Compiler für Eingebettete Systeme (CfES)

Sommersemester 2009

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

Kapitel 8

Compiler zur $WCET_{EST}$-Minimierung

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: Instruktionsauwahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- **Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung**
  - Einführung
  - Procedure Cloning & Positioning
  - Register-Allokation
  - Scratchpad-Allokation von Daten und Code
- Kapitel 9: Ausblick

Derzeitiger Entwurf von Realzeit-Systemen

1. (Spezifikation in grafischen Werkzeugen)
2. (Automatische) Erzeugung von ANSI-C Code
3. Übersetzen in ausführbaren Maschinencode für gegebenen Ziel-Prozessor
4. Vielfaches Ausführen / Simulieren des Maschinencodes, Verwendung „repräsentativer“ Eingabedaten
5. Zeitmessung liefert *„beobachtete Laufzeit”*
6. Aufschlag von Sicherheitspuffer (z.B. 20%) auf höchste beobachtete Laufzeit: *„beobachtete Worst-Case Execution Time (WCET)”*
7. Beobachtete WCET ≤ Realzeit-Bedingungen? Nein: Gehe zu 1

Probleme dieses Ablaufs

Sicherheit:

- Keine Gewähr, daß beobachtete WCET annähernd effektiver WCET entspricht
- ☞ Keine Garantie, daß hartes Realzeit-System *immer* pünktlich arbeitet

Entwurfszeit:

- Wie viele Iterationen notwendig, bis Schritt 7 erfolgreich?
- ☞ Abhängig davon, inwieweit Schritte 2-3 zu effektiver Code-Beschleunigung führen *(try & error)*

Derzeitiger Stand der Compiler-Technik

Zielfunktion heutiger Compiler-Optimierungen:

- I. d. R. Reduktion durchschnittlicher Laufzeiten *(Average-Case Execution Time, ACET)*: Beschleunige „typische” Ausführung eines Programms mit „typischen” Eingabedaten
- ☞ Keine Aussagen über WCET möglich

Optimierungsstrategie:

- Naiv: Compiler enthalten kein echtes ACET Zeitmodell
- Anwendung einer Optimierung, wenn „Erfolg versprechend”
- ☞ Auswirkung von Optimierungen auf ACET in Compiler vollkommen unbekannt

Motivation

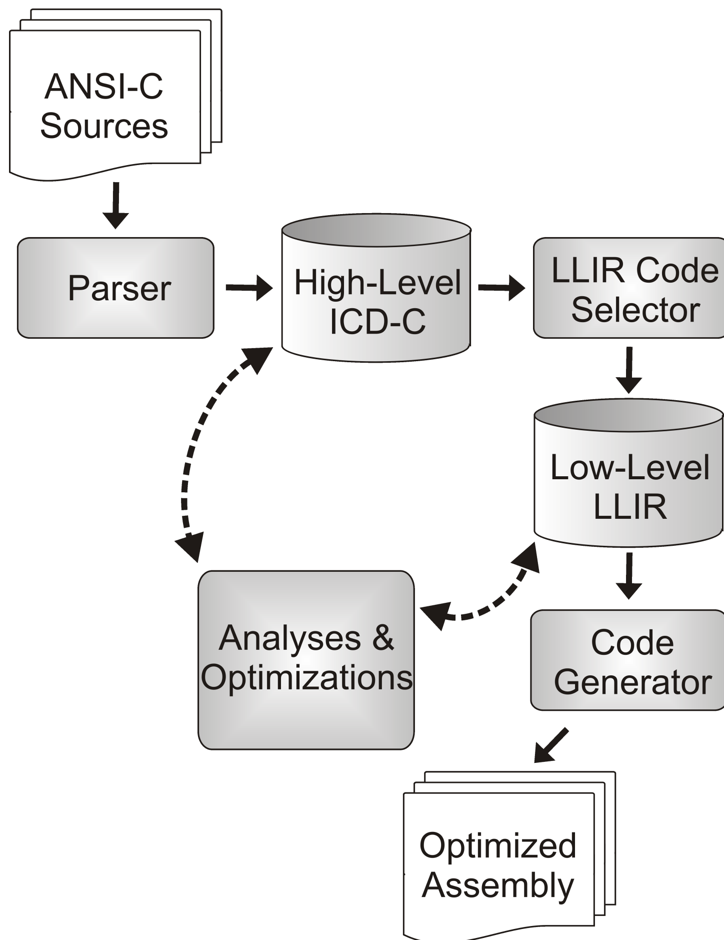
Entwurf eines Compilers, der

- statt durchschnittlicher Laufzeiten WCET betrachtet,
- statt beobachteter WCETs sichere WCET-Abschätzungen erlaubt,
- automatische Optimierungen zur WCET-Minimierung durchführt.

Ansatz:

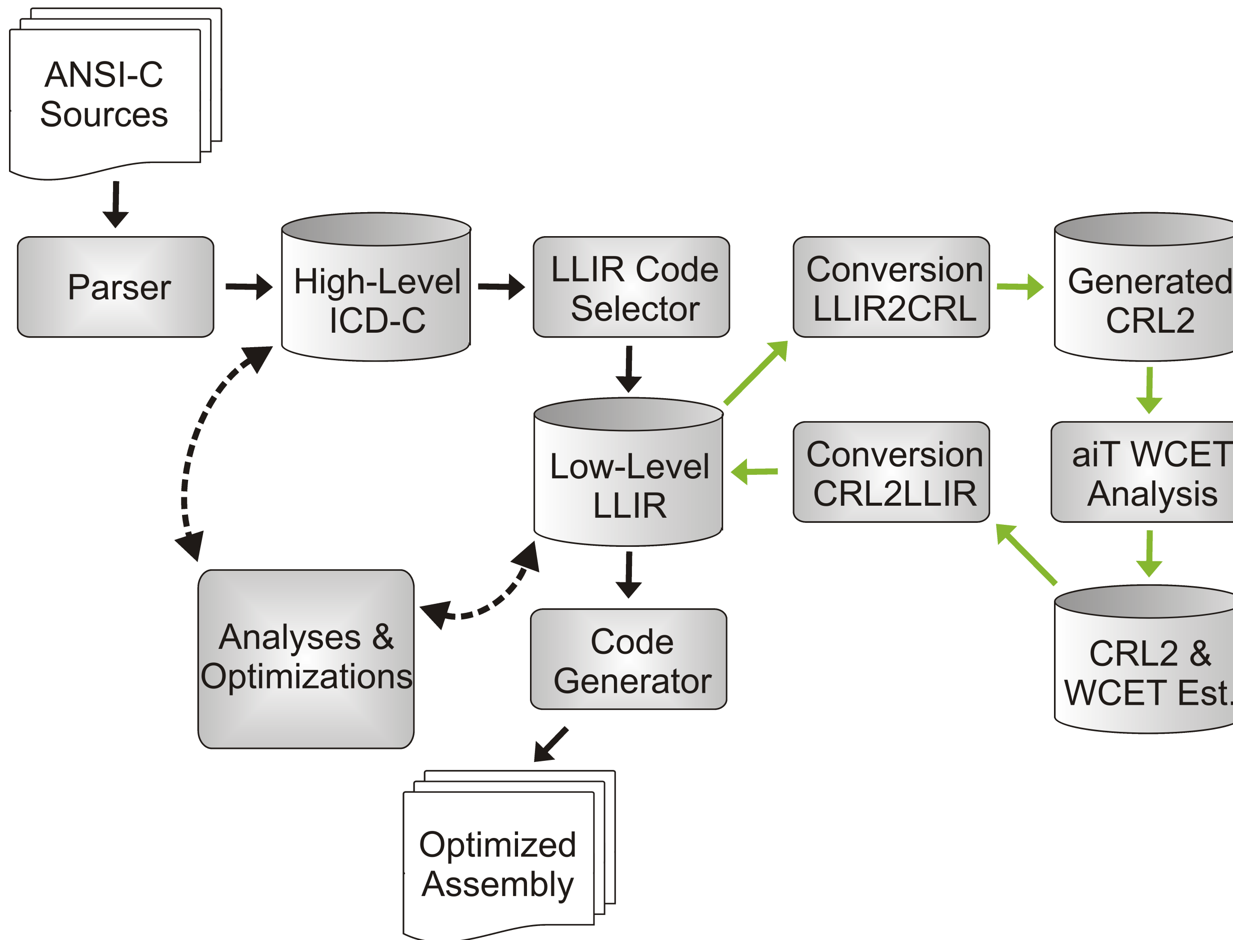
- Integration eines WCET Zeitmodells in Compiler durch Anbindung statischer WCET-Analysewerkzeuge
- Ausnutzung des WCET Zeitmodells zur gezielten WCET-Minimierung in neuartigen Optimierungen

WCET-Integration in Compiler (1)



- Re-Implementierung eines WCET Zeitmodells in Compiler nicht sinnvoll
- Statt dessen: Einbindung von aiT
  (☞ siehe Kapitel 4)
- Kopplung in Prozessor-spezifischem Compiler Back-End *(LLIR)*
- Informationsaustausch durch Konvertierung LLIR ↔ CRL2
- Transparenter Aufruf von aiT innerhalb des Compilers
- Import WCET-relevanter Daten in Compiler Back-End

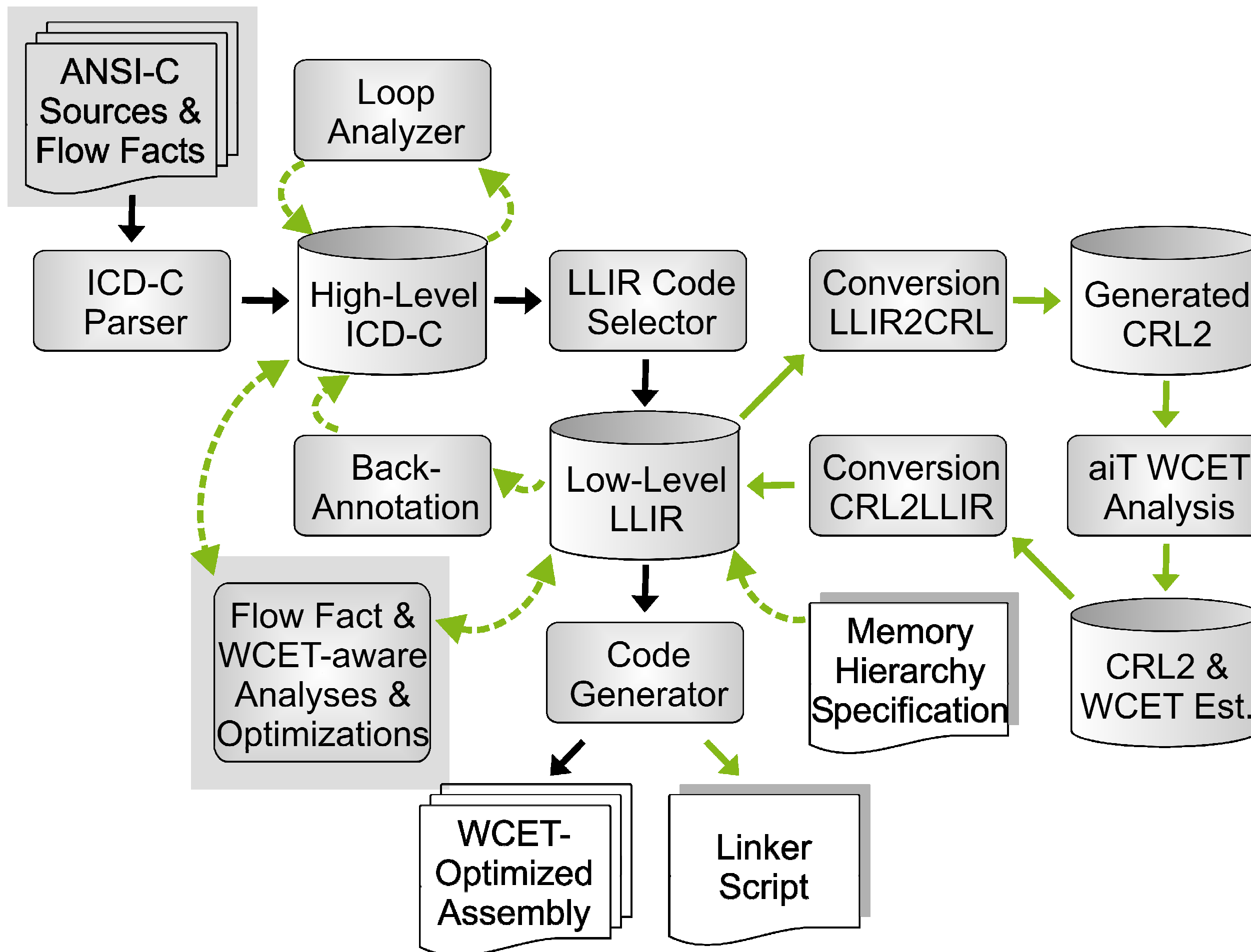
WCET-Integration in Compiler (2)



Relevante WCET-Daten:

- $WCET_{EST}$ für Programm, Funktion, Basisblock
- Worst-Case Ausführungshäufigkeit pro Funktion, Basisblock, Kante im CFG
- Potentielle Registerinhalte
- Cache Hits / Misses pro Basisblock

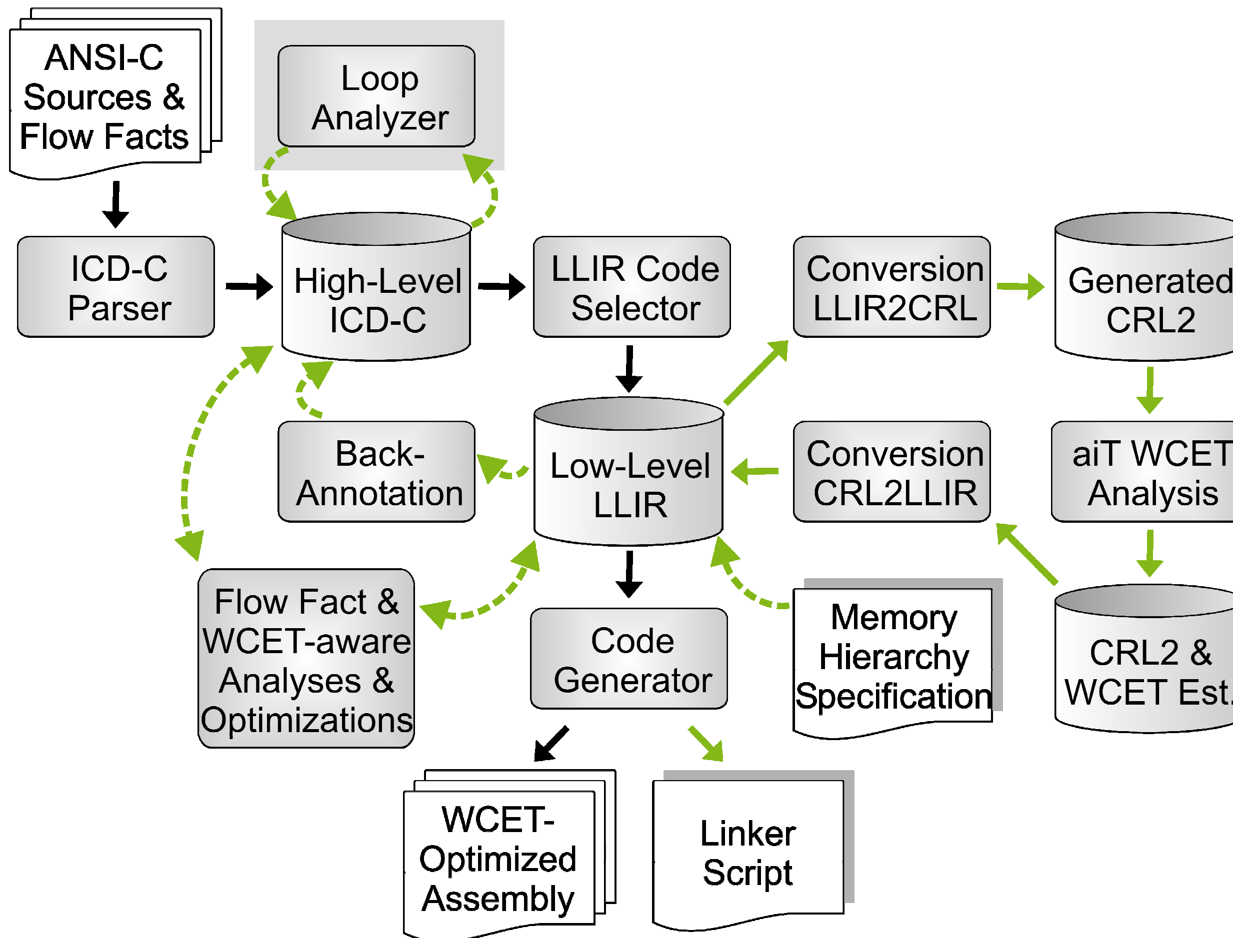
WCC – WCET-aware C Compiler (1)



Flow Facts:

- WCET-Analyse: max. Iterationszahlen & Rekursionstiefen
- WCC: Annotation direkt im C Quellcode: `_Pragma( „loopbound min 10 max 10“ );`
- Optimierungen, die Kontrollfluß ändern, aktualisieren Flow Facts automatisch.

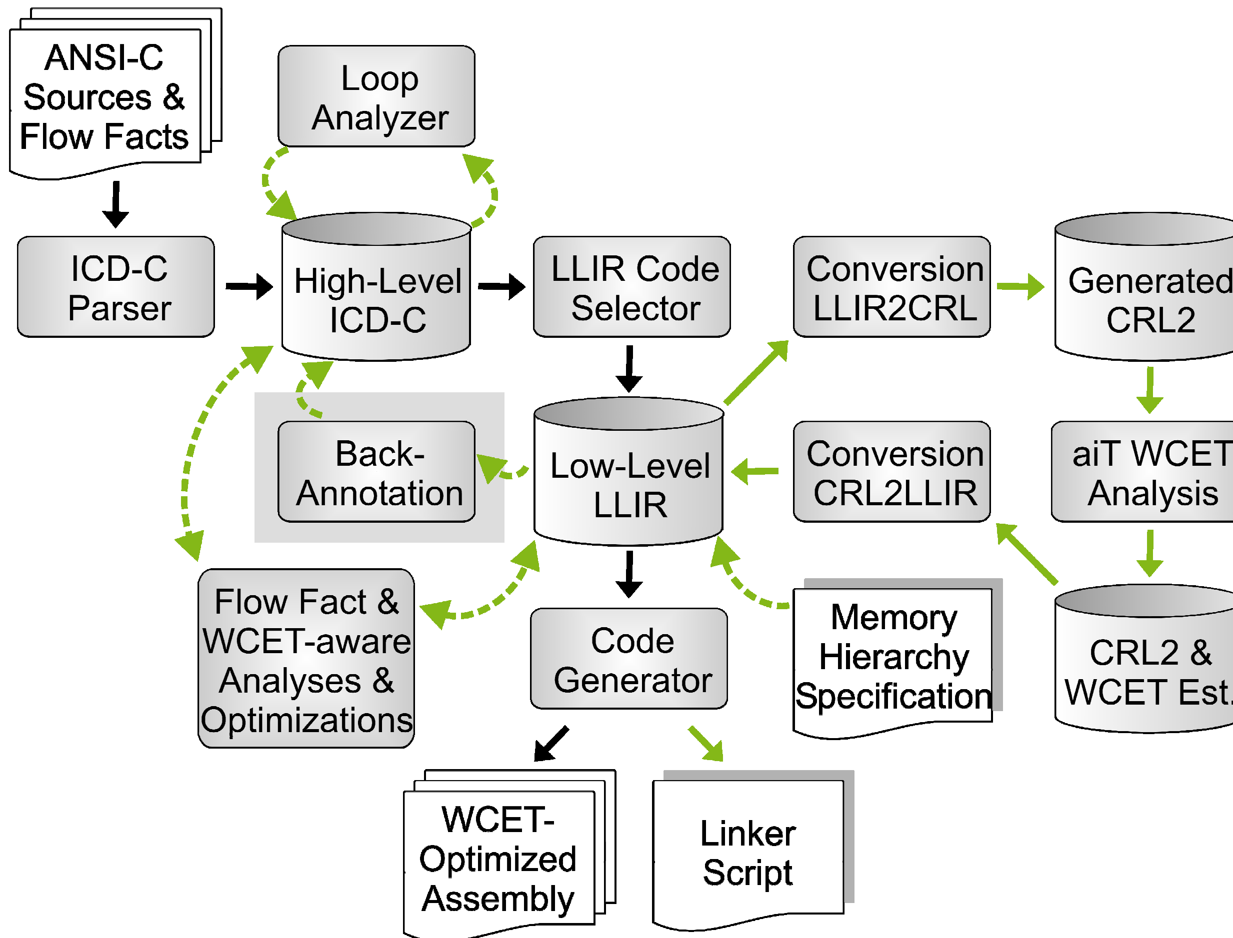
WCC – WCET-aware C Compiler (2)



Loop Analyzer:

- Manuelle Flow Fact Annotation umständ-lich und fehleranfällig
- WCC: Automatische Schleifenanalyse, die max. Iterationszahlen ermittelt.
- Basiert z.T. auf Polytop-Modellen (☞ *siehe Kapitel 3)*

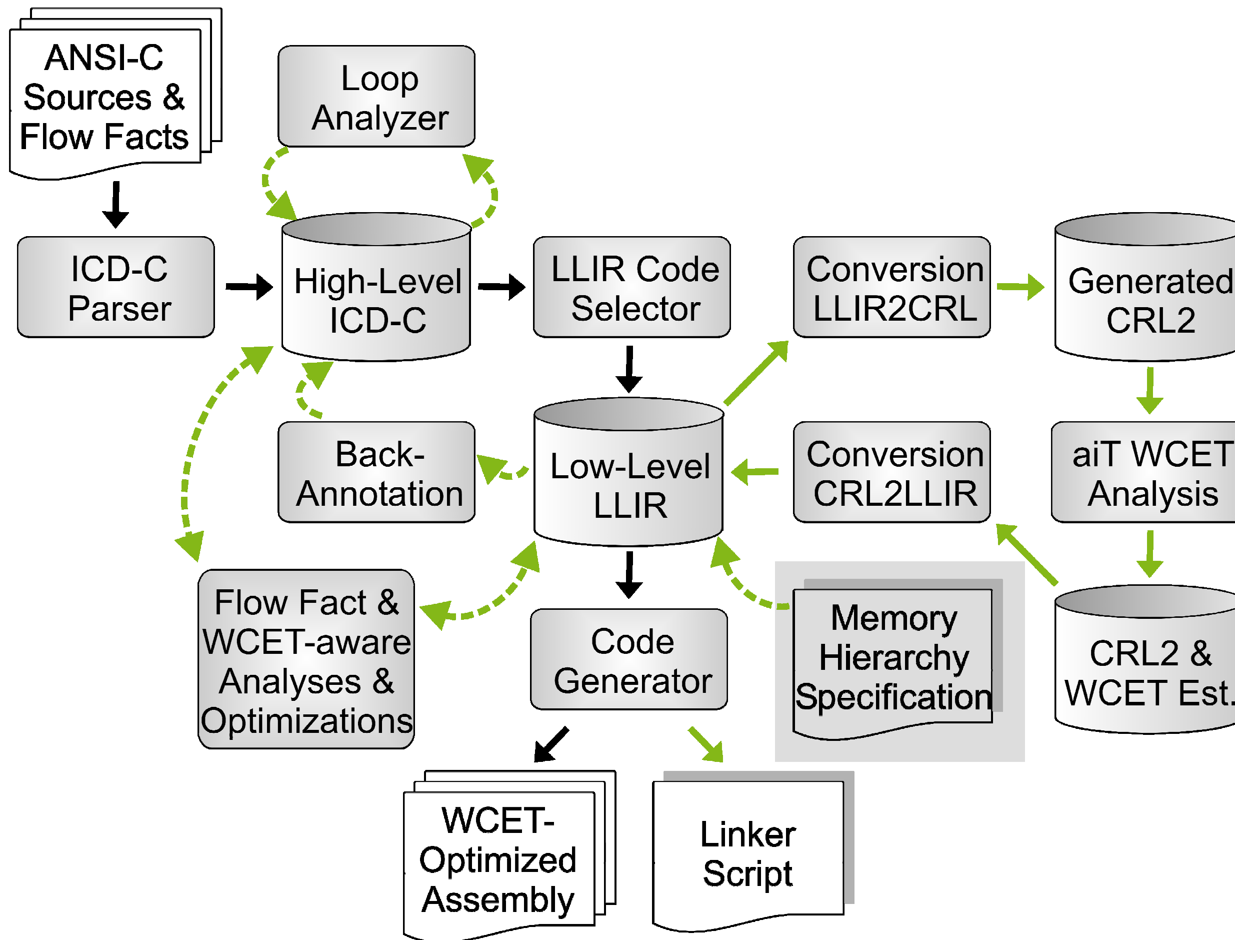
WCC – WCET-aware C Compiler (3)



Back-Annotation:

- WCET-Daten von aiT nur im Back-End
- HIR-Optimierungen haben keinen Zugriff auf WCET-Daten
- $WCET_{EST}$-Minimierung auf HIR-Ebene nicht möglich
- WCC: Back-Annotation übersetzt WCET-Daten von LIR nach HIR

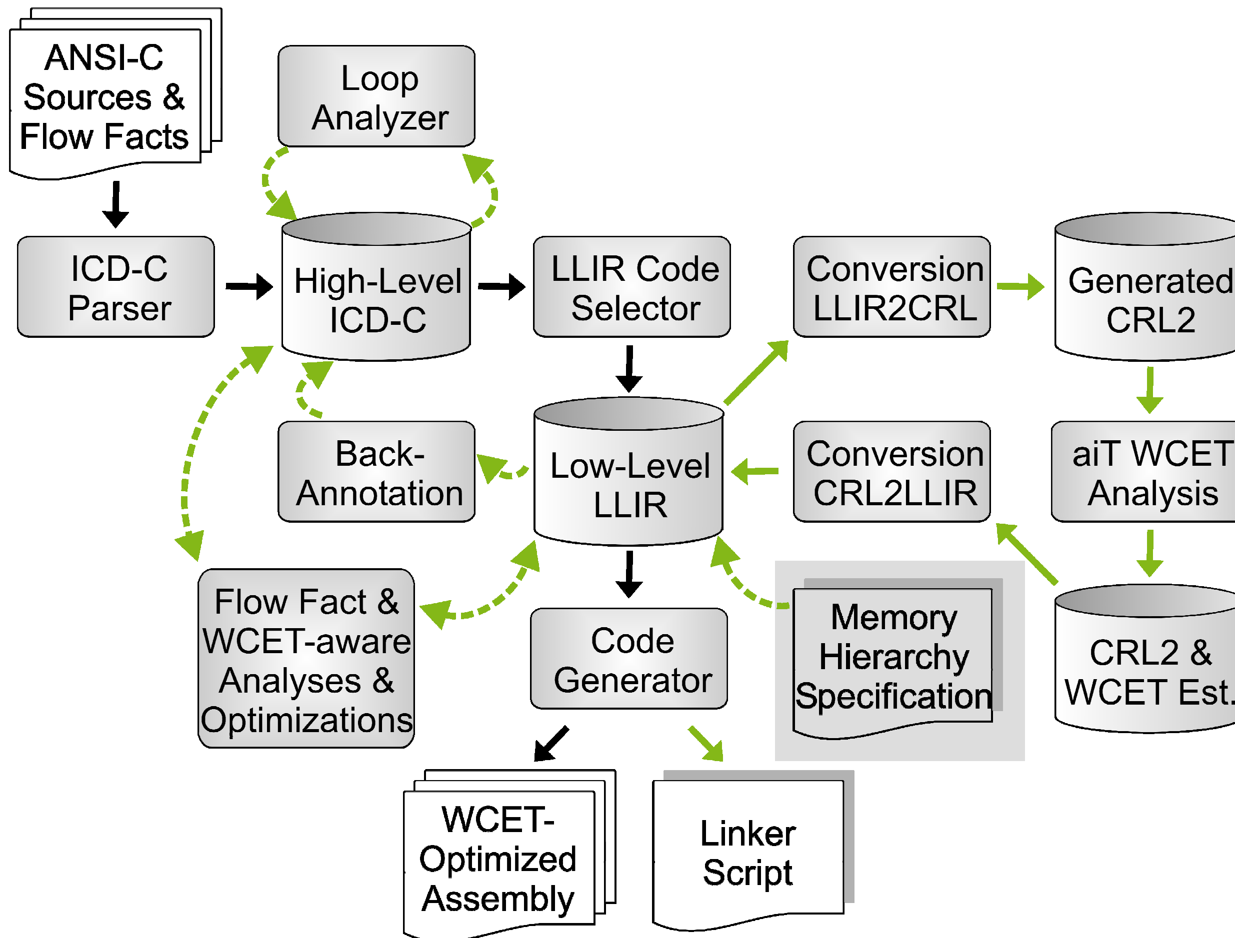
WCC – WCET-aware C Compiler (4)



Memory Hierarchy:

- aiT arbeitet auf Binär-code mit physikali-schen Adressen
- WCC muß aiT kor-rekte phys. Adressen für Code, Daten, Sprünge und Load- / Store-Operationen bereitstellen
- WCC braucht Detail-wissen über Speicher

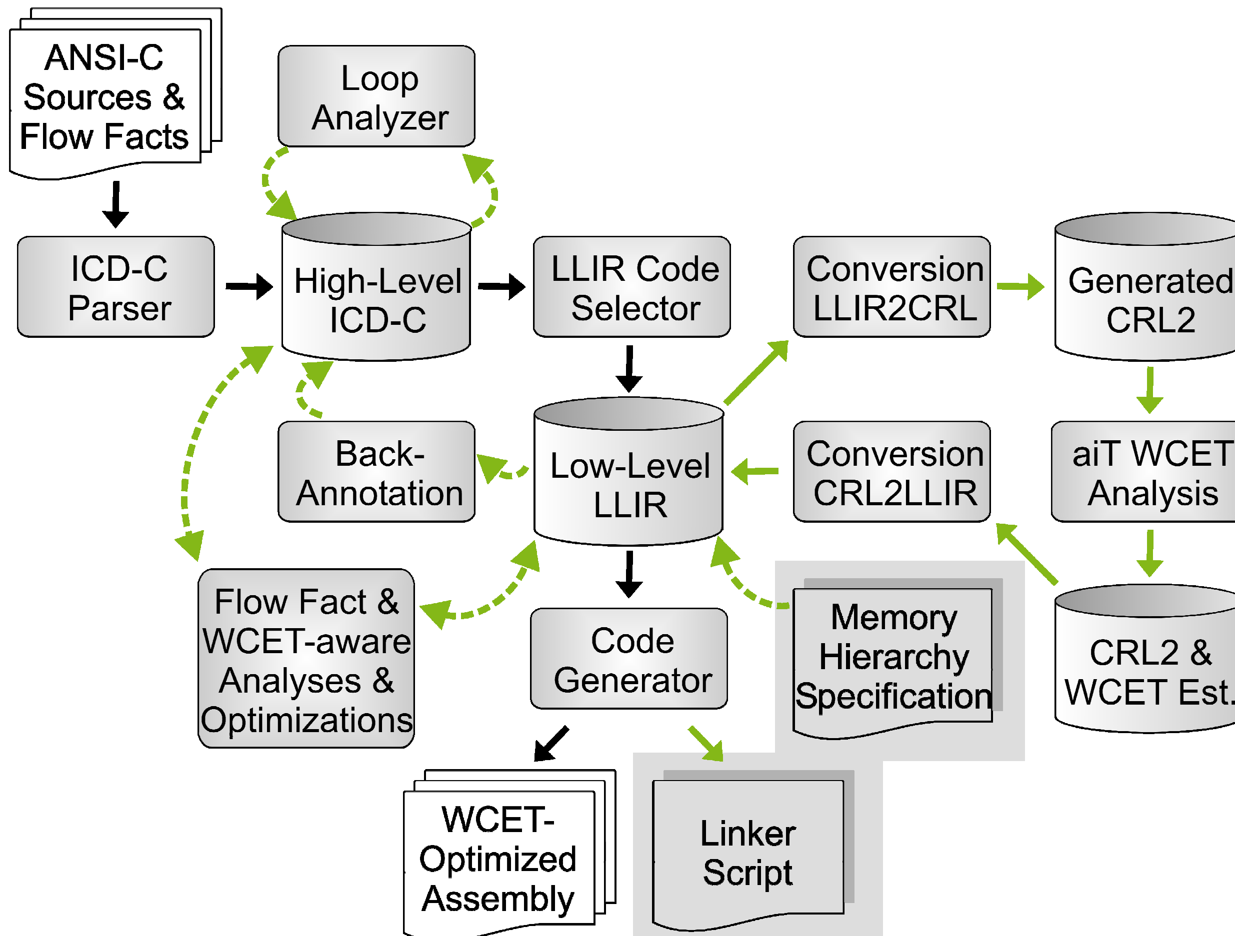
WCC – WCET-aware C Compiler (5)



Memory Hierarchy:

- Speicher-Regionen, Startadresse, Größe, Zugriffsgeschwindigkeit, Verwendbarkeit (Code, Daten, les- / schreibbar), ...
- SPM-Allokationen benötigen ebenfalls diese Informationen zur Optimierung

WCC – WCET-aware C Compiler (6)



[http://ls12-www.cs.tu-dortmund.de/research/activities/wcc]

Memory Hierarchy:

- WCC entscheidet über Speicher-Layout von Code & Daten, produziert aber keinen Binärcode
- Linker muß Binärcode exakt gemäß WCC Speicher-Layout produzieren
- WCC: automatische Generierung eines Linkerskripts

Problematik der $WCET_{EST}$-Minimierung

Der Worst-Case Execution Path (WCEP):

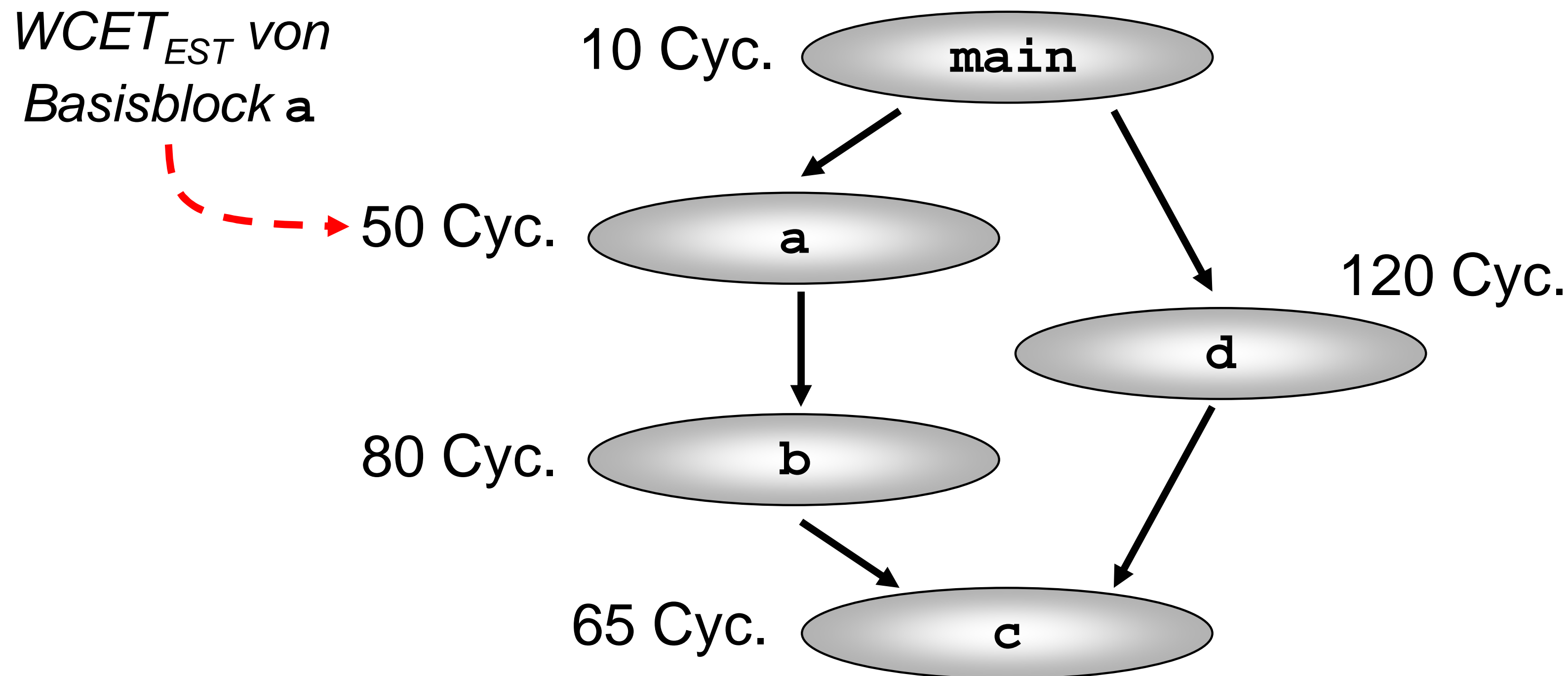
- Die WCET eines Programmes entspricht der Länge des längsten Ausführungspfades des Programmes *(WCEP)*
- Um die $WCET_{EST}$ eines Programmes zu minimieren, muß man sich ausschließlich auf die Teile des Programmes konzentrieren, die auf dem WCEP liegen.

☞ Optimierung von Programmteilen, die nicht auf dem WCEP liegen, führt zu keinerlei Verbesserung!

☞ Optimierungsstrategien zur $WCET_{EST}$-Minimierung brauchen zwingend Detailwissen über den WCEP.

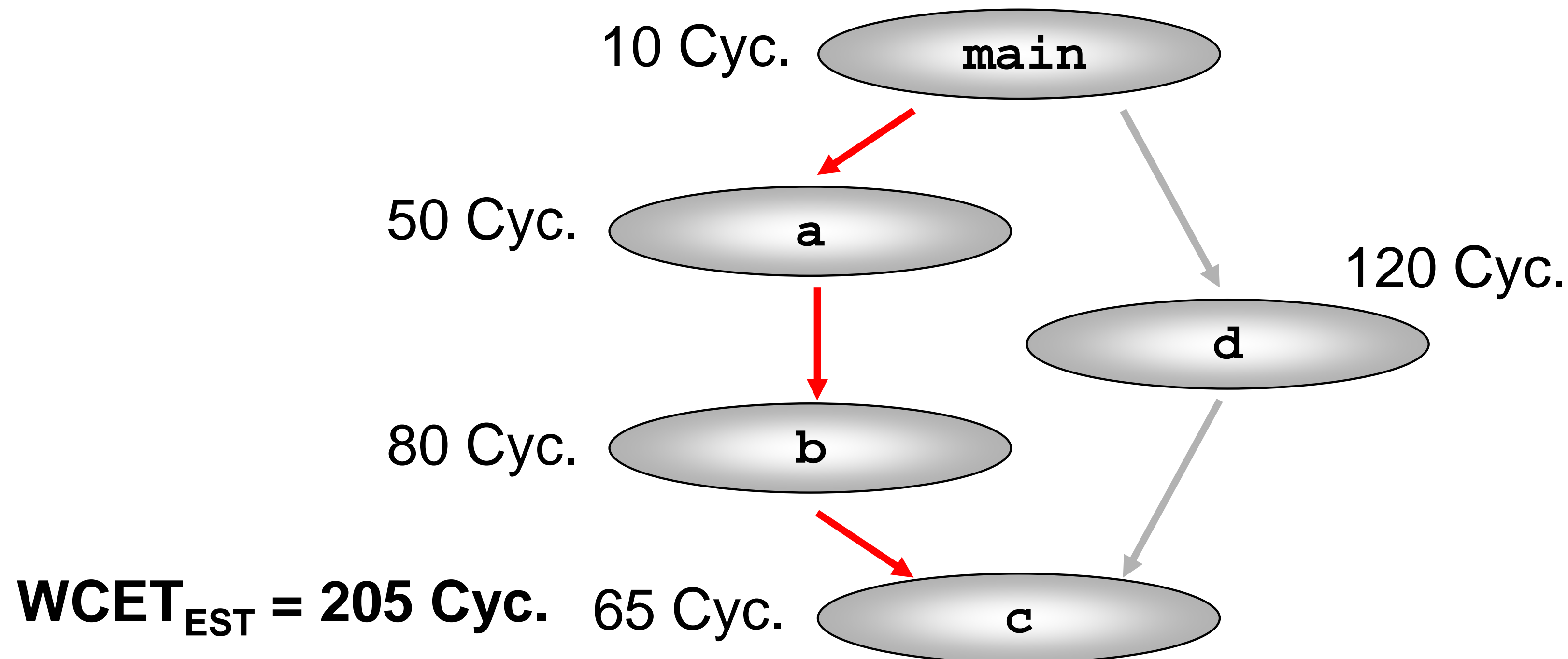
☞ ***WCET-Analyzer aiT liefert diese Detail-Informationen in Form von Ausführungshäufigkeiten von CFG-Kanten, aber...***

Instabilität des WCEPs (1)



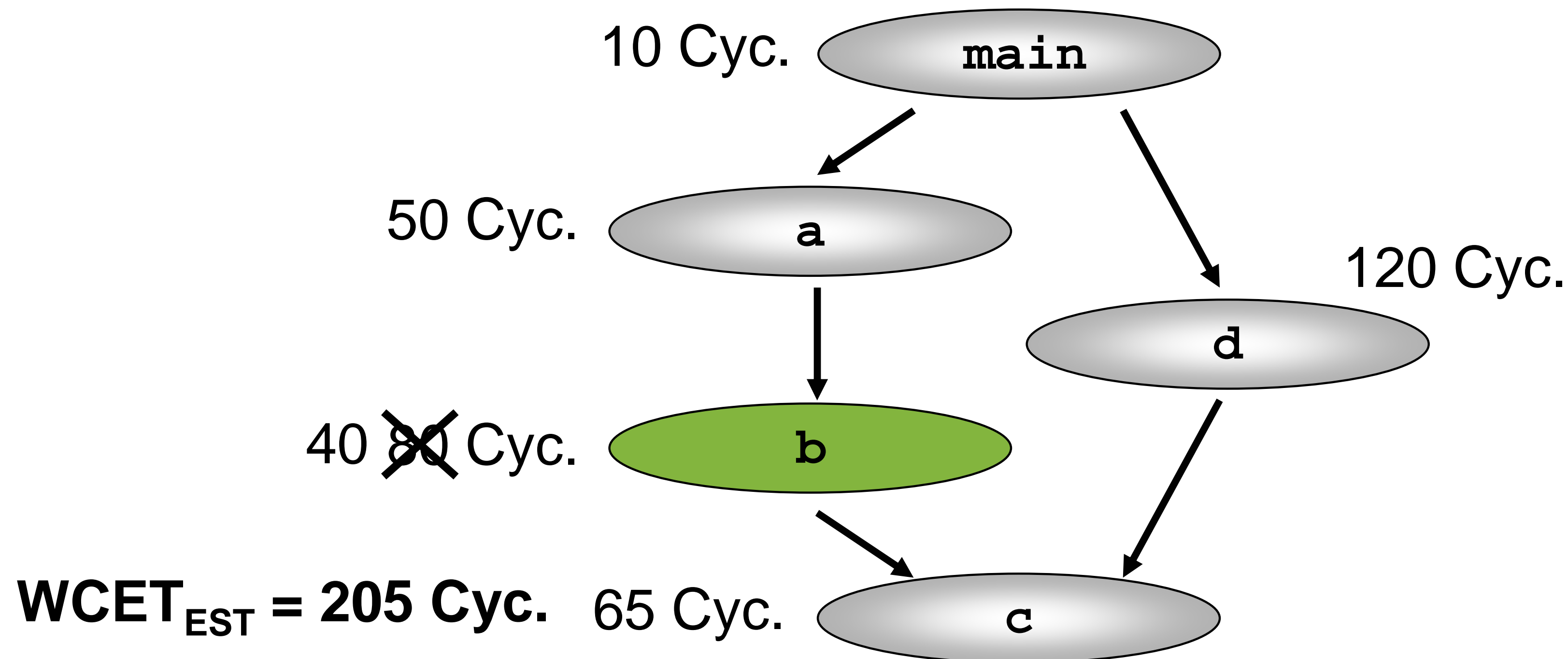
- Beispiel: Einfacher CFG mit 5 Basisblöcken

Instabilität des WCEPs (2)



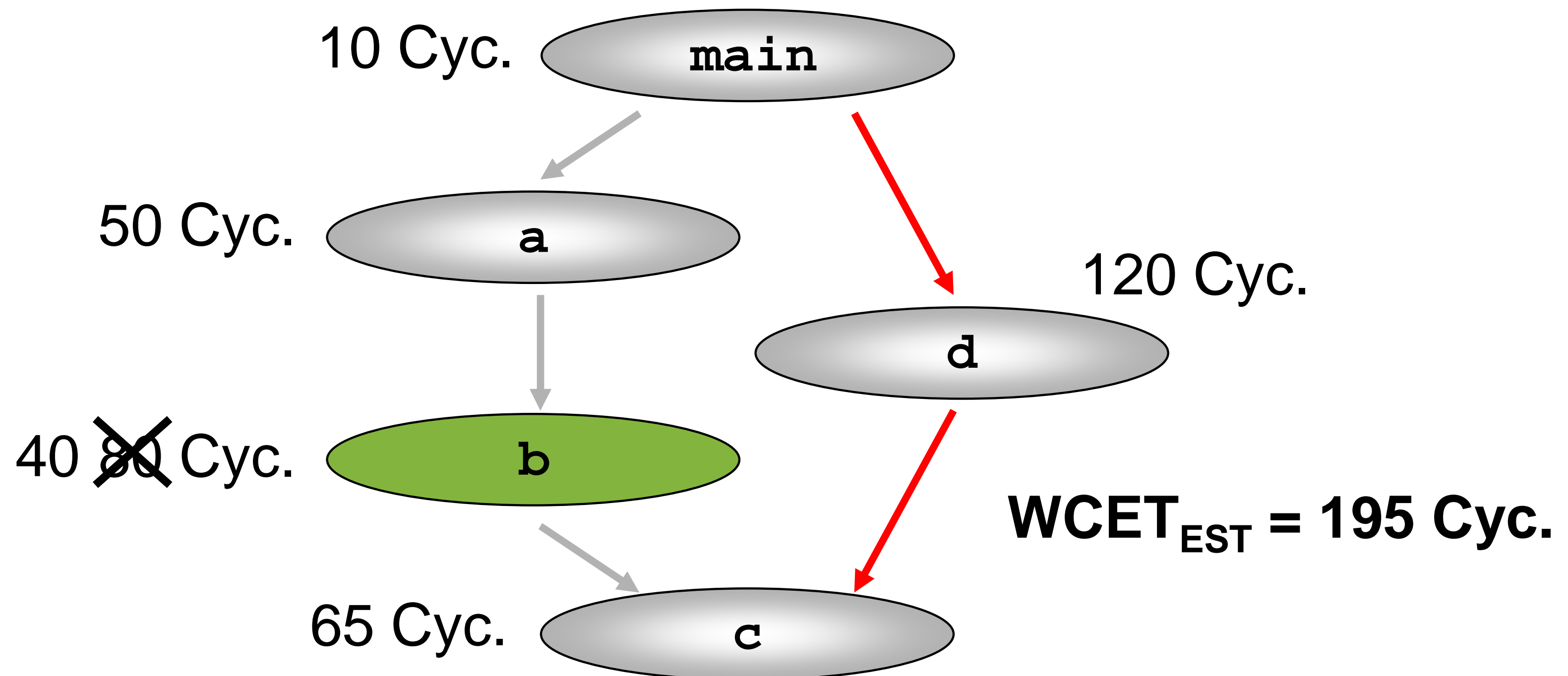
- Initialer WCEP: **`main`**, **`a`**, **`b`**, **`c`**
- WCEP-Länge = $WCET_{EST}$: 205
- Im folgenden: Optimierung von **`b`** reduziert $WCET_{EST}$ von **`b`** von 80 auf 40

Instabilität des WCEPs (3)



- Initialer WCEP: **main**, **a**, **b**, **c**
- WCEP-Länge = $WCET_{EST}$: 205
- Im folgenden: Optimierung von **b** reduziert $WCET_{EST}$ von **b** von 80 auf 40

Instabilität des WCEPs (4)



- Neuer WCEP: `main`, `d`, `c`
- Neue $WCET_{EST}$: 195
- ☞ ***WCEP hat durch Optimierung gewechselt!***

Konsequenzen für Optimierungen

Optimierungen zur $WCET_{EST}$-Minimierung...

- ...müssen immer berücksichtigen, daß sich der WCEP nach jeder Entscheidung, die eine Optimierung trifft, ändern kann.
- ...sollten ihre Entscheidungen, wo was zu optimieren ist, nicht nur aufgrund lokaler Informationen treffen, sondern sollten stets die globalen Auswirkungen einer Entscheidung berücksichtigen.
(Die Optimierung von **b** *im vorherigen Beispiel führt lokal zu einer Reduktion der $WCET_{EST}$ von* **b** *um 40 Zyklen. Global werden aber nur 10 Zyklen eingespart!)*

☞ ***Herausforderung: Entwurf neuartiger Optimierungen für den WCC, die obigen Anforderungen gerecht werden und dabei stets den ganzen CFG mit dem jeweiligen WCEP betrachten.***

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: Instruktionsauwahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- **Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung**
  - Einführung
  - Procedure Cloning & Positioning
  - Register-Allokation
  - Scratchpad-Allokation von Daten und Code
- Kapitel 9: Ausblick

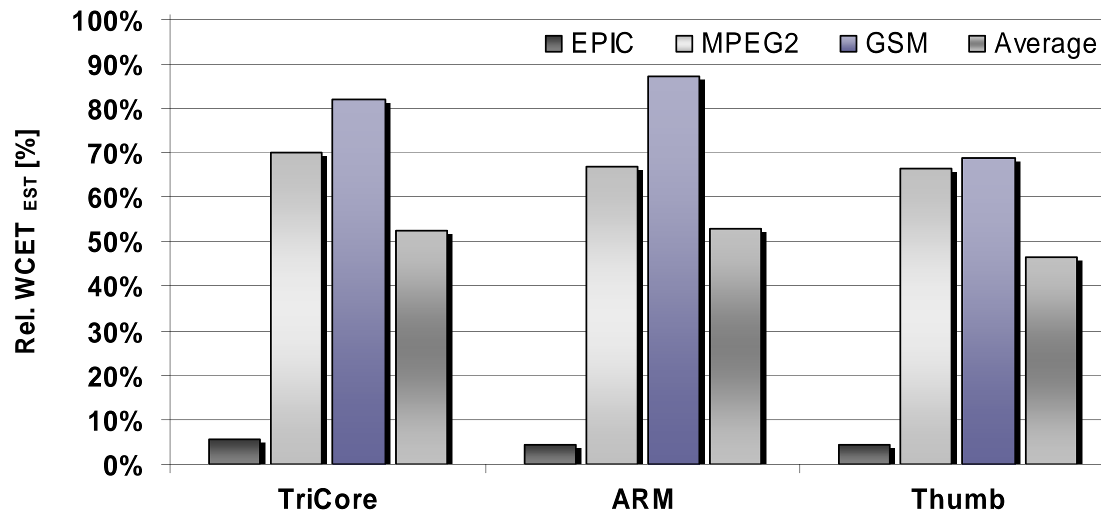
Wieso Procedure Cloning und $WCET_{EST}$?

Motivation: *(☞ siehe Kapitel 4)*

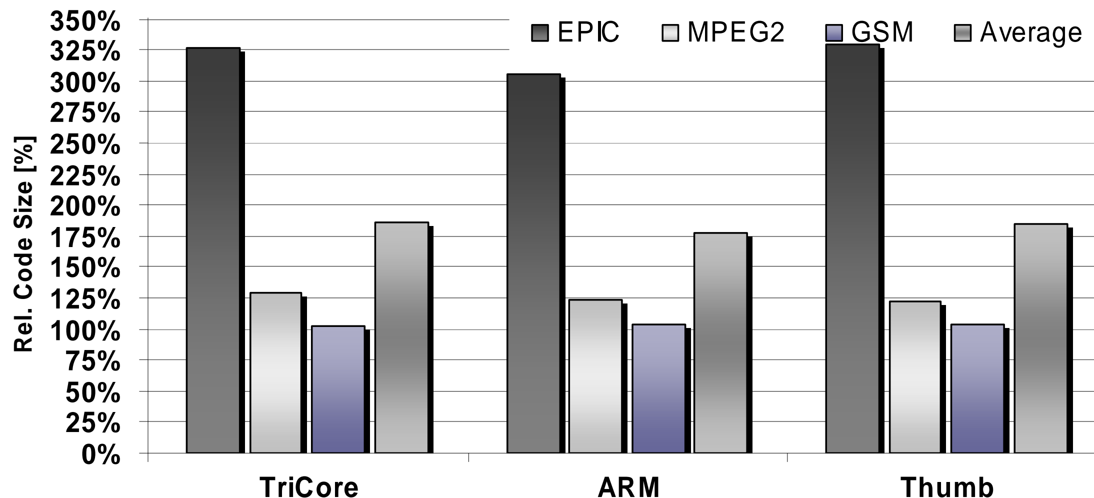
- Häufiges Vorkommen von General Purpose-Funktionen in Special Purpose-Kontexten in Eingebetteter Software
- Gerade Schleifengrenzen werden sehr oft durch Funktions-parameter gesteuert.
- Gerade Schleifengrenzen sind kritisch für eine präzise WCET-Analyse.
- Procedure Cloning ermöglicht hochgradig präzise Annotation von Schleifengrenzen.

*[**P. Lokuciejewski,** Influence of Procedure Cloning on WCET Prediction, Salzburg, CODES+ISSS 2007]*

Ergebnisse nach Standard-Cloning



- $WCET_{EST}$-Verbesserungen von 13% bis zu 95%!



- Code-Vergrößerungen von 2% bis zu 325%!

Hauptprobleme des Standard-Cloning

- $WCET_{EST}$ eines Programms entspricht Länge des WCEP
- Standard-Optimierung “Procedure Cloning” hat keine Kenntnis des WCEP
- Eigenschaften von Funktionen, die $WCET_{EST}$-Reduktion ermöglichen (parameterabhängige Schleifen) werden durch Standard-Optimierung nicht berücksichtigt

☞ Evtl. Cloning von Funktionen, die nicht auf WCEP liegen

☞ Evtl. Cloning von Funktionen, die nicht zur Verbesserung der $WCET_{EST}$ beitragen

☞ Unnötige Code-Vergrößerung ohne irgend eine $WCET_{EST}$-Reduktion

WCET-bewußtes Cloning (1)

Gegeben:

- Zu optimierendes Programm *P*, gegeben in einer HIR
- Float-Zahl *maxFactor*, die max. Code-Vergrößerung angibt

Initialisierung:

- *maxCodeSize* = getCodeSize(*P*) * *maxFactor*;

Phase 1 – Bestimmung des WCEP:

- Führe WCET-Analyse von *P* durch;
- Bestimme Menge *F* aller originalen Funktionen auf dem WCEP von P;
- $wcet_{orig}$ = getWCET(*P*);
- cs_{orig} = getCodeSize(*P*);

WCET-bewußtes Cloning (2)

Phase 2 – Ermittlung von WCET-Daten für Funktionen:

- for (*<alle Funktionen f* ∈ *F>*)
  - if (*<f wird mit Konstanten als Parameter p aufgerufen>* &&
    (*<p dient als Schleifengrenze>* ||
    <p wird in Bedingung von if-Statement genutzt> ||
    <p ist Argument in Funktionsaufruf innerhalb von f>))
    // Cloning von f u.U. vorteilhaft bzgl. $WCET_{EST}$
    - HIR *P'* = *P*.copy();
    - doCloning(*P'*, *f*); *// Führe probeweises Cloning aus*
    - updateLoopBounds(*P'*, *f*); deleteRedundantIfStmts(*P'*, *f*);
    - Führe WCET-Analyse von *P'* durch;
    - $wcet_f$ = getWCET(*P'*); cs_f = getCodeSize(*P'*);

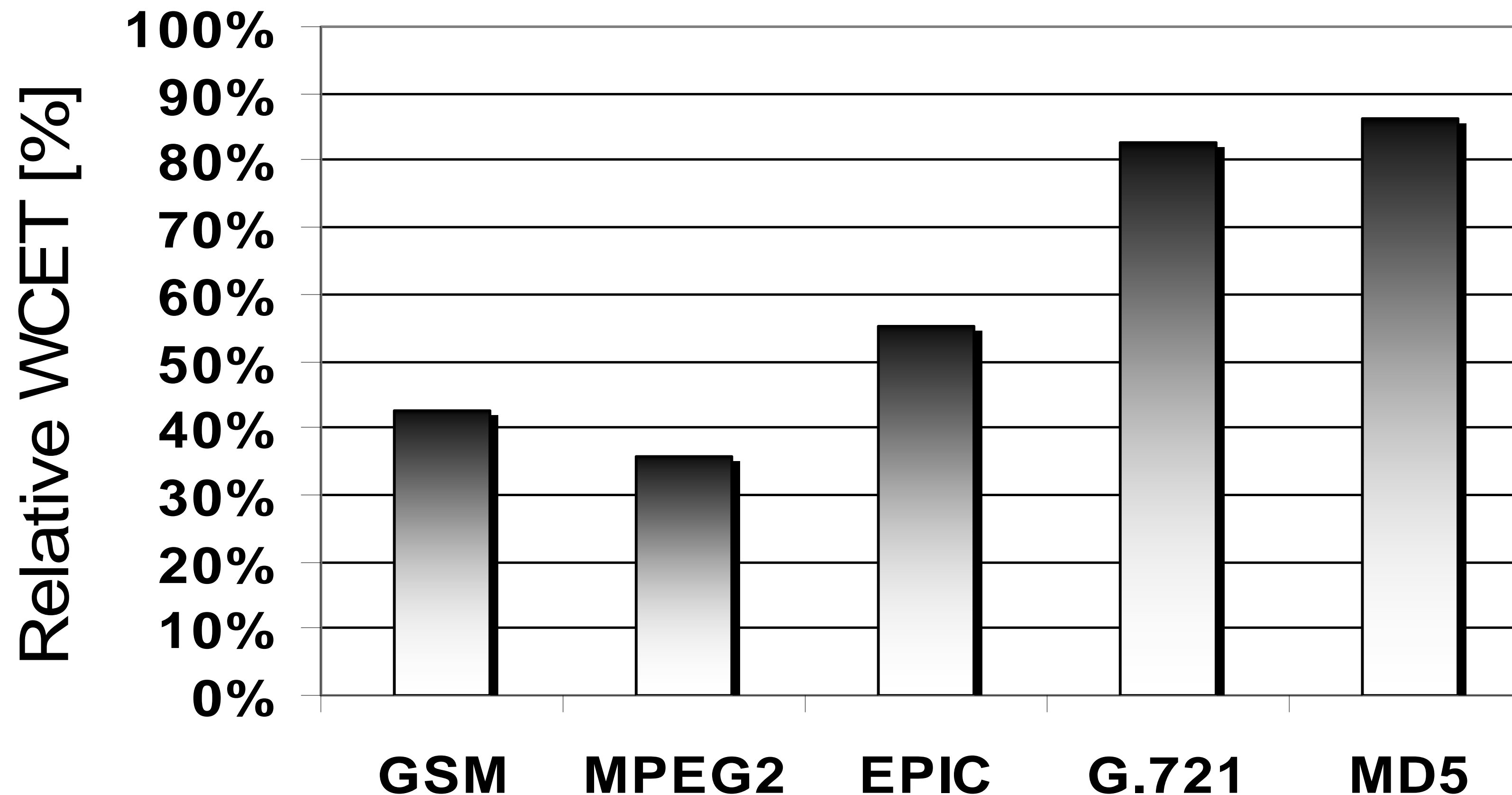
WCET-bewußtes Cloning (3)

Phase 3 – Ermittlung der Funktion mit größtem Profit:

- for (<*alle Funktionen f* ∈ *F*>)
  - $profit_f = (wcet_{orig} - wcet_f) / (cs_f - cs_{orig})$;
- Bestimme Funktion f_{opt} mit maximalem $profit_f$ UND cs_f <= *maxCodeSize*;
- if (<f_{opt} existiert>)
  - doCloning(*P*, f_{opt});
  - Gehe zu Phase 1;

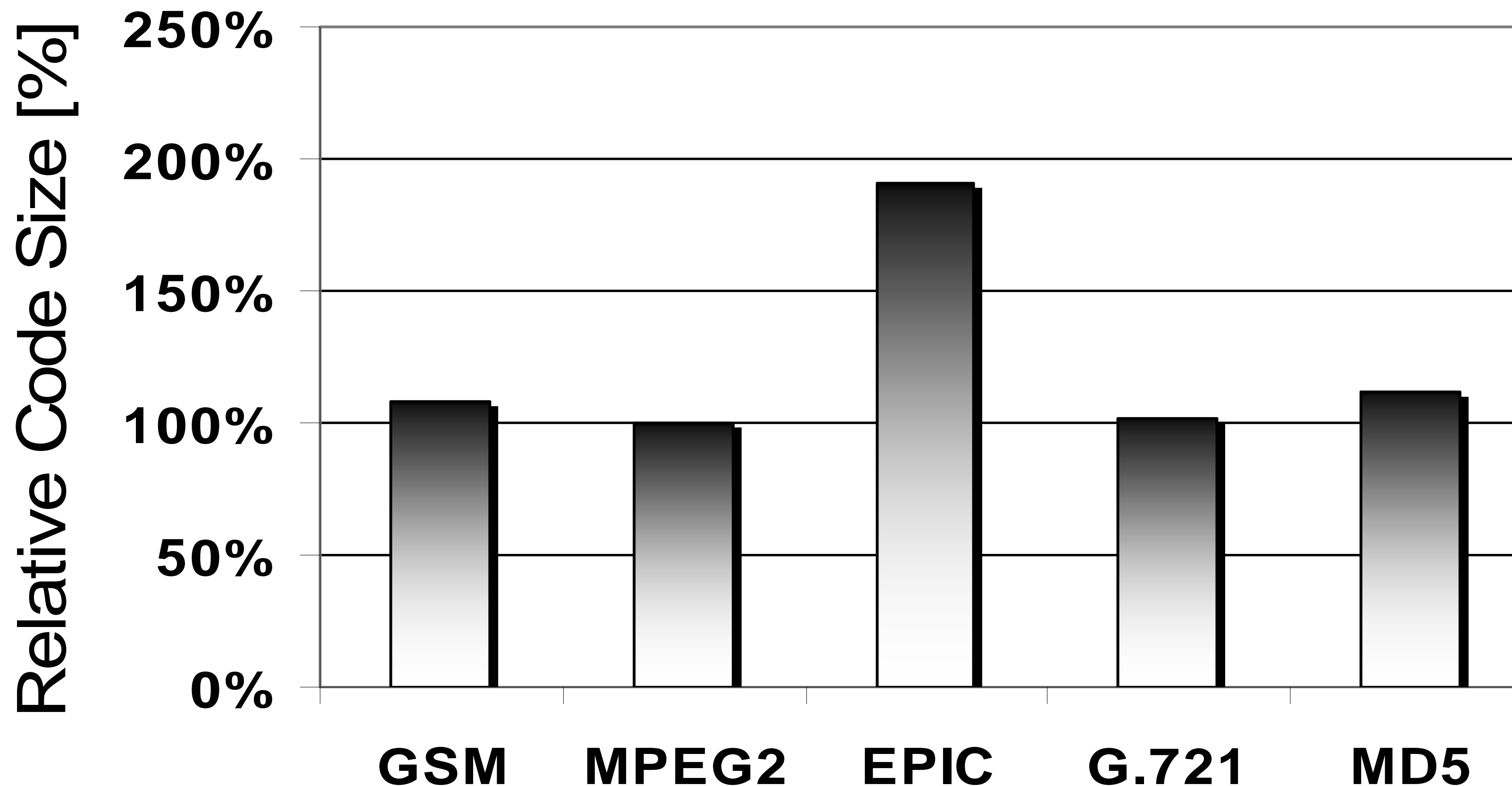
*[**P. Lokuciejewski et al.,** WCET-Driven, Code-Size Critical Procedure Cloning, München, SCOPES 2008]*

Relative $WCET_{EST}$ nach WCET-Cloning



- 100% = $WCET_{EST}$ ohne Procedure Cloning
- $WCET_{EST}$-Verbesserungen von 14% bis zu 64%!

Relative Codegrößen nach WCET-Cloning



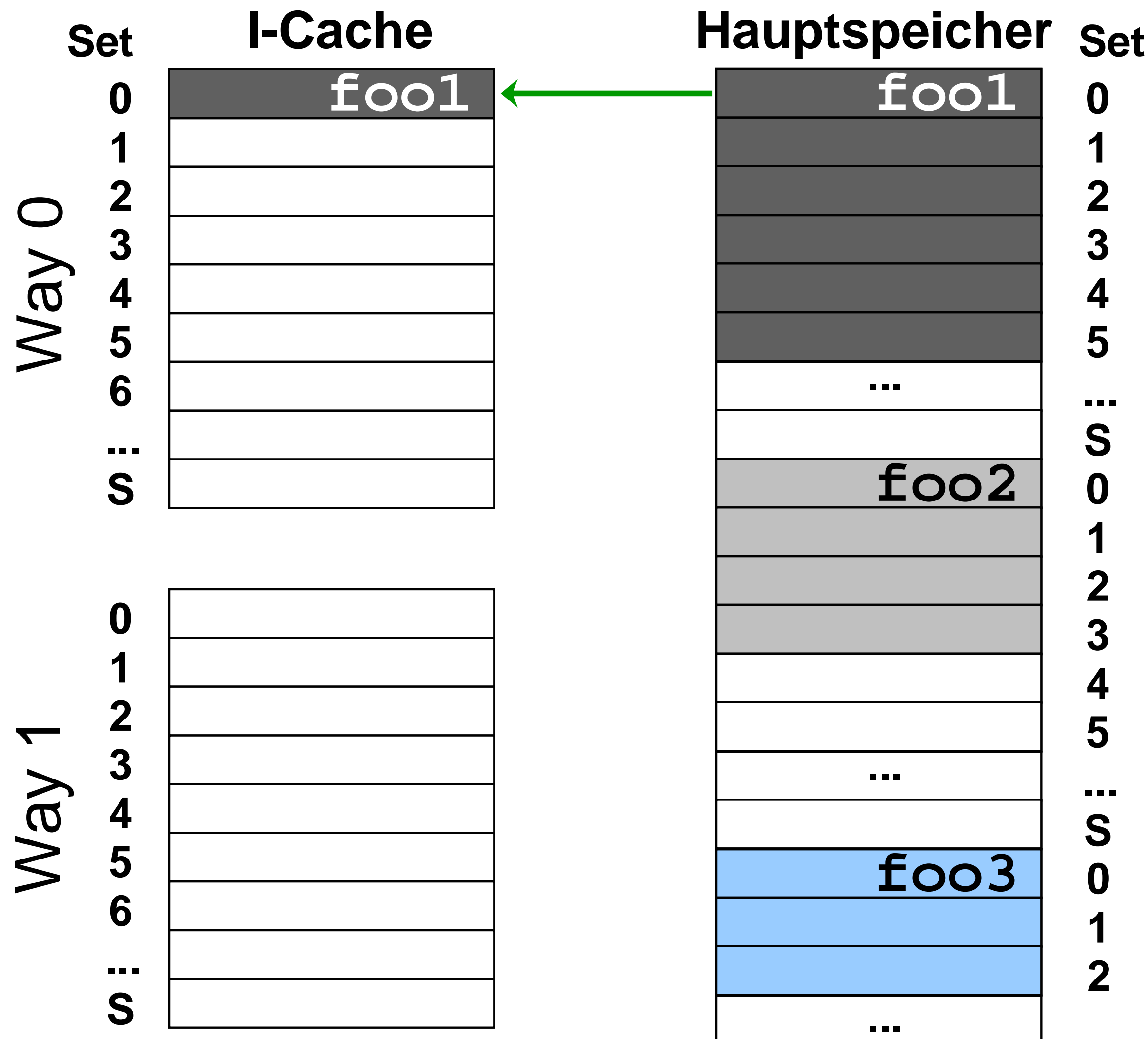
- 100% = Codegröße ohne Procedure Cloning
- Code-Vergrößerung von EPIC: 190% statt bisher 300%

Verdrängung aus Caches

- Caches nutzen Lokalität von Speicherzugriffen aus:
  - *Räumliche Lokalität:* Speicherzugriffe zielen auf einen räumlich kleinen Speicherbereich, der im Cache vorgehalten werden sollte.
  - *Zeitliche Lokalität:* in einer kurzen Zeitspanne wird oft auf räumlich verstreute Speicherbereiche zugegriffen, so daß diese Bereiche im Cache eingelagert sein sollten.
- Schlechte Anordnung von Code oder Daten im Speicher kann bei zeitlicher Lokalität aber zu schlechtem Cache-Verhalten führen:

☞ Speicherbereiche mit hoher zeitlicher Lokalität können sich bei ungünstiger Anordnung wiederholt gegenseitig aus dem Cache verdrängen, was zu vielen Cache-Misses, sog. *Conflict Misses*, führt.

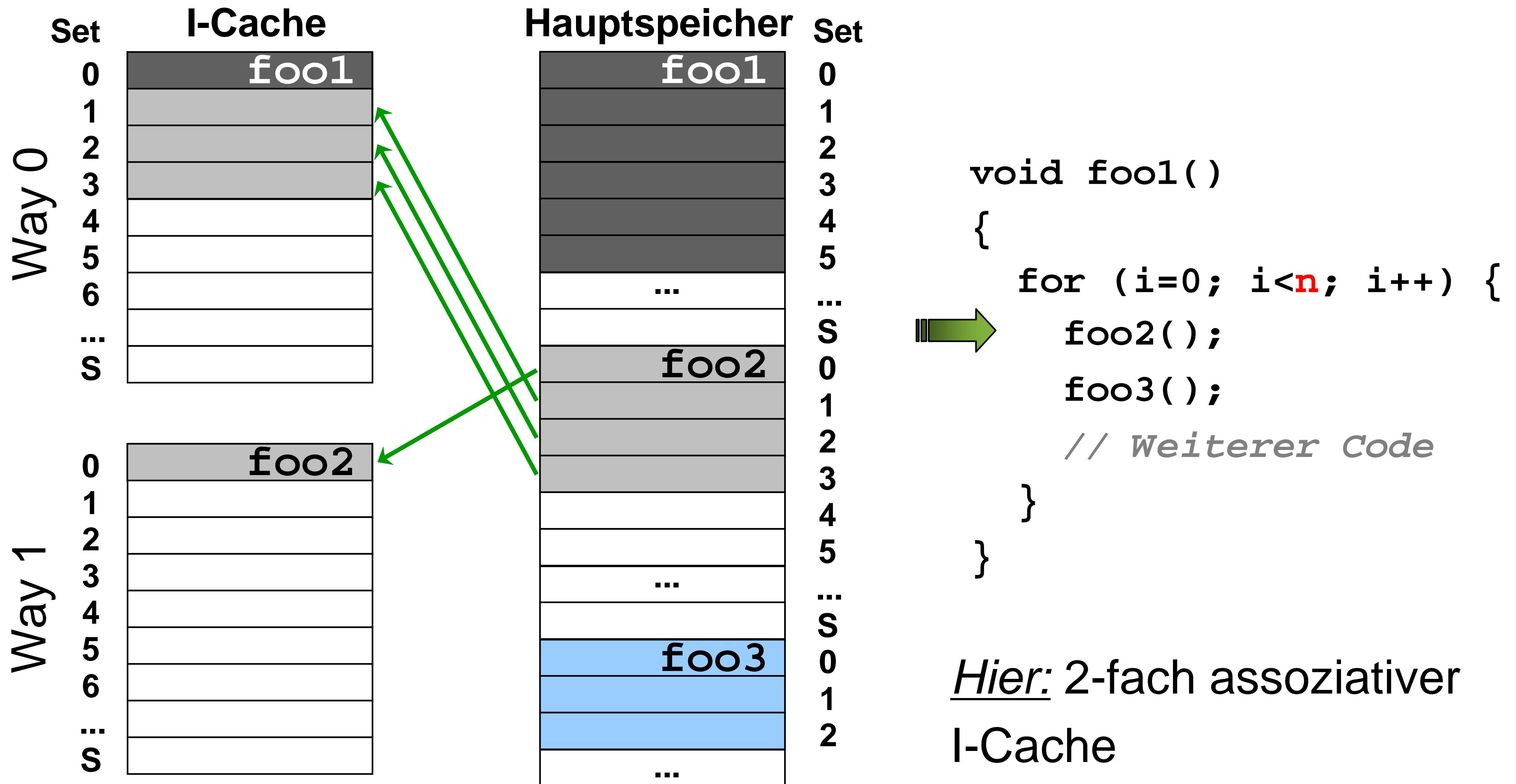
Beispiel für Verdrängung aus Befehls-Cache




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // Weiterer Code
  }
}
```

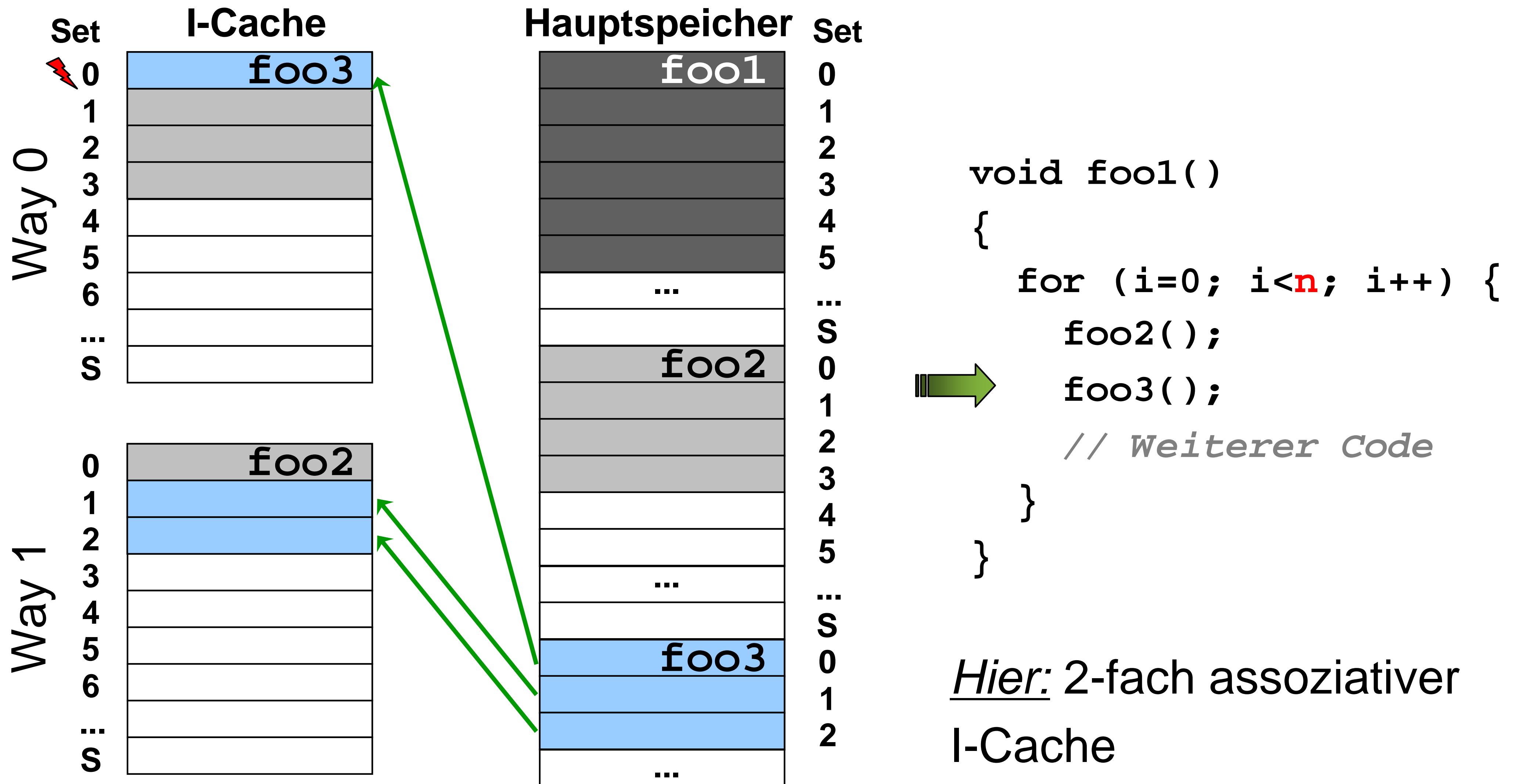
Hier: 2-fach assoziativer I-Cache

Beispiel für Verdrängung aus Befehls-Cache



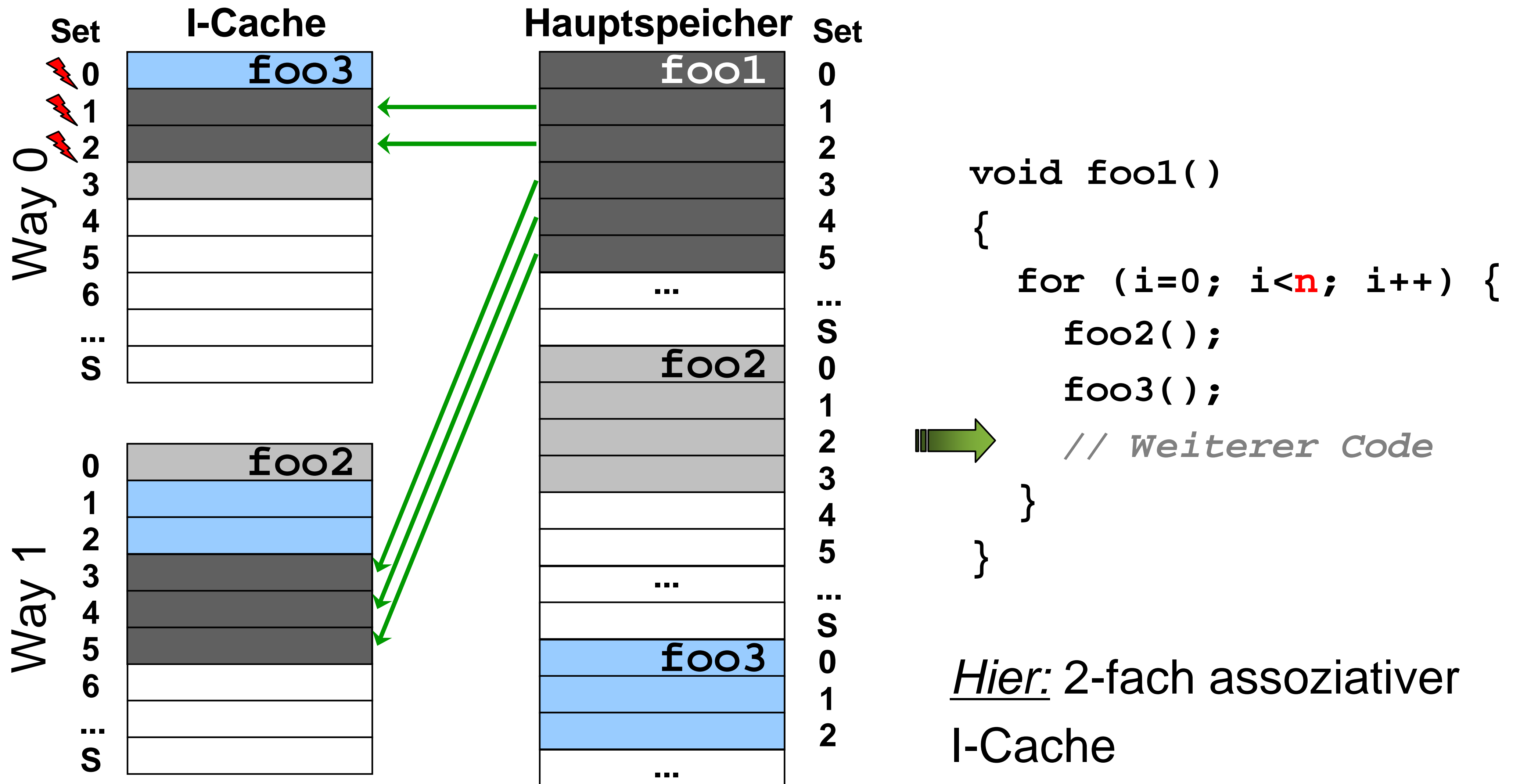
Hier: 2-fach assoziativer I-Cache

Beispiel für Verdrängung aus Befehls-Cache



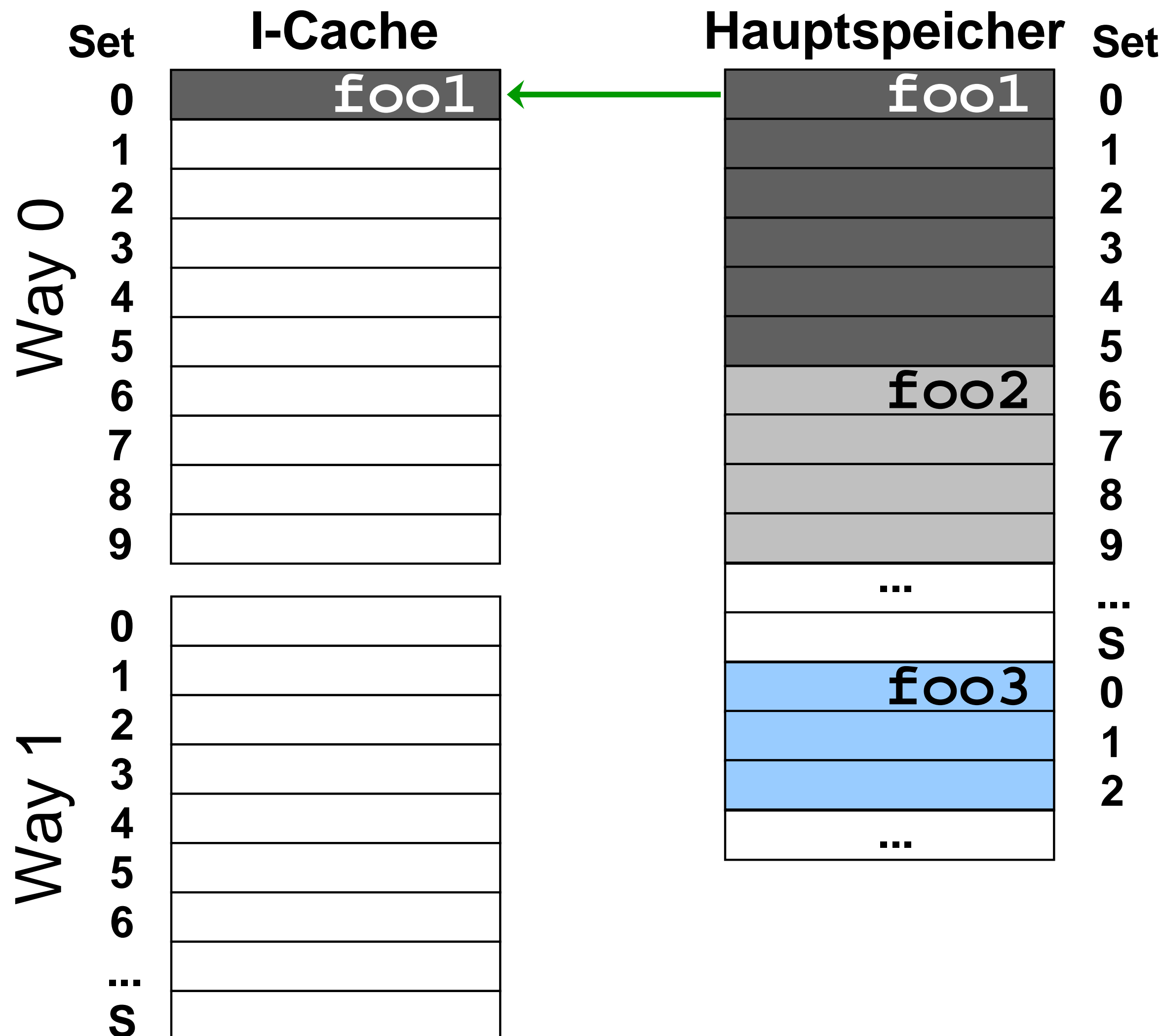
Hier: 2-fach assoziativer I-Cache

Beispiel für Verdrängung aus Befehls-Cache



Hier: 2-fach assoziativer I-Cache

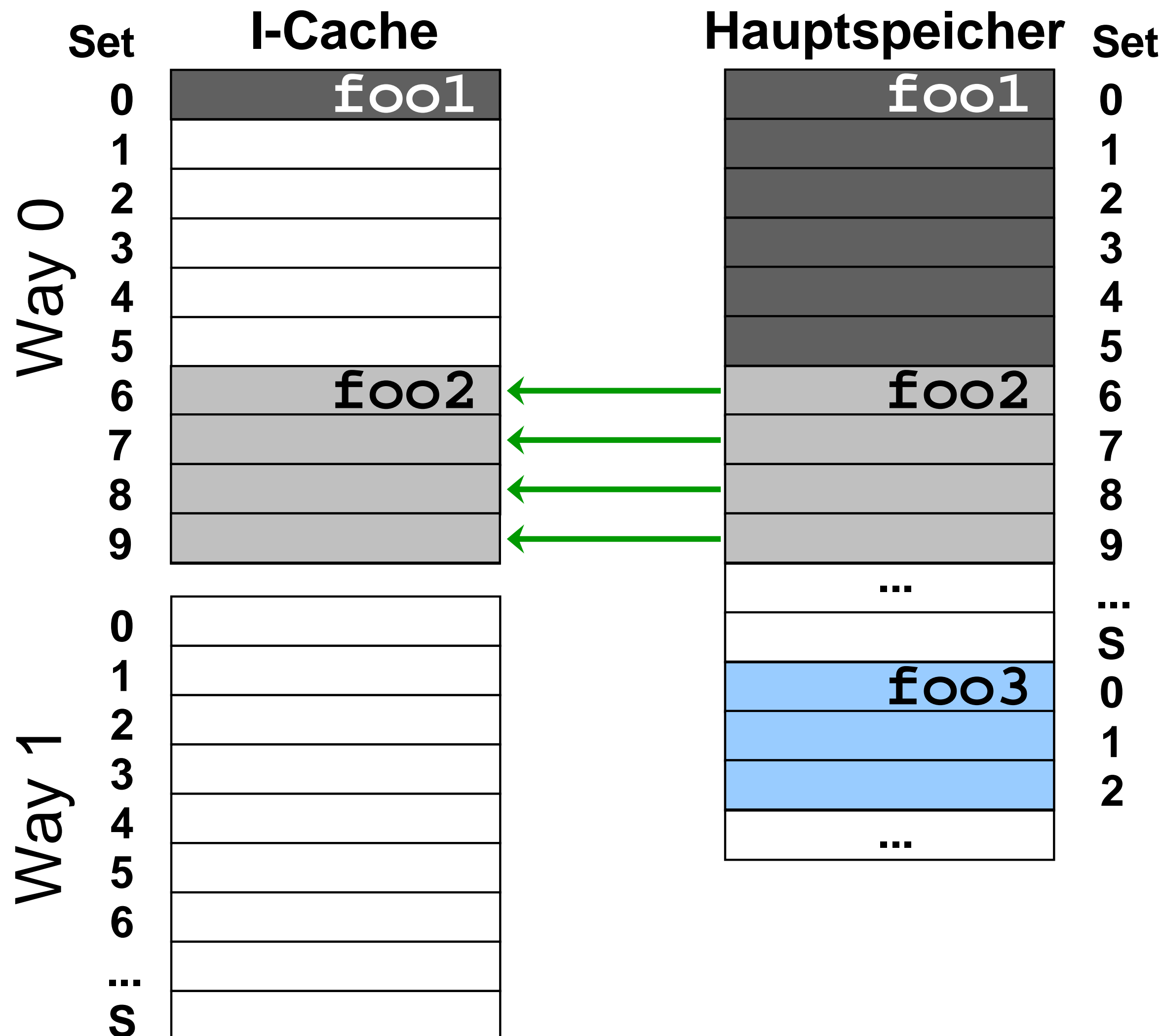
Keine Verdrängung bei besserer Anordnung




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // Weiterer Code
  }
}
```

Hier: 2-fach assoziativer I-Cache

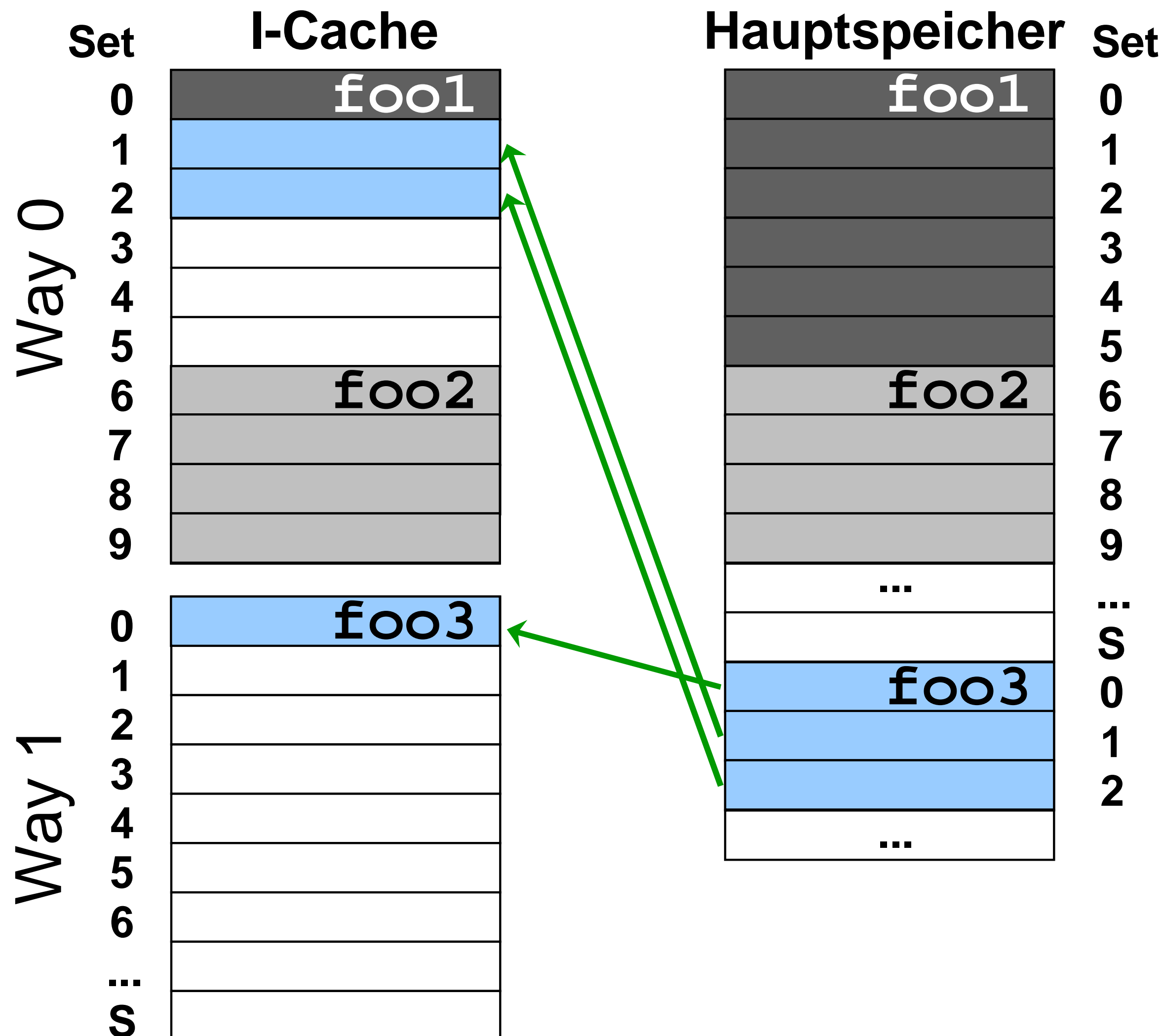
Keine Verdrängung bei besserer Anordnung




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // Weiterer Code
  }
}
```

Hier: 2-fach assoziativer I-Cache

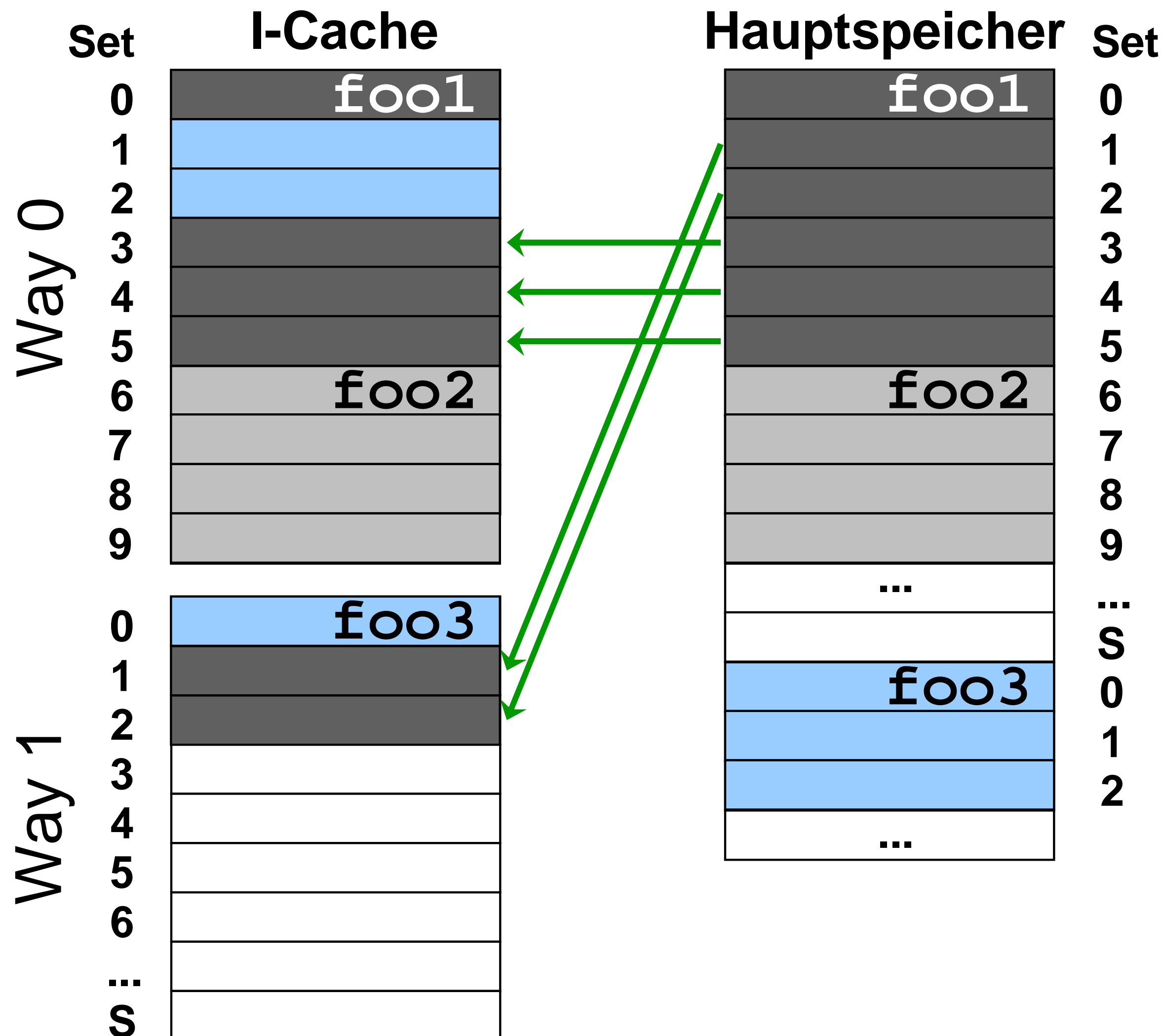
Keine Verdrängung bei besserer Anordnung




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // Weiterer Code
  }
}
```

Hier: 2-fach assoziativer I-Cache

Keine Verdrängung bei besserer Anordnung




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // Weiterer Code
  }
}
```

Hier: 2-fach assoziativer I-Cache

Procedure Positioning mit Call-Graph

Definition *(Call-Graph)*: Der *Call-Graph* ist ein ungerichteter gewichteter Graph $G = (V, E, w)$ mit:

- Knotenmenge V enthält Knoten v pro Funktion eines Programms
- Kantenmenge E enthält Kante $e = (v, w)$, wenn Funktion v eine andere Funktion w aufruft
- Jede Kante $e = (v, w)$ ist mit der Häufigkeit $w(e)$ gewichtet, mit der sich v und w gegenseitig aufrufen.

Idee eines WCET-bewußten Procedure Positioning:

- Erzeuge Call-Graph mit worst-case Aufrufhäufigkeiten gemäß statischer WCET-Analyse als Kantengewichten
- Plaziere je zwei Funktionen mit hohem Kantengewicht unmittelbar nebeneinander

WCET-bewußtes Procedure Positioning (1)

Gegeben:

- Zu optimierendes Programm *P*, gegeben in einer LIR

Initialisierung:

- Führe WCET-Analyse von *P* durch;
- Erzeuge Call-Graph $G_{orig} = (V_{orig}, E_{orig}, w_{orig})$ für *P* anhand von WCET-Daten;
- Erzeuge Call-Graph $G_{new} = (V_{new}, E_{new}, w_{new})$ als Kopie von G_{orig};

*[**P. Lokuciejewski et al.,** WCET-driven Cache-based Procedure Positioning Optimizations, Prag, ECRTS 2008]*

WCET-bewußtes Procedure Positioning (2)

Optimierungsschleife:

- do
  - $wcet_{current}$ = getWCET(*P*);
  - for (<alle Kanten $e = (v, w) \in E_{new}$,
    absteigend nach w_{new} sortiert>)
    - if (Positioning(*e*, G_{new}, G_{orig}, *P*, $wcet_{current}$) == true)
      // Wenn Plazierung von Knoten v und w nebeneinander
      // zu WCET-Reduktion führt, breche for-Schleife ab, fahre
      // mit do-while-Schleife fort.
      break;
- while (<*P* wurde in letzter Iteration modifiziert>);

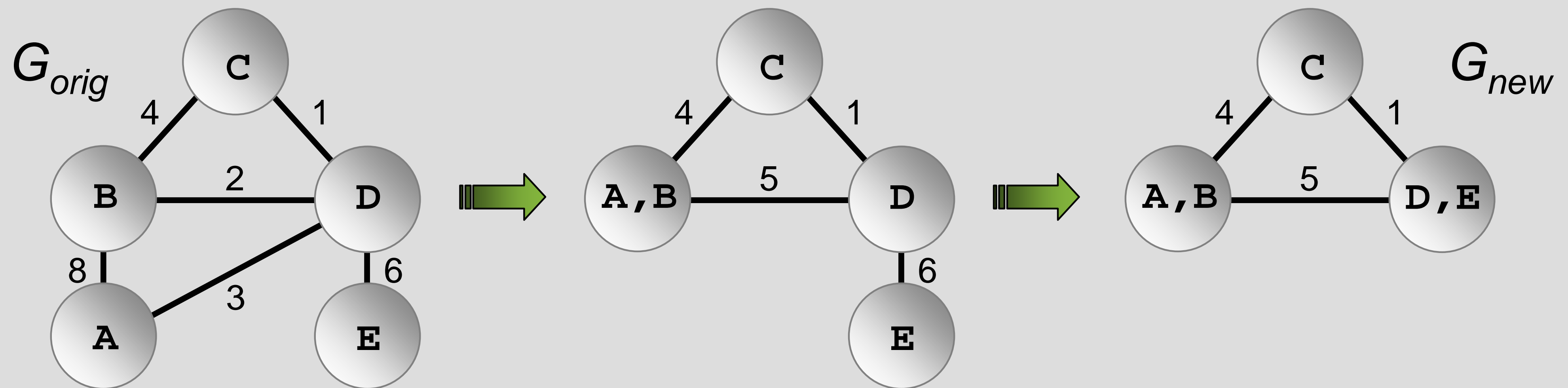
WCET-bewußtes Procedure Positioning (3)

Positioning($e = (v, w) \in E_{new}, G_{new}, G_{orig}, P, wcet_{current}$):

- Erzeuge LIR *P'*, so daß *v* und *w* nacheinander im Speicher plaziert sind;
- Führe WCET-Analyse von *P'* durch;
- $wcet_{new}$ = getWCET(*P'*);
- if ($wcet_{new} < wcet_{current}$)
  - *P* = *P'*;
  - verschmelze Knoten *v* und *w* in G_{new};
  - Aktualisiere w_{new} gemäß neuer WCET-Daten;
  - return true;
- else
  - return false;

Verschmelzen von Knoten

Verschmelzen von Knoten:



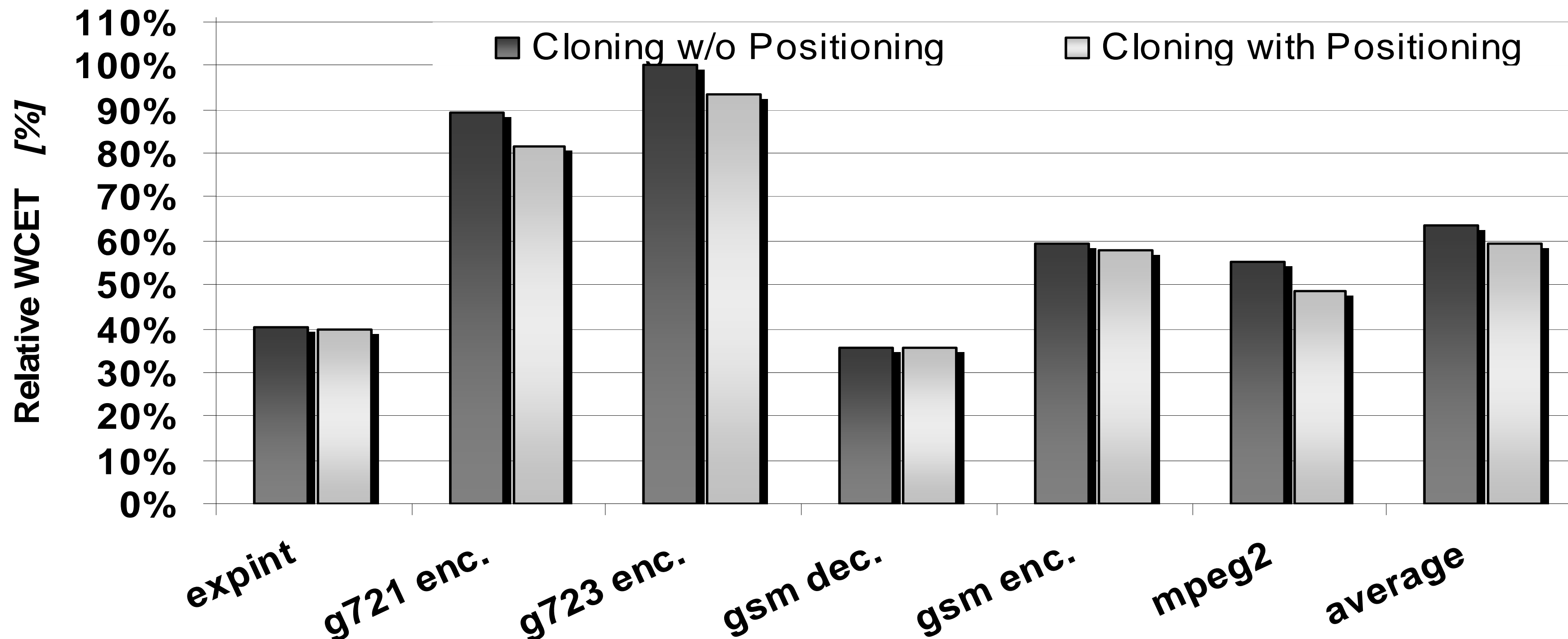
Plazierung verschmolzener Knoten nebeneinander:

- Problem: Wie sind als nächstes (**A**,**B**) und (**D**,**E**) zu plazieren?
- G_{orig} zeigt, daß **A** und **D** nebeneinander liegen sollten
- ☞ Beste Plazierung ist (**B**,**A**,**D**,**E**).
- ☞ Aus diesem Grund braucht Positioning-Algorithmus G_{orig}!

Eigenschaften

- Algorithmus geht greedy vor und plaziert jeweils zwei Knoten des aktuellen Graphen G_{new} in einer Iteration nebeneinander.
- Hierbei werden stets die beiden Knoten betrachtet, die sich gemäß w_{new} am häufigsten gegenseitig aufrufen.
- Instabile WCEPs werden vom WCET-Positioning betrachtet, da für jede Plazierung eine eigene WCET-Analyse durchgeführt und die Kantengewichte w_{new} anhand dieser neuen WCET-Daten aktualisiert werden.
- Da WCET-bewußtes Procedure Cloning die neuen Clones stets nur an das Ende des Programms P hängt, ist es sehr sinnvoll, WCET-Cloning und WCET-Positioning miteinander zu kombinieren.

Relative $WCET_{EST}$ nach WCET-Cloning & Positioning



- 100% = $WCET_{EST}$ ohne Procedure Cloning und Positioning
- I-Cache: 16kB, 2-fach mengenassoziativ, LRU-Ersetzung
- WCET-Positioning der Clone: zusätzliche $WCET_{EST}$-Reduktion um bis zu 7% gegenüber Cloning

Vorsicht: *Dieses Diagramm nicht mit Folie 29 vergleichen, da in Folie 29 I-Cache deaktiviert!*

Zwischenfazit

Abgleich mit Konsequenzen für $\mathbf{WCET_{EST}}$-Optimierungen:

Optimierungsstrategien zur $WCET_{EST}$-Minimierung...

- ...brauchen zwingend Detailwissen über den WCEP.
- ✓ *WCET-Cloning und WCET-Positioning berücksichtigen WCEP.*
- ...müssen immer berücksichtigen, daß sich der WCEP nach jeder Entscheidung, die eine Optimierung trifft, ändern kann.
- ✓ *Beide aktualisieren WCEP nach jeder Codeveränderung.*
- ...sollten ihre Entscheidungen, wo was zu optimieren ist, nicht nur aufgrund lokaler Informationen treffen, sondern sollten stets die globalen Auswirkungen einer Entscheidung berücksichtigen.
- ✗ *WCET-Cloning und WCET-Positioning sind Greedy-Heuristiken, die nur lokale Daten pro Funktion betrachten.*

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: Instruktionsauwahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- **Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung**
  - Einführung
  - Procedure Cloning & Positioning
  - Register-Allokation
  - Scratchpad-Allokation von Daten und Code
- Kapitel 9: Ausblick

Register-Allokation per Graphfärbung

1. **Initialisierung:** Erzeuge Interferenzgraph $G = (V, E)$ mit
   V = {*virtuelle Register*} ∪ {*K physikalische Prozessor-Register*},
   $e = (v, w) \in E \Leftrightarrow$ VREGs v und w sollen nie das selbe PHREG teilen, i. e. v und w interferieren
2. **Vereinfachung:** Entferne alle Knoten $v \in V$ mit Grad $< K$
3. **Spilling:** Nach Schritt 2 hat jeder Knoten von G Grad $\geq K$. Wähle ein $v \in V$; markiere v als *potenziellen Spill*; entferne v aus G
4. **Wiederhole** Schritte 2 und 3 bis $G = \emptyset$
5. **Färbung:** Füge Knoten v in umgekehrter Folge wieder in G ein; gibt es freie Farbe k_v, färbe v; sonst markiere v als *echten Spill*
6. **Füge Spill-Code** vor/nach echten Spills ein; gehe zu 1 falls #VREGs > 0

*[**A. W. Appel,** Modern compiler implementation in C, 1998]*

Problem der Standard-Graphfärbung

3. **Spilling:** Nach Schritt 2 hat jeder Knoten von *G* Grad ≥ *K*. Wähle ein $v \in V$; markiere *v* als *potenziellen Spill*; entferne *v* aus *G*

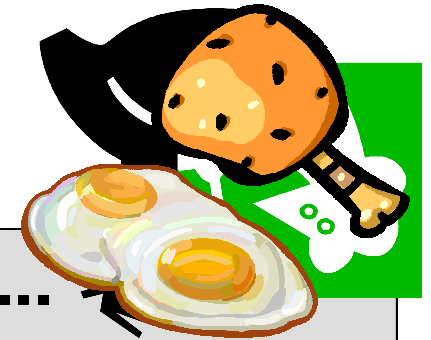
Welchen Knoten v als potenziellen Spill wählen?

Übliche Implementierungen wählen heuristisch ...

- … einen Knoten gemäß Reihenfolge des Vorkommens in interner Compiler-Zwischendarstellung,
- … den Knoten mit höchstem Grad,
- … einen Knoten mit hohem Grad, mit vielen Verwendungen, in innerster Schleife.

☞ ***Unkontrollierte Erzeugung von Spill-Code – möglicherweise entlang des WCEP, der die WCET bestimmt!***

Ein Henne-Ei-Problem



Eine Register-Allokation zur WCET-Minimierung...

- ...benötigt WCET-Daten von statischer WCET-Analyse,
- ...kann aber keine WCET-Daten erhalten, da Code mit virtuellen Registern nicht analysierbar ist!

Ausweg:

- Jedes VREG wird zu Beginn auf Laufzeit-Stack ausgelagert
  ☞ Code hat schlechte Qualität, ist aber vollständig analysierbar
- Durchführung einer WCET-Analyse, Bestimmung des WCEP *P*
- Wende Standard-Graphfärbung auf alle VREGs des Basisblocks $b \in P$ mit meisten Ausführungen von Spill-Code im worst case an
- Ermittle neuen WCEP

WCET-bewußte Graphfärbung (1)

```
LIR WCET_GC_RA( LIR P )
{
  // Iteriere bis aktueller WCEP komplett allokiert.
  while ( true )
  {
    // Kopiere P, alle VREGs von P′ werden auf Stack gespillt.
    LIR P′ = P.copy();
    P′.spillAllVREGs();

    // Ermittle WCEP für komplett gespillte LIR.
    set<basic_blocks> WCEP = computeWCEP( P′ );

    // Enthält WCEP keine VREGs, breche Allokationsschleife ab.
    if ( getVREGs( WCEP ) == ∅ )
      break;
```

WCET-bewußte Graphfärbung (2)

```
    // Bestimme den Block auf dem WCEP mit dem höchsten Produkt
    // von Worst-Case Ausführungshäufigkeit * Spill-Code.
    basic_block b′ = getMaxSpillCodeBlock( WCEP );
    basic_block b = getBlockOfOriginalP( b′ );

    // Bestimme alle VREGs dieses kritischen Blocks.
    list<virtualRegister> vregs = getVREGs( b );

    // Sortiere VREGs nach #Vorkommen, Standard-Graphfärbung.
    vregs.sort( occurrences of VREG in b );
    traditionalGraphColoring( P, vregs );
  }

  // Wende Standard-Graphfärbung auf alle restlichen VREGs an.
  traditionalGraphColoring( P, getVREGs( P ) );
  return P;
}
```

Eigenschaften (1)

- Algorithmus arbeitet gleichzeitig mit zu allokierender LIR *P* und einer Kopie *P'*, die komplett gespillt ist, um WCET-Analyse zu ermöglichen.
- Register-Allokation geschieht Basisblock-weise entlang des WCEP.
- Nach Allokation eines Basisblocks wird WCEP in *P'* aktualisiert.
- In einer Iteration des Algorithmus: Allokation desjenigen Basis-blocks, der im worst-case zur maximalen Ausführung von Spill-Code führt, d.h. der in *P'* viel Spill-Code enthält und sehr oft ausgeführt wird.

☞ Die VREGs dieses kritischen Basisblocks *b* sollten nach Möglichkeit in PHREGs gehalten werden.

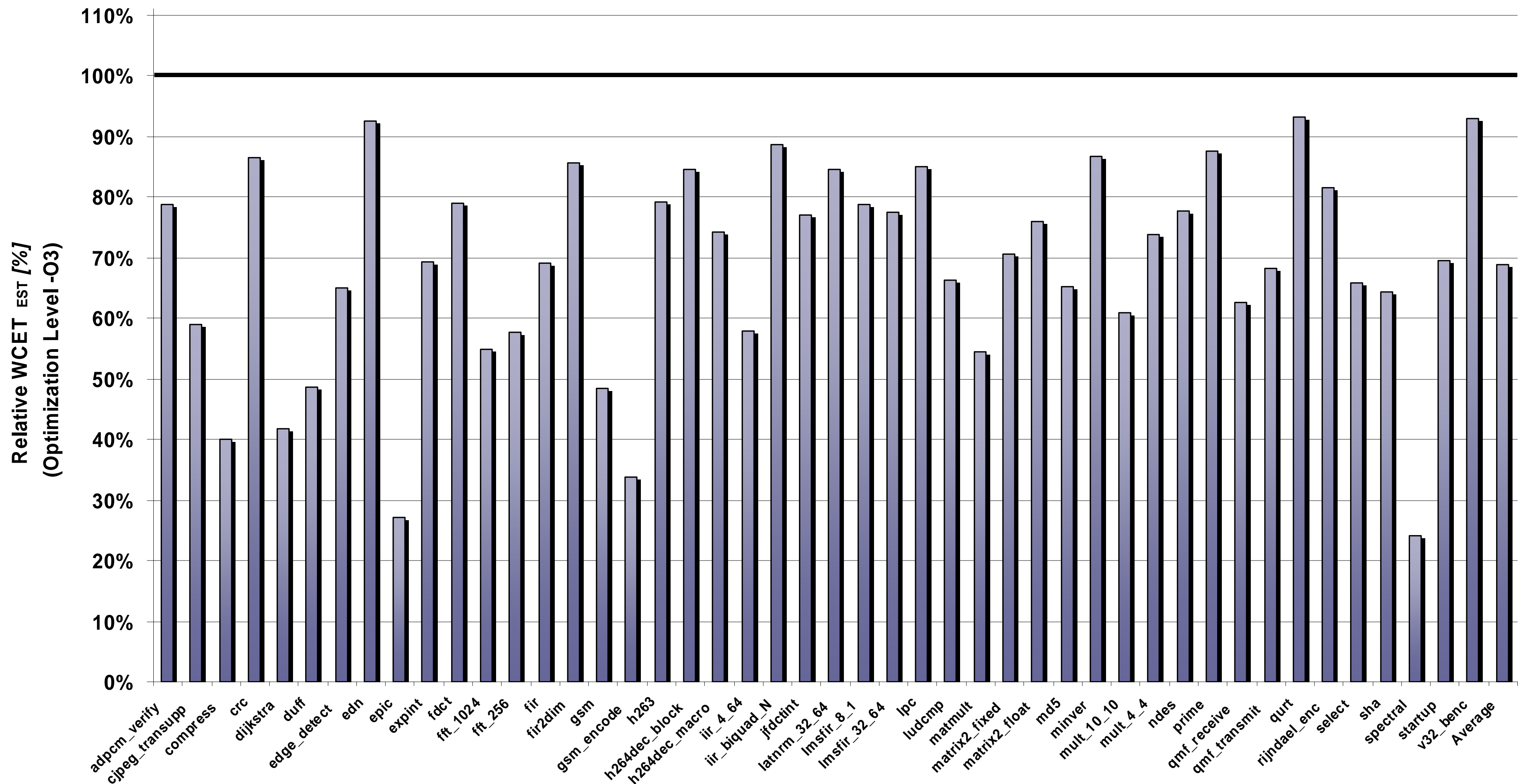
Eigenschaften (2)

- Spilling von VREGs in *b* kann u.U. jedoch nicht vermieden werden. Wenn in *b* zu spillen ist, sollen die VREGs von *b* gespillt werden, die am seltensten in *b* vorkommen, da wenige Vorkommen wenig Spill-Code in *b* bedeuten.
- Register-Allokation und evtl. Spilling von *b* wird durch Standard-Graphfärbung vorgenommen.
- Nach Abbruch der Allokationsschleife ist WCEP komplett allokiert. Es kann aber noch VREGs in Blöcken abseits des WCEP geben.

☞ Abschließende Standard-Graphfärbung, um diese restlichen VREGs zu allokieren.

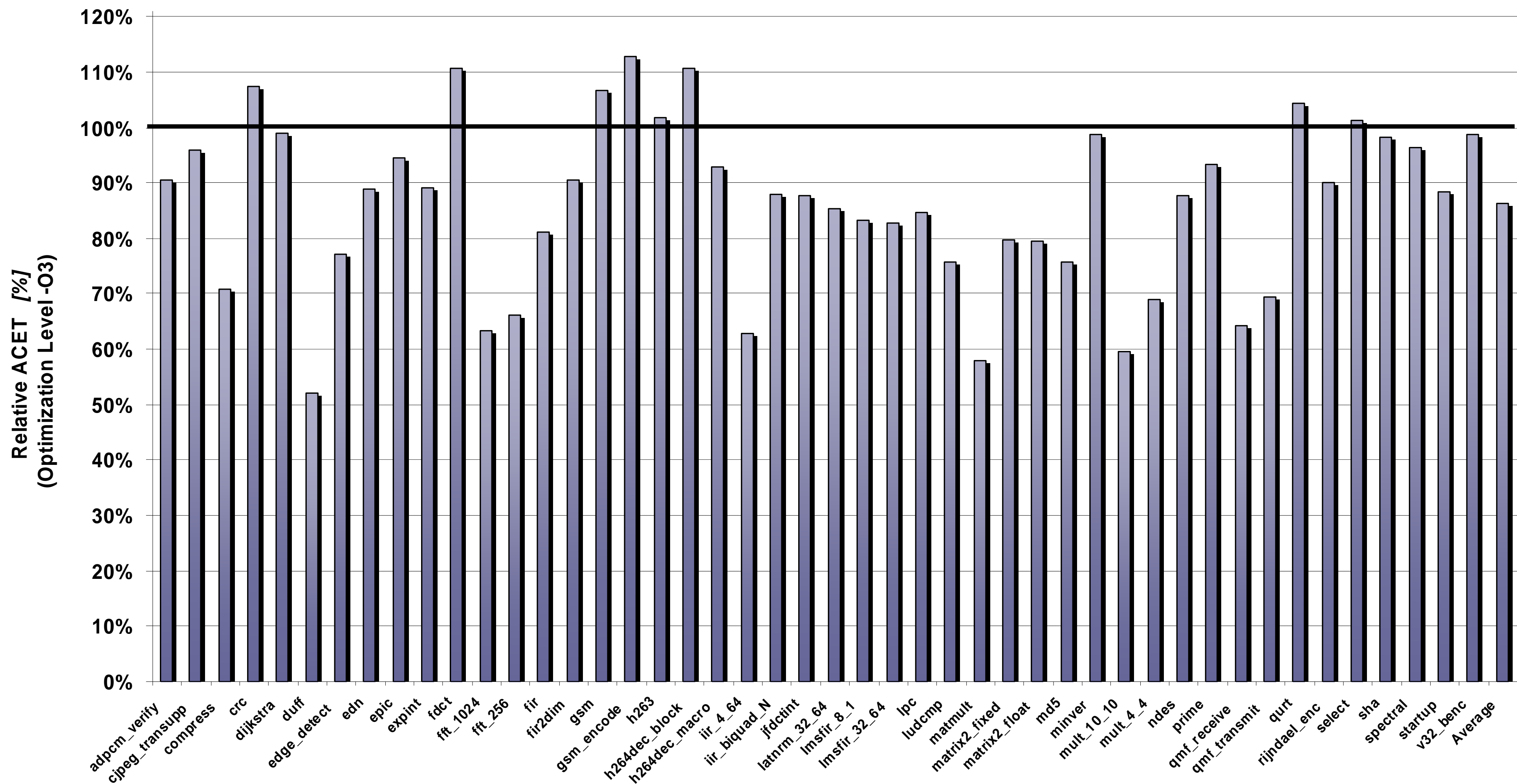
*[**H. Falk,** WCET-aware Register Allocation based on Graph Coloring, San Francisco, DAC 2009]*

Relative $WCET_{EST}$ nach WCET-Graphfärbung



100% = $WCET_{EST}$ mit Standard-Graphfärbung (höchster Grad)

Relative ACET nach WCET-Graphfärbung



100% = ACET mit Standard-Graphfärbung (höchster Grad)

Diskussion

- $WCET_{EST}$-Reduktionen von 6,9% bis 75,9%, im Schnitt 31,2%.
- Allokation aller 46 Benchmarks führt 1.979 WCET-Analysen aus.
- Laufzeit der WCET-Graphfärbung: 12:15 Stunden für alle 46 Benchmarks ☞ 16 Minuten pro Benchmark im Schnitt.
- ACET-Reduktionen von 12,7% Verschlechterung bis 47,9% Verbseerung, im Schnitt 13,8% Verbesserung.
- Benchmarks verhalten sich z.T. sehr unterschiedlich:
  `gsm`-Familie: 51,5% - 66,2% $WCET_{EST}$-Reduktion
  6,8% - 12,7% ACET-Verschlechterung
- *Grund:* WCET-Graphfärbung vermeidet Spill-Code entlang WCEP, fügt aber Spill-Code an anderen Stellen im CFG ein, die in einem average-case Szenario oft ausgeführt werden.

Zwischenfazit

Abgleich mit Konsequenzen für WCET_{EST}-Optimierungen:

Optimierungsstrategien zur WCET_{EST}-Minimierung...

- ...brauchen zwingend Detailwissen über den WCEP.
- ✓ *WCET-Graphfärbung berücksichtigt WCEP.*
- ...müssen immer berücksichtigen, daß sich der WCEP nach jeder Entscheidung, die eine Optimierung trifft, ändern kann.
- ✓ *WCET-Graphfärbung aktualisiert WCEP nach jeder Iteration.*
- ...sollten ihre Entscheidungen, wo was zu optimieren ist, nicht nur aufgrund lokaler Informationen treffen, sondern sollten stets die globalen Auswirkungen einer Entscheidung berücksichtigen.
- ✗ *WCET-Graphfärbung ist Greedy-Heuristik, die nur lokale Daten pro Basisblock betrachtet.*

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: Instruktionsauwahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- **Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung**
  - Einführung
  - Procedure Cloning & Positioning
  - Register-Allokation
  - Scratchpad-Allokation von Daten und Code
- Kapitel 9: Ausblick

ILP zur SPM-Allokation von Daten

Motivation:

- TriCore TC1796 enthält separate Daten- und Code-SPMs
- Getrennte Optimierungen zur SPM-Allokation von Daten und Code

Ziel:

- Bestimme Menge globaler oder lokaler statischer Variablen (skalar und zusammengesetzte Typen) zur Allokation in Daten-SPM,
- so daß ausgewählte Datenobjekte $WCET_{EST}$-Minimierung erzielen.

Ansatz:

- Ganzzahlig-lineare Programmierung → optimale Resultate
- Inhärente Modellierung wechselnder WCEPs im ILP
- Notation: Großbuchstaben ≅ Konstanten,
  Kleinbuchstaben ≅ Variablen

ILP zur SPM-Allokation von Daten

- ***Binäre Entscheidungsvariablen pro Datenobjekt:***

$$y_i = \begin{cases} 1 & \text{if data object } d_i \text{ is assigned to } mem_{spm} \\ 0 & \text{if data object } d_i \text{ is assigned to } mem_{main} \end{cases}$$

- ***Kosten eines Basisblocks b_j:***

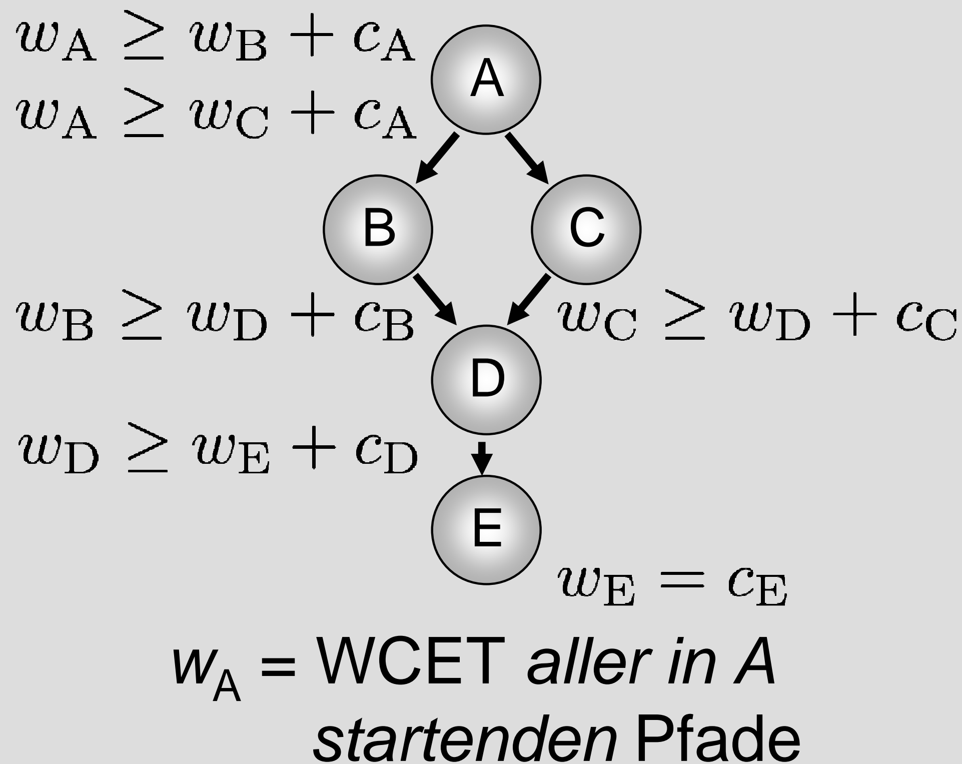
$$c_j = C_j - \sum_{d_i \in \text{data objects}} G_{i,j} * y_i$$

WCET$_{EST}$ von b_j mit allen Datenobjekten im Hauptspeicher, minus Gewinn für b_j, der durch Verschieben von d_i in SPM entsteht.

ILP zur SPM-Allokation von Daten

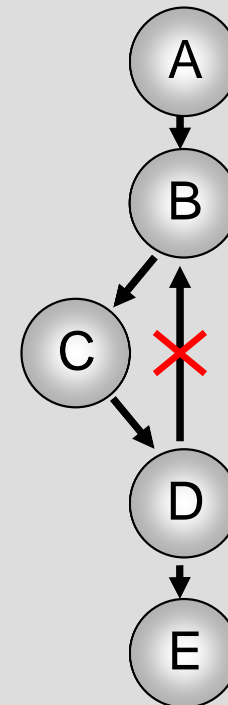
- ***Modellierung des intraprozeduralen Kontrollflusses:***

Azyklische Teilgraphen:



w_A = WCET *aller in A startenden* Pfade

(Reduzierbare) Schleifen:

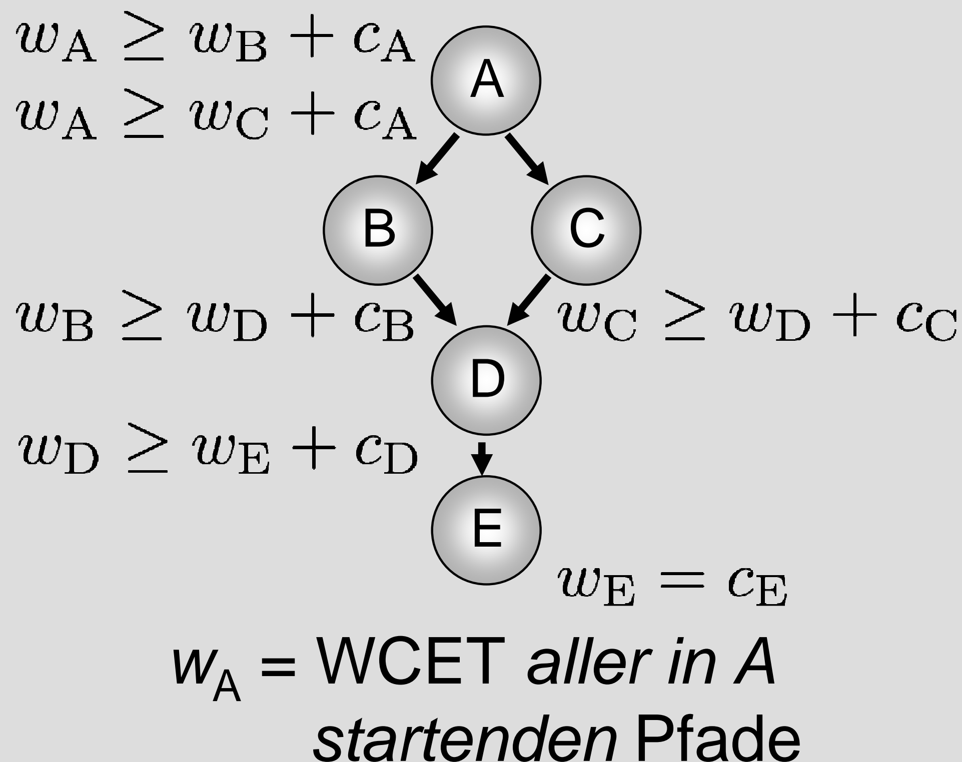


- Innerste Schleife *L* wie azyklischen Graph betrachten

ILP zur SPM-Allokation von Daten

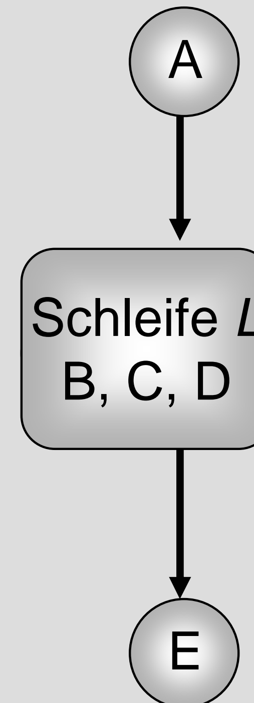
- ***Modellierung des intraprozeduralen Kontrollflusses:***

Azyklische Teilgraphen:



w_A = WCET *aller in A startenden* Pfade

(Reduzierbare) Schleifen:



- Innerste Schleife *L* wie azyklischen Graph betrachten
- Schleife *L* falten
- Kosten von *L*:
  $c_L = w_B * C^L_{max}$
- Iteriere mit nächsten innersten Schleifen

[**V. Suhendra et al.**, *WCET Centric Data Allocation to Scratchpad Memory, RTSS 2005*]

ILP zur SPM-Allokation von Daten

- ***Modellierung des intraprozeduralen Kontrollflusses:***
  - Für Endknoten b_j eines azyklischen Teilgraphen wird w_j auf die Kosten c_j gesetzt.
  - Für alle anderen Knoten b_j eines azyklischen Teilgraphen muß die WCET der in b_j startenden Pfade größer gleich der WCET jedes Nachfolgers b_{succ} sein, plus der Kosten c_j von b_j.
  - Für *jeden* Nachfolger b_{succ} von b_j wird eine Nebenbedingung im ILP formuliert.
  - ☞ Variable w_j modelliert tatsächlich alle in b_j startenden Pfade. Durch ≥-Operator in den Ungleichungen wird Maximum-Bildung über alle in w_j startenden Pfade vorgenommen.
  - ☞ Eventuelle *WCEP-Wechsel* zwischen zwei Nachfolgern b_{succ1} und b_{succ2} von b_j werden automatisch berücksichtigt.

ILP zur SPM-Allokation von Daten

- ***Modellierung des intraprozeduralen Kontrollflusses:***
  - *Reduzierbare Schleifen L* haben jeweils genau einen Eintritts- und Austrittspunkt (b^L_{entry}, b^L_{exit}).
  - Durch „Ausblenden" der Rücksprungkante einer reduzierbaren Schleife *L* wird der CFG des Schleifenkörpers azyklisch.
  - Erzeuge Nebenbedingungen für Schleifenkörper wie auf voriger Folie.
  - ☞ Variable w^L_{entry} modelliert WCET des kompletten Schleifen-körpers von *L*, wenn dieser *genau einmal* ausgeführt wird.
  - Multiplikation von w^L_{entry} mit der max. Iterationszahl C^L_{max} von *L* liefert WCET für *alle Ausführungen* der Schleife.
  - ☞ Kosten von *L* gleich der WCET von *L* für alle Ausführungen

ILP zur SPM-Allokation von Daten

- ***Modellierung des interprozeduralen Kontrollflusses:***
  - Für Funktion *F* mit Eintrittspunkt b^F_{entry}:
    Variable $w^F_{entry} \cong WCET_{EST}$ von *F* für exakt 1 Ausführung von *F*
  - Ruft Block b_j Funktion *F* auf: Addiere $WCET_{EST}$ von *F* zu $WCET_{EST}$ von b_j.
  - Hierzu wird Strafe für Funktionsaufrufe *(„Call Penalty“)* pro Basisblock b_j definiert:

$$cp_j = \begin{cases} w^F_{entry} & \text{if } b_j \text{ calls } F \\ 0 & \text{else} \end{cases}$$

☞ Endgültige Nebenbedingung für alle Nachfolger b_{succ} von b_j:

$$\forall (b_j, b_{succ}) : w_j \geq w_{succ} + c_j + cp_j$$

ILP zur SPM-Allokation von Daten

- ***Scratchpad-Kapazitätsbeschränkung:***

$$\sum_{d_i \in \text{data objects}} (S_i * y_i) \leq S_{spm}$$

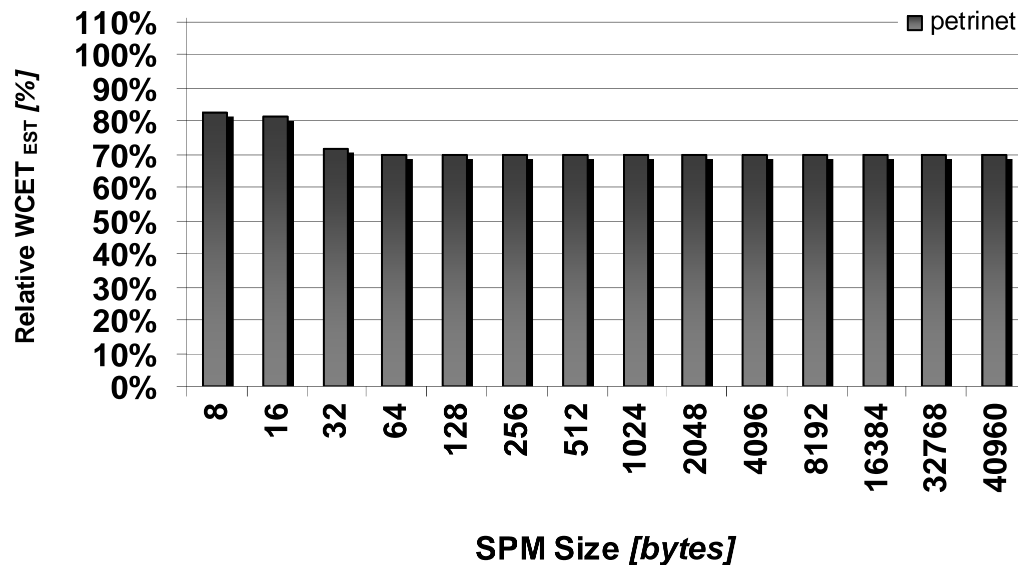
Summe der Größen aller auf SPM plazierten Datenobjekte kleiner gleich verfügbarer SPM-Kapazität.

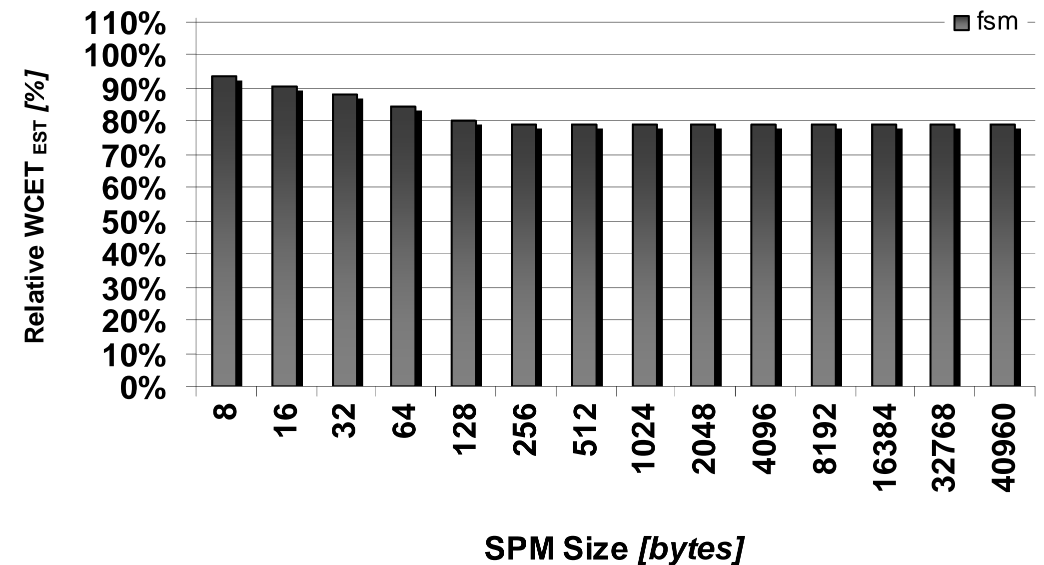
- ***Zielfunktion:***

$$w_{entry}^{\texttt{main}} \leadsto min.$$

Minimiere WCET_{EST} der ausgezeichneten Einstiegsfunktion **`main`**, d.h. minimiere WCET_{EST} des gesamten Programms.

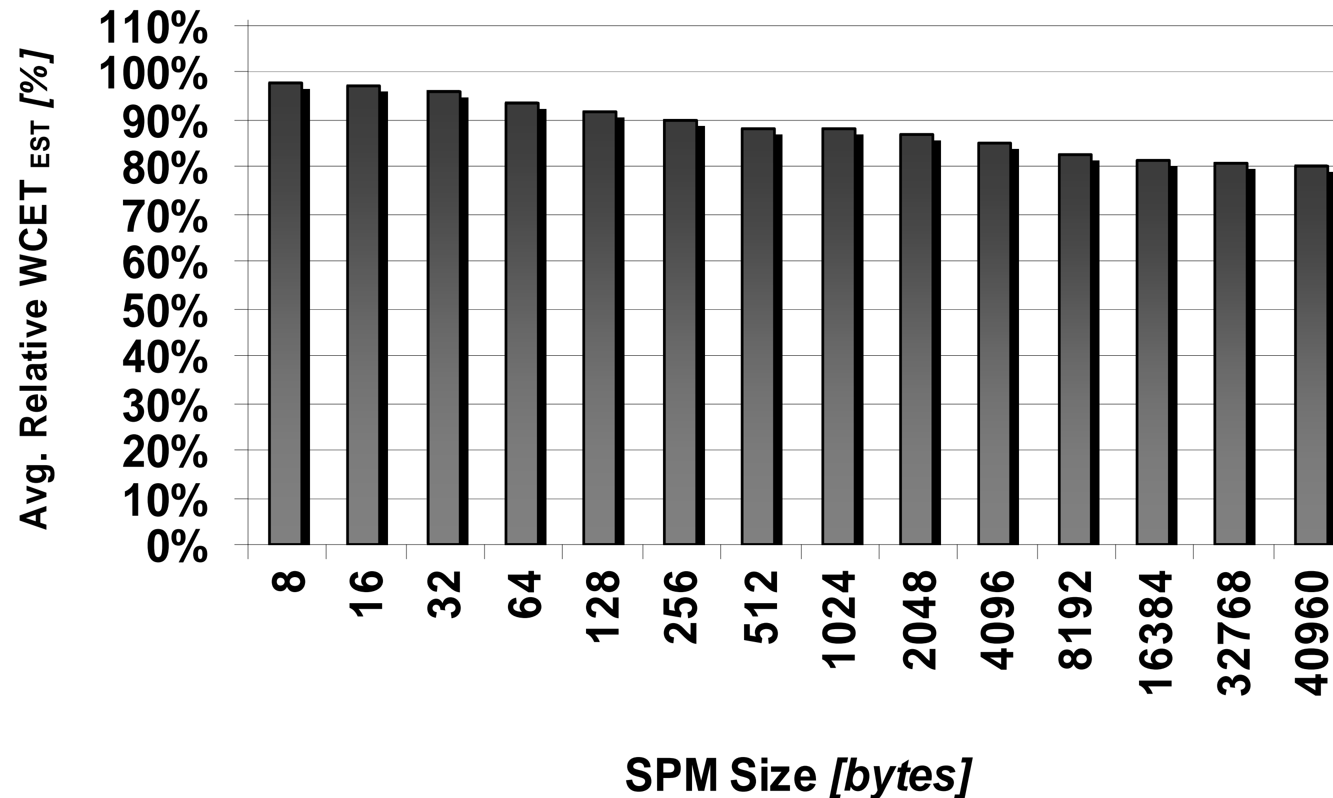
Relative $WCET_{EST}$ nach Daten-SPM-Allokation





- 100% = $WCET_{EST}$ ohne Daten-SPM
- `petrinet`: 6 globale Daten von insgesamt 72 Bytes;
  Bereits 8 Bytes Daten-SPM reduzieren $WCET_{EST}$ um 17,5%.
- `fsm`: 98 globale Variablen a 4 Bytes;
  kontinuierliche $WCET_{EST}$-Reduktionen bis zu 21,4%.

Relative $WCET_{EST}$ nach Daten-SPM-Allokation



- 100% = $WCET_{EST}$ ohne Daten-SPM
- Durchschnitt über 14 verschiedene Benchmarks
- $WCET_{EST}$-Reduktionen von 2,6% bis 20,2%.

ILP zur SPM-Allokation von Code

- ***Binäre Entscheidungsvariablen pro Basisblock:***

$$x_i = \begin{cases} 1 & \text{if basic block } b_i \text{ is assigned to } mem_{spm} \\ 0 & \text{if basic block } b_i \text{ is assigned to } mem_{main} \end{cases}$$

- ***Kosten eines Basisblocks b_i:***

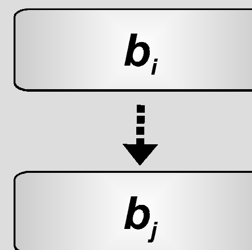
$$c_i = C^i_{main} * (1 - x_i) + C^i_{spm} * x_i$$

$WCET_{EST}$ von b_i abhängig davon, ob b_i aus Hauptspeicher oder aus SPM heraus ausgeführt wird.

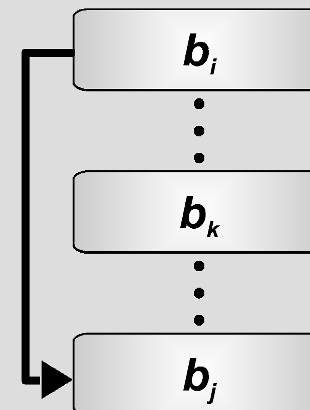
- ***Modellierung des intraprozeduralen Kontrollflusses:***
Wie zuvor zur WCET-bewußten SPM-Allokation von Daten.

ILP zur SPM-Allokation von Code

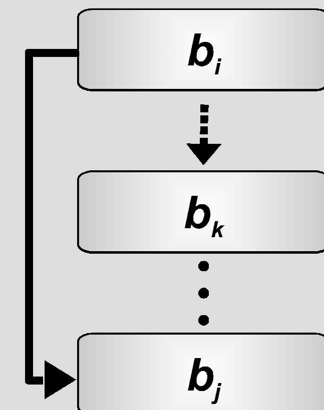
- ***Allokation aufeinander folgender Basisblöcke:***
  - Allokation aufeinander folgender Basisblöcke in verschiedene Speicher erfordert Einfügen spezieller Sprung-Befehle.
  - Sprünge zwischen Speichern teuer: mehr als 1 Instruktion
  - Sprung-Befehle: *Variablen* im ILP bzgl. $WCET_{EST}$ und Code-Größe, abhängig von Entscheidungsvariablen *(☞ siehe Kap. 6)*
- ***Sprung-Szenarien: („Jump Scenarios“, JS)***



a) Implicit



b) Unconditional



c) Conditional

ILP zur SPM-Allokation von Code

- ***Sprung-Strafe („Jump Penalty") ($\otimes \cong$ Boolesches XOR):***
  - *Impliziter Sprung:*

    $$jp^i_{impl} = (x_i \otimes x_j) * P_{high}$$

    Hohe Strafe für BBs b_i und b_j in verschiedenen Speichern.
  - *Unbedingter Sprung:*

    $$jp^i_{uncond} = (x_i \otimes x_j) * P_{high} + \overline{(x_i \otimes x_j)} * (1 - \prod_{b_k \in \mathrm{JS\ \ b)}} (x_i \otimes x_k)) * P_{low}$$

    Hohe Strafe für BBs b_i und b_j in verschiedenen Speichern ODER geringe Strafe für b_i und b_j im gleichen Speicher, aber ein BB b_k zwischen b_i und b_j muß umsprungen werden.
  - *Bedingter Sprung:* Kombination von jp^i_{impl} und jp^i_{uncond}.

ILP zur SPM-Allokation von Code

- ***Sprung-Strafe pro Basisblock b_i:***

$$jp_i = \begin{cases} jp^i_{impl} & \text{if Jump Scenario of } b_i \text{ is } \mathit{implicit} \\ jp^i_{uncond} & \text{if Jump Scenario of } b_i \text{ is } \mathit{unconditional} \\ jp^i_{cond} & \text{if Jump Scenario of } b_i \text{ is } \mathit{conditional} \\ 0 & \text{else} \end{cases}$$

- ***Strafe für Funktionsaufrufe pro Basisblock b_i:***

$$cp_i = \begin{cases} w^F_{entry} + (x_i \otimes x^F_{entry}) * P_{high} & \text{if } b_i \text{ calls } F \\ \quad + \overline{(x_i \otimes x^F_{entry})} * P_{low} & \\ 0 & \text{else} \end{cases}$$

Ruft Block b_j Funktion F auf: Addiere $WCET_{EST}$ von F zu $WCET_{EST}$ von b_j. Addiere zusätzlich P_{high} wenn Funktionsaufruf über Speicher hinweg springt, P_{low} sonst.

ILP zur SPM-Allokation von Code

- ***Nebenbedingung für alle Nachfolger b_{succ} von b_i:***

$$\forall(b_i, b_{succ}) : w_i \geq w_{succ} + c_i + cp_i + jp_i$$

- ***Größe eines Basisblocks b_i:***
  - Größe von b_i hängt von Sprung-Code in b_i ab
  - Größe des Sprung-Codes in b_i hängt von Sprung-Szenario ab:

$$s_i = \begin{cases} (x_i \wedge \overline{x_j}) * S_{impl} & \text{if JS of } b_i \text{ is } \mathit{implicit} \\ (x_i \wedge \overline{x_j}) * S_{uncond} & \text{if JS of } b_i \text{ is } \mathit{uncond.} \\ (x_i \wedge \overline{x_k}) * S_{impl} + & \text{if JS of } b_i \text{ is } \mathit{cond.} \\ \quad (x_i \wedge \overline{x_j}) * S_{uncond} & \\ (x_i \wedge \overline{x^F_{entry}}) * S_{call} & \text{if } b_i \text{ calls } F \\ 0 & \text{else} \end{cases}$$

ILP zur SPM-Allokation von Code

- ***Scratchpad-Kapazitätsbeschränkung:***

$$\sum_{b_i}(S_i * x_i + s_i) \leq S_{spm}$$

Summe der Größen aller auf SPM plazierten Basisblöcke ohne Sprungcode (S_i) plus Größe des Sprung-Codes von b_i kleiner gleich verfügbarer SPM-Kapazität.

- ***Zielfunktion:***

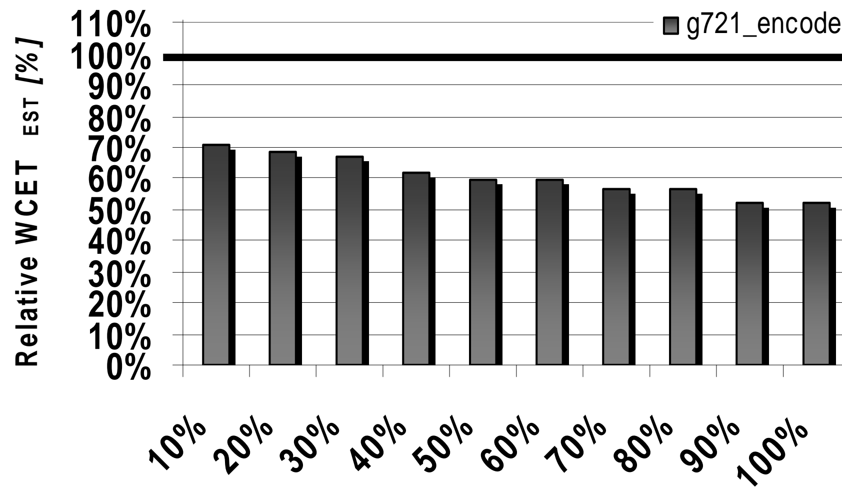
$$w_{entry}^{\texttt{main}} \rightsquigarrow min.$$

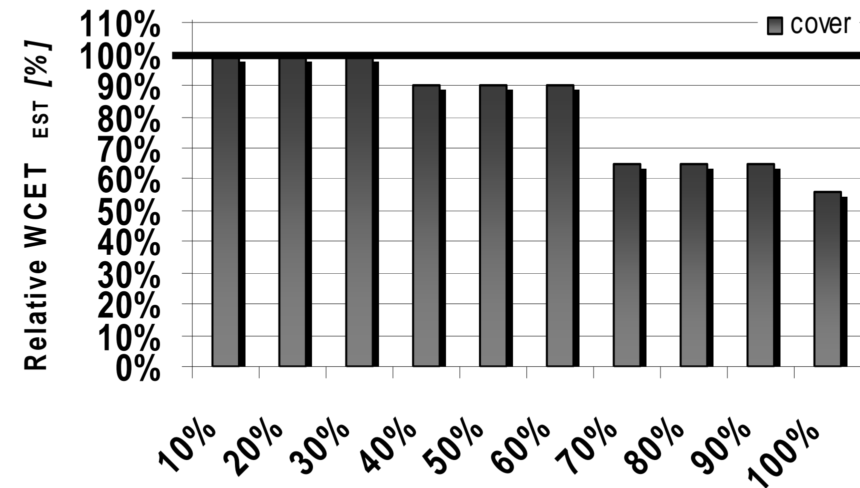
[**H. Falk, J. C. Kleinsorge,** *Optimal Static WCET-aware Scratchpad Allocation of Program Code, San Francisco, DAC 2009]*

Bestimmung der Konstanten des ILP

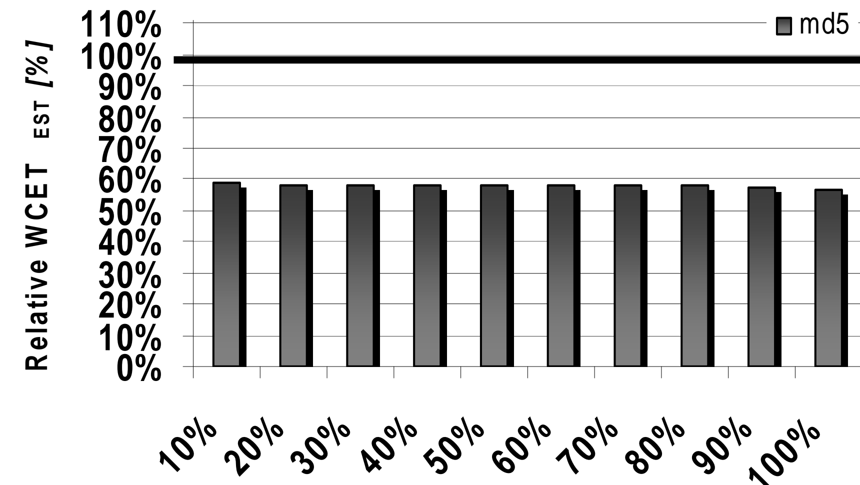
- *WCET$_{EST}$ C^i_{main}, C^i_{spm} pro Basisblock b_i für beide Speicher:*
  Durch zwei WCET-Analysen ermittelt, in denen alle Basisblöcke einmal im SPM, das andere Mal im Hauptspeicher liegen.
- *Max. Iterationszahl von Schleifen C^L_{max}:*
  Entweder durch Flow Facts im Quellcode annotiert, oder durch WCC's Loop Analyzer bestimmt.
- *Größe S_i eines Basisblocks ohne Sprung-Code:*
  Durch Abzählen der LIR-Operationen
- *Größe S_{spm} des Scratchpads:*
  WCC's Speicher-Beschreibung entnommen.
- *Übrige Konstanten experimentell ermittelt:*
  $P_{high} = 16$ $P_{low} = 8$
  $S_{impl} = S_{uncond} = 10$ $S_{call} = 12$

Relative $WCET_{EST}$ nach Code-SPM-Allokation

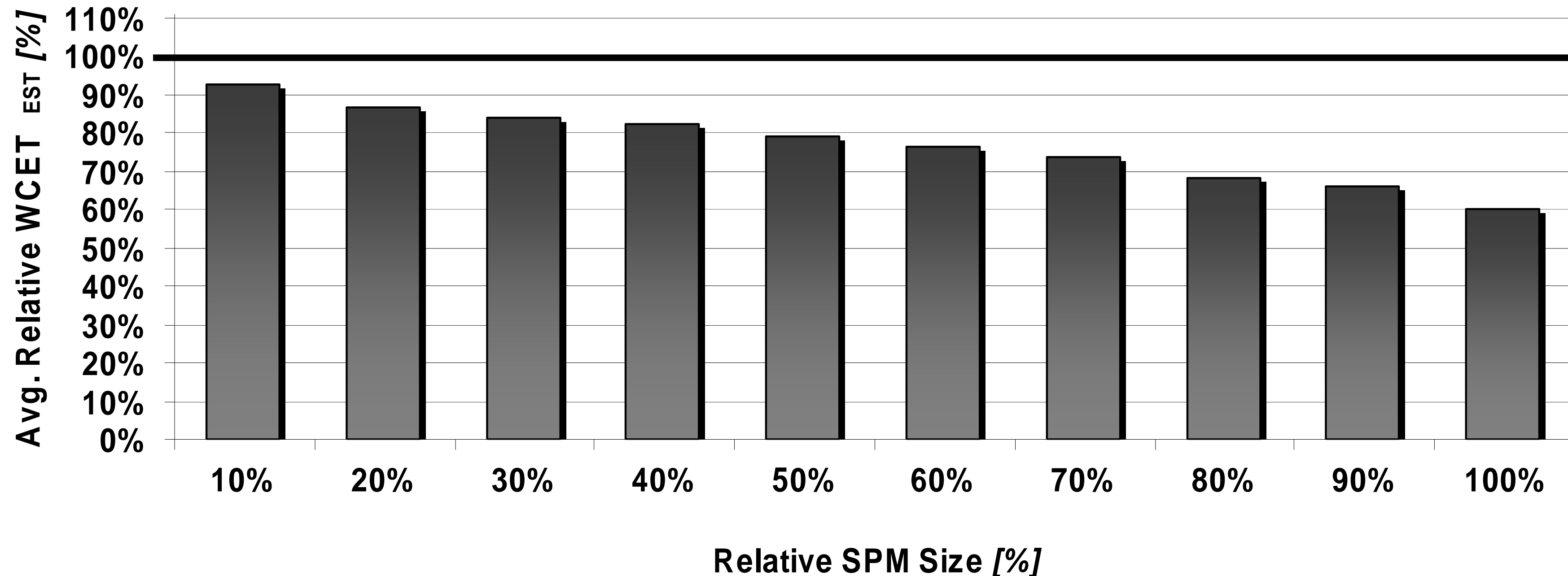




- X-Achse: SPM-Größe = *x*% der Code-Größe des Benchmarks
- Y-Achse: 100% = $WCET_{EST}$ ohne Code-SPM



Relative $WCET_{EST}$ nach Code-SPM-Allokation



- X-Achse: SPM-Größe = *x*% der Code-Größe des Benchmarks
- Y-Achse: 100% = $WCET_{EST}$ ohne Code-SPM
- Durchschnitt über 73 verschiedene Benchmarks
- $WCET_{EST}$-Reduktionen von 7,4% bis 40%.

Zwischenfazit

Abgleich mit Konsequenzen für $\mathbf{WCET_{EST}}$-Optimierungen:

Optimierungsstrategien zur $WCET_{EST}$-Minimierung...

- ...brauchen zwingend Detailwissen über den WCEP.

✓ *WCET SPM-Allokationen berücksichtigen WCEP.*

- ...müssen immer berücksichtigen, daß sich der WCEP nach jeder Entscheidung, die eine Optimierung trifft, ändern kann.

✓ *WCET SPM-Allokationen betrachten inhärent aktuellen WCEP.*

- ...sollten ihre Entscheidungen, wo was zu optimieren ist, nicht nur aufgrund lokaler Informationen treffen, sondern sollten stets die globalen Auswirkungen einer Entscheidung berücksichtigen.

✓ *Zielfunktionen der ILPs modellieren die globale WCET eines Programms, die zu minimieren ist.*

Literatur

- Wegen der großen Nähe zur aktuellen Forschung existiert keine gute einführende Literatur mit Übersichtscharakter.
(Ein Übersichts-Paper über dieses Kapitel ist gerade bei einer Zeitschrift eingereicht...)
- ☞ Daher bitte die in diesem Foliensatz angegebenen Referenzen zu den jeweiligen einzelnen Optimierungen und die WCC-Webseite verwenden.

Zusammenfassung

- **Compiler zur $WCET_{EST}$-Minimierung**
  - Integration eines formalen WCET-Zeitmodells in Compiler
  - Herausforderung: Berücksichtigung instabiler WCEPs im Verlauf von Optimierungen
- **WCET-bewußte Optimierungen**
  - Procedure Cloning & Positioning: Greedy-Heuristiken, die aktuellen WCEP durch wiederholte WCET-Analysen ermitteln
  - Register-Allokation: zyklische Abhängigkeiten zwischen Register-Allokation und WCET-Analyse; Graphfärbung entlang des stets aktuellen WCEP
  - Scratchpad-Allokationen: inhärente WCEP-Modellierung in ILPs; daher keine wiederholten WCET-Analysen nötig