Superblock-basierte Quellcodeoptimierungen zur WCET-Reduktion

Timon Kelter

Lehrstuhl für Eingebette Systeme
Technische Universität Dortmund, 44221 Dortmund
timon.kelter@tu-dortmund.de

Zusammenfassung. Das Konzept der *Superblöcke* wurde auf dem Gebiet der Compileroptimierungen in der Vergangenheit bereits erfolgreich zur Optimierung der *ACET* (Average Case Execution Time) verwendet. Superblöcke sind dabei spezielle Ketten von Basisblöcken, die es erleichtern Optimierungen über Basisblockgrenzen anzuwenden und somit ein höheres Optimierungspotential zu schaffen. In der vorliegenden Arbeit wurde dieses Konzept zum ersten Mal zur Reduktion der *WCET* (Worst Case Execution Time) von Programmen für eingebettete Systeme ausgenutzt. Die WCET ist im Kontext der eingebetteten Systeme eine wichtige Metrik, da viele eingebettete Systeme unter Echtzeitbedingungen arbeiten müssen und hierzu eine sichere obere Schranke für die Laufzeit eines Programms unabdingbar ist.
Die vorgestellte Superblockbildung baut auf einem neuartigen Trace-Selektions-Algorithmus auf, der WCET-Daten auswertet. Außerdem wurde das Konzept der Superblöcke zum ersten Mal auf der Quellcodeebene angewandt. Auf diese Weise findet die Optimierung früher statt, so daß eine größere Anzahl nachfolgender Optimierungen von der erzielten Umstrukturierung profitieren kann. Weiterhin wurden die klassischen Optimierungen *Common Subexpression Elimination (CSE)* und *Dead Code Elimination (DCE)* an die Anwendung in Quellcode-Superblöcken angepasst. Mit diesem Techniken wurde auf einer Testmenge von 55 bekannten Standard-Benchmarks eine durschnittliche WCET-Reduktion von bis zu 10.2% erzielt.

1 Einleitung

Moderne eingebettete Systeme sind durch vielfältige Anforderungen geprägt. Sowohl die Effizienz in Bezug auf ACET, Codegröße, Herstellungskosten und Energieverbrauch als auch das Zeitverhalten in Worst-Case-Szenarien sind wichtige, teilweise gegenläufige Bewertungsgrundlagen. In Bereichen wie z.B. Automotiver Software oder Avioniksystemen, wo Zeit eine kritische Ressource ist, ist die WCET hierbei das dominierende Kriterium. Sie wird als Schlüsselparameter für die Ablaufplanung (Scheduling) der Prozesse benötigt und dient zur Sicherstellung einer korrekten Interaktion der Software mit der Systemumgebung.

Durch die zunehmende Komplexität der heutigen eingebetteten Software findet die Entwicklung großteils nicht mehr in Assembler statt, sondern in Hochsprachen wie C. Für diese Sprachen steht in aktuellen Compilern eine Vielzahl

von Optimierungen zur Verfügung, die i.W. darauf zielen die ACET zu optimieren. Im Gegensatz dazu ist die compilergesteuerte Reduktion der WCET eine neue Forschungsrichtung von steigendem industriellem und akademischem Interesse. WCET-gesteuerte Kompilierung setzt die Integration eines WCET-Analysewerkzeugs in den Compiler vorraus, das die WCET-Daten zur Verfügung stellt, die von WCET-sensitiven Optimierungen ausgelesen werden können. Die in dieser Arbeit vorgestellte Optimierung wurde in den Forschungscompiler *WCC* (WCET-aware C Compiler) [11] integriert, in dem das kommerziell eingesetzte WCET-Analysewerkzeug *aiT* [6] Verwendung findet.

Anders als die klassische Superblock-Optimierung, die auf Profiling-Daten aufbaut, wertet unsere WCET-gesteuerte Optimierung die Ausgaben von aiT aus und erhält auf diese Weise die WCET und die *WCEC* (Worst Case Execution Count) pro Basisblock und die WCEC pro Kante im Kontrollflußgraph (CFG) des Programms. Die Informationen stehen initial nur auf der Assembler-Ebene zur Verfügung, da aiT auf dieser Ebene arbeitet, und werden dann durch die im WCC vorhandene *Back-Annotation* [12] auf die Quellcodeebene transformiert. Mit diesen Informationen werden iterativ Superblöcke gebildet und optimiert. Ein besonderes Problem bei der Optimierung der WCET ist die Instabilität des *WCEP* (Worst Case Execution Path), also des längten Ausführungspfads im Programm, dessen Ausführungsdauer die WCET ist. Änderungen an einem WCEP P können zu einem *Pfadwechsel* führen, so daß danach ein anderer Pfad P' der längste Pfad im CFG ist. Durch eine Analyse mit adaptiver Genauigkeit stellen wir sicher, daß nachfolgende Optimierungsschritte auf P' weiterarbeiten und nicht auf dem veralteten Pfad P.

Die einzige Arbeit, die ebenfalls Superblöcke zur WCET-Reduktion einsetzt von Zhao [7], weicht in entscheidenen Punkten von der vorliegenden ab:

- Zhao bildet die Superblöcke auf Assembler-Ebene, was, wie in [2] angesprochen, das Optimierungspotential senkt.
- Unsere Arbeit setzt weiter, superblock-spezifische Optimierungen ein, was in [7] nicht geschieht.
- Die Art und Weise der WCEP-Neuberechnung in [7] ist nicht auf größere Programme übertragbar, weshalb dort auch nur kleine Benchmarks Verwendung finden. Im Gegensatz dazu führen wir unsere Tests auf Standard-Benchmarks mit realistischer Größe aus.

Die einzelnen Bestandteile des im Rahmen dieser Arbeit verwendeten Optimierungsframework werden in den folgenden Abschnitten kurz vorgestellt. Der generelle Ablauf ist:

1. Es findet eine initiale Kompilierung des Codes mit integrierter WCET-Analyse statt um die initialen WCET-Daten zu erhalten
2. Es wird ein Quellcode-Superblock selektiert und ggf. erweitert. Nähere Informationen hierzu finden sich in Abschnitt 2.
3. Der erzeugte Superblock wird optimiert (s. Abschnitt 3)
4. Die WCET-Daten werden aktualisiert. Die hierbei verwendete Analysetechnik wird in Abschnitt 4 beschrieben.

5. Wenn alle Blöcke Teil eines Superblocks sind, terminiert das Verfahren, ansonsten fährt es mit Punkt 2 fort.

Eine Beschreibung der verwendeten Experimente und der erzielten Ergebnisse folgt in Abschnitt 5.

2 WCET-sensitive Bildung von Quellcode-Superblöcken

2.1 Traces und Superblöcke

Superblöcke sind eine Weiterentwicklung von *Traces*, die erstmals von Fisher [3] für ein effektiveres Instruction-Scheduling über Basisblockgrenzen hinweg eingesetzt wurden.

Definition 1. *Zu einem gegebenen Programm P sei $G = (V, E)$ ein gerichteter, zyklischer oder azyklischer Graph, im folgenden als Kontrollflußgraph (CFG) bezeichnet. Die Knotenmenge V entspricht der Menge B der Basisblöcke in P. E besteht aus Kanten (b_i, b_j) für jeden möglichen Kontrollflußübergang in P zwischen zwei Basisblöcken $b_i, b_j \in B$. Die Menge L von Schleifen im CFG sei eindeutig bestimmbar und zu jeder Schleife $l_i \in L$ seien die Basisblöcke $b_l \in l_i$ bekannt. Ein* ***Trace*** *T ist eine Folge von Basisblöcken $T = (b_a, ..., b_k)$, so daß gilt:*

- $\forall a \leq i < k : (b_i, b_{i+1}) \in E$
- $\forall l_i \in L : \exists b_l \in l_i : b_l \in T \Rightarrow \forall b_r \in T : b_r \in l_i$

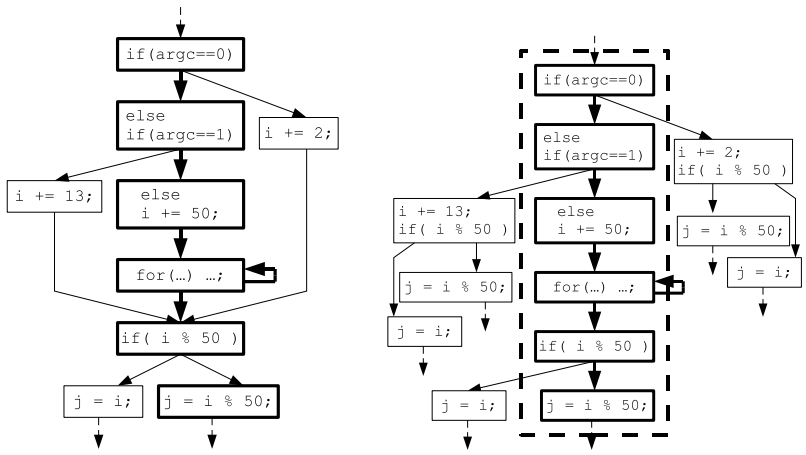
T erstreckt sich also nie über Schleifengrenzen hinweg, kann aber innere Schleifen, innerhalb der den Trace umgebenden Schleife, mit einschließen.

Chang [1] entwickelte das Konzept des Superblocks auf Assembler-Ebene um einige Probleme des Trace-Scheduling zu umgehen.

Definition 2. *Ein* ***Superblock*** *$S = \{b_{start}, ..., b_{end}\}$ ist ein Trace, der nur durch den ersten Block betreten werden kann, so daß gilt: $\forall b \in T \setminus \{b_{start}\} : \forall (b_i, b) \in E : b_i \in T$. Ein Knoten in einem Trace, der diese Bedingung verletzt, wird* ***Einsprungpunkt*** *genannt.*

Da die Größe von natürlich in Programmen vorkommenden Superblöcken beschränkt ist wurden Superblockvergrößerungs-Optimierungen [5] eingeführt. Die hautpsächliche Strategie ist die *Tail Duplication*, die Einsprungpunkte in den Trace $T = (b_{start}, ..., b_{end})$ durch Verdoppelung eines Trace-Endstücks eliminiert. Dazu wird ab dem ersten Einsprungpunkt $b_{first\_in} \in T$ eine Kopie K der Blöcke $(b_{first\_in}, ..., b_{end})$ erstellt und alle $e \in \{(b_{pred}, b_{first\_in}) \in E \mid b_{pred} \notin T\}$ werden zur Kopie $b'_{in} \in K$ umgeleitet. Da der Kontrollfluß auf der Assemblerebene beliebig verändert werden kann, ist hier nur eine einzige Kopie des Trace-Endstücks nötig.

Auf der Quellcodeebene kann man Basisblöcke analog zur Assembler-Ebene als Menge von Anweisungen definieren, die nur über die erste Anweisung betreten und nur über die letzte verlassen werden können. Dadurch ist die Definition



(a) Ursprünglicher Quellcode (b) Nach der Endstück-Duplikation

Abb. 1. Superblockbildung auf der Quellcodeebene

des Superblocks direkt auf die Hochsprachenebene übertragbar. Allerdings ist im Gegensatz zur Assembler-Ebene die beliebige Modifikation des Kontrollflusses in Hochsprachen teilweise nicht möglich (fehlendes `goto` z.B. in Java) oder nicht wünschenswert, da viele High-Level Optimierungen strukturierten Code benötigen. Daher müssen wir in diesem Fall für jeden Einsprungpunkt $b_{in} \in T$ vom Ende aus bis zum Anfang von T pro eingehender Off-Trace-Kante eine Kopie K der Anweisungen erzeugen, die die Blöcke $(b_{in}, ..., b_{end})$ enthalten und K in das High-Level Konstrukt der eingehenden Kante verschieben. In `goto`-freiem `C` muß hierfür zwischen Einsprungkanten unterschieden werden, die durch `if`-, `switch`- oder Schleifenanweisungen erzeugt werden. Für eine vollständige Beschreibung der verschiedenen Fälle sei auf die Diplomarbeit [13] verwiesen. Sie funktioniert allerdings immer analog zu dem Beispiel in Abbildung 1 (dort: Elimination von Einsprungkanten durch `if`-Anweisungen). In Abbildung 1 erkennt man zwei elementare Merkmale der High-Level Tail Duplication:

- Der Block `j = i` im abschließenden `if` musste dupliziert werden, obwohl er nicht Teil des Traces ist, da er Teil des Syntaxkonstrukts ist, daß den Trace-Tail enthält. Diese Extra-Kopie lässt sich nicht vermieden, ohne die Strukturiertheit des Codes zu verlieren.
- Die `for`-Anweisung ist eine innere Schleife, die als ganzes Teil des Superblocks ist.

Die Erzeugung von Superblöcken hat einige für die WCET-Optimierung günstige Eigenschaften:

- Es kann explizit ein bestimmter Pfad (hier: WCEP) optimiert werden.
- Die Menge der Punkte, an denen der Kontrollfluß sich vereinigt, wird entlang des gewählten Pfades verkleinert. Diese Punkte führen in statischen Analysen zu Überabschätzungen, daher kann ihre Reduktion die Genauigkeit der

Analyse erhöhen. Außerdem können Sprungbefehle für die Umsetzung des Kontrollflusses im Assemblercode entfallen.
– Datenabhängigkeiten werden reduziert, so daß nachfolgende Optimierungen mehr Freiheit in ihren Optimierungsentscheidungen haben.

2.2 Trace Selektion

Der entscheidende Schritt bei der Superblockbildung ist die Auswahl geeigneter Traces. Die bekannten Trace-Selektionsverfahren (s. [4]) sind i.W. Greedy-Ansätze, die zu durch Profiling-Daten gegebenen Ausführungshäufigkeiten iterativ einen Trace bilden indem sie:

– Bei einem Block b_{start}, der noch auf keinem anderen Trace liegt und maximale Ausführungshäufigkeit $w(b_{start})$ besitzt, beginnen.
– Den Trace abwechselnd an beiden Enden um jeweils den Nachbar-Block, der noch auf keinem Trace liegt und unter allen solchen Blöcken höchste Ausführungshäufigkeit besitzt, erweitern.
– Terminieren, wenn eine vorgegebene Trace-Länge erricht ist, oder der Trace in keine Richtung mehr verlängert werden kann.

Das Verfahren lässt sich für Knoten-Ausführungshäufigkeiten und Kanten-Ausführungshäufigkeiten formulieren, wobei letztes etwas besser ist, da zusammenhängende Pfade besser erfasst werden.

Längster-Pfad-Ansatz

Für die Selektion sind wir an einem Pfad interessiert, der möglichst viel zur WCET beiträgt. Die beiden Greedy-Verfahren können durch ihren begrenzten lokalen Blickwinkel diesen Pfad allerdings nicht in jeden Fall finden.

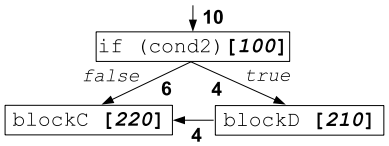


Abb. 2. Fehlerszenario für die bestehenden Trace-Selektionsverfahren

Im Beispiel in Abbildung 2 sind die Basisblöcke mit ihrer WCET (in Klammern) und den WCECs annotiert, die Kanten jeweils mit ihrer WCEC. Beide Greedy-Ansätze würden von der `if`-Anweisung aus, den Trace nach `blockC` verlängern, was einen Trace mit $6 * 100 + 6 * 220 = 1920$ Zyklen ergibt. Wenn der Trace hingegen in Richtung des `blockD` verlängert würde ergibt sich eine Laufzeit von $4 * 100 + 4 * 210 + 4 * 220 = 2120$ Zyklen.

Bei gegebenem Kontrollflußgraph $G = (V, E)$ mit Schleifenmenge L und WCET-Ergebnissen $wcet_v : V \to \mathbb{N}$, $wcec_v : V \to \mathbb{N}$ und $wcec_e : E \to \mathbb{N}$ geht der neue Längster-Pfad-Ansatz zur Trace-Selektion folgendermaßen vor:

1. Selektiere b_{start} mit maximaler WCET, der noch auf keinem Trace liegt. Liegt dieser Knoten in einer Schleife $l_i \in L$, so wird ein Trace innerhalb der Schleife selektiert, sonst wird ein Trace im Funktionsrumpf selektiert. Wir gehen im folgenden vom ersten Fall aus, der zweite funktioniert analog.
2. Ein gerichteter, kreisfreier Graph $G_{l_i} = (V_{l_i}, E_{l_i})$ wird konstruiert, der den Kontrollfluß zwischen allen Blöcken innerhalb der Schleife l_i wiederspiegelt, wobei innere Schleife zu jeweils einem einzigen Block komprimiert werden. Also gilt $V_{l_i} = \{b | b \in l_i \wedge (\nexists l_j \in L : l_j \in l_i \wedge b \in l_j)\} \cup I_{l_i} \cup \{b_{source_i}, b_{sink_i}\}$ wobei I_{l_i} die Menge der Platzhalterknoten für innere Schleifen innerhalb von l_i ist und E_{l_i} die Menge aller Kontrollflußkanten zwischen den Blöcken ist, die bei inneren Schleifen entsprechend zusammengefasst werden müssen. $wcet_v$, $wcec_v$ bzw. $wcec_e$ lassen sich durch Summation über die zusammengefassten Knoten / Kanten auch auf V_{l_i} bzw. E_{l_i} definieren. Zusätzlich ist b_{source_i} bzw. b_{sink_i} ein Knoten der mit Extrakanten an die Eintrittspunkte bzw. Austrittspunkte der Schleife angebunden wird (mit WCET/WCEC von 0).
3. Wir definieren auf G_{l_i} die WCET-Kostenfunktion $c_{l_i} : E_{l_i} \rightarrow \mathbb{N}$.

$$c_{l_i}(e = \{b_a, b_e\}) = wcet_v(b_a) \cdot \frac{wcec_e(e)}{wcec_v(b_e)} \tag{1}$$

   c_{l_i} gibt zu jeder Kante die WCET zurück, die beim Durchlaufen dieser Kante insgesamt anfällt. Da G_{l_i} per Konstruktion kreisfrei ist können wir den längsten Pfad durch die Schleife, der dann dem Pfad mit größtem WCET-Beitrag entspricht, berechnen indem wir c_{l_i} negieren und mit dem Bellman-Ford Algorithmus den kürzesten Pfad von b_{source_i} über b_{start} nach b_{sink_i} berechnen.
4. Der Trace wird gebildet indem er, ausgehend von b_{start} abwechselnd an beiden Ende entlang des berechneten längsten Pfades verlängert wird. Dabei wird in jedem Schritt eine Vorhersage des Codegrößenzuwachs bei der Superblockbildung errechnet und das Tracewachstum gestoppt sobald benutzerdefinierbare Grenzen überschritten werden.

Da der Längeste-Pfad-Algorithmus in jedem Fall den Trace mit der höchsten WCET findet, selektiert er auch im Beispiel aus Abbildung 2 den korrekten Pfad über `blockD`.

3 WCET-sensitive Superblock Optimierungen

3.1 Superblock Common Subexpression Elimination

Bei der klassischen Common Subexpression Elimination werden entweder global oder lokal in einem Basisblock auftretende, redundante Neuberechnugen von Ausdrücken vermieden indem das Ergebnis der ersten Berechnung zwischengespeichert wird und in den folgenden Auswertungen nur noch gelesen werden muß. Der Rechenaufwand für die Neuberechnung entfällt dadurch, allerdings steigt der Registerdruck durch die erhöhte Anzahl von lebendigen Variablen, was zu schlechteren Ergebnissen im Registerallokator führen kann. Daher wird

die Optimierung in unserem Falle nur innerhalb der Superblöcke durchgeführt, so daß nur solche Neuauswertungen vermieden werden, die Teil des WCEP sind. Zur Berechnung der eliminerbaren Ausdrücke wird über eine Datenflußanalyse die Menge der *verfügbaren* Ausdrücke berechnet, d.h. von Ausdrücken deren Ergebnis an nachfolgenden Programmstellen noch gültig ist, deren Operanden sich also nicht verändert haben können. Wenn an einem Ausdruck B innerhalb eines Superblocks derselbe Ausdruck bereits durch eine vorhergehende Auswertung A verfügbar ist, wird A durch eine Zuweisung $t_i = A$ und B durch eine Auswertung von t_i mit einer passenden, neuen temporären Variable t_i ersetzt.

3.2 Superblock Dead Code Elimination

Während bei der Standard Dead Code Elimination Ausdrücke gelöscht werden, deren Ergebnisse im Verlauf des Programms nicht mehr benötigt werden, versucht die Superblock Dead Code Elimination, Ausdrücke, deren Ergebnisse im Superblock nicht mehr gelesen werden, aus dem Superblock heraus zu verschieben. Dies soll bewirken, daß die Auswertung der Ausdrücke möglichst nicht mehr in dem dem Superblock zugrundeliegenden WCEP stattfindet, sondern abseits davon. Hierzu wird die *Lebendigkeit* von Ausdrücken auf der Hochsprachen-Ebene über eine klassische Datenflußanalyse berechnet (LIVE-IN$_{\text{may}}(b)$ und LIVE-OUT$_{\text{may}}(b)$ für alle Basisblöcke $b \in B$). Zusätzlich werden Informationen über die Menge der Variablen, die eine Anweisung s lesen (USE$_{\text{may}}(s)$) oder schreiben (DEF$_{\text{may}}(s)$) kann benötigt. Wenn in einem Superblock $S = (s_0, ..., s_n)$ eine Anweisung s_i Ergebnisse berechnet, die in allen Anweisungen $s_j \in S_{>i} = \{s_j | s_j \in S \wedge j > i\}$ nicht mehr gelesen werden (USE$_{\text{may}}(s_j) \cap$ DEF$_{\text{may}}(s_i) = \emptyset$), dann wird s_i stückweise durch den Superblock nach hinten verschoben. Wenn dabei ein Austrittspunkt zu einem Block b_{out} mit LIVE-IN$_{\text{may}}(b_{out}) \cap$ DEF$_{\text{may}}(s_i) \neq \emptyset$ passiert wird, so wird eine Kopie von s_i an den Anfang von b_{out} gelegt. Bevor die Verschiebung durchgeführt wird muß außerdem noch überprüft werden, ob keine Write-After-Write-Abhängigkeit vorliegt ($\nexists s_j \in S_{>i}$: DEF$_{\text{may}}(s_j) \cap$ DEF$_{\text{may}}(s_i) \neq \emptyset$) und ob s_i am Ende des Superblocks entfernt werden kann (DEF$_{\text{may}}(s_i) \cap$ LIVE-OUT$_{\text{may}}(s_n) = \emptyset$), denn nur dann ist die Verschiebung potentiell gewinnbringend.

4 IPET-basierte WCEP-Aktualisierung

Um Wechsel des WCEP zu beachten die durch die Änderungen am Code entstehen können, wird nach jeder Optimierung eines einzelnen Superblocks der WCEP neu berechnet. Dies passiert jedoch nicht in Form einer vollen Low-Level WCET-Analyse mit aiT, sondern mit einem angepassten IPET-Verfahren [14], direkt auf der Quellcode-Ebene. Eine Analyse mit aiT würde eine Übersetzung des C Codes nach jeder einzelnen Superblockbildung erfordern, inklusive Codeselektion, Registerallokation und langwieriger aiT-Analyse. Dies würde die Laufzeiten in inakzeptable Höhen treiben, da für manche Programme die aiT-Analyse allein mehrere Minuten benötigt. Daher aktualisieren wir den WCEP

über die Lösung eines ILPs nach dem bekannten IPET-Verfahren. Die Änderungen der Basisblock-WCETs werden über eine Syntaxbaum-basierte Analyse [9] heuristisch erfasst, so daß eine Abschätzung von WCEP-Änderungen ohne Neuübersetzung möglich wird. Um den Schätzfehler hierbei so gering wie möglich zu halten wird alle X Iterationen (Superblockbildungen) ein kompletter aiT-Lauf durchgeführt, der den Fehler wieder auf 0 zurücksetzt. Da die aktualisierten WCET-Informationen nur zum Steuern der Trace-Selektion benötigt werden, und nicht zur Abschätzung der finalen WCET, ist die Sicherheit des Verfahrens gewährleistet.

5 Auswertung

Um die praktische Anwendbarkeit der vorgestellten Optimierungen zu zeigen wurden Versuche mit insgesamt 55 Benchmarks aus verschiedenen anerkannten Standard-Testsuiten (DSPstone, MediaBench, MiBench, MRTC WCET Benchmark Suite, UTDSP) durchgeführt. Die Größe der Benchmarks reicht dabei von 302 Bytes bis 14 kByte.

5.1 Versuchsumgebung

Die vorgestellte Optimierung wurde in den Forschungscompiler *WCC* [11] integriert. Der Compiler übersetzt dabei C Quellcode mit *Flow Facts* (Schleifeniterationsgrenzen, Rekursionsschranken) in die High-Level Darstellung *ICD-C*, die über den Code Selektor in die Low-Level Darstellung *ICD-LLIR* überführt wird. Insgesamt werden bei diesem Prozeß auf dem höchsten Optimierungslevel *O3* auf beiden Ebenen zusammen 43 verschiedene ACET-Optimierungen aktiviert, inklusive einer lokalen CSE und einer Standard-DCE.

Für die Experimente wurde das maximal zulässige Codegrößenwachstum der Traces auf Faktor 5, das der Funktionen auf Faktor 3 und das der Programme insgesamt auf Faktor $2,5$ begrenzt. Die WCET-Analyse mit aiT wurde nur bei jeder vierten Superblockbildung neu gestartet, der Rest der Itertionen wurde mit dem High-Level IPET (Abschnitt 4) abgewickelt. Diese Einstellungen wurden empirisch ermittelt.

5.2 Ergebnisse

WCET Abbildung 3 zeigt die Auswirkungen der verschiedenen CSE Algorithmen auf die WCET für eine Auswahl von 15 Benchmarks und das Durchschnittsergebnis für alle 55 Benchmarks. Gelistet ist jeweils das Ergebnis für die lokale Standard-CSE (1. Balken), das Ergebnis für die WCET-gesteuerte Superblockbildung (ohne CSE) (2. Balken) und das Ergebnis für die Superblockbildung mit Superblock-CSE (3. Balken). Alle Ergebnisse sind relativ zu Optimierungsstufe O3 ohne CSE. Während die lokale CSE nur Verbesserungen von durchschnittlich 3.4% liefert, steigt dies durch die Superblockbildung allein auf 4.0%, im Zusammenspiel mit der Superblock-CSE sogar auf 10.2%. Der Compiler erkennt hierbei

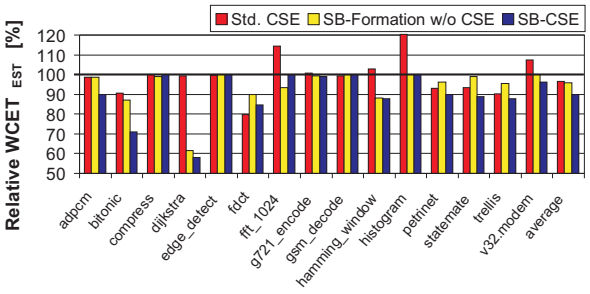


Abb. 3. Relative WCET-Ergebnisse für die Superblock-CSE (100% $\widehat{=}$ O3 ohne CSE)

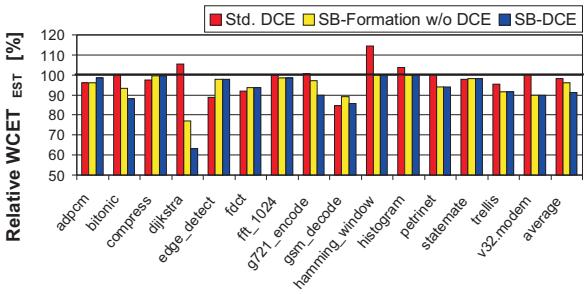


Abb. 4. Relative WCET-Ergebnisse für die Superblock-DCE (100% $\widehat{=}$ O3 ohne DCE)

Fälle in denen die WCET-sensitive Optimierung zu WCET-Verschlechterungen führen würde und bricht die Superblockbildung in diesen Fällen ab. Solche Fälle können vorkommen, da die Effekte der Superblockbildung nicht exakt vorhersagbar sind. In Einzelfällen kann es daher dazu kommen, daß durch Cache-Line-Crossing oder Registerallokations-Fehlentscheidungen die WCET steigen statt sinken würde.

Die Ergebnisse für die DCE sind analog in Abbildung 4 dargestellt. Verglichen mit der Ausgabe auf *O3* ohne DCE erreicht die Standard-DCE eine WCET-Reduktion um 2.0%. Die Superblockbildung allein ist für eine WCET-Reduktion um 4.0% verantwortlich, was durch die Kombination mit der Superblock-DCE auf 8.8% gesteigert wird.

ACET Da die Superblockbildung von WCET-Daten gesteuert wird ist bei der Untersuchung ihrer Auswirkungen auf die ACET ein geringerer Einfluß zu erwarten. In den Versuchen hat sich genau dies bestätigt. Während die Standard-Versionen von CSE und DCE sich für WCET und ACET annähernd gleich verhalten (Unterschied $\leq 1,3\%$), sinkt die Verbesserung durch die WCET-gesteuerten, Superblock-adaptierten Versionen um $5,5\%$ bzw $6,7\%$. Dies betont die Notwendigkeit von WCET-gesteuerten Optimierungen für zeitkritische Anwendungen.

Codegröße Die Superblockbildung führt zwangsläufig zu einer Vergrößerung des Codes. Während bei den oben präsentierten Ergebnissen eine durchschnittliche Codevergrößerung um 23% für die Superblock-CSE und 28% für die Superblock-DCE gemessen wurde, wurde bei Wegnahme der Wachstumsbeschränkungen (s. 5.1) eine durchschnittliche Codevergrößerung um 107% ohne weitere WCET-Verbesserung beobachtet. Die Kontrolle des Codegrößenwachstums ist bei High-Level-Superblöcken daher ein wesentliches Element.

Kompilierdauer Die Laufzeit der Übersetzungen, Optimierungen und Analysen wurde auf einem Intel Xeon System (2.4 GHz, 8 GB RAM) gemessen. Wenn aiT in jedem vierten Durchlauf benutzt wurde, wie in den Ergebnissen oben, erhöht sich die Kompilierzeit um 540%, während bei Nutzung von aiT nach jeder zweiten Superblockbildung eine Erhöhung um 757% auftritt, ohne das sich die WCET-Ergebnisse verbessern. Die IPET-basierte High-Level WCEP-Neuberechnung ist daher in der Lage die Kompilierzeit effektiv zu senken.

Literaturverzeichnis

1. P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Software – Practice & Experience*, vol. 21, no. 12, 1991.
2. R. Kidd and W. W. Hwu, "Abstract Improved Superblock Optimization in GCC," in *GCC Summit*, 2006.
3. J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. 30, no. 7, 1981.
4. P. P. Chang and W. W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode," in *Proc. of MICRO*, 1988.
5. W. W. Hwu, S. A. Mahlke, and W. Y. C. et al., "The Superblock: an Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, vol. 7, 1993.
6. AbsInt GmbH, "Worst-Case Execution Time Analyzer aiT for TriCore," `http://www.absint.com/ait`, 2010.
7. W. Zhao, W. Kreahling, D. Whalley *et al.*, "Improving WCET by Optimizing Worst-Case Paths," in *Proc. of RTAS*, 2005.
8. Y.-T. S. Li, S. Malik, and A. Wolfe, "Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software," in *Proc. of RTSS*, 1995.
9. A. C. Shaw, "Reasoning About Time in Higher-Level Language Software," *IEEE Transactions on Software Engineering*, vol. 15, 1989.
10. A. W. Appel, *Modern Compiler Implementation in C*. New York, NY, USA: Cambridge University Press, 1997.
11. H. Falk, P. Lokuciejewski, and H. Theiling, "Design of a WCET-Aware C Compiler," in *Proc. of ESTIMEDIA*, 2006.
12. Lokuciejewsi, Paul and Falk, Heiko and Marwedel, Peter and Theiling, Henrik, "WCET-Driven, Code-Size Critical Procedure Cloning," in *Proc. of SCOPES*, 2008
13. Kelter, Timon, "Superblock-basierte High-Level WCET-Optimierungen", Diplomarbeit, Lehrstuhl Informatik 12, TU Dortmund, 2009
14. Li, Yau-Tsun Steven and Malik, Sharad, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", in *Proc. of DAC*, 1995