Compilerbau

Wintersemester 2009 / 2010

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

Kapitel 8

Code-Optimierung

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- Kapitel 5: Semantische Analyse
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- **Kapitel 8: Code-Optimierung**
  - HIR: Parallelisierung für Homogene Multi-DSPs
  - LIR: Generierung von Bit-Paket Operationen für NPUs
  - LIR: Optimierungen für Scratchpad-Speicher
- Kapitel 9: Ausblick

Parallelisierung für Multi-DSPs

- **Material freundlicherweise zur Verfügung gestellt von:**
  Björn Franke und *Michael O'Boyle*
  University of Edinburgh, UK
  School of Informatics

Parallelisierung für Multi-DSPs

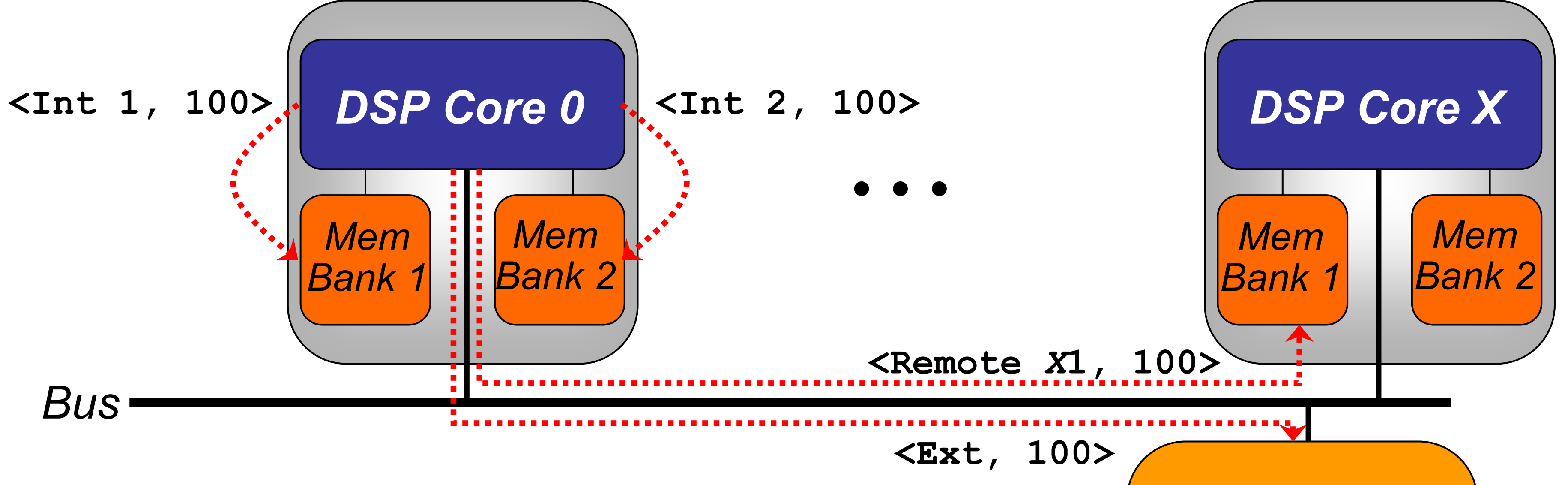
- **Motivation:**
  - Performance-Anforderungen eines gesamten Systems übersteigen oft Fähigkeiten eines einzelnen Prozessors. *(z.B. Radar, Sonar, medizinische Bildverarbeitung, HDTV, ...)*
  - Parallel arbeitende DSPs stellen hinreichend Performance zur Verfügung, aber...
    - ✗ Wenig bis keine Hardware-Unterstützung für parallele Ausführung
    - ✗ Noch weniger Unterstützung paralleler Programmiertechniken durch Entwurfswerkzeuge.
    - ✗ Existierende Quellcodes oft in low-level Stil programmiert, welcher Parallelisierung erschwert.

Parallelisierende Compiler

- **Spezialgebiet „High Performance Computing”:**
  - Seit über 25 Jahren Forschung zu Vektorisierenden Compilern
  - Traditionell Fortran-Compiler
  - Derartige Vektorisierende Compiler i.d.R. für Multi-DSPs nicht geeignet, da Annahmen über Speicher-Modell unrealistisch:
    - ✗ Kommunikation zwischen Prozessen via gemeinsamen Speicher (shared memory)
    - ✗ Speicher hat nur einen einzelnen gemeinsamen Adressraum
    - ✗ Caches können dezentral vorkommen, Cache-Kohärenz aber in HW gelöst

☞ **De Facto keine parallelisierenden Compiler für Multi-DSPs**

Multi-DSPs



- Mehrfache Adressräume:
  Intern 1 & 2, Extern, Entfernter DSP-Core
- Nutzung interner Speicher:
  Höhere Bandbreite, geringere Latenzzeit
- Nutzung entfernter Speicher:
  ID des entfernten DSPs muss bekannt sein

Ablauf der Parallelisierung

- **Programm-Wiederherstellung**
  - Entfernen „ungewünschter” low-level Konstrukte im Code
  - Ersetzung durch high-level Konstrukte
- **Entdeckung von Parallelität**
  - Identifizierung parallelisierbarer Schleifen
- **Partitionierung und Zuordnung von Daten**
  - Minimierung des Kommunikations-Overheads zwischen DSPs
- **Erhöhung der Lokalität von Speicherzugriffen**
  - Minimierung der Zugriffe auf Speicher entfernter DSPs
- **Optimierung der Speicher-Transfers**
  - Nutzung von *Direct Memory Access (DMA)* für Block-Transfers

Code-Beispiel für 2 parallele DSPs

```
/* Array-Deklarationen */
int A[16], B[16], C[16], D[16];

/* Deklaration & Initialisierung von Zeigern */
int *p_a = A, *p_b = &B[15], *p_c = C, *p_d = D;

/* Schleife über alle Array-Elemente */
for (i = 0; i < 16; i++)
  *p_d++ = *p_c++ + *p_a++ * *p_b--;
```

- Low-level Array-Zugriffe über Zeiger; explizite Zeiger-Arithmetik *(Auto-Increment Adressierung, vgl. ☞ Kapitel 1)*
- Nachteilig für Parallelisierung, da ad hoc keine Struktur in Array-Zugriffen erkenn- und analysierbar.

Programm-Wiederherstellung

```
/* Array-Deklarationen */
int A[16], B[16], C[16], D[16];

/* Schleife über alle Array-Elemente */
for (i = 0; i < 16; i++)
  D[i] = C[i] + A[i] * B[15-i];
```

- Ersetzen der Zeiger-Zugriffe durch explizite Array-Operatoren `[]`
- Struktur der Array-Zugriffe besser erkennbar, für nachfolgende Analysen zugänglicher

Programm-Wiederherstellung

```
/* Array-Deklarationen */
int A[16], B[16], C[16], D[16];

/* Schleife über alle Array-Elemente */
for (i = 0; i < 16; i++)
  D[i] = C[i] + A[i] * B[15-i];
```

- Eindimensionale „flache” Arrays für Parallelisierung für Multi-DSP Architektur zu unstrukturiert.
- Aufteilung der Arrays auf verfügbare parallele DSPs unklar.

Daten-Partitionierung

```
/* Partitionierte Array-Deklarationen */
int A[2][8], B[2][8], C[2][8], D[2][8];

/* Schleife über alle Array-Elemente */
for (i = 0; i < 16; i++)
  D[i/8][i%8] = C[i/8][i%8] +
                A[i/8][i%8] * B[(15-i)/8][(15-i)%8];
```

- Neue zweidimensionale Array-Deklarationen
- Erste Dimension entspricht Anzahl paralleler DSPs
- Ursprüngliche flache Arrays in disjunkte Bereiche zerlegt, die unabhängig voneinander bearbeitet werden können.

Daten-Partitionierung

```
/* Partitionierte Array-Deklarationen */
int A[2][8], B[2][8], C[2][8], D[2][8];

/* Schleife über alle Array-Elemente */
for (i = 0; i < 16; i++)
  D[i/8][i%8] = C[i/8][i%8] +
                A[i/8][i%8] * B[(15-i)/8][(15-i)%8];
```

- Sehr kostspielige, komplexe Adressierung notwendig.
- Grund: Arrays sind mehrdimensional; Schleifenvariable `i`, mit der Arrays indiziert werden, läuft aber sequentiell.
- Sog. Zirkuläre Adressierung *(circular buffer addressing)*.

Strip Mining der i-Schleife

```
/* Partitionierte Array-Deklarationen */
int A[2][8], B[2][8], C[2][8], D[2][8];

/* Verschachtelte Schleife über alle Array-Elemente */
for (j = 0; j < 2; j++)
  for (i = 0; i < 8; i++)
    D[j][i] = C[j][i] + A[j][i] * B[1-j][7-i];
```

- Aufteilen des sequentiellen Iterationsraums von `i` in zwei unabhängige zweidimensionale Iterationsräume
- Iterationsräume der neuen verschachtelten Schleifen spiegeln Daten-Layout wieder
- Nur noch lineare Ausdrücke zur Array-Adressierung

Strip Mining der i-Schleife

```
/* Partitionierte Array-Deklarationen */
int A[2][8], B[2][8], C[2][8], D[2][8];

/* Verschachtelte Schleife über alle Array-Elemente */
for (j = 0; j < 2; j++)
  for (i = 0; i < 8; i++)
    D[j][i] = C[j][i] + A[j][i] * B[1-j][7-i];
```

- Wie kann dieser Code für zwei DSPs parallelisiert werden?

Parallelisierung (für Prozessor 0)

```
/* Definition der Prozessor-ID */
#define MYID 0

/* Partitionierte Array-Deklarationen */
int A[2][8], B[2][8], C[2][8], D[2][8];

/* Simple Schleife über alle Array-Elemente für DSP Nr. MYID */
for (i = 0; i < 8; i++)
  D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];
```

- Einfügen einer expliziten Prozessor-ID

- Array-Adressierung unter Verwendung der Prozessor-ID
- Bei *N* parallelen Prozessoren Generierung von *N* verschiedenen HIR-Codes mit jeweils unterschiedlichen Prozessor-IDs

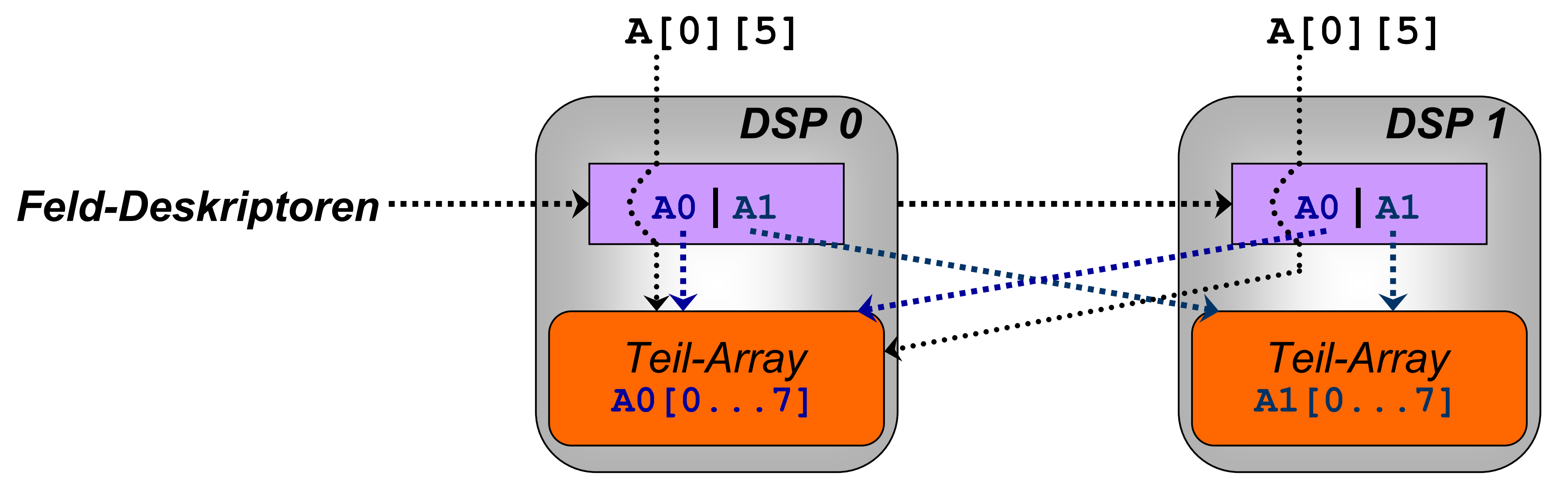
Parallelisierung (für Prozessor 0)

```
/* Definition der Prozessor-ID */
#define MYID 0

/* Partitionierte Array-Deklarationen */
int A[2][8], B[2][8], C[2][8], D[2][8];

/* Simple Schleife über alle Array-Elemente für DSP Nr. MYID */
for (i = 0; i < 8; i++)
  D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];
```

- Mit dieser Struktur ist klar, welcher Code auf welchem DSP läuft.
- Unklar ist, wie die Arrays auf lokale Speicherbänke der DSPs oder auf externe Speicher verteilt werden, und wie Zugriffe auf Speicherbänke entfernter DSPs geschehen.

Feld-Deskriptoren

- Zweidimensionales Feld `A[2][8]` wird entlang erster Dimension in 2 Teil-Arrays `A0` und `A1` zerlegt.
- Jedes Teil-Array `An` wird in Speicher von Prozessor *n* abgelegt.
- Ursprüngliche 2-dimensionale Array-Zugriffe müssen mit Hilfe von Deskriptoren auf `A0` und `A1` umgelenkt werden.



Speicher-Aufteilung (für Prozessor 0)

```
/* Definition der Prozessor-ID */
#define MYID 0

/* Partitionierte Array-Deklarationen & Feld-Deskriptoren */
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...

/* Simple Schleife über alle Array-Elemente für DSP Nr. MYID */
for (i = 0; i < 8; i++)
  D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];
```

- Felder in DSP-internem und entferntem Speicher

- Array-Zugriffe über Deskriptoren in unveränderter Syntax

Speicher-Aufteilung (für Prozessor 0)

```
/* Definition der Prozessor-ID */
#define MYID 0

/* Partitionierte Array-Deklarationen & Feld-Deskriptoren */
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...

/* Simple Schleife über alle Array-Elemente für DSP Nr. MYID */
for (i = 0; i < 8; i++)
  D[MYID][i] = C[MYID][i] + A[MYID][i] * B[1-MYID][7-i];
```

- Deskriptor-Zugriff auf lokale Arrays wegen zus. Indirektion ineffizient.
- Scheduling-Probleme: `A[i][j]` kann unterschiedliche Latenzzeit haben, wenn `i` lokalen oder entfernten Speicher referenziert.

Lokalitätserhöhung von Feld-Zugriffen

```
/* Definition der Prozessor-ID */
#define MYID 0

/* Partitionierte Array-Deklarationen & Feld-Deskriptoren */
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...

/* Simple Schleife über alle Array-Elemente für DSP Nr. MYID */
for (i = 0; i < 8; i++)
  D0[i] = C0[i] + A0[i] * B[1-MYID][7-i];
```

- Direkte Zugriffe auf lokale Felder; wann immer möglich, auf Zugriff via Deskriptoren verzichten.
- Maximale Ausnutzung der hohen Bandbreite lokaler Speicher.

Lokalitätserhöhung von Feld-Zugriffen

```
/* Definition der Prozessor-ID */
#define MYID 0

/* Partitionierte Array-Deklarationen & Feld-Deskriptoren */
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...

/* Simple Schleife über alle Array-Elemente für DSP Nr. MYID */
for (i = 0; i < 8; i++)
  D0[i] = C0[i] + A0[i] * B[1-MYID][7-i];
```

- 8 sequentielle Zugriffe auf aufeinanderfolgende Array-Elemente in entferntem Speicher
- Ineffizient, da 8 komplette Bus-Zyklen benötigt werden

Einfügen von DMA Block-Transfers

```
/* Definition der Prozessor-ID */
#define MYID 0

/* Partitionierte Array-Deklarationen & Feld-Deskriptoren */
int A0[8]; extern int A1[8]; int *A[2] = { A0, A1 };
int B0[8]; extern int B1[8]; int *B[2] = { B0, B1 }; ...

/* Temporärer Puffer für DMA */
int temp[8];
DMA_get( temp, &(B[1-MYID]), 8 * sizeof( int ) );

/* Simple Schleife über alle Array-Elemente für DSP Nr. MYID */
for (i = 0; i < 8; i++)
  D0[i] = C0[i] + A0[i] * temp[7-i];
```

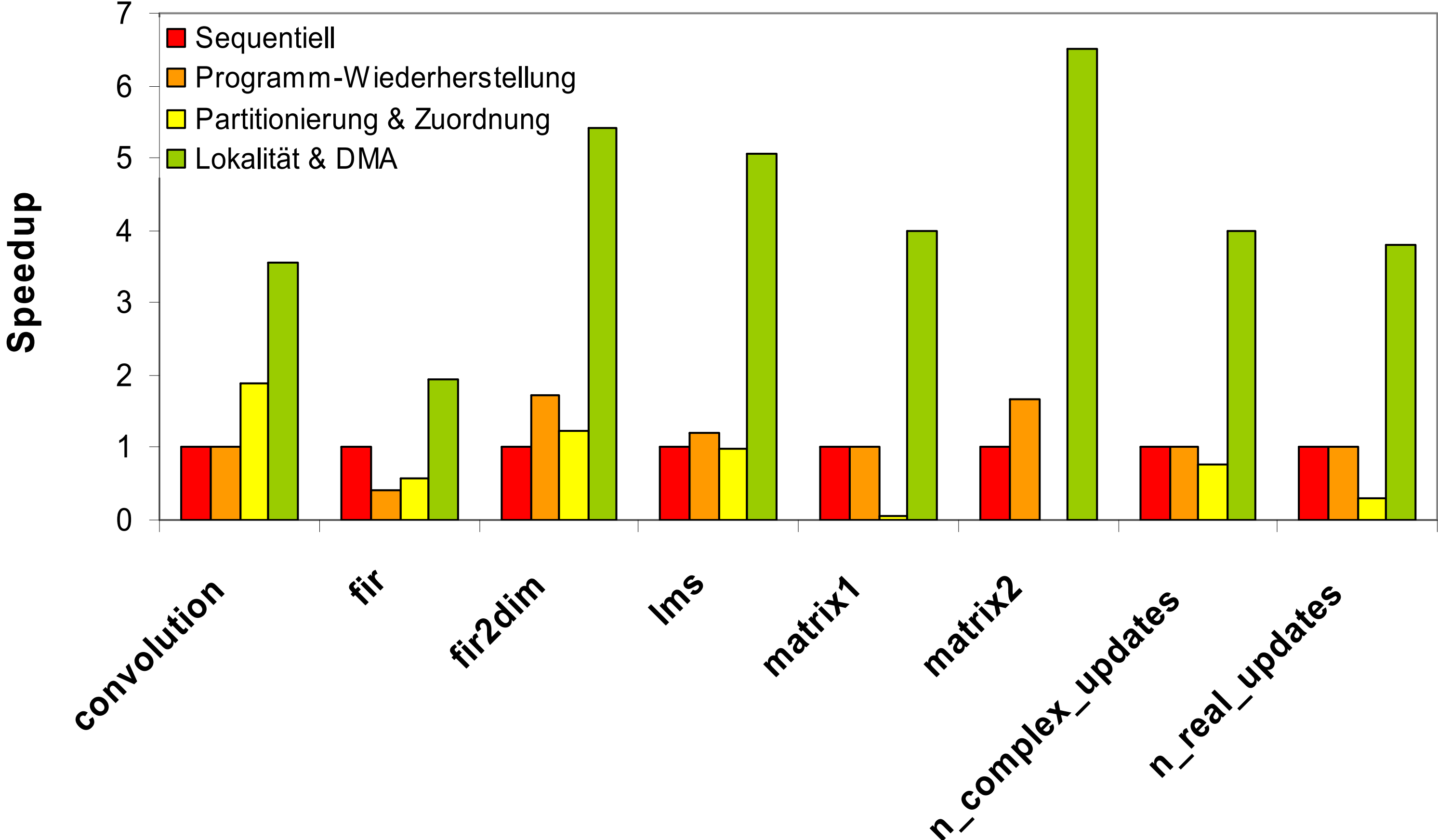
- Blockweises Laden eines lokalen Puffers aus entferntem Speicher per DMA

- Feld-Zugriffe in Schleife nur noch auf lokalen Speicher

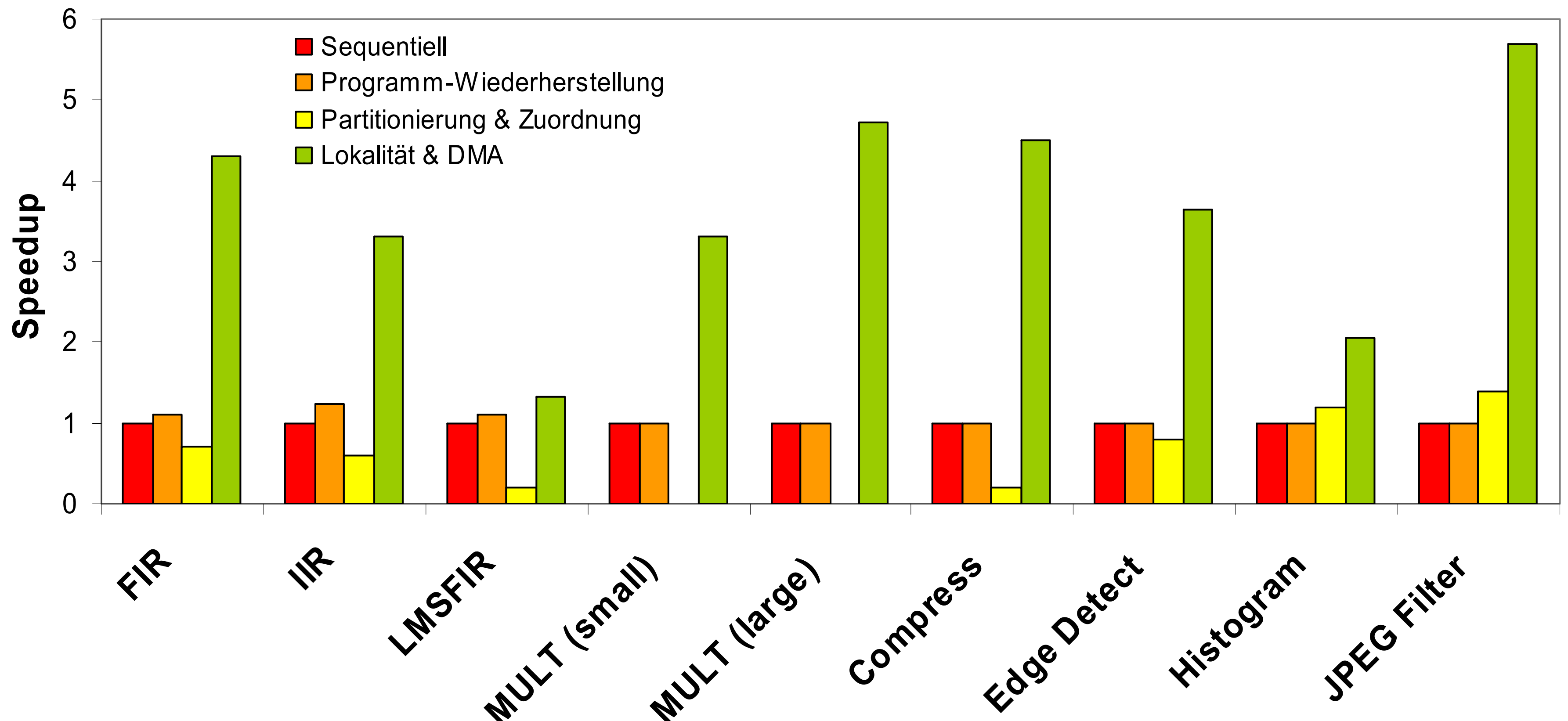
Durchführung der Parallelisierung

- **Multi-DSP Hardware**
  - 4 parallele Analog Devices TigerSHARC TS-101 @250 MHz
  - 768 kB lokales SRAM pro DSP, 128 MB externes DRAM
- **Parallelisierte Benchmark-Programme**
  - DSPstone: *kleine DSP-Routinen, geringe Code-Komplexität*
  - UTDSP: *komplexe Anwendungen, rechenintensiver Code*
- **Ergebnisse:** Laufzeiten
  - für rein sequentiellen Code auf 1 DSP laufend
  - für Code nach Programm-Wiederherstellung
  - für Code nach Daten-Partitionierung und Zuordnung
  - für Code nach Erhöhung der Lokalität & DMA

Ergebnisse – DSPstone



Ergebnisse – UTDSP



Diskussion der Ergebnisse

- **Durchschnittliche Gesamt-Speedups:**
  - DSPstone: Faktor 4,28
  - UTDSP: Faktor 3,65
  - Alle Benchmarks: Faktor 3,78
- **Sehr erstaunlich:** *Wie kann für DSPstone ein Speedup über Faktor 4 erzielt werden, wenn eine Parallelisierung für 4 DSPs erfolgt?*

Gründe für Super-Lineare Speedups

- **Super-Lineare Speedups > 4 bei 4 parallelen DSPs:**
  - Parallelisierter Code ist nachfolgenden Compiler-Optimierungen evtl. besser zugänglich als originaler sequentieller Code.
  - *Beispiel:*
    Sequentielle `i`-Schleife *(Folie 13)*: 16 Iterationen.
    Auf 2 DSPs parallelisierte `i`-Schleife *(Folie 14)*: 8 Iterationen.
  - ☞ Parallelisierte Schleifen u.U. Kandidaten für *Loop Unrolling*:

```
for (i = 0; i < 8; i++)
  <Schleifenkörper>;
```

```
<Schleifenkörper>;
<Schleifenkörper>;
...
<Schleifenkörper>;
```

8-mal

  - Abgerollte Schleife ohne Sprünge!
  - ☞ Keine Delay-Slots, Sprung-Vorhersage liegt stets richtig.

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- Kapitel 5: Semantische Analyse
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- **Kapitel 8: Code-Optimierung**
  - HIR: Parallelisierung für Homogene Multi-DSPs
  - LIR: Generierung von Bit-Paket Operationen für NPUs
  - LIR: Optimierungen für Scratchpad-Speicher
- Kapitel 9: Ausblick

Wiederholung: Datenflussgraphen

Datenflussgraph:

- Knoten repräsentiert eine Operation
- Kanten zwischen Knoten repräsentieren Definitionen *(DEFs)* und Benutzungen *(USEs)* von Daten.

Genauigkeit eines DFGs:

- Auf LIR-Ebene repräsentiert ein DFG-Knoten eine Maschinen-Operation
- Da die Operanden von Maschinen-Operationen i.d.R. Prozessor-Register sind, repräsentieren Kanten den Datenfluss durch *ganze* Register.

DFGs & Bit-Pakete

Bit-Pakete:

- Menge aufeinanderfolgender Bits
- beliebiger Länge
- an beliebiger Position startend
- u.U. Wortgrenzen überschreitend

DFGs und Bit-Pakete:

- DFGs modellieren Datenfluss auf Basis von atomaren Registern
- ☞ Information über unregelmäßig angeordnete Teilbereiche von Registern werden nicht bereitgestellt.

☞ ***Klassische DFG-basierte Verfahren i.d.R. ungeeignet zur Erzeugung von Bit-Paket Operationen!***

Beispiel: TPM und Bit-Pakete

- Zusammengesetzte Regel
  dreg: **tpm_BinaryExpAND(tpm_BinaryExpSHR(**
  dreg, const), const)
  kann Ausdruck **(c >> 4) & 0x7** überdecken und effiziente Operation **EXTR.U d_0, d_c, 4, 3** generieren.

- ***Aber:*** TPM stößt an Grenzen, wenn Muster komplexer werden:
  - ✘ Zahlen **4** / **0x7** nicht als Konstanten, sondern als Inhalt von Variablen vorliegend?
  - ✘ Andere Operator-Kombinationen als **&** / **>>** zum Erzeugen und Einfügen von Bit-Paketen in C?
  - ☞ Baum-Grammatik würde schnell ausufern und trotzdem relativ schlechten Code erzeugen!

Lösungsansatz

- **Durchführen einer konventionellen Instruktionsauswahl:**
  - Baum-Grammatik erzeugt LIR mit Maschinen-Operationen, die atomare Register als Operanden verwenden
  - Baum-Grammatik erzeugt <u>*keine*</u> Bit-Paket Operationen
  - Über Regeln
    ```
    dreg: tpm_BinaryExpAND( dreg, const )
    dreg: tpm_BinaryExpSHR( dreg, const )
    ```
    würde Ausdruck `(c >> 4) & 0x7` naiv überdeckt durch
    ```
    SH d_0, d_c, -4; AND d_1, d_0, 7;
    ```
- **Nachträgliche LIR-Optimierung:**
  - Erkennt Operationen, die Bit-Pakete extrahieren / einfügen und erzeugt entsprechende `extr` / `insert` Bit-Paket Operationen

Klassische Datenfluss-Analyse

Problem:

- Klassische Datenfluss-Analysen *(DFA)* erlauben Aussagen über *Fluss von Information*, bezogen auf die Register-Ebene:
  - ✓ Welche Operation benutzt / definiert ein bestimmtes Datum, vorliegend in einem bestimmten Register?
  - ✓ Zwischen welchen Operationen bestehen Daten-Abhängigkeiten?
- Klassische Datenfluss-Analysen treffen *keinerlei* Aussage über
  - ✘ den *Wert von Information*, d.h. den potentiellen Wert, den ein Register zu einem bestimmten Zeitpunkt haben kann.
  - ✘ den potentiellen Wert, den ein *Teil eines Registers* zu einem bestimmten Zeitpunkt haben kann.

Bitgenaue Wertfluss-Analyse

Wertfluss-Analyse *(WFA)*:

- Analysiert ebenso wie DFA den Datenfluss,
- nimmt aber zusätzlich Abschätzungen über den Inhalt der an der Datenverarbeitung beteiligten Speicherzellen vor.

Bitgenaue Daten- und Wertfluss-Analyse *(BDWFA)*:

- Wertfluss-Abschätzung wird für jedes einzelne Bit der an der Datenverarbeitung beteiligten Speicherzellen vorgenommen.
- ☞ Erlaubt Aussagen über den potentiellen Wert jedes einzelnen Bits einer Speicherzelle zu einem bestimmten Zeitpunkt.

☞ ***Im folgenden: Präsentation einer BDWFA mit mehrwertiger Logik für Register als Speicherzellen.***

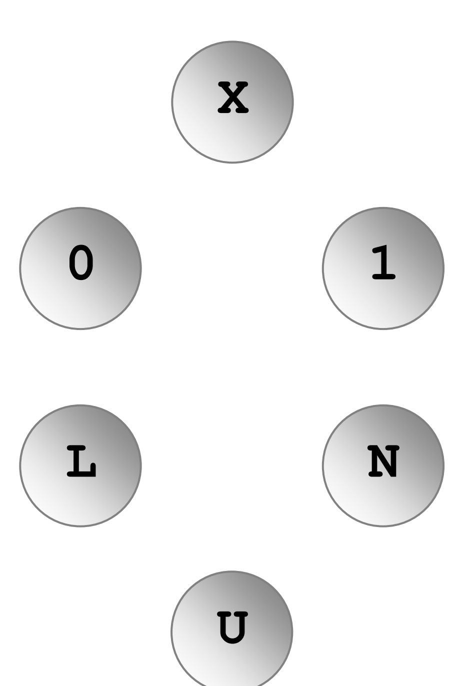
Daten- und Wertflussgraph (DWFG)

Definition *(Daten- und Wertflussgraph)*:

Sei F eine LIR-Funktion. Der *Daten- und Wertflussgraph (DWFG)* zu F ist ein gerichteter Graph $DWFG = (V, E, d, u)$ mit

- Knotenmenge V identisch mit DFG-Definition *(vgl. ☞ Kapitel 6)*
- Seien $op_i(p_{i,1}, ..., p_{i,n})$ und $op_j(p_{j,1}, ..., p_{j,m})$ zwei Operationen aus F mit Parametern $p_{i,x}$ bzw. $p_{j,y}$, und v_i und v_j die zu op_i und op_j gehörenden Knoten. Für jede Benutzung eines Registers r durch $p_{j,y}$, das von $p_{i,x}$ definiert wird, existiert eine Kante $e = (v_i, v_j) \in E$.
- d und u stellen bitgenaue Wert-Informationen zu Kanten $e \in E$ bereit (*Down*- und *Up-Werte*). Sei r das Register, das durch e modelliert wird, und r sei k Bits breit. Dann sind d und u Abbildungen $d \mid u: E \rightarrow \mathcal{L}_4{}^k$ mit $\mathcal{L}_4$ als Halbordnung zur Darstellung des potentiellen Wertes eines einzelnen Bits.

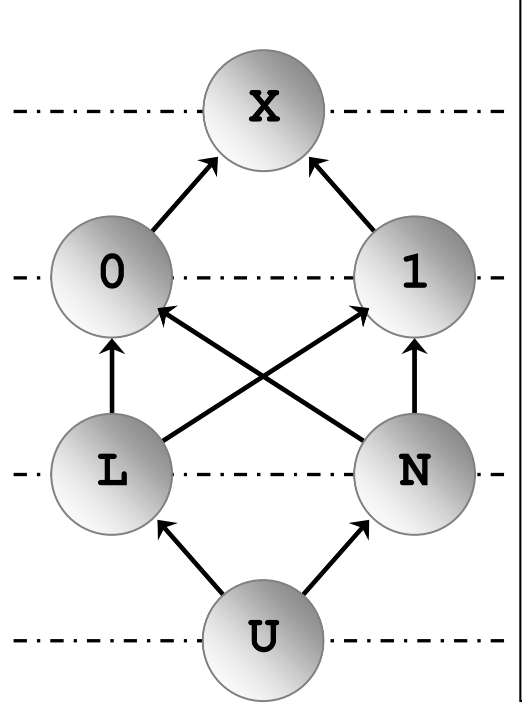
Die Halbordnung $\mathcal{L}_4$ (1)



Pro Bit eines Registers wird mit einem Element aus $\mathcal{L}_4$ gespeichert, welche Werte das Bit haben kann:

- `0` – Das betrachtete Bit hat den Wert 0
- `1` – Ein Bit hat den Wert 1
- `U` – Der Wert eines Bits ist völlig unbekannt
- `X` – Der Wert eines Bits ist irrelevant *(don't care)*
- `L` – Der Wert eines Bits ist unbekannt, aber dessen Herkunft *(Location)* ist bekannt.
- `N` – Wie `L`, nur ist das Bit bei bekannter Herkunft einmal negiert worden *(negated Location)*

Die Halbordnung $\mathcal{L}_4$ (2)



$\mathcal{L}_4$ ist eine Halbordnung:

- Es gibt einen Operator '<', der die Elemente in $\mathcal{L}_4$ gemäß gerichteter Kanten in Relation setzt.
- **U** hat geringsten Informationsgehalt und ist gemäß <-Operator am kleinsten.
- **X** hat höchsten Informationsgehalt und ist gemäß <-Operator am größten.

Diagramm links enthält vier horizontale Ebenen

$\mathcal{L}_4$

Beispiele zu $\mathcal{L}_4$ (1)

Für eine Kante *e* sei *r* ein 8-Bit Register, das durch *e* modelliert wird.

Graphische Notation: ⇧ repräsentiert Up-Wert, ⇩ Down-Wert

Little Endian Darstellung (höchstwertiges Byte links geschrieben)

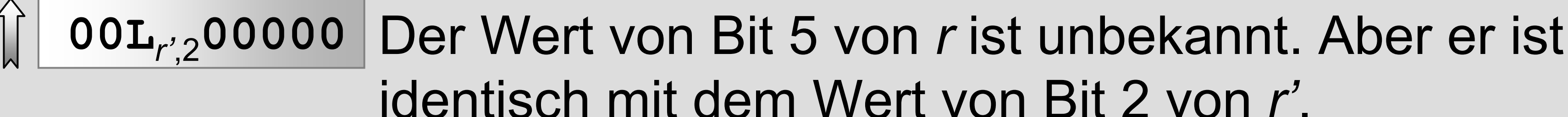
Beispielhafte Kanten-Informationen für *e* und deren Interpretation:

⇧	`UUUUUUUU`	Der Wert von *r* ist komplett unbekannt.
⇧	`00101010`	Der Wert von *r* ist gleich 42.
⇧	`0010X010`	Bit 3 von *r* ist irrelevant; *r* kann gleich 34 oder 42 sein.
⇧	`XXXXXXXX`	*r* ist komplett irrelevant ☞ *r* hat keine Auswirkung auf den Datenfluss

Beispiele zu $\mathcal{L}_4$ (2)

Sei zusätzlich *r’* ein 8-Bit Register, das einen Eingangswert für die betrachtete LIR-Funktion *F* darstellt (z.B. Funktionsparameter).

Beispielhafte Kanten-Informationen für *e* und deren Interpretation:



$\texttt{00L}_{r',2}\texttt{00000}$ Der Wert von Bit 5 von *r* ist unbekannt. Aber er ist identisch mit dem Wert von Bit 2 von *r’*.

$\texttt{00N}_{r',4}\texttt{00000}$ Der Wert von Bit 5 von *r* ist unbekannt. Aber er ist identisch mit dem negierten Wert von Bit 4 von *r’*.

$\texttt{00L}_{r',2}\texttt{L}_{r',1}\texttt{L}_{r',0}\texttt{000}$ *r* enthält ein Bit-Paket bestehend aus den Bits 2 bis 0 von *r’*, platziert ab Bit-Position 3 in *r*.

Rechnen in $\mathcal{L}_4$ – Konjunktion

$\wedge$	1	0	L_i	N_i	U	X
1	1	0	L_i	N_i	U	X
0	0	0	0	0	0	0
L_i	L_i	0	L_i	0	U	U
N_i	N_i	0	0	N_i	U	U
U	U	0	U	U	U	U
X	X	0	U	U	U	X

Bemerkung:

- Werden **L**/**N**-Werte mit verschiedener Herkunft verknüpft, resultiert stets **U**.

Rechnen in $\mathcal{L}_4$ – Disjunktion

$\vee$	1	0	L_i	N_i	U	X
1	1	1	1	1	1	1
0	1	0	L_i	N_i	U	X
L_i	1	L_i	L_i	1	U	U
N_i	1	N_i	1	N_i	U	U
U	1	U	U	U	U	U
X	1	X	U	U	U	X

Bemerkung:

- Werden **L**/**N**-Werte mit verschiedener Herkunft verknüpft, resultiert stets **U**.

Rechnen in $\mathcal{L}_4$ – Negation

¬	1	0	L_i	N_i	U	X
	0	1	N_i	L_i	U	X

Ablauf der BDWFA

Gegeben:

- Eine zu optimierende Zwischendarstellung *LIR*

Zweiphasige Vorgehensweise:

- Für jede Funktion *F* aus *LIR*:
  - Bestimme initialen Daten- und Wertflussgraph $D = (V, E, \emptyset, \emptyset)$ mit leeren Abbildungen *d* und *u* von *F*
  - Bestimme Down-Werte $d(e)$ aller Kanten durch Vorwärts-Analyse
  - Bestimme Up-Werte $u(e)$ aller Kanten durch Rückwärts-Analyse

Vorwärts-Analyse

Ziel:

- Ausschließliche Berechnung von ⇓-Werten für *D*
- ⇓-Wert *d*(*e*) repräsentiert bitgenaues Resultat eines Knotens $v \in V$ (d.h. aus *v* ausgehende Kante *e*), wenn man Operator von *v* auf Operanden (d.h. ⇓-Werte in *v* eingehender Kanten) anwendet.

Ansatz:

- (Wiederholter) Durchlauf durch *D* entlang Kanten-Richtung
  ☞ „*Vorwärts*"-Analyse
- Für jeden aktuell besuchten Knoten $v \in V$:
  - Führe *Vorwärts-Simulation* des Operators von *v* auf ⇓-Werten aller eingehenden Kanten aus
  - Speichere neue ⇓-Werte der ausgehenden Kanten von *v*

Ablauf der Vorwärts-Analyse (1)

- queue<DWFG_node> *q* = <*Menge aller Quell-Knoten in D*>;
  $d(e) = \mathtt{U}^*$ für alle Kanten $e \in E$;
- while (!*q*.empty())
  - DWFG_node *v* = <*erstes Element aus q*>; *q*.remove(*v*);
    $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
  - if (<*v repräsentiert konstante Zahl c*>)
    $d'(e) = \{\mathtt{0}, \mathtt{1}\}^*$ = <*Binärdarstellung von c*> für alle $e \in E_{out}$;
  - else
    if (<*v repräsentiert unbekannte Eingangsvariable i von F*>)
    $d'(e) = \{\mathtt{L}_i\}^*$ = <*Bit-Locations von i*> für alle $e \in E_{out}$;
  - else
    $d'(e)$ = <*Vorwärts-Simulation von v*> für alle $e \in E_{out}$;

Ablauf der Vorwärts-Analyse (2)

- while (!*q*.empty())
  - ... *<siehe vorige Folie>*;
  - for (*<alle Kanten e* = $(v, v_x) \in E_{out}$*>*)
    - if (*<bisheriges d*(*e*) *ist bitweise kleiner gemäß* <*-Operator in* $\mathcal{L}_4$ *als d'*(e)*>*)
      - *d*(*e*) = *d'*(*e*);
      - if (!*q*.contains(v_x))
        q.insert(v_x);

Bemerkungen zur Vorwärts-Analyse

- Konstanten und Eingangsvariablen liefern initiale Belegung der ⇓-Werte mit Elementen **0**, **1** und **L**.
- Für ein *k*-Bit Register *r*, modelliert durch Kante *e*, berechnet die Vorwärts-Analyse zunächst einen temporären ⇓-Wert $d'(e)$.
- $d(e) \in \mathcal{L}_4^k$ wird erst auf $d'(e) \in \mathcal{L}_4^k$ gesetzt, wenn
  - für mindestens eine Bit-Position i $(0 \leq i \leq k)$ gilt: $d_i(e) < d'_i(e)$, und
  - für keine Bit-Position i $(0 \leq i \leq k)$ gilt: $d'_i(e) < d_i(e)$
- ☞ Da ⇓-Werten im Laufe der Analyse nur stetig höherer Informationsgehalt zugewiesen wird, gelangt jeder Knoten $v \in V$ nur endlich oft in die Queue q.
- ☞ Vorwärtsanalyse terminiert zwangsläufig, Komplexität $O(|E|)$.

Vorwärts-Simulation (1)

Ziel:

- Für jeden Knoten $v \in V$, der eine Maschinen-Operation *op* in der LIR-Darstellung von F repräsentiert, und jede ausgehende Kante $e \in E_{out}$ berechnet die Vorwärts-Simulation den ⇓-Wert, in Abhängigkeit von den ⇓-Werten aller eingehenden Kanten $e_{in,1}, \ldots, e_{in,N} \in E_{in}$:

$$d'(e) = VS_{op}(d(e_{in,1}), \ldots, d(e_{in,N}))$$

Herausforderung:

- Für *jede* mögliche Maschinen-Operation aus LIR ist eine bitgenaue Simulationsfunktion VS_{op} auf $\mathcal{L}_4^k$ bereitzustellen.
- VS_{op} muss das Verhalten von op für den betrachteten Ziel-Prozessor *exakt und bitgenau* modellieren!

Vorwärts-Simulation (2)

Prinzipielle Vorgehensweise:

- Jede Maschinen-Operation *op* kann grundsätzlich mit den Booleschen Standard-Operatoren $\wedge$, $\vee$ und $\neg$ auf Basis einzelner Bits dargestellt werden.
- ☞ Beschreibe VS_{op} als Formel über den Operatoren $\wedge$, $\vee$ und $\neg$ auf $\mathcal{L}_4{}^k$, analog zur Booleschen Darstellung von *op*.

Bitweise logische Operationen:

- Maschinen-Operationen op zur logischen Verknüpfung (AND, OR, NOT, XOR, NOR, NAND, ...) können leicht mit Hilfe von $\wedge$, $\vee$ und $\neg$ in $\mathcal{L}_4{}^k$ dargestellt werden.
- ☞ Vorgehensweise klar.

Vorwärts-Simulation (3)

Arithmetische Operationen:

- Ableitung von logischen Operationen in $\mathcal{L}_4^k$ aus arithmetischer Maschinen-Operation *op* aufwändig, aber machbar.

☞ Beispiel Addition:

- Halbaddierer: Addiert Bits a, $b \in \mathcal{L}_4$, erzeugt s, $c \in \mathcal{L}_4$:
  $s = a \otimes b = (a \wedge \neg b) \vee (\neg a \wedge b)$; $c = a \wedge b$;
- Volladdierer: Addiert Bits a, b, $c_{in} \in \mathcal{L}_4$, erzeugt s, $c_{out} \in \mathcal{L}_4$:
  $s = (a \otimes b) \otimes c_{in}$; $c_{out} = (a \wedge b) \vee (a \wedge c_{in}) \vee (b \wedge c_{in})$;
- k-Bit Addition in $\mathcal{L}_4^k$:
  Wende sukzessiv Formeln für Volladdierer für Bit-Positionen 0, ..., k an und berechne Summen-Bits.

Vorwärts-Simulation (4)

Register-Transfer-Operationen:

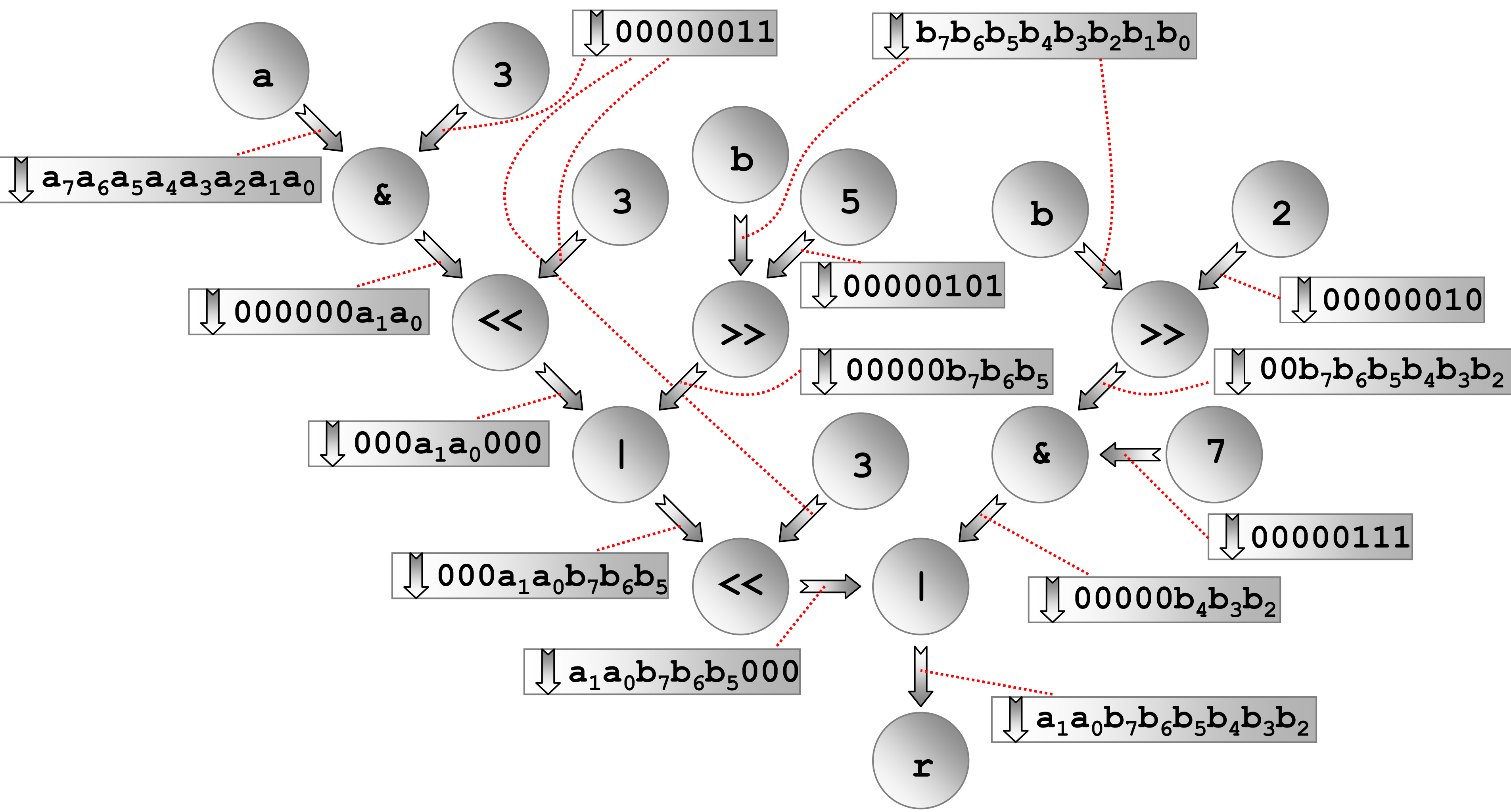
- Kopieren von Registerinhalten (Register-Register-Move) wird durch Kopieren von ⇓-Werten in $\mathcal{L}_4^k$ erreicht.

Speicher-Transfer-Operationen:

- Da die BDWFA bitgenaue Daten- und Wertflussinformation explizit nur für Register und nicht für externe Speicher vorhält,
  - generieren Store-Operationen ohnehin keine ⇓-Werte, da dies typischerweise Senken im DWFG sind,
  - generieren Load-Operationen $\mho^*$ als ⇓-Werte.

☞ ***Andere Klassen von Operationen werden ähnlich modelliert.***

Beispiel Vorwärts-Simulation



Rückwärts-Analyse

Motivation & Ziel:

- Konjunktion, Disjunktion & Negation in $\mathcal{L}_4$ erzeugen **`X`** nur dann, wenn einer ihrer Operanden schon **`X`** ist.
- ☞ Da $\Downarrow$-Werte für Quell-Knoten ausschließlich aus **`0`**, **`1`** und **`L`** bestehen, erzeugt die Vorwärts-Analyse niemals **`X`**.
- Rückwärts-Analyse erzeugt für einzelne Bit-Positionen **`X`** unter Ausnutzung der berechneten $\Downarrow$-Werte.

Rückwärts-Analyse

Ansatz:

- (Wiederholter) Durchlauf durch *D* entgegen der Kanten-Richtung
  ☞ *„Rückwärts"*-Analyse
- Für jeden aktuell besuchten Knoten $v \in V$:
  - Beantwortung der Frage, welche Bits der ⇑-Werte in *v* eingehender Kanten irrelevant sind, um exakt die ⇑-Werte der aus *v* ausgehenden Kanten zu erzeugen.
  - Führe *Rückwärts-Simulation* des Operators von *v* auf ⇑-Werten der ein- und ausgehenden Kanten aus
  - Speichere neue ⇑-Werte der eingehenden Kanten von *v*

Ablauf der Rückwärts-Analyse

- queue<DWFG_node> *q* = <*Menge aller Senken-Knoten in D*>;
  $u(e) = d(e)$ für alle Kanten $e \in E$;
- while (!*q*.empty())
  - DWFG_node *v* = <*erstes Element aus q*>; *q*.remove(*v*);
    $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
  - for (<*alle Kanten* $e = (v_x, v) \in E_{in}$>)
    - $u'(e)$ = <*Rückwärts-Simulation von v über* E_{out} *und* $E_{in} \setminus \{e\}$>;
    - if (<*bisheriges* $u(e)$ *ist bitweise* < *als* $u'(e)$>)
      - $u(e) = u'(e)$;
      - if (!*q*.contains(v_x))
        q.insert(v_x);

Rückwärts-Simulation (1)

Ziel:

- Für jeden Knoten $v \in V$, der eine Maschinen-Operation *op* in der LIR-Darstellung von *F* repräsentiert, und jede eingehende Kante $e \in E_{in}$ berechnet die Rückwärts-Simulation den ⇑-Wert, in Abhängigkeit von den ⇑-Werten aller ausgehenden Kanten $e_{out,1}, ..., e_{out,N} \in E_{out}$ *und* aller eingehenden Kanten außer *e*:

$$u'(e) = RS_{op}(u(e_{out,1}), ..., u(e_{out,N}), u(e_{in} \in E_{in} \setminus \{e\}))$$

Analog zur Vorwärts-Simulation:

- Für *jede* mögliche Maschinen-Operation *op* aus LIR ist eine bitgenaue Simulationsfunktion RS_{op} auf $\mathcal{L}_4^k$ bereitzustellen.

Rückwärts-Simulation (2)

Prinzipielle Vorgehensweise:

- Ausnutzung von Neutralen Elementen und von Null-Elementen einzelner Operatoren, um Don't Cares zu identifizieren.

Bitweise logische Operationen:

- Seien $b_{1,k}$ und $b_{2,k} \in \mathcal{L}_4$ einzelne Bits an Position k des ⇑-Wertes zweier Operanden einer logischen Operation und $b_{3,k}$ das k-te Bit des ⇑-Wertes des Resultats der Operation.
- Für $b_{2,k} = b_{3,k} = \mathbf{0}$:
  $b_{1,k}$ AND $b_{2,k} = b_{3,k}$ ⇔ $b_{1,k}$ AND **0** = **0** ⇔ **X** AND **0** = **0**
- Analog für OR: $b_{1,k}$ OR **1** = **1** ⇔ **X** OR **1** = **1**

☞ Vorgehensweise für weitere logische Operationen ähnlich.

Rückwärts-Simulation (3)

Arithmetische Operationen:

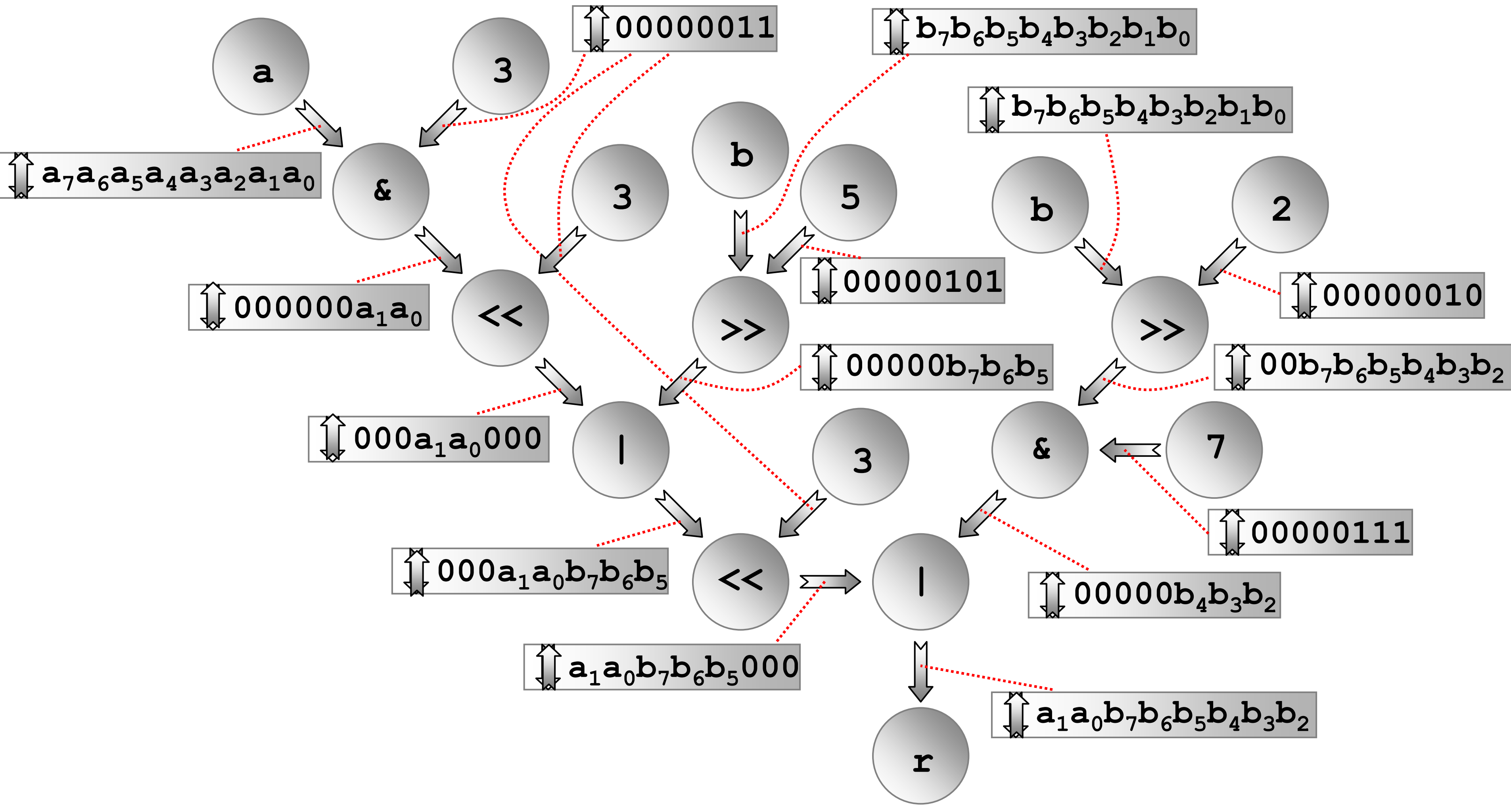
- Identifizierung irrelevanter Bits wegen Komplexität arithmetischer Operationen oft sehr schwer.

☞ Leicht verständliches Beispiel – Schiebe-Operator:

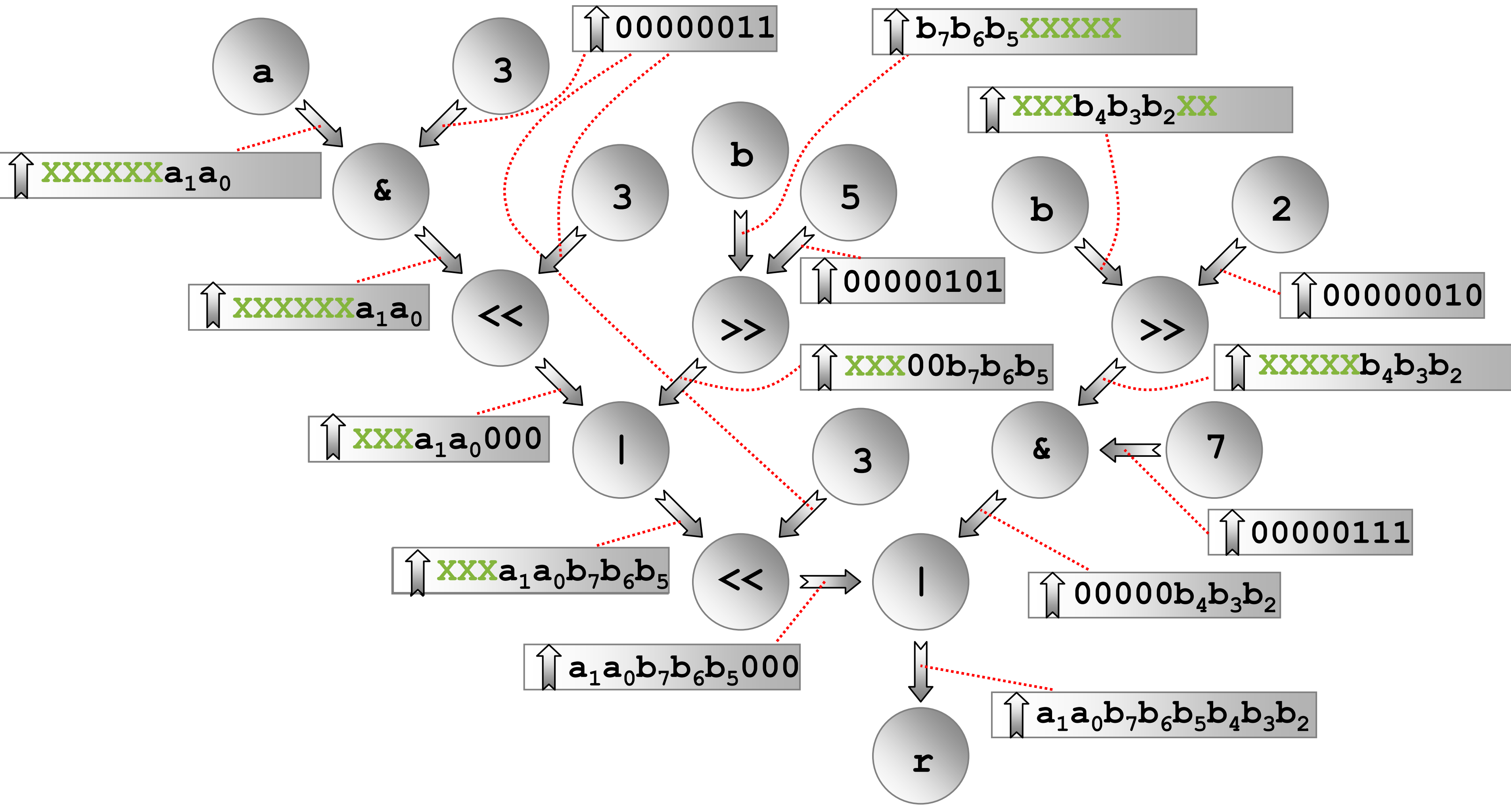
- `a << 3`: Verschiebt Inhalt von `a` um 3 Bits nach links
  Niederwertige 3 Bits werden mit `0` gefüllt,
  Höchstwertige 3 Bits werden abgeschnitten.
  ☞ Im ⇧-Wert für `a` sind die 3 höchstwertigen Bits `X`
- `a >> 3`: Analog für niederwertige 3 Bits

☞ ***Andere Maschinen-Operationen müssen sorgfältig analysiert und ähnlich modelliert werden.***

Beispiel Rückwärts-Simulation



Beispiel Rückwärts-Simulation



Anwendungen der BDWFA: Dead Code Elimination *(DCE)*

Definition *(Dead Code)*:

(LIR-) Operationen, die nachweislich keinen Effekt auf das Ergebnis einer Berechnung haben, heißen Dead Code.

Dead Code und die BDWFA:

- In ⇧-Werten von Kanten des DWFG sind einzelne Bits, die nachweislich keinerlei Effekt auf das Ergebnis einer Berechnung haben, auf **x** gesetzt.

☞ *Eine LIR-Operation, für die jede ausgehende Kante den ⇧-Wert* **x**** *hat, ist Dead Code.*

DC

...

⇧ xxxxxxxx

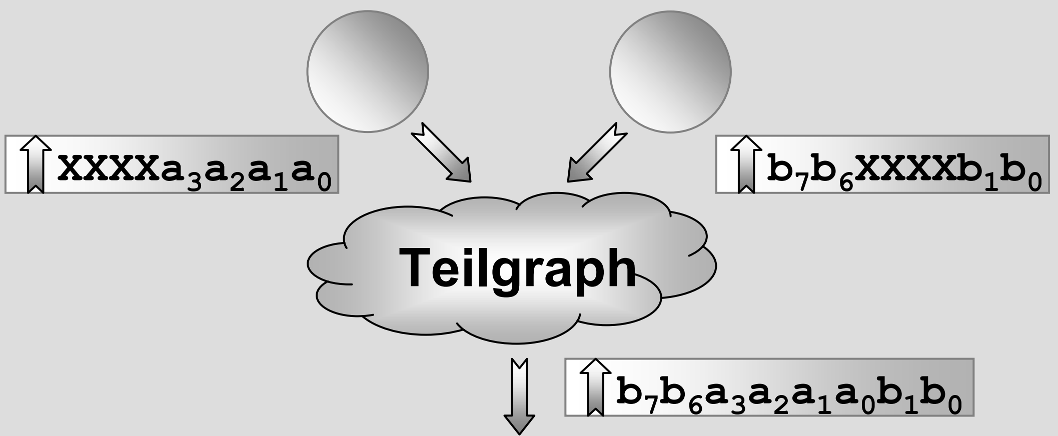
Ablauf der bitgenauen DCE

- suche Kante $e = (v, w) \in E$ mit $u(e) = \mathbf{x}^*$;
  queue<DWFG_node> $q = v$;
- while (!q.empty())
  - DWFG_node v = <*erstes Element aus q*>; q.remove(v);
    $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
  - if (($u(e) = \mathbf{x}^*$ für alle $e \in E_{out}$) &&
    (<*v hat keine Seiteneffekte*>))
    - markiere v;
    - for (alle Kanten $e = (v_x, v) \in E_{in}$)
      - $u(e) = \mathbf{x}^*$;
      - q.insert(v_x);
- lösche alle zu markierten Knoten gehörenden LIR-Operationen;

Anwendungen der BDWFA: Erzeugung von Insert-Operationen für Bit-Pakete

Insert-Operationen und die BDWFA:

- Einfügen eines Bit-Pakets in ein Wort durch beliebigen Teil-Graph des DWFG direkt an ⇧ -Werten ablesbar:

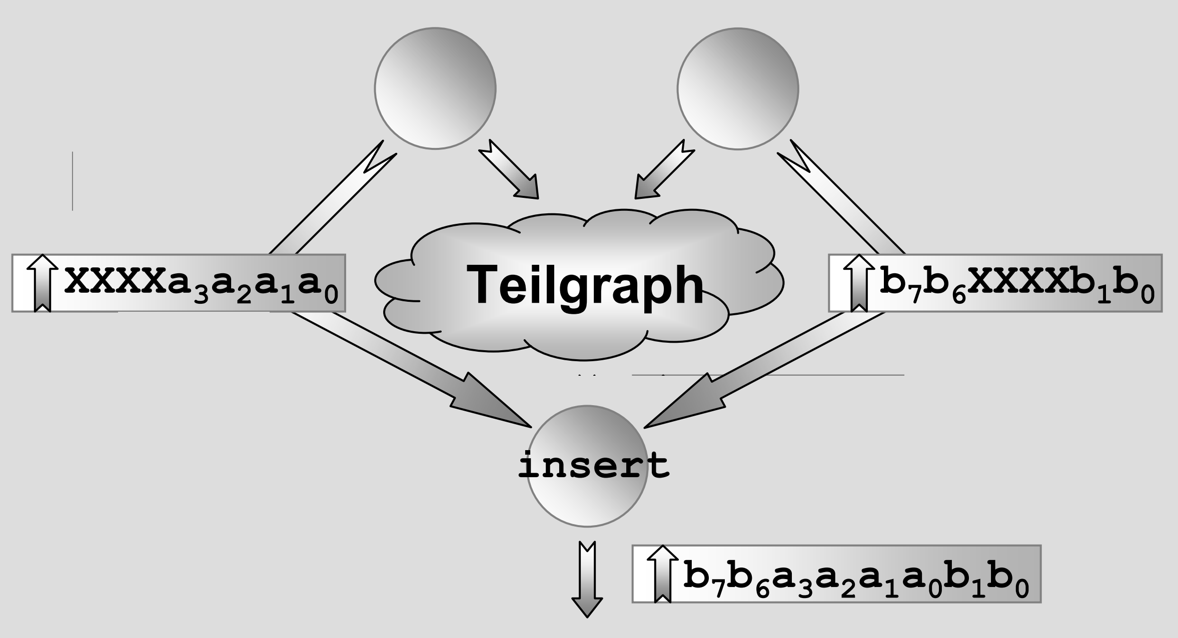


☞ *Eine Optimierung muss den ⇧ -Wert einer Kante mittels* **`L`**-*Werten in disjunkte Bit-Pakete zerlegen und aus dieser Zerlegung passende insert-Operationen erzeugen.*

Anwendungen der BDWFA: Erzeugung von Insert-Operationen für Bit-Pakete

Insert-Operationen und die BDWFA *(Fortsetzung)*:

- Optimierung des Beispiels durch Einfügen einer insert-Operation und Anpassen von Kanten:

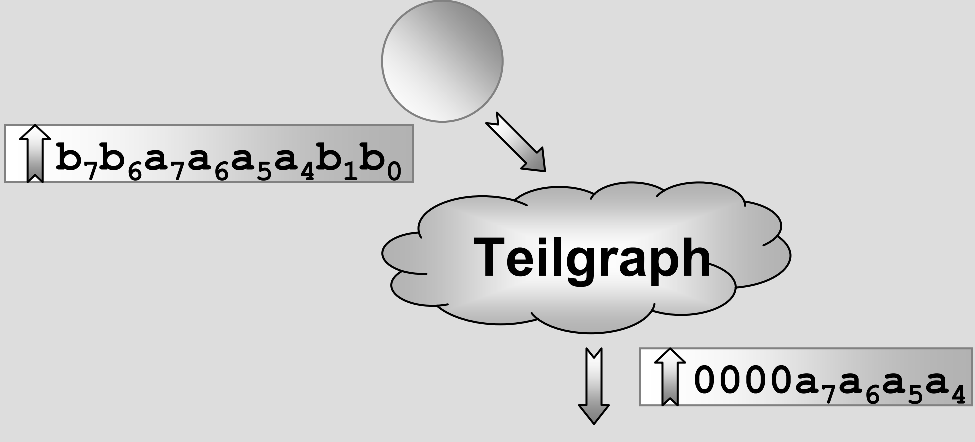


☞ *Sofern nach dem Kanten-Anpassen keine weiteren Kanten den Teil-graphen verlas-sen, kann dieser durch eine DCE entfernt werden!*

Anwendungen der BDWFA: Erzeugung von Extract-Operationen für Bit-Pakete

Extract-Operationen und die BDWFA:

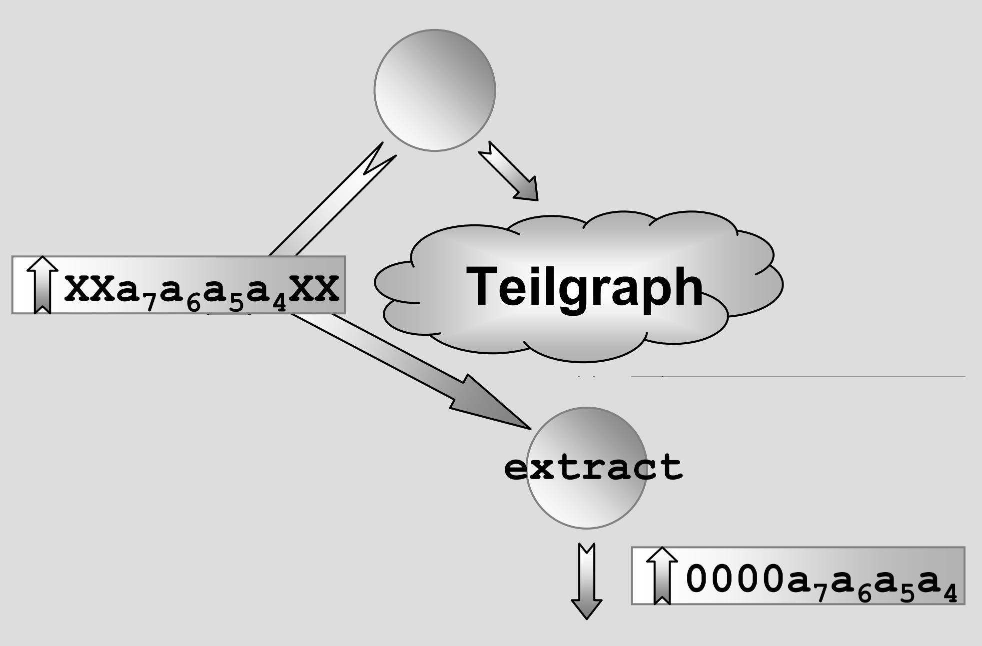
- Extraktion eines Bit-Pakets durch beliebigen Teil-Graph des DWFG auch anhand von ⇑-Werten ablesbar:



Anwendungen der BDWFA: Erzeugung von Extract-Operationen für Bit-Pakete

Extract-Operationen und die BDWFA *(Fortsetzung)*:

- Optimierung des Beispiels durch Einfügen einer extract-Operation und Anpassen von Kanten:

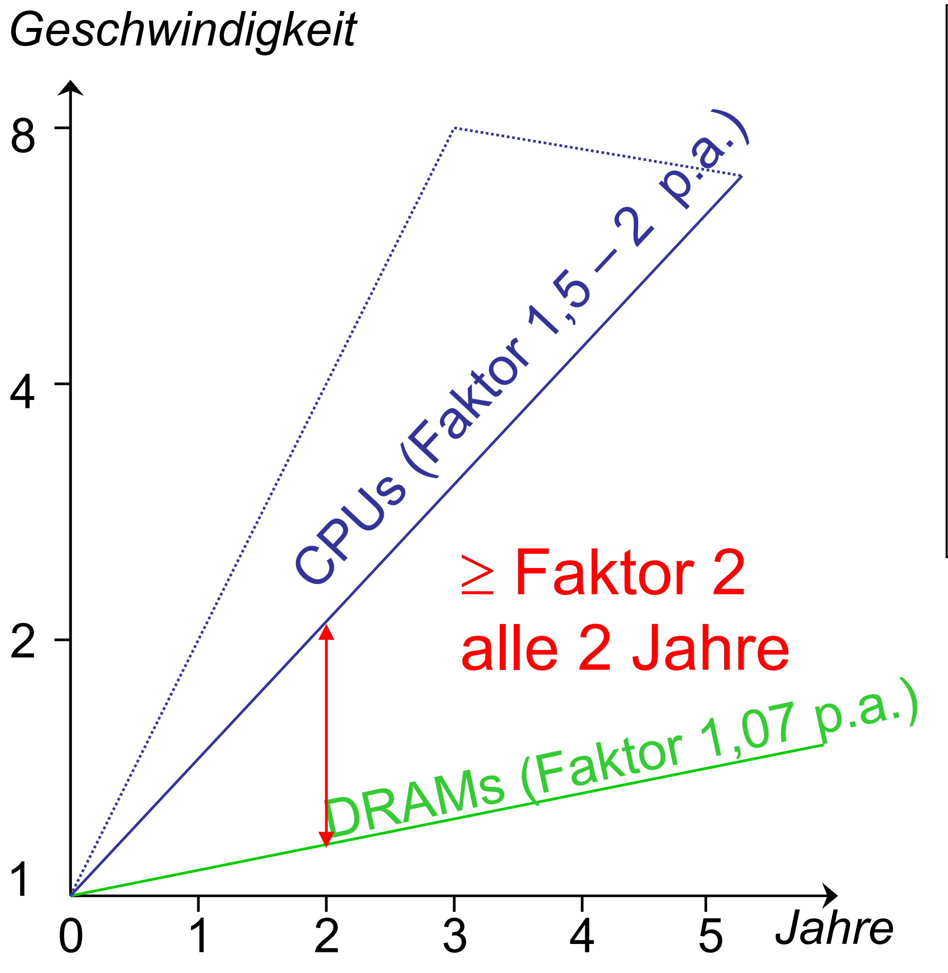


☞ *U.U. kann der Teilgraph auch wieder durch eine DCE entfernt werden.*

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- Kapitel 5: Semantische Analyse
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- **Kapitel 8: Code-Optimierung**
  - HIR: Parallelisierung für Homogene Multi-DSPs
  - LIR: Generierung von Bit-Paket Operationen für NPUs
  - LIR: Optimierungen für Scratchpad-Speicher
- Kapitel 9: Ausblick

Eigenschaften heutiger Speicher (1)

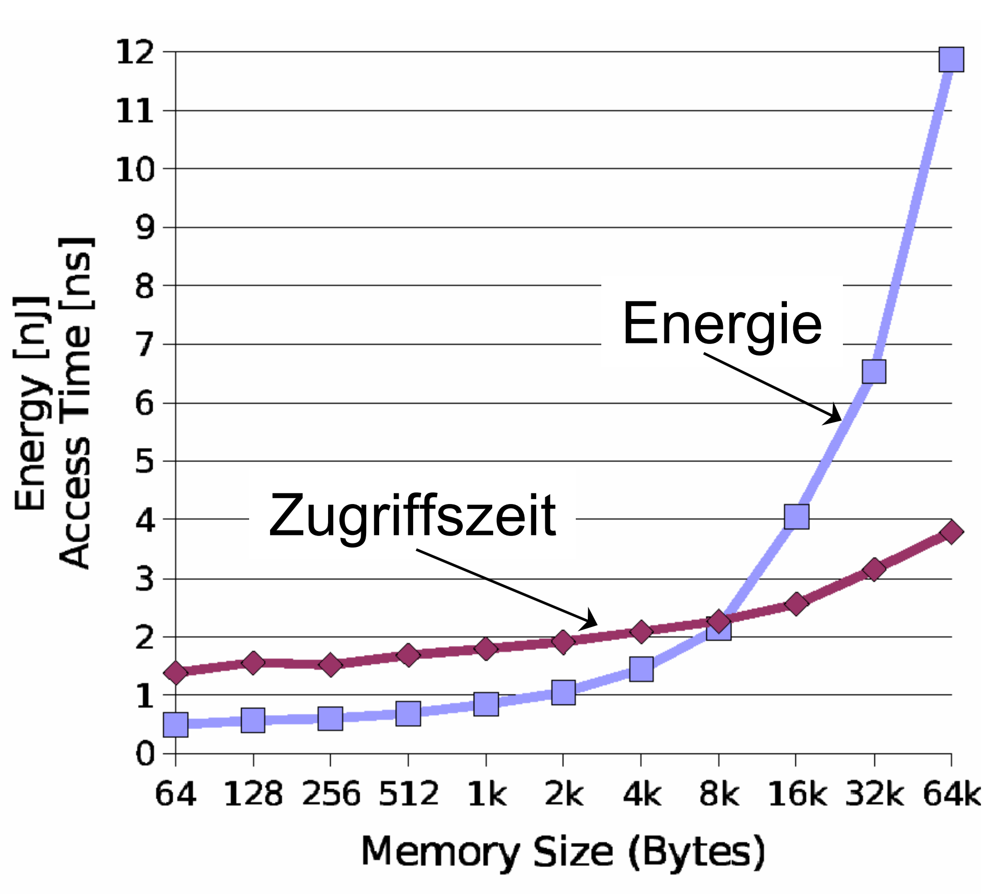


- Geschwindigkeitsunterschied zwischen CPUs und DRAMs verdoppelt sich alle 2 Jahre.
- Schnelle CPUs werden massiv durch langsame Speicher ausgebremst.

☞ *„Memory Wall"-Problem*

*[**P. Machanik,** Approaches to Addressing the Memory Wall, Technical Report, Universität Brisbane, Nov. 2003]*

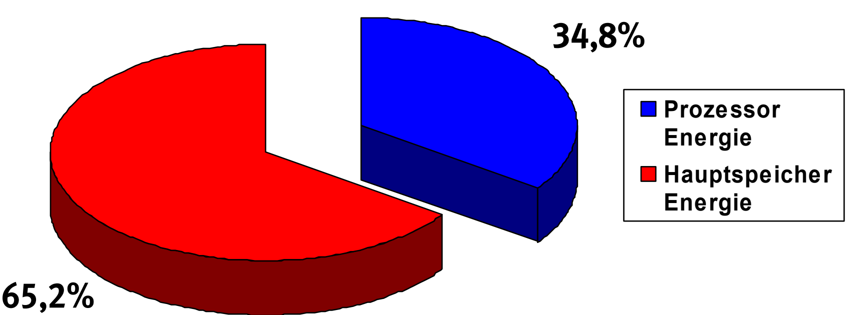
Eigenschaften heutiger Speicher (2)



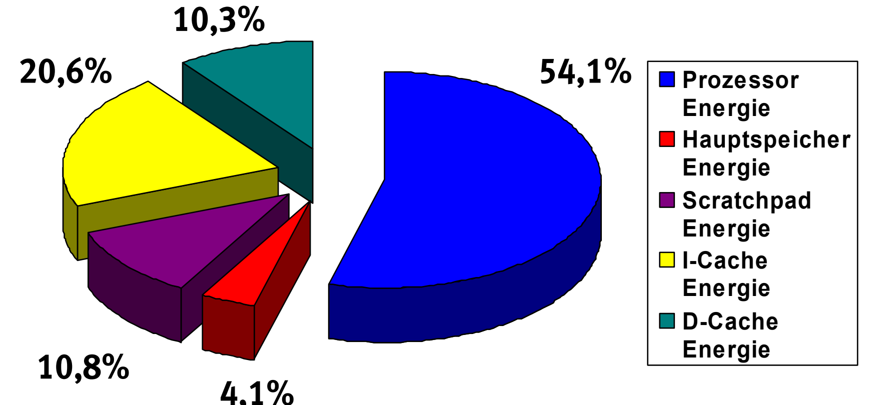
- Mit zunehmender Größe eines Speichers verbraucht ein Speicherzugriff überproportional mehr Energie.
- Mit zunehmender Größe dauern Speicherzugriffe auch proportional länger.
- ☞ *Fertigungstechnologie von Speichern legt Nutzung kleiner Speicher nahe!*

Eigenschaften heutiger Speicher (3)

- ARM7 Mono-Prozessor ohne Cache:



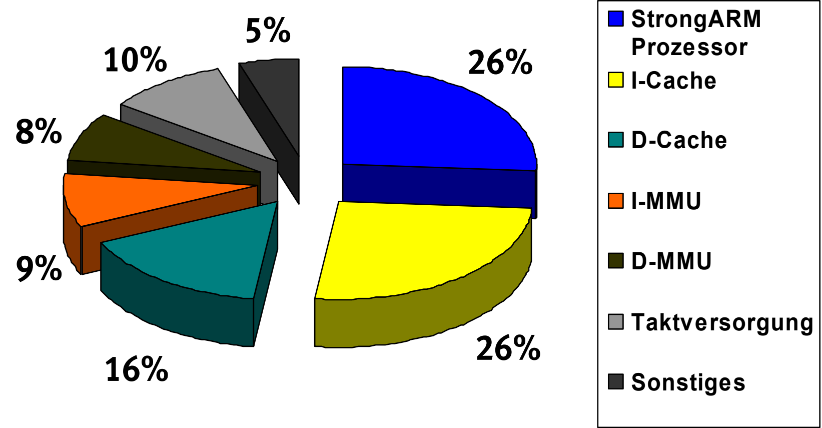
- ARM7 Multi-Prozessor mit Caches:

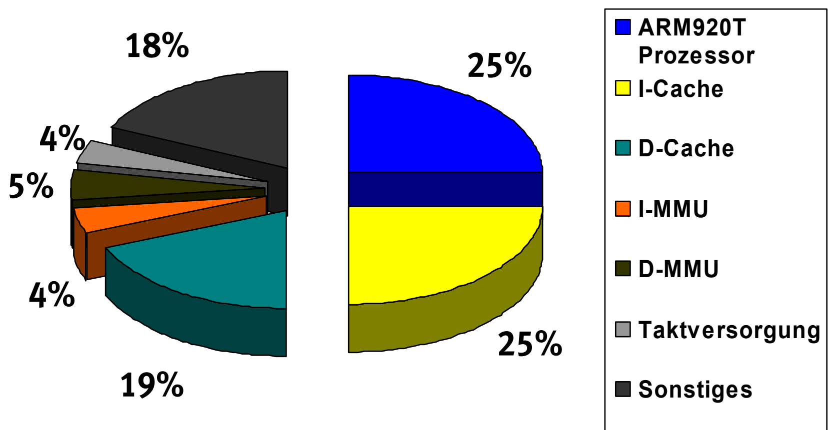


- Speicher-Subsystem verursacht häufig weit mehr als 50% des gesamten Energieverbrauchs.
- Tortendiagramme zeigen Durchschnitt über jeweils mehr als 160 verschiedene Energie-Messungen

*[**M. Verma, P. Marwedel,** Advanced Memory Optimization Techniques for Low-Port Embedded Processors, Springer, 2007]*

Eigenschaften heutiger Speicher (4)





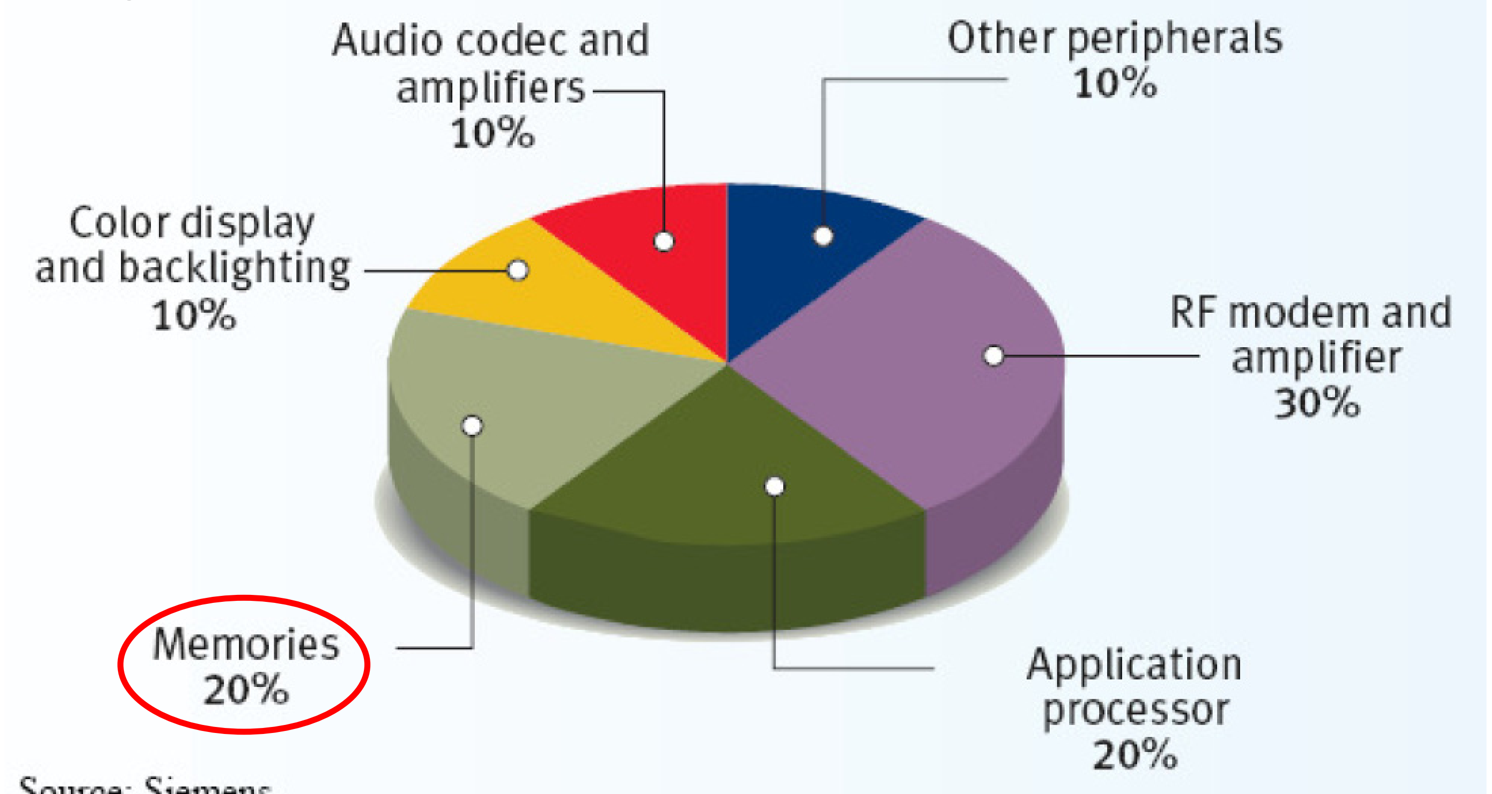
*[**O. S. Unsal, I. Koren, C. M. Krishna, C. A. Moritz,** U. of Massachusetts, Amherst, 2001]*

- Größenordnungen des Energieverbrauchs von Speichern durch Arbeiten anderer Gruppen aus Industrie & Forschung bestätigt.

*[**S. Segars** (ARM Ltd.), Low power design techniques for microprocessors, ISSCC 2001]*

Eigenschaften heutiger Speicher (5)

- Energieverbrauch mobiler Geräte:



*[**O. Vargas** (Infineon)**,** Minimum power consumption in mobile-phone memory subsystems, Pennwell Portable Design, Sep. 2005]*

Scratchpad-Speicher

- Scratchpads *(SPMs)* sind kleine, physikalisch separate Speicher.
- Sie sind meist auf dem selben Chip platziert wie der Prozessor (sog. *on-chip Speicher*).

☞ *Durch geringe Größe und on-chip Platzierung: extrem schnelle und energieeffiziente Speicher*

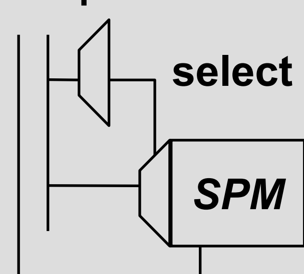
- Sind in den Adressraum des Prozessors nahtlos eingeblendet:

0x000…

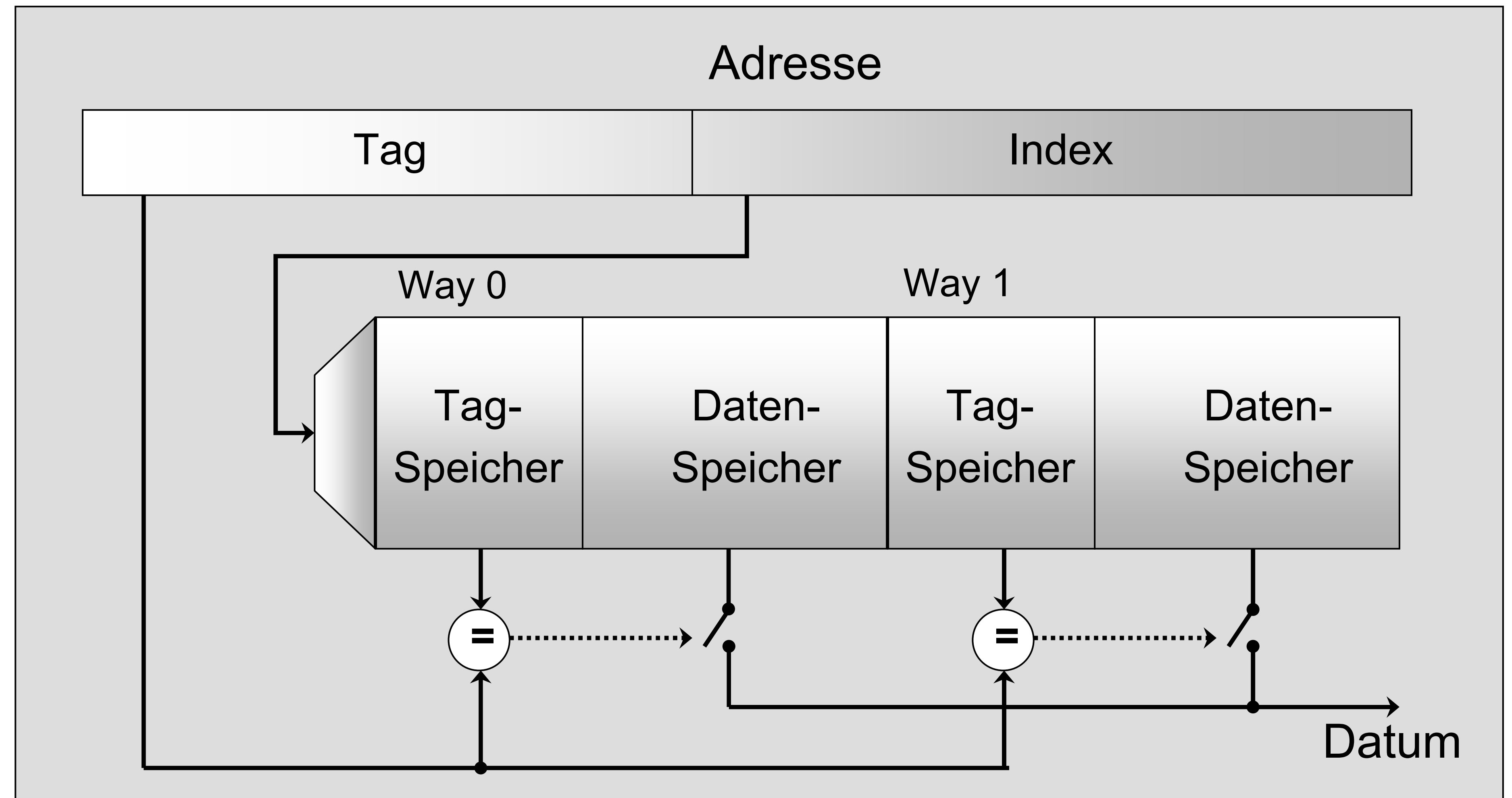
Scratchpad-Speicher

0xFFF…

- Zugriff über Erkennen einer am Bus anliegenden Adresse aus SPM-Adressbereich (simpler Adress-Decoder):



Aufbau mengenassoziativer Caches



Eigenschaften von Scratchpad-Speichern (1)

Vorhersagbarkeit:

- Jeder Zugriff auf SPM braucht lediglich konstante Zeit, üblicherweise 1 Taktzyklus.
- Im Gegensatz dazu: ein Cache-Zugriff kann variable Zeit brauchen, je nachdem, ob er zu einem Cache-Hit oder Cache-Miss führt.
- ☞ Laufzeitverhalten von Scratchpad-Speichern ist zu 100% vorhersagbar, während Verhalten von Caches schwer bis unmöglich vorherzusagen ist.
- ☞ Caches sind nur eingeschränkt realzeitfähig, während SPMs gerade in harten Echtzeitsystemen eingesetzt werden.

Eigenschaften von Scratchpad-Speichern (2)

Stromverbrauch im Vergleich zu Hauptspeicher:

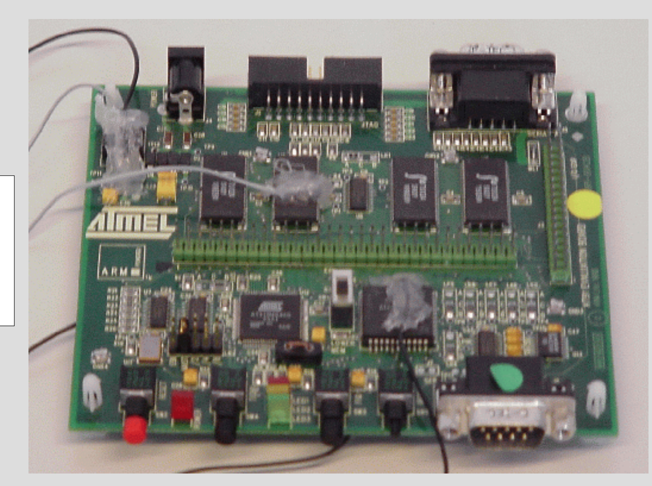
- Messungen an realer Hardware (Atmel ARM7-Evaluationsboard) zeigen, dass z.B. ein Lade-Befehl um Faktor 3 weniger Strom verbraucht, wenn sowohl Lade-Befehl als auch zu ladendes Datum im SPM anstatt im (off-chip) Hauptspeicher liegen:

Stromverbrauch Lade-Befehl

mA

200
150
100
50
0

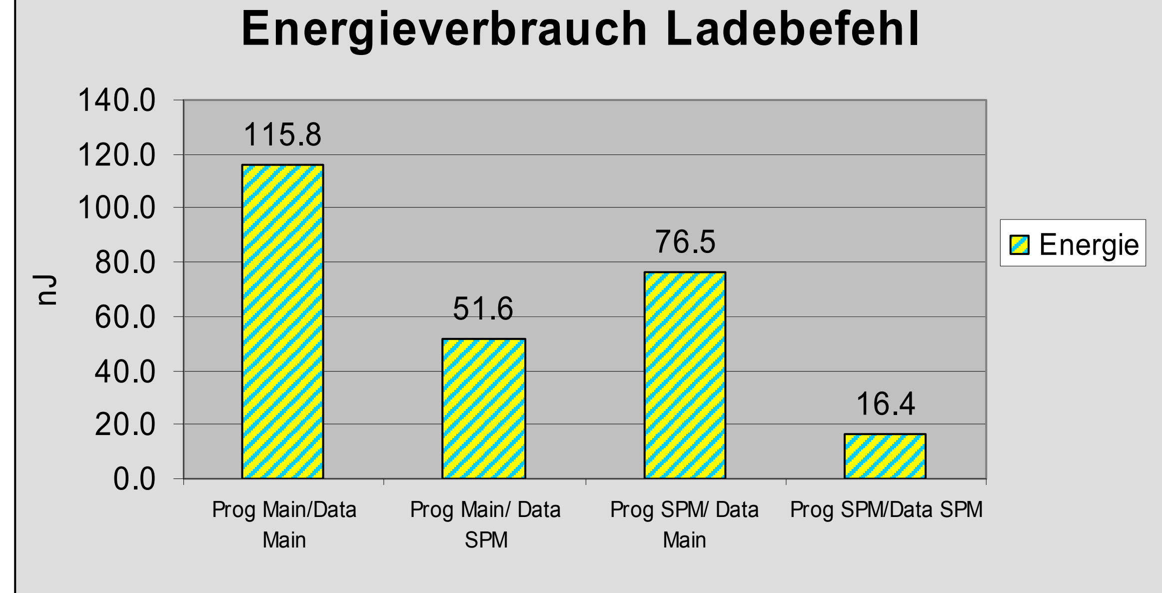
	Prog Main/ Data Main	Prog Main/ Data SPM	Prog SPM/ Data Main	Prog SPM/ Data SPM
Haupt-Speicher	116	77.2	82.2	1.16
ARM7 + SPM	48.2	50.9	44.4	53.1



Eigenschaften von Scratchpad-Speichern (3)

Energieverbrauch im Vergleich zu Hauptspeicher:

- Ähnliche Messungen an gleicher Hardware zeigen, dass Energieverbrauch eines Lade-Befehls um Faktor 7 reduziert werden kann:



Erinnerung:

$$E = \int P \, dt = \int (V * I) \, dt$$

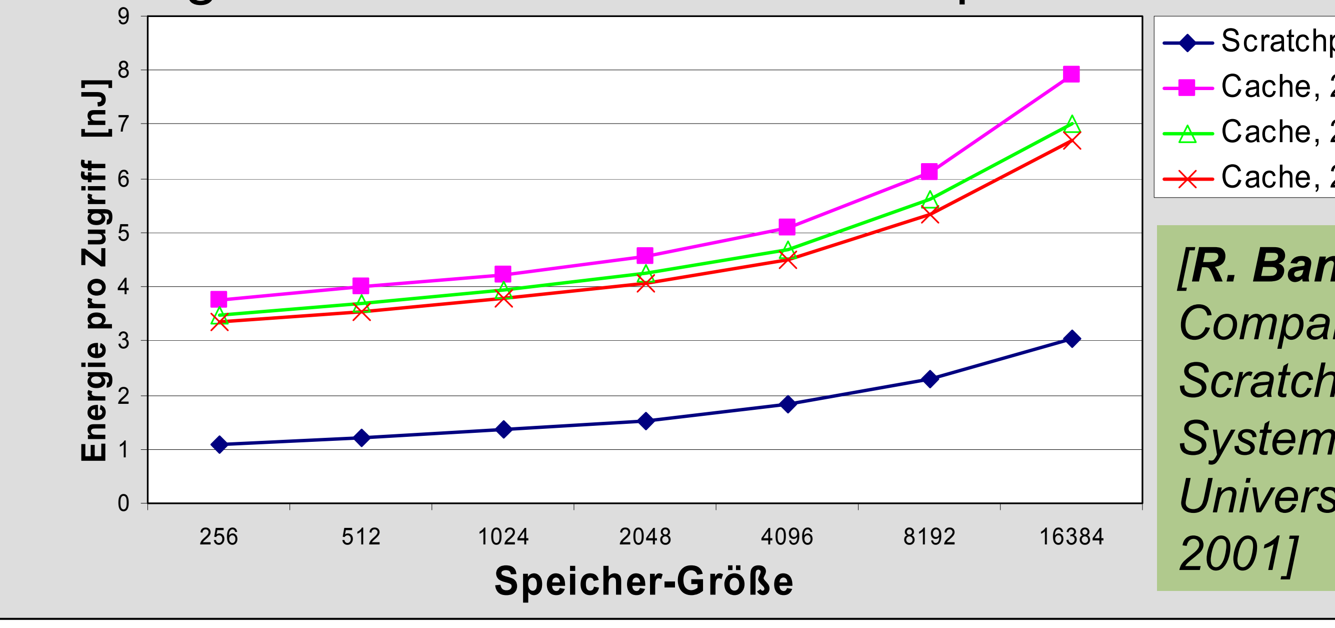
Annahme: Versorgungsspannung konstant und Stromverbrauch nicht zu variabel über Zeit

$$E \cong V * I * t$$

Eigenschaften von Scratchpad-Speichern (4)

Energieverbrauch im Vergleich zu Caches:

- Größe und Anzahl von Tag-Speichern, Vergleichern und Multiplexern hängt von Größe des gecacheten Speicherbereichs ab.
- Energieverbrauch dieser HW-Komponenten beträchtlich:



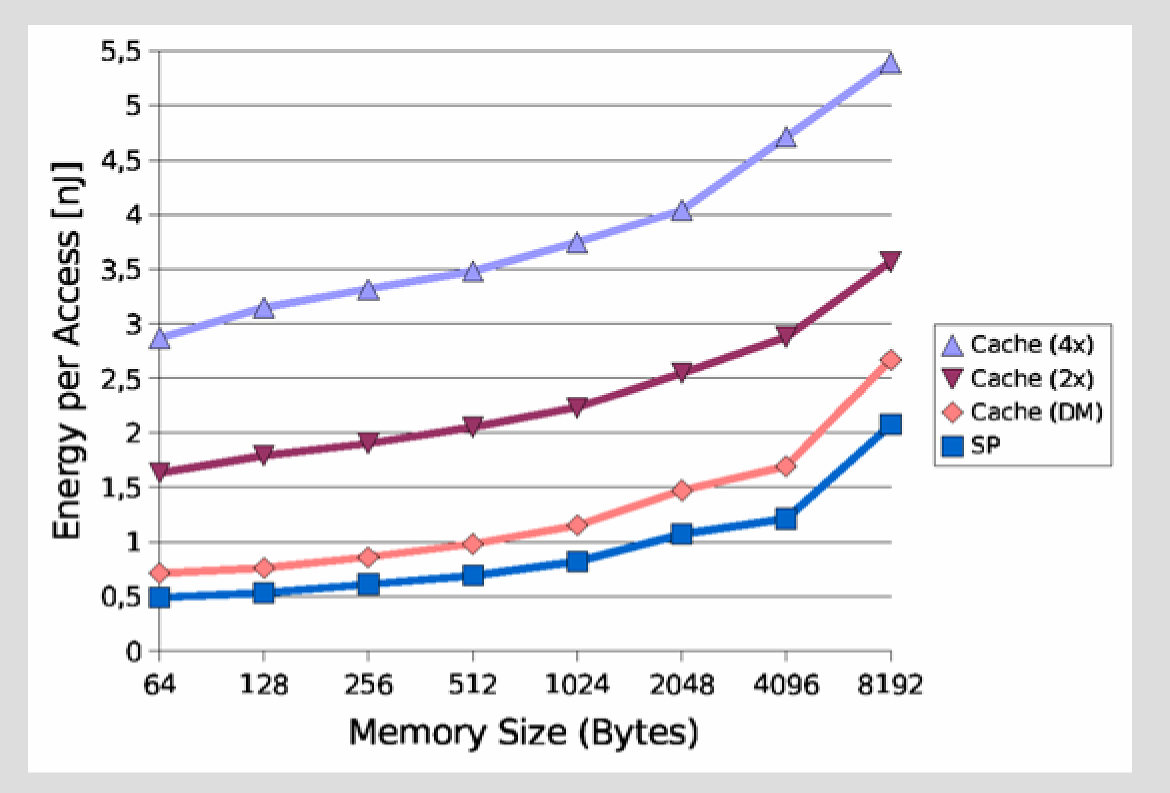
*[**R. Banakar et al.**, Comparison of Cache- and Scratch-Pad based Memory Systems..., Report #762, Universität Dortmund, Sep. 2001]*

Eigenschaften von Scratchpad-Speichern (5)

Energieverbrauch im Vergleich zu Caches:

- Energieverbrauch von Caches hängt zusätzlich stark vom Grad der Assoziativität ab:

<u>Vorsicht:</u> Technologie bei diesem Diagramm unterschiedlich zu der von Folie 79. Daher Abweichungen in absoluten Zahlen-werten!



Ganzzahlig lineare Programmierung

Modellierungstechnik für lineare Optimierungsprobleme:

- Optimierung einer *Zielfunktion z* unter Beachtung von *Nebenbedingungen* $n_1, ..., n_m$
- Zielfunktion und Nebenbedingungen sind lineare Ausdrücke über den *ganzzahligen Entscheidungsvariablen* $x_1, ..., x_n$

$$z: \sum_{i=1}^{n} c_i * x_i \quad \rightarrow \text{minimieren oder maximieren}$$

$$n_j: \sum_{i=1}^{n} a_{j,i} * x_i \leq b_j$$

Konstanten $a_{j,i}, b_j, c_i \in \mathbb{R}$
Variablen $x_i \in \mathbb{Z}$

- Optimale Lösung sog. ILPs *(Integer Linear Programs)* mit Hilfe von Standard-Algorithmus *(☞ Simplex-Verfahren)*;
  Komplexität: im worst-case exponentiell, üblicherweise aber „OK".

Fixe SPM-Allokation: Funktionen & Globale Variablen (1)

Ziel:

- Verschiebung des Codes von kompletten LIR-Funktionen und von globalen Variablen in den SPM
  (lokale Variablen liegen üblicherweise auf dem Stack und werden daher nicht betrachtet)
- Compiler ermittelt zur *Übersetzungszeit*, welche Funktionen und globalen Variablen den SPM belegen.
  Diese SPM-Belegung bleibt zur *Ausführungszeit* eines optimierten Programms fix, d.h. der SPM-Inhalt ändert sich zur gesamten Ausführungszeit nicht.

Fixe SPM-Allokation: Funktionen & Globale Variablen (2)

Definitionen:

- $MO = \{mo_1, ..., mo_n\} = F \cup V$ — Menge aller für die Verschiebung auf den SPM in Frage kommender Speicherobjekte *(memory objects)*, d.h. Funktionen F bzw. globale Variablen V.
- S — Größe des verfügbaren SPMs in Bytes.
- S_i — Größe von Speicherobjekt mo_i in Bytes.
- Δe_i — Eingesparte Energie, wenn mo_i von Hauptspeicher in SPM verschoben wird, pro einzelner Ausführung von $mo_i \in F$ bzw. pro einzelnem Zugriff auf $mo_i \in V$.

Fixe SPM-Allokation: Funktionen & Globale Variablen (3)

Definitionen *(Fortsetzung)*:

- n_i — Gesamt-Anzahl von Ausführungen bzw. Zugriffen auf mo_i
- ΔE_i — Gesamte eingesparte Energie, wenn mo_i von Hauptspeicher in SPM verschoben wird, pro kompletter Ausführung des zu optimierenden Programms (= $n_i * \Delta e_i$)
- x_i — Binäre Entscheidungsvariable zu mo_i
  $x_i = 1 \Leftrightarrow mo_i$ wird in SPM verschoben

Fixe SPM-Allokation: Funktionen & Globale Variablen (4)

Bestimmung der Parameter:

- S: Vom Anwender vorgegeben, konstant
- S_i: Mit Hilfe einer LIR leicht zu bestimmen: Entweder Summe über die Größe aller Instruktionen einer Funktion, oder Summe über die Größen aller Teil-Variablen, z.B. bei Feldern oder Strukturen
- Δe_i: Für $mo_i \in V$: Energiemodell *(vgl. ☞ Kapitel 2)* liefert Differenz zwischen Zugriff auf Hauptspeicher und SPM

  Für $mo_i \in F$: Energiemodell liefert Differenz Δe_{IFetch} zwischen Instruction Fetch aus Hauptspeicher bzw. SPM. Simulation des zu optimierenden Programms liefert Anzahl $n_{i,instr}$ ausgeführter Instruktionen für mo_i.

  $\Delta e_i = n_{i,instr} * \Delta e_{IFetch}$.

Fixe SPM-Allokation: Funktionen & Globale Variablen (5)

Bestimmung der Parameter *(Fortsetzung)*:

- n_i: Gleicher Simulationsdurchlauf wie zur Bestimmung von Δe_i liefert Ausführungs- und Zugriffshäufigkeiten für mo_i.
- ☞ Vor der eigentlichen Scratchpad-Optimierung eines Programms findet ein Simulationsdurchlauf statt, um zur Optimierung notwendige Parameter zu ermitteln.
- ☞ Ein solcher Simulationsdurchlauf erzeugt ein Laufzeit-Profil des zu optimierenden Programms. Daher:
  Eine solche Simulation vor einer Optimierung heißt *Profiling*.

Fixe SPM-Allokation: Funktionen & Globale Variablen (6)

ILP-Formulierung:

- Zielfunktion: Maximiere Energieeinsparung für gesamtes Programm

  $$z: \sum_{i=1}^{n} \Delta E_i * x_i \quad \rightarrow \text{max.}$$

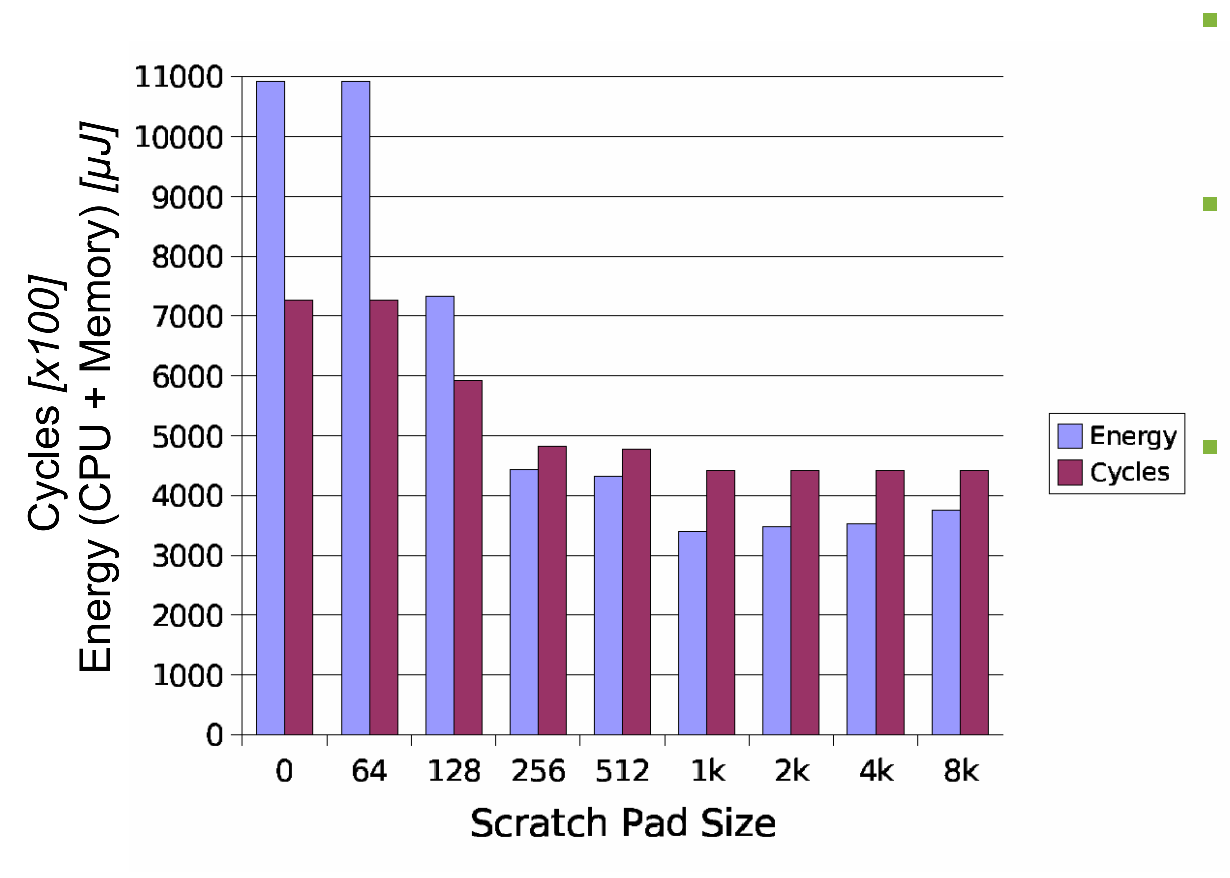
- Nebenbedingung: Einhaltung der Kapazität des SPMs

  $$n_1: \sum_{i=1}^{n} S_i * x_i \leq S$$

*[**S. Steinke,** Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik, Dortmund 2002]*

Fixe SPM-Allokation: Funktionen & Globale Variablen (7)

Ergebnisse *(MultiSort-Benchmark)*:



- 64b SPM zu klein, um für globale Variablen / Funktionen ausgenutzt zu werden.
- Bis 1kB SPM stetige Verbesserung von Energie & Laufzeit wg. Einlagerung von mehr MOs in den SPM.
- Ab 2kB SPM leichte Verschlechterungen, da keine weiteren MOs mehr in SPM eingelagert werden können (alle MOs bereits im SPM enthalten), der Energieverbrauch größerer SPMs aber technologiebedingt ansteigt.

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (1)

Motivation:

Verschiebung kompletter Funktionen unter Umständen nachteilig:

- ✘ Ganze Funktionen haben viel Code und benötigen viel SPM-Platz.
- ✘ Einzelne Code-Teile einer Funktion (z.B. Code außerhalb von Schleifen) werden nur selten ausgeführt, führen daher nur zu geringer Energieeinsparung, werden aber dennoch auf SPM gelegt.
- ☞ *(Knappe) SPM-Kapazität wird nur suboptimal ausgenutzt.*

Ziel:

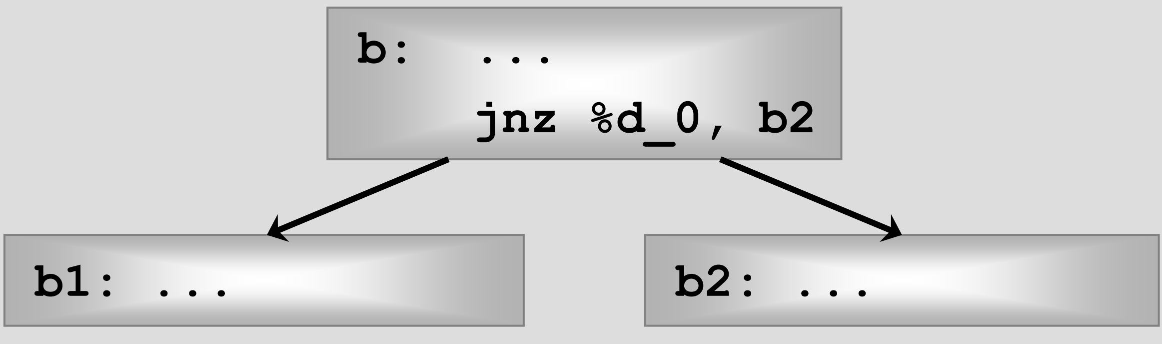
Verschiebung des Codes von kompletten LIR-Funktionen, *von einzelnen Basisblöcken* und von globalen Variablen in den SPM.

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (2)

Problem beim Verschieben einzelner Basisblöcke:

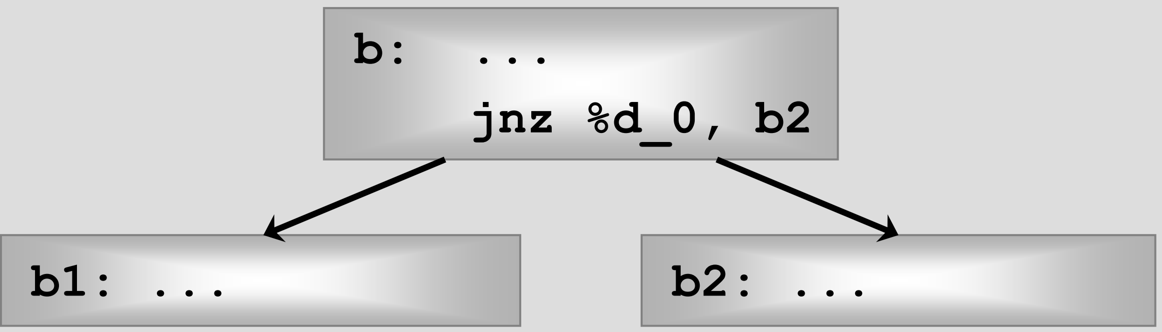
Erinnerung: Ein Basisblock *b* darf nur als letzten Befehl einen Sprungbefehl enthalten

- Ist der Sprung am Ende von *b* bedingt, so hat *b* im CFG zwei Nachfolger *b*1 und *b*2, die ausgeführt werden, wenn der bedingte Sprung entweder genommen wird oder nicht:



Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (3)

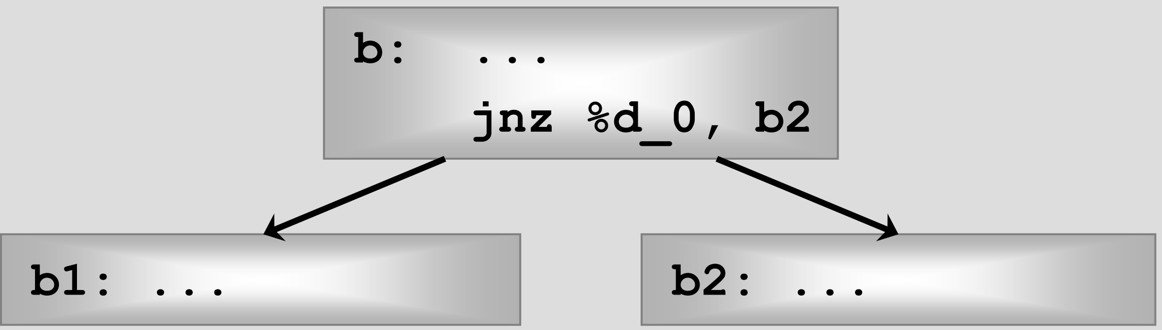
Problem beim Verschieben einzelner Basisblöcke:



- *b*1 wird von *b* aus implizit erreicht, wenn der bedingte Sprung nicht genommen wird, da
- ✓ der Programmzähler nach dem nicht genommenen Sprung inkrementiert wird und auf den nächsten folgenden Befehl zeigt, und
- ✓ der Code von *b*1 im Speicher direkt auf *b* folgt.

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (4)

Problem beim Verschieben einzelner Basisblöcke:



Was, wenn b im SPM liegt, b1 aber nicht (oder umgekehrt)?

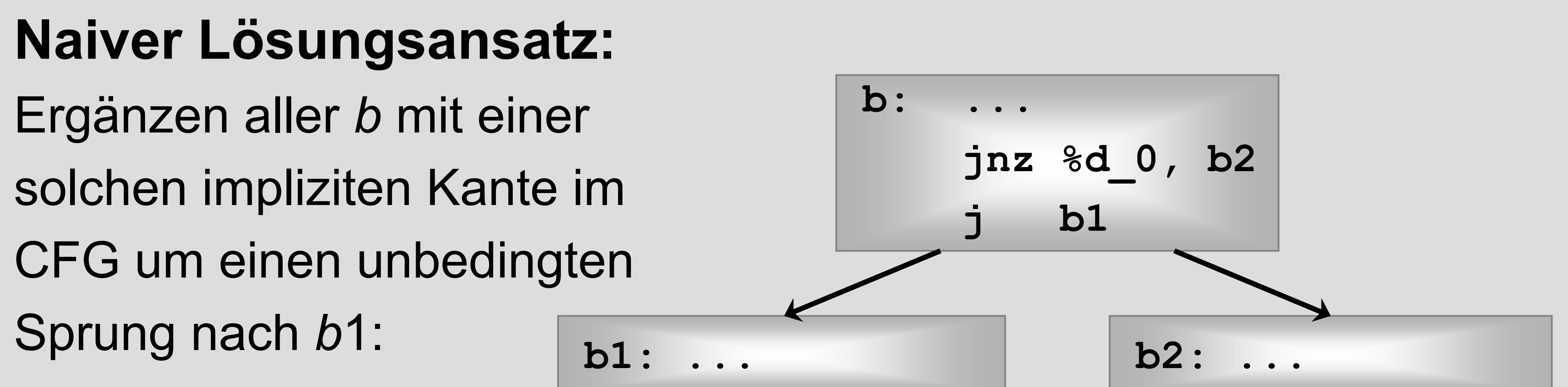
- Wird der Sprung nicht genommen, wird der nächste folgende Befehl aus dem Scratchpad-Speicher ausgeführt.

☞ *Da b1 nicht mehr auf b im Speicher folgt, wird inkorrekter Code ausgeführt!*

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (5)

Naiver Lösungsansatz:

Ergänzen aller *b* mit einer solchen impliziten Kante im CFG um einen unbedingten Sprung nach *b*1:



Nachteil:

- Unbedingter Sprung extrem ineffizient (Codegröße, Laufzeit und Energie), wenn *b* und *b*1 doch im gleichen Speicher liegen sollten.

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (6)

Eleganter Lösungsansatz:

Ergänzen eines Basisblocks *b* mit solcher impliziten Kante um unbedingten Sprung wirklich nur dann, wenn *b* und *b*1 unterschiedlichen Speichern zugeordnet sind.

Vorteil:

- Unbedingte Sprünge werden nur zusätzlich eingefügt, wo dies auch wirklich notwendig ist.

Problem:

- Codegröße S_b von *b* hängt jetzt von der Entscheidungsvariablen x_b ab, die die Speicher-Zuordnung von *b* im ILP modelliert.

☞ *Wie modelliert man nicht-konstanten Parameter S_b im ILP?*

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (7)

Multi-Basisblöcke:

Mengen einzelner Basisblöcke, die jeweils im CFG zusammenhängend sind.

Sei *G* der CFG einer Funktion *f*, *G'* ein zusammenhängender Teilgraph von *G*. Die Menge aller Basisblöcke aus *G'* stellt einen Multibasisblock dar.

```
b:  ...
b1:  ...
b2:  ...
```

{*b*, *b*1}, {*b*, *b*2} und {*b*, *b*1, *b*2} sind Multi-Basisblöcke.

{*b*1, *b*2} ist kein Multi-Basisblock: *G'* hierzu ist unzusammenhängend.

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (8)

(Multi-) Basisblöcke in einer ILP-Formulierung:

- ILP zur SPM-Allokation betrachtet Mengen aller Funktionen F, aller einzelnen Basisblöcke B, aller Multi-Basisblöcke MB und aller globalen Variablen V als Speicherobjekte.
- MB wird gebildet durch Betrachtung aller zusammenhängender Teilgraphen G' des CFGs.

Definitionen:

- $MO = \{mo_1, ..., mo_n\} = F \cup B \cup MB \cup V$ Menge aller für die Verschiebung auf den SPM in Frage kommender Speicherobjekte *(memory objects)*
- Bedeutung aller anderen Ausdrücke (S, S_i, Δe_i, ...) wie vorher

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (9)

Bestimmung der Parameter:

- S_i: Für $mo_i \in F$ oder $mo_i \in V$: wie vorher;
  Für $mo_i \in B$: Größe aller Instruktionen des Basisblocks, plus Größe eines unbedingten Sprungs, falls mo_i impliziten Nachfolger hat;
  Für $mo_i \in MB$: Größe aller Instruktionen aller in mo_i enthaltenen Basisblöcke, plus Größe von k unbedingten Sprüngen, falls mo_i im CFG k implizite Nachfolger hat.
- Δe_i: wie vorher, nur jetzt analog zu S_i unter Berücksichtigung der neu zu beachtenden unbedingten Sprünge.
- n_i: wie vorher per Profiling, nur jetzt auch für $mo_i \in B \cup MB$.

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (10)

ILP-Formulierung:

- Zielfunktion: Maximiere Energieeinsparung für gesamtes Programm

$$z: \sum_{i=1}^{n} \Delta E_i * x_i \rightarrow \text{max.}$$

- Nebenbedingung 1: Einhaltung der Kapazität des SPMs

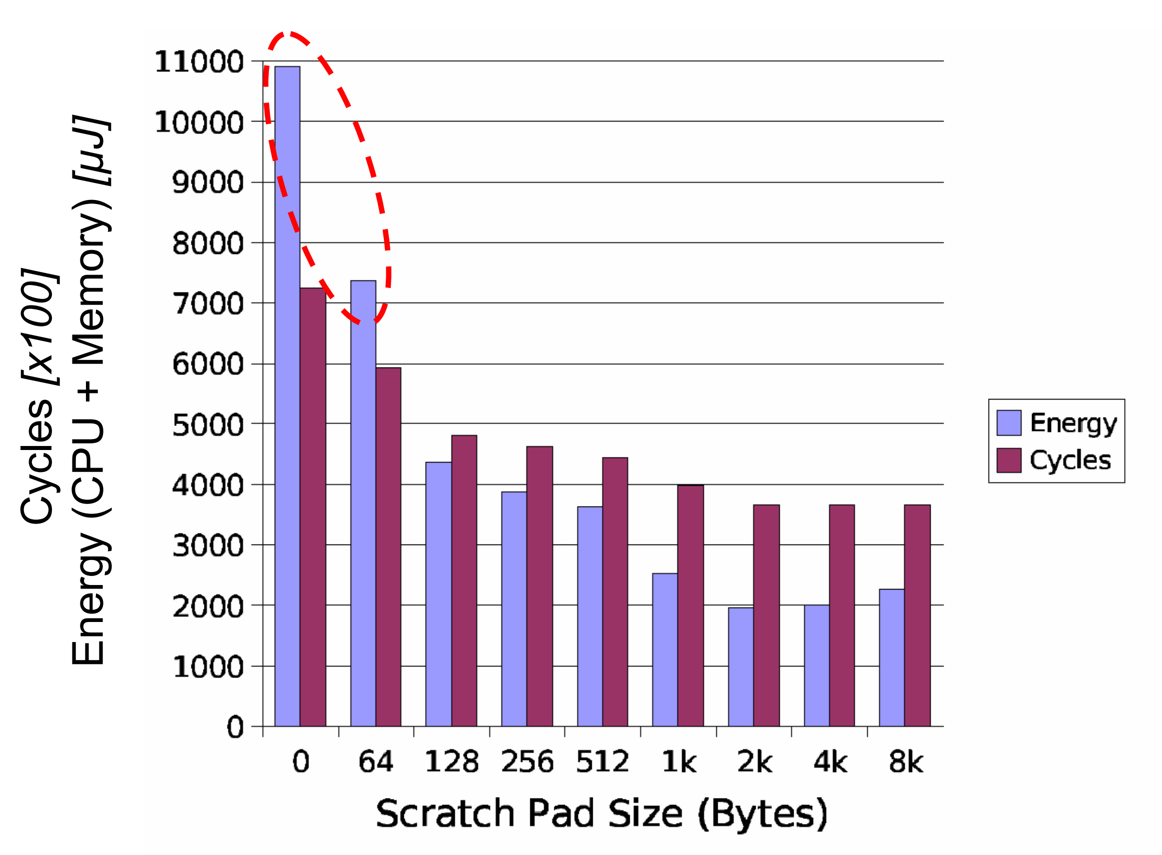
$$n_1: \sum_{i=1}^{n} S_i * x_i \leq S$$

- Nebenbedingung pro $mo_i \in B$: mo_i darf nur durch max. eine Variable (für *b* selbst, für *b*'s Funktion oder für alle Multi-Basisblöcke, die *b* enthalten) dem SPM zugewiesen werden

$$\forall\, mo_i \in B: x_i + x_{f(i)} + \sum_{mo_j \in MB:\, mo_i \in mo_j} x_j \leq 1$$

Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (11)

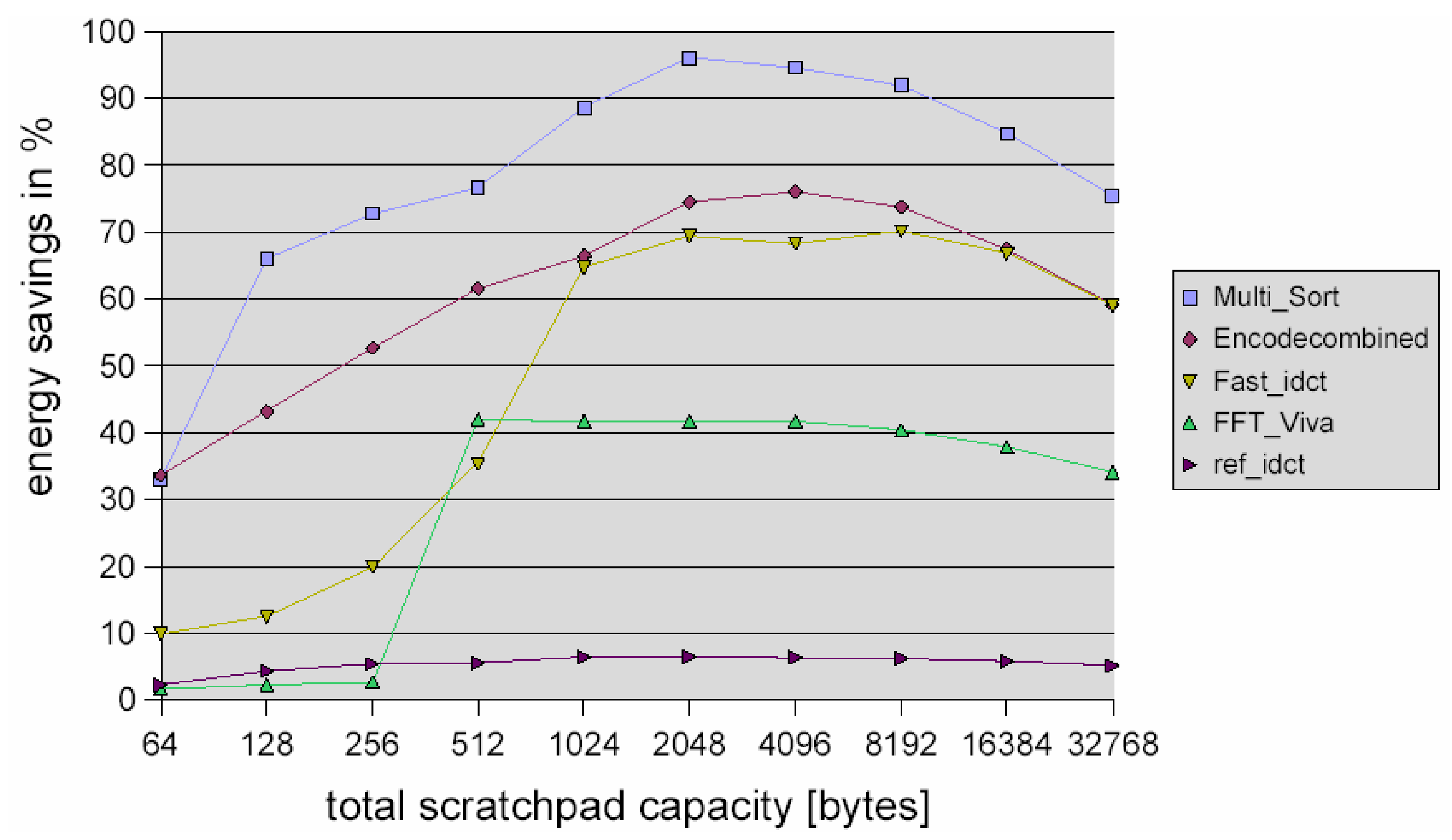
Ergebnisse *(MultiSort-Benchmark)*:



- 64b SPM jetzt nicht mehr zu klein, um für Code-Teile ausgenutzt zu werden.
- *Vorsicht: Für links stehendes Diagramm ist auch der Laufzeit-Stack als Speicher-Objekt betrachtet und somit auf den SPM verschoben worden. Daher ist dieses Diagramm nur bedingt mit Folie 88 vergleichbar!*

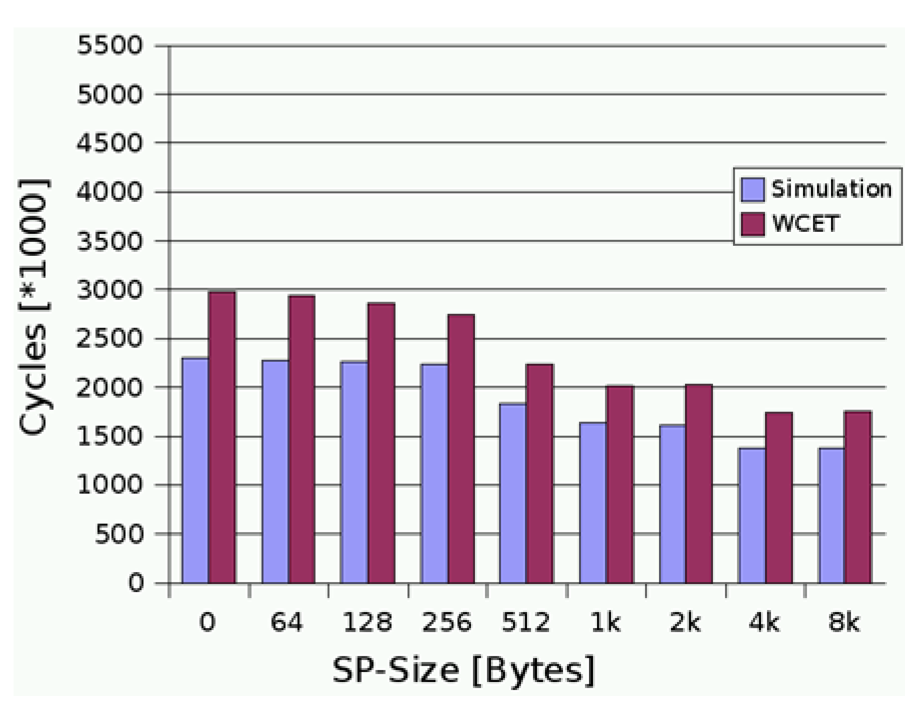
Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (12)

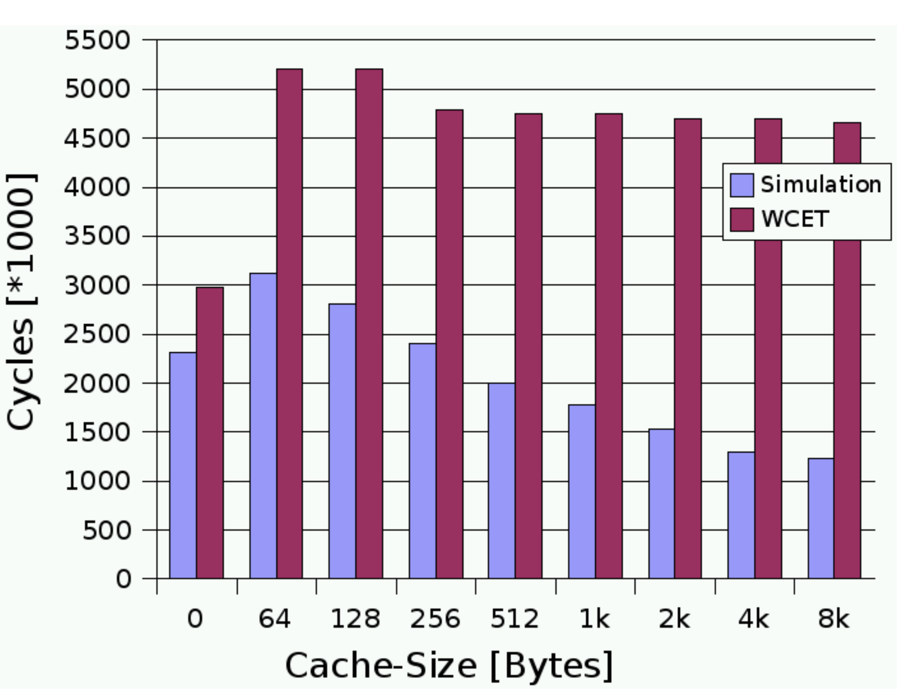
Detail-Ergebnisse nur für Speicher-Subsystem:



Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (13)

Vergleich ACET / $WCET_{EST}$ für Scratchpads und Caches:





- SPMs im Gegensatz zu Caches bestens vorhersagbar: $WCET_{EST}$ skaliert mit ACETs
- Erst für größere Speicher (ab 2kB) sind Caches auch bzgl. ACET besser als SPMs

Literatur

Parallelisierung für Homogene Multi-DSPs:

- B. Franke, M. O'Boyle, *A Complete Compiler Approach to Auto-Parallelizing C Programs for Multi-DSP Systems*, IEEE Transactions on Parallel and Distributed Systems 16(3), März 2005.

Code-Generierung für Netzwerk-Prozessoren:

- J. Wagner, *Retargierbare Ausnutzung von Spezialoperationen für Eingebettete Systeme mit Hilfe bitgenauer Wertflussanalyse*, Dissertation, Dortmund 2006.

Literatur

Optimierungen für Scratchpad-Speicher:

- S. Steinke, *Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik*, Dissertation, Dortmund 2002.
- M. Verma, P. Marwedel, *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, Springer, 2007.

Zusammenfassung

- **Parallelisierung für Homogene Multi-DSPs**
  - Fokus auf Ausnutzung lokaler Speicher & Adressbereiche
  - Speedups im wesentlichen linear zu Anzahl paralleler DSPs
- **Generierung von Bit-Paket Operationen für NPUs**
  - Konventionelle Datenflussanalysen sind nicht bitgenau
  - Bitgenaue DWFA per Vor- / Rückwärts-Simulation
  - Entdeckung von Bit-Paketen mittels ⇕-Werten der BDWFA
- **Optimierungen für Scratchpad-Speicher**
  - Scratchpads extrem vorteilhaft bzgl. Energieverbrauch, Laufzeit und $WCET_{EST}$, verglichen mit Caches und Hauptspeicher
  - SPM-Inhalt: Funktionen, Basisblöcke und globale Variablen