_{k_2=0}^{n_2-1} x_{n_1 k_2 + k_1} \omega_{n_2}^{j_2 k_2}\right) \omega_{n_1}^{j_1 k_1}$$

Rest an der Tafel ….

- Definition DFT
- Interpretation mit dem Chinesischen Restesatz (CRS)
- DFT Eigenschaften, Zirkulante, Toeplitzmatrizen
- Herleitung der Cooley-Tukey FFT mit dem CRS
- Andere DFT Algorithmen (FFTs)
- Komplexität der DFT

- FIR Filter
- FIR Filter Algorithmen

FFT Literatur

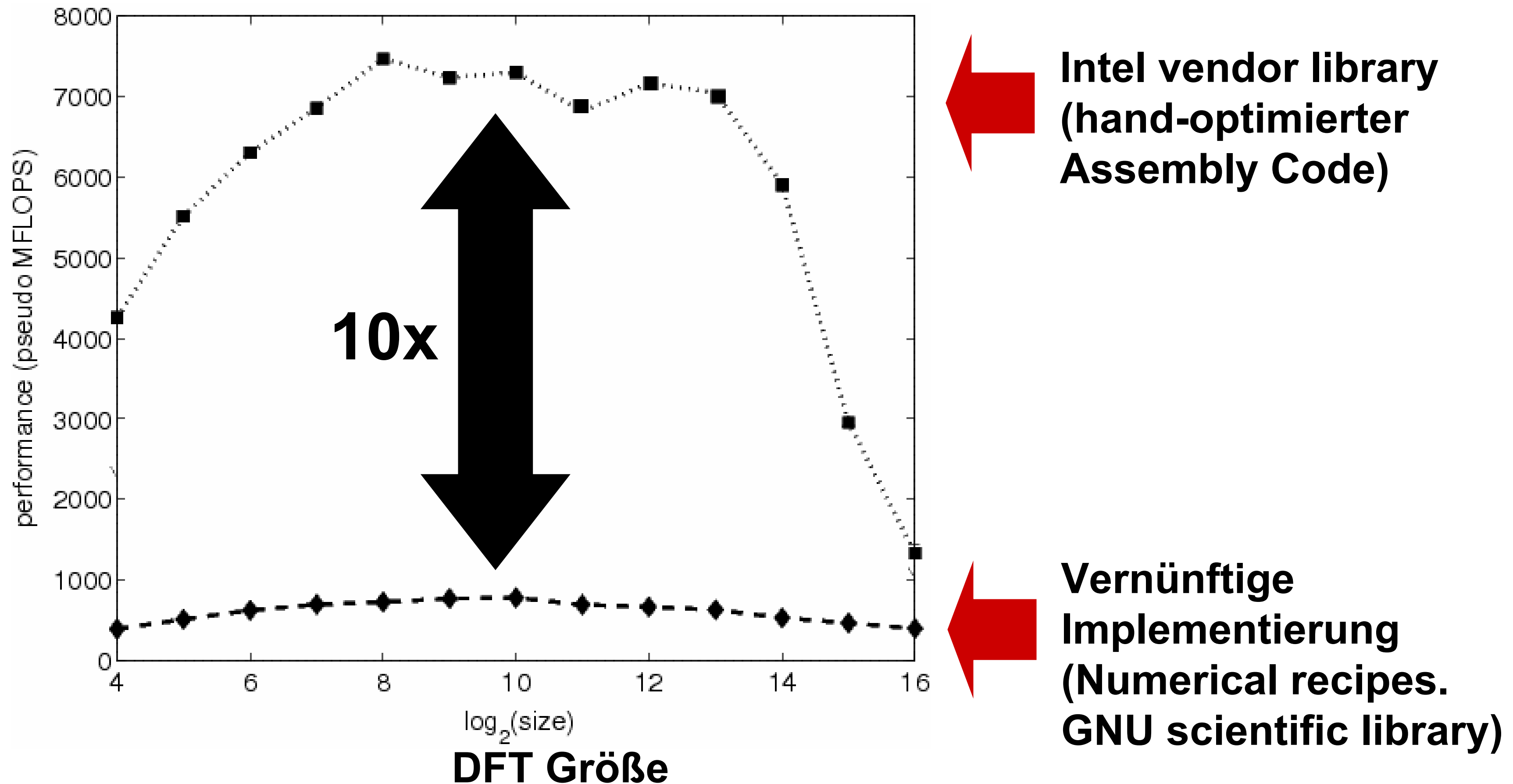
- Nussbaumer (1982): Fast Fourier Transforms and Convolution Algorithms
- Van Loan (1992): Computational Frameworks for the Fast Fourier Transform
- Clausen/Baum (1993): Fast Fourier Transforms
- Tolimieri/An/Lu (1997): Algorithms for discrete Fourier transform and convolution

Teil III: Implementierung

Implementierung: Überblick

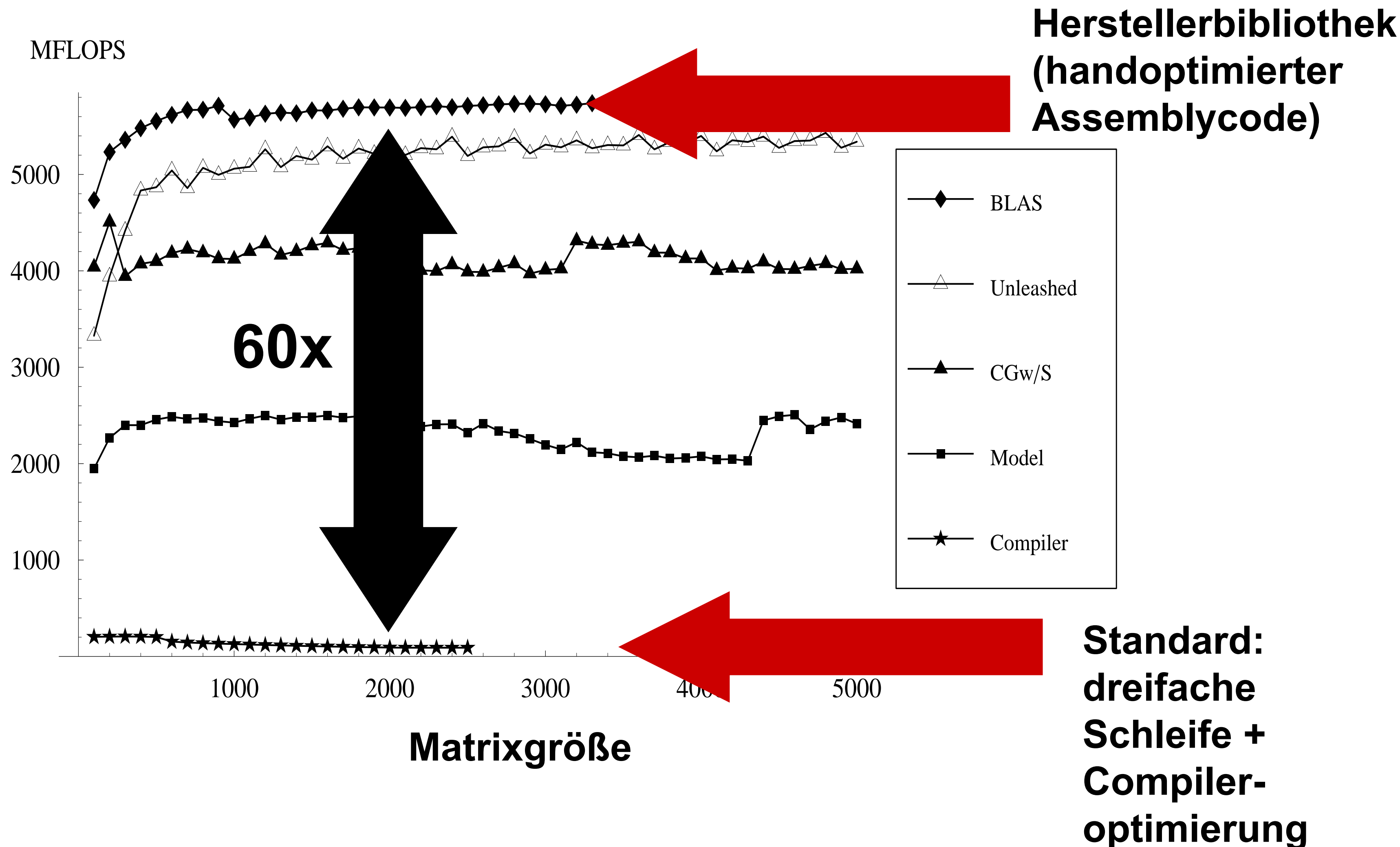
- Das Problem: Moore's Law und Numerische Software
- Mikroarchitektur und Architektur
- Algorithmus, Implementierung und Mikroarchitektur: Analyse der DFT und Herleitung von FFTW
- Softwaregenerierung und Experimente mit SPIRAL
- Zusammenfassung: Wie schreibt man schnellen numerischen Code?

Das Problem: Beispiel DFT auf Pentium 4



Ok, aber die DFT ist ja auch kompliziert, nehmen wir was einfacheres …

Das Problem: Matrix-matrix Multiplikation

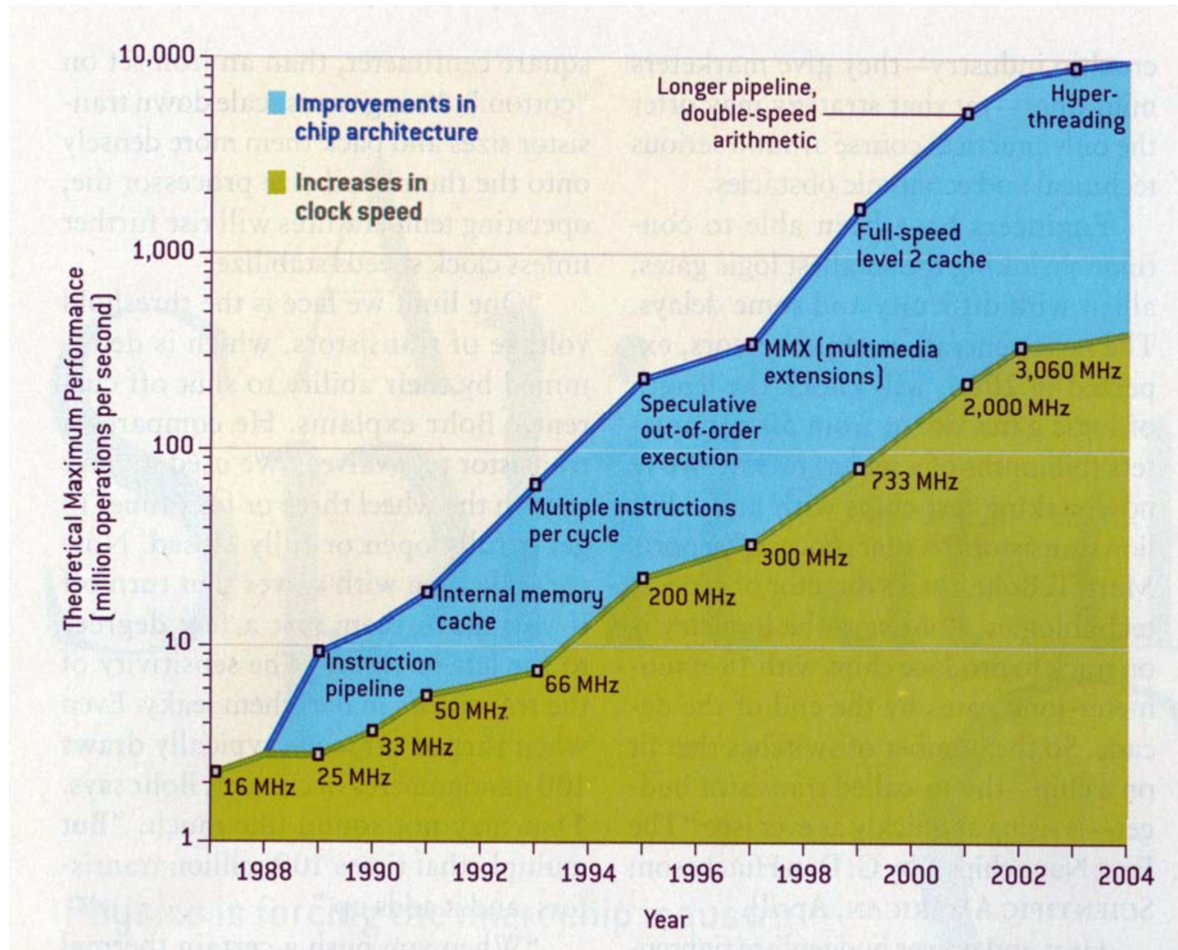


Woran liegt das?

graph: Pingali, Yotov, Cornell U.

Moore's Law

- **Moore's Law: exponentielles (x2 in ~18 Monaten) Wachstum Anzahl Transistoren/Chip**



source: Scientific American, Nov 2004, p. 98

Aber alles hat seinen Preis …

Moore's Law: Auswirkungen

- **Computer sind komplex**
  - Mehrstufige Speicherhierarchie
  - Spezialinstruktionen
  - Undokumentierte Hardwareoptimierungen
- **Konsequenzen:**
  - Laufzeit hängt nur grob von der Anzahl Operationen ab
  - Laufzeitverhalten schwer zu verstehen
  - Compilerentwicklung kann kaum mithalten
  - **Der schnellste Code (inkl. Algorithmus) ist computerabhängig**
  - **Es ist sehr schwierig schnellen Code zu schreiben**
- **Computer entwickeln sich schnell**
  - Hochoptimierter Code veraltet fast so schnell (schneller?) wie man ihn schreibt

Wie produziert man schnellen Code?
und mit erträglichem Aufwand?

Implementierung: Überblick

- Das Problem: Moore's Law und Numerische Software
- Mikroarchitektur und Architektur
- Algorithmus, Implementierung und Mikroarchitektur: Analyse der DFT und Herleitung von FFTW
- Softwaregenerierung und Experimente mit SPIRAL
- Zusammenfassung: Wie schreibt man schnellen numerischen Code?

Definitionen

- **Architektur:** (auch ISA) Die Teile eines Prozessordesigns die man verstehen muss um Assemblycode zu schreiben, z.B. die Instruktionsspezifikation und die Register.
  Die Teile des Designs die nur die Implementierung betreffen, z.B. Cachegröße und Taktfrequenz gehören nicht dazu.
  Beispiel: ia32 (x86), ia64

- **Mikroarchitektur:** Implementierung der Architektur
  Beispiel: Pentium4 Mikroarchitekturen

Speicherhierarchie, Cachestruktur

- Mach's besser an der Tafel

SIMD (Signal Instruction Multiple Data) Vektorinstruktionen

- **Was ist das?**
  - Erweiterung der ISA. Datentypen und Instruktionen zum parallelen Rechnen mit kurzen (2-8) Vektoren von Integers oder Floats

- **Warum gibt's sie?**
  - **Nützlich:** viele Anwendungen (z.B. Multimedia) haben den nötigen feinkörnigen Parallismus – Code potentiell schneller
  - **Machbar:** Chipdesigner haben genügend Transistoren zum spielen

Überblick Vektor ISAs

Vendor	Name	ν-way	Precision	Processor
Intel	SSE	4-way	single	Pentium III Pentium 4
Intel	SSE2	2-way	double	Pentium 4
Intel	SSE3	4-way 2-way	single double	Pentium 4
Intel	IPF	2-way	single	Itanium Itanium 2
AMD	3DNow!	2-way	single	K6
AMD	Enhanced 3DNow!	2-way	single	K7, Athlon XP Athlon MP
AMD	3DNow! Professional	4-way	single	Athlon XP Athlon MP
AMD	AMD64	2-way 4-way 2-way	single single double	Athlon 64 Opteron
Motorola	AltiVec	4-way	single	MPC 74xx G4
IBM	AltiVec	4-way	single	PowerPC 970 G5
IBM	Double FPU	2-way	double	PowerPC 440 FP2

Die Evolution von Intel Vektorinstruktionen

- **MMX (1996, Pentium)**
  - Nur Integers, 64-bit geteilt in 2 x 32 bis 8 x 8
  - MMX Register = Floatregister
  - Nicht mehr sehr wichtig mit modernen Grafikkarten
- **SSE (1999, Pentium III)**
  - Obermenge von MMX
  - 4-weg Floatops, einfache Genauigkeit
  - 8 neue 128 Bit Register
  - 100+ Instruktionen
- **SSE2 (2001, Pentium 4)**
  - Obermenge von SSE
  - "MMX" auf SSE Registern, 2 x 64
  - 2-weg Floatops, doppelte Genauigkeit, gleiche Register wie 4-weg einfach
- **SSE3 (2004, Pentium 4 Prescott)**
  - Obermenge von SSE2
  - Mehr 2-weg Vektorinstruktionen speziell für komplexen Code

Wie benutzt man Vektorinstruktionen?

- **Voraussetzung: feinkörniger Parallelismus**
- **Am einfachsten: Benutze existierende Bibliothek**
- **Selbst schreiben:**
  - Compilervektorisierung
  - Erweiterung von C (Intrinsics, compilerspezifisch)
  - Assembly schreiben (alles oder Teile inline)

Probleme

- Verlangt richtig ausgerichtete Daten
- Datenumordnungen zerstören leicht die Laufzeit
- Verlangt oft eine Anpassung des Algorithmus
- Optimierungen/Implementierungen maschinenabhängig
- Verlangt gutes Verständnis der ISA + Mikroarchitektur

Beispiel: 4x4 Matrix-Vektor Multiplikation

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix} \bullet \begin{bmatrix} a \\ a \\ a \\ a \end{bmatrix}$$

4-weg Vektoren (ab SSE) mit Intrinsics:

```
__m128 BLAS_II_4x4 (__m128 *matrix, __m128 vector) {
   __m128 t[4];

   for (int i=0; i<4; i++)
      t[i] = _mm_mul_ps(matrix[i], vector);

   _MM_TRANSPOSE_PS(t[0], t[1], t[2], t[3]);

   return _mm_add_ps(_mm_add_ps(t[0], t[1]),
                     _mm_add_ps(t[2], t[3])); }
```

4 vmults

1 Transposition

3 vadds

Beispiel: Komplexe Multiplikation SSE3

Komplexer C99 Code + Compilervektorisierung funktioniert sehr gut

Komplexer Code hat natürlichen 2-weg Vektorparallelismus

Und so sieht der Assemblycode aus

SSE3:

```
movapd    xmm0, XMMWORD PTR A
movddup   xmm2, QWORD PTR   B
mulpd     xmm2, xmm0
movddup   xmm1, QWORD PTR   B+8
shufpd    xmm0, xmm0, 1
mulpd     xmm1, xmm0
addsubpd  xmm2, xmm1
movapd    XMMWORD PTR C, xmm2
```

SSE2:

```
movsd     xmm3, QWORD PTR A
movapd    xmm4, xmm3
movsd     xmm5, QWORD PTR A+8
movapd    xmm0, xmm5
movsd     xmm1, QWORD PTR B
mulsd     xmm4, xmm1
mulsd     xmm5, xmm1
movsd     xmm2, QWORD PTR B+8
mulsd     xmm0, xmm2
mulsd     xmm3, xmm2
subsd     xmm4, xmm0
movsd     QWORD PTR C, xmm4
addsd     xmm5, xmm3
movsd     QWORD PTR C, xmm5
```

In SSE2 ist Skalarcode besser

Wie schreibt man guten Vektorcode?

- **Nimm den “richtigen” Algorithmus und die “richtigen” Datenstrukturen**
  - Feinkörniger Parallelismus
  - Korrekte Ausrichtung im Speicher
  - Zusammenhängende Felder (arrays)
- **Verwende eine guten Compiler (z.B. vom Hersteller)**
- **Zuerst: probiere Compilervektorisierung**
  - Richtige Optionen + Pragmas (Information an den Compiler: z.B. Datenausrichtung, Schleifenunabhängigkeit)
  - Überprüfe Assemblycode und Laufzeit
- **Wenn nötig: schreibe selbst Vektorcode**
  - Zuerst die wichtigste (teuerste) Subroutine
  - Benutze Intrinsics, kein Inlineassembly
  - Wichtig: Verstehe ISA

Implementierung: Überblick

- Das Problem: Moore's Law und Numerische Software
- Mikroarchitektur und Architektur
- Algorithmus, Implementierung und Mikroarchitektur: Analyse der DFT und Herleitung von FFTW
- Softwaregenerierung und Experimente mit SPIRAL
- Zusammenfassung: Wie schreibt man schnellen numerischen Code?

Auch besser an der Tafel

FFTW Codeletbeispiel

- No-twiddle-codelet, size 8

FFTW Zusammenfassung

- Eine Bibliothek für FFTs beliebiger Größe, Dimension, reell und komplex
- Sehr schnell (Adaptierung + benutzt Vektorcode)
- Stabil, einfache Installation (configure – make)
- Alles in allem: tolle Software, sehr populär

Wie evaluiert man Code?

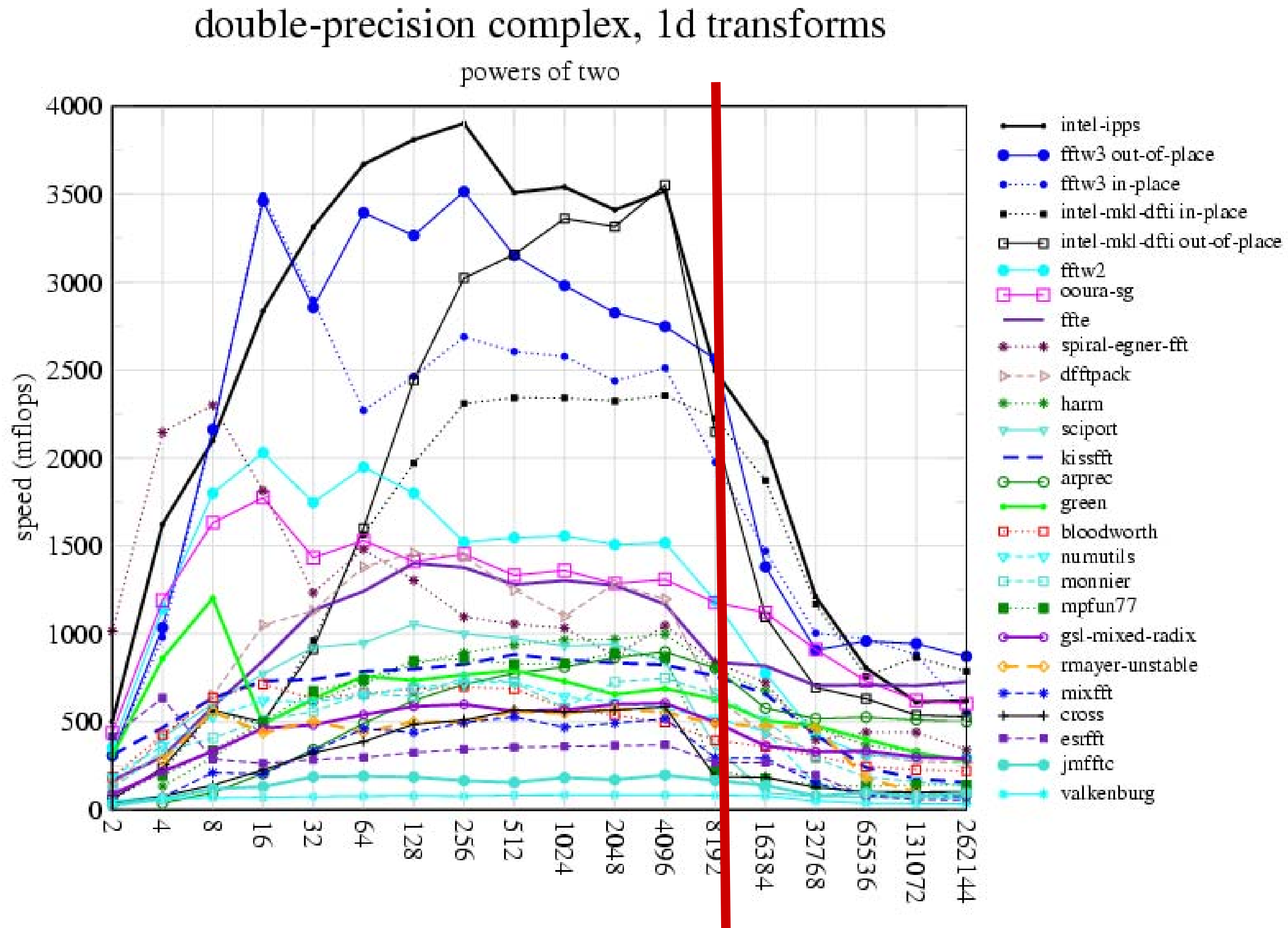
- **Vergleiche gegen die beste erhältliche Software**
  - Stelle sicher daß die gleiche Zeitmessmethode verwendet wird

- **Berechne die Leistung (in MFLOPS)**

$$MFLOPS = Anzahl\ Operationen/(10^6 \cdot Laufzeit\ [s])$$

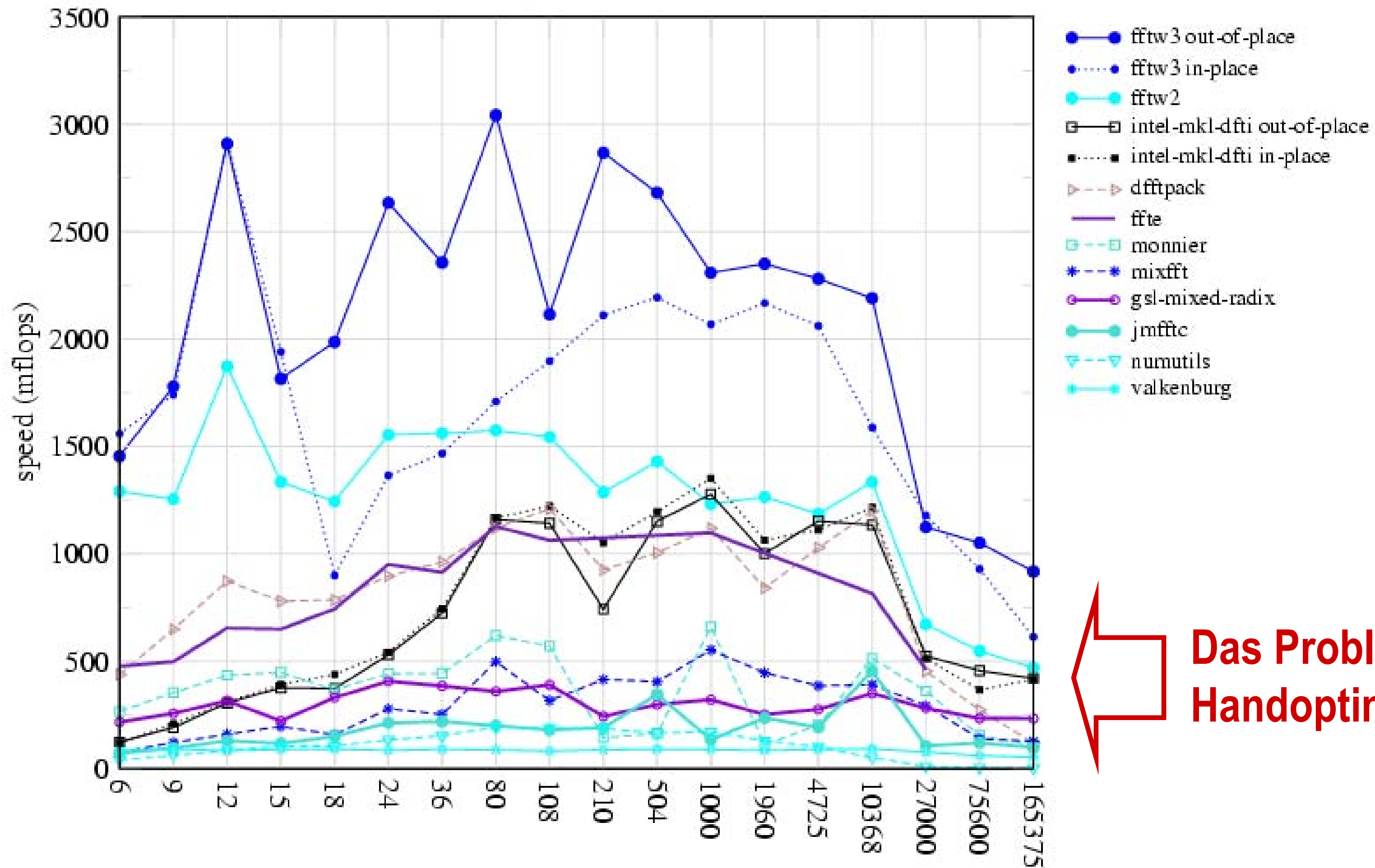
und vergleiche zur Maximalleistung der Maschine
  - Erfordert die Berechnung/Messung der Anzahl der Operationen
  - Maximalleistung aus dem Manual oder die richtigen Leute fragen
  - Vorsicht: Code mit höherer Leistung kann langsamer sein

FFTW Benchmarks, Pentium 4



Quelle: www.fftw.org/speed

FFTW Benchmarks, Pentium 4



Das Problem mit Handoptimierung

Quelle: www.fftw.org/speed

Andere FFTs: CT, GT, Rader, Bluestein

recht billig

$$\mathbf{DFT}_{km} \to (\mathbf{DFT}_k \otimes \mathrm{I}_m)\, \mathsf{T}^n_m (\mathrm{I}_k \otimes \mathbf{DFT}_m)\, \mathsf{L}^n_k$$

teuer

$$\mathbf{DFT}_{km} \to P_n (\mathbf{DFT}_k \otimes \mathbf{DFT}_m) Q_n, \qquad \mathrm{gcd}(k, m) = 1$$

$$\mathbf{DFT}_p \to R_p^T (\mathrm{I}_1 \oplus \mathbf{DFT}_{p-1}) D_p (\mathrm{I}_1 \oplus \mathbf{DFT}_{p-1}) R_p$$

sehr teuer

$$\mathbf{DFT}_n \to D_m\, \mathbf{DFT}_m\, D'_m\, \mathbf{DFT}_m\, D''_m, \qquad m > 2n$$

Probleme:

- Verschmelzung der Permutationen
- erst kürzlich (teilweise) gelöst (Paper eingereicht)
- können nicht einfach in FFTW integriert werden

Implementierung: Überblick

- Das Problem: Moore’s Law und Numerische Software
- Mikroarchitektur und Architektur
- Algorithmus, Implementierung und Mikroarchitektur: Analyse der DFT und Herleitung von FFTW
- Softwaregenerierung und Experimente mit SPIRAL
- Zusammenfassung: Wie schreibt man schnellen numerischen Code?

SPIRAL Information

Seit: Mitte 1998

Gefördert von:
DARPA
NSF ACR-0234293
NSF ITR/NGS-0325687

und:
Cylab, CMU
Austrian Science Fund
Intel
ITRI, Taiwan
ENSCO, Inc.

www.spiral.net

SPIRAL Team: Faculty
James C. Hoe (ECE, CMU)
Jeremy Johnson (CS, Drexel)
José M. F. Moura (ECE, CMU)
David Padua (CS, UIUC)
Markus Püschel (ECE, CMU)
Manuela Veloso (CS, CMU)
Robert W. Johnson (Quarry Comp. Inc.)

Studenten:
Bryan W. Singer (CS, CMU)
Jianxin Xiong (CS, UIUC)
Franz Franchetti (ECE, CMU, vorher TU Wien)
Aca Gacic (ECE, CMU)
Yevgen Voronenko (ECE, CMU)
Anthony Breitzman (CS, Drexel)
Kang Chen (CS, Drexel)
Pinit Kumhom (ECE, Drexel)
Adam Zelinski (ECE, CMU)
Peter Tummeltshammer (CS, TU Wien)
...

Kollaboration mit:
Christoph Überhuber (Math, TU Wien)

Spiral

- Codegenerierung für Signaltransformationen
  (DFT, DCT, Filter, Wavelets, …)
- Automatische Optimierung und Adaptierung von Algorithmus und Implementierung
- Verschiedene Codetypen
  (skalar, Vector, FMA, Fixpunkt, multiplikationsfrei, …)
- Ermöglicht vielseitige Experimente um verschiedene Algorithmen zu testen

SPIRAL: Ziele

- **Ein flexibles, erweiterbares Codegenerierungsystem dass nicht veraltet (soweit möglich) für eine ganze Klasse von Algorithmen**

- **Suche nach Antworten auf die Fragen:**
  - Bis zu welchem Grad kann man Handcodierung abschaffen?
  - Wie bringt man dem Computer Algorithmenwissen bei?

Codegenerierung und Adaptierung als Optimierungsproblem

T	eine Signaltransformation
P	die Zielplattform
$\mathcal{I} = \mathcal{I}(\mathrm{T}, \mathrm{P})$	Menge aller Implementierungen von T auf P
$\mathrm{C} = \mathrm{C}(\mathrm{T}, \mathrm{I}, \mathrm{P})$	Laufzeit von der Implementierung I von T auf P

Die Implementierung von T adaptiert zu P ist:

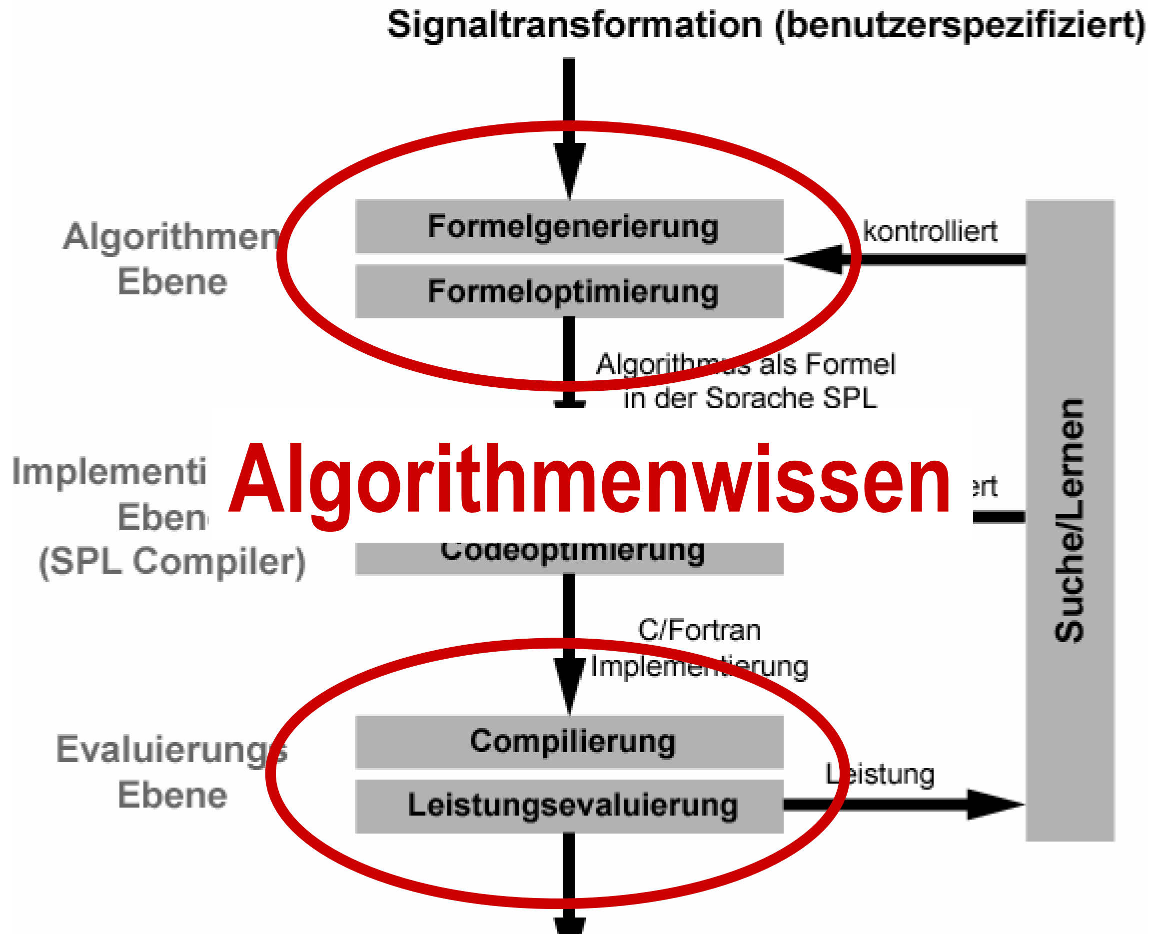
$$\widehat{\mathrm{I}} = \widehat{\mathrm{I}}(\mathrm{P}) = \arg\min_{\mathrm{I} \in \mathcal{I}(\mathrm{P})} \mathrm{C}\left(\mathrm{T}, \mathrm{P}, \mathrm{I}\right)$$

Probleme:

- Wie beschreibt und generiert man die Menge der Implementierungen?
- Wie minimiert man effizient C?

SPIRAL nützt die spezifische Struktur von Transformationsalgorithmen um die Optimierung automatisch durchzuführen

Spiral's Architektur

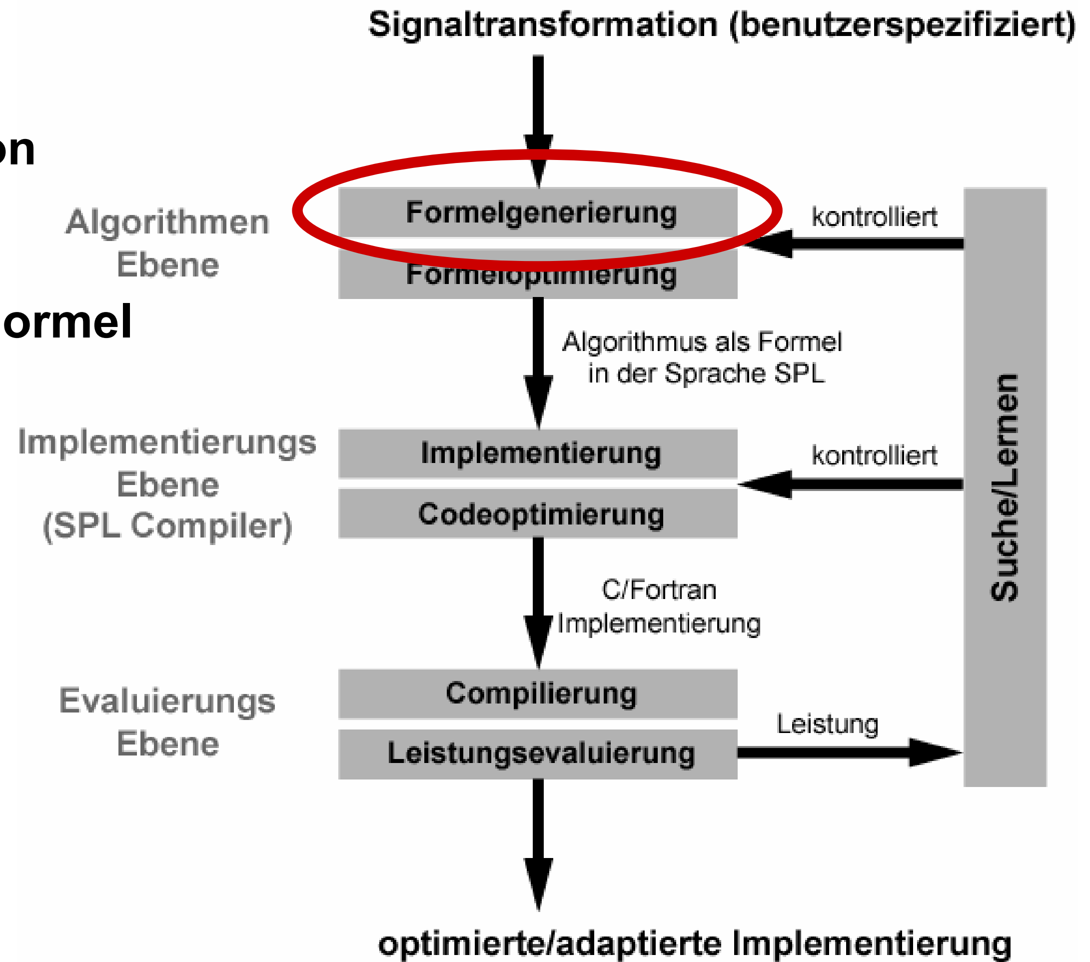


Information über den Computer

Von Transformation zum Algorithmus (Formel)

Eingabe:
Transformationsspezifikation

Ausgabe:
Schneller Algorithmus als Formel



Algorithmen: Beispiel DFT der Größe 4

Cooley/Tukey FFT (Größe 4):

$$\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & -1 & -i\\ 1 & -1 & 1 & -1\\ 1 & -i & -1 & i\end{bmatrix} = \begin{bmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & 1\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & -1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & i\end{bmatrix}\begin{bmatrix}1 & 1 & 0 & 0\\ 1 & -1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 1 & -1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Fouriertransformation | Diagonalmatrix (twiddles)

$$DFT_4 = (DFT_2 \otimes I_2) \cdot T_2^4 \cdot (I_2 \otimes DFT_2) \cdot L_2^4$$

Kroneckerprodukt | Identität | Permutation

- Mathematische Darstellung zeigt die Struktur des Algorithmus:
  SPL = Signal Processing Language
- Enthält die notwendige Information um Code zu generieren

SPL: Definition (BNF)

$$
\begin{array}{rcll}
\langle\text{spl}\rangle & ::= & \langle\text{generic}\rangle \mid \langle\text{symbol}\rangle \mid \langle\text{transform}\rangle \mid & \\
 & & \langle\text{spl}\rangle \cdot\cdots\cdot \langle\text{spl}\rangle \mid & \text{(product)} \\
 & & \langle\text{spl}\rangle \oplus\ldots\oplus \langle\text{spl}\rangle \mid & \text{(direct sum)} \\
 & & \langle\text{spl}\rangle \otimes\cdots\otimes \langle\text{spl}\rangle \mid & \text{(tensor product)} \\
 & & \mathrm{I}_n \otimes_k \langle\text{spl}\rangle \mid \mathrm{I}_n \otimes^k \langle\text{spl}\rangle \mid & \text{(overlapped tensor product)} \\
 & & \overline{\langle\text{spl}\rangle} \mid & \text{(conversion to real)} \\
 & & \ldots & \\
\langle\text{generic}\rangle & ::= & \mathrm{diag}(a_0,\ldots,a_{n-1}) \mid \ldots & \\
\langle\text{symbol}\rangle & ::= & \mathrm{I}_n \mid \mathrm{J}_n \mid \mathrm{L}^n_k \mid \mathrm{R}_\alpha \mid \mathrm{F}_2 \mid \ldots & \\
\langle\text{transform}\rangle & ::= & \mathbf{DFT}_n \mid \mathbf{WHT}_n \mid \mathbf{DCT\text{-}2}_n \mid \mathbf{Filt}_n\,(h[z]) \mid \ldots &
\end{array}
$$

Einige Definitionen:

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

$$A \otimes B = \begin{bmatrix} a_{k,\ell} B \end{bmatrix}, \quad \text{where } A = \begin{bmatrix} a_{k,\ell} \end{bmatrix}$$

$$\mathrm{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathrm{R}_\alpha = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix}$$

$$I_s \otimes_k A = \begin{bmatrix} \boxed{A} & & & \\ & \boxed{A} & & \\ & & \ddots & \\ & & & \boxed{A} \end{bmatrix}$$

$$I_s \otimes^k A = \begin{bmatrix} \boxed{A} & & & & \\ & \boxed{A} & & & \\ & & \boxed{A} & & \\ & & & \ddots & \\ & & & & \boxed{A} \end{bmatrix}$$

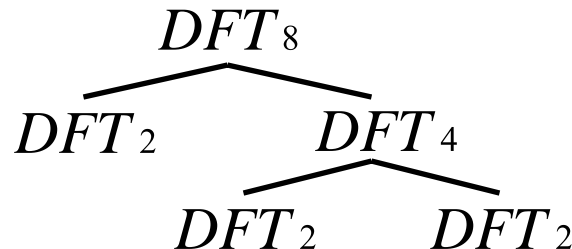
Algorithmen: Spiral's Terminologie

Transformation DFT_n parametrisierte Matrix

Regel $DFT_{nm} \rightarrow (DFT_n \otimes I_m) \cdot D \cdot (I_n \otimes DFT_m) \cdot P$

- Zerlegungsregel (teile-und-herrsche)
- Produkt dünnbesetzter Matrizen

Regelbaum



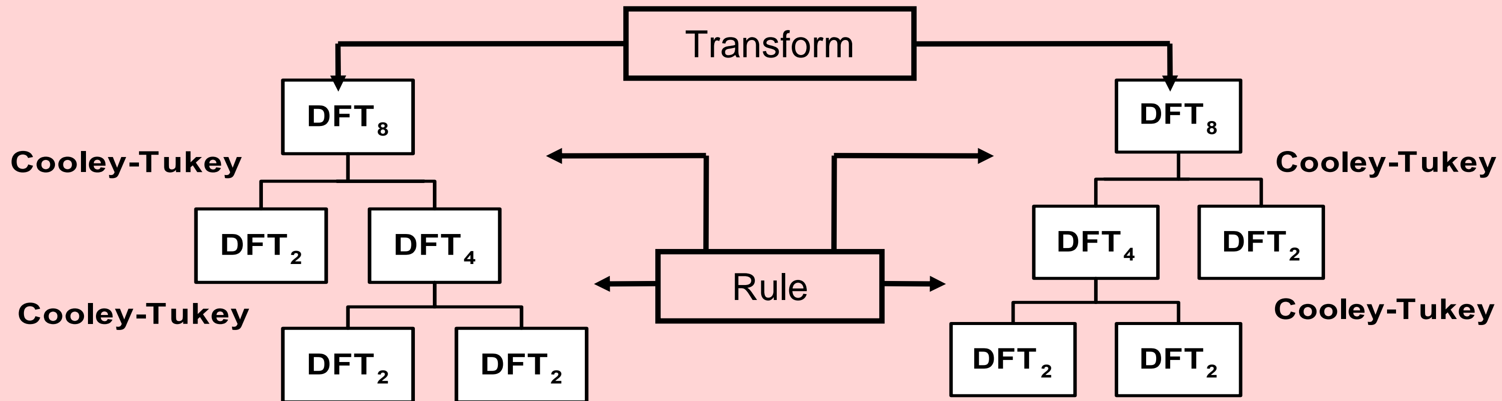
- rekursive Anwendung von Regeln
- definiert einen Algorithmus
- effiziente Darstellung
- einfache Manipulierung

Formel $DFT_8 = (F_2 \otimes I_4) \cdot D \cdot (I_2 \otimes (I_2 \otimes F_2 \cdots)) \cdot P$

- wenig Symbole
- definiert einen Algorithmus
- kann in Code übersetzt werden

Regelbäume = Formeln = Algorithmen

Regelbäume



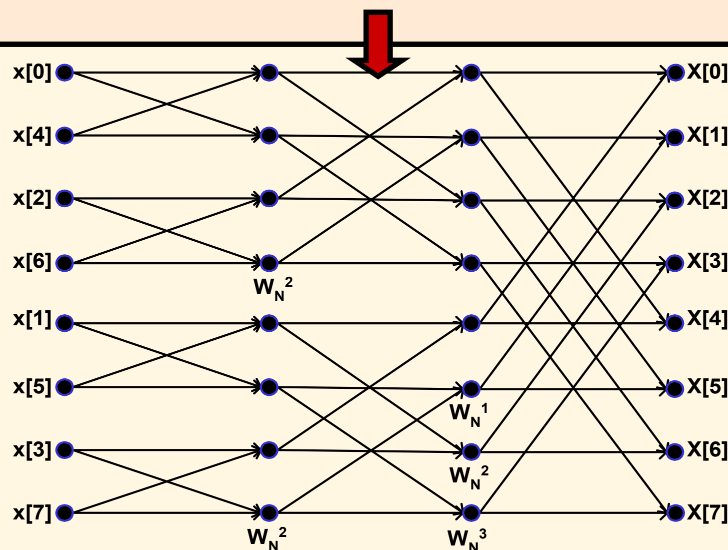
Formeln

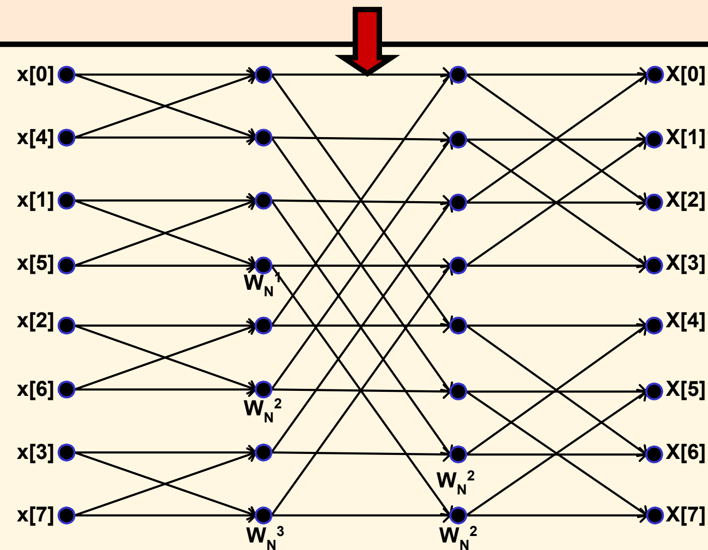
$$DFT_8 = (DFT_2 \otimes I_4) \cdot T_2^8 \cdot (I_2 \otimes DFT_4) \cdot L_4^8$$

$$DFT_8 = (DFT_4 \otimes I_2) \cdot T_4^8 \cdot (I_4 \otimes DFT_2) \cdot L_2^8$$

$$(DFT_2 \otimes I_2) \cdot T_2^4 \cdot (I_2 \otimes DFT_2) \cdot L_2^4$$

Algorithmen





Höhere Abstraktionsebene

Einige Transformationen

$$
\begin{aligned}
\mathbf{DCT\text{-}2}_n &= \left[\cos(k(2\ell+1)\pi/2n)\right]_{0\le k,\ell<n},\\
\mathbf{DCT\text{-}3}_n &= \mathbf{DCT\text{-}2}_n^T \quad \text{(transpose)},\\
\mathbf{DCT\text{-}4}_n &= \left[\cos((2k+1)(2\ell+1)\pi/4n)\right]_{0\le k,\ell<n},\\
\mathbf{IMDCT}_n &= \left[\cos((2k+1)(2\ell+1+n)\pi/4n)\right]_{0\le k<2n,0\le \ell<n},\\
\mathbf{RDFT}_n &= [r_{k\ell}]_{0\le k,\ell<n}, \quad r_{k\ell} = \begin{cases} \cos\frac{2\pi k\ell}{n}, & k \le \lfloor\frac{n}{2}\rfloor \\ -\sin\frac{2\pi k\ell}{n}, & k > \lfloor\frac{n}{2}\rfloor \end{cases},\\
\mathbf{WHT}_n &= \begin{bmatrix} \mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2} \\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2} \end{bmatrix}, \quad \mathbf{WHT}_2 = \mathbf{DFT}_2,\\
\mathbf{DHT} &= \left[\cos(2k\ell\pi/n) + \sin(2k\ell\pi/n)\right]_{0\le k,\ell<n}.
\end{aligned}
$$

Spiral “kennt” zur Zeit 36 Transformationen

Filter

$$h(z) = h_l z^{-l} + \cdots + h_0 z + \cdots + h_{-r} z^r$$

FIR Filtertransformation

$$\mathbf{Filt}_n(h(z)) = \begin{bmatrix} h_l & \cdots & h_{-r} & & & \\ & h_l & \cdots & h_{-r} & & \\ & & \ddots & & \ddots & \\ & & & h_l & \cdots & h_{-r} \end{bmatrix}$$

Faltungstransformation

$$\mathbf{Conv}_n(h(z)) = \begin{bmatrix} h_{-r} & & & \\ \vdots & \ddots & & \\ h_l & \cdots & h_{-r} & \\ & \ddots & & \ddots \\ & & h_l & \cdots & h_{-r} \\ & & & \ddots & \vdots \\ & & & & h_l \end{bmatrix}$$

Triviale Regel:

$$\mathbf{Filt}_n(h(z)) \rightarrow \mathrm{I}_s \otimes_{l+r} \mathbf{Filt}_1(h(z))$$

Zirkulante

$$\mathbf{C}_n(h(z)) = \left[h_{(i-j) \bmod n}\right]_{n \times n}$$

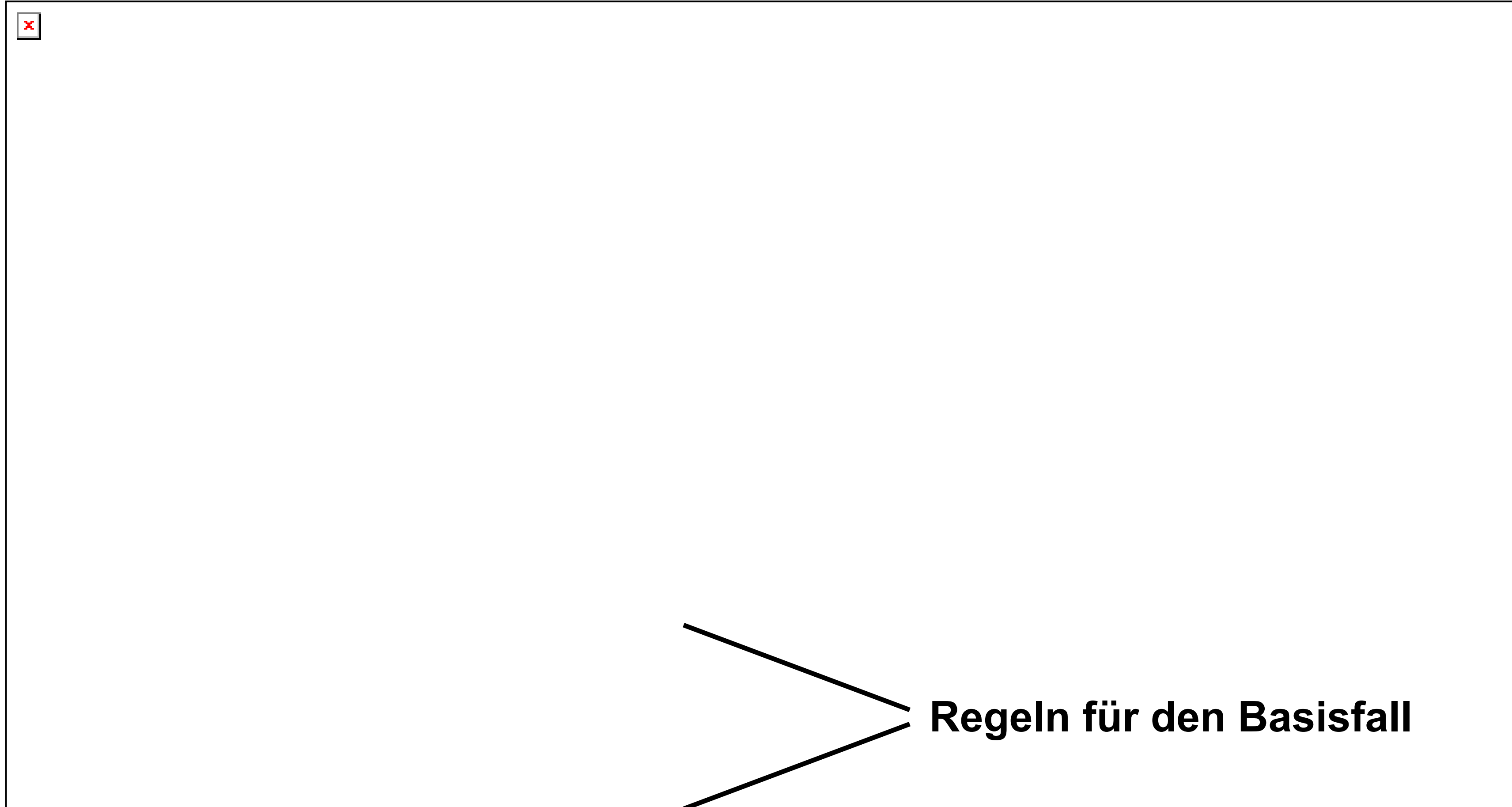
$$\mathbf{C}_n(h(z)) = \mathbf{Filt}_n^{\mathrm{per}}(h(z))$$

Toeplitz

$$\mathbf{T}_n(h(z)) = \left[h_{i-j}\right]_{n \times n}$$

$$\mathbf{T}_n(h(z)) = \mathbf{Filt}_n^{\mathrm{zero}}(h(z))$$

Einige Zerlegungsregeln



Spiral “kennt” 100+ Regeln

Regeln für Filter

“Blocken” Regel

$$\mathrm{Filt}_n(h(z)) \rightarrow \mathrm{I}_{\lfloor \frac{n}{b} \rfloor} \otimes_{l+r} \left(\left[\;\; | \;\; \right]_{i=0}^{\left\lceil \frac{l+r}{b} \right\rceil} \mathrm{T}_b\left(h(z) z^{l-ib}\right) \oplus^k \mathrm{T}_k \left(h(z) z^{l-\left\lceil \frac{l+r}{b} \right\rceil b-k} \right) \right)$$

Karatsubaregel

$$\mathrm{Filt}_n(h(z)) \rightarrow \mathsf{L}^n_{\frac{n}{2}} \, \mathrm{Filt}_{\frac{n}{2}} \left(\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \right) \cdot$$

$$\mathrm{Filt}_{\frac{n}{2}} \left(\begin{bmatrix} h_0(z) & & \\ & h_1(z) & \\ & & h_0(z)+h_1(z) \end{bmatrix} \right) \cdot \mathrm{Filt}_{\frac{n}{2}+\frac{r+l-1}{2}} \left(\begin{bmatrix} 1 & -1 \\ z & -1 \\ 0 & 1 \end{bmatrix} \right) \cdot \mathsf{L}_2^{n+r+l}$$

“Einbetten in Zirkulante” Regel

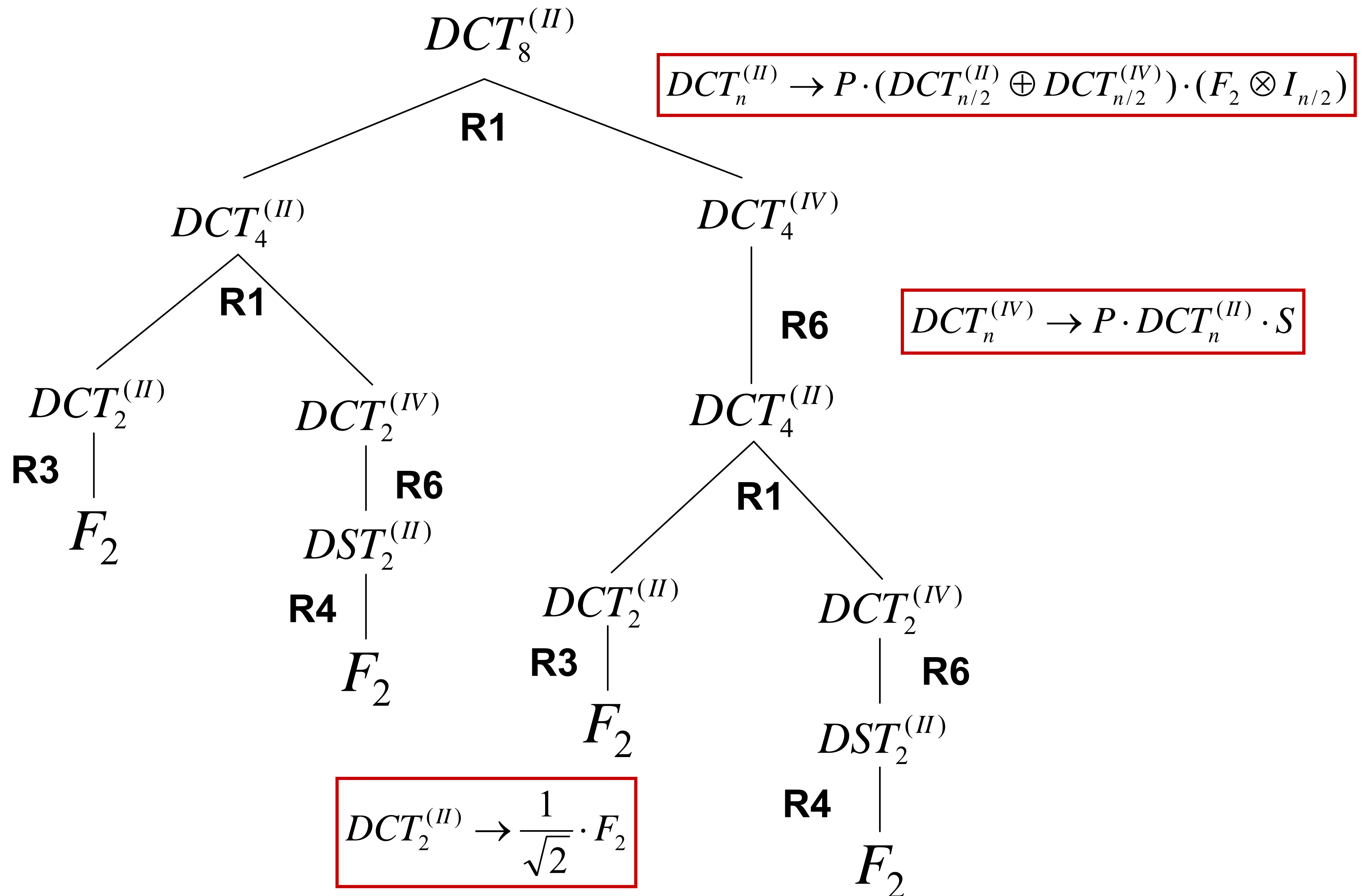
$$\mathrm{Filt}_n(h(z)) \rightarrow \mathsf{R}^{\mathsf{zero}}_{n,l,r} \cdot \mathrm{C}_{n+l+r}(h(z))$$

“Zirkulante über DFT” Regel

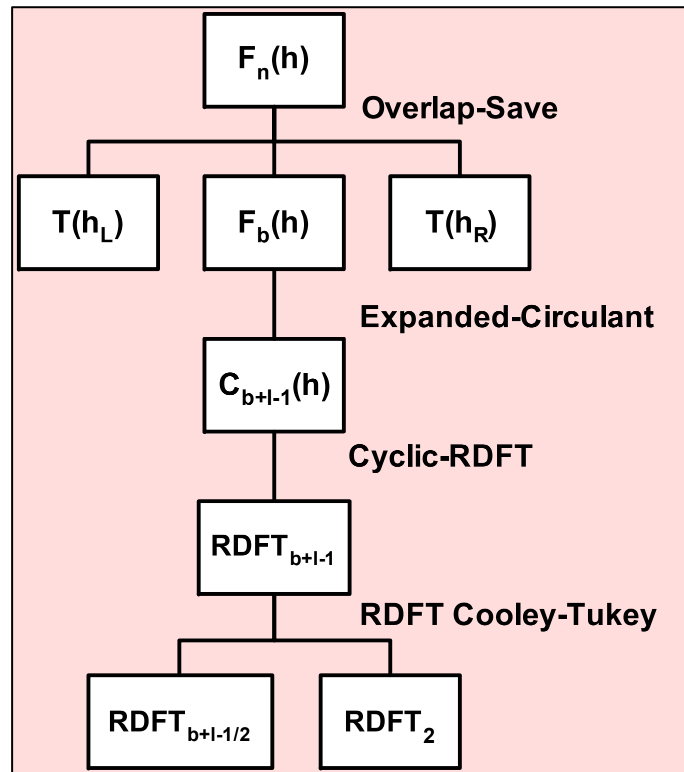
$$\mathrm{C}_n(h(z)) \rightarrow \mathrm{RDFT}_n^{-1} \cdot X(\hat{\mathrm{h}}) \cdot \mathrm{RDFT}_n,$$

$$\hat{\mathrm{h}} = \mathrm{RDFT}_n \cdot \mathrm{h}$$

Regelbaumbeispiel (I)

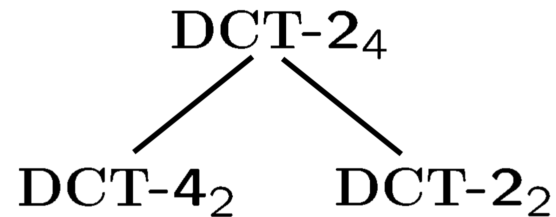


Regelbaumbeispiel (II)



Formelbeispiel

Regelbaum:



Formel:

$$\mathrm{DCT\text{-}2}_4 = \mathrm{L}^4_2(\mathrm{diag}(1, 1/\sqrt{2})\, \mathrm{F}_2 \oplus \mathrm{J}_2\, \mathrm{R}_{13\pi/8})(\mathrm{F}_2 \otimes \mathrm{I}_2)(\mathrm{I}_2 \oplus \mathrm{J}_2)$$

Algorithmen- (Formel) generierung

- **Gegeben eine Transformation**
  - Wende Zerlegungsregeln an bis alle Transformationen expandiert sind
  - Die Auswahl der Regeln produziert eine exponentielle Menge von Algorithmen
    - ungefähr gleich bzgl. arithmetischer Kosten (nicht für Filter)
    - unterschiedlicher Datenfluss

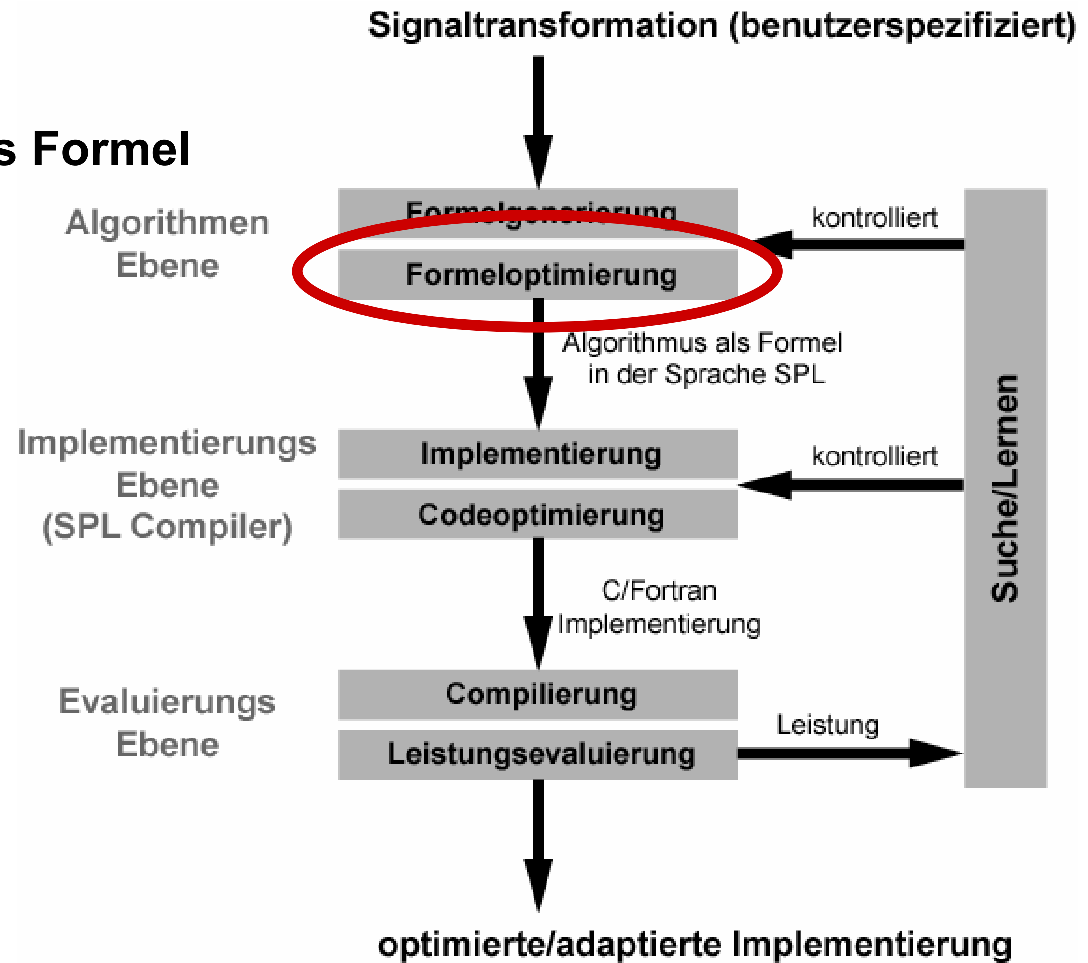
exponentiell

k	# DFTs, Größe 2^k	# DCT IV, Größe 2^k
1	1	1
2	6	10
3	40	126
4	296	31242
5	27744	1924443362
6	162570361280	7343815121631354242
7	~1.01 • 10^27	~1.07 • 10^38
8	~2.31 • 10^61	~2.30 • 10^76
9	~2.86 • 10^133	~1.06 • 10^153

Algorithmen- (Formel) optimierung

Eingabe:
Schneller Algorithmus als Formel

Ausgabe:
Optimierte Formel



Motivation: Schleifenverschmelzung

$(I_4 \otimes F_2)\, L^8_4$

direkte Codegener.

```
void I4xF2_L84(double *y, double *x) {
   double t[8];
   for (int i=0; i<8; i++)
      t[i==7 ? 7 : (i*4)%7] = x[i];
   for (int i=0; i<4; i++){
      y[2*i] = t[2*i] + t[2*i+1];
      y[2*i+1] = t[2*i] - t[2*i+1];
   }
}
```

kein Compiler kann das

```
void I4xF2_L84(double *y, double *x) {
   for (int j=0; j<4; j++){
      y[2*j] = x[j] + x[j+4];
      y[2*j+1] = x[j] - x[j+4];
   }
}
```

Lösung:

Formel-manipulierung

$$\sum_{j=0}^{3} S_{(j)_4 \otimes \imath_2}\, F_2\, G_{\imath_2 \otimes (j)_4}$$

Optimierung auf der Formelebene

- **Hauptziele:**
  - Verschmelzung iterative Schritte (Schleifen)
  - Strukturmanipulierung für Vektorinstruktionen
- **Überwindet Compilerlimitierungen**
- **Möglich durch Σ-SPL und mathematische Regeln**
- **Implementierung durch mehrfache Ebenen von Rewritingsystemen**
- **Gibt SPIRAL “Algorithmenwissen”**

Σ-SPL: Generating Functions

- **Symbolische Funktionen**

Identität

$$\imath_n : \begin{cases} \{0,\ldots,n-1\} \to \{0,\ldots,n-1\} \\ i \mapsto i \end{cases}$$

Basis Funktion

$$(j)_n : \begin{cases} \{0\} \to \{0,\ldots,n-1\} \\ i \mapsto j \end{cases}$$

Stride Funktion

$$\ell_m^{mn} : \begin{cases} \{0,\ldots,mn-1\} \to \{0,\ldots,mn-1\} \\ i \mapsto \begin{cases} (im) \bmod (mn-1) & \text{if } 0 \le i < mn-1 \\ mn-1 & \text{if } i = mn-1 \end{cases} \end{cases}$$

Twiddle Funktion

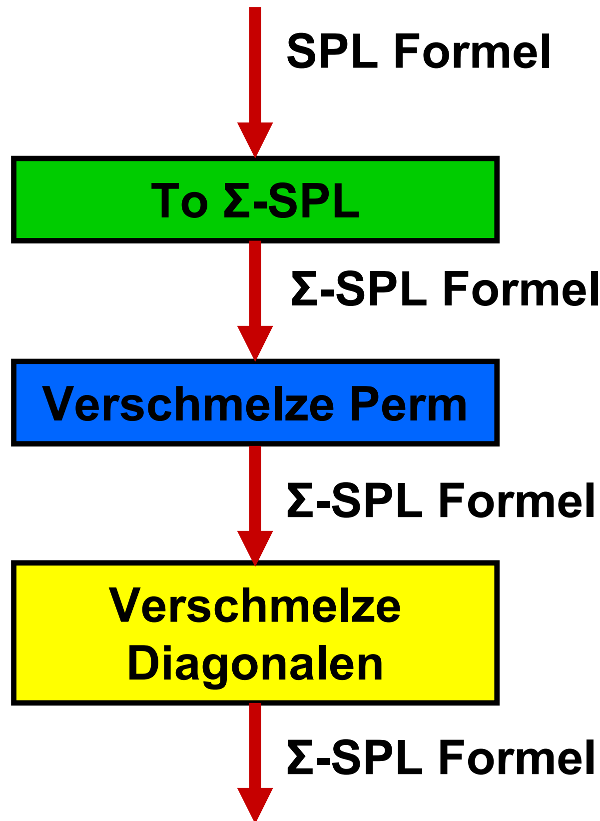
$$\mathrm{t}_n^{mn} : \begin{cases} \{0,\ldots,mn-1\} \to \mathbb{C} \\ i \mapsto \omega_{mn}^{(i \bmod n)\left\lfloor \frac{i}{n} \right\rfloor} \end{cases}$$

- **Funktionsoperatoren**

$$(j)_m \otimes \imath_n : \begin{cases} \{0,\ldots,n-1\} \to \{0,\ldots,mn-1\} \\ i \mapsto jn+i \end{cases}$$

$$\imath_n \otimes (j)_m : \begin{cases} \{0,\ldots,n-1\} \to \{0,\ldots,mn-1\} \\ i \mapsto j+im \end{cases}$$

Beispiel: Schleifenoptimierung



$$\mathsf{T}_n^{mn}(\mathrm{I}_m \otimes C_n)\,\mathsf{L}_m^{mn}$$

$$\mathrm{diag}\left(\mathrm{t}_n^{mn}\right)\left(\sum_{j=0}^{n-1} \mathsf{S}_{(j)_m\otimes\imath_n}\, C_n\, \mathsf{G}_{(j)_m\otimes\imath_n}\right)\mathrm{perm}\left(\ell_m^{mn}\right)$$

$$\mathrm{diag}\left(\mathrm{t}_n^{mn}\right)\left(\sum_{j=0}^{n-1} \mathsf{S}_{(j)_m\otimes\imath_n}\, C_n\, \mathsf{G}_{\imath_n\otimes(j)_m}\right)$$

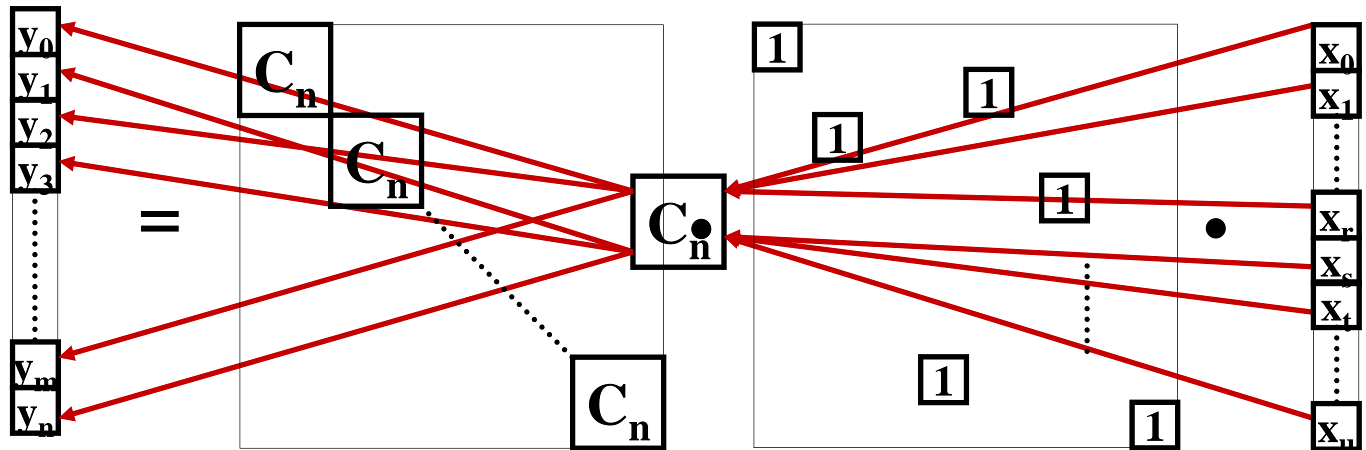
$$\sum_{j=0}^{n-1} \mathsf{S}_{(j)_m\otimes\imath_n}\, \mathrm{diag}\left(\mathrm{t}_n^{mn}\circ\left((j)_m\otimes\imath_n\right)\right) C_n\, \mathsf{G}_{\imath_n\otimes(j)_m}$$

Regeln: $\mathsf{G}_r\,\mathrm{perm}\left(\pi\right) = \mathsf{G}_{\pi\circ r}\,, \quad \ell_m^{mn}\circ\left((j)_m\otimes\imath_n\right) = \imath_n\otimes(j)_m$

$\mathrm{diag}\left(f\right)\mathsf{S}_w = \mathsf{S}_w\,\mathrm{diag}\left(f\circ w\right)$

Visualisierung

$$(\mathrm{I}_m \otimes C_n)\, \mathrm{L}^{mn}_m = \sum_{j=0}^{n-1} \mathrm{S}_{(j)_m \otimes \imath_n}\, C_n\, \mathrm{G}_{\imath_n \otimes (j)_m}$$



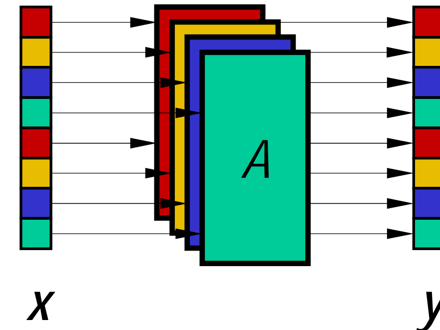
Iteration

Vektorcodegenerierung von SPL Formeln

Natürlich vektorisierbar

$$A \otimes I_4$$

Vektorlänge



Verallgemeinerung (auch komplett vektorisierbar):

$$\prod_{i=1}^{k} P_i D_i (A_i \otimes I_\upsilon) E_i Q_i$$

P_i, Q_i	**Permutationen**
D_i, E_i	**Diagonalmatrizen**
A_i	**beliebige Formel**
ν	**Vektorlänge**

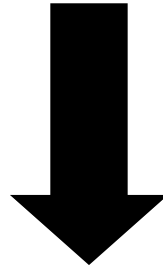
Vektorisierung in zwei Schritten:

1. Formelmanipulierung
2. Codegenerierung (Vektorcode + C code)

Beispiel DFT

Standard FFT

$$(\mathbf{DFT}_k \otimes \mathrm{I}_m)\, \mathsf{T}^n_m (\mathrm{I}_k \otimes \mathbf{DFT}_m)\, \mathsf{L}^n_k$$



Formelmanipulierung

$$\left(\mathrm{I}_{\frac{mn}{\nu}} \otimes \mathsf{L}^{2\nu}_{\nu}\right)\left(\overline{\mathbf{DFT}_m \otimes \mathrm{I}_{\frac{n}{\nu}}} \otimes \mathrm{I}_{\nu}\right) \overline{\mathsf{T}}'^{mn}_{n} \left(\mathrm{I}_{\frac{m}{\nu}} \otimes \left(\mathsf{L}^{2n}_{\nu} \otimes \mathrm{I}_{\nu}\right)\left(\mathrm{I}_{\frac{2n}{\nu}} \otimes \mathsf{L}^{\nu^2}_{\nu}\right)\left(\overline{\mathbf{DFT}_n} \otimes \mathrm{I}_{\nu}\right)\right)\left(\mathsf{L}^{\frac{mn}{\nu}}_{\frac{m}{\nu}} \otimes \mathsf{L}^{2\nu}_{2}\right)$$

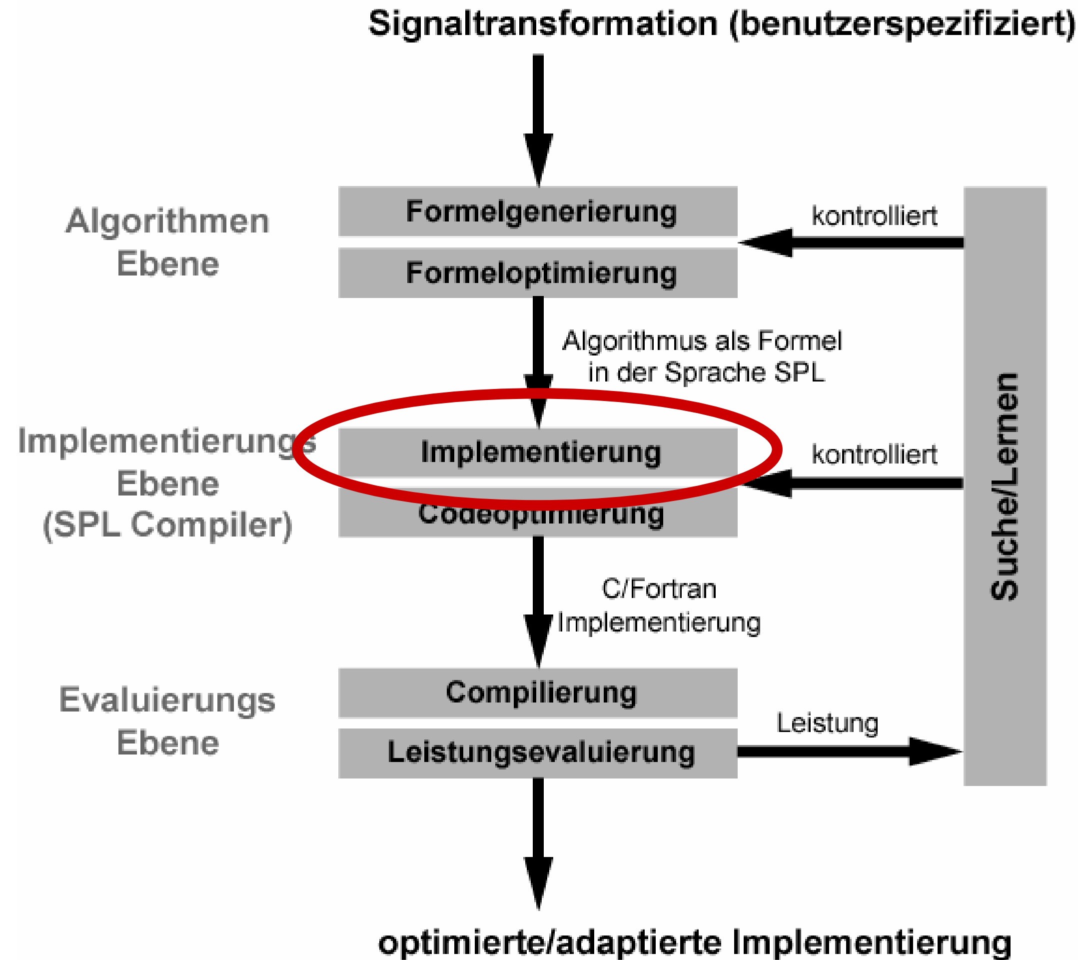
Vektor FFT für ν-weg Vektorinstruktionen

Optimierter Algorithmus (Formel) zu Code

Eingabe:
Optimierte Formel

Ausgabe:
Code

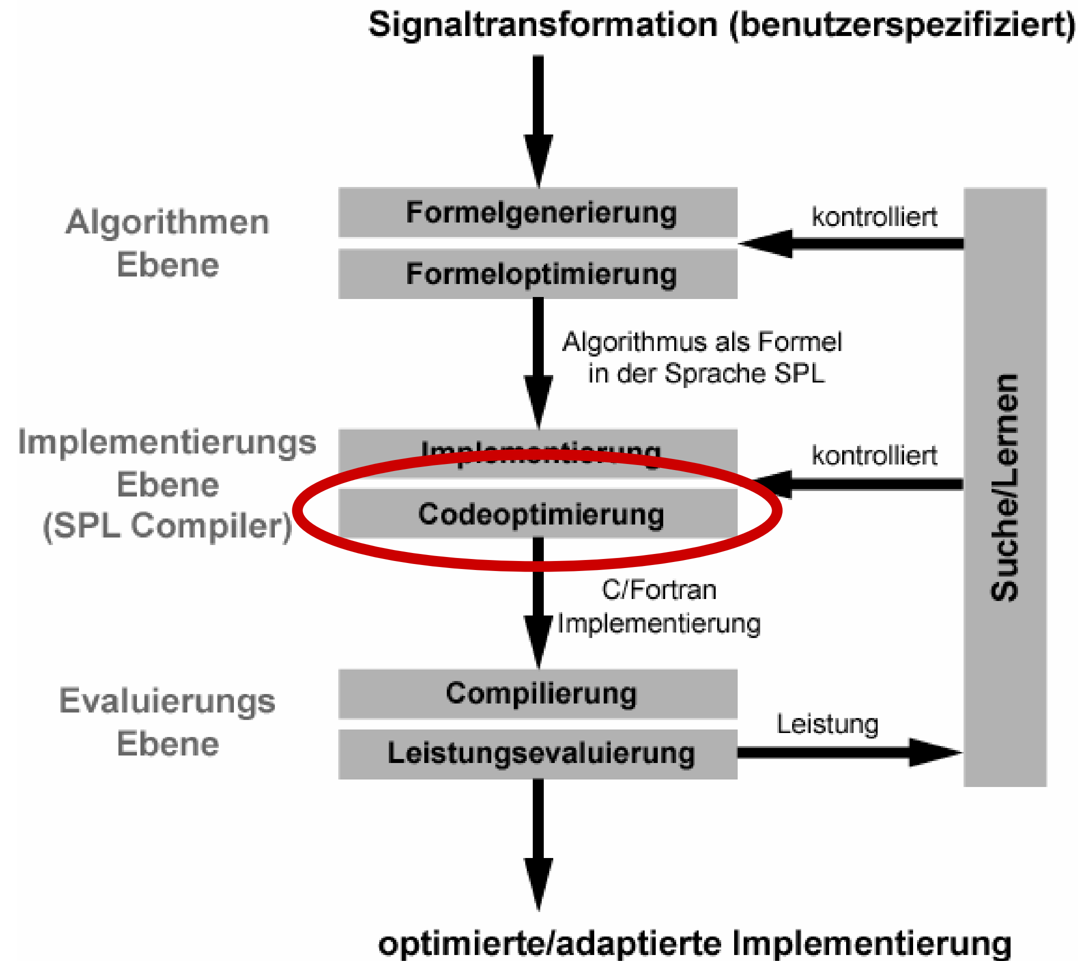
Einfach



Codeoptimierung

Eingabe:
Code

Ausgabe:
Optimierter Code



Optimierungen auf der Codeebene

- Vorberechnung von Konstanten
- Schleifenaufrollen (kontrolliert vom Suchmodul)
- Inlining der Konstanten
- SSA code, Array Skalarisierung, CSE
- Codescheduling für Lokalität (optional)
- Umwandlung in FMA Code (optional)
- Umwandlung in Fixpunktcode (optional)

- Am Ende: Unparsing nach C (or Fortran)

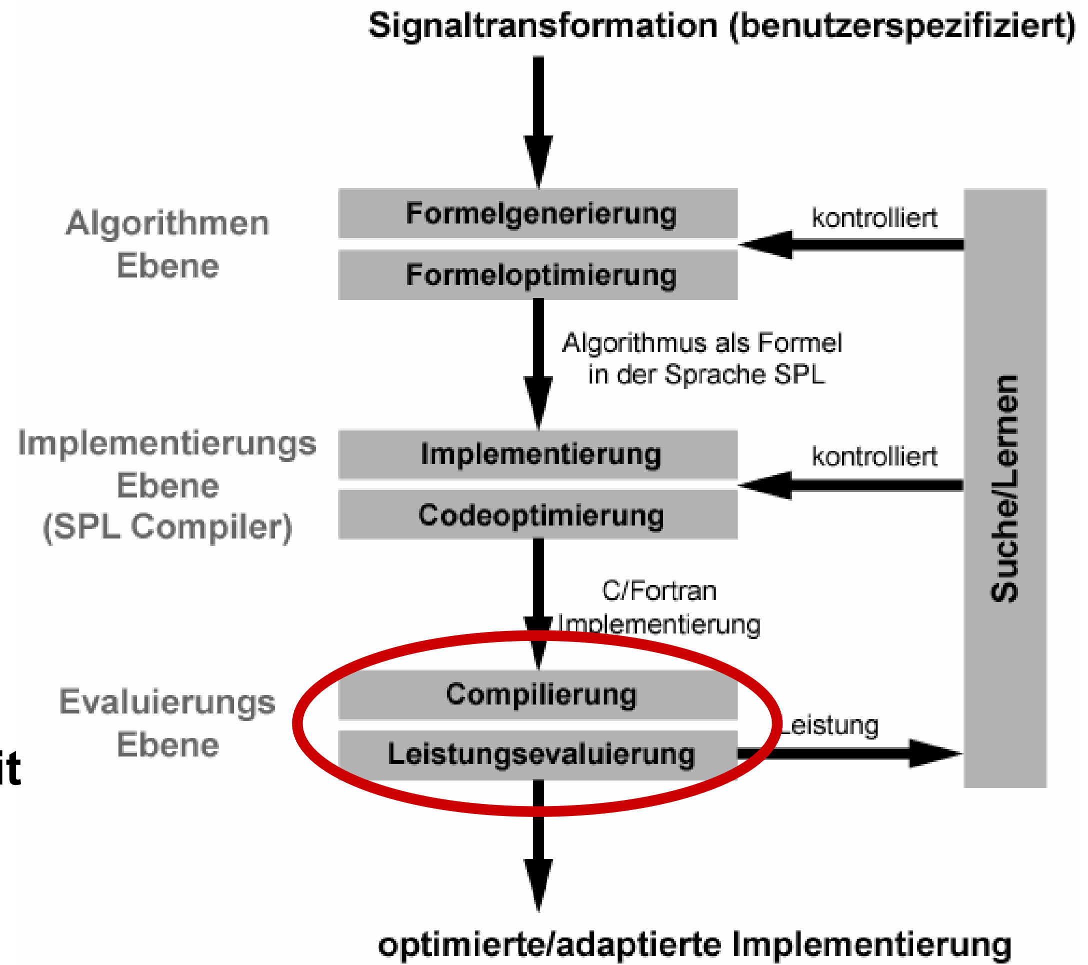
Codeevaluierung

Eingabe:
Optimierter C code

Ausgabe:
Leistungszahl

Klar

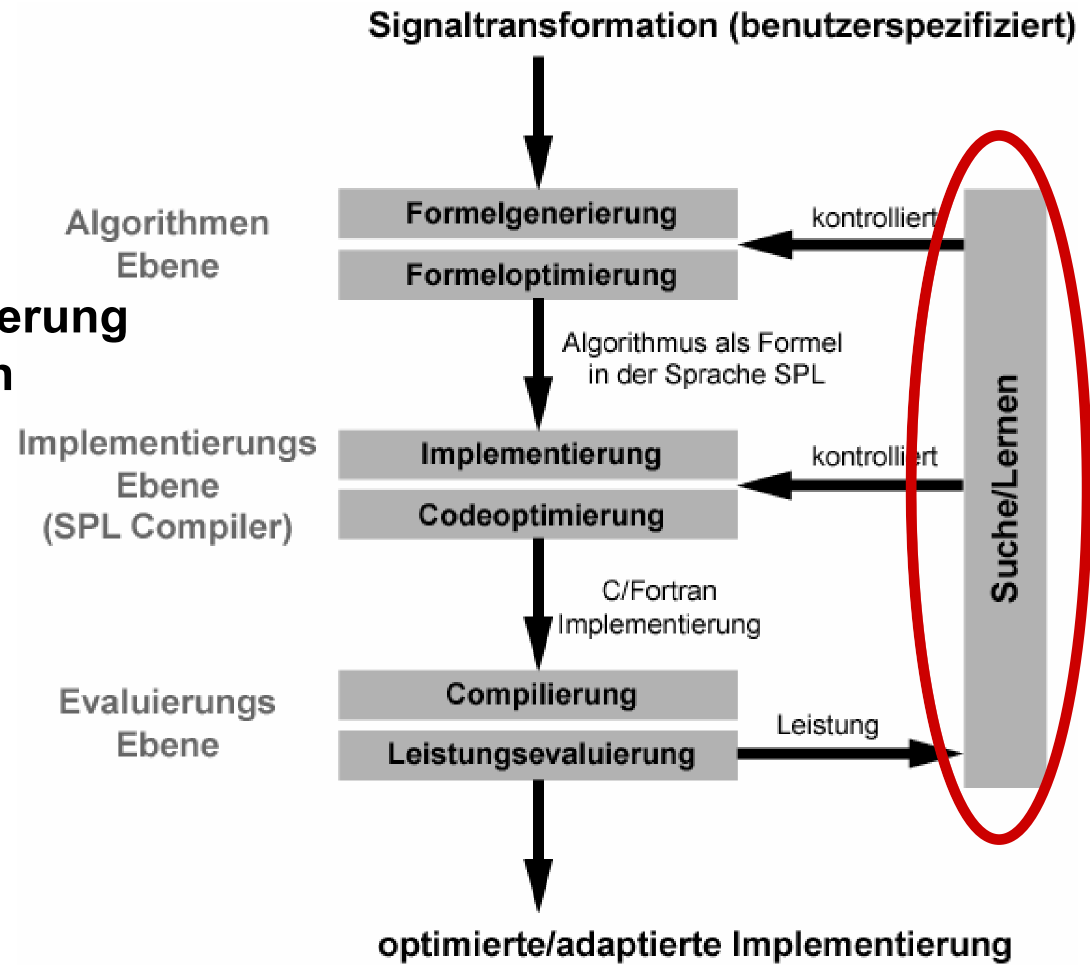
Beispiele:
- Laufzeit
- Numerische Genauigkeit
- Arithmetische Kosten



Search (Learning) for the Best

Eingabe:
 Leistungszahl

Ausgabe:
 Kontrolle der Formelgenerierung
 Kontrolle der Codeoptionen

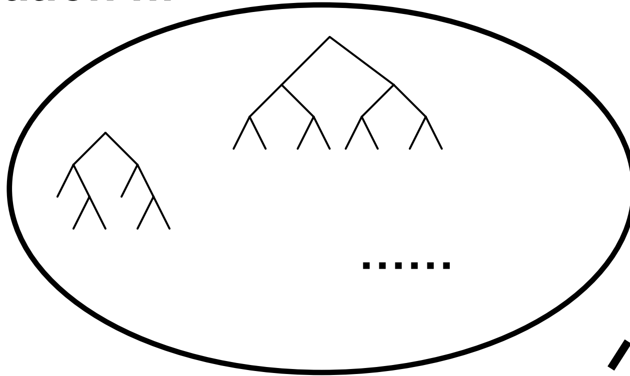


Suche

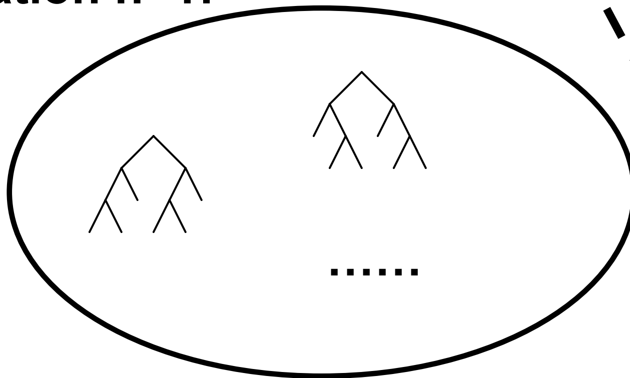
- **Suche über:**
  - Algorithmische Optionen (Auswahl der Zerlegungsregeln)
  - Codeoptionen (Grad des Schleifenaufrollens)
- **Suche arbeitet mit der Regelbaumdarstellung eines Algorithmus**
  - transformationsunabhängig
  - effizient
- **Suchmethoden**
  - Vollständige Suche
  - Dynamisches Programmieren (DP)
  - Zufällige Suche
  - Hill Climbing
  - STEER (ein evolutionärer Algorithmus)

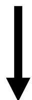
STEER

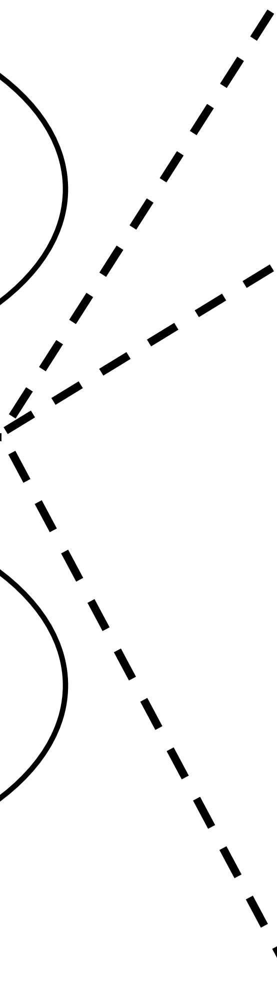
Population n:



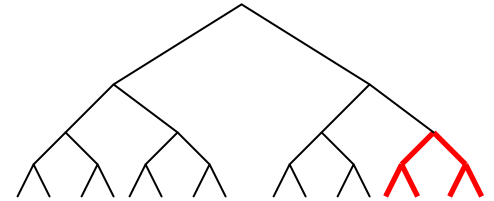
Population n+1:





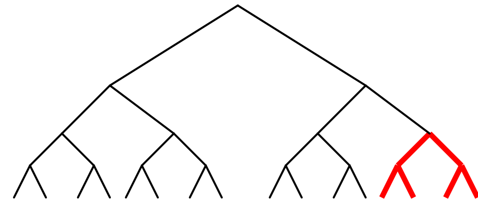


Mutation



expandiere anders

Kreuzung



tausche Expansionen

Überleben des Fittesten

(Maschinen-) Lernen

- **Vorgehensweise:**
  - Generiere eine Menge von (z.B. 1000) Algorithmen und deren Laufzeiten (für eine Transformation und Größe); repräsentiere den Algorithmus durch Merkmale
  - Von diesen Daten (Paare Merkmale/Laufzeiten), lerne eine Menge von Algorithmusdesignregeln
  - Von dieser Menge (dem Wissen), generiere schnelle Algorithmen (Theorie der Markov Entscheidungsprozesse)

- **Evaluierung:**
  - Getestet für die WHT und DFT
  - Von den generierten Daten für eine Größe (2^15) war es möglich, die besten Algorithmen für mehrere Größen (2^12-2^18) zu erzeugen

Bryan Singer and Manuela Veloso
Learning to Construct Fast Signal Processing Implementations
Journal of Machine Learning Research, 2002, Vol. 3, pp. 887-919

Experimente mit SPIRAL

(kurze Demo ?)

Wie evaluiert man Code?

- **Vergleiche gegen die beste erhältliche Software**
  - Stelle sicher daß die gleiche Zeitmessmethode verwendet wird

- **Berechne die Leistung (in MFLOPS)**

$$MFLOPS = Anzahl\ Operationen/(10^6 \cdot Laufzeit\ [s])$$

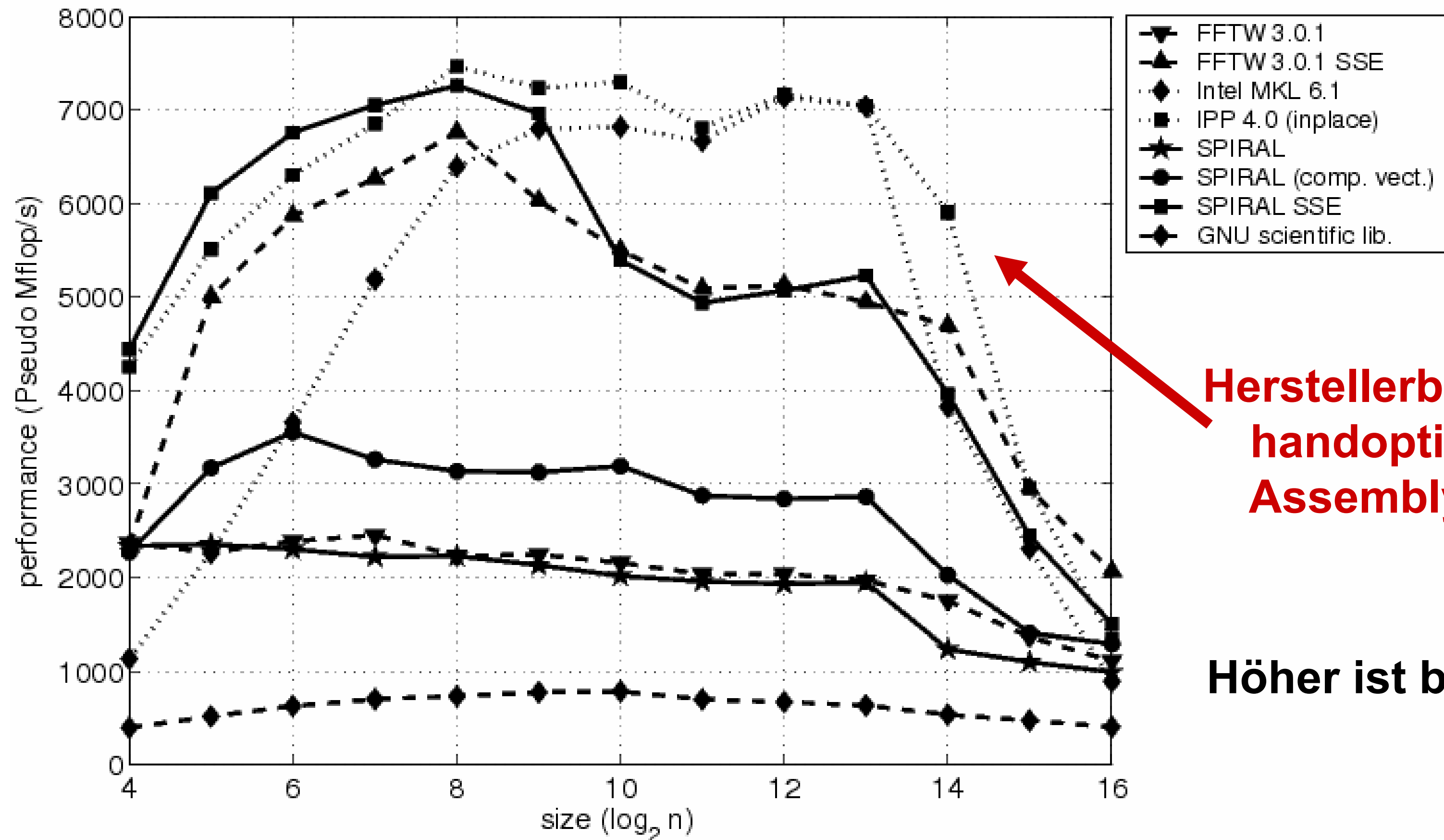
und vergleiche mit Maximalleistung der Maschine

  - Erfordert die Berechnung/Messung der Anzahl der Operationen
  - Maximalleistung aus dem Manual oder die richtigen Leute fragen
  - Vorsicht: Code mit höherer Leistung kann langsamer sein

Benchmarks

Benchmark: DFT, 2er-Potenzen

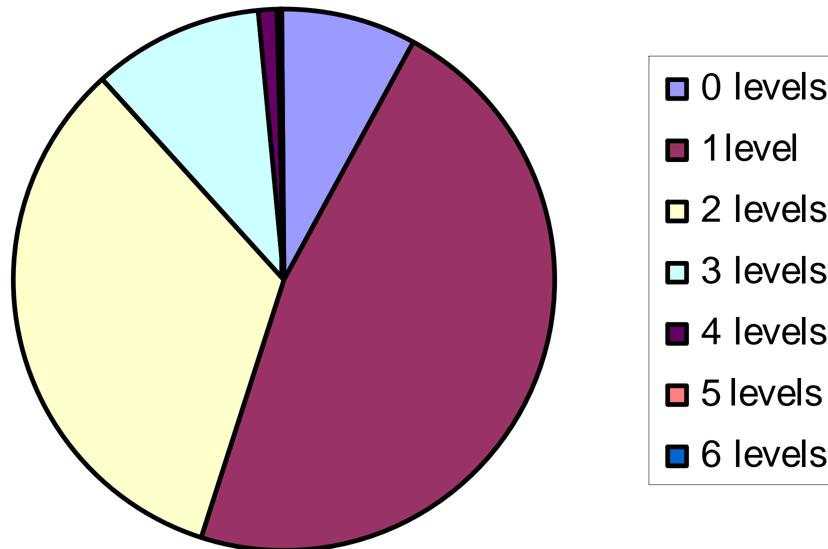
P4, 3.2 GHz, icc 8.0



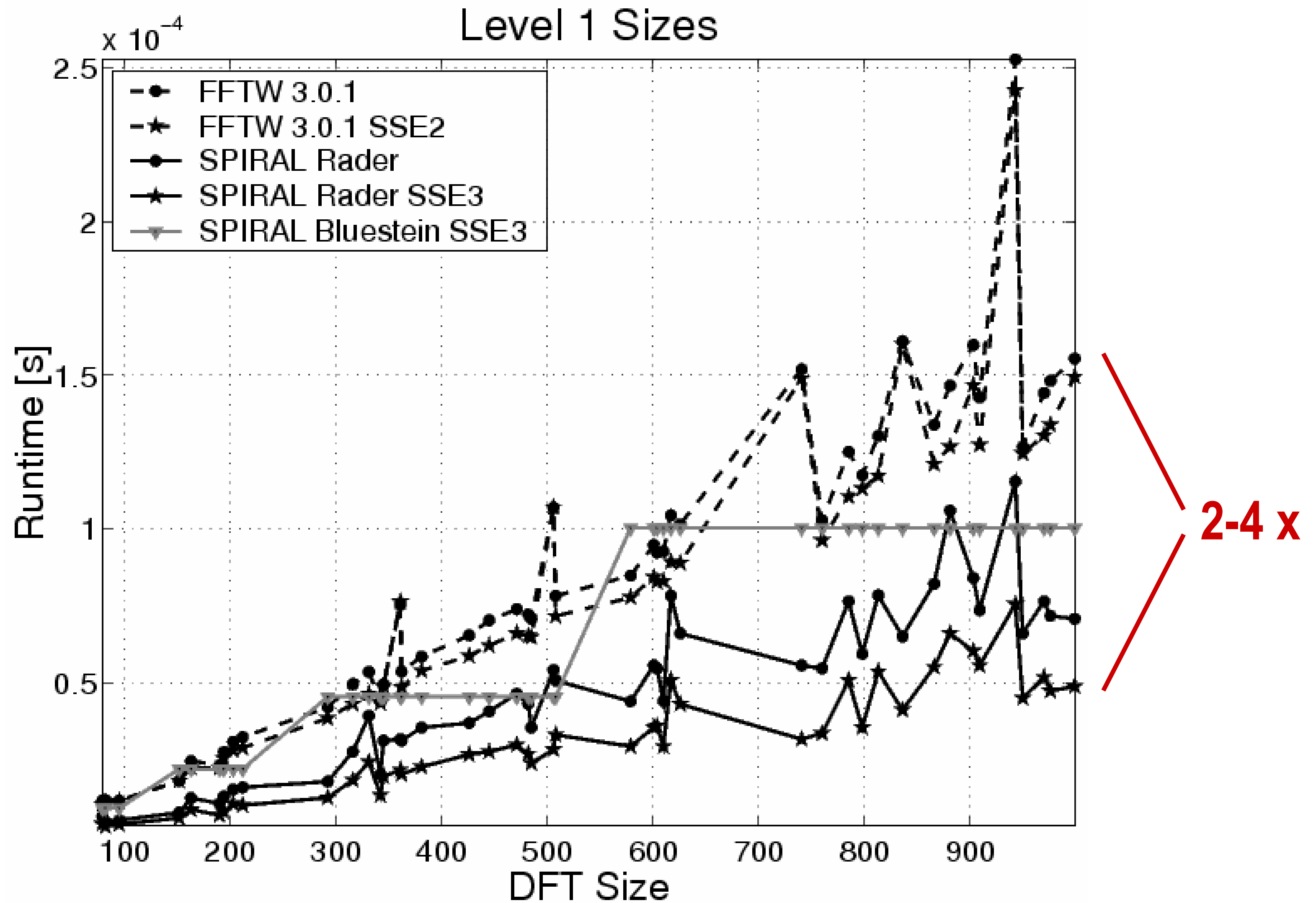
Einfache Genauigkeit

Benchmark: DFT, Andere Größen

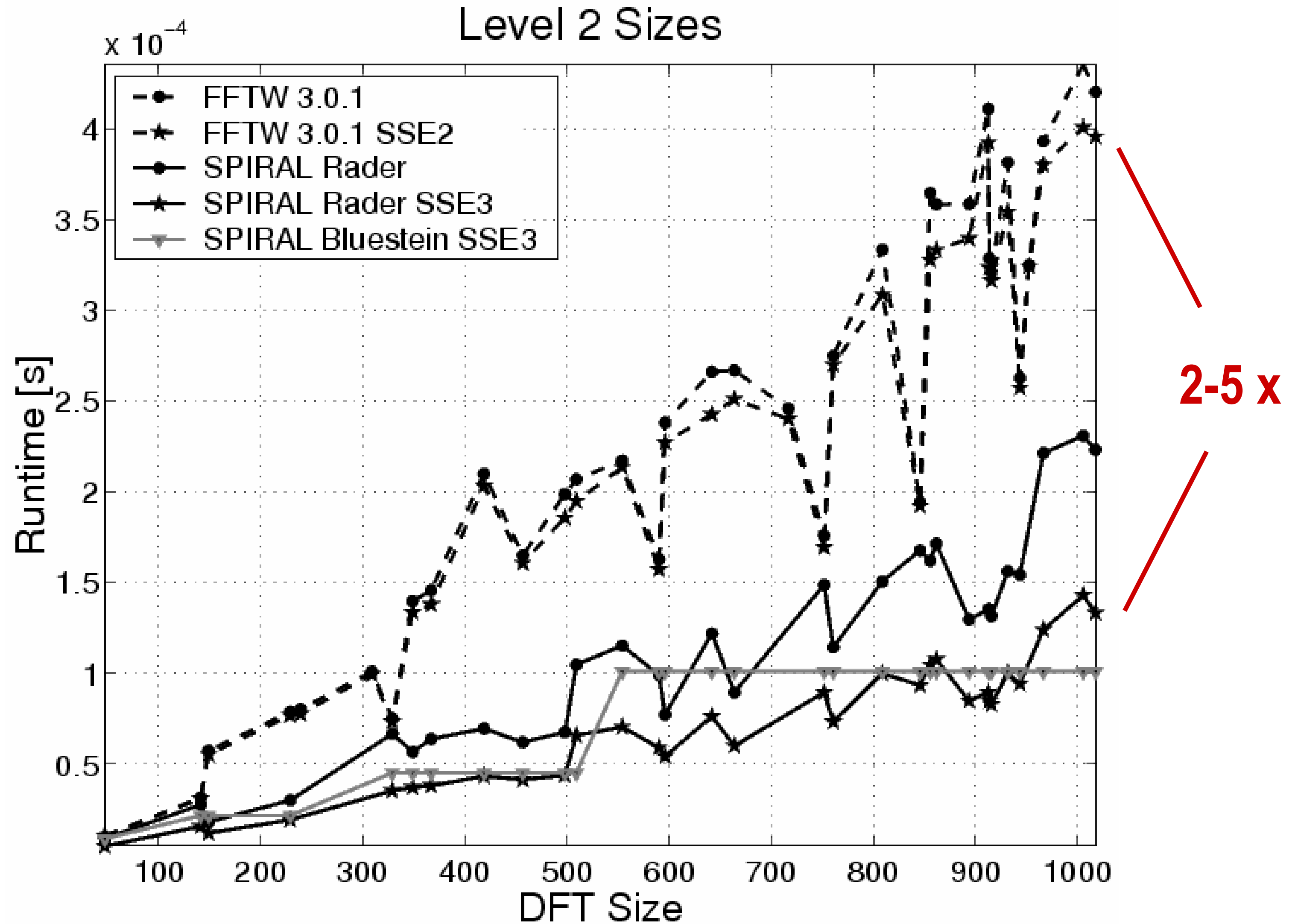
- Teile Größen ein nach Anzahl von nötigen Raderschritten
- n < 8192



Benchmark: DFT, Level 1 Größen

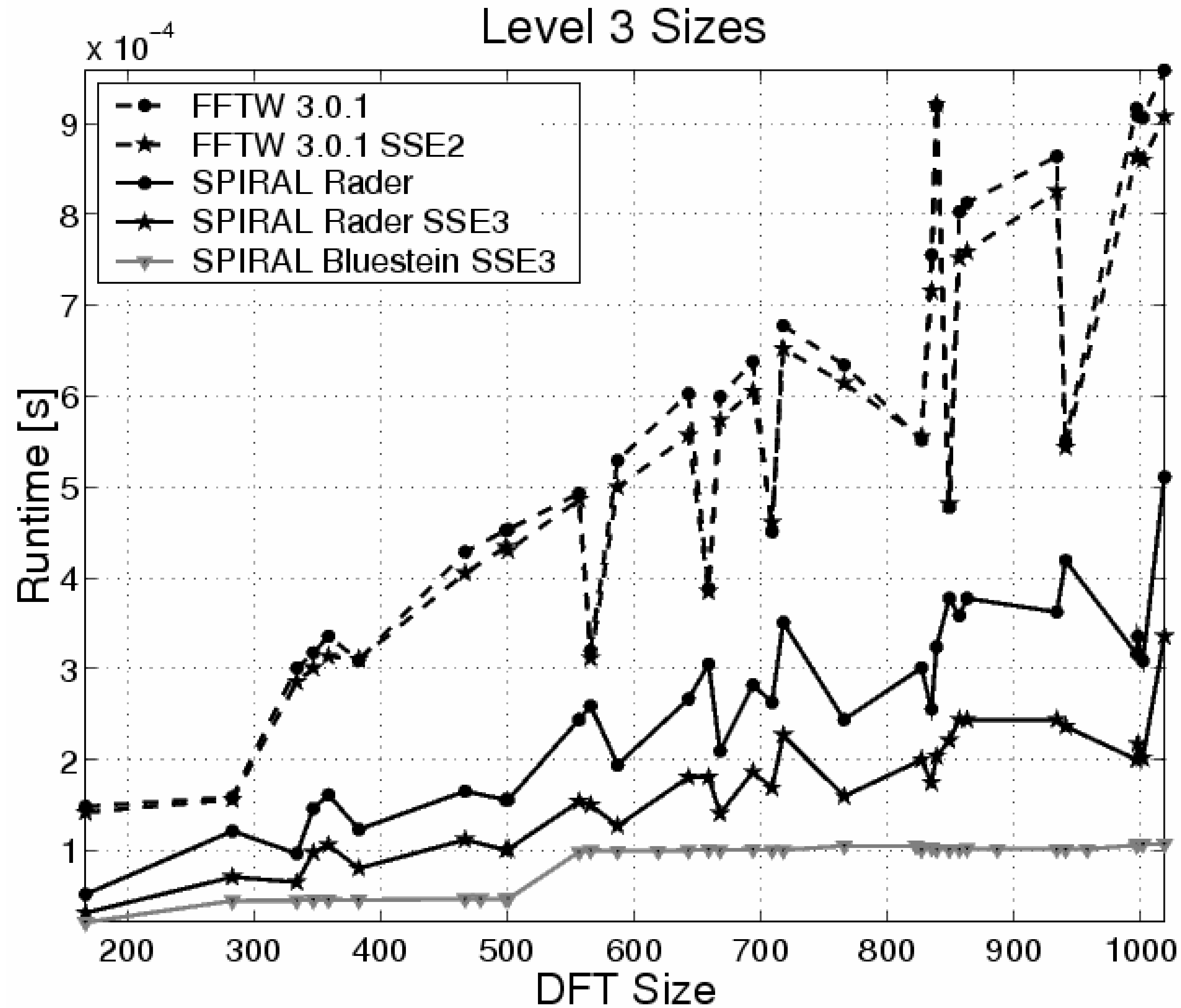


Benchmark: DFT, Level 2 Größen



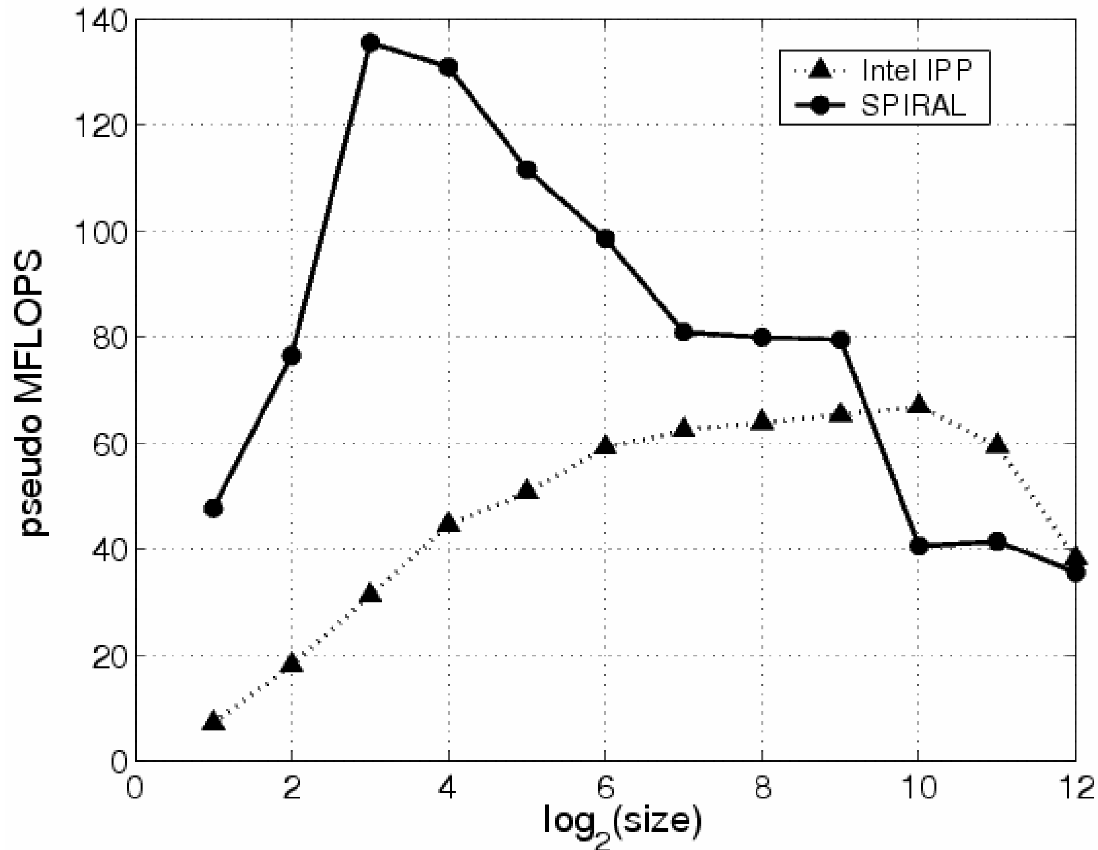
2-5 x

Benchmark: DFT, Level 3 Größen



4-9 x

Benchmark: Fixpunkt DFT, IPAQ



IPAQ
Xscale arch.
400 MHz
hat nur Fixpunkt

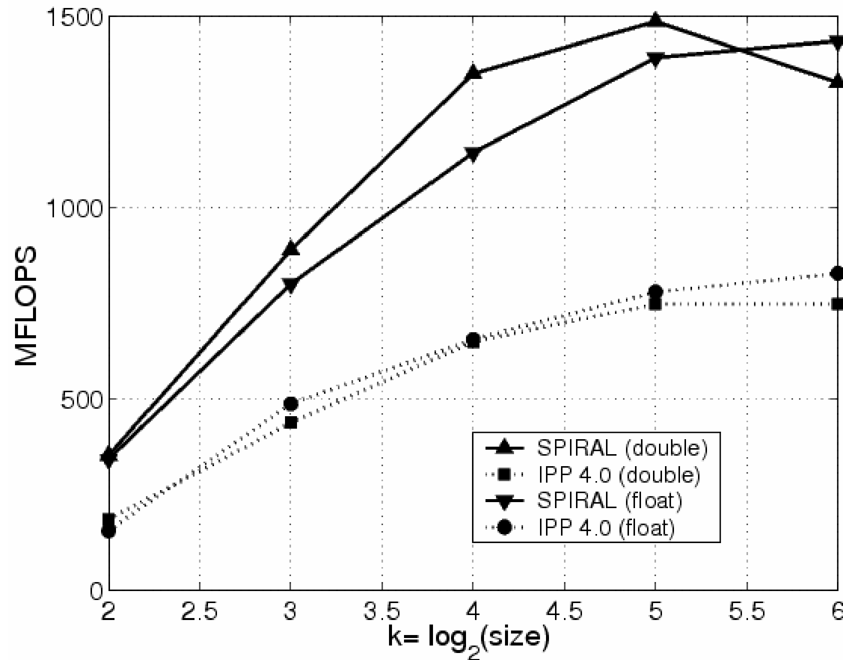
Höher ist besser

Intel hat sich weniger bemüht?

Benchmark: DCT

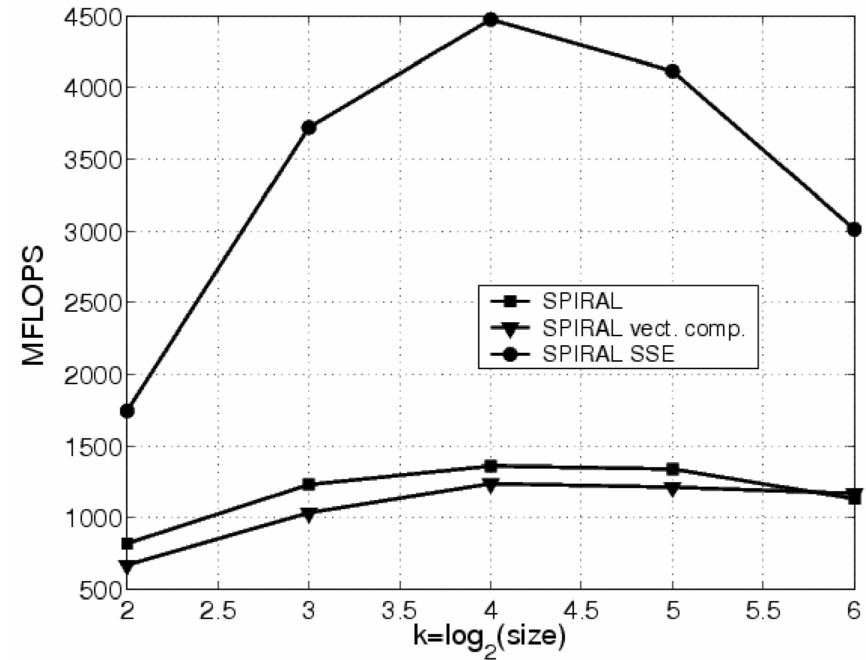
P4, 3.2 GHz, icc 8.0

1-D DCT



Scalar code

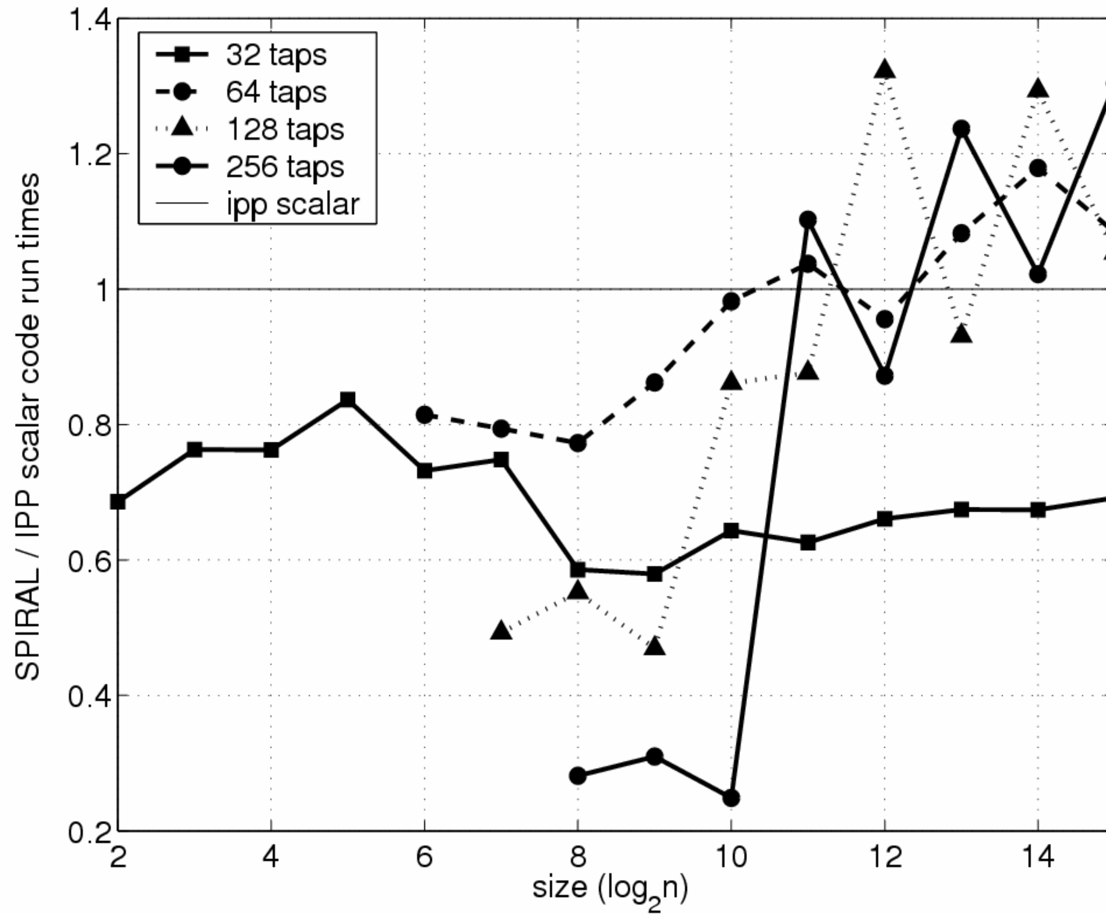
2-D DCT



Scalar vs. SSE code

- Faktor 2 gegenüber Herstellerbibliothek
- Ein weiterer Faktor 3 mit Vektorinstruktionen

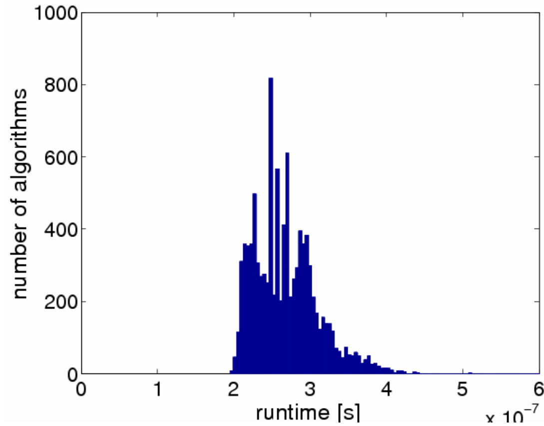
Benchmark: Filter



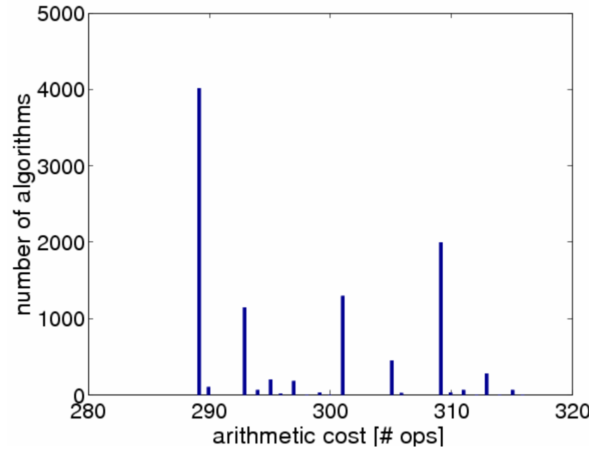
Lehrreiche Experimente

Laufzeitverteilung: DCT, size 32
Histogramme, 10,000 Algorithmen

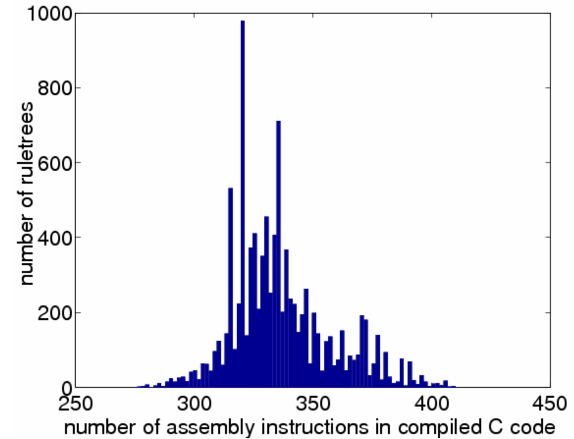
P4, 3.2 GHz, gcc



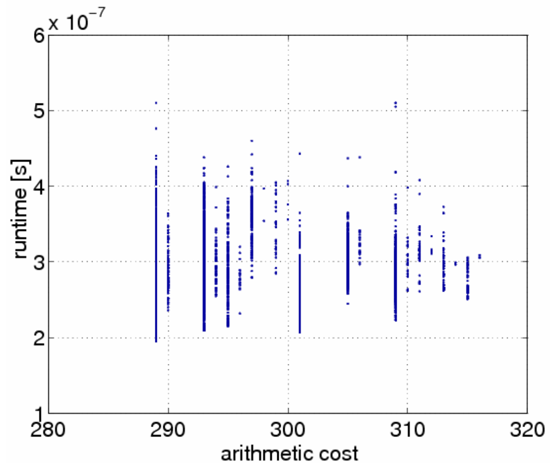
Laufzeit: x2



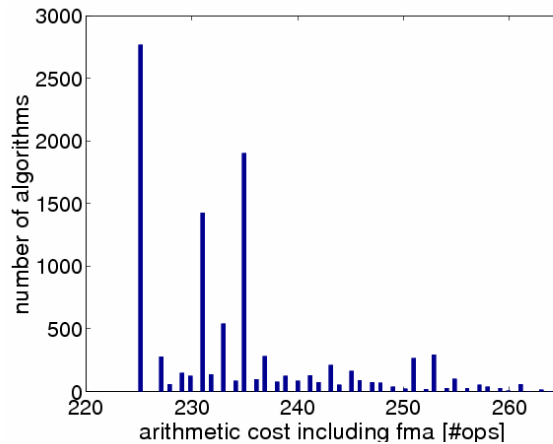
#Ops: x1.08



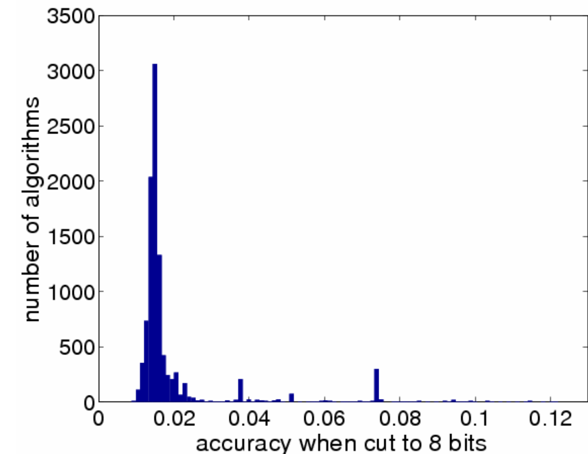
#Assembly Instr.: x1.5



#Ops und Laufzeit: keine Korrelation



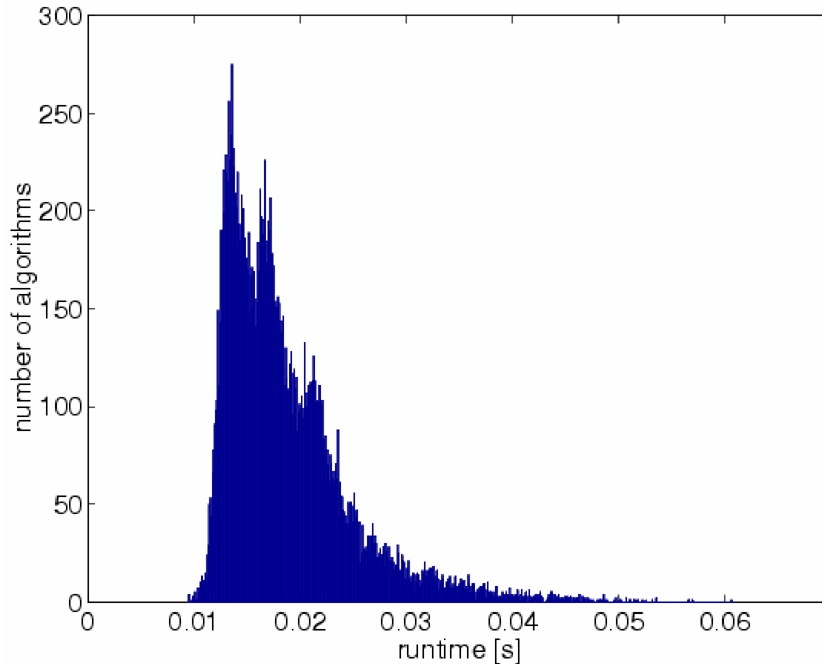
#FMA Ops: x1.2



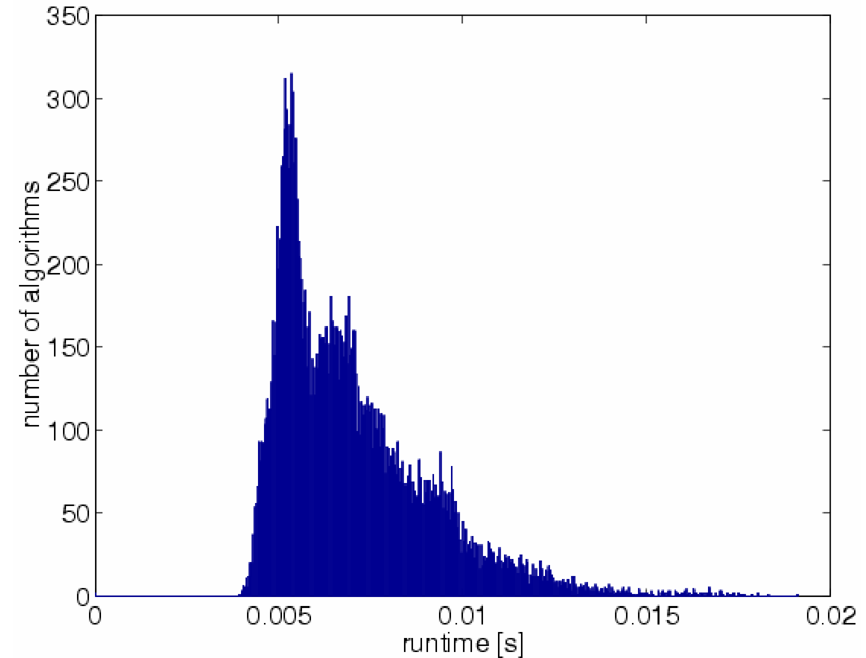
Numerische Genauigkeit: x10, die meisten x2

Laufzeitverteilung: DFT 2^16 Histogramme, 20,000 Algorithms

P4, 3.2 GHz, icc 8.0



Generierter Skalarcode

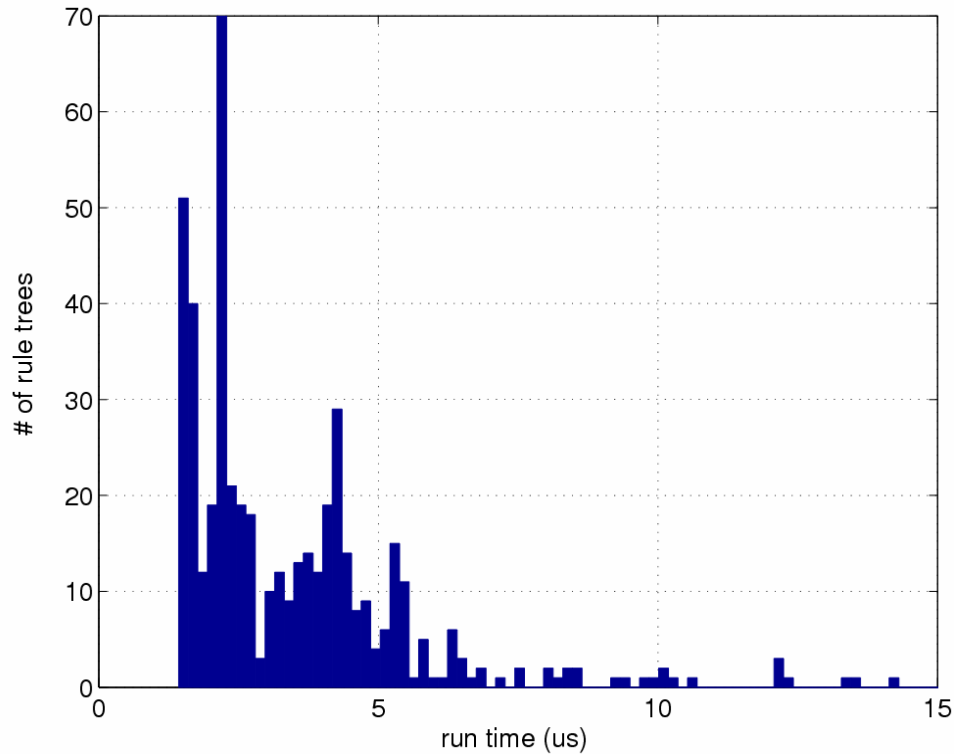


Generierter SSE Code (4-weg Vektor, einfache Genauigkeit)

- Alle Algorithmen werden schneller
- x 2.5 Verbesserung

Laufzeitverteilung: Filter(128, 16)

Pentium 4 – 3.2

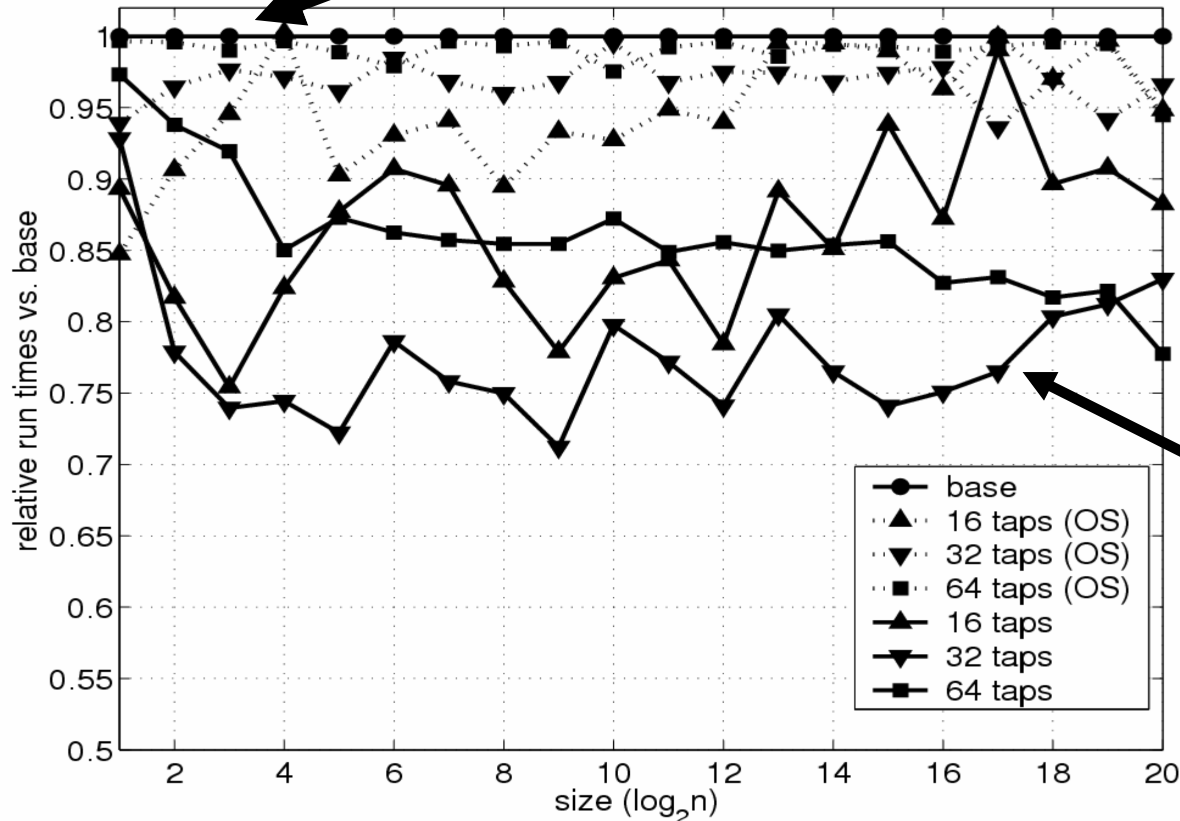


Filter: Zeitbereichmethoden

$$\mathrm{Filt}_n(h(z)) \rightarrow \mathrm{I}_n \otimes_{k-1} (h_0, \ldots, h_{k-1})$$

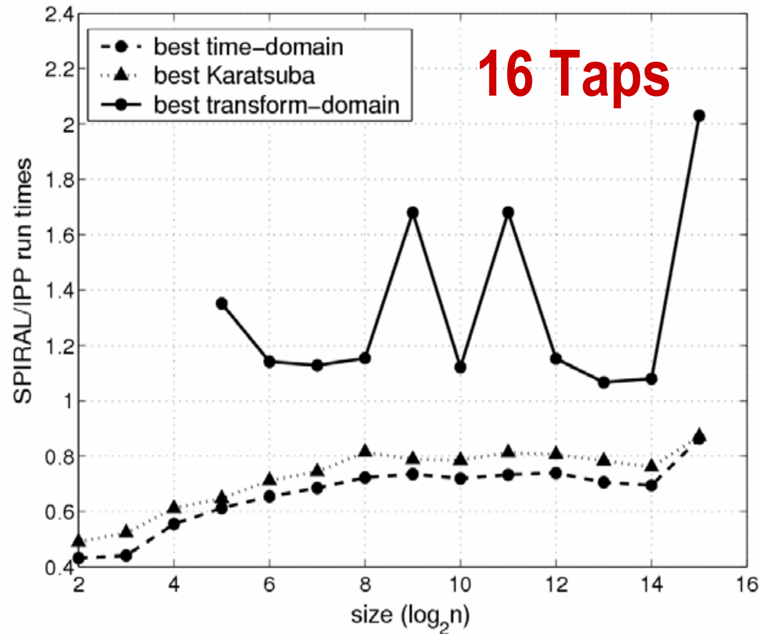
```
for i = 0..n-1
   y[i] = h[0]*x[0+i]+h[1]*x[1+i]+...+h[n-1]*x[n-1+i]
end
```

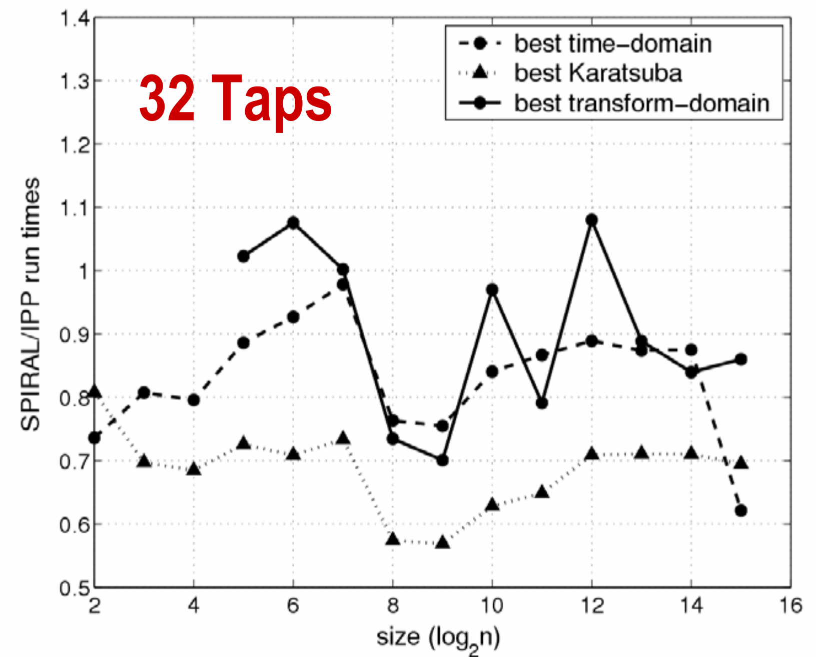
Xeon-1.7



best blocking strategy

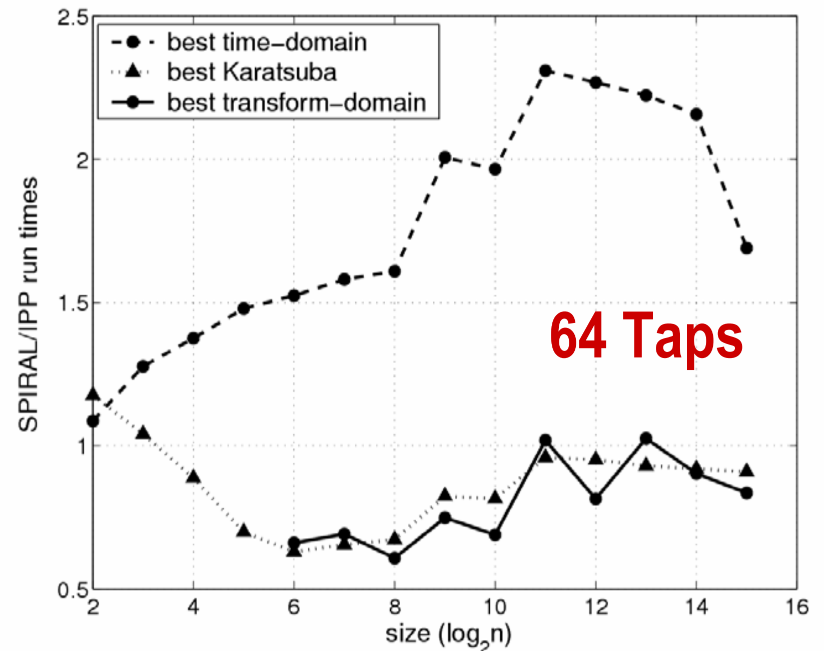
Filter: Alle Methoden



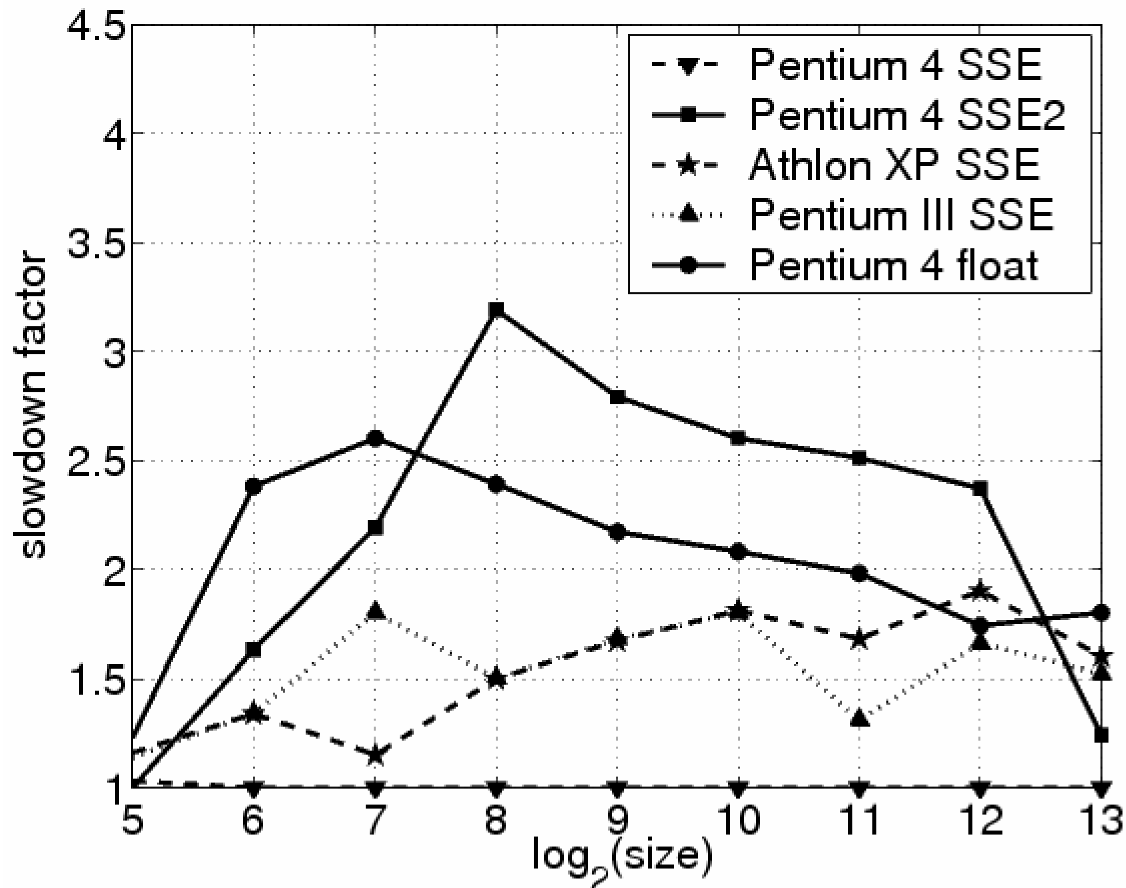


Athlon XP 1.73

- 16: Zeitbereich gewinnt
- 32: Karatsuba gewinnt
- 64: Karatsuba/DFT ~gleich



Plattformabhängigkeit: DFT



50% Verlust durch Portierung von PIII auf P4

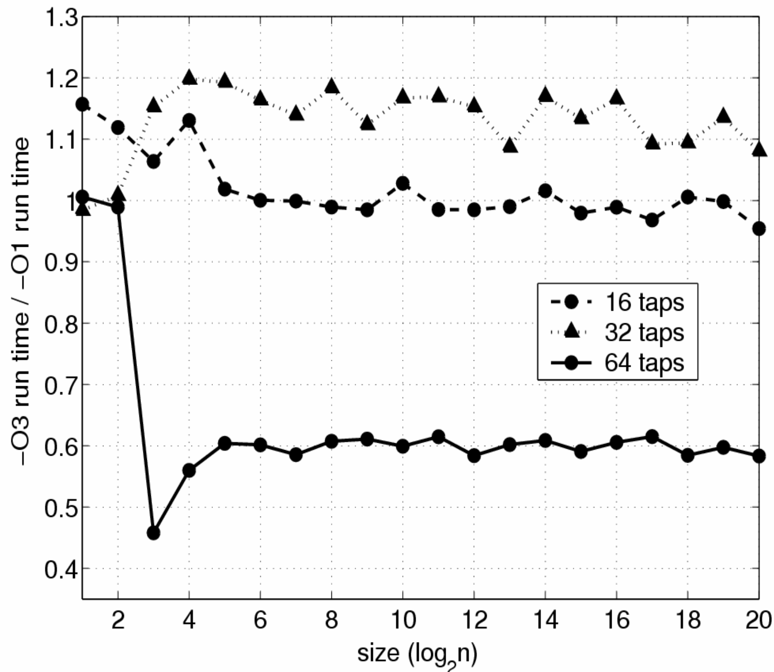
Plattformabhängigkeit: Filter

	16-tap	32-tap	64-tap	128-tap
Pentium 4 3.0GHz Northwood	Blocking	Karatsuba	RDFT	RDFT
Pentium 4 3.6GHz Prescott	Blocking	Karatsuba	Karatsuba	RDFT
Macintosh	Karatsuba	Karatsuba	RDFT	RDFT
Xeon 1.7 GHz	Blocking	Blocking	Blocking	RDFT
Athlon 1.73GHz	Karatsuba/ Blocking	Karatsuba	Karatsuba/ RDFT	RDFT

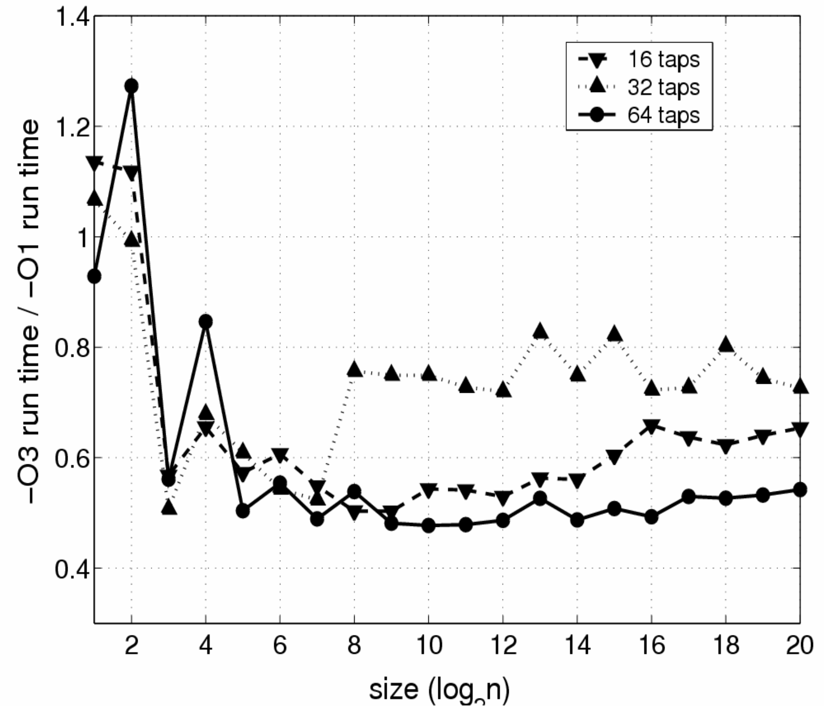
Compileroptionen: Filter

Macintosh - GNU C 3.3 (Apple)

Blocking/nesting



+ Karatsuba

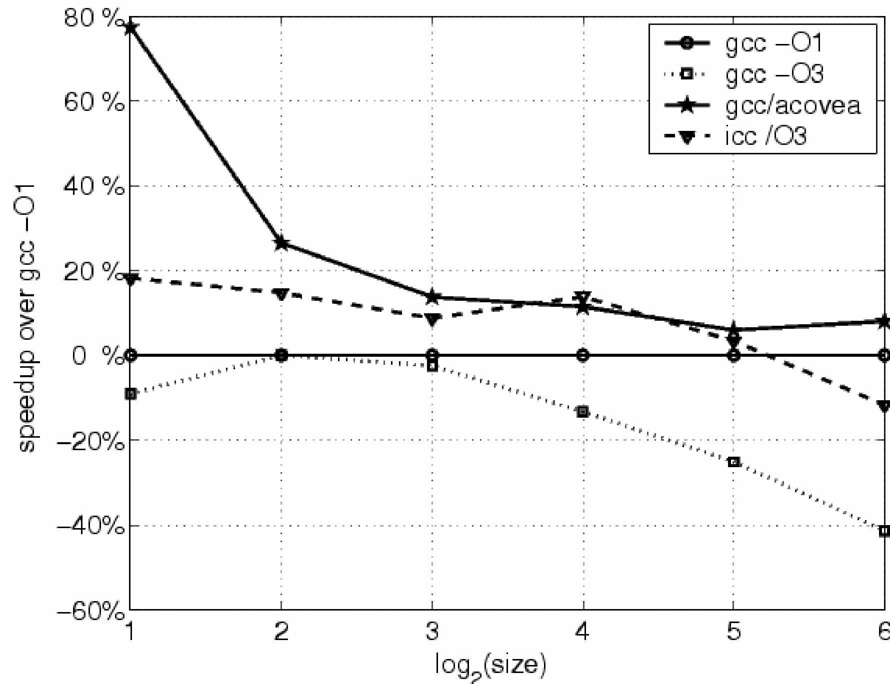


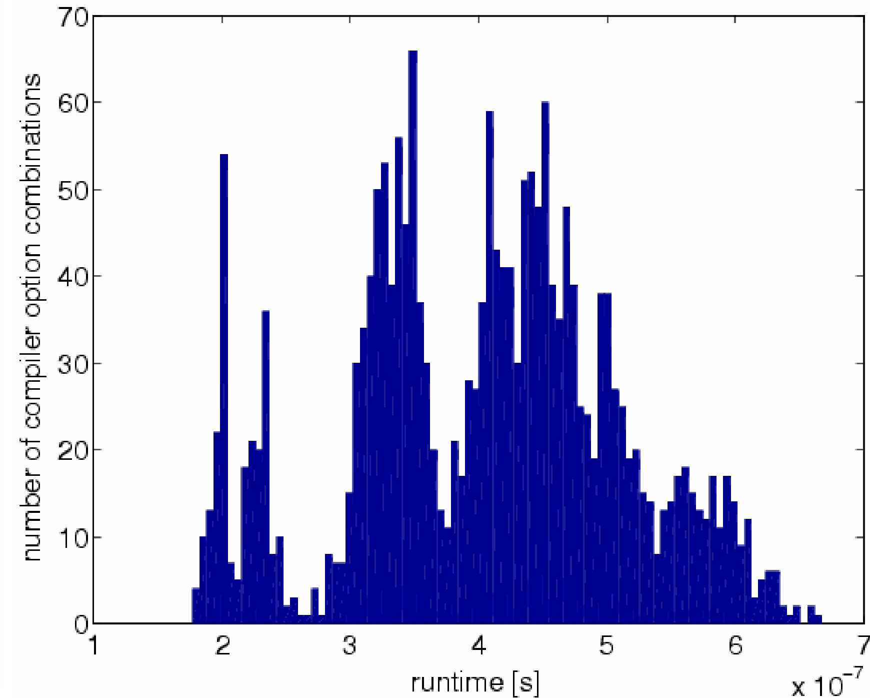

```
gcc {-O1/-O3} -fomit-frame-pointer -std=c99 -fast -mcpu=7450
```

Compileroptionen DCT

P4, 3.2 GHz, gcc

ACOVEA: Evolutionäre Suche nach Compileroptionen (gcc hat ~500)





Laufzeithistogramm zufällige Compileroptionen inkl. -O1 –march=pentium4

10% Verbesserung des besten, Spiral-generierten Codes

SPIRAL: Designprinzipien

- **Codegenerierung und –adaptierung als Optimierungsproblem auf der Menge aller Algorithmen und Implementierungen**

  SPIRAL nützt die Struktur der Algorithmen zur Lösung

- **Deklarative Sprache (SPL + Regeln) zur Computerrepräsentation des Algorithmenwissens**

  Ermöglicht Algorithmengenerierung and -optimierung
  (bringt dem System die Mathematik bei)

- **Compiler um Mathe in Code zu übersetzen**
- **Suche und Lernen in der Menge der Alternativen zur Optimierung**

  Schließt den feedback loop
  gibt dem System die "Intelligenz"

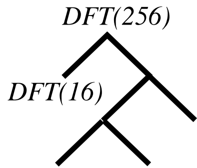
- **Flexible, erweiterbar**

  Jeder Schritt in der Codegenerierung ist explizit in SPIRAL

SPIRAL: Abstraktionsebenen

DFT(256) **Transformation**

- Problemspezifikation



Regelbaum

- natürlicher erster Schritt der Algorithmengenerierung
- speichereffizient
- einfache Manipulation in der Suche

$\ldots(\mathrm{I}_4 \oplus \mathrm{J}_4) D_8 (\mathrm{F}_2 \otimes \mathrm{I}_4) \ldots$ **SPL**

- Strukturbeschreibung des Algorithmus
- zeigt Parallelismus, Vektorisierbarkeit
- Manipulation für Vektorcode

$\ldots \sum_{j=0}^{n-1} \mathrm{S}_{(j)_m \otimes \imath_n} C_n \, \mathrm{G}_{(j)_m \otimes \imath_n}$ **Σ-SPL**

- macht Schleifen explizit
- explizite Indexfunktionen
- Schleifenoptimierung

```
…
for (i=0; i<8; i++) {
  t[2i+4]=x[i-4]+x[i+4];
  … }
```

C Code

- Standard DAG/Compileroptimierungen

Philosophie: Optimiere auf der richtigen Abstraktionsebene

Implementierung: Überblick

- Das Problem: Moore's Law und Numerische Software
- Mikroarchitektur und Architektur
- Algorithmus, Implementierung und Mikroarchitektur: Analyse der DFT und Herleitung von FFTW
- Softwaregenerierung und Experimente mit SPIRAL
- Zusammenfassung: Wie schreibt man schnellen numerischen Code?

Schneller Numerischer Code: Richtlinien (I)

- **Vermeide offensichtliche Fehler**
  - Kenne die verfügbaren Algorithmen: Verwende gute Algorithmen
  - Vorberechnung von Konstanten (z.B., Sinus, Kosinus)
  - Gib dem Compiler eine Chance: Schreibe einfachen Code
  - Vermeide komplizierte Daten/Kontroll/Infrastrukturen
  - Verstehe wo die Laufzeit bleibt: Codeprofiling
  - Verwende gute Compileroptionen, probiere Alternativen
- **Datenlokalität für Speicherhierarchie (Caches)**
  - Rekursiv ist besser als iterativ
  - Kenne die Cachegrößen!
- **Blockoptimierung (Block=Codebaustein der auf kleinem Datenbereich operiert)**
  - Ohne Schleifen und Kontrollstrukturen (aufgerollter Code)
  - Dann wichtig: ordne Instruktionen für Lokalität + evtl. andere Optimierungen
  - Eventuell: Prüfe den Assemblycode

Schneller Numerischer Code: Richtlinien (II)

- **Mögliche Suche über Alternativen**
  - akzeptiere dass man nicht alles durch Wissen entscheiden kann
  - Such über eine relevante Teilmenge von Algorithmen und/oder Implementierungsweisen
- **Verwende Vektorinstruktionen**
  - versuch es zuerst mit dem Compiler
(konsultiere Manual für Codestil, Pragmas, Alignment, etc.)
  - dann verwende Intrinsics, kein Inlineassembly
  - umfangreiche Datenumordnungen zerstören die Laufzeit
  - prüfe den Assemblycode

- **Nach jeder dieser Optimierungen, überprüfe ob Du noch den richtigen Algorithmus verwendest**

Abschließende Bemerkungen

Wichtigste Erkenntnisse (Algorithmen)

- **DFT und die Cooley-Tukey FFT können mit dem Chinesischen Restesatz erklärt und hergeleitet werden**

$$\mathrm{DFT}_n \quad \leftrightarrow \quad \mathbb{C}[x]/(x^n - 1)$$

$$\mathrm{FFT} \quad \leftrightarrow \quad x^n - 1 = (x^k)^m - 1 \text{ dekomponiert für } n = km$$

- **Es gibt sehr viele verschieden Algorithmen für die gleiche Transformation (oder Filter)**
  - ~ gleiche Anzahl an Operationen
  - unterschiedliche Struktur/Eigenschaften = unterschiedliche Laufzeit und unterschiedliche Eignung für Cachestrukturen, Speizleinstruktionen, etc.

Wichtigste Erkenntnisse (Implementierung)

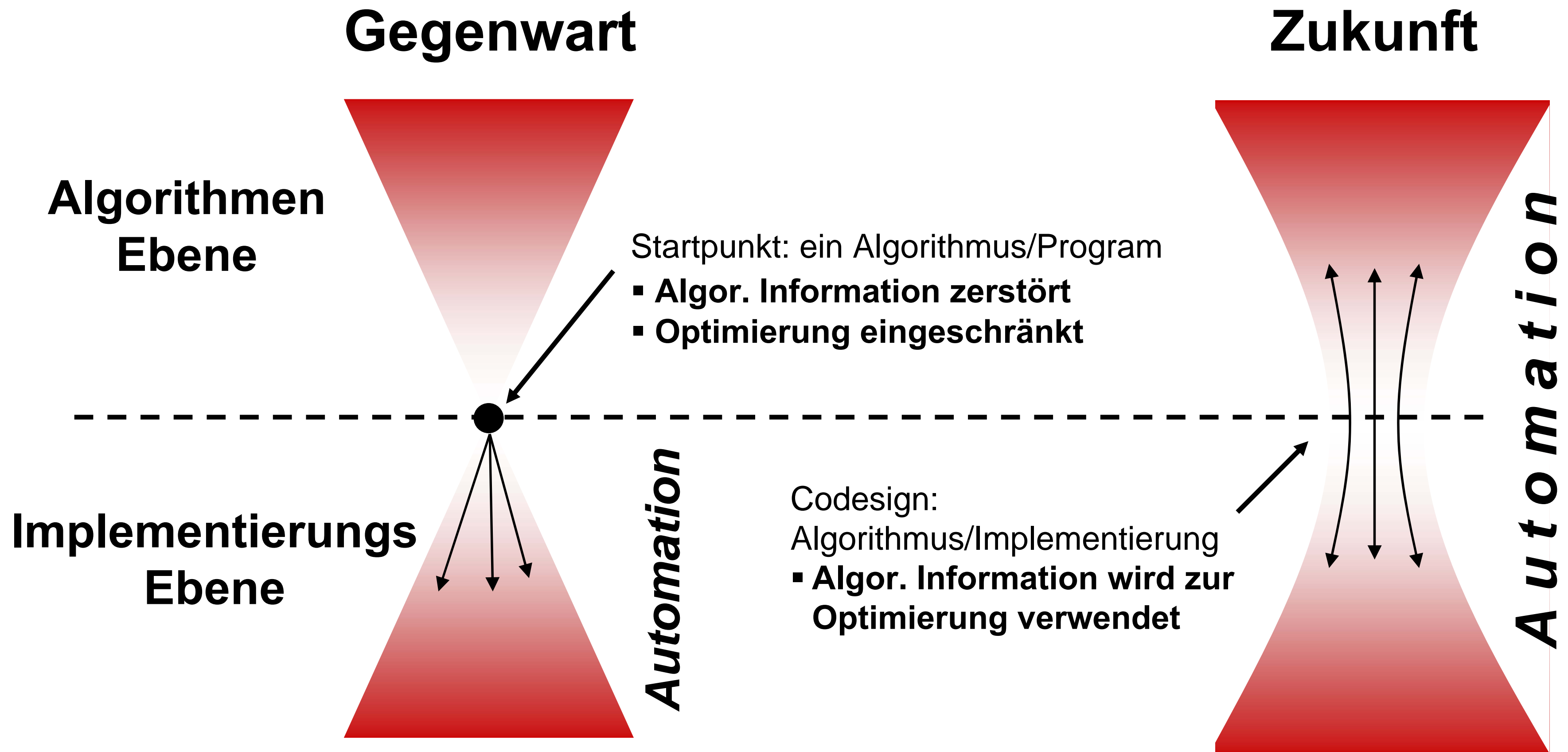
- **Schneller Code ist schwierig zu schreiben**
  - man verliert schnell einen Faktor 10-100
  - man braucht gutes Verständnis von Algorithmen, Software/Compilern, und der Mikroarchitektur
  - Computer entwickeln sich sehr schnell
- **Operationen zählen gibt wenig Information über die Laufzeit**
- **In der Zukunft wird es eher schlimmer**
  - Das Ende von Moore's Law rückt näher
  - Mögliche Zukunft: Multicoresysteme.
    Siehe Scientific American Artikel (Nov. 2004) "A Split at the Core";
    Untertitel: "[…]That is bad news for software companies."
- **Schneller Code mit wenig Aufwand ist aktuelles Forschungsproblem**

Überblick aktuelle Forschung

- **Linear Algebra:**
  - LAPACK, ATLAS (Whaley, Dongarra et al., U. Tennessee)
  - BeBOP (Vuduc, Yelick, U. Berkeley)
- **Tensorberechnungen (Quantumchemie):** Sadayappan, Baumgartner et al., Ohio State
- **Finite Elemente Methoden: Fenics (U. Chicago)**
- **Suchen:** Padua, UIUC
- **Signalverarbeitung:**
  - FFTW
  - SPIRAL
  - VSIPL (Kepner, Lebak et al., MIT Lincoln Labs)
- **Neue Compilermethoden (problemspezifisch):**
  - Modelbasiertes ATLAS (Yotov, Pingali, Cornell U.)
  - Broadway (Lin, Guyer, U. Texas Austin)
  - SIMD Optimierungen (Franchetti, Ueberhuber et al., Univ. Techn. Vienna)
  - Teleskopsprachen (Kennedy et al., Rice)
- **Siehe auch: Proceedings of the IEEE Special Issue "Program Generation, Optimization, and Adaptation" (Feb. 05, link auf Spiral Webseite)**

Wir haben gerade erst angefangen

Philosophie



eine neue Generation von problemspezifische Methoden/Werkzeugen

Automation auf allen Abstraktionsebenen

1954: der Fortran Compiler wurde erfunden

2004: heute

2054: wenn Menschen immer noch code für FFTs etc.
selbst schreiben müssen,
vielleicht sogar in Assembly,
dann haben wir was falsch gemacht

Das war’s, danke für’s kommen