ulm university universität ulm

Compiler für Eingebettete Systeme

[CS7506]

Sommersemester 2014

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm

ulm university universität ulm

Kapitel 5

HIR Optimierungen und Transformationen

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

5. HIR Optimierungen und Transformationen

- Motivation
- *Function Specialization / Procedure Cloning*
  - Allzweck-Funktionen in speziellen Kontexten
  - Einschub: Standard-Optimierungen & WCET-Abschätzung
  - *Function Specialization* und WCETs
  - Ergebnisse ($WCET_{EST}$, ACET, Codegröße)
- Parallelisierung für Homogene Multi-DSPs
  - Einführung, Multi-DSP Architekturen
  - Programm-Wiederherstellung
  - Daten-Partitionierung & *Strip Mining*
  - Parallelisierung
  - Speicher-Aufteilung, Feld-Deskriptoren, DMA
  - Ergebnisse

Motivation von HIR Optimierungen

***High-Level* IRs:**

– Sehr eng an Quellsprache angelehnt

– *High-Level* Sprachkonstrukte (insbes. Schleifen, Funktionsaufrufe mit Parameterübergabe, *Array*-Zugriffe) bleiben erhalten

***High-Level* Optimierungen**

– Nutzen Features der HIR stark aus

– Konzentrieren sich auf starkes Umstrukturieren von Schleifen und Funktionsstruktur

– Sind auf niedriger Ebene nur schwer zu realisieren, da *high-level* Informationen erst wieder rekonstruiert werden müssten

Inhalte des Kapitels

5. HIR Optimierungen und Transformationen

- Motivation
- *Function Specialization / Procedure Cloning*
  - Allzweck-Funktionen in speziellen Kontexten
  - Einschub: Standard-Optimierungen & WCET-Abschätzung
  - *Function Specialization* und WCETs
  - Ergebnisse ($WCET_{EST}$, ACET, Codegröße)
- Parallelisierung für Homogene Multi-DSPs
  - Einführung, Multi-DSP Architekturen
  - Programm-Wiederherstellung
  - Daten-Partitionierung & *Strip Mining*
  - Parallelisierung
  - Speicher-Aufteilung, Feld-Deskriptoren, DMA
  - Ergebnisse

Function Specialization

Auch „*Procedure Cloning*“ genannt

– Altbekannte Compiler-Technik (1993)
– Ziele: Ermöglichen weiterer Optimierungen, Reduktion des Aufwands zur Parameterübergabe bei Funktionsaufrufen
– Vorgehen: Von manchen Funktionen werden spezialisierte Kopien *(„Clones“)* mit weniger Parametern als im Original erzeugt

Specialization* sog. *Interprozedurale Optimierung

– Betrachtung der Aufruf-Relationen zwischen Funktionen zur Durchführung einer Optimierung

Gegenteil *Intraprozedurale Optimierung*

– Optimierung lokal in einer Funktion *f*, ohne Betrachtung von Funktionen, die *f* aufruft, oder von denen *f* aufgerufen wird

Typische Funktionsaufrufe

```
int f( int *x, int n, int p ) {
  for (i=0; i<n; i++) {
    x[i] = p * x[i];
    if ( i == 10 ) { ... }
  }
  return x[n-1];
}

int main() {
  ... f( y, 5, 2 ) ...
  ... f( z, 5, 2 ) ...
  return f( a, 5, 2 );
}
```

Beobachtungen

- **f** wird mehrfach aufgerufen
- **f** hat 3 Argumente
- Argumenten **n** und **p** werden mehrfach Konstanten 5 und 2 übergeben

Weitere Beobachtungen

```
int f( int *x, int n, int p ) {
  for (i=0; i<n; i++) {
    x[i] = p * x[i];
    if ( i == 10 ) { ... }
  }
  return x[n-1];
}

int main() {
  ... f( y, 5, 2 ) ...
  ... f( z, 5, 2 ) ...
  return f( a, 5, 2 );
}
```

- **f** ist sog. Allzweck-Routine *(general-purpose function)*: Via Parameter **n** kann ein Feld **x** beliebiger Größe bearbeitet werden
- Kontrollfluss in **f** von Funktionsparameter abhängig

- **f** wird in **main** in ganz speziellem Kontext *(special-purpose)* benutzt: Bearbeitung von Feldern der Größe **n** = 5.

Spezialisierung von Allzweck-Routinen

```
int f( int *x, int n, int p ) {
  for (i=0; i<n; i++) {
    x[i] = p * x[i];
    if ( i == 10 ) { ... }
  }
  return x[n-1];
}

int main() {
  ... f_5_2( y ) ...
  ... f_5_2( z ) ...
  return f_5_2( a );
}
```

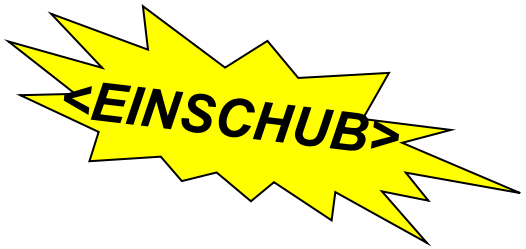
```
int f_5_2( int *x ) {
  for (i=0; i<5; i++) {
    x[i] = 2 * x[i];
    if ( i == 10 ) { ... }
  }
  return x[4];
}
```

Erwartete Effekte der Spezialisierung

Reduktion der *Average-Case* Laufzeit ACET wegen

– Ermöglichens weiterer Standard-Optimierungen in spezialisierter Funktion
– Weniger auszuführenden Codes zur Parameter-Übergabe

Standard-Optimierungen nach *Cloning* (1)



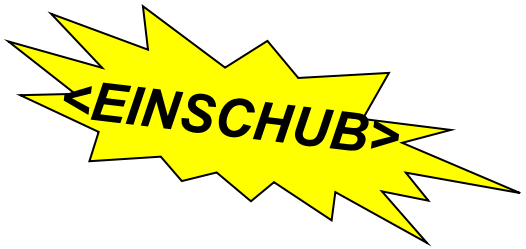
Konstanten-Faltung *(Constant Folding)*

– Ausrechnen von Ausdrücken mit rein konstanten Operanden schon zur Laufzeit des Compilers

– Beispiel:

Original-Code	*+ Cloning*	*+ ConstFold*
`a = p * n;`	`a = 2 * 5;`	`a = 10;`

Standard-Optimierungen nach *Cloning* (2)

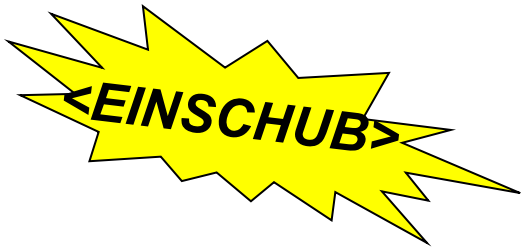


Konstanten-Propagierung *(Constant Propagation)*

- Ersetzen von Zugriffen auf Variablen, die nachweislich konstanten Inhalt haben, durch die Konstante selbst
- Beispiel:

Original-Code	*+ Cloning*	*+ ConstFold* *+ ConstProp*
`a = p * n;`	`a = 2 * 5;`	`a = 10;`
`for (…; i<a; …)`	`for (…; i<a; …)`	`for (…; i<10; …)`

Standard-Optimierungen nach *Cloning* (3)



Strength Reduction

– Ersetzen kostspieliger Operationen (z.B. Multiplikationen, Divisionen, ...) durch günstigere Operationen (z.B. Additionen, Subtraktionen, ...)

– Beispiel:

Original-Code	*+ Cloning*	*+ StrengthRed*
`x[i] = p*x[i];`	`x[i] = 2*x[i];`	`x[i] = x[i]<<1;`

Standard-Optimierungen nach *Cloning* (4)

Kontrollfluss-Optimierung

– Ermitteln des Wertebereichs von Variablen nach Konstanten-Faltung und -Propagation, Entfernen von unnötigen *If-Statements*, deren Bedingungen stets wahr oder falsch sind
(☞ vgl. Kapitel 3 – Loop Nest Splitting / Condition Satisfiability)

– Beispiel:

Original-Code

```
for(i=0;i<n;i++)
{
  …;
  if (i==10) …;
}
```

+ Cloning

```
for(i=0;i<5;i++)
{
  …;
  if (i==10) …;
}
```

+ ControlFlowOpt

```
for(i=0;i<5;i++)
{
  …;
}
```

Erwartete Effekte der Spezialisierung

Reduktion der *Average-Case* Laufzeit ACET wegen

– Ermöglichens weiterer Standard-Optimierungen in spezialisierter Funktion
– Weniger auszuführenden Codes zur Parameter-Übergabe

Erhöhung der Codegröße wegen

– Unter Umständen vieler neuer spezialisierter Klone

Einfluss von *Cloning* auf WCET-Schätzung

***Function Specialization* und WCET erst in jüngster Forschung betrachtet**

Erinnerung

– *Worst-Case Execution Time* WCET:
  Obere Schranke für die Laufzeit eines Programmes über alle denkbaren Eingaben hinweg

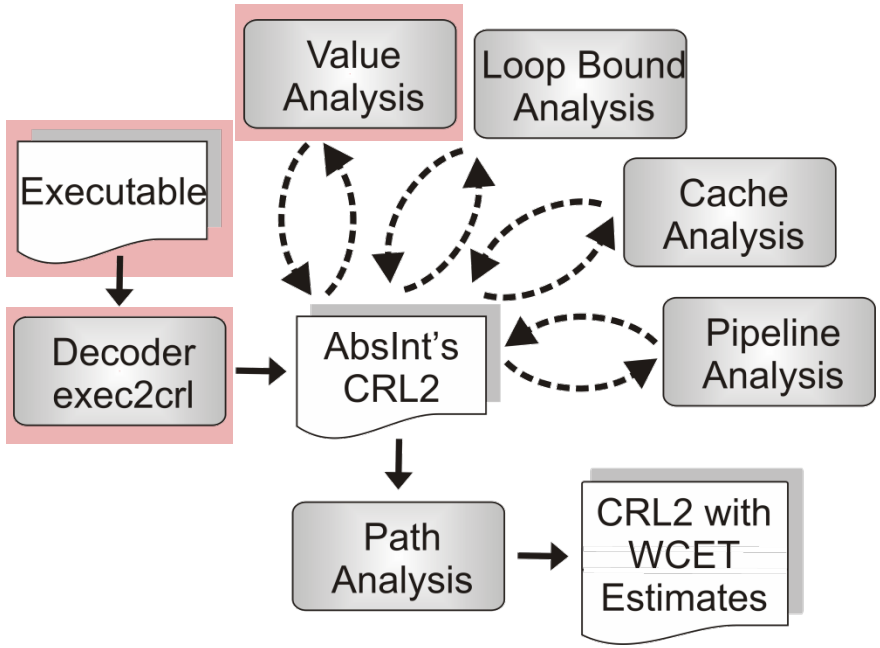
Abschätzung der WCET

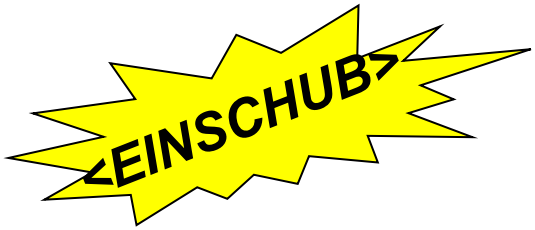
– Durch statische Analyse binärer ausführbarer Programme
– Im folgenden: Funktionsweise des WCET-Analysators aiT

[AbsInt Angewandte Informatik GmbH,
***http://www.absint.com/ait**, Saarbrücken, 2013]*

Ablauf des WCET-Analysators aiT (1)

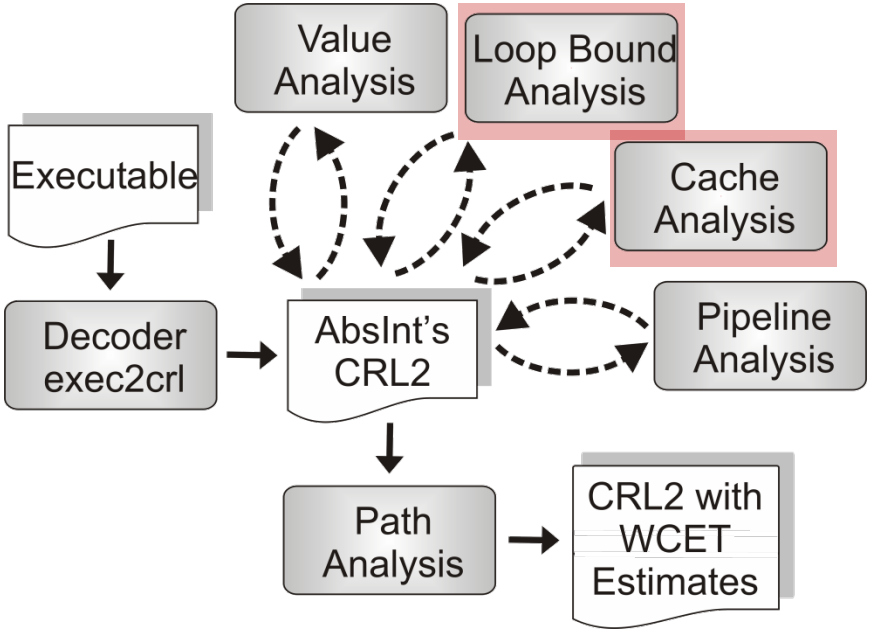
- **Eingabe:** Zu analysierendes Binär-Programm *P*
- **exec2crl:** Disassembler, übersetzt *P* in aiTs eigene *Low-Level* IR CRL2.
- **Wert-Analyse:** Bestimmung potentieller Inhalte von Prozessor-Registern zu jedem möglichen Ausführungszeitpunkt von *P*.
  Merke: *P wird von aiT niemals ausgeführt! P wird „lediglich“ analysiert.*

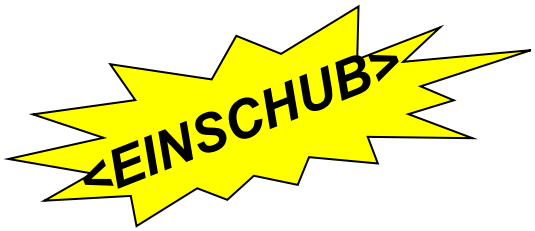




Ablauf des WCET-Analysators aiT (2)

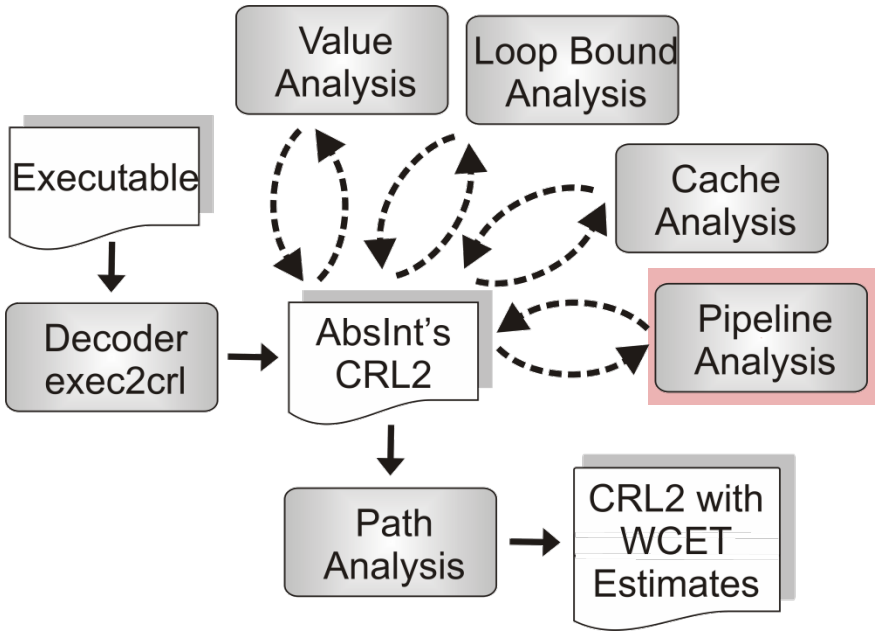
- **Schleifen-Analyse:** Versucht, obere und untere Schranken für die Anzahl von Iterationen jeder Schleife von *P* zu bestimmen.
- ***Cache*-Analyse:** Verwendet formales *Cache*-Modell, klassifiziert jeden Speicher-Zugriff in *P* als garantierten *Cache*-Hit, garantierten *Cache*-Miss, oder als unbekannt.

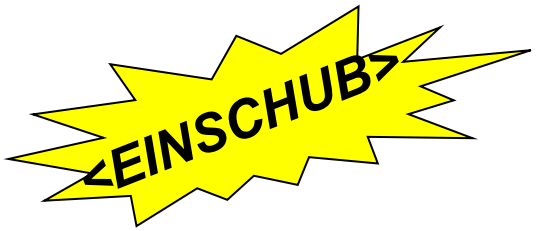




Ablauf des WCET-Analysators aiT (3)

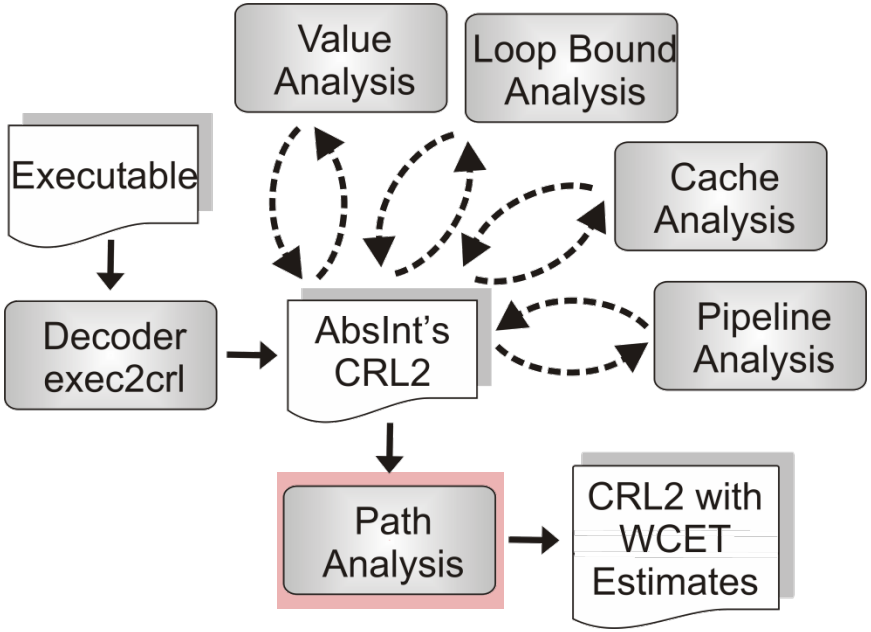
– **Pipeline-Analyse:** Enthält akkurates Modell der Prozessor-Pipeline. Abhängig von Start-Zuständen der Pipeline, möglichen *Cache*-Zuständen u.a. werden mögliche End-Zustände der Pipeline pro Basisblock ermittelt. Ergebnis ist die $WCET_{EST}$ jedes einzelnen Basisblocks.

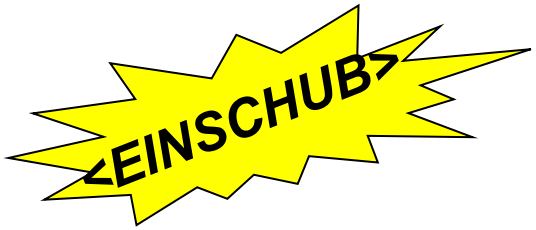




Ablauf des WCET-Analysators aiT (4)

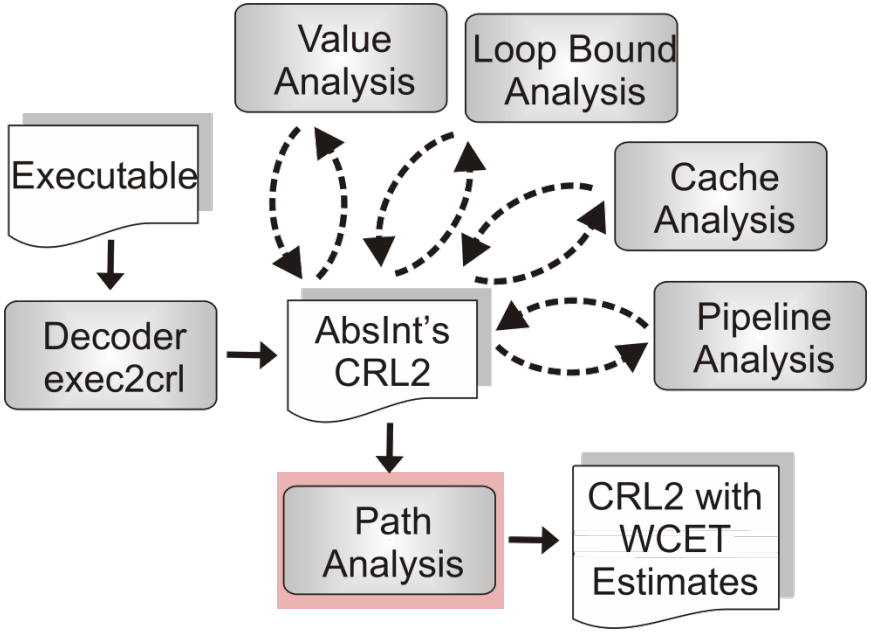
– **Pfad-Analyse:** Modelliert alle möglichen Ausführungspfade von *P* unter Berücksichtigung der $WCET_{EST}$ einzelner Basisblöcke und ermittelt den längsten möglichen Ausführungspfad, der zur Gesamt-$WCET_{EST}$ von *P* führt. Ausgabe ist u.a. die Länge dieses längsten Pfades, d.h. die $WCET_{EST}$ von *P*.

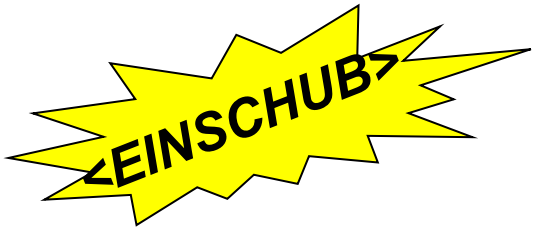




Ablauf des WCET-Analysators aiT (4)

– **Pfad-Analyse:** Modelliert alle möglichen Ausführungspfade von *P* unter Berücksichtigung der $WCET_{EST}$ einzelner Basisblöcke und ermittelt den längsten möglichen Ausführungspfad, der zur Gesamt-$WCET_{EST}$ von *P* führt. Ausgabe ist u.a. die Länge dieses längsten Pfades, d.h. die $WCET_{EST}$ von *P*.



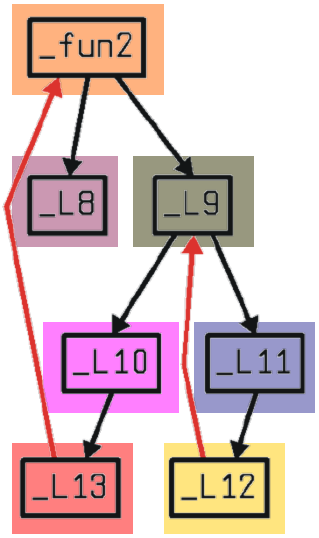


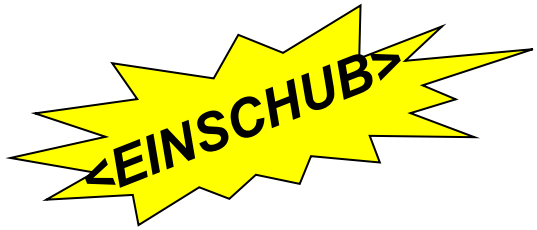
Ablauf des WCET-Analysators aiT (5)

Kontrollfluss-Graph *(control flow graph, CFG)*
Grundlegende Datenstruktur der Pfad-Analyse

Definition: *CFG = (V, E)* ist ein gerichteter Graph mit
– $V = \{ b_1, ..., b_n \}$ (Menge aller Basisblöcke, vgl. Kapitel 2)
– $E = \{ (b_i \rightarrow b_j) \}$ | Basisblock b_j kann direkt nach b_i ausgeführt werden }

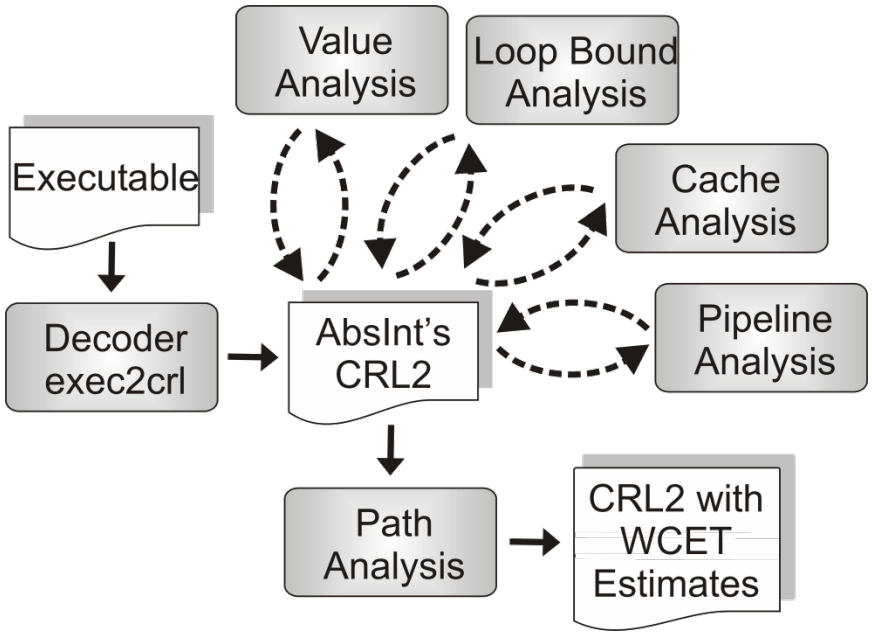
```
int fun2() {
  for (; a1<30; a1++ ) {
    for (; b1<a1; b1++ )
      printf( "%d\n", b1 );
    printf( „%d\n“, a1 );
  }
  return a1; }
```

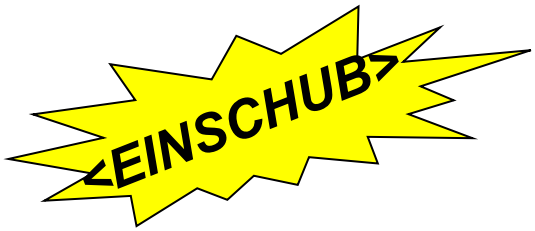




Komplexität der WCET-Analyse

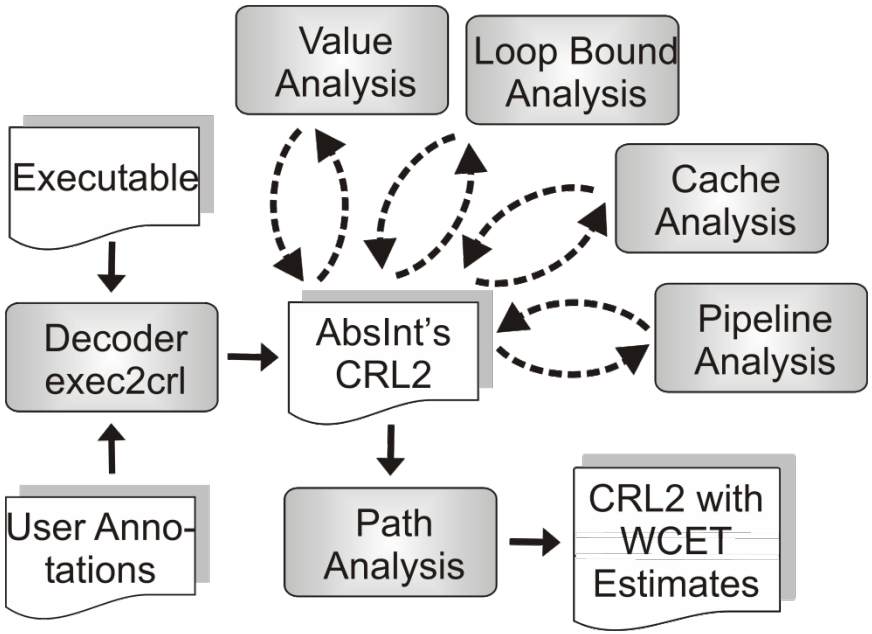
- **Problem:** WCET-Analyse nicht mit Computern berechenbar! Wäre WCET berechenbar, könnte man in $O(1)$ entscheiden, ob die WCET $< \infty$ ist und so das Halte-Problem lösen.
- **Grund:** Unberechenbar, wie lange P in Schleifen verweilt; automatische Schleifen-Analyse nur auf einfache Klassen von Schleifen anwendbar. Dito bei rekursiven Funktionen.





Annotationen zur WCET-Analyse

- **Ausweg:** Der Nutzer von aiT muss Informationen u.a. über minimale und maximale Iterationszahlen von Schleifen und Rekursionstiefen zwingend bereitstellen.
- **Annotations-File:** Enthält derartige Benutzer-Annotationen *(„Flow Facts“)*, ist neben zu analysierendem Programm *P* weiterer externer Input für aiT.



Wieso Spezialisierung und $WCET_{EST}$?

Motivation

- 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.
- *Function Specialization* ermöglicht hochgradig präzise Annotation von Schleifengrenzen.

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

Schleifenannotationen & *Cloning* (1)

```
int f( int *x, int n, int p ) {
  // loopbound min 0, max n
  for (i=0; i<n; i++ ) {
    x[i] = p * x[i];
    if ( i == 10 ) { ... }
  }
  return x[n-1];
}
```

Original-Code

– Schleifen-Annotation extrem unpräzise, da Annotation ***alle*** Aufrufe von **f** sicher überdecken muss.

– Wird **f** irgendwo mit **n** = 2000 als Maximalwert aufgerufen, ist stets von max. 2000 Iterationen auszugehen.

– Bei Aufrufen wie **f(a, 5, 2)** werden ebenfalls 2000 Iterationen zugrunde gelegt

☞ *Massive WCET-Überschätzung*

Schleifenannotationen & *Cloning* (2)

```
int f_5_2( int *x ) {
  // loopbound min 5, max 5
  for (i=0; i<5; i++ ) {
    x[i] = 2 * x[i];
    if ( i == 10 ) { ... }
  }
  return x[4];
}
```

Spezialisierter Code

– Schleifen-Annotation extrem präzise, da Annotation ***alle*** Aufrufe von **`f_5_2`** exakt erfasst.

– Bei Aufrufen wie **`f_5_2( a )`** werden nun exakt 5 Iterationen zugrunde gelegt

☞ *Exakte WCET-Abschätzung*

– Originale Funktion **`f`** bzw. weitere zusätzliche Spezialisierungen von **`f`** unabhängig von Annotationen von **`f_5_2`**

☞ *Keine Wechselwirkungen in WCET-Analyse dieser Derivate von* **`f`**.

Durchführung der Spezialisierung (1)

Vorsicht bei Anwendung von *Function Specialization*

– Anwachsen der Codegröße darf während Spezialisierung nicht außer Acht gelassen werden!
– Klonen jeder beliebigen Funktion, die mindestens zweimal irgendwo mit einem gleichen konstanten Argument aufgerufen wird, ist i.d.R. inakzeptabel.

Parametergesteuerte Durchführung der Spezialisierung

– Größe *G*: Spezialisiere niemals eine Funktion *f*, die größer als Parameter *G* ist.
– Anteil gleicher konstanter Argumente *K*: *f* nur klonen, wenn min. *K*% aller Aufrufe von *f* gleiche konstante Argumente enthalten.

Durchführung der Spezialisierung (2)

Gegeben

– Zu spezialisierende Funktion *f*
– *Dictionary arg*, das zu spezialisierende Argumente von *f* auf zu verwendende konstante Werte abbildet

Vorgehensweise in ICD-C

– Erzeuge neuen Funktionstyp *T´*, der dem Typ *T* von *f* entspricht, jedoch ohne zu spezialisierende Argumente aus *arg*
– Erzeuge neues Funktionssymbol *S´* vom Typ *T´*; trage *S´* in die Symboltabelle ein, in der *f* deklariert ist
– Erzeuge eine Kopie *f´* des Codes von *f* mit Funktionssymbol *S´*; trage *f´* in die gleiche *Compilation Unit* von *f* ein

Durchführung der Spezialisierung (3)

Gegeben

– Zu spezialisierende Funktion *f*

– *Dictionary arg*, das zu spezialisierende Argumente von *f* auf zu verwendende konstante Werte abbildet

Vorgehensweise in ICD-C *(fortges.)*

– Für jedes zu spezialisierende Argument *a* aus *arg*:

  – Erzeuge neue lokale Variable *v´* in *f´*

  – Erzeuge Zuweisung ***v´ = <arg[a]>;*** zu Beginn von *f´*

  – Ersetze jede Verwendung von *a* in *f´* durch *v´*

Durchführung der Spezialisierung (4)

Gegeben

– Zu spezialisierende Funktion *f*

– *Dictionary arg*, das zu spezialisierende Argumente von *f* auf zu verwendende konstante Werte abbildet

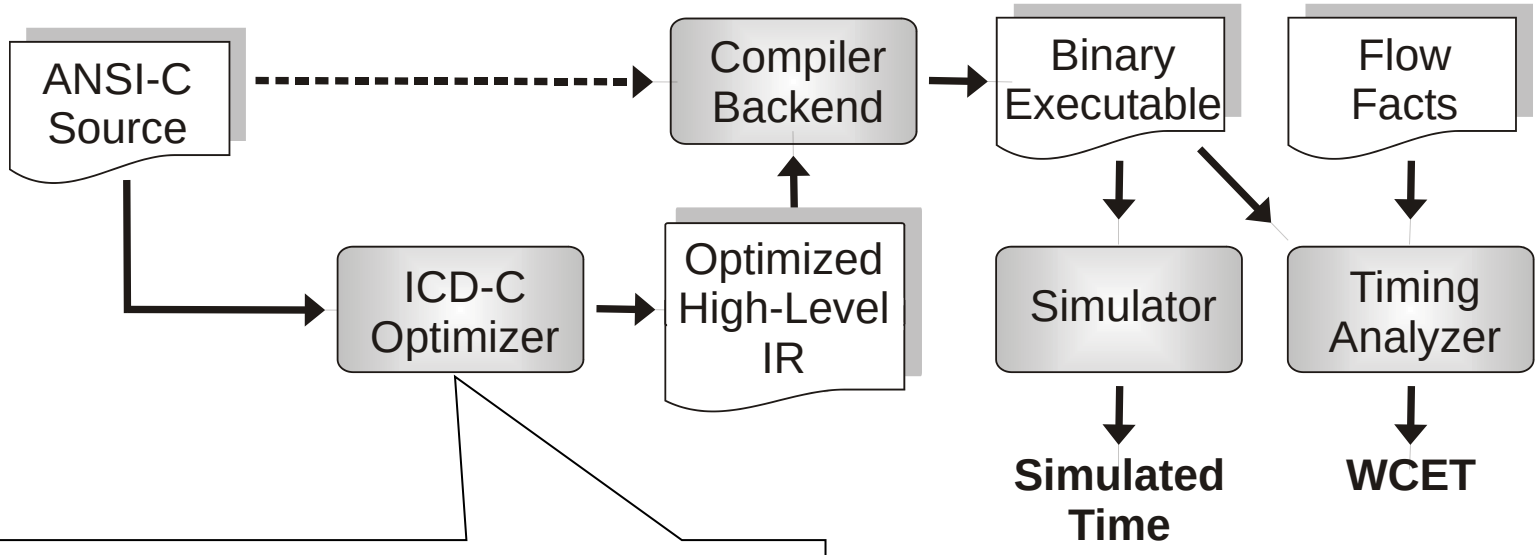
Vorgehensweise in ICD-C *(fortges.)*

– Für jeden zu spezialisierenden Funktionsaufruf *c* von *f*:

  – Entferne alle zu spezialisierenden Argumente in *arg* aus der Parameter-Übergabeliste von *c*

  – Ersetze aufgerufene Funktion *f* in *c* durch *f´*

Ablauf der Spezialisierung

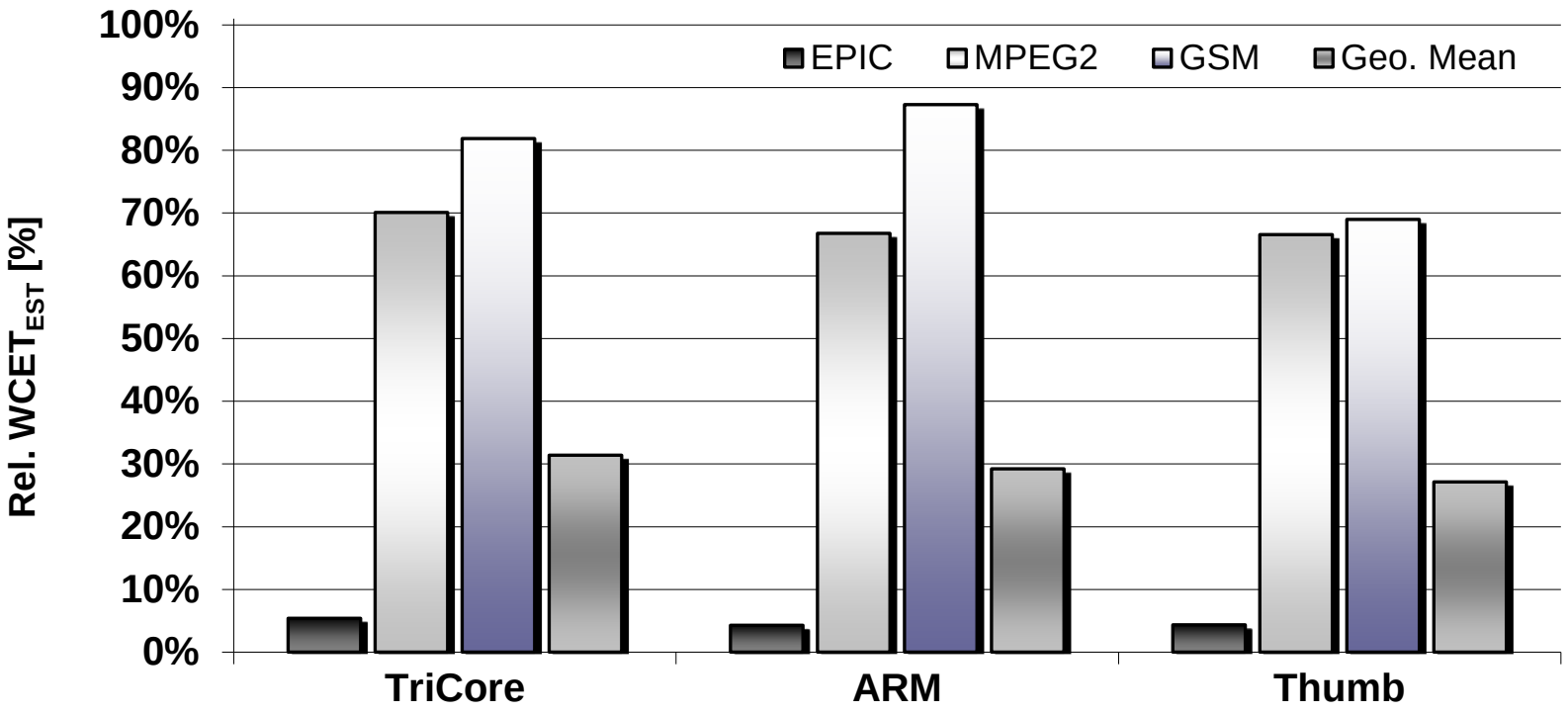


- *Function Specialization*
- *Constant Folding & Propagation*
- *Strength Reduction*
- *If-Statements*
- *G* = 2000 ICD-C *Expressions*
- *K* = 50%

Betrachtete Prozessoren

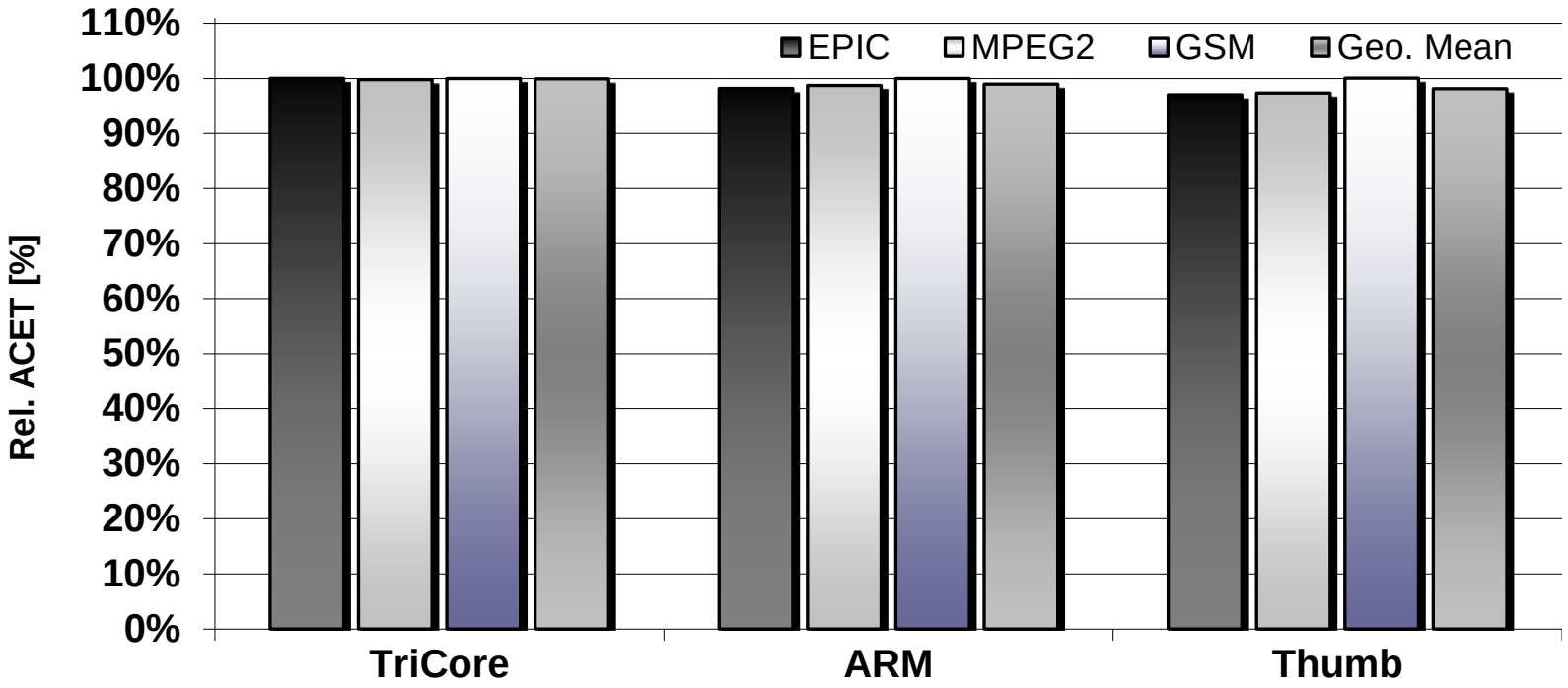
- Infineon TriCore TC1796
- ARM 7 TDMI (ARM- und THUMB-Befehlssätze)

Relative $WCET_{EST}$ nach *Cloning*



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

Relative ACETs nach *Cloning*



– Geringfügige ACET-Verbesserungen von maximal 3%.

☞ Einfluss des Overheads zur Parameterübergabe auf ACET und Effekt der nachfolgenden Optimierungen offenbar marginal

Beziehung zwischen $WCET_{EST}$ und ACET

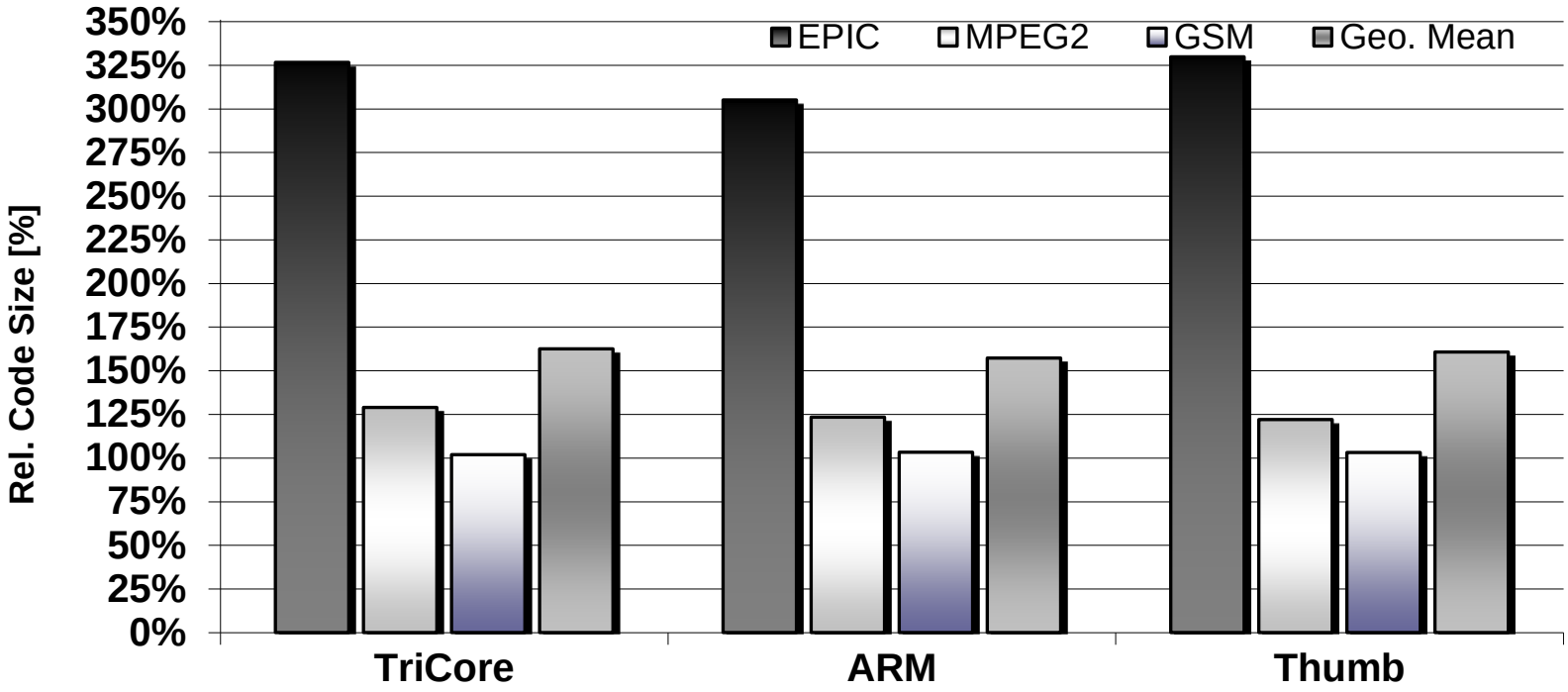
Sehr erstaunlich

Wie kann sich eine Optimierungstechnik dermaßen unterschiedlich auf scheinbar so ähnliche Zielfunktionen wie WCET und ACET auswirken?

Gründe

- Code vor Spezialisierung schlecht annotierbar
  - ☞ aiT liefert sehr unpräzise WCET-Abschätzungen
- Code nach Spezialisierung präziser annotierbar
  - ☞ aiT liefert wesentlich präzisere WCET-Abschätzungen
- 👍 Spezialisierung erhöht offenbar die *Tightness* der $WCET_{EST}$ beträchtlich *(☞ vgl. Kapitel 2 – Zielfunktionen)*
- 👎 Die effektive (unbekannte weil unberechenbare) WCET dürfte durch Spezialisierung nur ähnlich wie ACET reduziert werden

Relative Codegröße nach *Cloning*



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

Inhalte des Kapitels

5. HIR Optimierungen und Transformationen

- Motivation
- *Function Specialization / Procedure Cloning*
  - Allzweck-Funktionen in speziellen Kontexten
  - Einschub: Standard-Optimierungen & WCET-Abschätzung
  - *Function Specialization* und WCETs
  - Ergebnisse ($WCET_{EST}$, ACET, Codegröße)
- Parallelisierung für Homogene Multi-DSPs
  - Einführung, Multi-DSP Architekturen
  - Programm-Wiederherstellung
  - Daten-Partitionierung & *Strip Mining*
  - Parallelisierung
  - Speicher-Aufteilung, Feld-Deskriptoren, DMA
  - Ergebnisse

Parallelisierung für Multi-DSPs

Material freundlicherweise zur Verfügung gestellt von

Björn Franke und *Michael O'Boyle*
University of Edinburgh, UK
School of Informatics

Parallelisierung für Multi-DSPs

Motivation

– Performance-Anforderungen eines gesamten Systems übersteigen oft Fähigkeiten eines einzelnen Prozessors
(z.B. Radar, Sonar, medizinische Bildverarbeitung, HDTV, ...)

– Parallel arbeitende DSPs stellen hinreichend Performance zur Verfügung, aber...

- Wenig bis keine Hardware-Unterstützung für parallele Ausführung
- Noch weniger Unterstützung paralleler Programmiertechniken durch Entwurfswerkzeuge
- Existierende Quellcodes oft in *low-level* Stil programmiert, welcher Parallelisierung erschwert

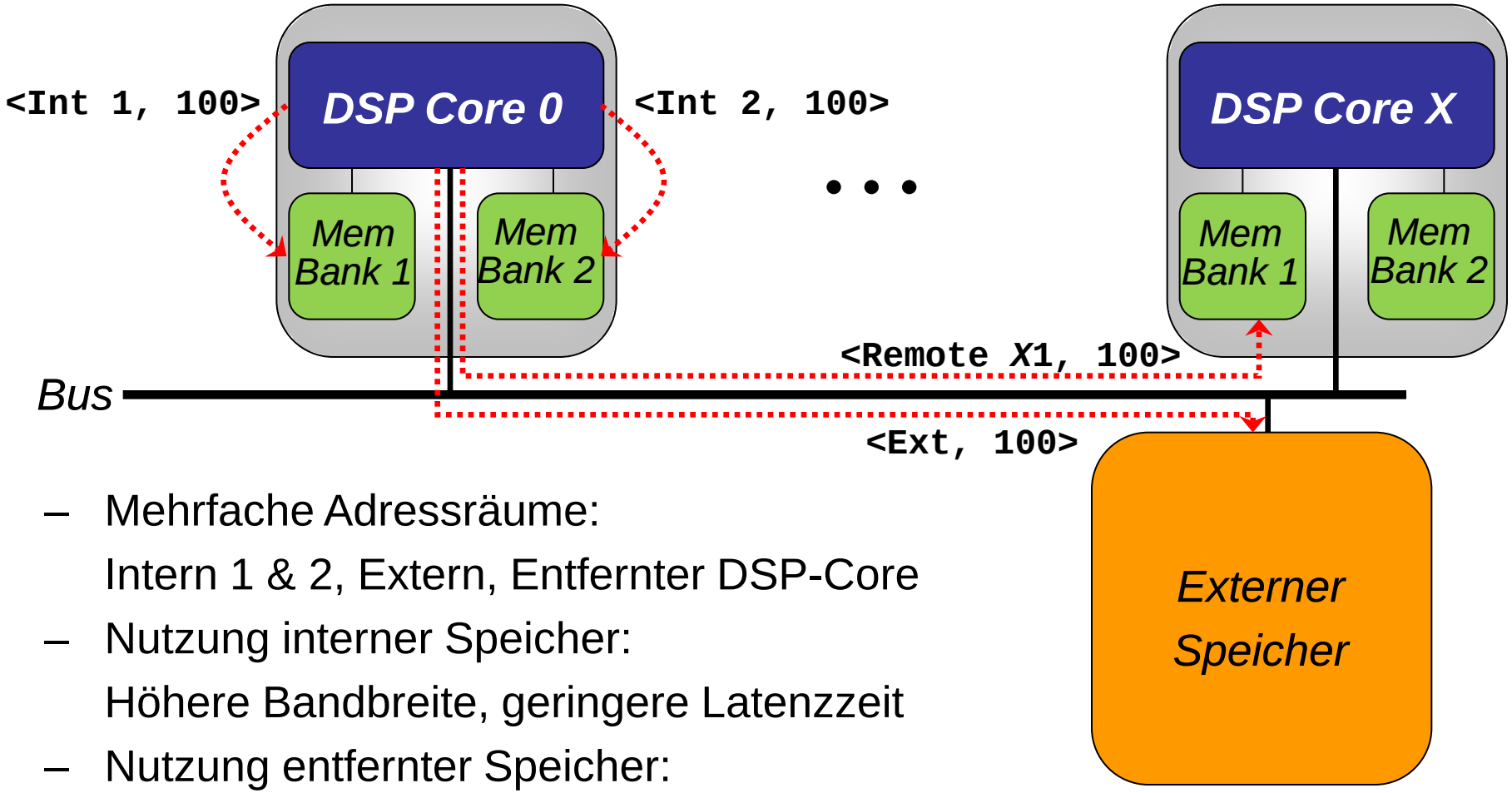
Parallelisierende Compiler

Spezialgebiet *„High Performance Computing”*

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

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

Multi-DSPs



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

Ablauf der Parallelisierung

Programm-Wiederherstellung
– Entfernen „ungewünschter“ *low-level* Konstrukte im Code
– Ersetzung durch *high-level* Konstrukte

Entdeckung von Parallelität
– Identifizierung parallelisierbarer Schleifen

Partitionierung und Zuordnung von Daten
– Minimierung des Kommunikations-Overheads zwischen DSPs

Erhöhung der Lokalität von Speicherzugriffen
– Minimierung der Zugriffe auf Speicher entfernter DSPs

Optimierung der Speicher-Transfers
– Nutzung von DMA für Massen-Transfers

Code-Beispiel für 2 parallele DSPs

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

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

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

- *Low-level Array*-Zugriffe über Zeiger; explizite Zeiger-Arithmetik *(☞ vgl. Kapitel 2, Auto-Increment Adressierung)*
- Nachteilig für Parallelisierung, da *ad hoc* keine Struktur in *Array*-Zugriffen erkenn- und analysierbar

Programm-Wiederherstellung

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

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

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

Programm-Wiederherstellung

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

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

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

Daten-Partitionierung

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

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

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

Daten-Partitionierung

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

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

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

Strip Mining der `i`-Schleife

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

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

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

Strip Mining der i-Schleife

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

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

– Wie kann dieser Code für zwei DSPs parallelisiert werden?

Parallelisierung (für Prozessor 0)

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

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

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

- Einfügen einer expliziten Prozessor-ID

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

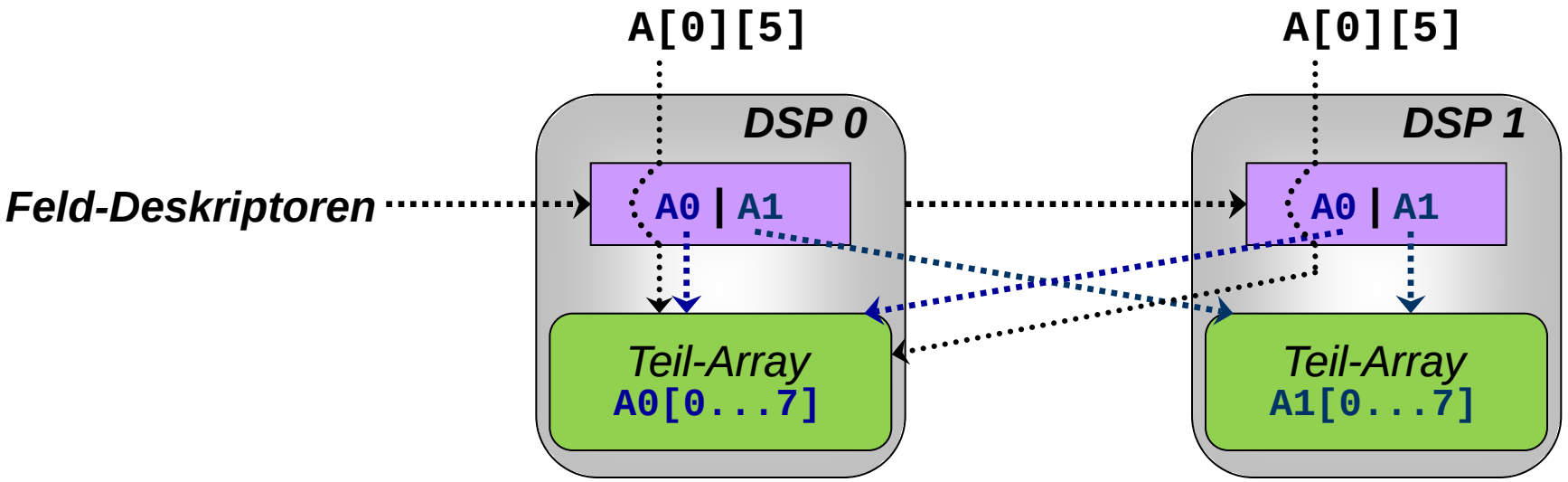
Parallelisierung (für Prozessor 0)

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

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

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

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

Feld-Deskriptoren

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



Speicher-Aufteilung (für Prozessor 0)

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

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

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

– Felder in DSP-internem und entfernten Speicher

– *Array*-Zugriffe über Deskriptoren in unveränderter Syntax

Speicher-Aufteilung (für Prozessor 0)

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

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

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

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

Lokalitätserhöhung von Feld-Zugriffen

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

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

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

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

Lokalitätserhöhung von Feld-Zugriffen

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

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

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

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

Einfügen von DMA Block-Transfers

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

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

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

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

– Blockweises Laden eines lokalen Puffers aus entferntem Speicher per DMA

– Feld-Zugriffe in Schleife nur noch auf lokalen Speicher

Durchführung der Parallelisierung

Multi-DSP Hardware

– 4 parallele Analog Devices TigerSHARC TS-101 @250 MHz
– 768 kB lokales SRAM pro DSP, 128 MB externes DRAM

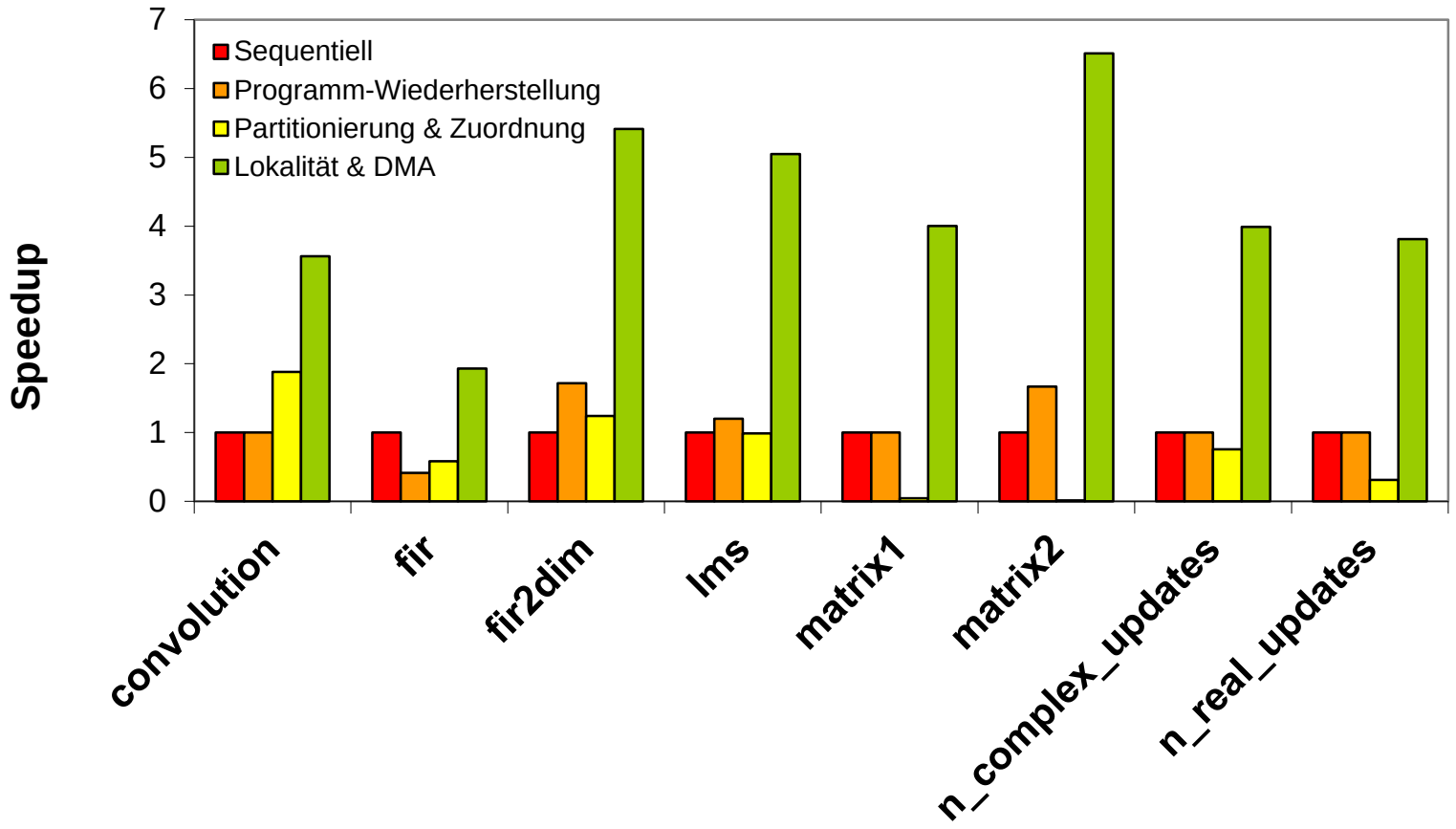
Parallelisierte Benchmark-Programme

– *DSPstone:* kleine DSP-Routinen, geringe Code-Komplexität
– *UTDSP:* komplexe Anwendungen, rechenintensiver Code

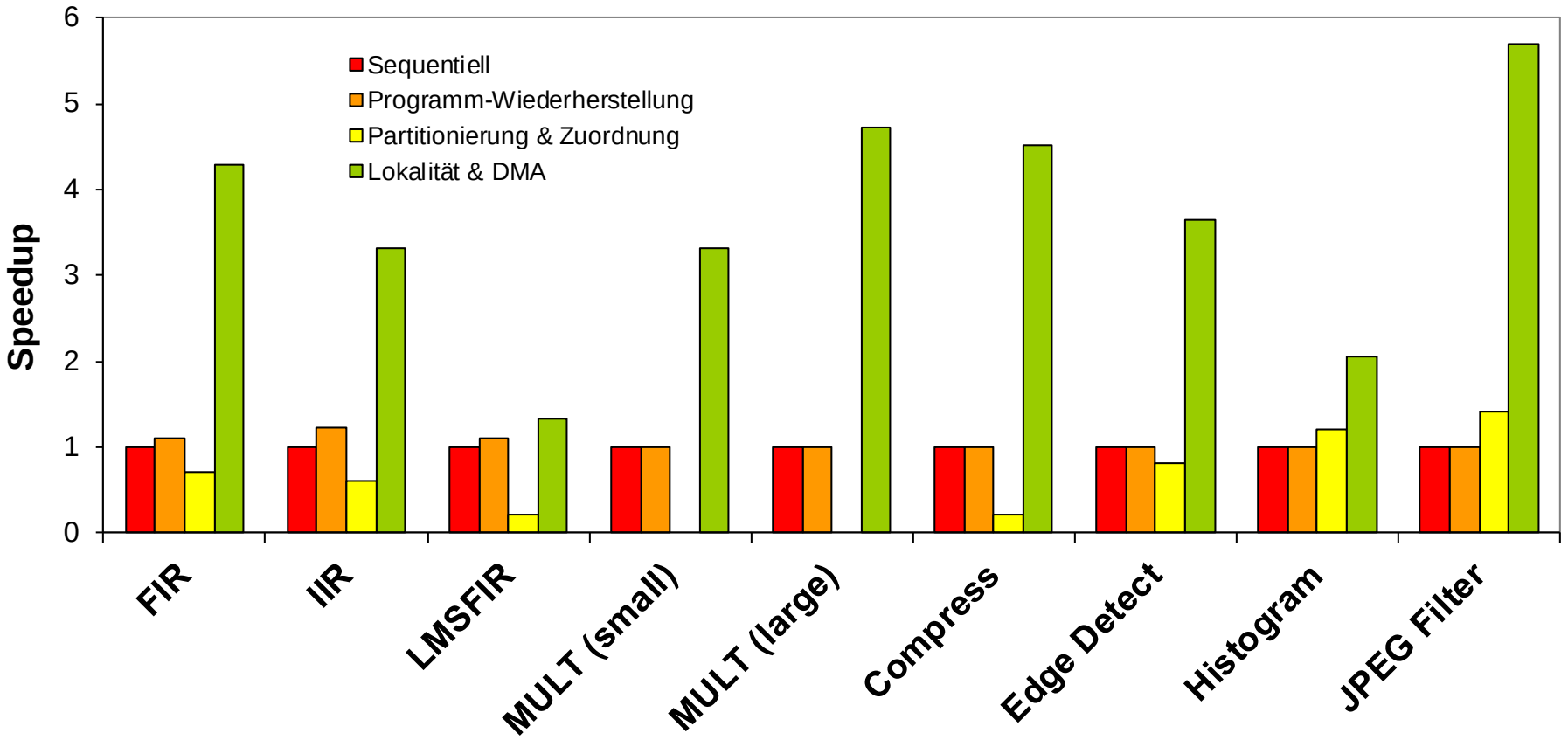
Ergebnisse: Laufzeiten

– für rein sequentiellen Code auf 1 DSP laufend
– für Code nach Programm-Wiederherstellung
– für Code nach Daten-Partitionierung und -Zuordnung
– für Code nach Erhöhung der Lokalität & DMA

Ergebnisse – DSPstone



Ergebnisse – UTDSP



Diskussion der Ergebnisse

Mittlere Gesamt-*Speedups*

– DSPstone: Faktor 4,06
– UTDSP: Faktor 3,38
– Alle Benchmarks: Faktor 3,68

Sehr erstaunlich

– *Wie kann für DSPstone ein Speedup über Faktor 4 erzielt werden, wenn eine Parallelisierung für 4 DSPs erfolgt?*

Gründe für Super-Lineare *Speedups*

Super-Lineare *Speedups* > 4 bei 4 parallelen DSPs

– Parallelisierter Code ist nachfolgenden Compiler-Optimierungen evtl. besser zugänglich als originaler sequentieller Code

– *Beispiel:*
Sequentielle **`i`**-Schleife (☞ *vgl. Folie 47*): 16 Iterationen
Auf 2 DSPs parallelisierte **`i`**-Schleife (*Folie 49*): 8 Iterationen

☞ Parallelisierte Schleifen u.U. Kandidaten für *Loop Unrolling*:

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

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

} *8 Mal*

– Abgerollte Schleife ohne Sprünge!

☞ Keine *Delay-Slots*, Sprung-Vorhersage liegt stets richtig

Literatur

Function Specialization

– D. Bacon, S. Graham, O. Sharp. *Compiler Transformations for High-Performance Computing*. ACM Computing Surveys 26(4), 1994. *(Sehr gutes Übersichts-Paper zu allg. Compiler-Optimierungen!)*

– P. Lokuciejewski, H. Falk, H. Theiling. *Influence of Procedure Cloning on WCET Prediction*. CODES+ISSS, Salzburg 2007.

Parallelisierung für Homogene Multi-DSPs

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

Zusammenfassung

HIR-Optimierungen

– Umstrukturieren von Schleifen
– Umstrukturieren von Funktionen und deren Aufrufstruktur

Function Specialization / Procedure Cloning

– Spezialisierung von Allzweck-Routinen
– Ermöglichen von Standard-Optimierungen in spezialisierten Funktionen, Vereinfachung des Codes zur Parameterübergabe
– Auswirkung auf ACET gering, massiv bei $WCET_{EST}$

Parallelisierung für Homogene Multi-DSPs

– Fokus auf Ausnutzung lokaler Speicher und Adressbereiche
– *Speedups* im wesentlichen linear zur Anzahl paralleler DSPs