

Compiler für Eingebettete Systeme (CfES)

Sommersemester 2009

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

Kapitel 6

LIR Optimierungen und Transformationen

Gliederung der Vorlesung

- Kapitel 1: Compiler für Eingebettete Systeme
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- Kapitel 5: Instruktionsauswahl
- **Kapitel 6: LIR Optimierungen und Transformationen**
 - Generierung von Bit-Paket Operationen für NPUs
 - Optimierungen für Scratchpad-Speicher
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur WCET_{EST}-Minimierung
- 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 $(c \gg 4) \& 0x7$ 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 Instruktionauswahl:**

- Baum-Grammatik erzeugt LIR mit Maschinen-Operationen, die atomare Register als Operanden verwenden
- Baum-Grammatik erzeugt keine Bit-Paket Operationen

- Über Regeln

```
dreg: tpm_BinaryExpAND( dreg, const )
```

```
dreg: tpm_BinaryExpSHR( dreg, const )
```

würde Ausdruck $(c \gg 4) \& 0x7$ 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.***

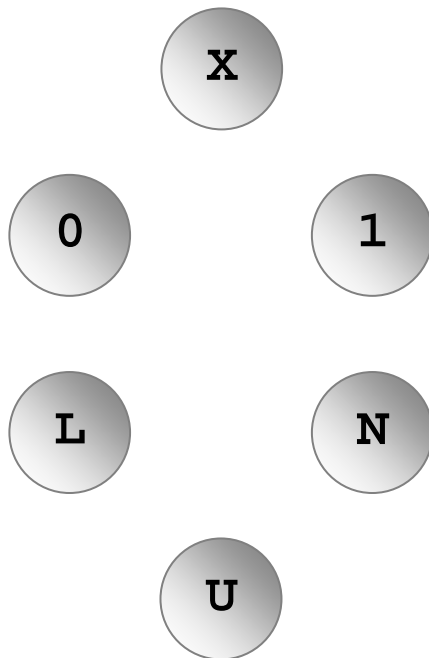
Daten- und Wertflußgraph (DWFG)

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 5)
- Seien $op_i(p_{i,1}, \dots, p_{i,n})$ und $op_j(p_{j,1}, \dots, p_{j,m})$ 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.

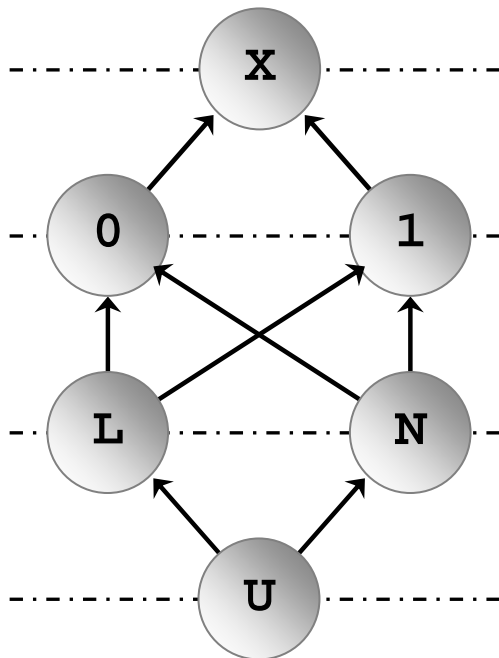
Die Halbordnung \mathcal{L}_4 (1)



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: \Uparrow repräsentiert Up-Wert, \Downarrow Down-Wert

Little Endian Darstellung (höchstwertiges Byte links geschrieben)

Beispielhafte Kanten-Informationen für e und deren Interpretation:

- \Uparrow UUUUUUUU Der Wert von r ist komplett unbekannt.
- \Uparrow 00101010 Der Wert von r ist gleich 42.
- \Uparrow 0010X010 Bit 3 von r ist irrelevant; r kann gleich 34 oder 42 sein.
- \Uparrow XXXXXXXX r ist komplett irrelevant
 \rightarrow 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:

↑ $00L_{r',2}00000$ Der Wert von Bit 5 von r ist unbekannt. Aber er ist identisch mit dem Wert von Bit 2 von r' .

↑ $00N_{r',4}00000$ Der Wert von Bit 5 von r ist unbekannt. Aber er ist identisch mit dem negierten Wert von Bit 4 von r' .

↑ $00L_{r',2}L_{r',1}L_{r',0}000$ r enthält ein Bit-Paket bestehend aus den Bits 2 bis 0 von r' , plaziert 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

\neg	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 \Downarrow -Werten für D
- \Downarrow -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. \Downarrow -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 \Downarrow -Werten aller eingehenden Kanten aus
 - Speichere neue \Downarrow -Werte der ausgehenden Kanten von v

Ablauf der Vorwärts-Analyse (1)

- $queue\langle DWFG_node \rangle q = \langle \text{Menge aller Quell-Knoten in } D \rangle$;
 $d(e) = \top^*$ für alle Kanten $e \in E$;
- while (! $q.empty()$)
 - $DWFG_node v = \langle \text{erstes Element aus } q \rangle$; $q.remove(v)$;
 $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
 - if ($\langle v \text{ repräsentiert konstante Zahl } c \rangle$)
 $d'(e) = \{0, 1\}^* = \langle \text{Binärdarstellung von } c \rangle$ für alle $e \in E_{out}$;
 - else
if ($\langle v \text{ repräsentiert unbekannte Eingangsvariable } i \text{ von } F \rangle$)
 $d'(e) = \{L_i\}^* = \langle \text{Bit-Locations von } i \rangle$ für alle $e \in E_{out}$;
 - else
 $d'(e) = \langle \text{Vorwärts-Simulation von } v \rangle$ 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 \Downarrow -Werte mit Elementen 0, 1 und \perp .
- Für ein k -Bit Register r , modelliert durch Kante e , berechnet die Vorwärts-Analyse zunächst einen temporären \Downarrow -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 \Downarrow -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 \Downarrow -Wert, in Abhängigkeit von den \Downarrow -Werten aller eingehenden Kanten

$$e_{in,1}, \dots, e_{in,N} \in E_{in}:$$

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

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 $0, \dots, k$ an und berechne Summen-Bits.

Vorwärts-Simulation (4)

Register-Transfer-Operationen:

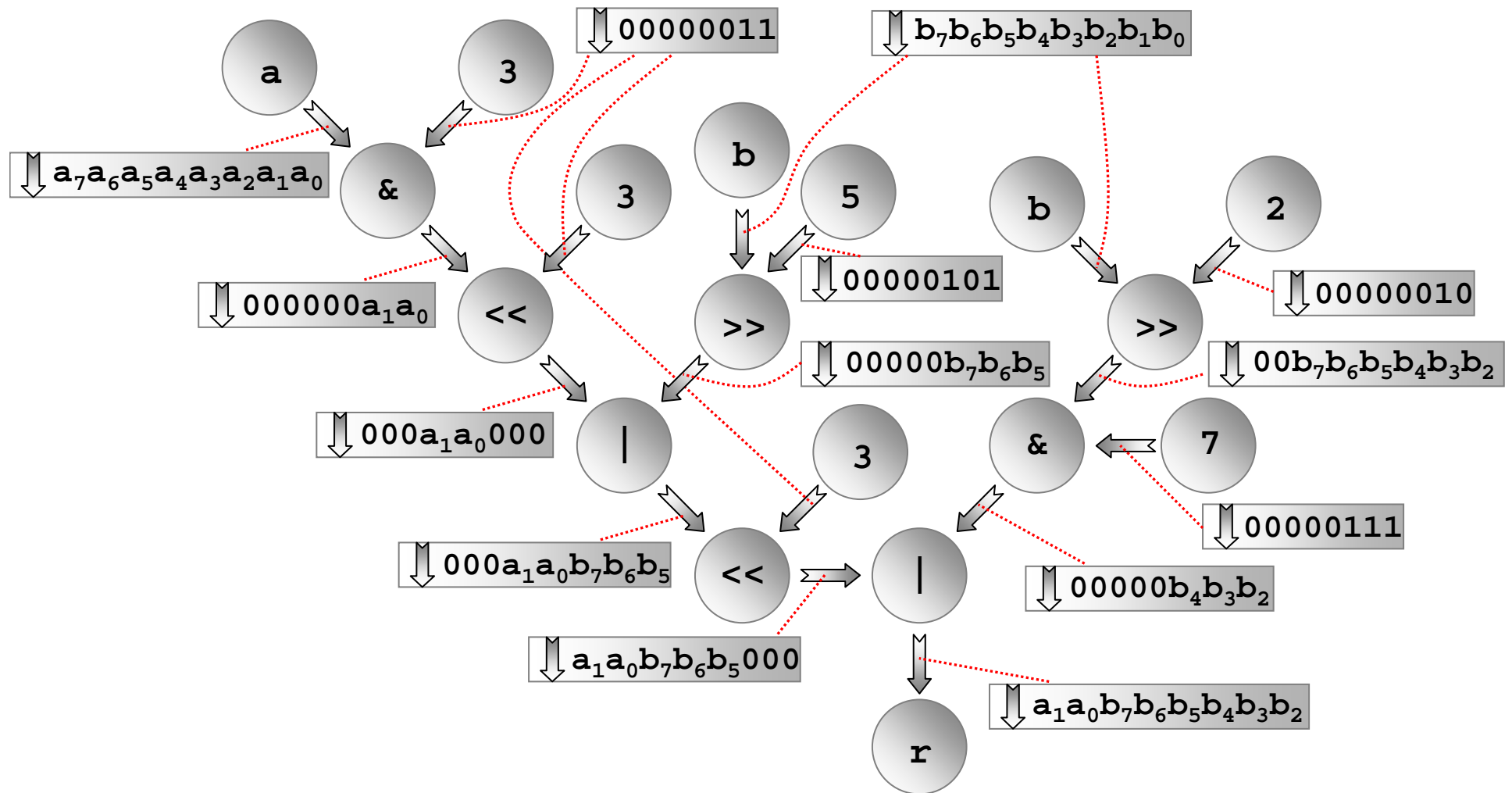
- Kopieren von Registerinhalten (Register-Register-Move) wird durch Kopieren von \Downarrow -Werten in \mathcal{L}_4^k erreicht.

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 \Downarrow -Werte, da dies typischerweise Senken im DWFG sind,
 - generieren Load-Operationen Υ^* als \Downarrow -Werte.

☞ Andere Klassen von Operationen werden ähnlich modelliert.

Beispiel Vorwärts-Simulation



Rückwärts-Analyse

Motivation & Ziel:

- Konjunktion, Disjunktion & Negation in \mathcal{L}_4 erzeugen \mathbf{x} nur dann, wenn einer ihrer Operanden schon \mathbf{x} ist.
- ☞ Da \Downarrow -Werte für Quell-Knoten ausschließlich aus 0, 1 und \perp bestehen, erzeugt die Vorwärts-Analyse niemals \mathbf{x} .
- Rückwärts-Analyse erzeugt für einzelne Bit-Positionen \mathbf{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 \uparrow -Werte in v eingehender Kanten irrelevant sind, um exakt die \uparrow -Werte der aus v ausgehenden Kanten zu erzeugen.
 - Führe *Rückwärts-Simulation* des Operators von v auf \uparrow -Werten der ein- und ausgehenden Kanten aus
 - Speichere neue \uparrow -Werte der eingehenden Kanten von v

Ablauf der Rückwärts-Analyse

- $queue\langle DWFG_node \rangle q = \langle \text{Menge aller Senken-Knoten in } D \rangle$;
 $u(e) = d(e)$ für alle Kanten $e \in E$;
- while (! $q.empty()$)
 - $DWFG_node v = \langle \text{erstes Element aus } q \rangle$; $q.remove(v)$;
 $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
 - for ($\langle \text{alle Kanten } e = (v_x, v) \in E_{in} \rangle$)
 - $u'(e) = \langle \text{Rückwärts-Simulation von } v \text{ über } E_{out} \text{ und } E_{in} \setminus \{e\} \rangle$;
 - if ($\langle \text{bisheriges } u(e) \text{ ist bitweise } < \text{ als } u'(e) \rangle$)
 - $u(e) = u'(e)$;
 - 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 \Uparrow -Wert, in Abhängigkeit von den \Uparrow -Werten aller ausgehenden Kanten $e_{out,1}, \dots, e_{out,N} \in E_{out}$ und aller eingehenden Kanten außer e :

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

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 bitweise logische Operationen:

- Seien $b_{1,k}$ und $b_{2,k} \in \mathcal{L}_4$ einzelne Bits an Position k des \uparrow -Wertes zweier Operanden einer logischen Operation und $b_{3,k}$ das k -te Bit des \uparrow -Wertes des Resultats der Operation.
- Für $b_{2,k} = b_{3,k} = 0$:
 $b_{1,k} \text{ AND } b_{2,k} = b_{3,k} \quad \Leftrightarrow \quad b_{1,k} \text{ AND } 0 = 0 \quad \Leftrightarrow \quad \mathbf{x} \text{ AND } 0 = 0$
- Analog für OR:
 $b_{1,k} \text{ OR } 1 = 1 \quad \Leftrightarrow \quad \mathbf{x} \text{ OR } 1 = 1$
- 👉 Vorgehensweise für weitere logische Operationen ähnlich.

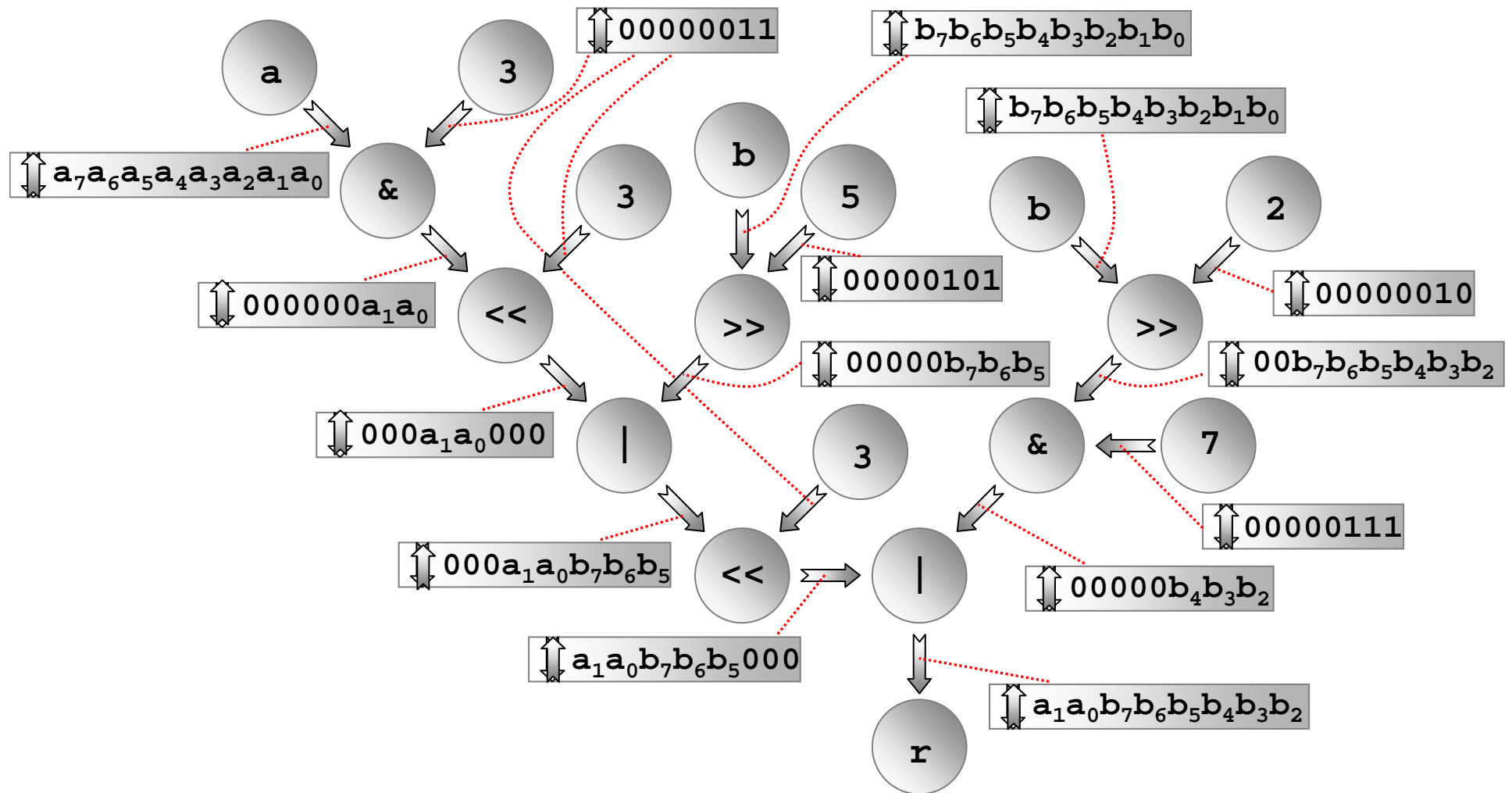
Rückwärts-Simulation (3)

Arithmetische Operationen:

- Identifizierung irrelevanter Bits wegen Komplexität arithmetischer Operationen oft sehr schwer.
- ☞ Leicht verständliches Beispiel – Schiebe-Operator:
 - $a \ll 3$: Verschiebt Inhalt von a um 3 Bits nach links
Niederwertige 3 Bits werden mit 0 gefüllt,
Höchstwertige 3 Bits werden abgeschnitten.
☞ Im \uparrow -Wert für a sind die 3 höchstwertigen Bits x
 - $a \gg 3$: Analog für niederwertige 3 Bits

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

Beispiel Rückwärts-Simulation



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

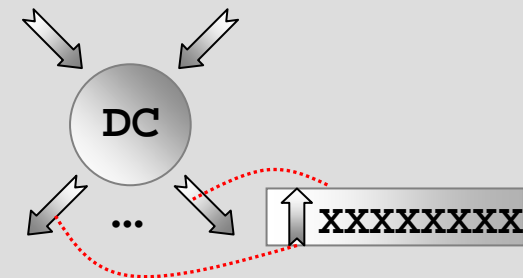
Definition (*Dead Code*):

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

Dead Code und die BDWFA:

- In \uparrow -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 \uparrow -Wert x^* hat, ist Dead Code.



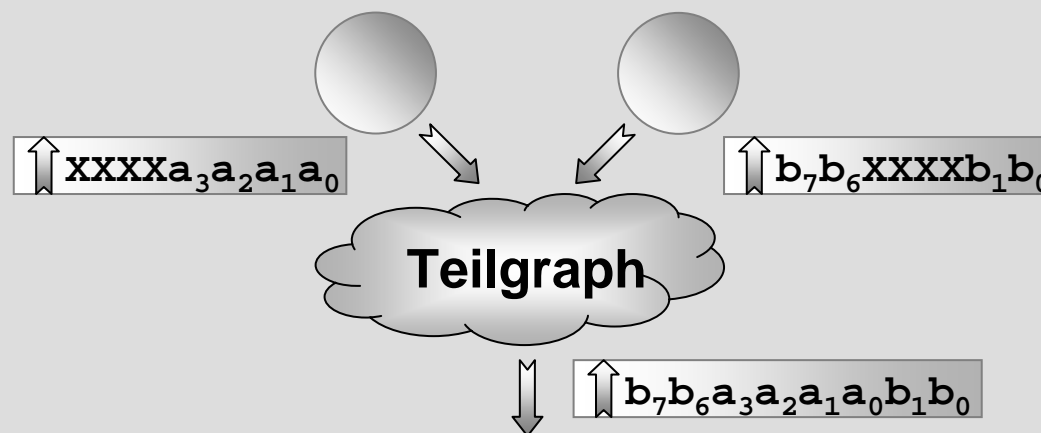
Ablauf der bitgenauen DCE

- suche Kante $e = (v, w) \in E$ mit $u(e) = \mathbf{x}^*$;
 queue<DWFG_node> $q = v$;
- while (! $q.empty()$)
 - DWFG_node $v = \langle \text{erstes Element aus } q \rangle$; $q.remove(v)$;
 $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
 - if (($u(e) = \mathbf{x}^*$ für alle $e \in E_{out}$) &&
 ($\langle v \text{ hat keine Seiteneffekte} \rangle$))
 - 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 \uparrow -Werten ablesbar:

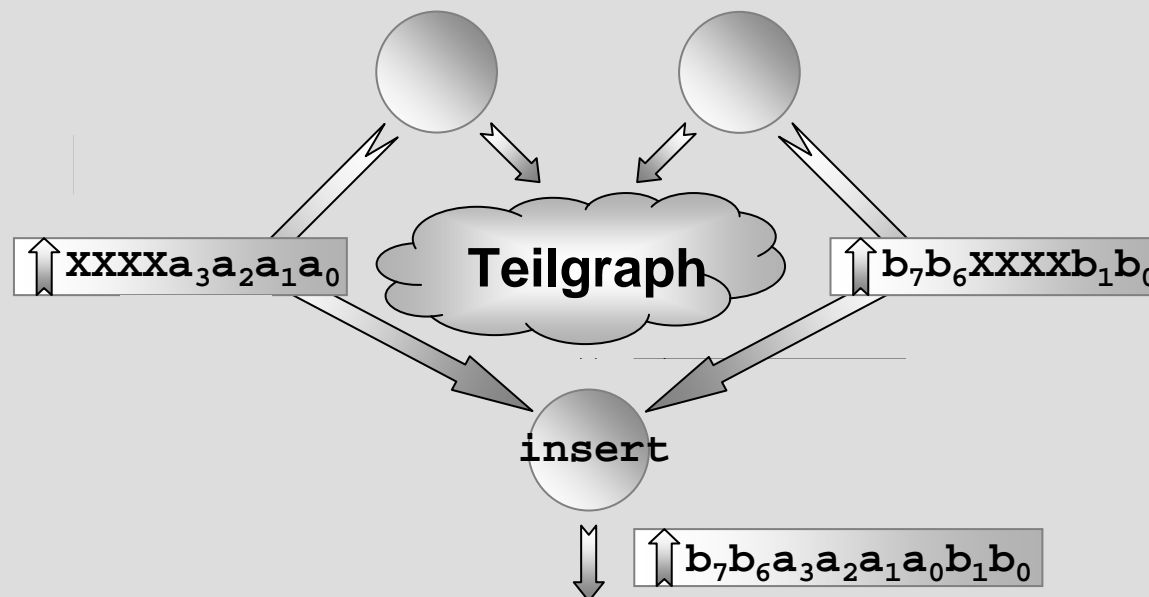


- ☞ *Eine Optimierung muß den \uparrow -Wert einer Kante mittels \perp -Werten in disjunkte Bit-Pakete zerlegen und aus dieser Zerlegung passende insert-Operationen erzeugen.*

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:

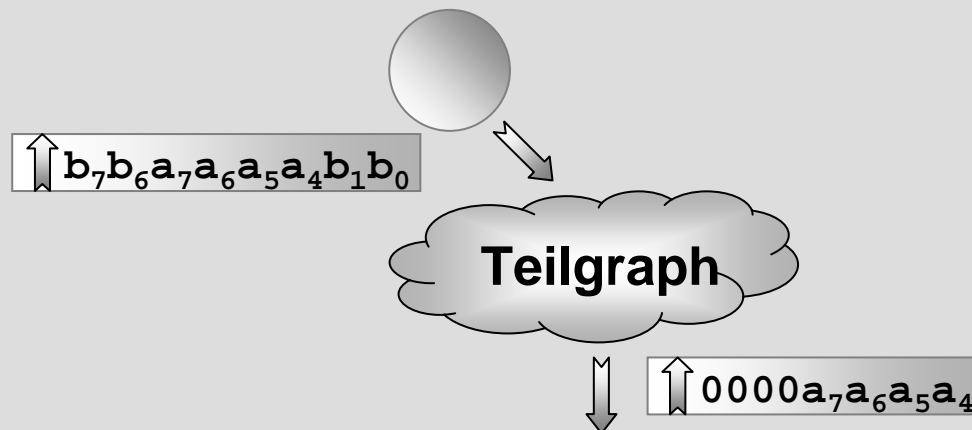


☞ *Sofern nach dem Kanten-Anpassen keine weiteren Kanten den Teilgraphen verlassen, kann dieser durch eine DCE entfernt werden!*

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

Extract-Operationen und die BDWFA:

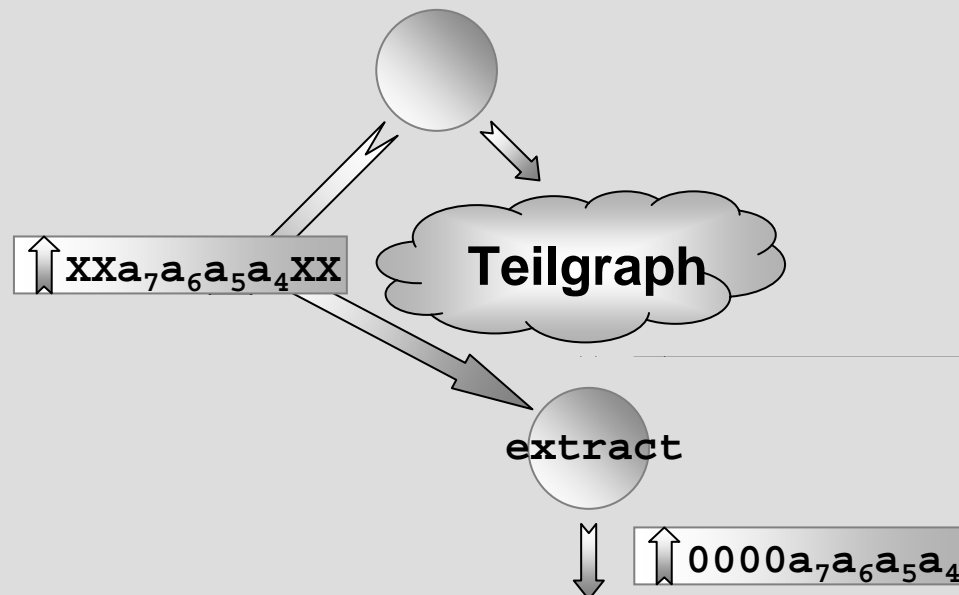
- Extraktion eines Bit-Pakets durch beliebigen Teil-Graph des DWFG auch anhand von \uparrow -Werten ablesbar:



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:



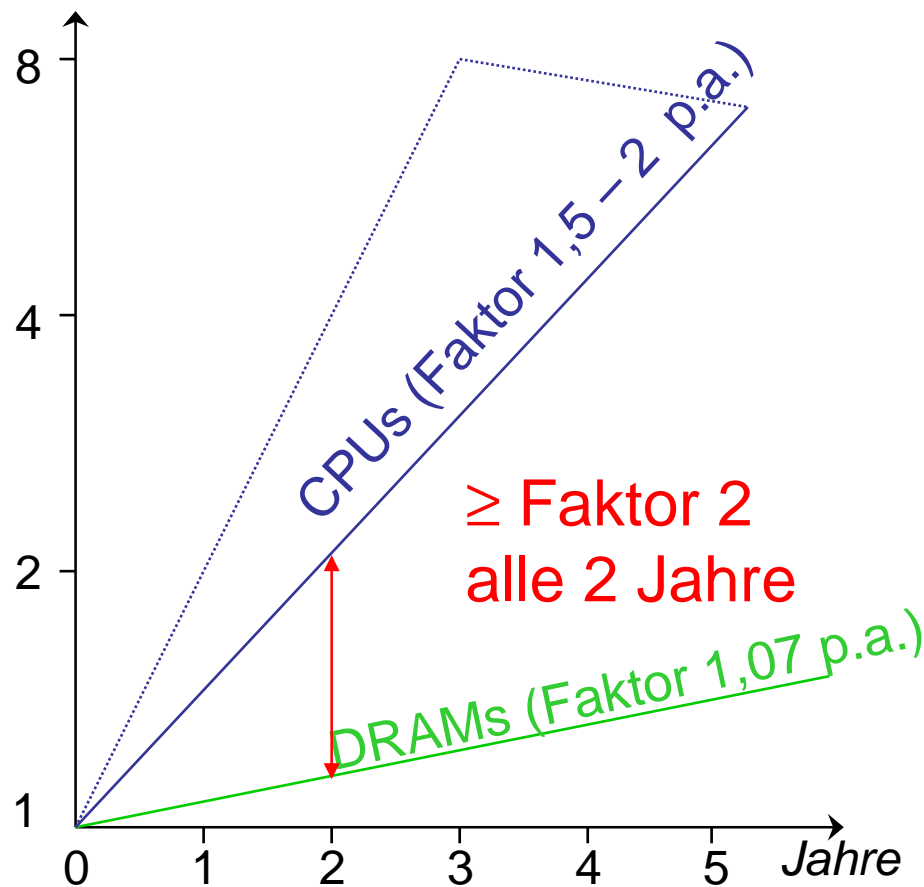
☞ *U.U. kann der Teilgraph auch wieder durch eine DCE entfernt werden.*

Gliederung der Vorlesung

- Kapitel 1: Compiler für Eingebettete Systeme
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- Kapitel 5: Instruktionsauswahl
- **Kapitel 6: LIR Optimierungen und Transformationen**
 - Generierung von Bit-Paket Operationen für NPUs
 - Optimierungen für Scratchpad-Speicher
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur WCET_{EST}-Minimierung
- Kapitel 9: Ausblick

Eigenschaften heutiger Speicher (1)

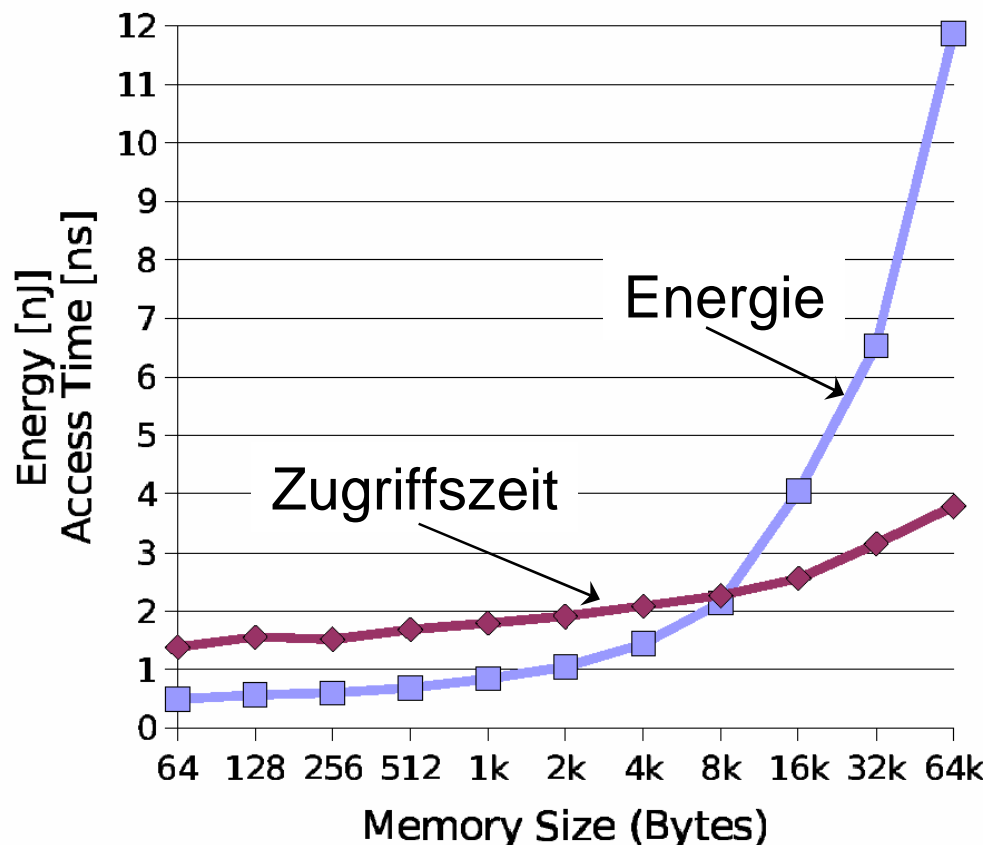
Geschwindigkeit



- 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]

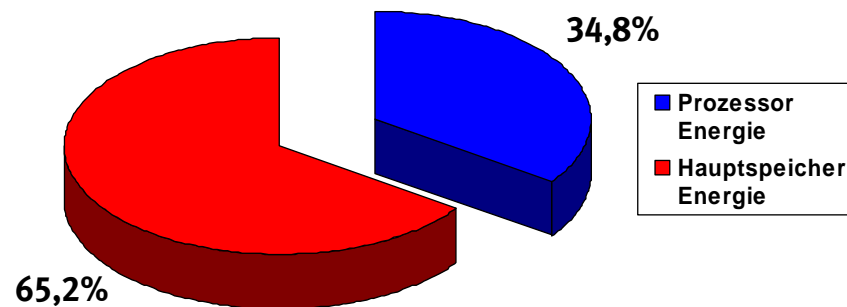
Eigenschaften heutiger Speicher (2)



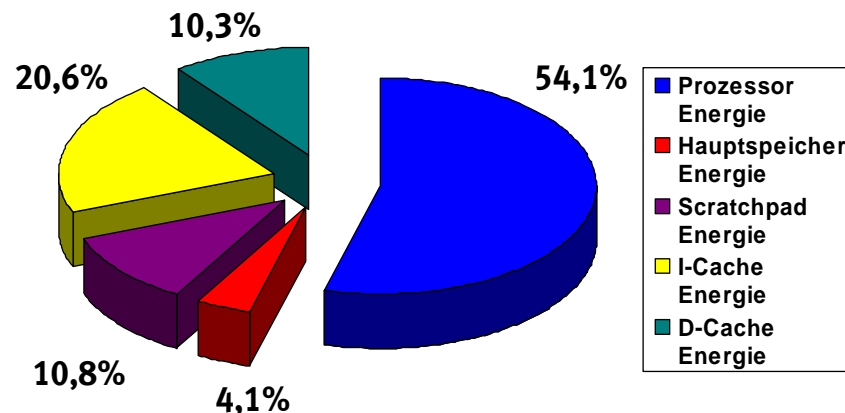
- Mit zunehmender Größe eines Speichers verbraucht ein Speicherzugriff überproportional mehr Energie.
- Mit zunehmender Größe dauern Speicherzugriffe auch proportional länger.
- 👉 *Fertigungstechnologie von Speichern legt Nutzung kleiner Speicher nahe!*

Eigenschaften heutiger Speicher (3)

- ARM7 Mono-Prozessor ohne Cache:



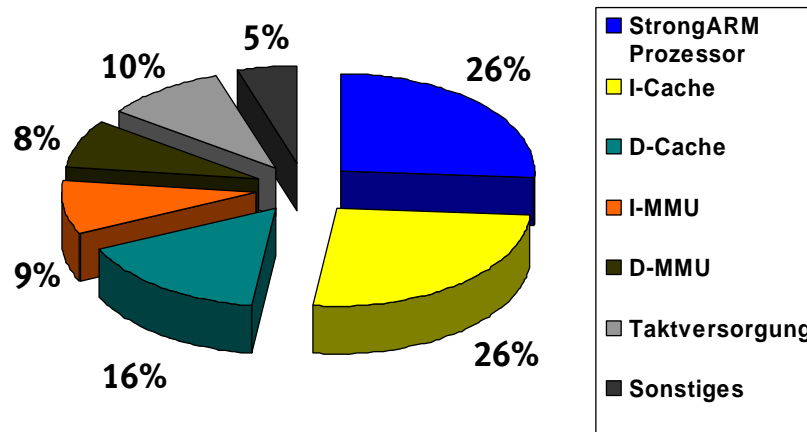
- ARM7 Multi-Prozessor mit Caches:



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

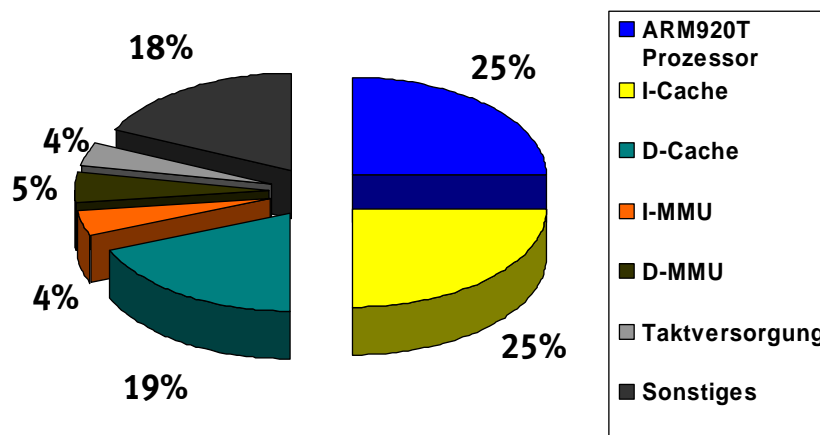
[M. Verma, P. Marwedel, *Advanced Memory Optimization Techniques for Low-Port Embedded Processors*, Springer, 2007]

Eigenschaften heutiger Speicher (4)



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

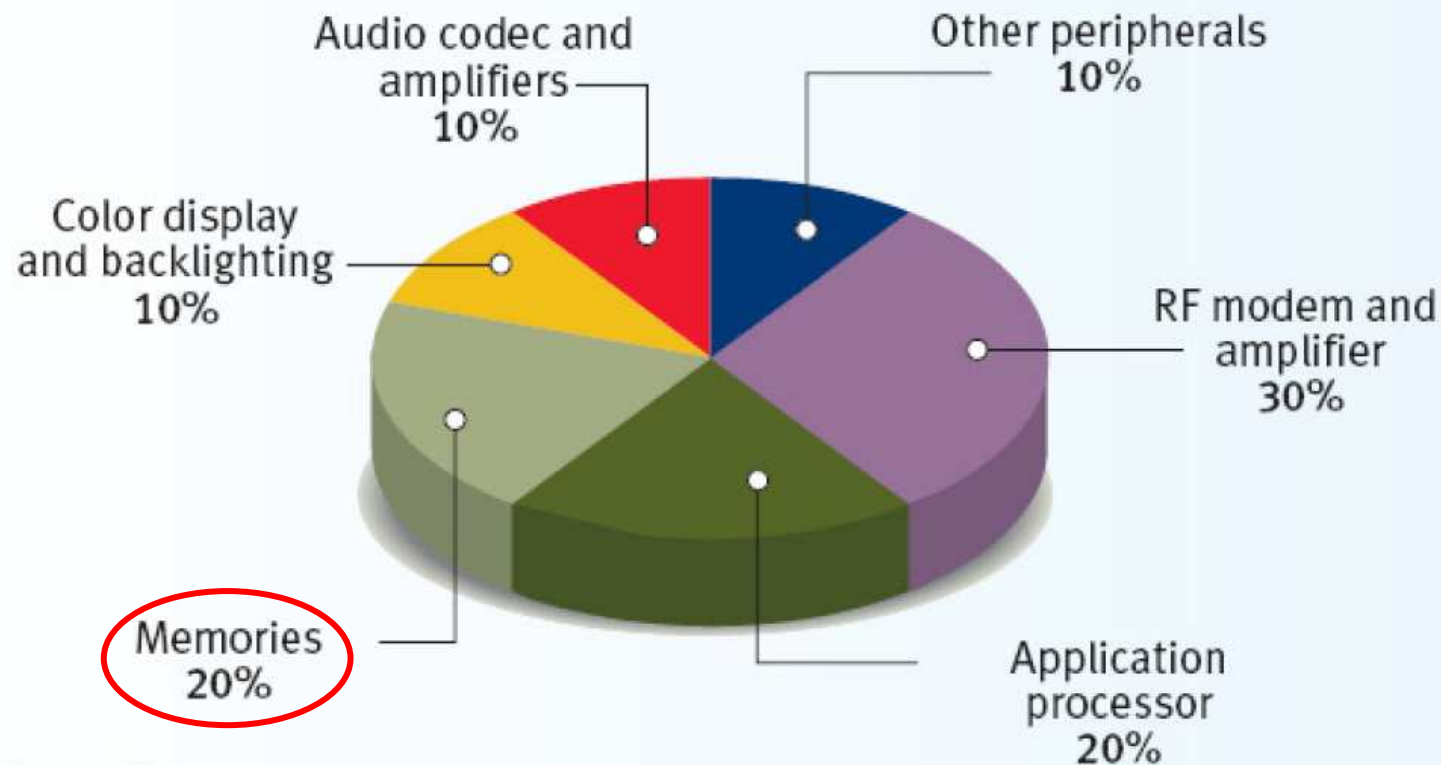
Größenordnungen des Energieverbrauchs von Speichern durch Arbeiten anderer Gruppen aus Industrie & Forschung bestätigt.



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

Eigenschaften heutiger Speicher (5)

- Energieverbrauch mobiler Geräte:

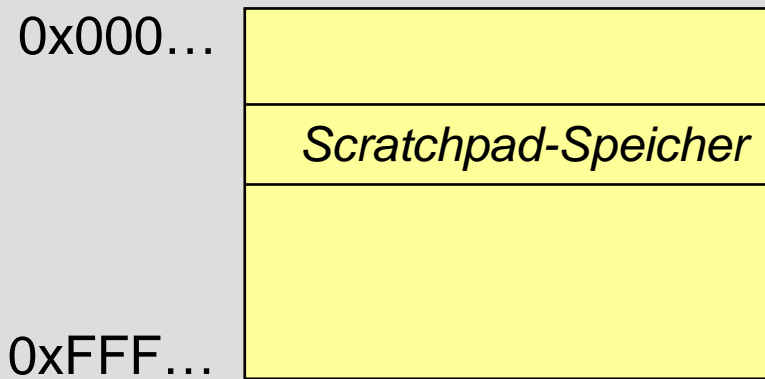


Source: Siemens

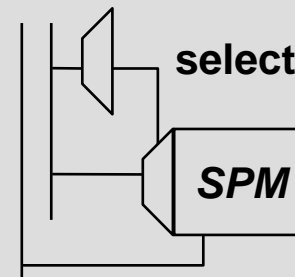
[O. Vargas (Infineon), Minimum power consumption in mobile-phone memory subsystems, Pennwell Portable Design, Sep. 2005]

Scratchpad-Speicher

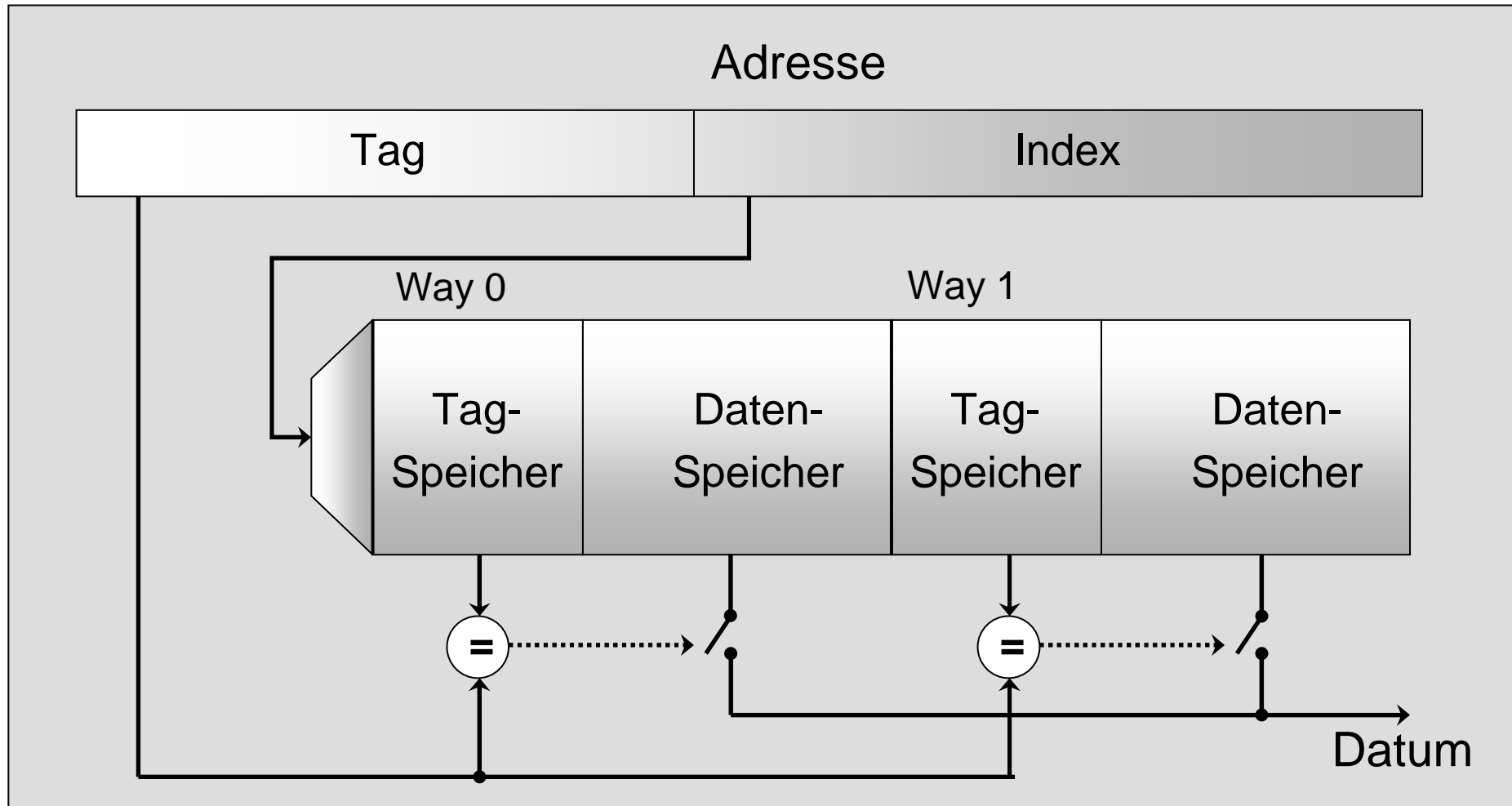
- Scratchpads (*SPMs*) sind kleine, physikalisch separate Speicher.
- Sie sind meist auf dem selben Chip plaziert wie der Prozessor (sog. *on-chip Speicher*).
- ☞ *Durch geringe Größe und on-chip Platzierung: extrem schnelle und energieeffiziente Speicher*
- Sind in den Adressraum des Prozessors nahtlos eingebettet:



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



Aufbau mengenassoziativer Caches



Eigenschaften von Scratchpad-Speichern (1)

Vorhersagbarkeit:

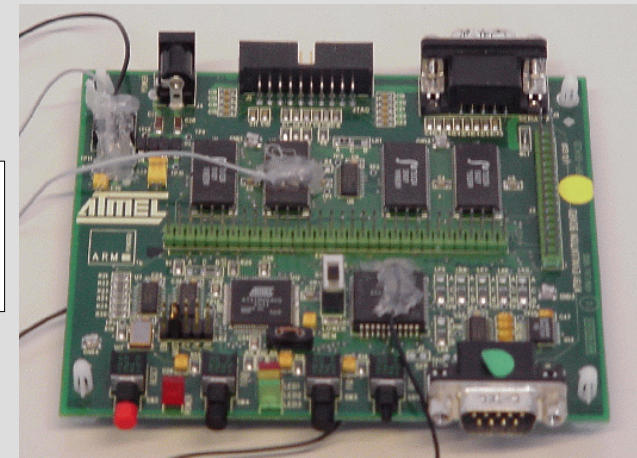
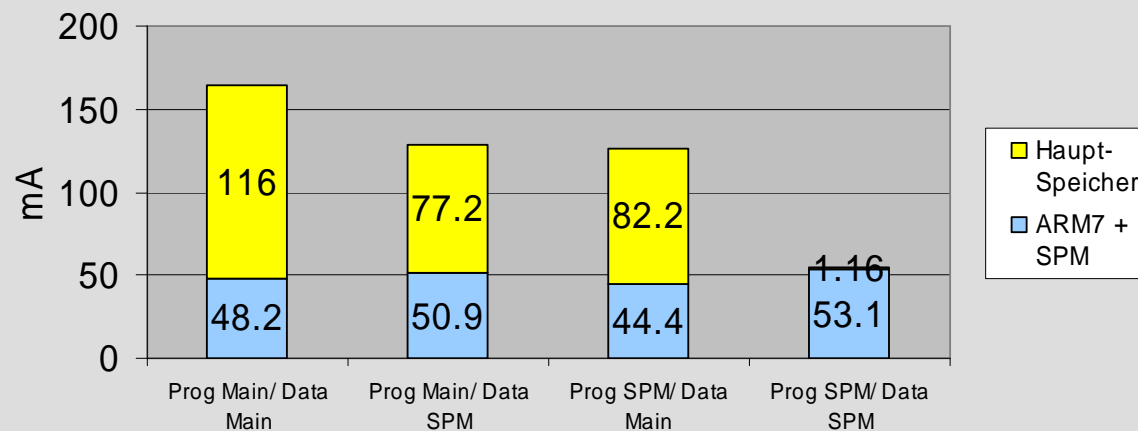
- Jeder Zugriff auf SPM braucht lediglich konstante Zeit, üblicherweise 1 Taktzyklus.
- Im Gegensatz dazu: ein Cache-Zugriff kann variable Zeit brauchen, je nachdem, ob er zu einem Cache-Hit oder Cache-Miss führt.
- ☞ Laufzeitverhalten von Scratchpad-Speichern ist zu 100% vorhersehbar, während Verhalten 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

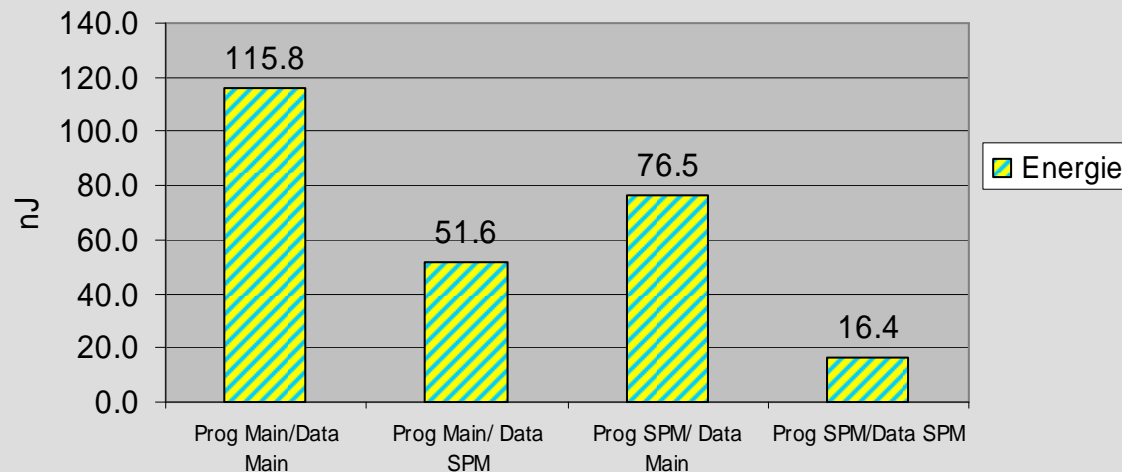


Eigenschaften von Scratchpad-Speichern (3)

Energieverbrauch im Vergleich zu Hauptspeicher:

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

Energieverbrauch Ladebefehl



Erinnerung:

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

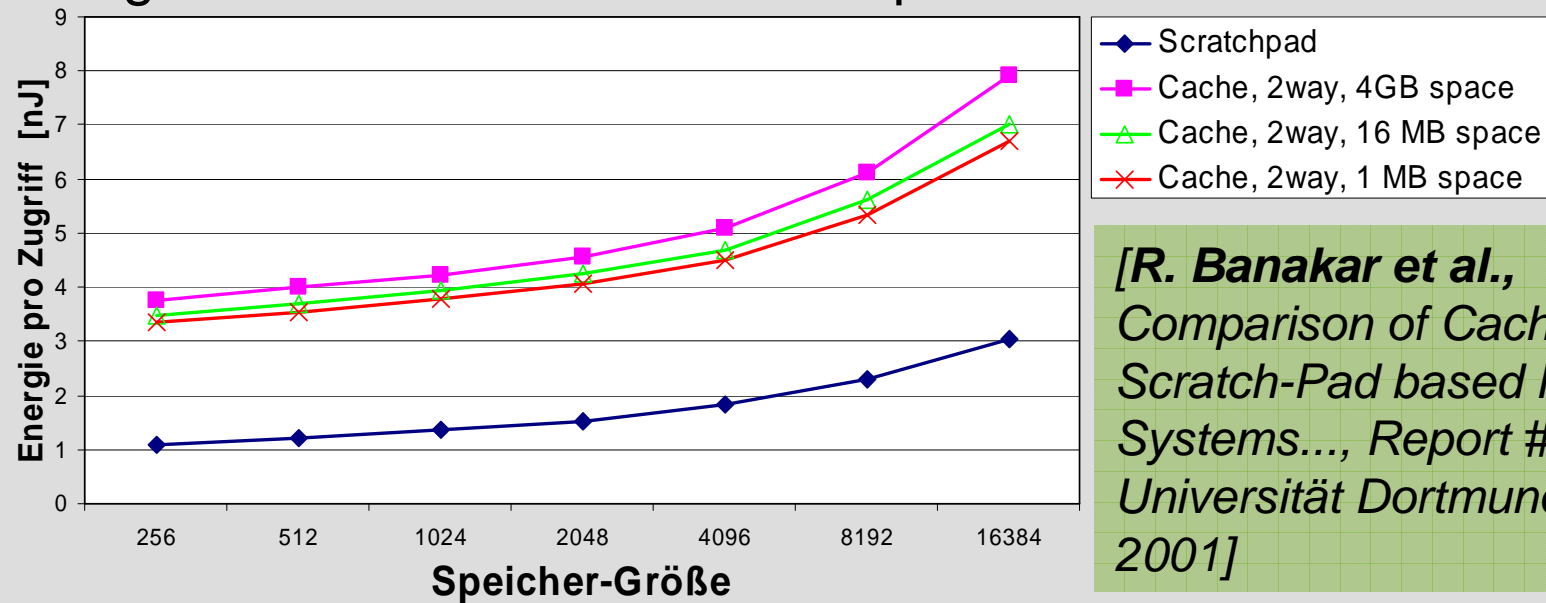
Annahme: Versorgungsspannung konstant und Stromverbrauch nicht zu variabel über Zeit

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

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:



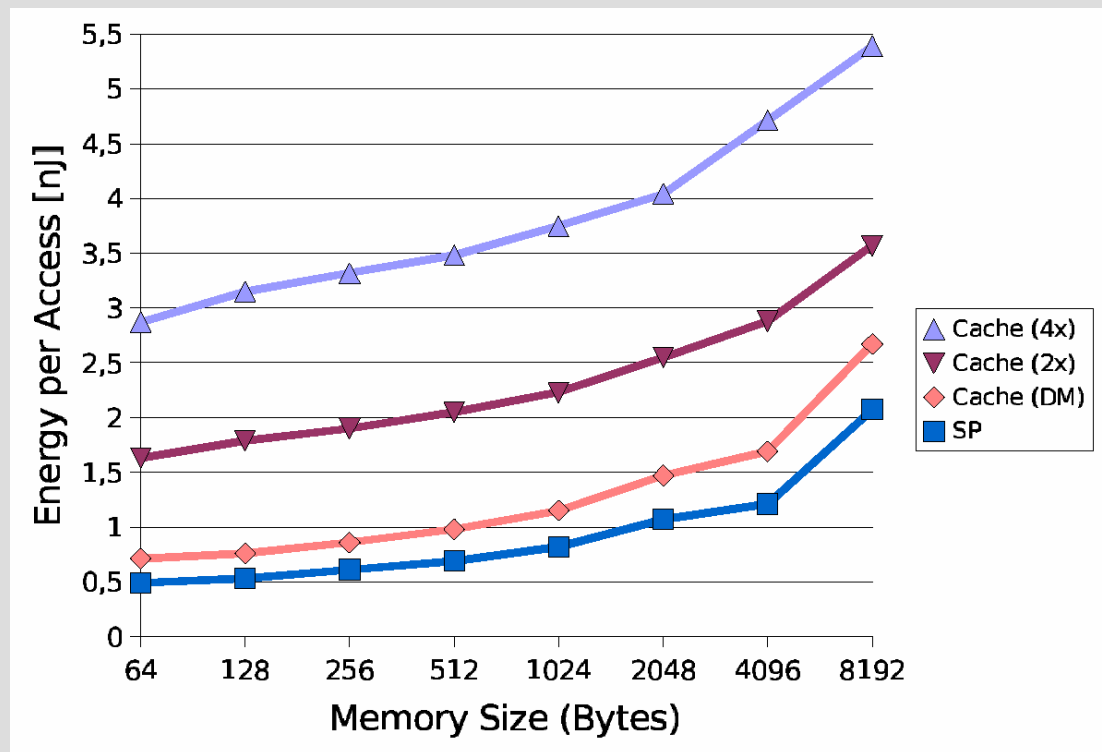
[R. Banakar et al., Comparison of Cache- and Scratch-Pad based Memory Systems..., Report #762, Universität Dortmund, Sep. 2001]

Eigenschaften von Scratchpad-Speichern (5)

Energieverbrauch im Vergleich zu Caches:

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

Vorsicht: Technologie bei diesem Diagramm unterschiedlich zu der von Folie 53. Daher Abweichungen in absoluten Zahlenwerten!



Ganzzahlig lineare Programmierung

Modellierungstechnik für lineare Optimierungsprobleme:

- Optimierung einer *Zielfunktion* z unter Beachtung von *Nebenbedingungen* n_1, \dots, n_m
- Zielfunktion und Nebenbedingungen sind lineare Ausdrücke über den *ganzzahligen Entscheidungsvariablen* x_1, \dots, 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:

- $MO = \{mo_1, \dots, mo_n\}$
 $= F \cup V$ 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 einzelner 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 Programmes ($= n_i * \Delta e_i$)
- 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 Programmes liefert Anzahl $n_{i,\text{instr}}$ ausgeführter Instruktionen für mo_i .
$$\Delta e_i = n_{i,\text{instr}} * \Delta e_{\text{IFetch}}$$

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 Programmes findet ein Simulationsdurchlauf statt, um zur Optimierung notwendige Parameter zu ermitteln.
- 👉 Ein solcher Simulationsdurchlauf erzeugt ein Laufzeit-Profil des zu optimierenden Programmes. Daher:
Eine solche Simulation vor einer Optimierung heißt *Profiling*.

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

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

ILP-Formulierung:

- Zielfunktion: Maximiere Energieeinsparung für gesamtes Programm

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

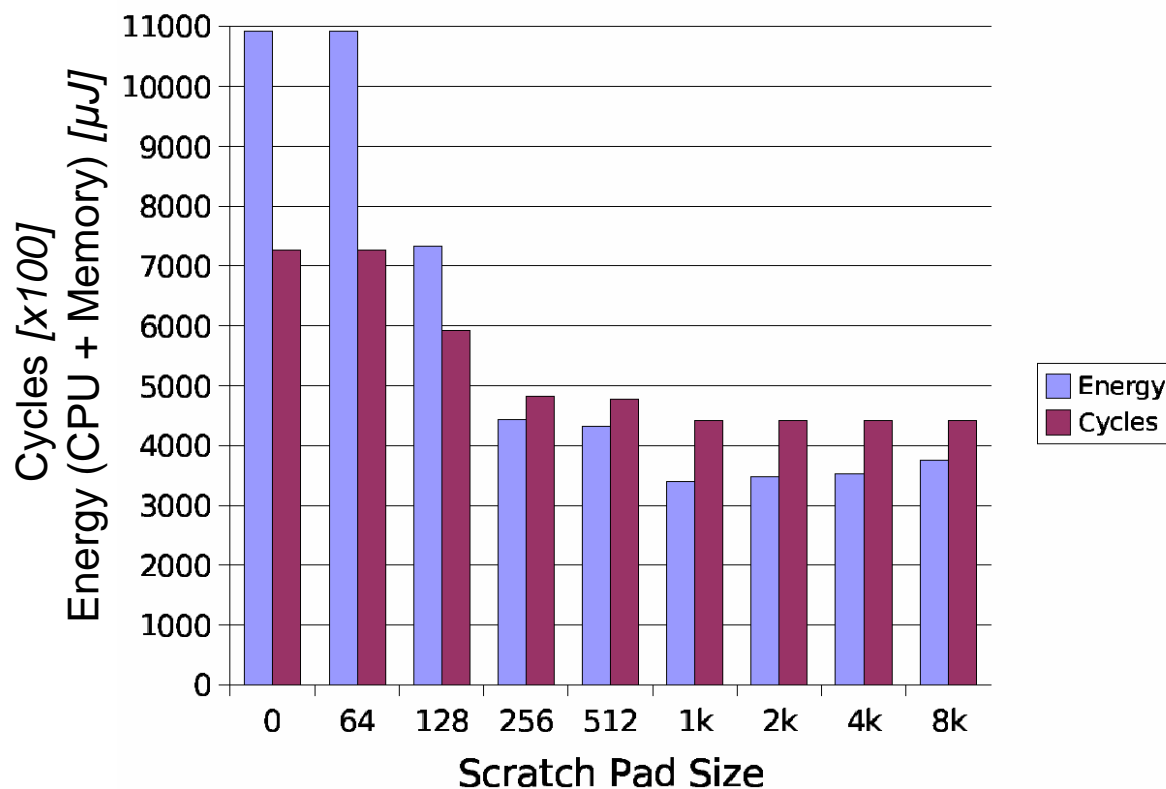
- Nebenbedingung: Einhaltung der Kapazität des SPMs

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

[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:

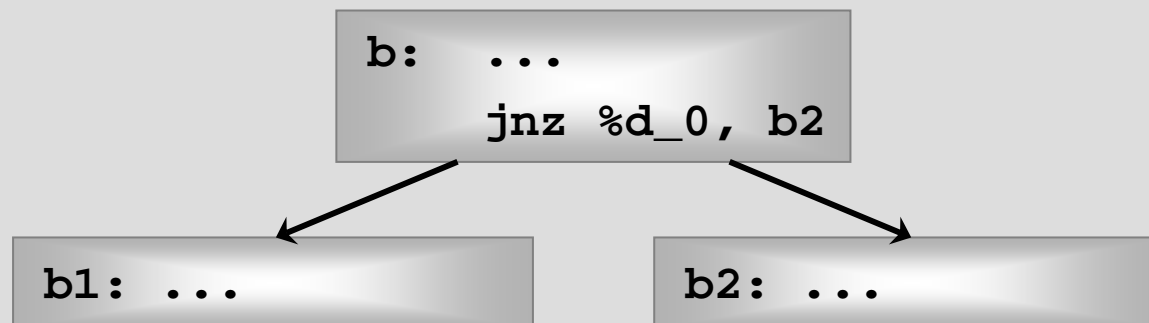
Verschiebung des Codes von kompletten LIR-Funktionen, von *einzelnen Basisblöcken* und von globalen Variablen in den SPM.

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

Problem beim Verschieben einzelner Basisblöcke:

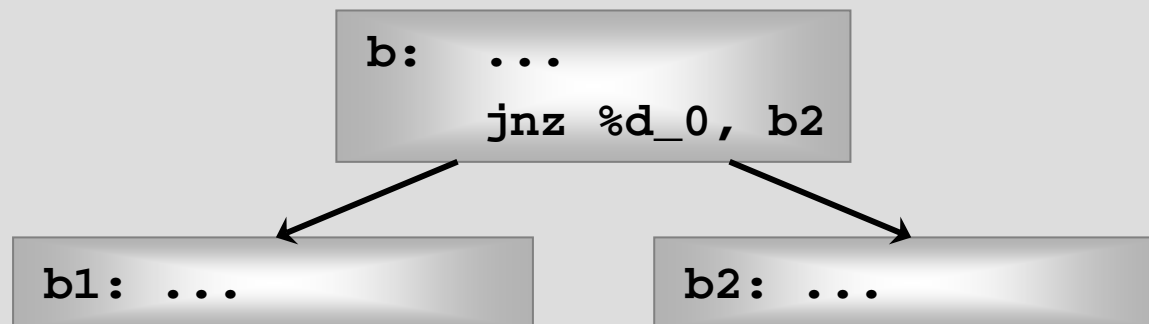
Erinnerung: Ein Basisblock b darf nur als letzten Befehl einen Sprungbefehl enthalten

- Ist der Sprung am Ende von b bedingt, so hat b im CFG zwei Nachfolger $b1$ und $b2$, die ausgeführt werden, wenn der bedingte Sprung entweder genommen wird oder nicht:



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

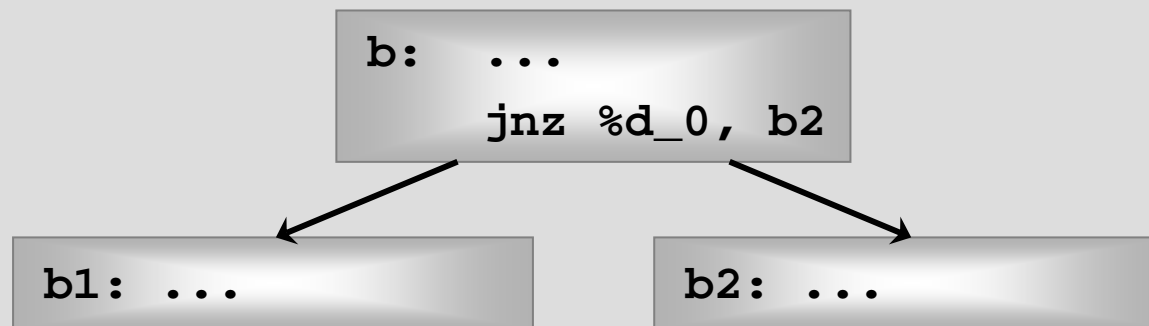
Problem beim Verschieben einzelner Basisblöcke:



- *b1* 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 *b1* im Speicher direkt auf *b* folgt.

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

Problem beim Verschieben einzelner Basisblöcke:



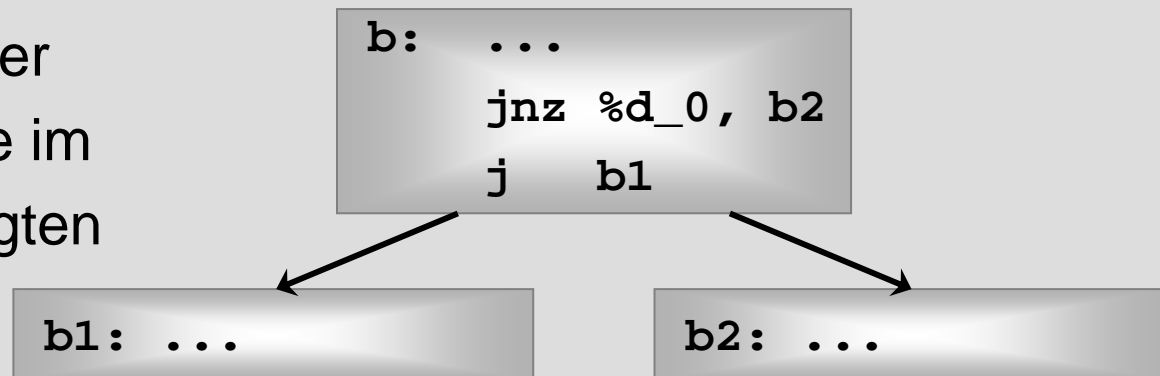
Was, wenn b im SPM liegt, $b1$ aber nicht (oder umgekehrt)?

- 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 inkorrekt 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 $b1$:



Nachteil:

- Unbedingter Sprung extrem ineffizient (Codegröße, Laufzeit und Energie), wenn b und $b1$ 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.

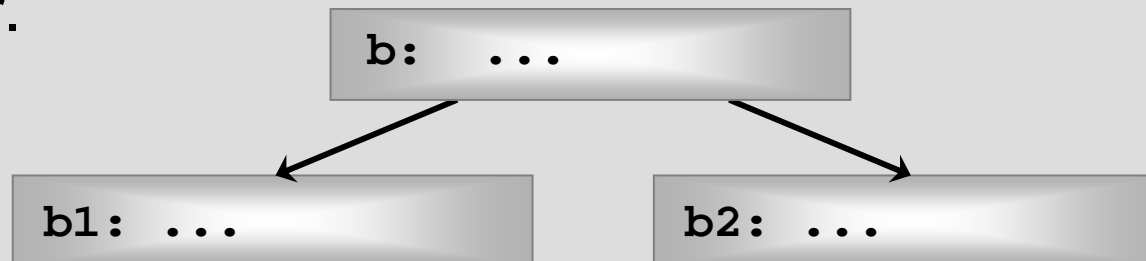
👉 *Wie modelliert man nicht-konstanten Parameter S_b im ILP?*

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 Multi-basisblock dar.



$\{b, b1\}$, $\{b, b2\}$ und $\{b, b1, b2\}$ sind Multi-Basisblöcke.

$\{b1, b2\}$ 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:

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

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

- Nebenbedingung 1: Einhaltung der Kapazität des SPMs

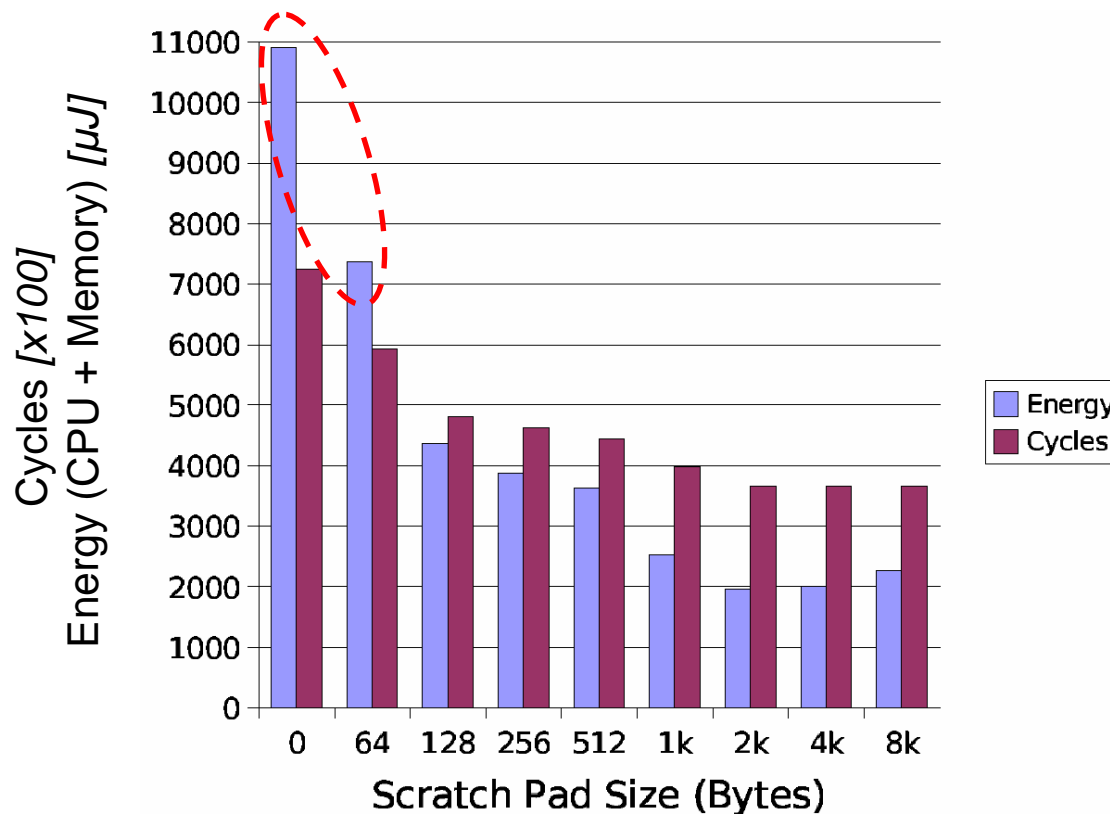
$$n_1: \sum_{i=1}^n S_i * x_i \leq S$$

- Nebenbedingung pro $mo_j \in B$: mo_j 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

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

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

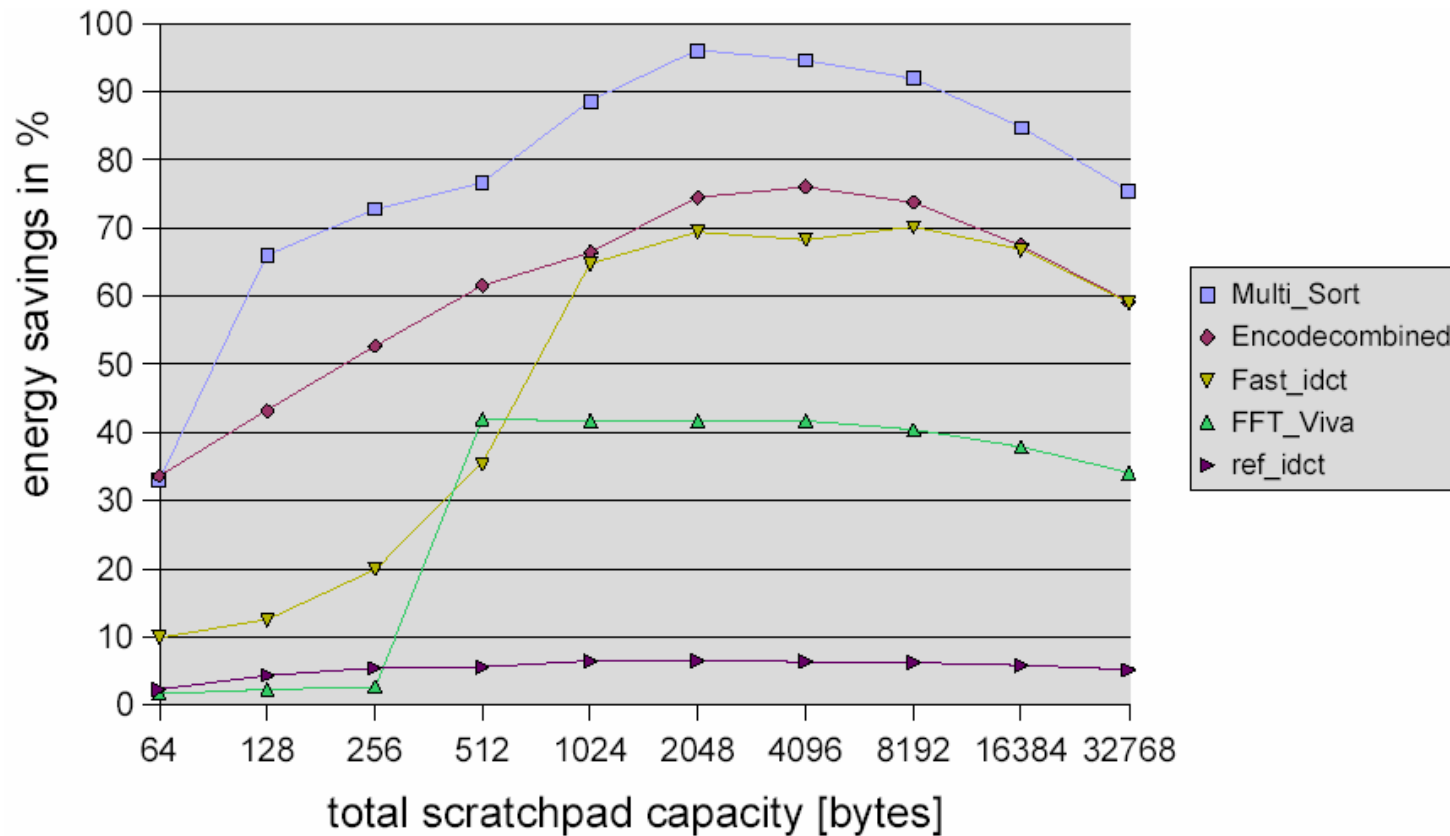
Ergebnisse (MultiSort-Benchmark):



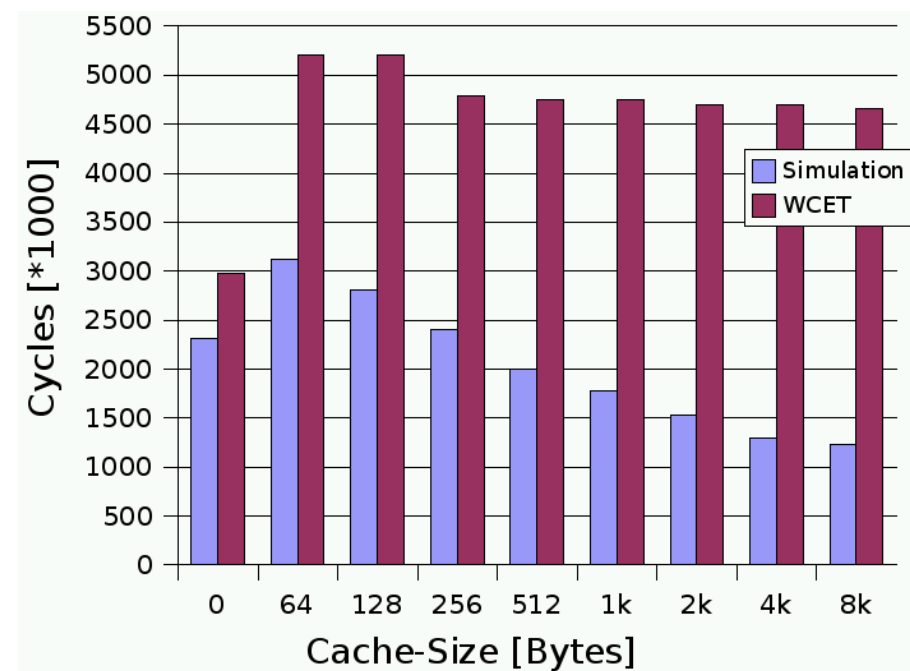
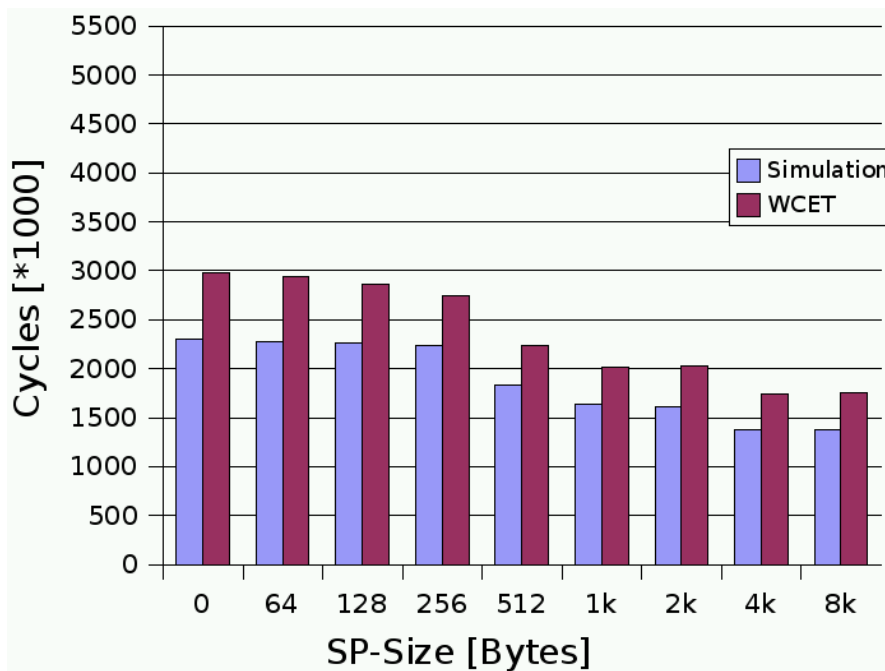
- 64b SPM jetzt nicht mehr zu klein, um für Code-Teile ausgenutzt zu werden.
- Vorsicht: Für links stehendes Diagramm ist auch der Laufzeit-Stack als Speicher-Objekt betrachtet und somit auf den SPM verschoben worden. Daher ist dieses Diagramm nur bedingt mit Folie 62 vergleichbar!

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

Detail-Ergebnisse nur für Speicher-Subsystem:

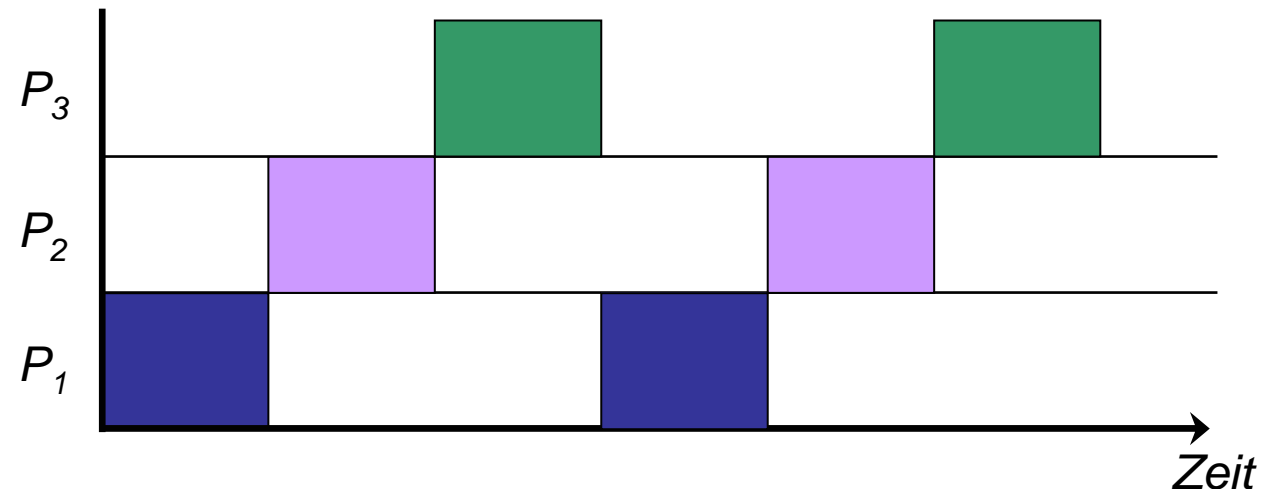
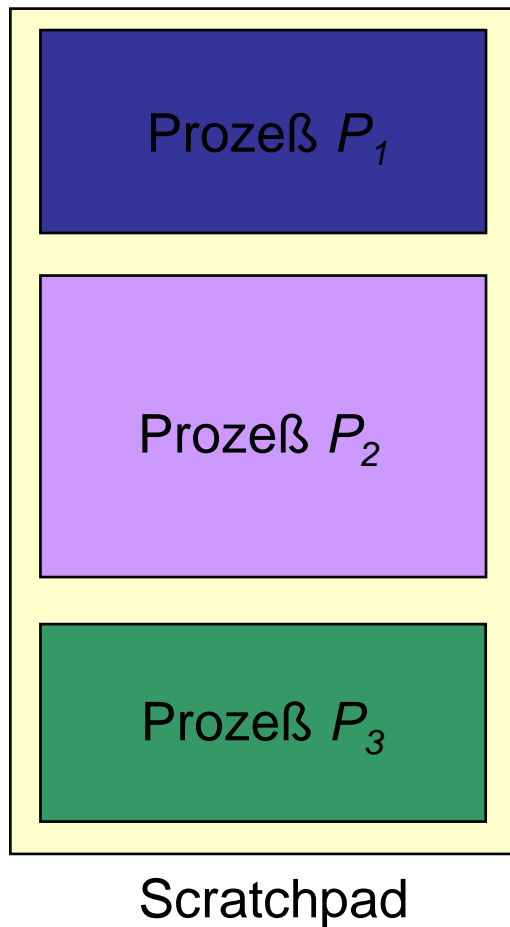


Fixe SPM-Allokation: Funktionen, Basisblöcke & Globale Variablen (13) Vergleich ACET / WCET_{EST} für Scratchpads und Caches:



- 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

Multi-Prozeß SPM-Allokation: Partitionierter SPM-Speicher (1)



- Gesamte SPM-Kapazität wird zur Übersetzungszeit in disjunkte Partitionen zerlegt
- Jeder Prozeß P_i hat eine SPM-Partition für Funktionen, (Multi-) BBs & globale Variablen von P_i
- *Erwartung:* Gute Ergebnisse für große SPMs

Multi-Prozeß SPM-Allokation: Partitionierter SPM-Speicher (2)

Energie-Arrays einzelner Prozesse:

- Seien P_1, \dots, P_N Prozesse einer Multi-Prozeß Anwendung
- S sei die Größe des verfügbaren SPMs in Bytes, $S' < S$ ein von außen vorgegebener Parameter, der den SPM in einzelne "Scheiben" von S' Bytes Größe zerteilt.
- Für jeden Prozeß P_i ist dessen Energie-Array $f'_i[x]$ zu bestimmen. $f'_i[x]$ gibt an, wieviel Energie P_i verbraucht, wenn P_i x Bytes SPM zur Verfügung hat.
- $f'_i[x]$ wird für alle Größen $x = S', 2S', 3S', \dots$ bestimmt, die ein Vielfaches von S' sind, durch Lösen des ILP zur fixen SPM-Allokation für Funktionen, Multi-BBs und globale Variablen für jedes dieser x .

Multi-Prozeß SPM-Allokation: Partitionierter SPM-Speicher (3)

Energie-Arrays für Multi-Prozesse und partitionierten SPM:

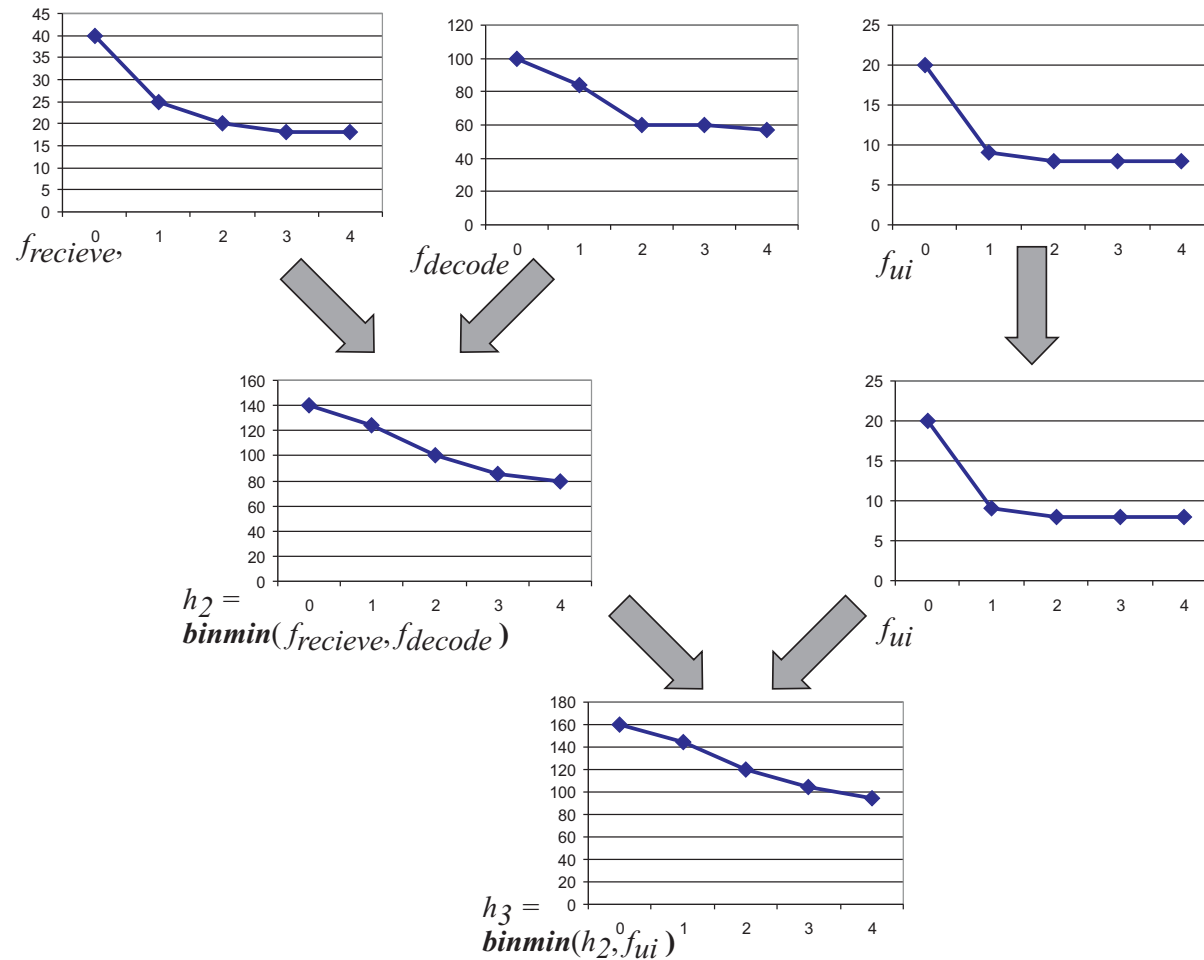
- Für eine Multi-Prozeß Anwendung bestehend aus Prozessen P_1, \dots, P_N ist deren Energie-Array $e'_N[x]$ zu bestimmen. $e'_N[x]$ gibt an, wieviel Energie die Anwendung verbraucht, wenn sie x Bytes SPM zur Verfügung hat.
- $e'_N[x] = \min\{ f'_1[x_1] + \dots + f'_N[x_N] \mid x_1 + \dots + x_N \leq x \}$ für alle Werte von x, x_1, \dots, x_N , für die die einzelnen f'_i definiert sind.
- Für eine gegebene SPM-Größe S und eine Multi-Prozeß Anwendung ist man an einer SPM-Allokation mit minimalem Wert $e'_N[S]$ interessiert.

Multi-Prozeß SPM-Allokation: Partitionierter SPM-Speicher (4)

Die binmin-Funktion:

- Annahme: Eine Multi-Prozeß Anwendung bestehe aus 3 Prozessen P_1 , P_2 und P_3 .
- $$e'_3[x] = \min\{ f'_1[x_1] + f'_2[x_2] + f'_3[x_3] \mid x_1 + x_2 + x_3 \leq x \}$$
$$= \text{binmin}(\text{binmin}(f'_1, f'_2), f'_3)$$
- binmin ist eine Funktion, die zwei Energie-Arrays g und h miteinander verknüpft und wiederum ein Energie-Array liefert:
$$\text{binmin}(g, h)[x] = \min\{ g[x_1] + h[x_2] \mid x_1 + x_2 \leq x \}$$
- Aufgrund der Distributivität kann die Minimum-Bildung über eine N -fache Summe in $e'_N[x]$ auf das binäre min in binmin reduziert werden.

Multi-Prozeß SPM-Allokation: Partitionierter SPM-Speicher (5)



Beispiel:
Applikation mit 3
Prozessen (*receive*,
decode, *ui*)

Multi-Prozeß SPM-Allokation: Partitionierter SPM-Speicher (6)

Algorithmus zur Berechnung von $\text{binmin}(g, h)$:

- for ($x = 0$; $x \leq S$; $x += S'$)
 - int $min = \infty$;
 - for ($tmp = 0$; $tmp \leq x$; $tmp += S'$)
 - if ($g[tmp] + h[x - tmp] < min$)
 $min = g[tmp] + h[x - tmp]$;
 - $b[x] = min$;
- return b ;

Multi-Prozeß SPM-Allokation: Partitionierter SPM-Speicher (7)

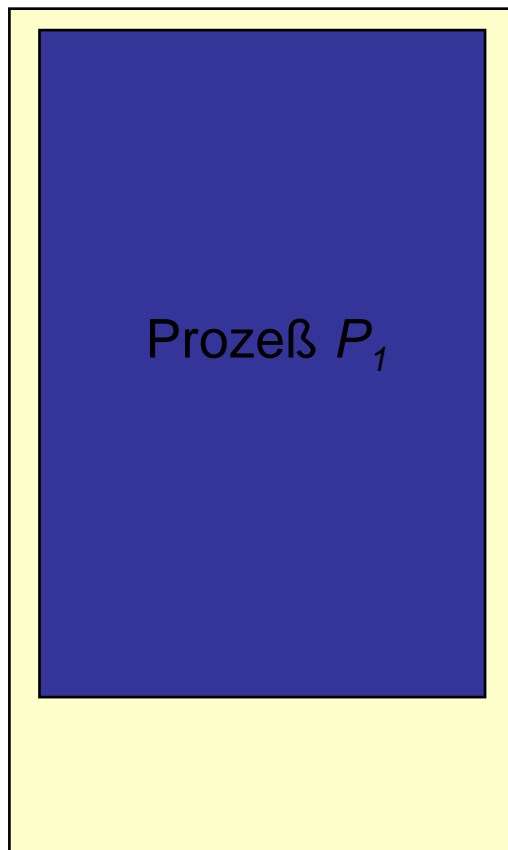
Algorithmus zur Berechnung von e'_N :

PartitionedSPM(f'_1, \dots, f'_N)

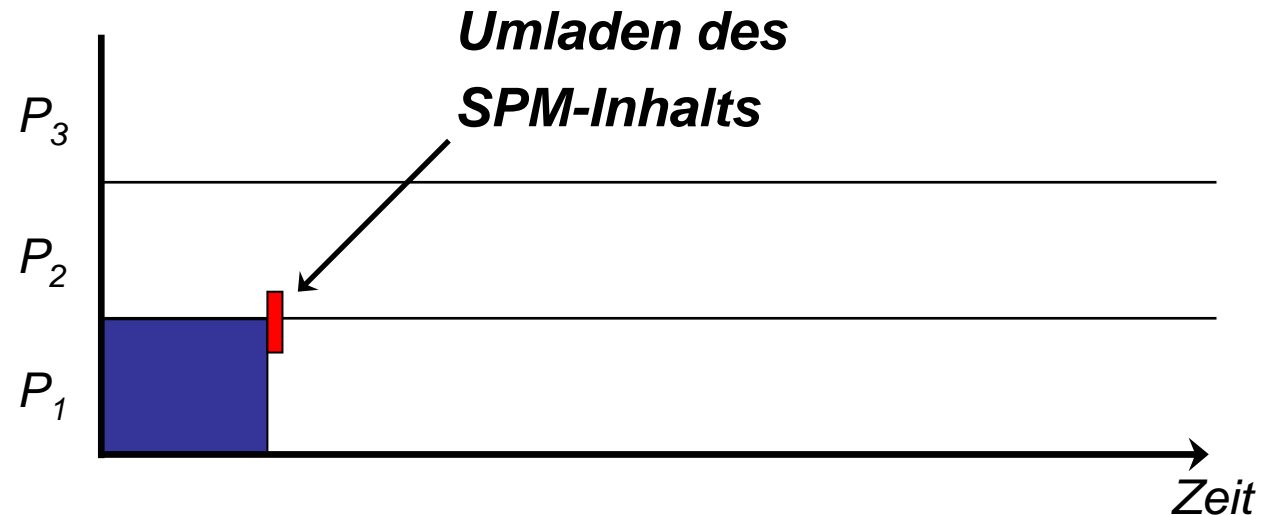
- if ($N > 1$)
 - $e'_{N-1} = \text{PartitionedSPM}(f'_1, \dots, f'_{N-1});$
 - $e'_N = \text{binmin}(e'_{N-1}, f'_N);$
- else
 - $e'_N = \text{binmin}(f'_1, Z);$
- return e'_N ;

- Z ist ein Energie-Array, das nur Nullen für alle SPM-Größen enthält.
- binmin kann leicht angepaßt werden, so daß es nicht nur Array $b[]$ zurückliefert, sondern auch den Wert tmp , für den $b[x]$ minimal ist.
- ☞ Damit liefert der Algorithmus nicht nur e'_N , sondern auch die Partitionsgröße x_i für alle P_i .

Multi-Prozeß SPM-Allokation: Exklusiver SPM-Speicher (1)

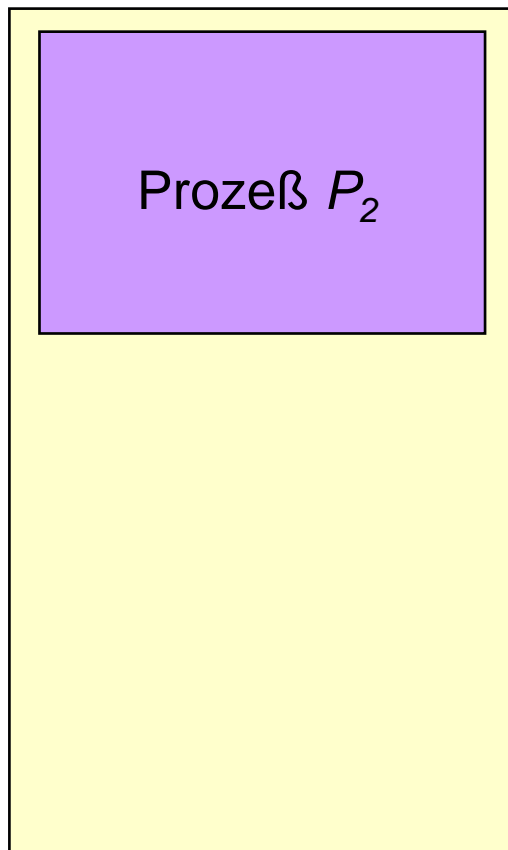


Scratchpad

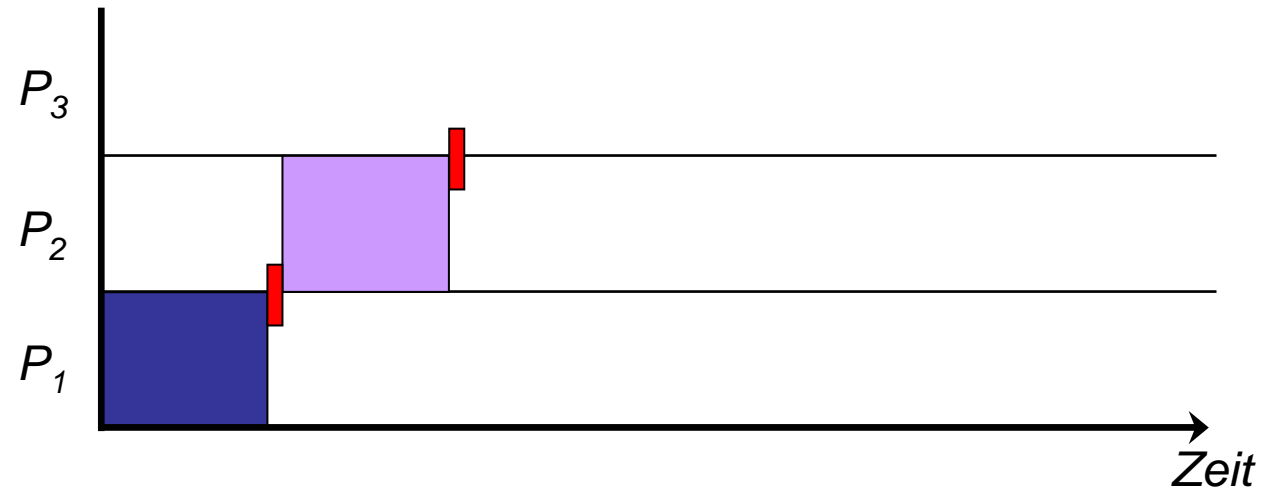


- Jeder Prozeß P_i verfügt über kompletten SPM für Funktionen, (Multi-) BBs & globale Variablen
- Bei Kontextwechsel muß SPM-Inhalt gesichert und neu geladen werden.

Multi-Prozeß SPM-Allokation: Exklusiver SPM-Speicher (1)

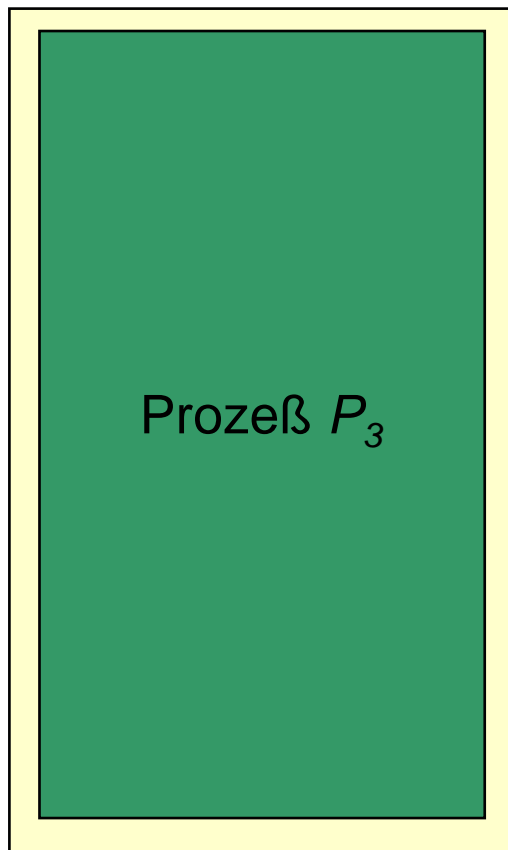


Scratchpad

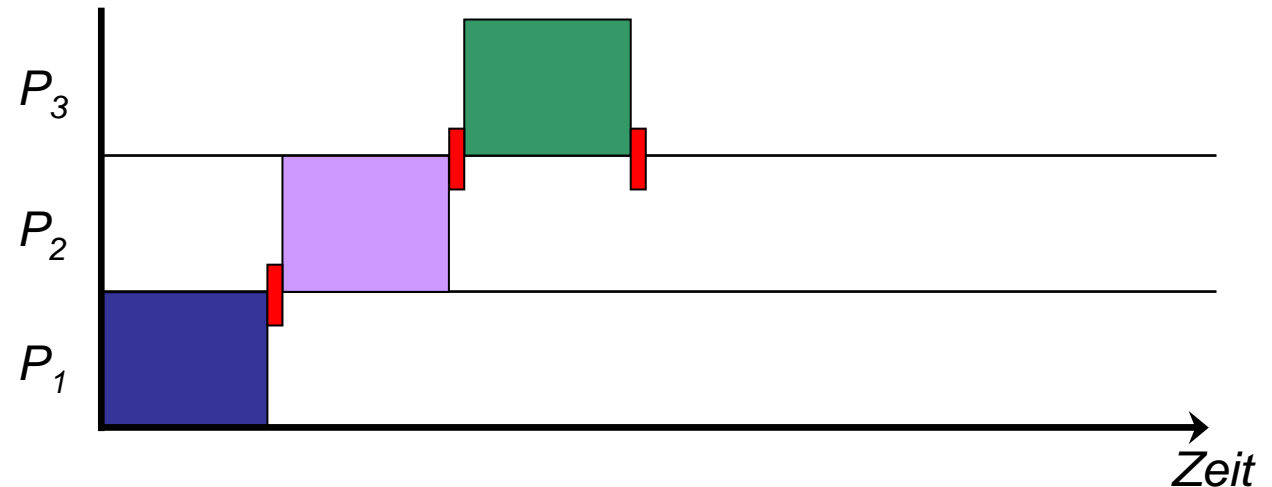


- Jeder Prozeß P_i verfügt über kompletten SPM für Funktionen, (Multi-) BBs & globale Variablen
- Bei Kontextwechsel muß SPM-Inhalt gesichert und neu geladen werden.

Multi-Prozeß SPM-Allokation: Exklusiver SPM-Speicher (1)

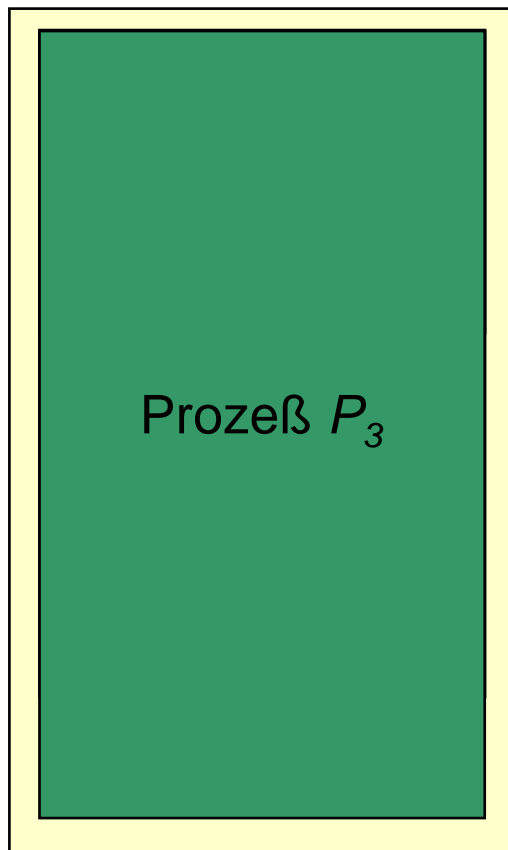


Scratchpad

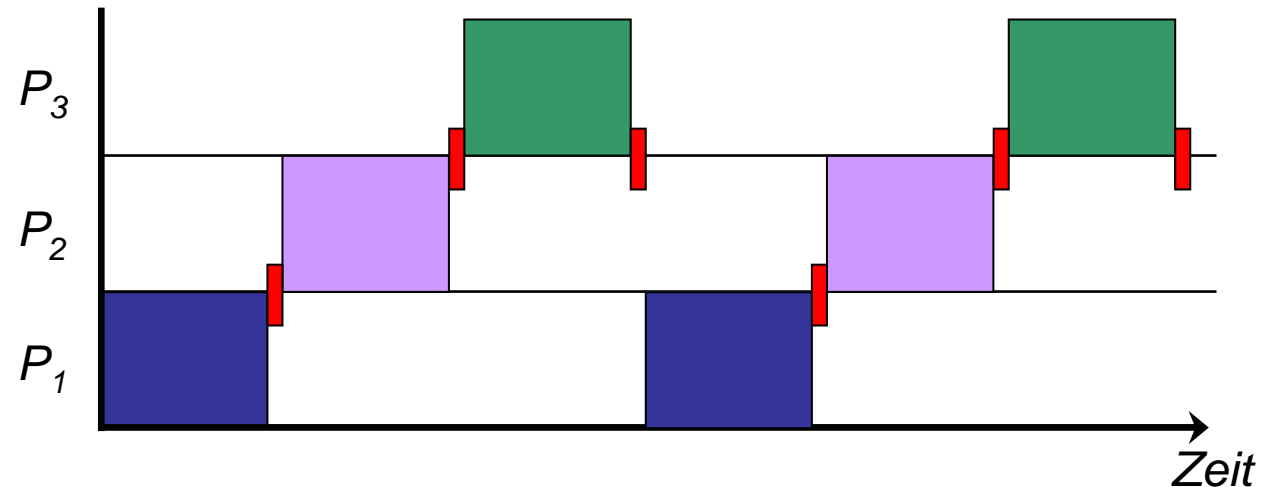


- Jeder Prozeß P_i verfügt über kompletten SPM für Funktionen, (Multi-) BBs & globale Variablen
- Bei Kontextwechsel muß SPM-Inhalt gesichert und neu geladen werden.

Multi-Prozeß SPM-Allokation: Exklusiver SPM-Speicher (1)



Scratchpad



- Jeder Prozeß P_i verfügt über kompletten SPM für Funktionen, (Multi-) BBs & globale Variablen
- Bei Kontextwechsel muß SPM-Inhalt gesichert und neu geladen werden.
- *Erwartung:* Gute Ergebnisse für keine SPMs

Multi-Prozeß SPM-Allokation: Exklusiver SPM-Speicher (2)

Energie-Arrays einzelner Prozesse:

- Für jeden Prozeß P_i bezeichne s_i dessen Schedule-Häufigkeit. s_i wird vorab mit Hilfe von Profiling ermittelt.
- Für P_i ist dessen Energie-Array $f''_i[x]$ zu bestimmen. $f''_i[x]$ gibt an, wieviel Energie P_i verbraucht, wenn P_i x beim Kontextwechsel umzuladende Bytes SPM zur Verfügung hat.
- $f''_i[x]$ wird analog für alle Größen $x = S', 2S', 3S', \dots$ bestimmt:

$$f''_i[x] = f'_i[x] + s_i * (CE_{SPM \rightarrow MM}(x) + CE_{MM \rightarrow SPM}(x))$$
- $f''_i[x] =$ Energie zur Ausführung von P_i mit x Bytes + zusätzlicher Energieverbrauch zum Kopieren von x Bytes zwischen SPM und Hauptspeicher, multipliziert mit der Häufigkeit von Kontextwechseln für P_i .

Multi-Prozeß SPM-Allokation: Exklusiver SPM-Speicher (3)

Energie-Arrays für Multi-Prozesse und exklusiven SPM:

- Für eine Multi-Prozeß Anwendung bestehend aus Prozessen P_1, \dots, P_N ist deren Energie-Array $e''_N[x]$ zu bestimmen. $e''_N[x]$ gibt an, wieviel Energie die Anwendung verbraucht, wenn sie x Bytes SPM zur Verfügung hat, die (z.T.) beim Kontextwechsel umzuladen sind.

- $$e''_N[x] = \sum_{P_i} \min\{ f''_i[x_j] \mid x_j \leq x \}$$
 für alle Werte von x und x_j ,

für die die einzelnen f''_i definiert sind.

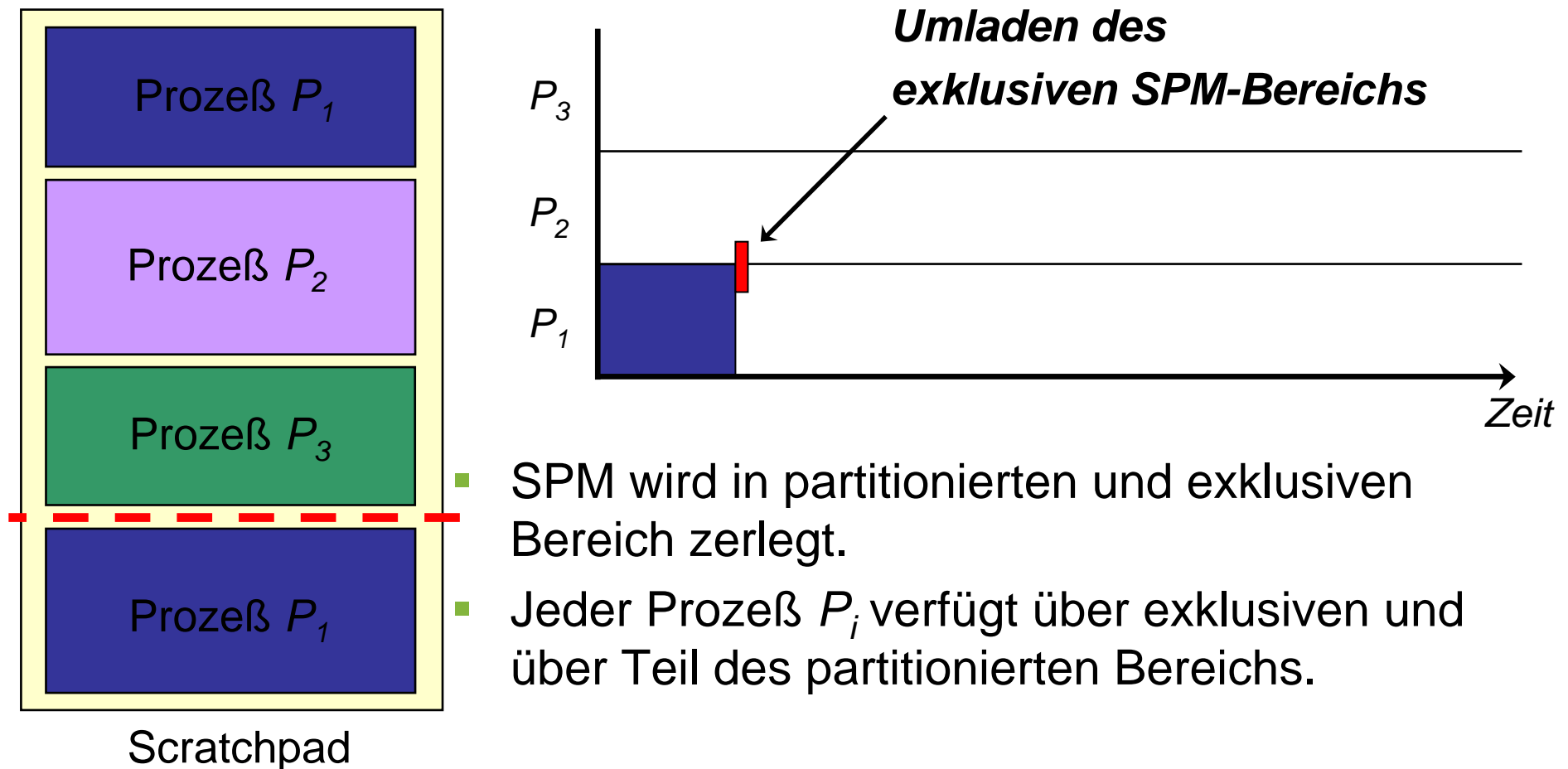
- Für eine gegebene SPM-Größe S und eine Multi-Prozeß Anwendung ist man an einer SPM-Allokation mit minimalem Wert $e''_N[S]$ interessiert.

Multi-Prozeß SPM-Allokation: Exklusiver SPM-Speicher (4)

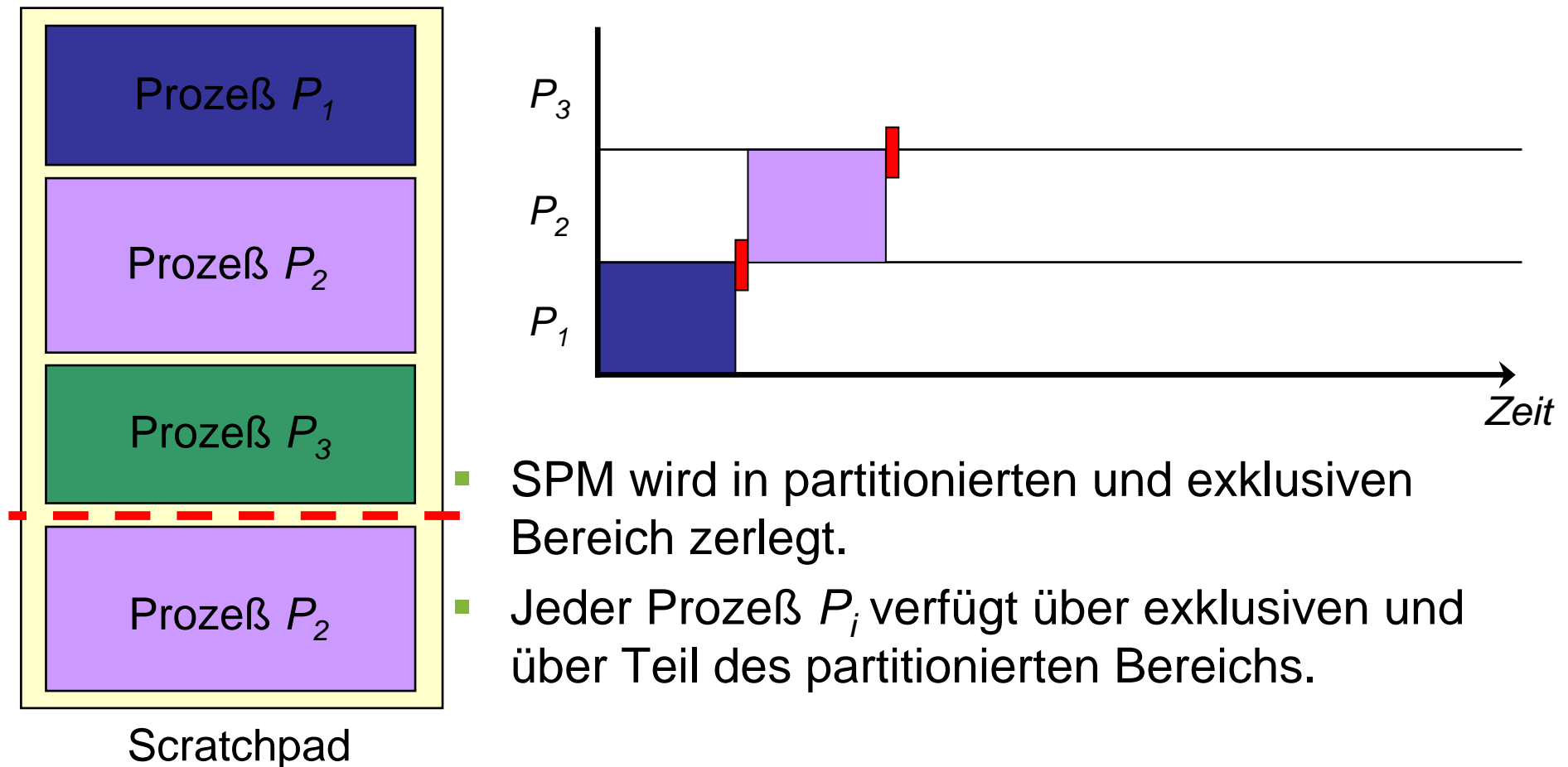
Algorithmus zur Berechnung von e''_N :

- for (*<alle Prozesse P_i >*)
 - $prev_min[i] = \infty$;
- for ($x = 0$; $x \leq S$; $x += S'$)
 - $e''_N[x] = 0$;
 - for (*<alle Prozesse P_i >*)
 - $min[i] = (f''_i[x] < prev_min[i]) ? f''_i[x] : prev_min[i]$;
 - $e''_N[x] += min[i]$;
 - $prev_min[i] = min[i]$;
- return e''_N ;

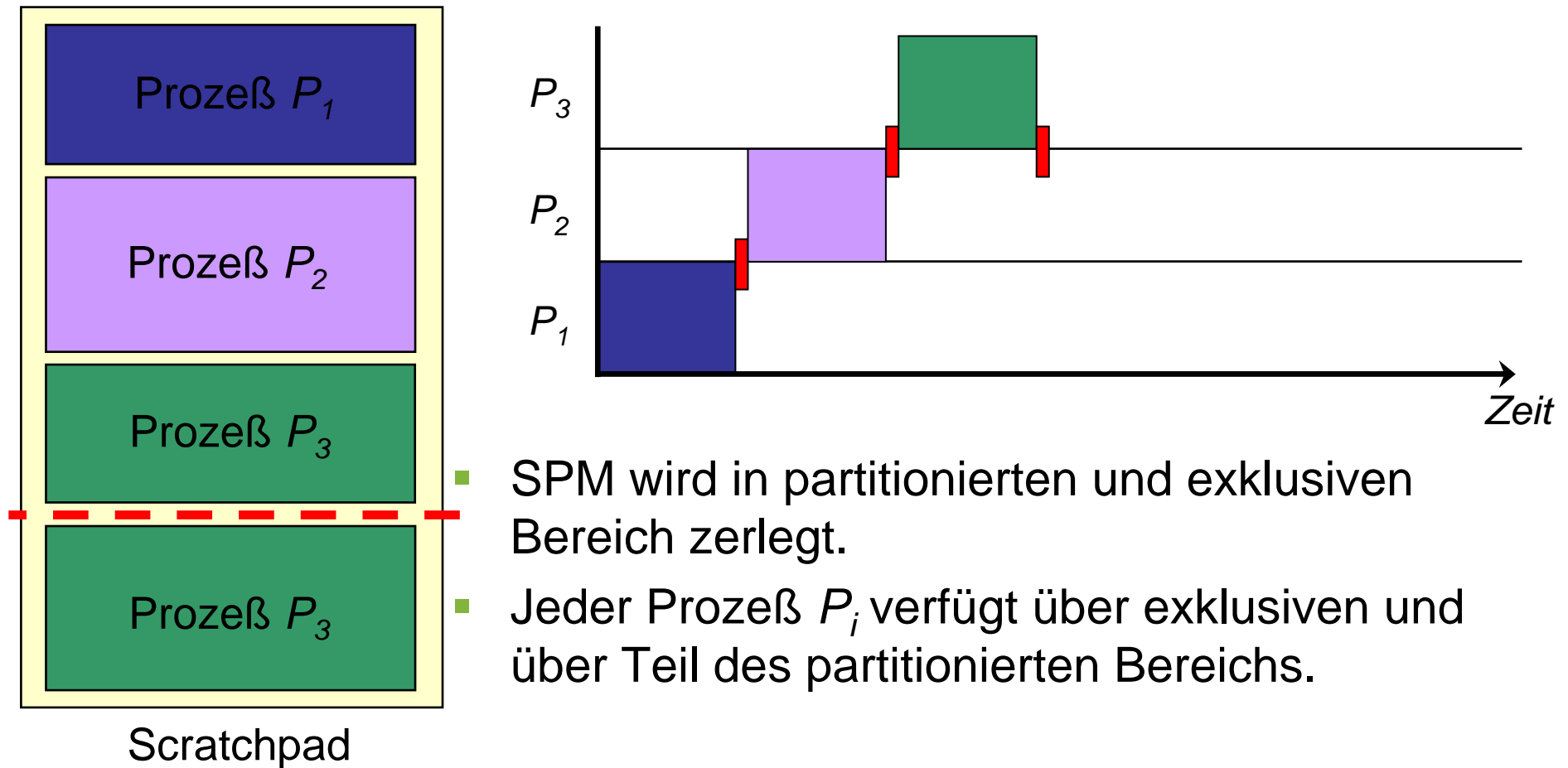
Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (1)



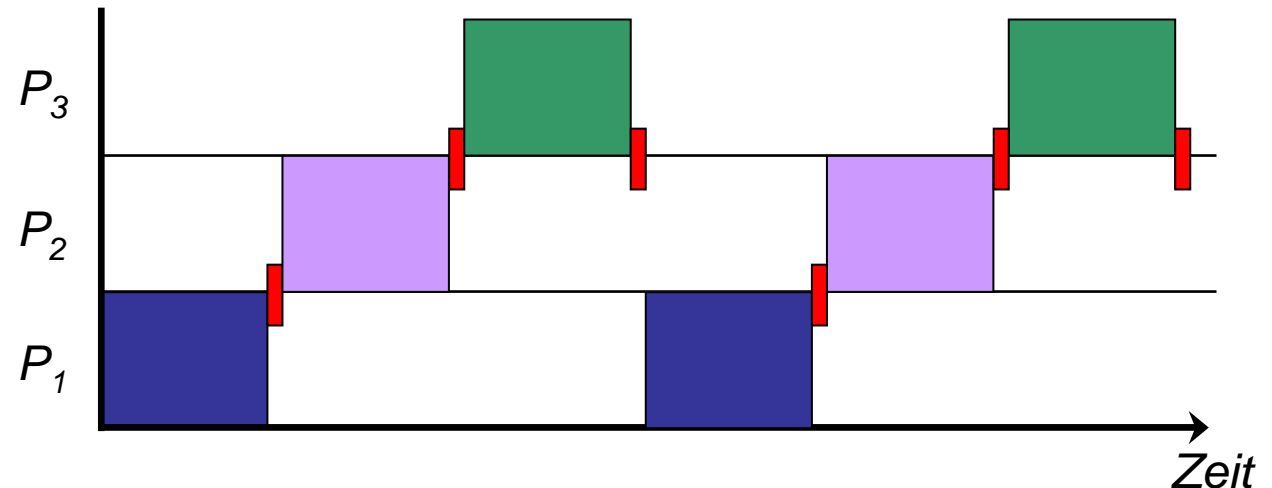
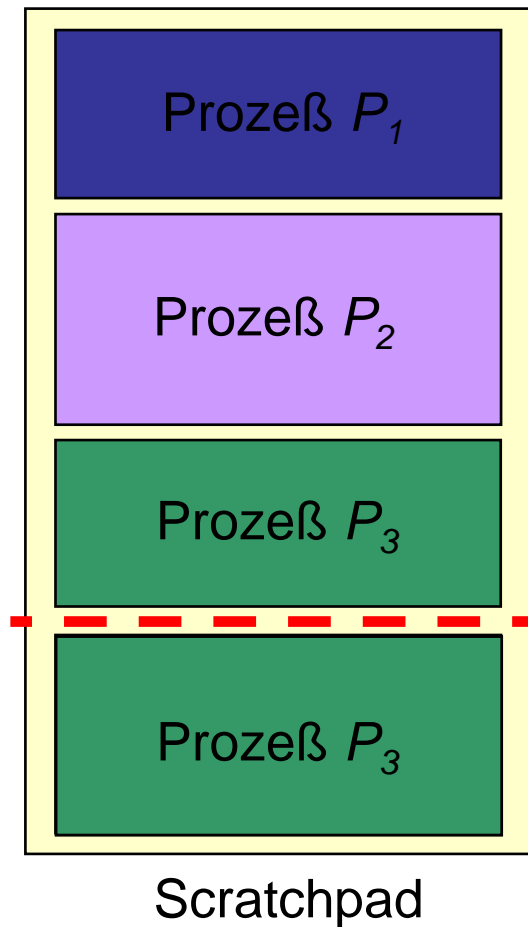
Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (1)



Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (1)



Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (1)



- SPM wird in partitionierten und exklusiven Bereich zerlegt.
- Jeder Prozeß P_i verfügt über exklusiven und über Teil des partitionierten Bereichs.
- *Erwartung:* Gute Ergebnisse für alle SPMs

Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (2)

Energie-Arrays einzelner Prozesse:

- Für P_i ist dessen Energie-Array $f''''_i[x, y]$ zu bestimmen. $f''''_i[x, y]$ gibt an, wieviel Energie P_i verbraucht, wenn P_i über x Bytes partitionierten und über y beim Kontextwechsel umzuladende Bytes exklusiven SPM verfügt.
- $f''''_i[x, y]$ wird wiederum für alle Größen $x, y = S', 2S', 3S', \dots$ bestimmt, indem eine spezielle Variante der fixen SPM-Allokation gelöst wird, die zwei SPMs der Größen x und y unterstützt.

Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (3)

Energie-Arrays für Multi-Prozesse und hybriden SPM:

- Für eine Multi-Prozeß Anwendung mit Prozessen P_1, \dots, P_N ist deren Energie-Array $e'''_N[x, y]$ zu bestimmen. $e'''_N[x, y]$ gibt den Energieverbrauch der Anwendung an, wenn sie x Bytes partitionierten und y Bytes exklusiven SPM zur Verfügung hat.
- $$e'''_N[x, y] = \min\{ f'''_1[x_1, y_1] + \dots + f'''_N[x_N, y_N] \mid x_1 + \dots + x_N \leq x \text{ und } y_i \leq y \text{ für alle } i = 1, \dots, N \}$$
 für alle Werte von $x, x_1, \dots, x_N, y_1, \dots, y_N$, für die die einzelnen f'''_i definiert sind.
- Für eine gegebene SPM-Größe S und eine Multi-Prozeß Anwendung ist man an einer SPM-Allokation mit minimalem Wert $e'''_N[x, y]$ interessiert, so daß $x + y \leq S$ gilt.

Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (4)

Die hybrid_binmin-Funktion:

- Annahme: Eine Multi-Prozeß Anwendung bestehe aus 3 Prozessen P_1 , P_2 und P_3 .
- $$e'''_3[x, y] = \min\{ f'''_1[x_1, y_1] + f'''_2[x_2, y_2] + f'''_3[x_3, y_3] \mid$$

$$x_1 + x_2 + x_3 \leq x \wedge y_1 \leq y \wedge y_2 \leq y \wedge y_3 \leq y \}$$

$$= \text{hybrid_binmin}(\text{hybrid_binmin}(f'''_1, f'''_2), f'''_3)$$
- hybrid_binmin ist eine Funktion, die zwei Energie-Arrays g und h miteinander verknüpft und wiederum ein Energie-Array liefert:

$$\text{hybrid_binmin}(g, h)[x, y] =$$

$$\min\{ g[x_1, y_1] + h[x_2, y_2] \mid x_1 + x_2 \leq x \wedge y_1 \leq y \wedge y_2 \leq y \}$$
- Aufgrund der Distributivität kann die Minimum-Bildung über eine N -fache Summe in $e'''_N[x, y]$ auf das binäre min in hybrid_binmin reduziert werden.

Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (5)

Algorithmus zur Berechnung von `hybrid_binmin(g, h)`:

- for ($y = 0; y \leq S; y += S'$)
 - int $min = \infty$;
 - for ($x = 0; x \leq S - y; x += S'$)
 - for ($tmp = 0; tmp \leq x; tmp += S'$)
 - if ($g[tmp, y] + h[x - tmp, y] < min$)
 $min = g[tmp, y] + h[x - tmp, y]$;
 - $b[x, y] = min$;
- return b ;

Multi-Prozeß SPM-Allokation: Hybrider SPM-Speicher (6)

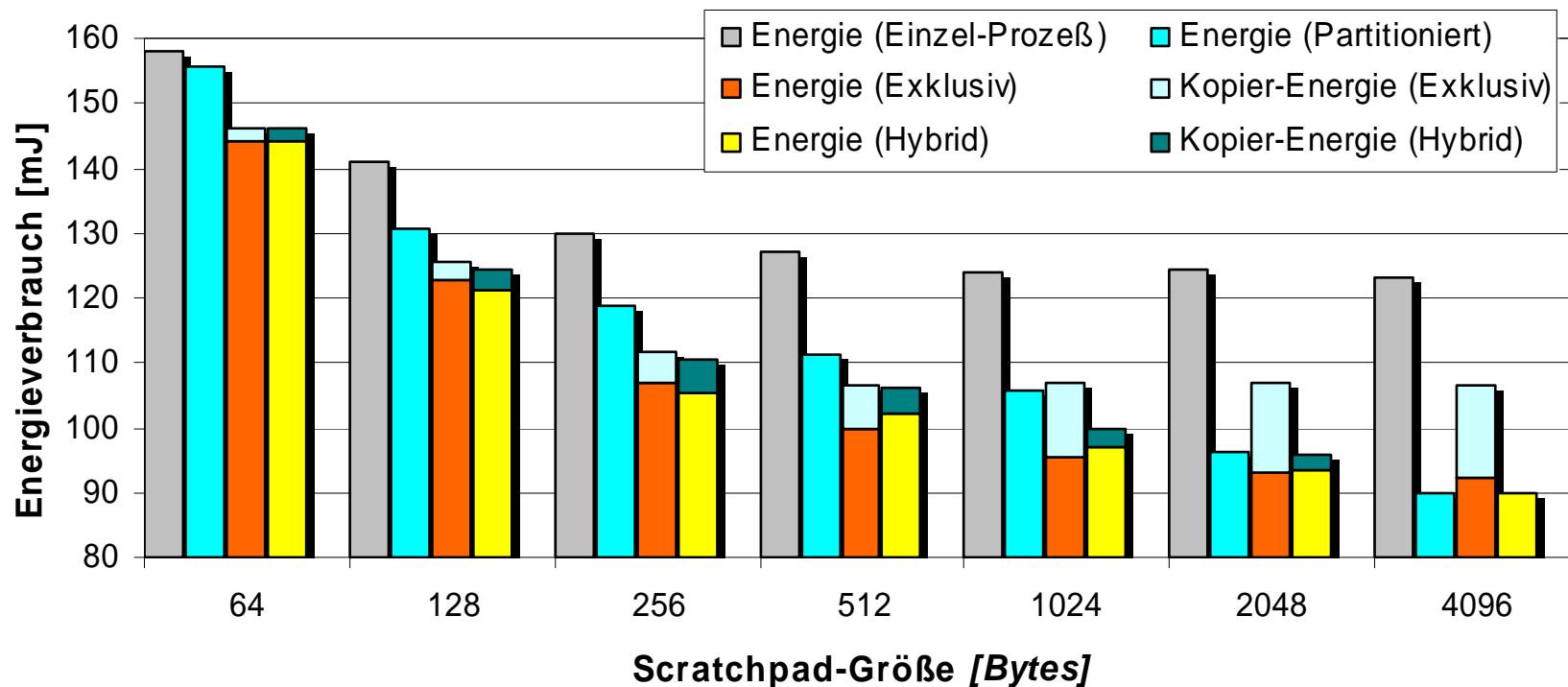
Algorithmus zur Berechnung von e'''_N :

HybridSPM(f'''_1, \dots, f'''_N)

- if ($N > 1$)
 - $e'''_{N-1} = \text{HybridSPM}(f'''_1, \dots, f'''_{N-1});$
 - $e'''_N = \text{hybrid_binmin}(e'''_{N-1}, f'''_N);$
- else
 - $e'''_N = \text{hybrid_binmin}(f'''_1, Z);$
- return e'''_N ;

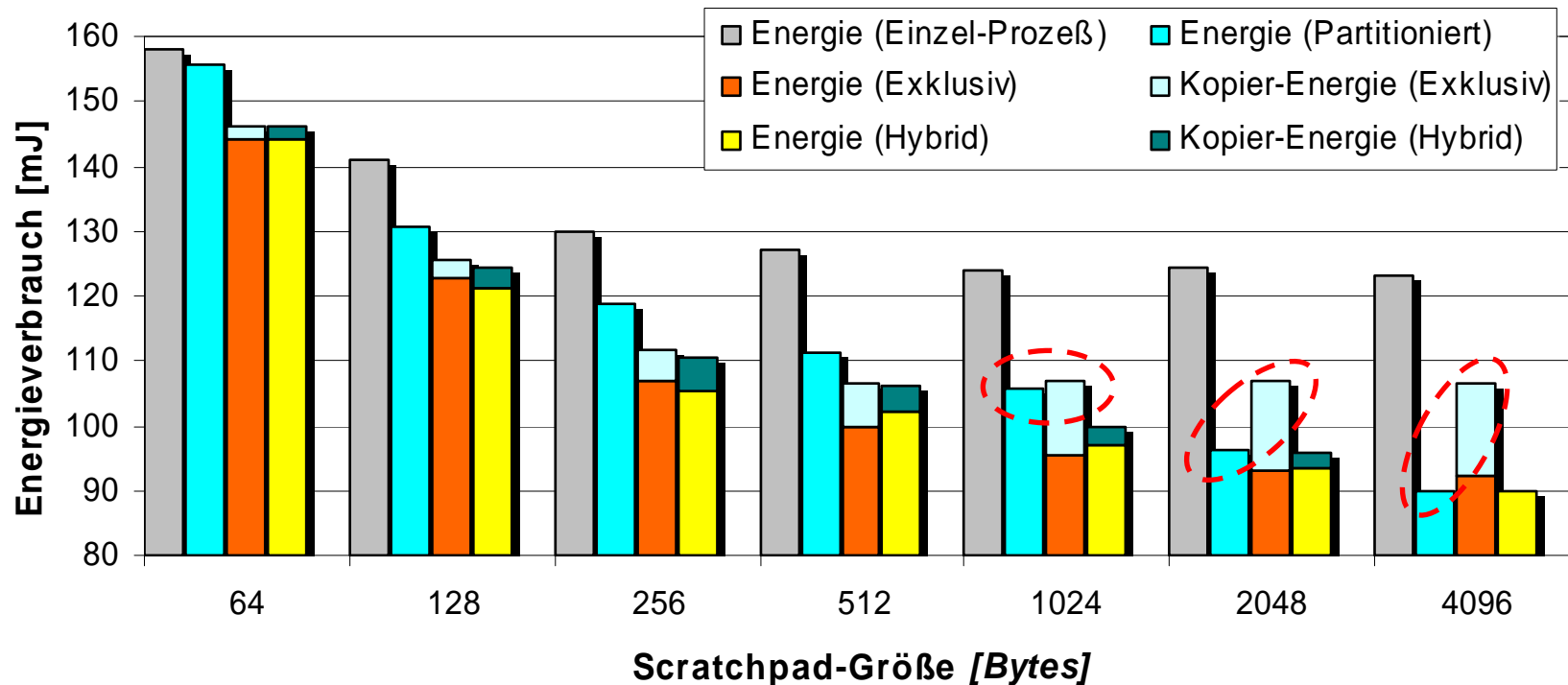
- Funktionsweise analog zu PartitionedSPM (☞ Folie 82)
- Z ist ein wiederum Energie-Array, das nur Nullen für alle SPM-Größen enthält.

Multi-Prozeß SPM-Allokation: Ergebnisse (1)



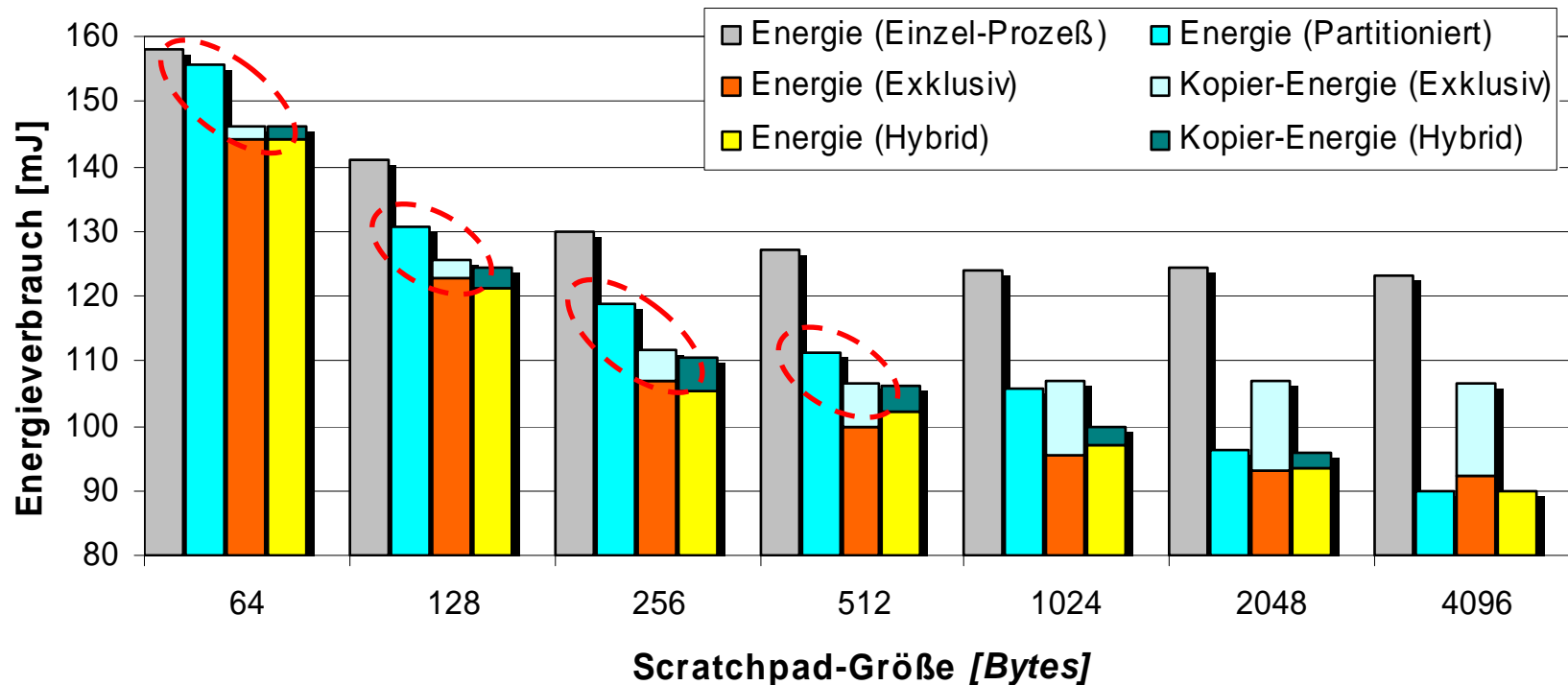
- Hier: Media-Applikation (Prozesse adpcm, g721, mpeg4, edge-detection)
- *Energie (Einzel-Prozeß)*: SPM wird komplett demjenigen Prozeß per fixer SPM-Allokation zugewiesen, der zur höchsten Energieeinsparung führt.

Multi-Prozeß SPM-Allokation: Ergebnisse (2)



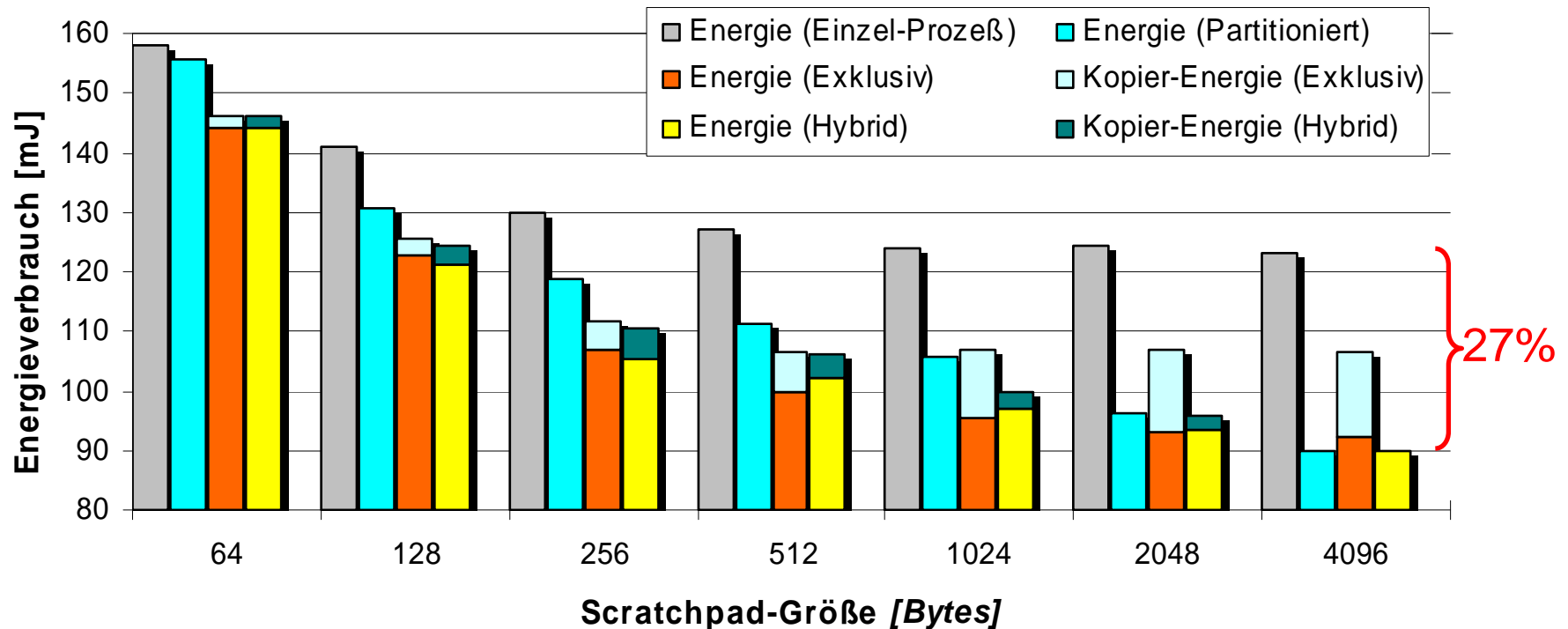
- *Partitionierter SPM*: Tatsächlich vorteilhaft für große Speicher
- Partitionierung des SPMs schlägt exklusive SPM-Nutzung ab 1kB SPM-Größe

Multi-Prozeß SPM-Allokation: Ergebnisse (3)



- *Exklusiver SPM*: Tatsächlich vorteilhaft für kleine Speicher
- Exklusive SPM-Nutzung mit Umladen des SPMs schlägt partitionierten SPM bis 512 Bytes SPM-Größe

Multi-Prozeß SPM-Allokation: Ergebnisse (4)



- *Hybride SPM-Einteilung*: Ist für alle SPM-Größen das beste Verfahren
- Bei 4kB SPM: 27% Energieeinsparung vgl. mit Balken “Einzel-Prozeß”
- Ähnliche Ergebnisse auch für “Video Phone” und “DSP”-Applikationen

Literatur

- **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.
 - M. Verma, K. Petzold, L. Wehmeyer et al., *Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A first Approach*, 3rd IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), Jersey City, September 2005.

Zusammenfassung

- **Generierung von Bit-Paket Operationen für NPUs**
 - Konventionelle Datenflußanalysen sind nicht bitgenau
 - Bitgenaue Daten- & Wertflußanalysen per Vor- / Rückwärts-Simulation
 - Entdeckung von Bit-Paketen mittels \uparrow -Werten der BDWFA
- **Optimierungen für Scratchpad-Speicher**
 - Scratchpads extrem vorteilhaft bzgl. Energieverbrauch, Laufzeit und $WCET_{EST}$, verglichen mit Caches und Hauptspeicher
 - Ganzzahlig-lineare Programmierung (ILP) zur Optimierung
 - SPM-Inhalt: Funktionen, Basisblöcke und globale Variablen
 - SPM-Allokation für Mono- und Multi-Prozeß Anwendungen