ulm university universität uulm

Compiler für Eingebettete Systeme

[CS7506]

Sommersemester 2014

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm

Kapitel 9

Compiler zur $WCET_{EST}$-Minimierung

Inhalte der Vorlesung

1. Einordnung & Motivation der Vorlesung
2. Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten
3. Interner Aufbau von Compilern
4. Prepass-Optimierungen
5. HIR Optimierungen und Transformationen
6. Instruktionsauswahl
7. LIR Optimierungen und Transformationen
8. Register-Allokation
9. **Compiler zur $WCET_{EST}$-Minimierung**
10. Ausblick

Inhalte des Kapitels

9. Compiler zur WCET_{EST}-Minimierung

- Einführung
  - Integration eines WCET-Zeitmodells in einen Compiler
  - Herausforderungen bei der WCET-Optimierung
- *Procedure Cloning & Positioning*
  - WCET-bewusstes *Procedure Cloning*
  - *Procedure Positioning* zur Reduktion von *Cache Misses*
- Register-Allokation
  - Problem der Standard Graph-Färbung
  - WCET-bewusste Graph-Färbung
- *Scratchpad*-Allokation von Daten und Code
  - Allokation von globalen Daten
  - Allokation von Basisblöcken

Entwurfsmethodik von Software für Echtzeitsysteme

Jetzige industrielle Praxis (Automotive, Avionik)

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. Zeitmessungen liefern „beobachtete Laufzeiten“
6. Aufschlag von Sicherheitspuffer (z.B. 20%) auf höchste beobachtete Laufzeit: „beobachtete *Worst-Case Execution Time (WCET)*“
7. Beobachtete WCET ≤ Echtzeit-Bedingungen? Nein: Gehe zu 1

Probleme dieser Methodik

Sicherheit

– Keine Gewähr, dass beobachtete WCET annähernd effektiver WCET entspricht

☞ Keine Garantie, dass hartes Echtzeitsystem *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*, bis Schritt 7 erfolgreich

Stand der Technik im Compilerbau

Zielfunktion heutiger Compiler-Optimierungen

– In der Regel Reduktion durchschnittlicher Laufzeiten *(Average-Case Execution Time, ACET)*: Beschleunigung einer „typischen“ Ausführung eines Programms für „typische“ Eingabedaten

☞ Keine gesicherte Aussage über Einfluss von Optimierungen auf WCETs möglich

Optimierungsstrategie

– Naiv: heutige Compiler enthalten kein echtes ACET-Zeitmodell

– Optimierungen werden angewendet wenn dies „Erfolg versprechend“ oder „plausibel“ erscheint

☞ Einfluss von Optimierungen auf ACETs sind dem Compiler unbekannt

☞ ACET-Optimierungen verschlechtern evtl. die WCET – Compiler werden oft ohne Optimierungen im Bereich von Echtzeitsystemen angewendet

Motivation

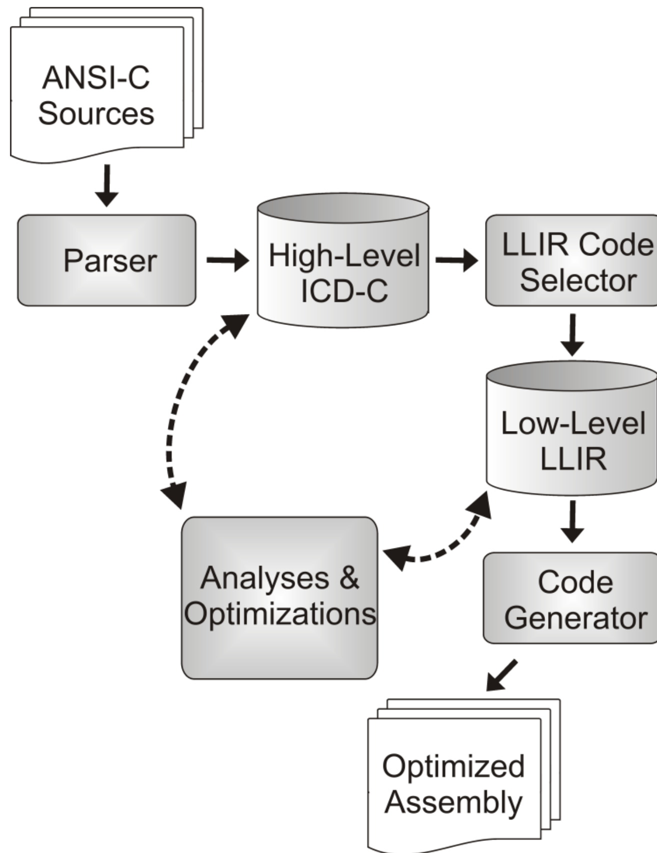
Entwurf eines Compilers, der

- statt durchschnittlicher Laufzeiten WCET-Abschätzungen betrachtet,
- formale Garantien zu *Worst-Case* Eigenschaften erlaubt, anstatt auf beobachteten WCETs zu basieren,
- automatische Optimierungen zur $WCET_{EST}$-Minimierung durchführt

Ansatz

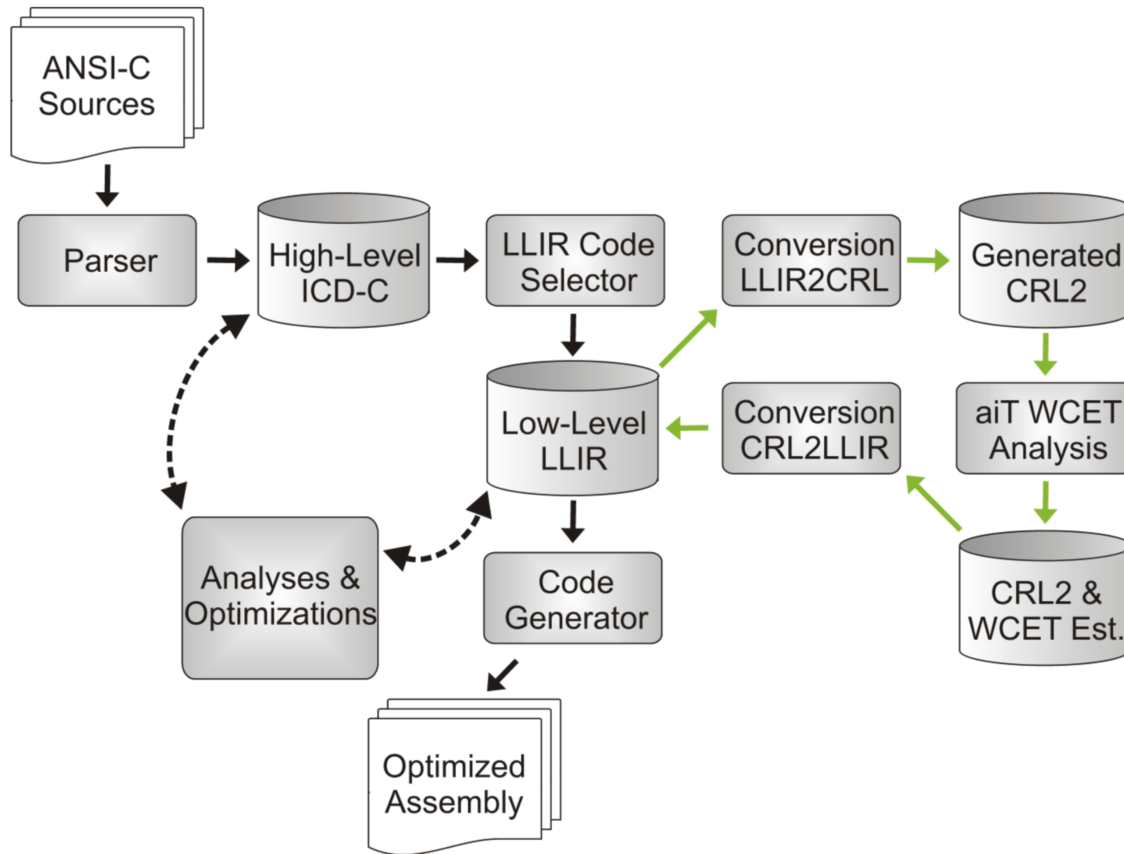
- Integration eines WCET-Zeitmodells in Compiler durch Anbindung statischer WCET-Analysewerkzeuge
- Ausnutzung des Zeitmodells zur systematischen WCET-Minimierung durch neuartige Optimierungen

Integration eines $WCET_{EST}$-Modells in Compiler (1)



- Re-Implementierung eines WCET Zeitmodells in Compiler nicht sinnvoll
- Statt dessen: Enge Integration von aiT *(☞ siehe Kapitel 5)*
- Kopplung in Prozessor-spezifischem Compiler *Back-End (LLIR)*
- Informationsaustausch durch Übersetzung LLIR ↔ CRL2
- Transparenter Aufruf von aiT innerhalb des Compilers
- Import WCET-relevanter Daten in Compiler *Back-End*

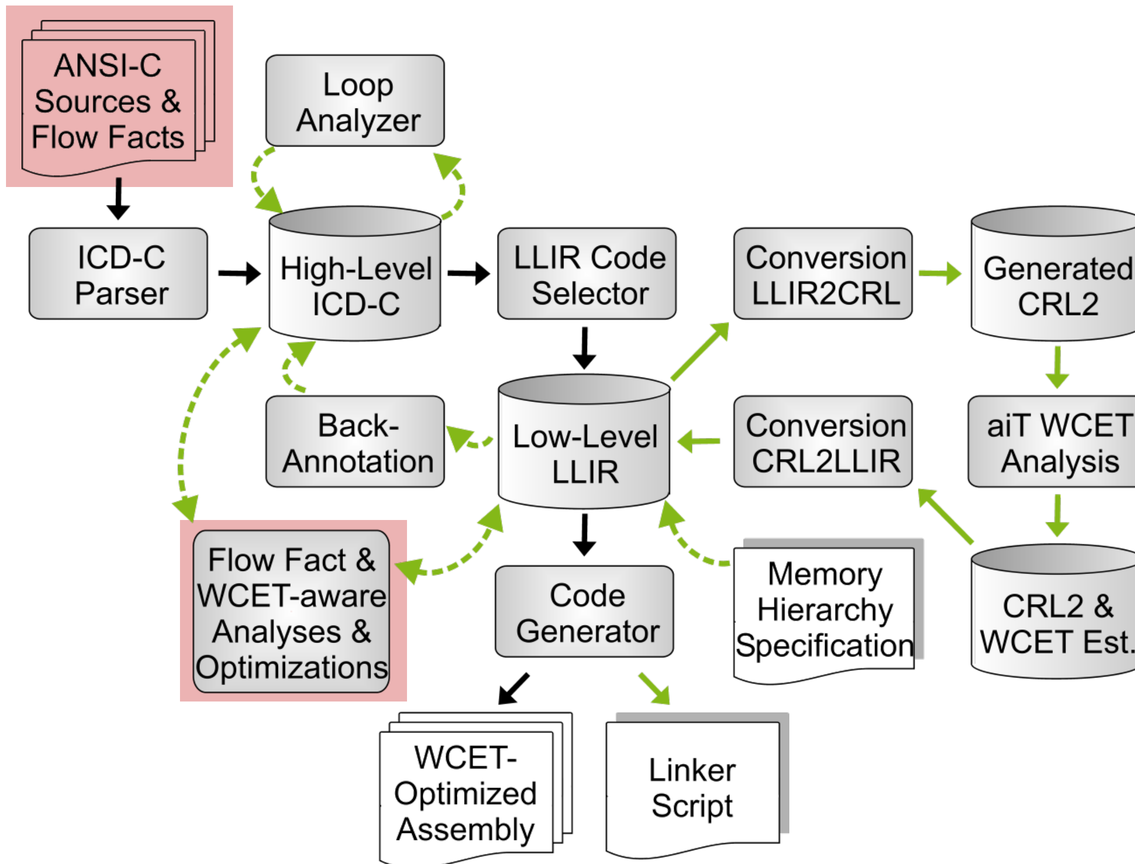
Integration eines WCET_{EST}-Modells in Compiler (2)



Relevante WCET-Daten:

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

WCC – Der *WCET-aware C Compiler* (1)



Flow Facts

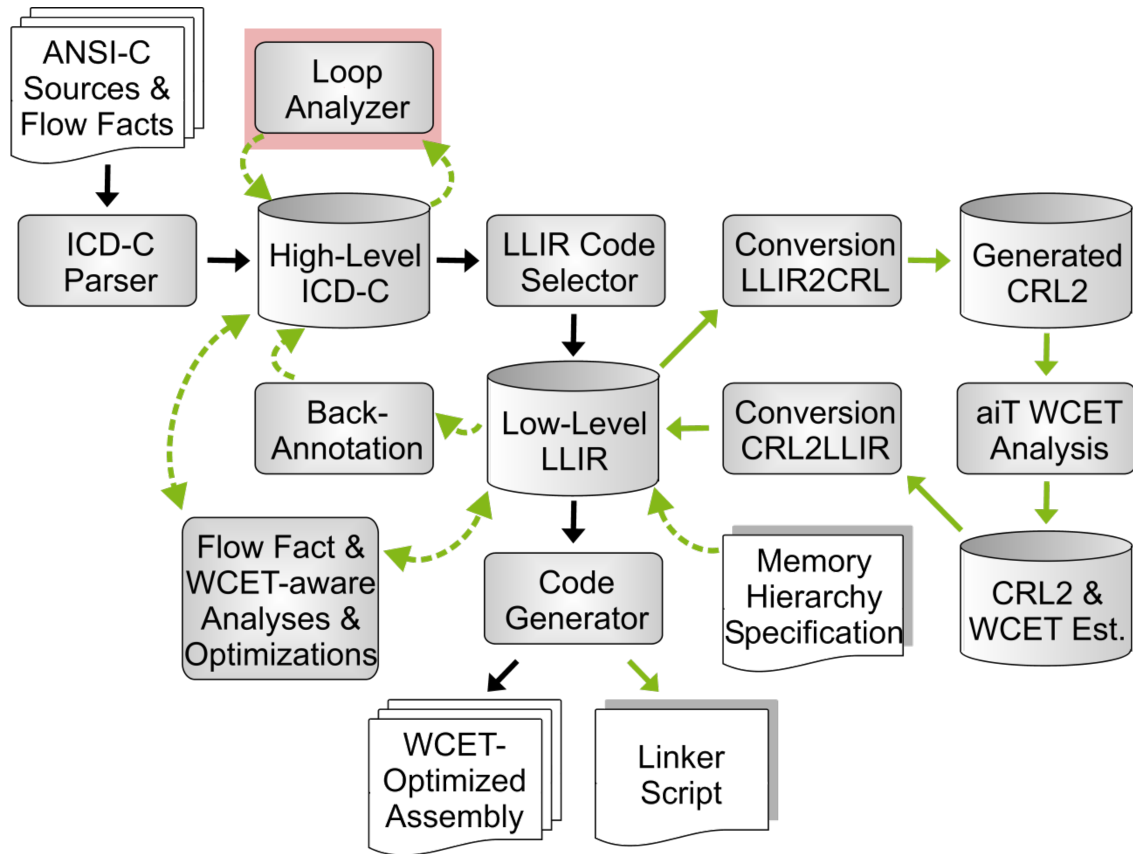
– WCET-Analyse: max. Iterationszahlen & Rekursionstiefen

– WCC: Annotation direkt im C Quellcode:

```
_Pragma( "loopbound min 10 max 10" );
```

– Optimierungen, die Kontrollfluss ändern, aktualisieren *Flow Facts* automatisch

WCC – Der *WCET-aware C Compiler* (2)

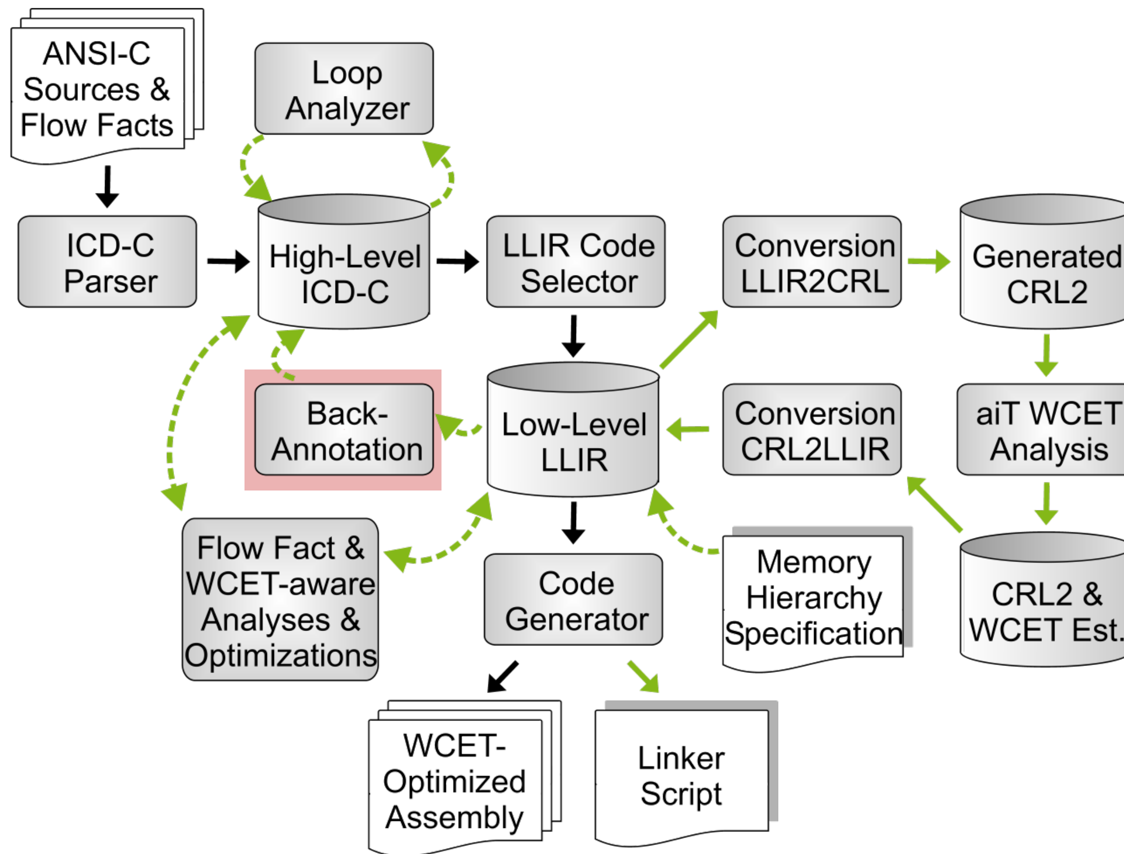


Loop Analyzer

- Manuelle *Flow Fact* Annotation umständlich und fehleranfällig
- WCC: Automatische Schleifenanalyse, die max. Iterationszahlen ermittelt
- Basiert z.T. auf Polytop-Modellen

  (☞ siehe Kapitel 4)

WCC – Der *WCET-aware C Compiler* (3)



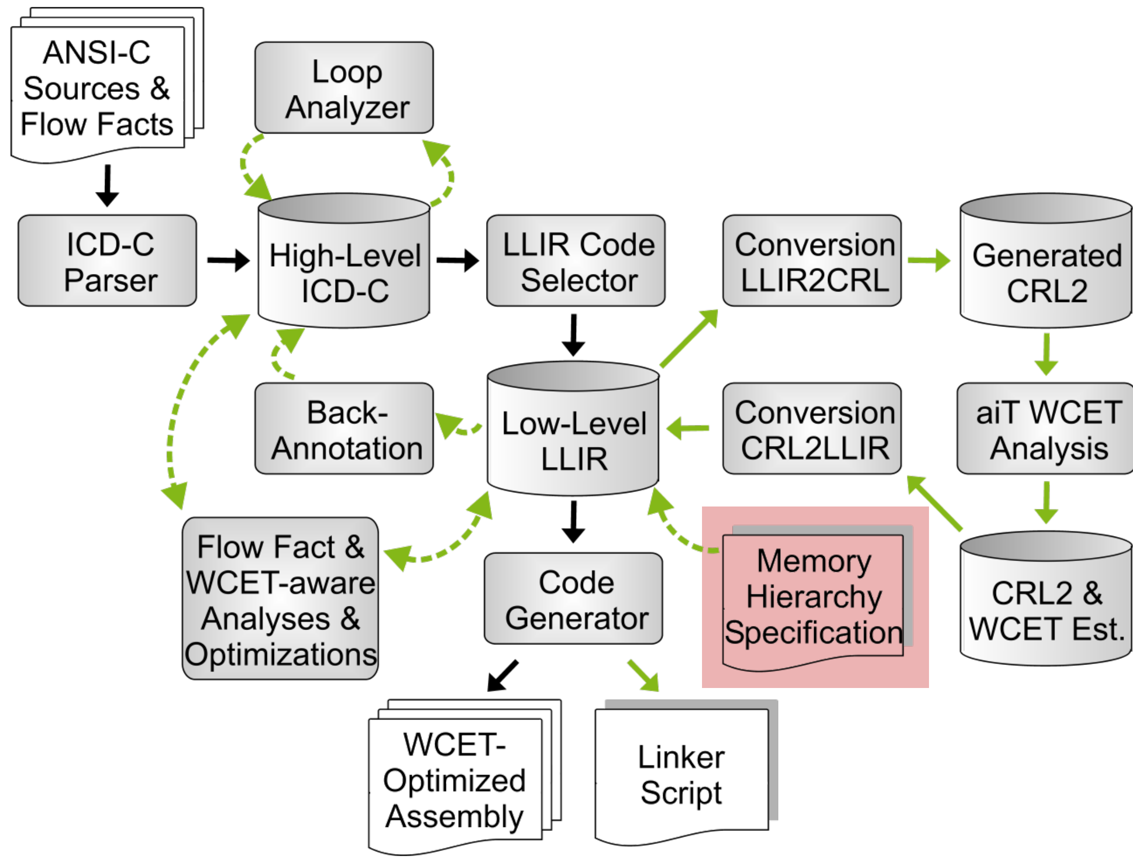
Back-Annotation

- $WCET_{EST}$-Daten von aiT nur im *Backend*
- HIR-Optimierungen haben keinen Zugriff auf $WCET_{EST}$-Daten

☞ $WCET_{EST}$-Minimierung auf HIR-Ebene nicht möglich

- WCC: *Back-Annotation* übersetzt $WCET_{EST}$-Daten von LIR nach HIR

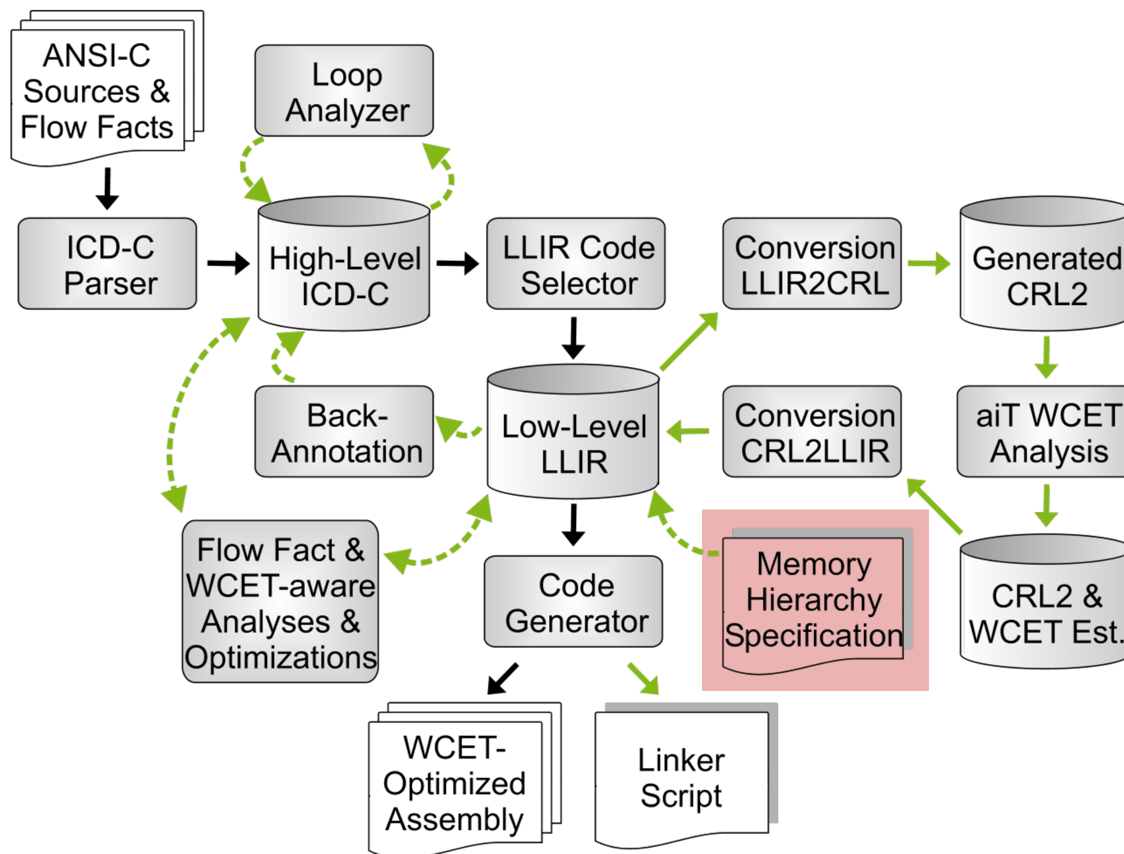
WCC – Der *WCET-aware C Compiler* (4)



Memory Hierarchy

- aiT arbeitet auf Binär-code mit physikalischen Adressen
- WCC muss aiT korrekte phys. Adressen für Code, Daten, Sprünge und *Load-/Store*-Operationen bereitstellen
- WCC braucht Detail-wissen über Speicher

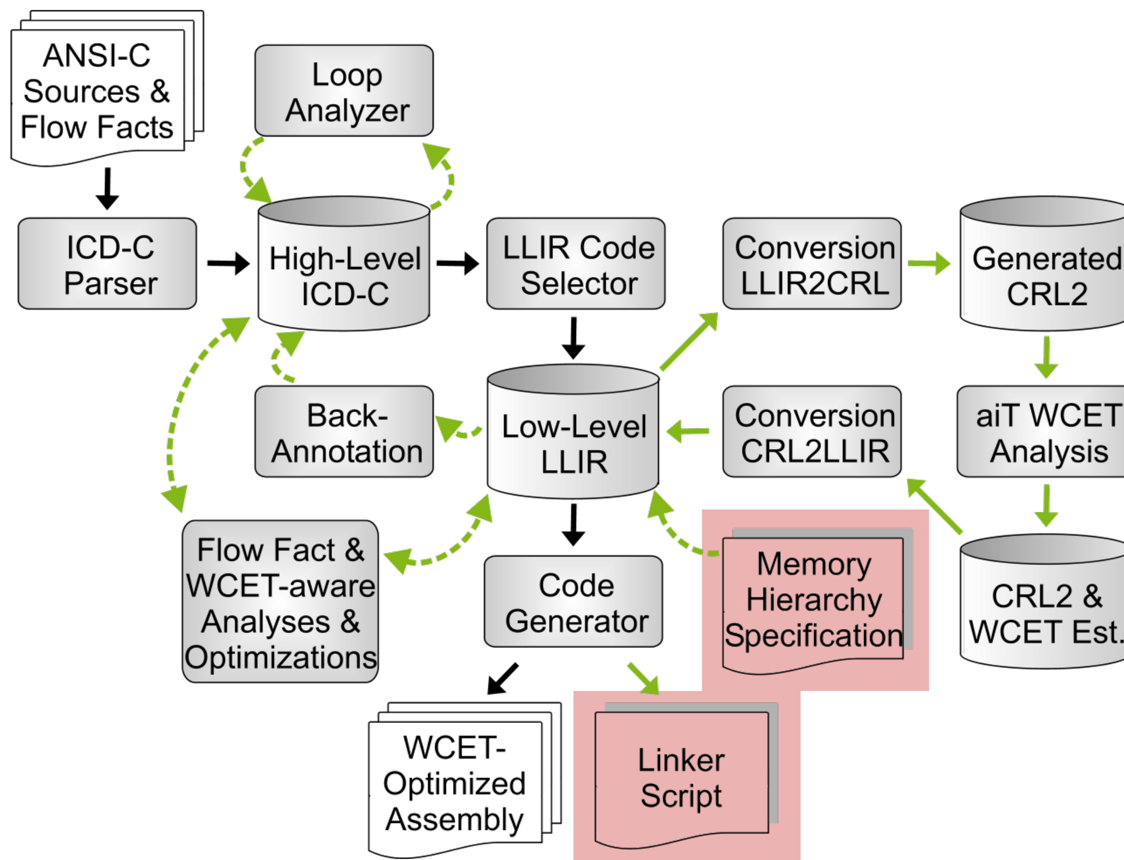
WCC – Der *WCET-aware C Compiler* (5)



Memory Hierarchy

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

WCC – Der *WCET-aware C Compiler* (6)



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 Linker-Skripts

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

Problematik der $WCET_{EST}$-Minimierung

Der *Worst-Case Execution Path (WCEP)*

– WCET eines Programms = Länge des längsten Ausführungspfades des Programms (WCEP)

– $WCET_{EST}$-Minimierung: Optimierung ausschließlich derjenigen Teile des Programms, die auf WCEP liegen

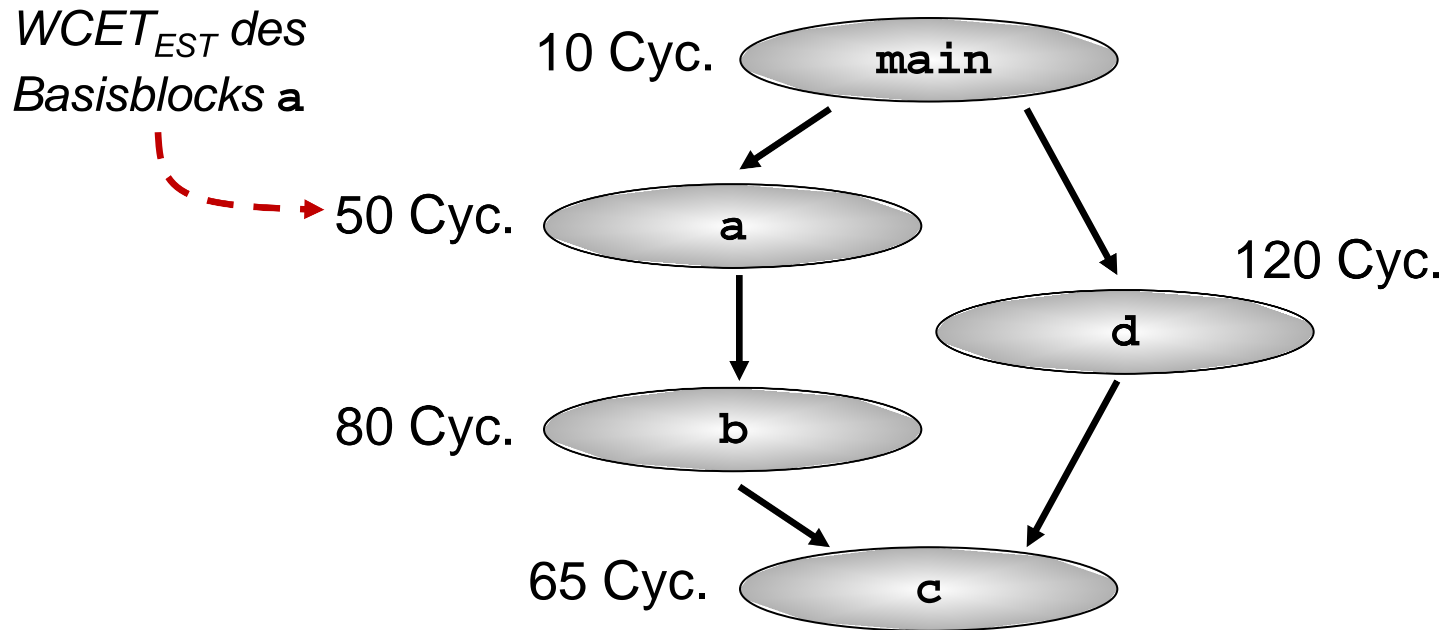
☞ Code-Optimierung abseits des WCEP bringt keine Verbesserung

☞ Optimierungen zur $WCET_{EST}$-Minimierung brauchen Detailwissen über WCEP!

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

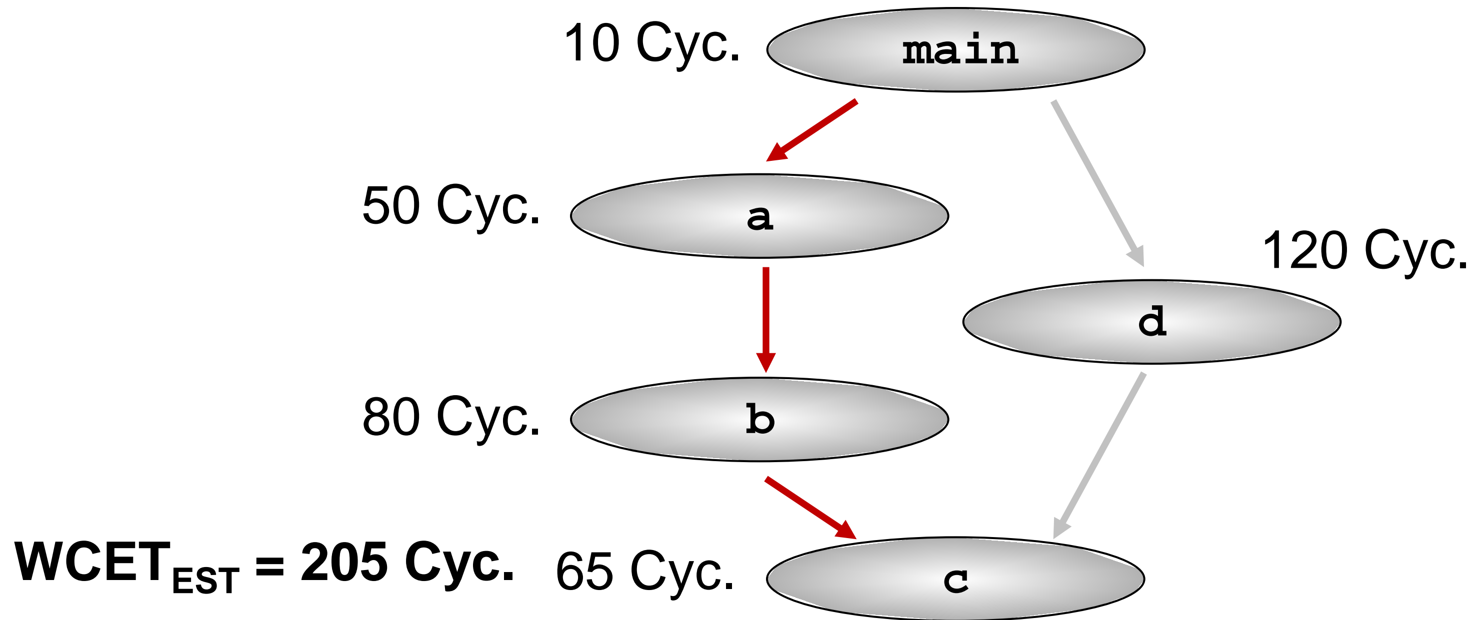
Jedoch...

Instabilität des WCEP (1)



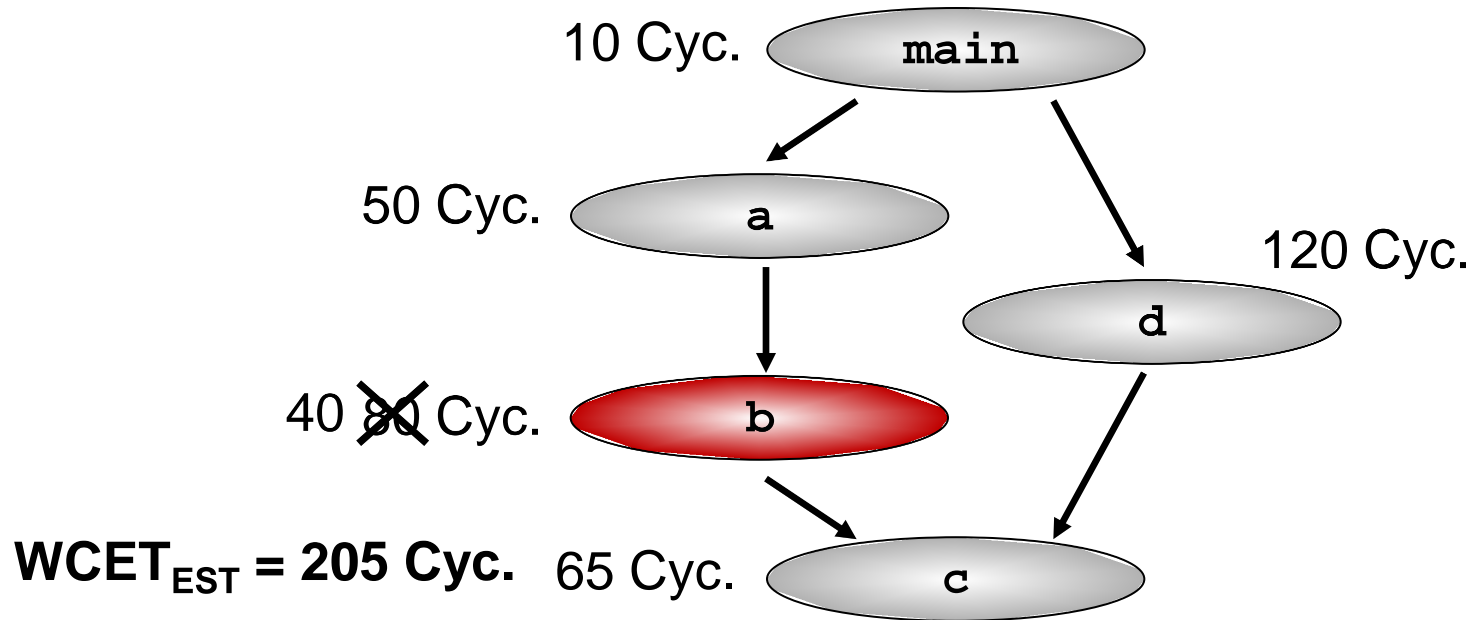
– Beispiel: Einfacher CFG mit 5 Basisblöcken

Instabilität des WCEP (2)



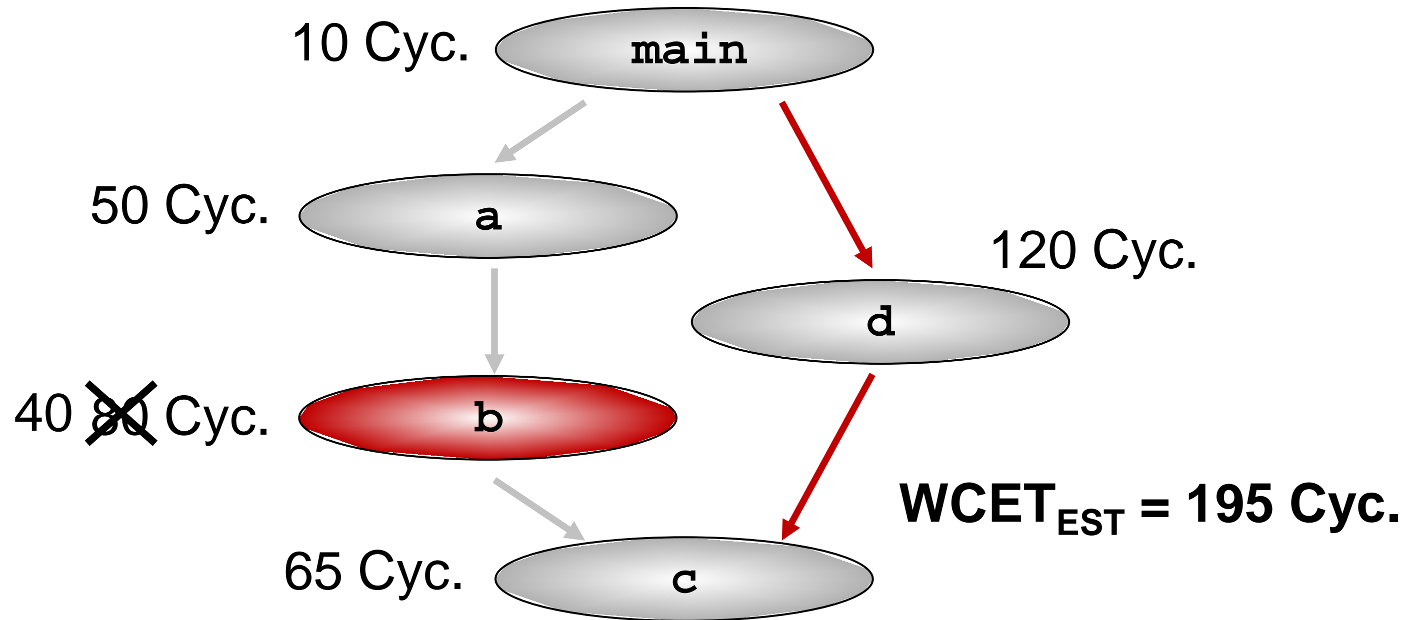
- Initialer WCEP: **`main`**, **`a`**, **`b`**, **`c`**
- WCEP-Länge = $WCET_{EST}$: 205
- Im folgenden: Optimierung von **`b`**

Instabilität des WCEP (3)



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

Instabilität des WCEP (4)



- Neuer WCEP: **`main`**, **`d`**, **`c`**
- Neue $WCET_{EST}$: 195

☞ ***WCEP ist wegen Optimierung umgeschlagen!***

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.***

Inhalte des Kapitels

9. Compiler zur WCET$_{EST}$-Minimierung

- Einführung
  - Integration eines WCET-Zeitmodells in einen Compiler
  - Herausforderungen bei der WCET-Optimierung
- *Procedure Cloning & Positioning*
  - WCET-bewusstes *Procedure Cloning*
  - *Procedure Positioning* zur Reduktion von *Cache Misses*
- Register-Allokation
  - Problem der Standard Graph-Färbung
  - WCET-bewusste Graph-Färbung
- *Scratchpad*-Allokation von Daten und Code
  - Allokation von globalen Daten
  - Allokation von Basisblöcken

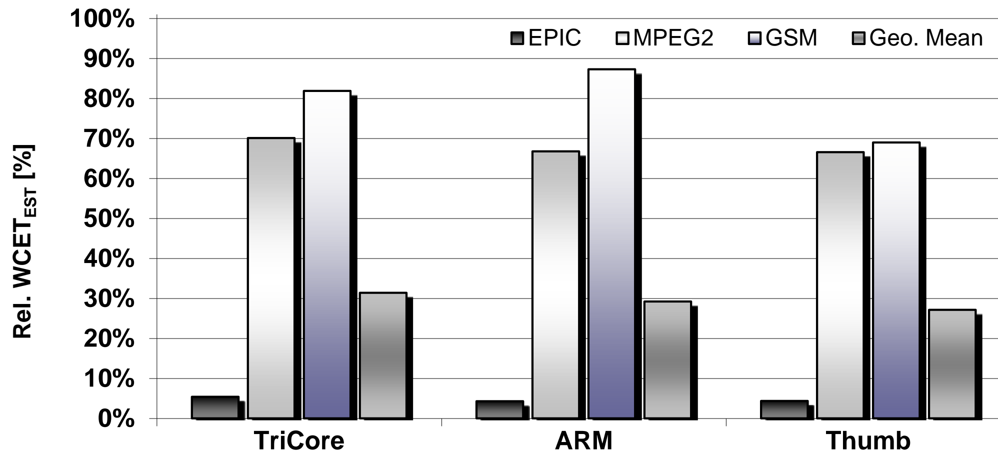
Wieso *Procedure Cloning* und WCET_{EST}?

Motivation *(☞ siehe Kapitel 5)*

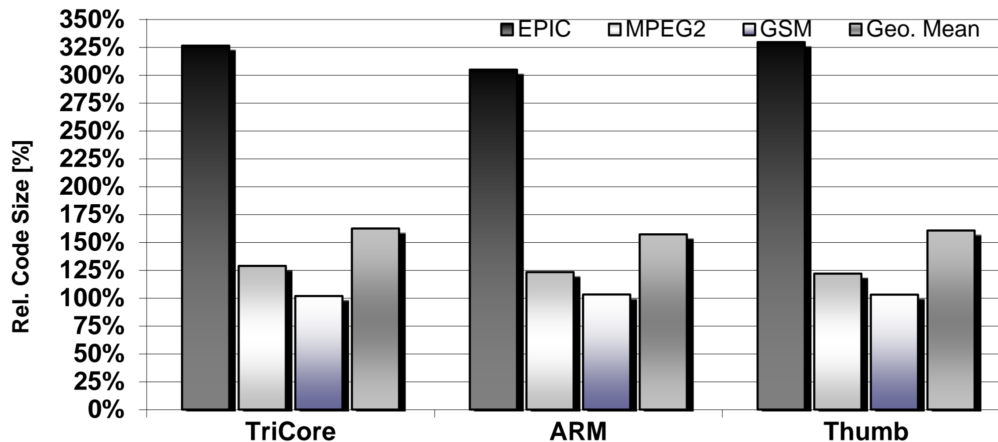
- Häufiges Vorkommen von *general-purpose* Funktionen in *special-purpose* Kontexten in Eingebetteter Software
- Gerade Schleifengrenzen werden sehr oft durch Funktionsparameter gesteuert
- Gerade Schleifengrenzen sind kritisch für eine präzise WCET-Analyse
- *Procedure Cloning* ermöglicht hochgradig präzise Annotation von Schleifengrenzen

*[**P. Lokuciejewski,** Influence of Procedure Cloning on WCET Prediction, CODES+ISSS, Salzburg, 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 der 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 der $WCET_{EST}$-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 von f 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* $\in$ *F*>)

$profit_f = (wcet_{orig} - wcet_f) / (cs_f - cs_{orig});$

Bestimme Funktion f_{opt} mit maximalem $profit_f$ UND

$cs_f \leq maxCodeSize;$

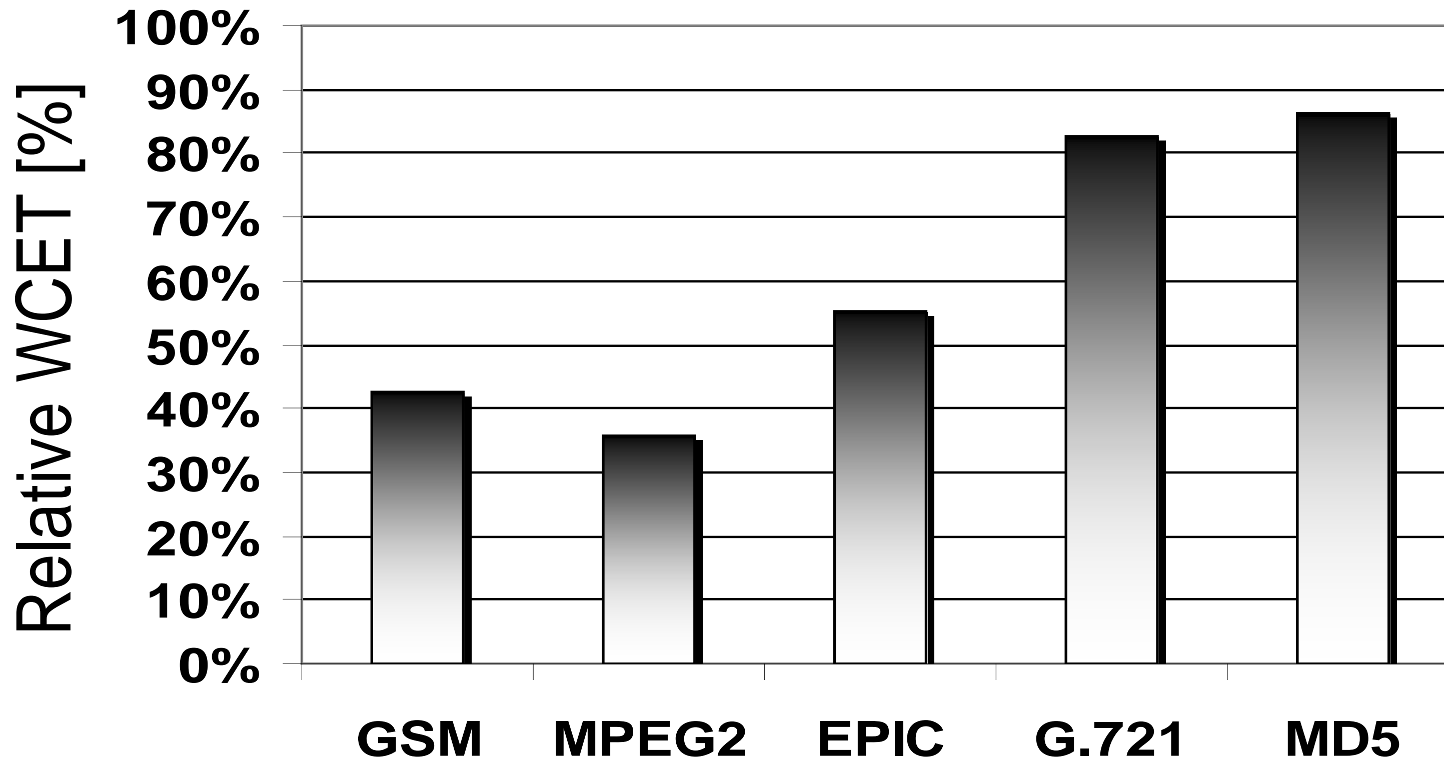
if (<f_{opt} existiert>)

doCloning(*P*, f_{opt});

goto <*Phase 1*>;

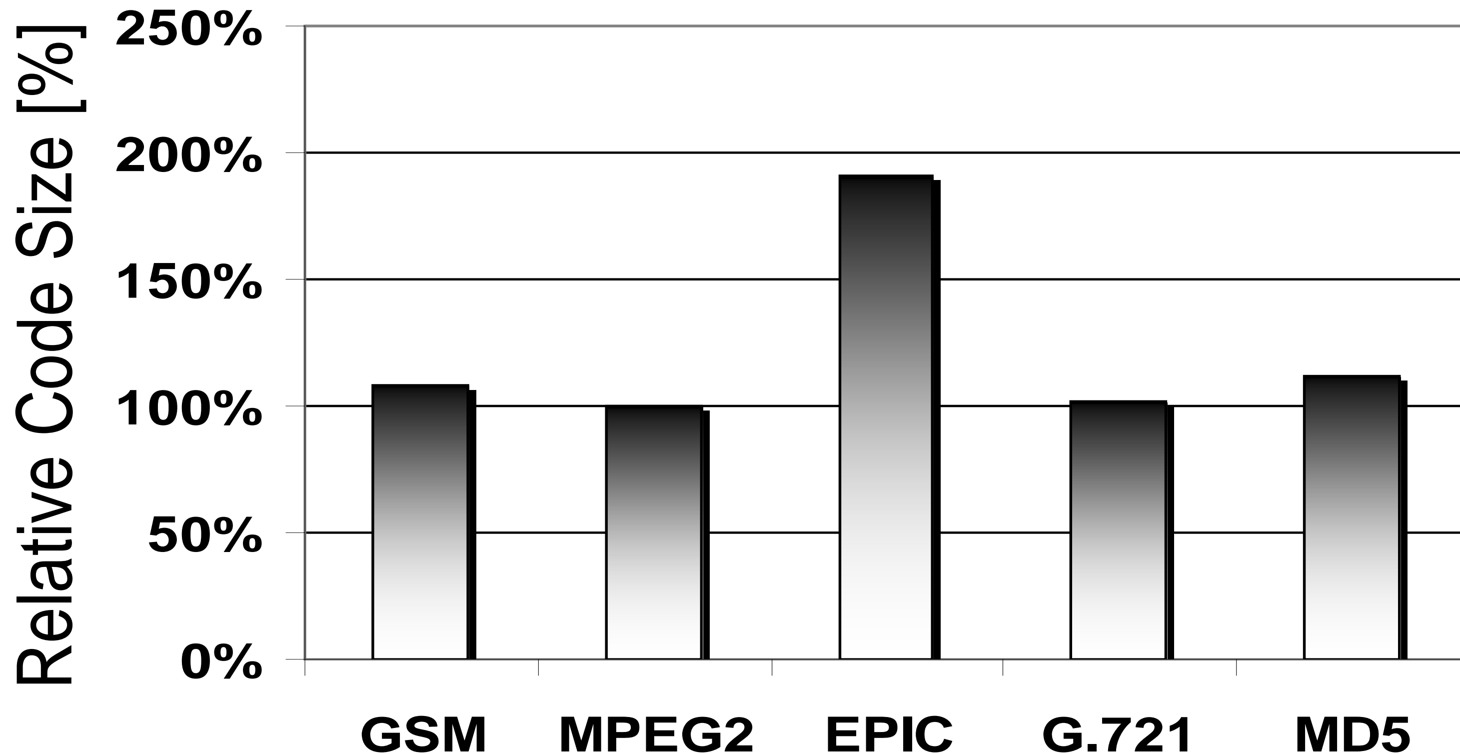
*[**P. Lokuciejewski,** WCET-Driven, Code-Size Critical Procedure Cloning, SCOPES, München, 2008]*

Relative $WCET_{EST}$ nach WCET-bewusstem *Cloning*



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

Relative Codegrößen nach WCET-bewusstem *Cloning*

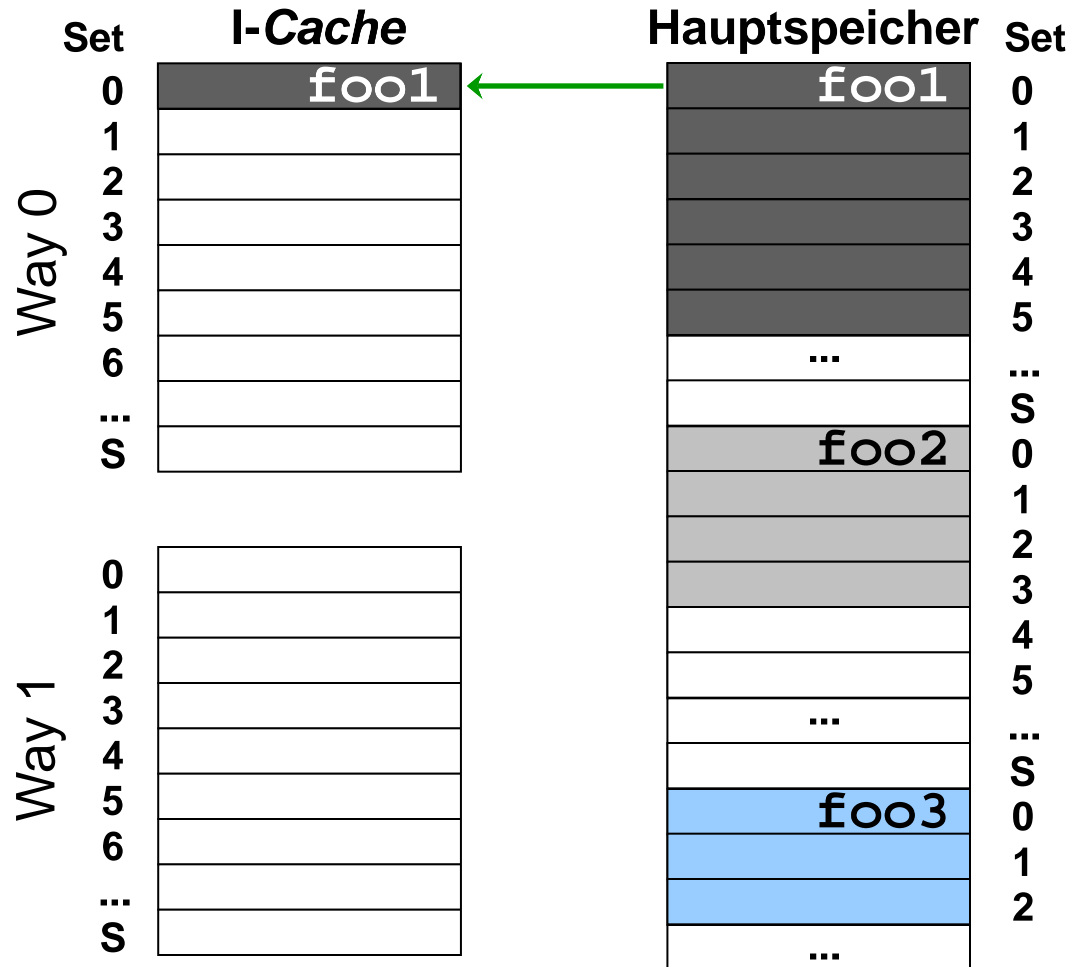


- 100% = Codegröße ohne *Procedure Cloning*
- Code-Vergrößerung von EPIC: 190% statt bisher 300%

Verdrängung von Code aus Befehls-*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.

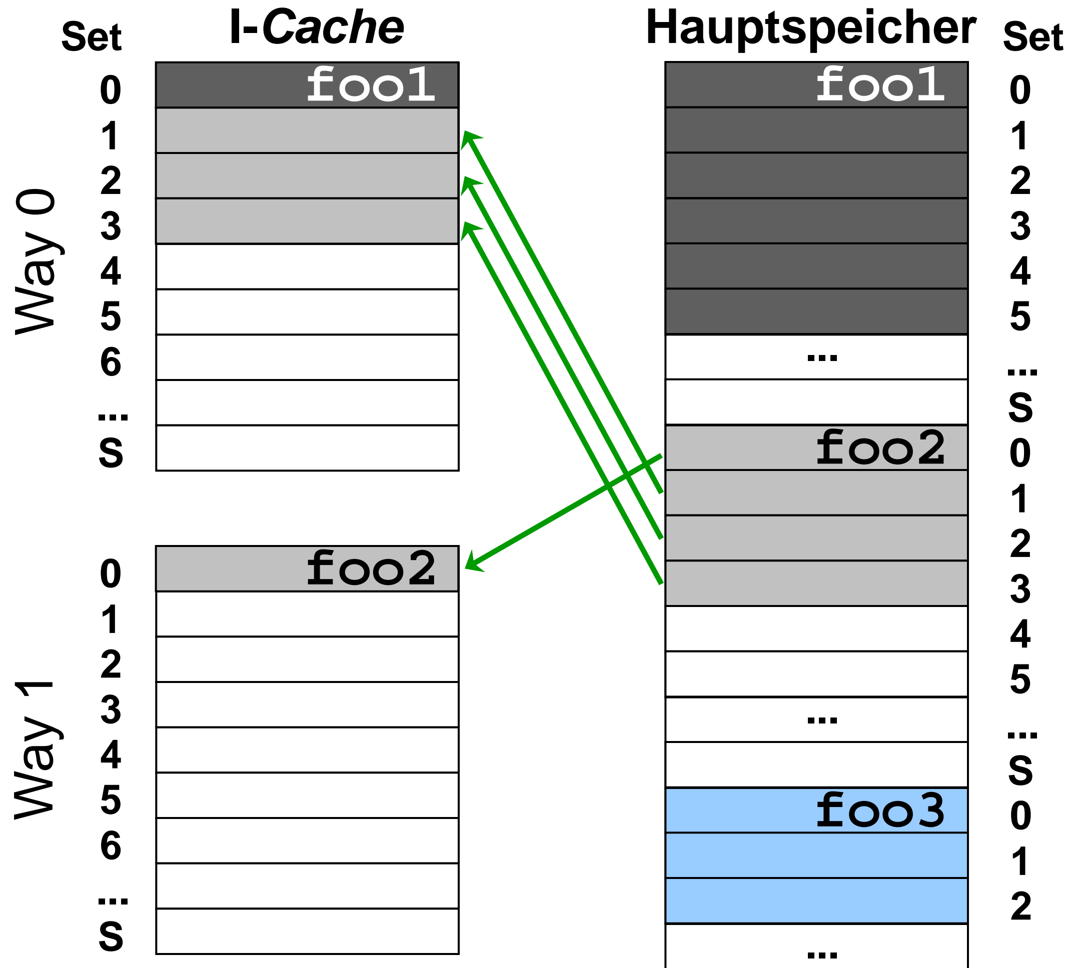
Beispiel für Verdrängung aus Befehls-*Cache* (1)




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

Hier: 2-fach assoziativer I-*Cache*

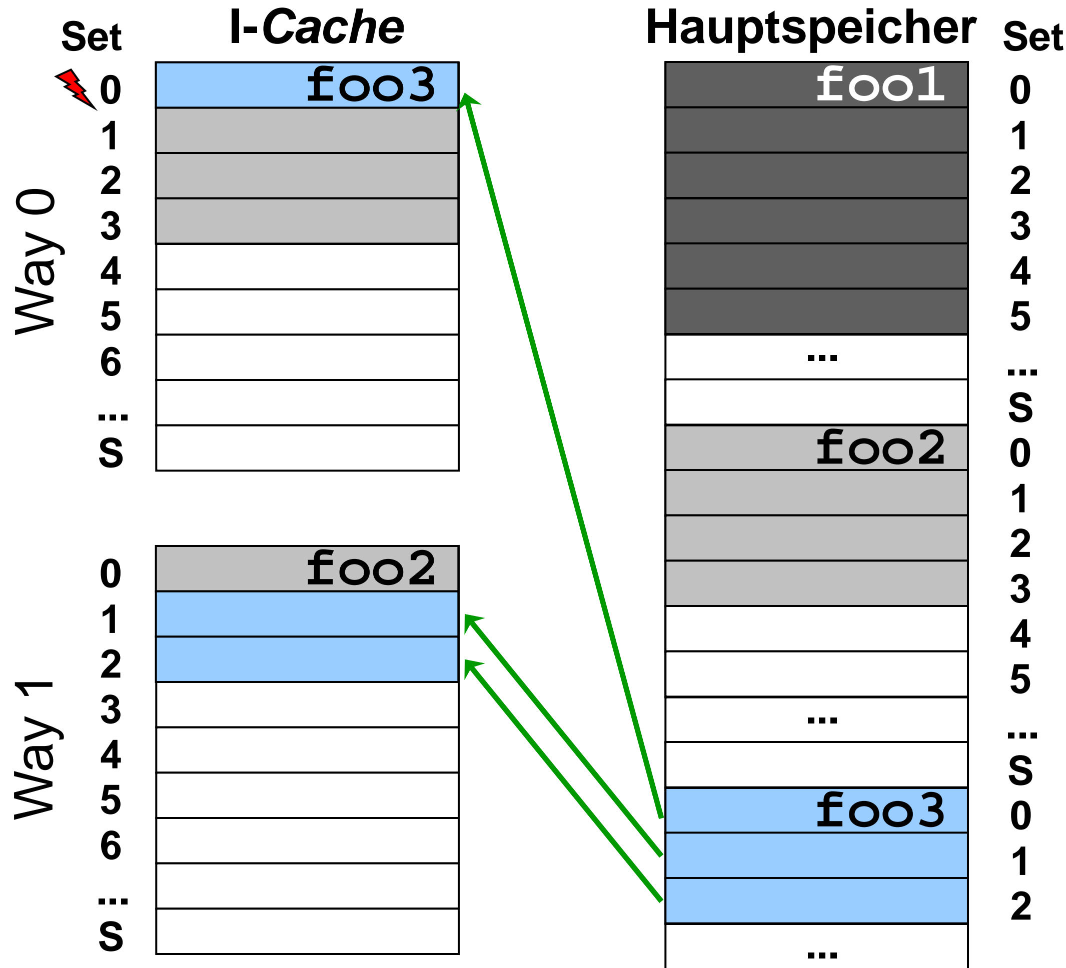
Beispiel für Verdrängung aus Befehls-*Cache* (2)




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

Hier: 2-fach assoziativer I-*Cache*

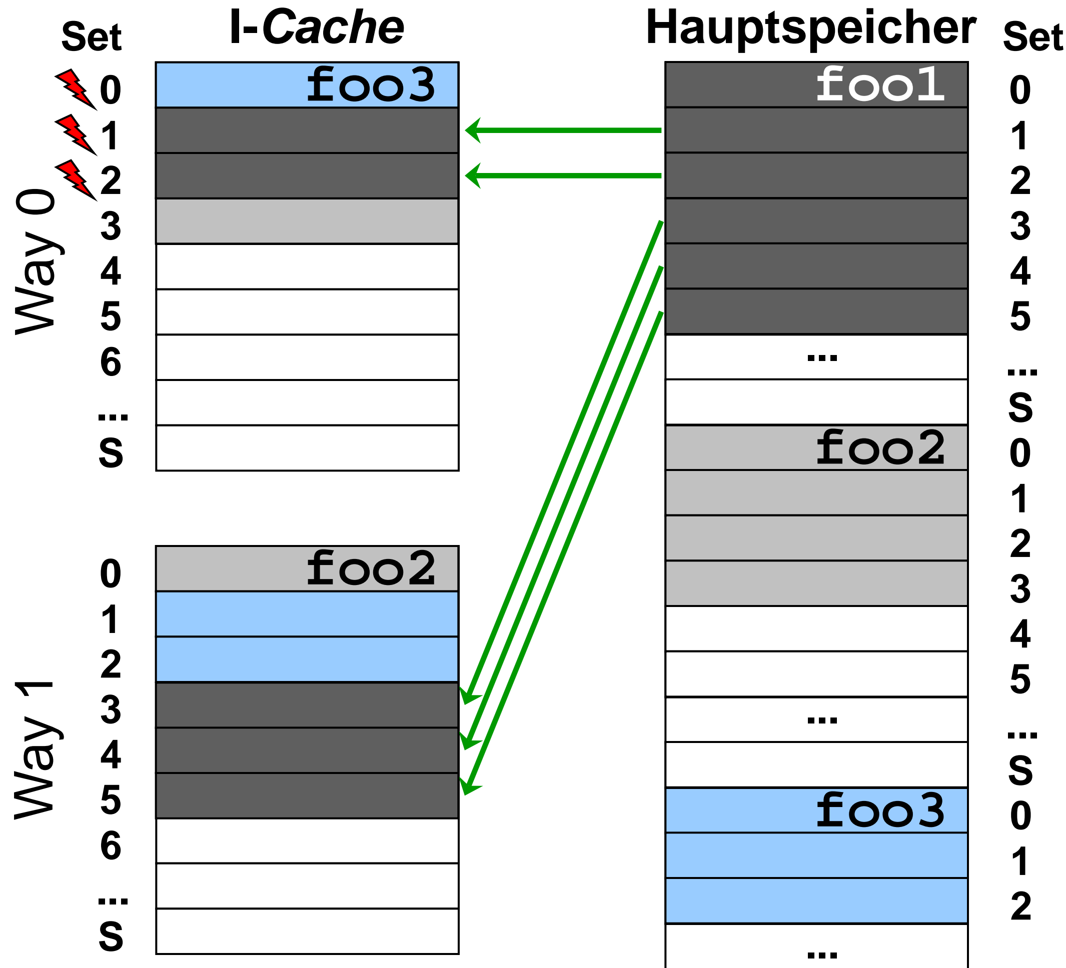
Beispiel für Verdrängung aus Befehls-*Cache* (3)




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

Hier: 2-fach assoziativer I-*Cache*

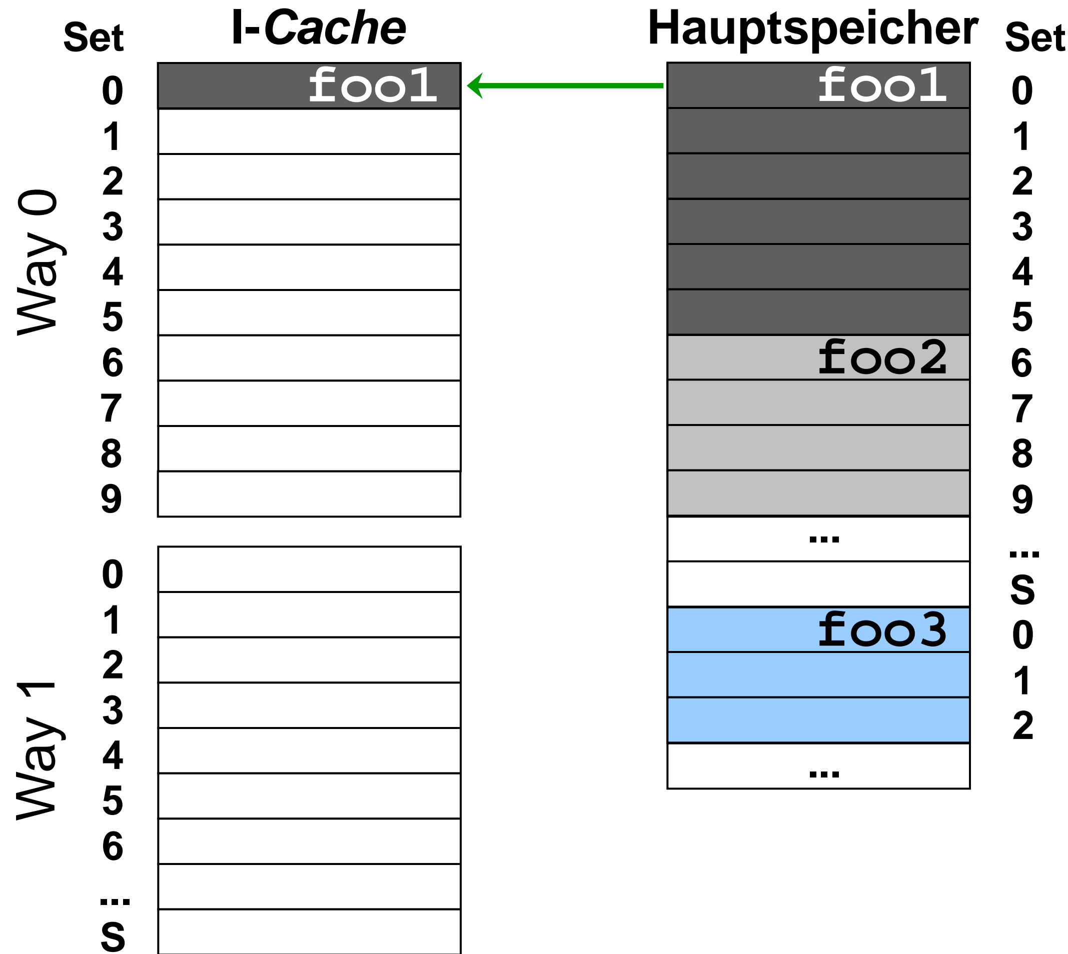
Beispiel für Verdrängung aus Befehls-*Cache* (4)




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

<u>*Hier:*</u> 2-fach assoziativer I-*Cache*

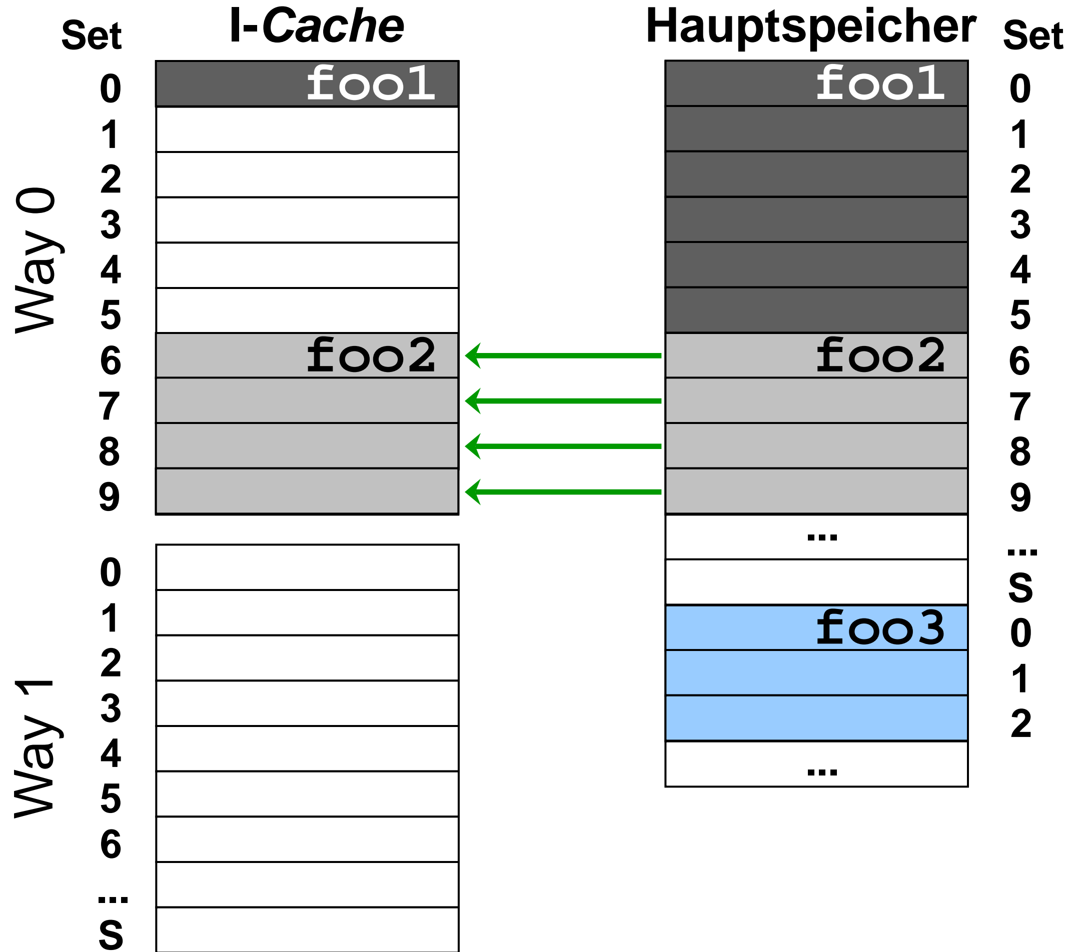
Eine bessere Anordnung ohne Verdrängungen (1)




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

Hier: 2-fach assoziativer I-*Cache*

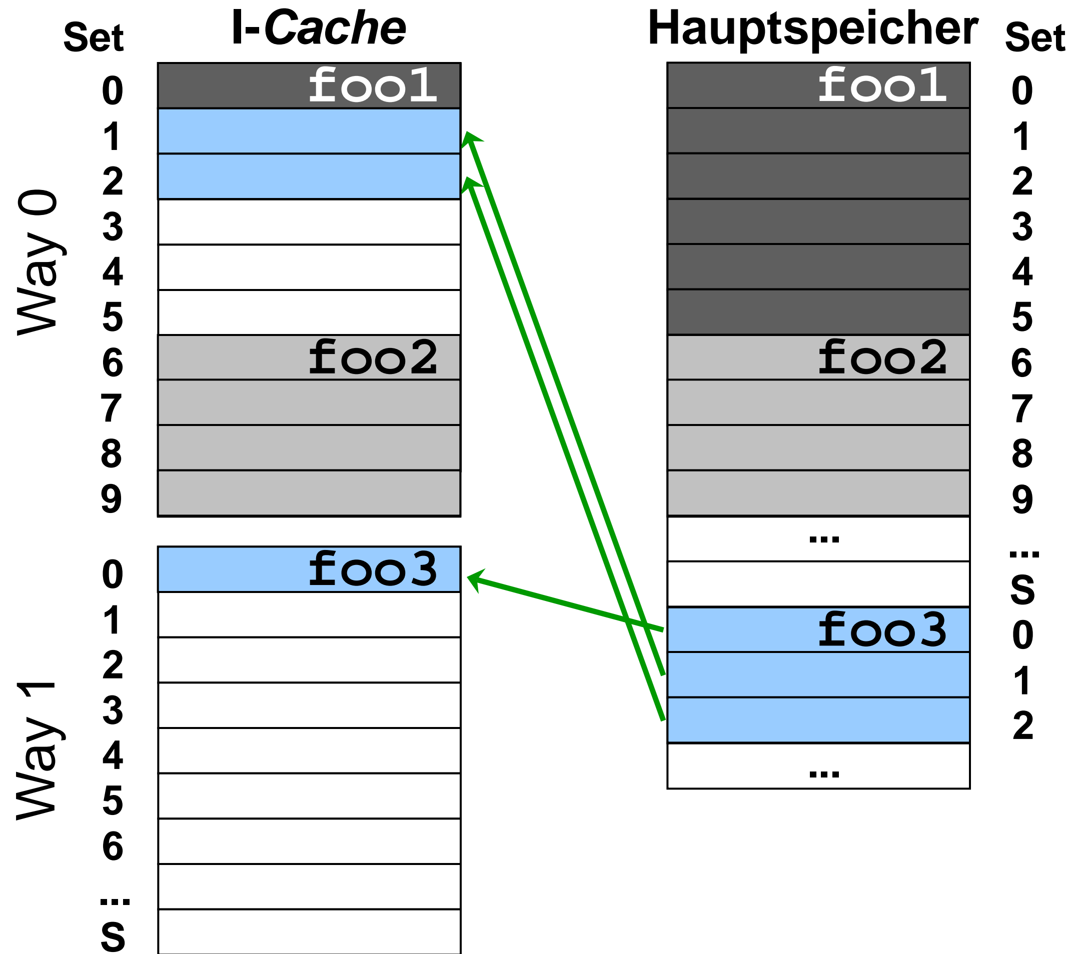
Eine bessere Anordnung ohne Verdrängungen (2)




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

Hier: 2-fach assoziativer I-*Cache*

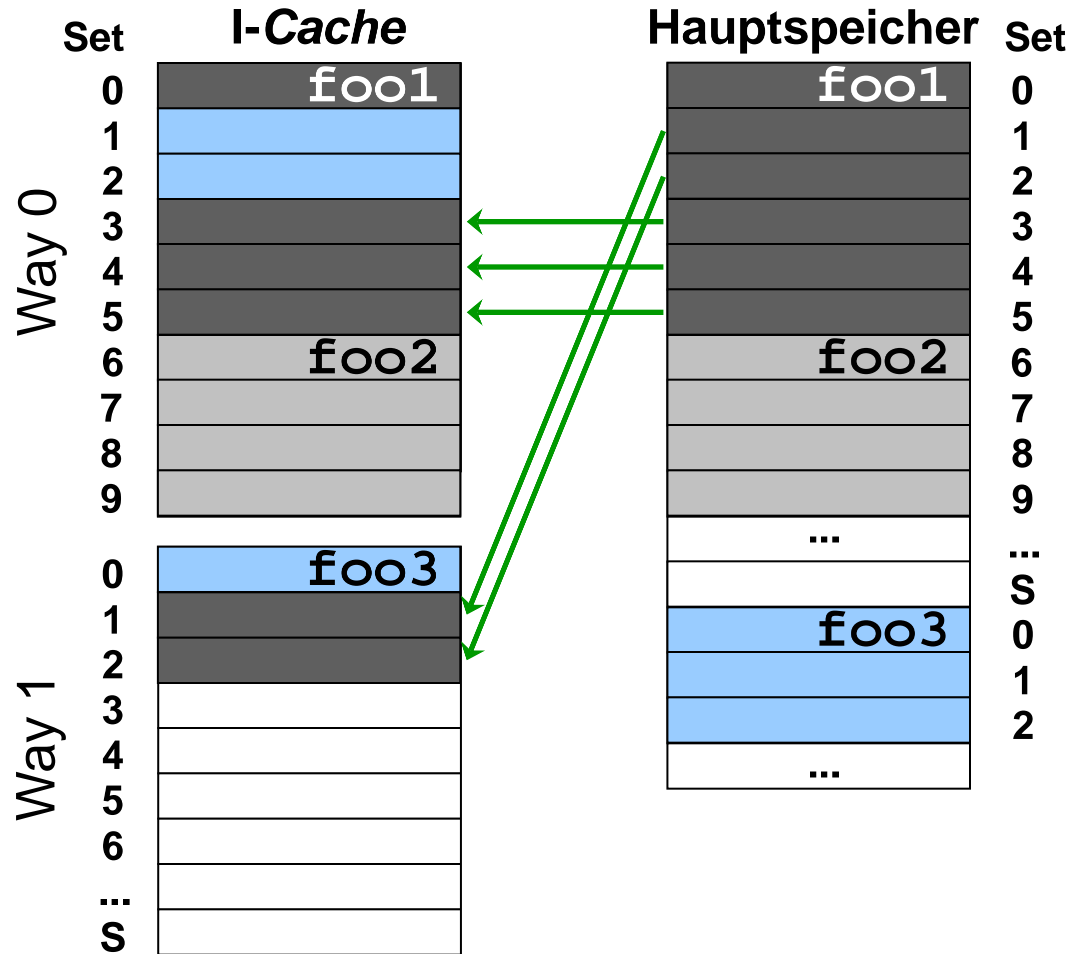
Eine bessere Anordnung ohne Verdrängungen (3)




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

Hier: 2-fach assoziativer I-*Cache*

Eine bessere Anordnung ohne Verdrängungen (4)




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // More 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
- Platziere je zwei Funktionen mit hohem Kantengewicht im Speicher unmittelbar hintereinander

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* $G_{new} = (V_{new}, E_{new}, w_{new}) = G_{orig}$.copy();

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

WCET-bewusstes *Procedure Positioning* (2)

Optimierungsschleife

```
do
  wcet_current = getWCET( P );
  for ( <alle Kanten e = {v, w} ∈ E_new,
           absteigend nach w_new sortiert> )
    if ( Positioning( e, G_new, G_orig, P, wcet_current ) == true )
      // Wenn Platzierung von Knoten v und w hintereinander zu
      // WCET_EST-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* hintereinander im Speicher platziert 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};

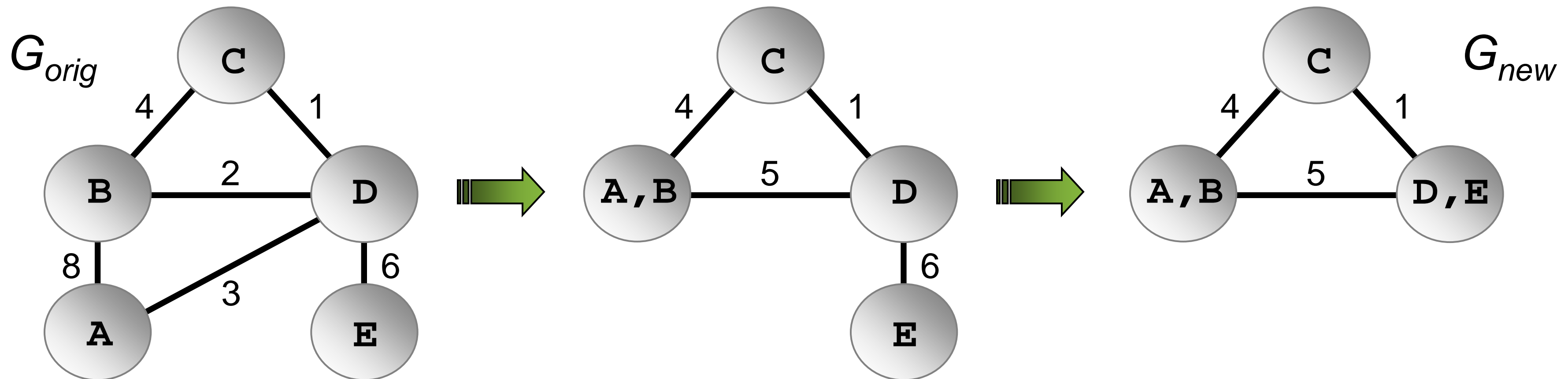
  Aktualisiere w_{new} gemäß neuer WCET-Daten;

  return true;

else

  return false;

Verschmelzen von Knoten



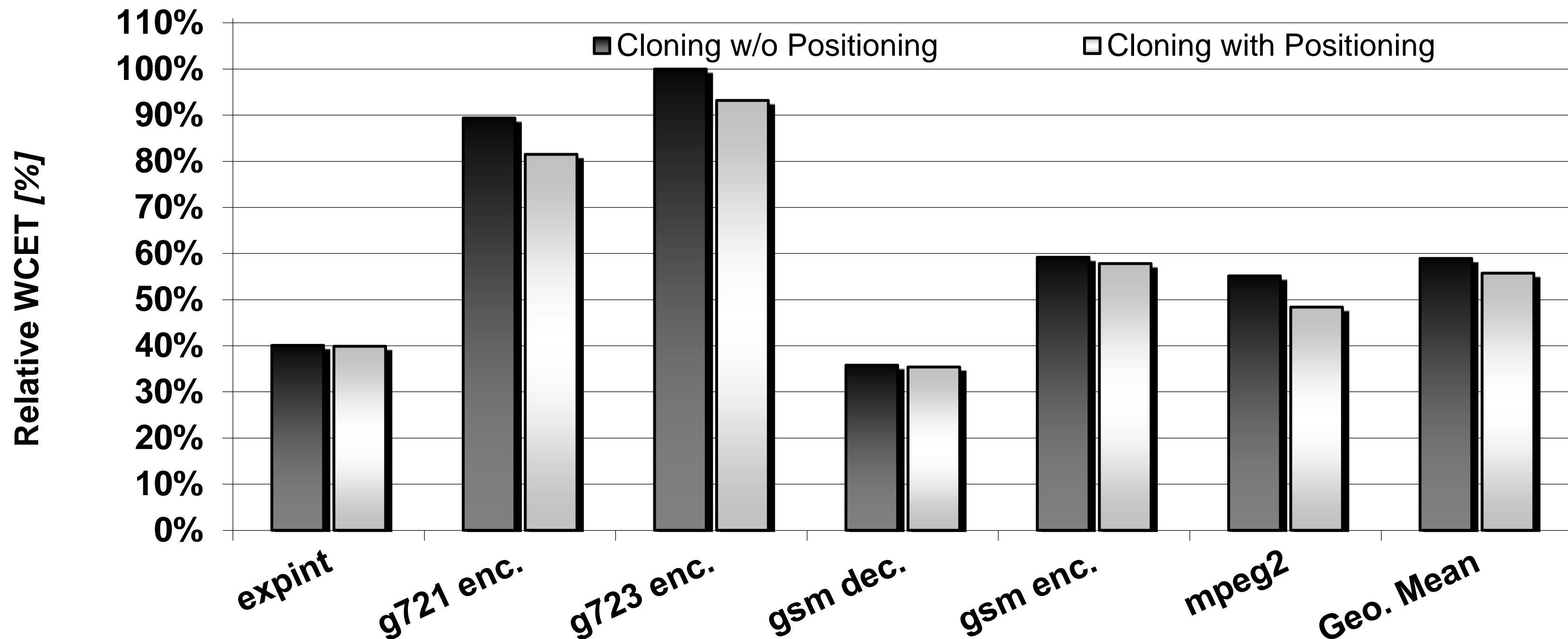
Platzierung verschmolzener Knoten hintereinander im Speicher

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

Eigenschaften

- Algorithmus geht *greedy* vor und platziert jeweils zwei Knoten des aktuellen Graphen G_{new} in einer Iteration hintereinander
- 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 Platzierung 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* und *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* alleine

Vorsicht: *Dieses Diagramm nicht mit Folie 30 vergleichen, da in Folie 30 I-Cache deaktiviert!*

Zwischenfazit

Abgleich mit Konsequenzen für $WCET_{EST}$-Optimierungen

Optimierungen 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*

Inhalte des Kapitels

9. Compiler zur WCET_{EST}-Minimierung

- Einführung
  - Integration eines WCET-Zeitmodells in einen Compiler
  - Herausforderungen bei der WCET-Optimierung
- *Procedure Cloning & Positioning*
  - WCET-bewusstes *Procedure Cloning*
  - *Procedure Positioning* zur Reduktion von *Cache Misses*
- Register-Allokation
  - Problem der Standard Graph-Färbung
  - WCET-bewusste Graph-Färbung
- *Scratchpad*-Allokation von Daten und Code
  - Allokation von globalen Daten
  - Allokation von Basisblöcken

Register-Allokation per Graph-Färbung

1. **Initialisierung:** Erzeuge Interferenzgraphen $G = (V, E)$ 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 *potentiellen 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, 2004]*

Problem der Standard Graph-Färbung

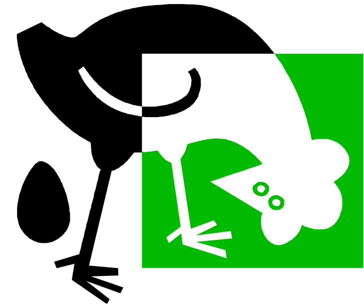
3. ***Spilling:*** Nach Schritt 2 hat jeder Knoten von G Grad $\geq K$. Wähle ein $v \in V$; markiere v als *potentiellen Spill*; entferne v aus G.

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 im Interferenzgraph,
- ... einen Knoten mit hohem Grad, mit vielen Verwendungen, in innerster Schleife, u.U. indem auf *Profiling* zurückgegriffen wird

Ein Henne-Ei-Problem



Eine Register-Allokation zur WCET_{EST}-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 Graph-Färbung auf alle VREGs desjenigen Basisblocks $b \in P$ mit den meisten Ausführungen von *Spill*-Code im *worst case* an
– Ermittle neuen WCEP

WCET-bewusste Graph-Färbung (1)

```
LLIR WCET_GC_RA( LLIR P )
{
  // Iteriere bis aktueller WCEP komplett allokiert.
  while ( true )
  {
    // Kopiere P, alle VREGs von P' werden auf Stack gespillt.
    LLIR 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 Graph-Färbung (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 Graph-Färbung.
    vregs.sort( occurrences of VREG in b );
    traditionalGraphColoring( P, vregs );
  }

  // Wende Standard Graph-Färbung 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 Basisblocks, 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 diejenigen VREGs von *b* gespillt werden, die am seltensten in *b* vorkommen, da wenige Vorkommen wenig *Spill*-Code in *b* bedeuten

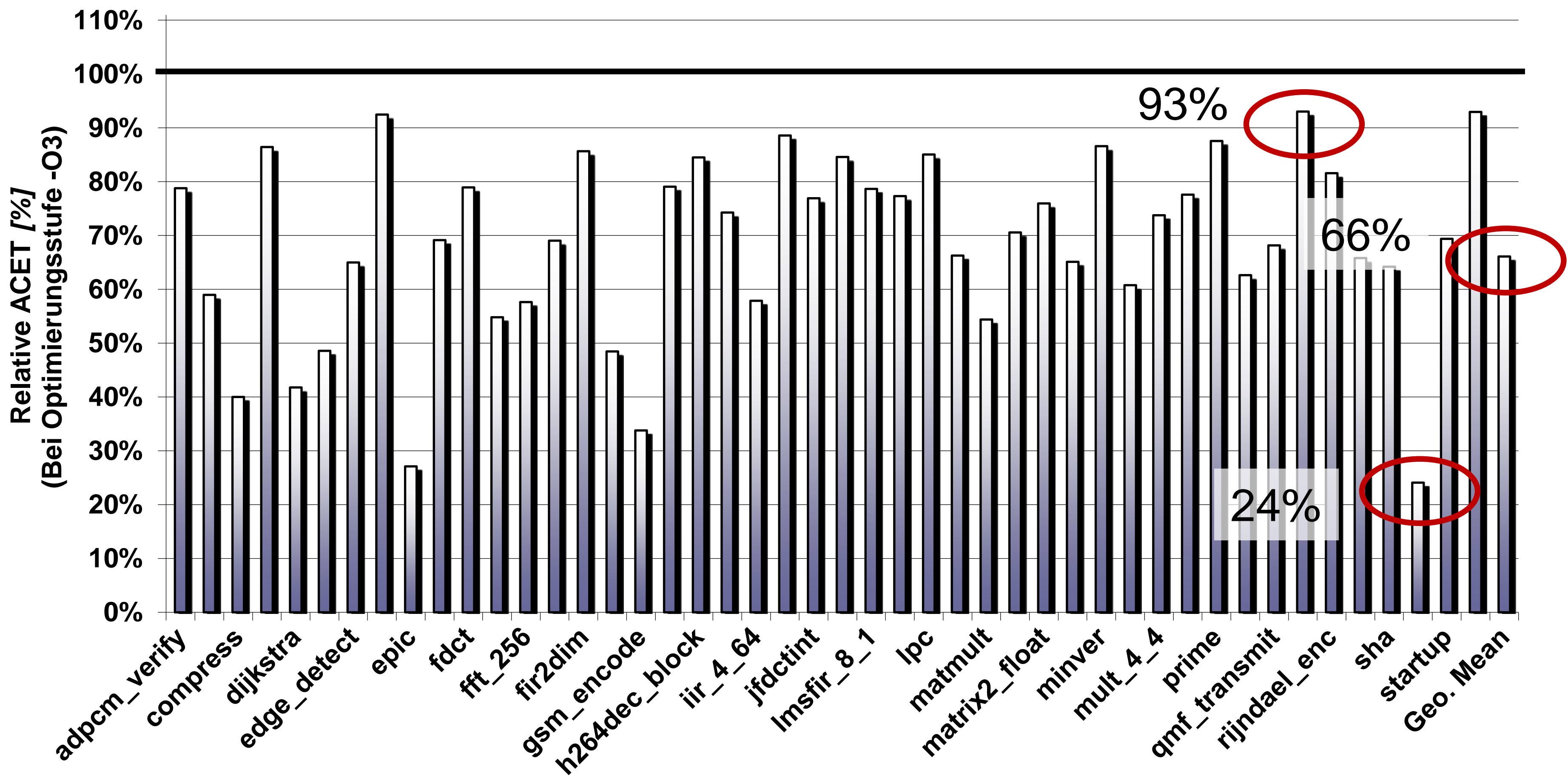
– Register-Allokation und evtl. *Spilling* von *b* wird durch Standard Graph-Färbung vorgenommen

– Nach Abbruch der Allokationsschleife ist WCEP komplett allokiert. Es kann aber noch VREGs in Blöcken abseits des WCEP geben

☞ Abschließende Standard Graph-Färbung, um diese restlichen VREGs zu allokieren

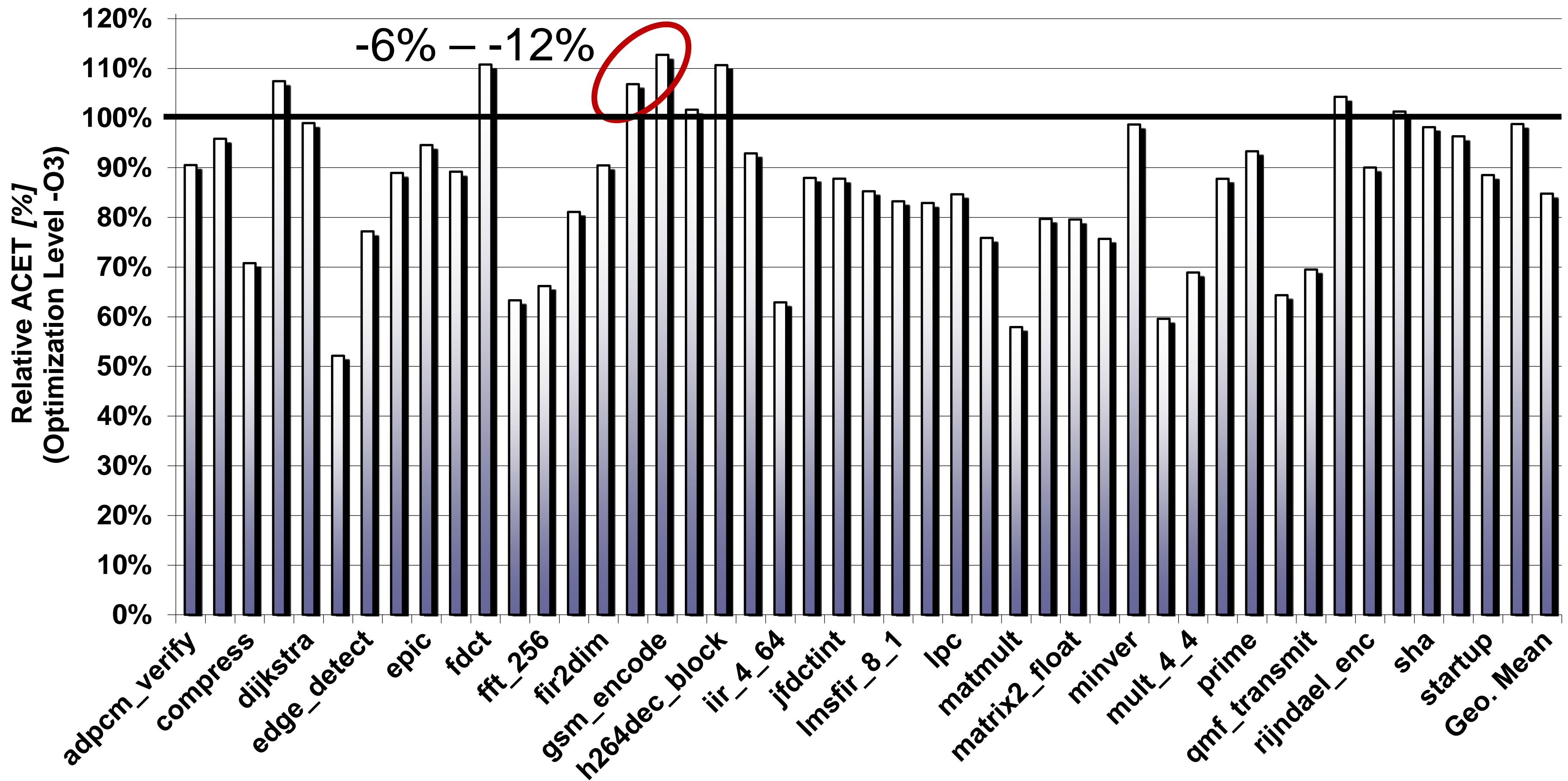
*[**H. Falk,** WCET-aware Register Allocation based on Graph Coloring, DAC, San Francisco, 2009]*

Relative $WCET_{EST}$ nach WCET Graph-Färbung



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

Relative ACET nach WCET Graph-Färbung



100% = ACET mit Standard Graph-Färbung (höchster Grad)

Diskussion

- $WCET_{EST}$-Reduktionen von 6,9% bis 75,9%, im Mittel 33,9%.
- Allokation aller 46 Benchmarks führt 1.979 WCET-Analysen aus.
- Laufzeit der WCET Graph-Färbung: 12:15 Stunden für alle 46 Benchmarks ☞ 16 Minuten pro Benchmark im Schnitt
- ACET-Reduktionen von bis zu 47,9%, aber Verschlechterungen bis zu 12,7%, im Mittel 15,2% ACET-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 Graph-Färbung 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

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

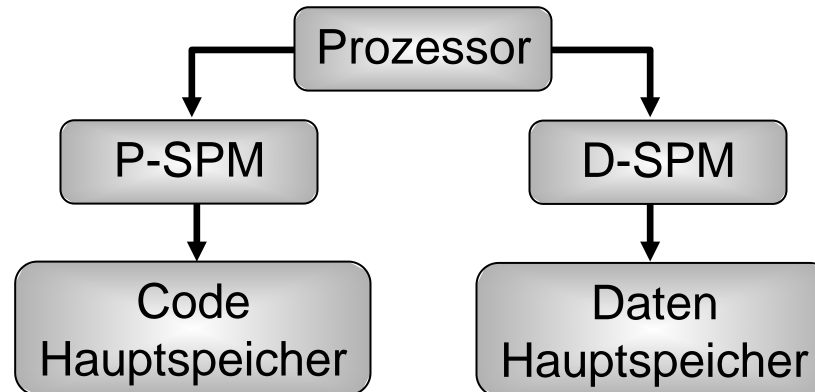
- ... brauchen zwingend Detailwissen über den WCEP
- 👍 *WCET Graph-Färbung berücksichtigt WCEP*
- ... müssen immer berücksichtigen, dass sich der WCEP nach jeder Entscheidung, die eine Optimierung trifft, ändern kann
- 👍 *WCET Graph-Färbung aktualisiert WCEP nach jeder Iteration*
- ... sollten ihre Entscheidungen,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

Inhalte des Kapitels

9. Compiler zur $\mathbf{WCET_{EST}}$-Minimierung

- Einführung
  - Integration eines WCET-Zeitmodells in einen Compiler
  - Herausforderungen bei der WCET-Optimierung
- *Procedure Cloning & Positioning*
  - WCET-bewusstes *Procedure Cloning*
  - *Procedure Positioning* zur Reduktion von *Cache Misses*
- Register-Allokation
  - Problem der Standard Graph-Färbung
  - WCET-bewusste Graph-Färbung
- *Scratchpad*-Allokation von Daten und Code
  - Allokation von globalen Daten
  - Allokation von Basisblöcken

Im Folgenden: Harvard-Architekturen



– Separate Busse und Speicher für Code und Daten
– *Scratchpad*-Allokation von Code und Daten können unabhängig voneinander gelöst werden

☞ Zwei separate ILPs für diese beiden Optimierungen

☞ Non-Harvard Architekturen mit gemeinsamen Bussen und Speichern für Code und Daten: Naheliegende Kombination beider Einzel-ILPs

ILP zur *Scratchpad*-Allokation von Daten

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,
- unter Berücksichtigung wechselnder WCEPs.

Ansatz

- Ganzzahlig-lineare Programmierung

☞ Optimale Resultate

- *Notation:* Großbuchstaben ≅ Konstanten,
  Kleinbuchstaben ≅ Variablen

Entscheidungsvariablen und Kosten

– ***Binäre Entscheidungsvariablen pro Datenobjekt:***

$$y_i = \begin{cases} 1 & \text{falls Datenobjekt } d_i \text{ in } mem_{spm} \\ 0 & \text{falls Datenobjekt } d_i \text{ in } mem_{main} \end{cases}$$

– ***Kosten eines Basisblocks b_j:***

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

c_j modelliert die WCET_{EST} von b_j, abhängig davon ob die Datenobjekte, auf die b_j zugreift, im SPM oder im Hauptspeicher liegen

C_j: b_j's WCET_{EST} wenn alle Datenobjekte im Hauptspeicher liegen

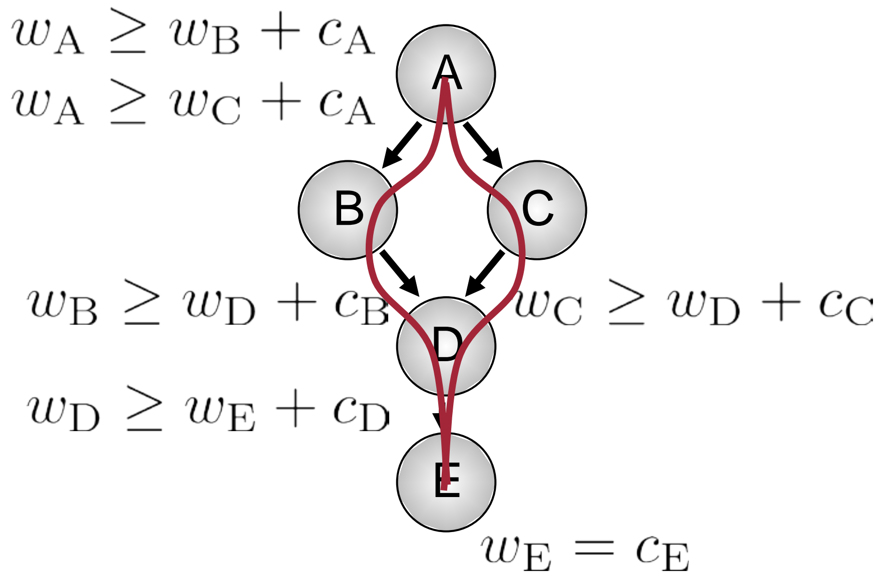
$G_{i,j}$: WCET-Reduktion von b_j, wenn Datenobjekt d_i im SPM allokiert wird

Intraprozeduraler Kontrollfluss (1)

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

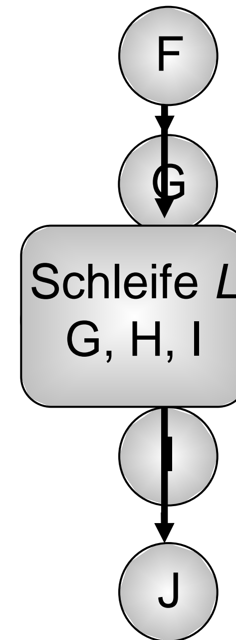
– ***Modellierung des Kontrollflusses einer Funktion:***

<u>Azyklische Teilgraphen:</u>



w_A = WCET des <u>*längsten*</u> in A startenden Pfades

<u>(Reduzierbare) Schleifen:</u>



– Innerste Schleifen L wie azyklische Graphen bearbeiten
– Schleife L falten
– Kosten von L:
$c_L = w_G * C_{max}^L$
– Iteriere mit nächsten innersten Schleifen

Intraprozeduraler Kontrollfluss (2)

- ***Modellierung des Kontrollflusses einer Funktion:***
  - 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

Intraprozeduraler Kontrollfluss (3)

- ***Modellierung des Kontrollflusses einer Funktion:***
  - Reduzierbare Schleifen *L* haben genau einen Eintrittspunkt b^L_{entry}
  - Durch „Ausblenden" der Rücksprungkante einer reduzierbaren Schleife *L* wird der CFG des Schleifenkörpers azyklisch
  - Erzeuge Nebenbedingungen für azyklischen Schleifenkörper wie auf Folie 65 links
  - ☞ Variable w^L_{entry} modelliert $WCET_{EST}$ des kompletten Schleifenkörpers von *L*, wenn dieser *genau einmal* ausgeführt wird
  - Multiplikation von w^L_{entry} mit der max. Iterationszahl C^L_{max} von *L* liefert $WCET_{EST}$ für *alle Ausführungen* der Schleife
  - ☞ Kosten von *L* gleich der $WCET_{EST}$ von *L* für alle Ausführungen

Interprozeduraler Kontrollfluss

- ***Modellierung von Funktionsaufrufen:***
  - Jede Funktion F hat eindeutigen Startknoten b_{entry}^{F}
  - Variable w_{entry}^{F} modelliert WCET$_{EST}$ des längsten Pfades in F, der in b_{entry}^{F} startet
  - ☞ w_{entry}^{F} modelliert WCET$_{EST}$ von F für exakt 1 Ausführung von F
  - ☞ Ruft F' Funktion F auf: addiere w_{entry}^{F} zu WCET$_{EST}$ von F'

- ***Funktionsaufrufe in Basisblock b_j:***
  - „Aufrufstrafe" für aufrufenden Basisblock:

$$cp_j = \begin{cases} w_{entry}^{F} & \text{falls } b_j \ F \text{ aufruft} \\ 0 & \text{sonst} \end{cases}$$

  - ILP-Nebenbedingung pro Basisblock:

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

Scratchpad-Kapazität und Zielfunktion

- ***Scratchpad-Kapazitätsbeschränkung:***

$$\sum_{\text{Datenobjekte } d_i} (S_i * y_i) \leq S_{spm}$$

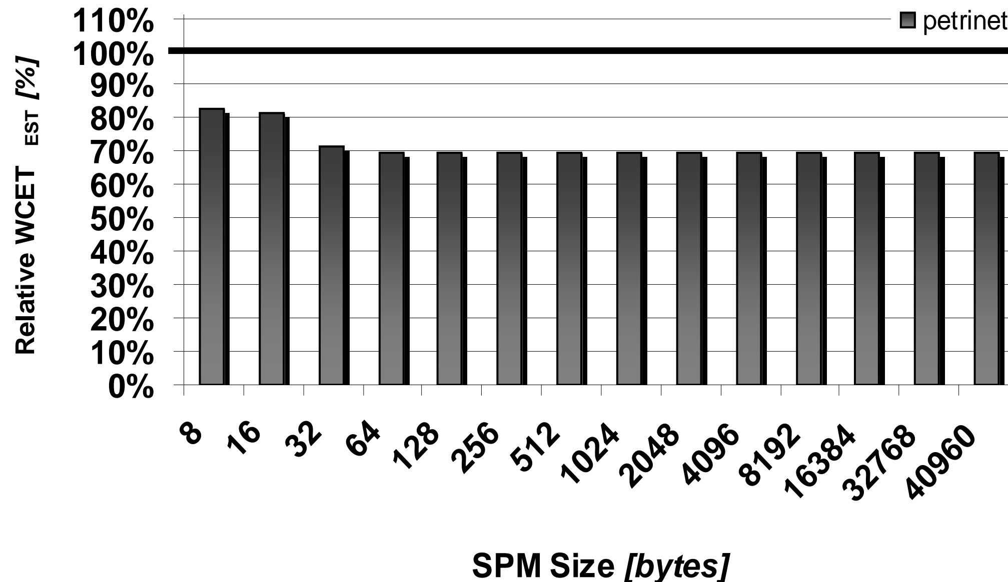
Die Summe der Größen aller auf dem SPM platzierten Datenobjekte muss kleiner gleich der verfügbaren SPM-Kapazität sein

- ***Zielfunktion:***
  - w_{entry}^{F} modelliert WCET_{EST} von F wenn F einmal ausgeführt wird
  - Variable $w_{entry}^{\texttt{main}}$ modelliert WCET_{EST} des gesamten Programms

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

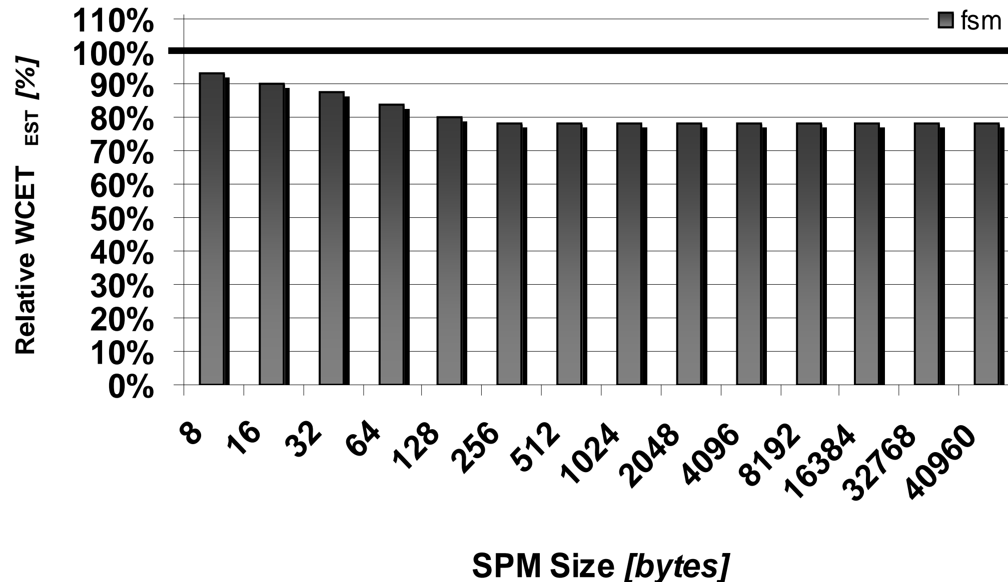
*[**F. Rotthowe,** Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung, Diplomarbeit, TU Dortmund, 2008]*

Rel. $WCET_{EST}$ nach D-SPM-Allokation für `petrinet`



- Beträchtliche $WCET_{EST}$-Reduktionen bereits für sehr kleine SPMs
- 6 globale Daten von insgesamt 72 Bytes Größe
- $WCET_{EST}$-Reduktionen um 28,6% für 32 Bytes SPM
- X-Achse: Absolute SPM-Größen
- Y-Achse: 100% = $WCET_{EST}$ ohne Daten-*Scratchpad*

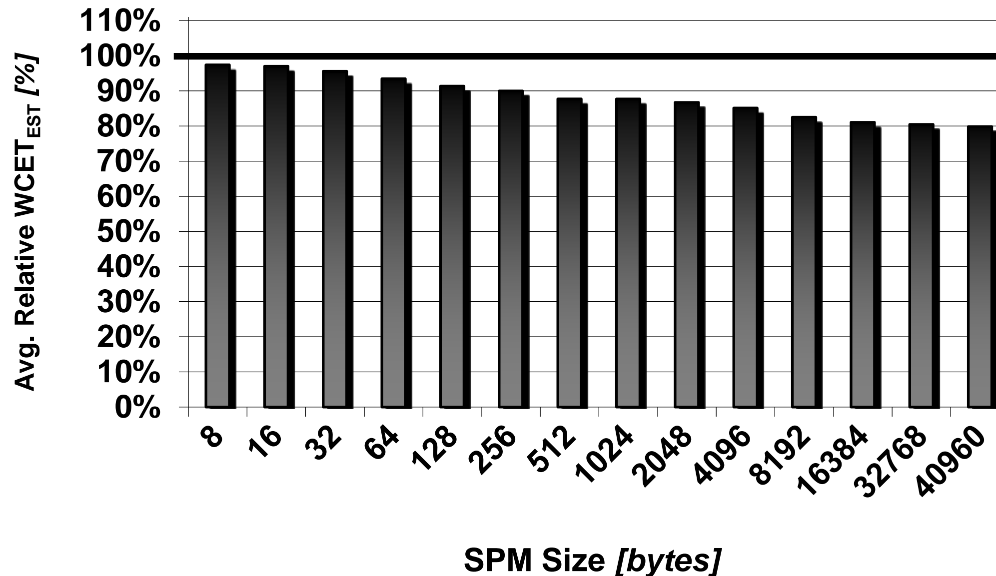
Rel. $WCET_{EST}$ nach D-SPM-Allokation für `fsm`



- Stetige $WCET_{EST}$-Reduktionen mit zunehmenden SPM-Größen
- 98 globale Variablen à 4 Bytes Größe
- $WCET_{EST}$-Reduktionen um 21,4% für 256 Bytes SPM

- X-Achse: Absolute SPM-Größen
- Y-Achse: 100% = $WCET_{EST}$ ohne Daten-*Scratchpad*

Rel. $WCET_{EST}$ nach D-SPM-Allokation für 14 Benchmarks



- Stetige $WCET_{EST}$-Reduktionen mit zunehmenden SPM-Größen
- $WCET_{EST}$-Reduktionen von 2,7% – 20,6%
- X-Achse: Absolute SPM-Größen
- Y-Achse: 100% = $WCET_{EST}$ ohne Daten-*Scratchpad*

ILP zur *Scratchpad*-Allokation von Programmcode

Ziel

- Bestimme Menge von Basisblöcken zur Allokation in Programm-SPM,
- so dass ausgewählte Basisblöcke $WCET_{EST}$ -Minimierung erzielen
- unter Berücksichtigung wechselnder WCEPs.

Ansatz

- Ganzzahlig-lineare Programmierung

☞ Optimale Resultate

- *Notation:* Großbuchstaben ≅ Konstanten,
  Kleinbuchstaben ≅ Variablen

Entscheidungsvariablen und Kosten

– ***Binäre Entscheidungsvariablen pro Basisblock:***

$$x_i = \begin{cases} 1 & \text{falls Basisblock } b_i \text{ in } mem_{spm} \\ 0 & \text{falls Basisblock } b_i \text{ in } mem_{main} \end{cases}$$

– ***Kosten eines Basisblocks b_i:***

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

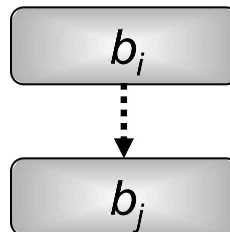
c_i modelliert die WCET_{EST} von b_i wenn b_i entweder im Hauptspeicher oder im SPM liegt

– ***Modellierung des intraprozeduralen Kontrollflusses:***
Wie zuvor zur WCET-bewussten SPM-Allokation von Daten

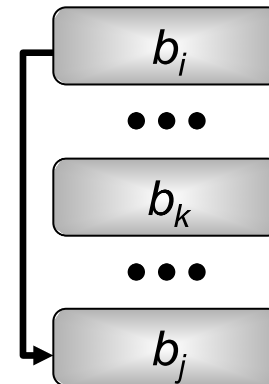
Sprünge zwischen Speichern

- ***Allokation aufeinander folgender Basisblöcke:***
  - Allokation aufeinander folgender Basisblöcke in verschiedene Speicher erfordert Einfügen spezieller Sprung-Befehle
  - Sprünge zwischen Speichern sehr teuer: mehr als 1 Instruktion
  - Sprung-Befehle: *Variablen* im ILP bzgl. WCET_{EST} und Code-Größe, abhängig von Entscheidungsvariablen *(☞ siehe Kapitel 7)*

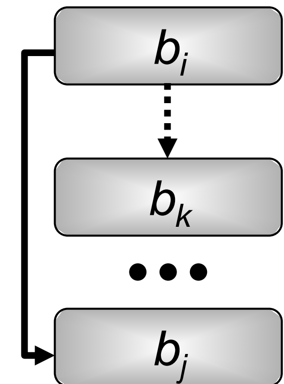
- ***Sprung-Szenarien: („Jump Scenarios", JS)***



a) Implizit



b) Unbedingt



c) Bedingt

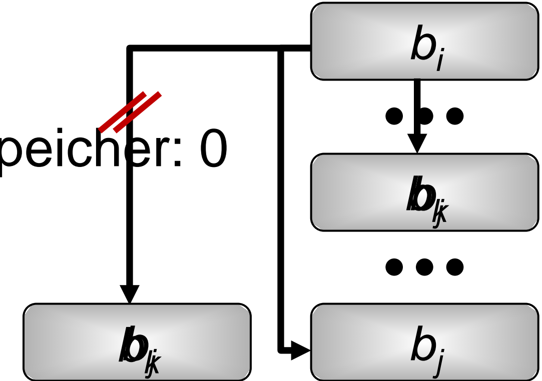
Strafen für Sprünge zwischen Speichern

- ***Sprung-Strafe – „Jump Penalty“ (⊗ ≅ Boolesches XOR):***
  - *Strafe für implizite Sprünge:* jp^{i}_{impl}

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

    Hohe Strafe, falls BBs *i* und *j* in verschiedenen Speichern liegen

  - *Strafe für unbedingte Sprünge:* jp^{i}_{uncond}
    - Falls b_i und b_j in versch. Speichern: P_{high}
    - Falls b_i und b_j nebeneinander im gleichen Speicher: 0
    - Falls b_i und b_j nicht nebeneinander im gleichen Speicher: P_{low}



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

  - *Bedingte Sprünge:* Kombination von jp^{i}_{impl} und jp^{i}_{uncond}

Sprung-Strafen und interprozeduraler Kontrollfluss

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

$$jp_i = \begin{cases} jp^i_{impl} & \text{wenn Sprung-Szenario von } b_i \text{ } \textit{implizit} \\ jp^i_{uncond} & \text{wenn Sprung-Szenario von } b_i \text{ } \textit{unbedingt} \\ jp^i_{cond} & \text{wenn Sprung-Szenario von } b_i \text{ } \textit{bedingt} \\ 0 & \text{sonst} \end{cases}$$

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

$$cp_i = \begin{cases} w^F_{entry} + (x_i \otimes x^F_{entry}) * P_{high} & \text{falls } b_i \text{ } F \text{ aufruft} \\ \quad +(1 - (x_i \otimes x^F_{entry})) * P_{low} & \\ 0 & \text{sonst} \end{cases}$$

Ruft Block b_i Funktion F auf: Addiere WCET$_{EST}$ von F zu WCET$_{EST}$ von b_i. Addiere zusätzlich P_{high}, wenn Funktionsaufruf über Speicher hinweg springt. Wenn Funktionsaufruf im gleichen Speicher bleibt, addiere P_{low}.

Basisblock-Größen

- ***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:

  $$s_i = \begin{cases} (x_i \wedge \overline{x_j}) * S_{impl} & \text{wenn Sprung-Szenario von } b_i \ \textit{implizit} \\ (x_i \wedge \overline{x_j}) * S_{uncond} & \text{wenn Sprung-Szenario von } b_i \ \textit{unbedingt} \\ (x_i \wedge \overline{x_k}) * S_{impl} + & \text{wenn Sprung-Szenario von } b_i \ \textit{bedingt} \\ \quad (x_i \wedge \overline{x_j}) * S_{uncond} & \\ (x_i \wedge \overline{x^F_{entry}}) * S_{call} & \text{wenn } b_i \ F \text{ aufruft} \\ 0 & \text{sonst} \end{cases}$$

  - Gesamtgröße von Basisblock b_i:
    Größe S_i von b_i ohne jeglichen Sprung-Code <u>*plus*</u>
    Größe s_i des Sprung-Codes von b_i

Scratchpad-Kapazität und Zielfunktion

- ***Scratchpad-Kapazitätsbeschränkung:***

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

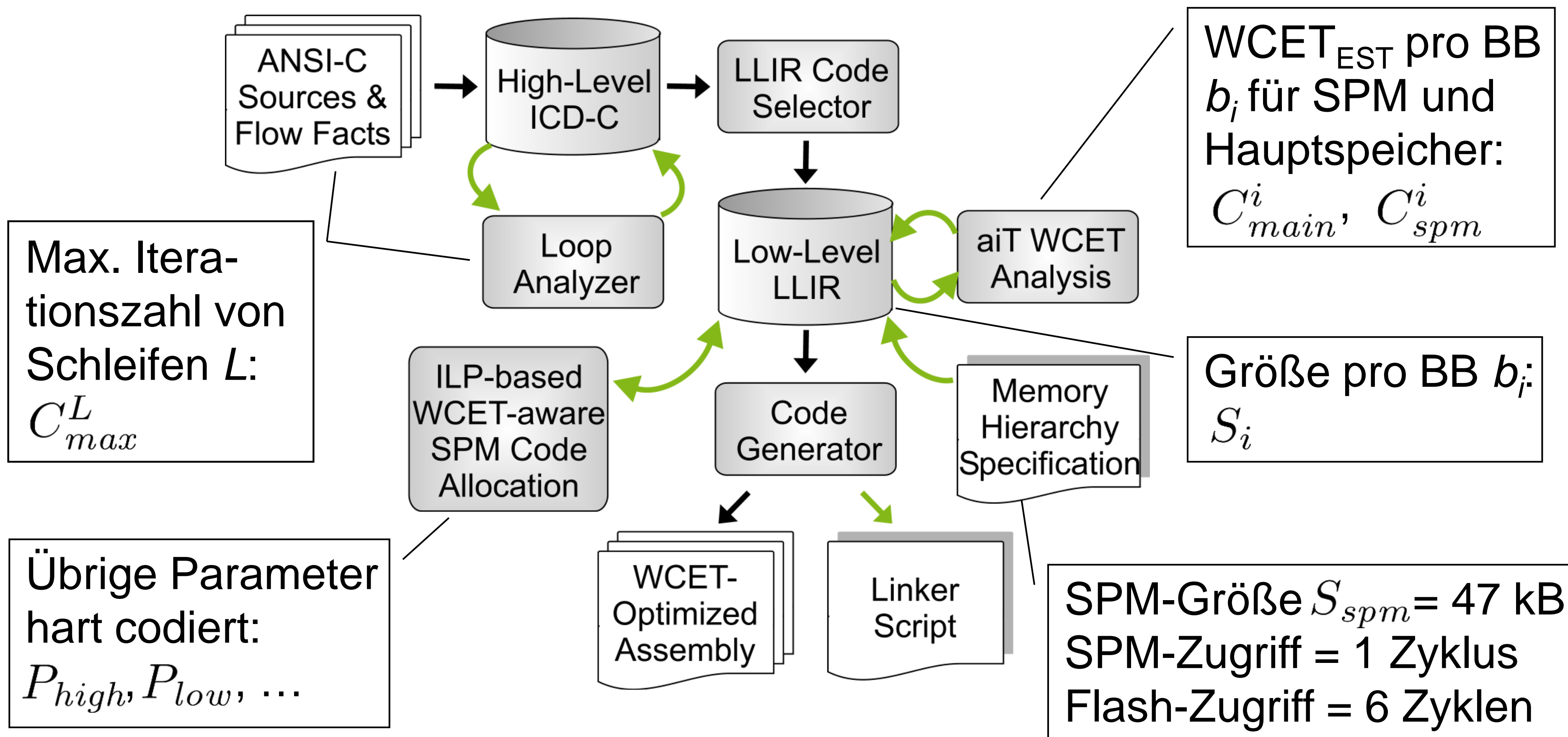
  Die Summe der Größen aller auf dem SPM platzierten Basisblöcke ohne Sprungcode plus Größe des Sprung-Codes von b_i muss kleiner gleich der verfügbaren SPM-Kapazität sein

- ***Zielfunktion:***

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

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

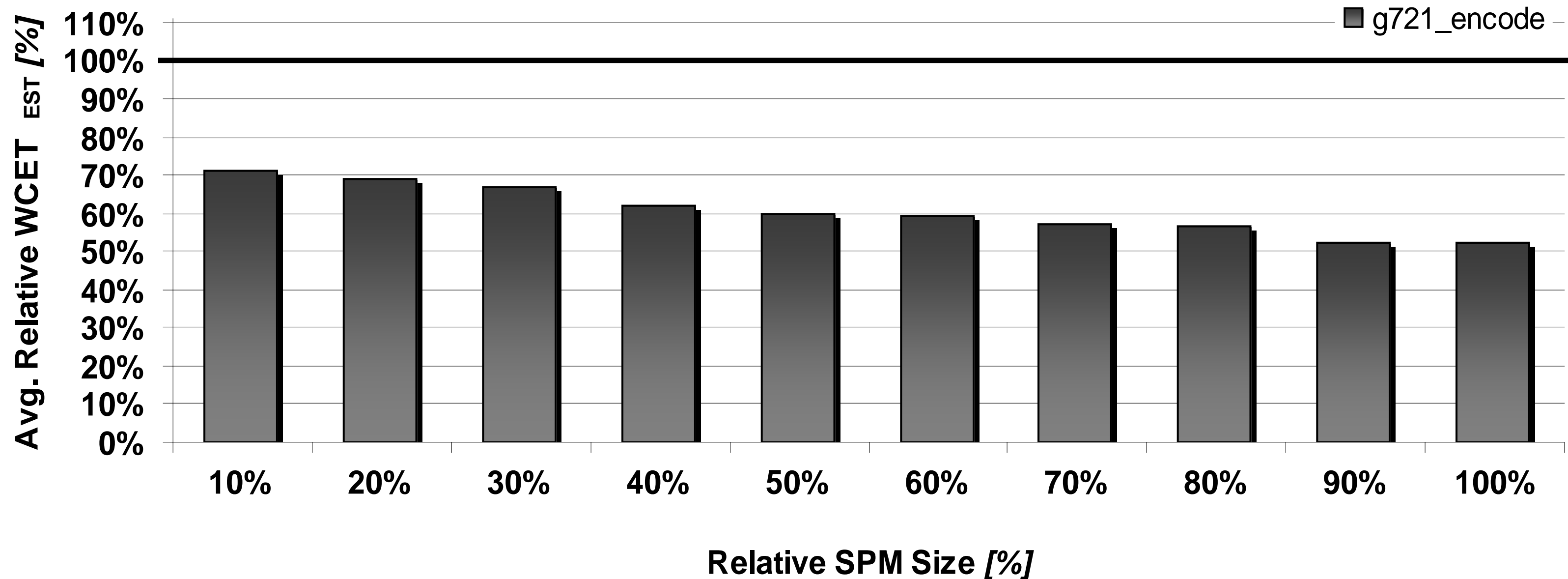
Bestimmung der Konstanten für die ILPs (1)



Bestimmung der Konstanten für die ILPs (2)

- *WCET*

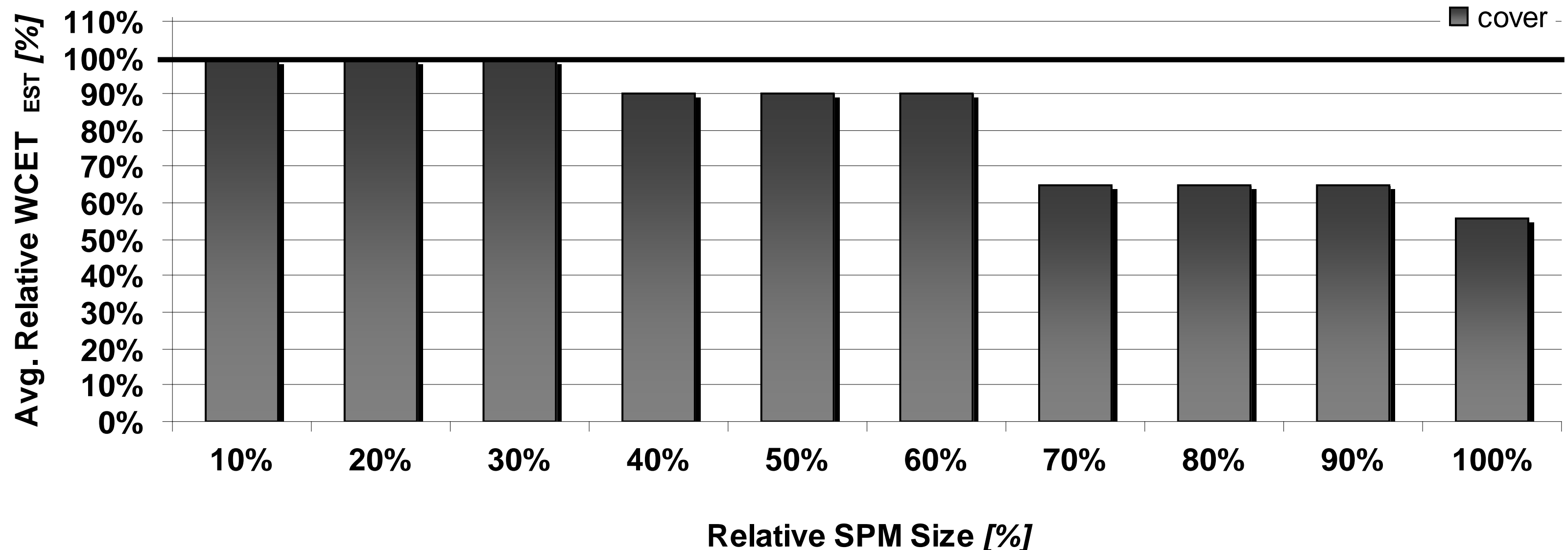
Rel. $WCET_{EST}$ nach P-SPM-Allokation für `g721 encoder`



- Stetige $WCET_{EST}$-Reduktionen für zunehmende SPM-Größen
- $WCET_{EST}$-Reduktionen von 29% – 48%

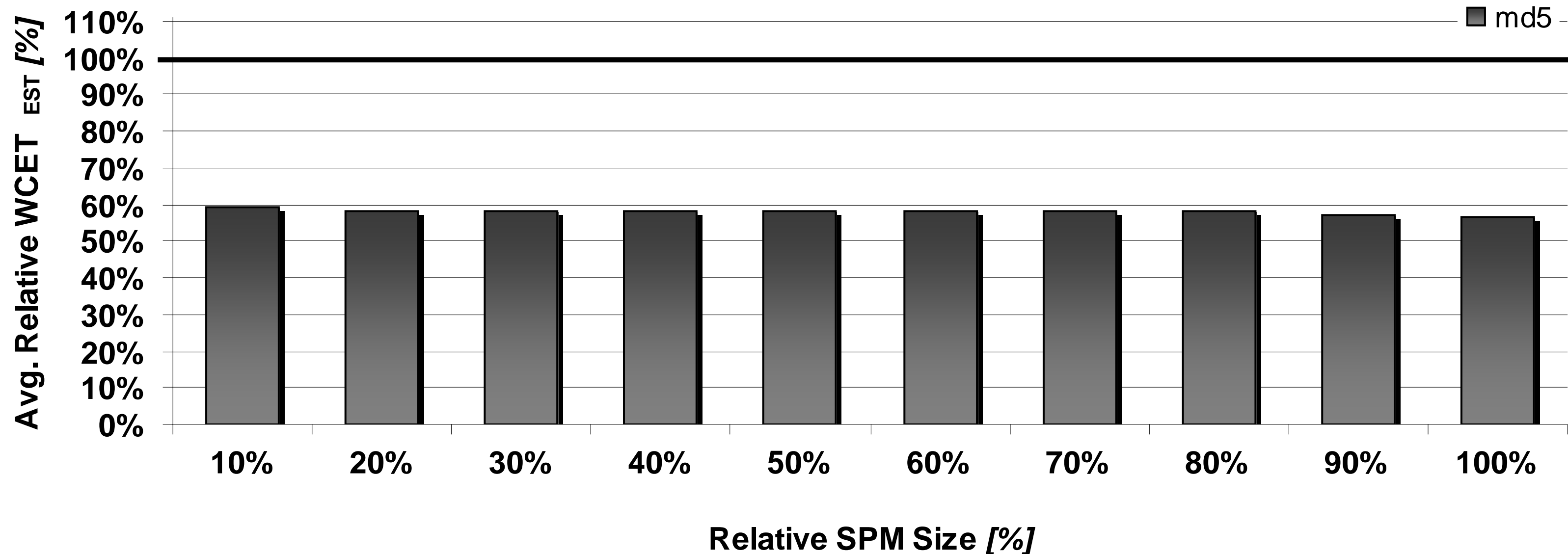
- X-Achse: SPM-Größe = *x*% der Code-Größe des Benchmarks
- Y-Achse: 100% = $WCET_{EST}$ ohne Code-*Scratchpad*

Rel. $WCET_{EST}$ nach P-SPM-Allokation für `cover`



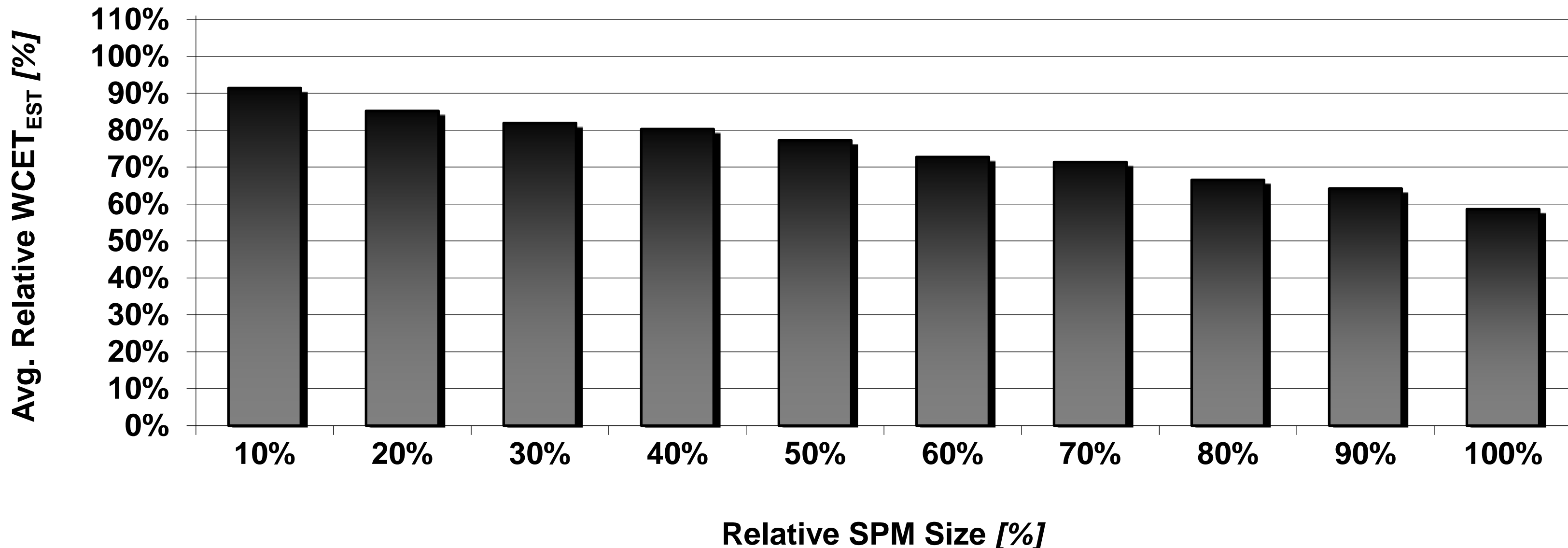
- Schrittweise $WCET_{EST}$-Reduktionen: vorteilhafter Code nur bei 40%, 70% und 100% rel. SPM-Größe auf *Scratchpad* allokiert
- $WCET_{EST}$-Reduktionen von 10%, 35% und 44%
- X-Achse: SPM-Größe = *x*% der Code-Größe des Benchmarks
- Y-Achse: 100% = $WCET_{EST}$ ohne Code-*Scratchpad*

Rel. $WCET_{EST}$ nach P-SPM-Allokation für `md5`



- Gleiche $WCET_{EST}$-Reduktionen für alle SPM-Größen: 40% - 44%
- ILP findet zielsicher kleinen aber zeitkritischen Hot Spot von **`md5`** und allokiert diesen auf *Scratchpad*
- X-Achse: SPM-Größe = *x*% der Code-Größe des Benchmarks
- Y-Achse: 100% = $WCET_{EST}$ ohne Code-*Scratchpad*

Rel. $WCET_{EST}$ nach P-SPM-Allokation für 73 Benchmarks



- Stetige $WCET_{EST}$-Reduktionen mit zunehmenden SPM-Größen
- $WCET_{EST}$-Reduktionen von 8% – 41%
- X-Achse: SPM-Größe = *x*% der Code-Größe der Benchmarks
- Y-Achse: 100% = $WCET_{EST}$ ohne Code-*Scratchpad*

Zwischenfazit

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

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

– ... brauchen zwingend Detailwissen über den WCEP

👍 *SPM-Allokationen berücksichtigen WCEP*

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

👍 *SPM-Allokationen erfassen WCEP-Änderungen inhärent im ILP*

– ... 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

WCC-Infrastruktur

– H. Falk, P. Lokuciejewski. *A compiler framework for the reduction of worst-case execution times*. Springer Real-Time Systems 46(2), Oktober 2010.

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; Graph-Färbung entlang des stets aktuellen WCEP
- *Scratchpad*-Allokationen: inhärente WCEP-Modellierung in ILPs; daher keine wiederholten WCET-Analysen nötig