technische universität
dortmund

fakultät für informatik

Lutz Krumme

Dynamische Scratchpad-Allokation von Code und Daten zur WCET-Minimierung

Diplomarbeit

23. August 2010

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik 12
Fakultät für Informatik
Technische Universität Dortmund

Gutachter:
Prof. Dr. Peter Marwedel
Dr. Heiko Falk

fakultät für informatik

GERMANY | D-44221 DORTMUND

An dieser Stelle möchte ich allen Menschen danken, die zur Erstellung dieser Arbeit beigetragen haben.
Der erste Dank gilt meinem Betreuer Dr. Heiko Falk. Mit seiner unermüdlichen Hilfsbereitschaft hat er diese Arbeit erst möglich gemacht.
Daneben danke ich der Forschungsgruppe „Entwurfsautomatisierung für Eingebettete Systeme“ der TU Dortmund für die Erstellung des WCET-Compilers, der die Grundlage für diese Arbeit stellt. Hierbei danke ich insbesondere Jan Kleinsorge und Sascha Plazar für die Beantwortung meiner Fragen und die Hilfe zu meinen Problemen mit Scratchpad-Speicher und DMA-Steuerung.
Ein besonderer Dank gilt Sabine Siebel, die mir aufopferungsvoll zur Seite stand und regelmäßig für die nötigen Motivationsmomente sorgte.
Für das detaillierte Korrekturlesen danke ich Christian Unkelbach.
Schließlich danke ich meinen Eltern für die Geduld, die sie mit mir und meinem Studium hatten.

Inhaltsverzeichnis

1 Einleitung

Eingebettete Systeme spielen im täglichen Leben eine immer größere Rolle. Angefangen bei Haushaltsgeräten, wie Waschmaschinen oder Mikrowellen, über Autos bis hin zu offensichtlicheren Beispielen wie MP3-Playern oder Mobiltelefonen. Für viele eingebettete Systeme gilt, dass sie in irgendeiner Form Realzeit-Bedingungen einhalten müssen. Beispielsweise muss die Airbag-Steuerung eines Autos bei einem Unfall innerhalb eines engen Zeitfensters auslösen, oder ein Mobiltelefon muss in der Lage sein, das gesprochene Wort in Echtzeit für den Versand zu codieren. Die Zeitschranken, die ein eingebettetes System einhalten muss, lassen sich in zwei Klassen unterscheiden. Zum einen in weiche Schranken, deren Nichteinhaltung nur zu einem Qualitätsverlust führen, wie bei einem MP3-Player, der die Daten nicht schnell genug decodiert. Zum anderen in harte Schranken, deren Nichteinhaltung in einer Katastrophe endet, wie zum Beispiel Sensoren eines Flugzeugs, die den Autopiloten steuern [Mar07].

Solche Realzeitbedingungen, insbesondere die harten Schranken, müssen bei jeder möglichen Eingabe in das System eingehalten werden. Das heißt, der ungünstigste Fall eines Programmablaufs muss immer noch innerhalb der Zeitschranke beendet werden. Es handelt sich hierbei um die *Worst Case Execution Time*, kurz WCET.

Compiler für allgemein einsetzbare Systeme, wie der *GNU C Compiler (gcc)*, optimieren den Maschinencode im Regelfall auf die durchschnittliche Laufzeit (*Average Case Execution Time*, ACET) hin. Im positiven Fall verringert sich hierbei auch die WCET oder bleibt unverändert, im negativen Fall kann es allerdings dazu kommen, dass sich die WCET erhöht.

In der Konsequenz muss ein Compiler für zeitkritische eingebettete Systeme bei allen Optimierungen die Auswirkungen auf die WCET hin berücksichtigen.

1.1 Motivation

Eine Möglichkeit, Programmlaufzeiten zu verringern, ist die Nutzung von Speichern, die eine besonders geringe Zugriffszeit haben. Sowohl Caches als auch Scratchpad-Speicher (*Scratchpad Memory/SPM*) gehören zu dieser Art von Speichern. Caches sind in der Regel sehr eng mit der Hardware verzahnt und für den Prozessor unsichtbar. Das heißt, dass die Software ohne spezielle Modifikationen vom Einsatz eines Caches profitiert. Nachteilig an Caches ist einerseits der hohe Hardware-Aufwand bei der Realisierung allgemein, andererseits sind sie für WCET-Abschätzungen nur schwer handhabbar. Die Zugriffszeit auf ein bestimmtes Datum hängt vom aktuellen, hardware-bestimmten Cache-Inhalt ab. Meist wird in diesem Fall von einem Cache-Miss ausgegangen und die Dauer eines Hauptspeicherzugriffs zu Grunde gelegt. Damit kann es zu deutlichen WCET-Überschätzungen kommen. Für einige Architekturen wurden inzwischen Techniken entwickelt, die den

Cache-Inhalt für WCET-Abschätzungen zu weiten Teilen voraussagen. Dies liefert zwar bessere WCET-Schranken, aber noch kaum Optimierungsmöglichkeiten, da der Cache nach wie vor hardware-gesteuert ist.

Während die Cache-Inhalte zur Laufzeit durch die Hardware festgelegt werden, wird der Inhalt von Scratchpad-Speichern bereits zur Entwurfs- oder Compilezeit festgelegt. Scratchpad-Speicher sind, ähnlich wie Caches, sehr nah an den Prozessor angegliedert und bieten eine konstante Zugriffszeit von meist einem Prozessorzyklus. Diese hohe Geschwindigkeit wird allerdings mit der vergleichsweise geringen Größe bezahlt.

Diese Speicher werden über einen eigenen Adressbereich in den Adressraum des Gesamtsystems eingebettet und damit ist der Speicherort eines Datums sicher bekannt. WCET-Abschätzungen können in Scratchpad-Architekturen von gesicherten und konstanten Zugriffszeiten ausgehen. Dadurch, dass der SPM-Inhalt im Vorfeld bekannt ist, kann die WCET durch gezielte Auswahl der im SPM platzierten Daten gesenkt werden. Grundsätzliches Ziel ist es dann, diejenige Speicherbelegung zu identifizieren, bei der die WCET minimal wird.

Das Ermitteln einer solchen optimalen Belegung für einen Scratchpad-Speicher ist zwar NP-vollständig, aber da die Zugriffszeit auf den Speicher sehr niedrig und konstant ist, bieten sie sich sehr für WCET-Optimierungen an. Um der NP-Vollständigkeit aus dem Weg zu gehen, bieten sich approximative Algorithmen an, die in effizienter Laufzeit abgearbeitet werden können.

Auf Grund der unterschiedlichen Charakteristika und bedingt durch die Tatsache, dass häufig getrennte Scratchpad-Speicher zur Verfügung stehen, wurden bereits gentrennte Optimierungsverfahren für Code und Daten entwickelt. Um die eingeschränkte Größe eines Scratchpad-Speichers bestmöglich auszunutzen wurden für beide sowohl statische als auch dynamische Verfahren entwickelt. Statische Optimierungen legen eine bestimmte Allokation des Scratchpad-Speichers fest, die während der gesamten Laufzeit nicht geändert wird. Dynamische Optimierungen versuchen durch die Änderung der Allokation während Laufzeit die WCET weiter zu reduzieren. Diese dynamischen Verfahren leiden meist unter den hohen Kopierkosten, deren Reduktion in dieser Arbeit betrachtet wird.

Die bisher im Rahmen des *WCET-aware C Compilers* (WCC) des Lehrstuhls Informatik 12 der Technischen Universität Dortmund entwickelten dynamischen Scratchpad-Optimierungen leiden insbesondere bei größeren Objekten, wie zum Beispiel Arrays, unter den auftretenden Kopierkosten. Darüber hinaus war die Platzierung des Spillcodes der dynamischen Datenallokation auf Funktions- und Schleifengrenzen eingeschränkt. Ziel dieser Arbeit war es, zunächst diese Platzierung zu flexibilisieren. Des Weiteren sollten sowohl für die Daten- als auch für die Code-Allokation, nach Möglichkeiten gesucht werden, die anfallenden Kopierkosten zu reduzieren.

1.2 Verwandte Arbeiten

Diese Arbeit baut auf den Diplomarbeiten von Jan Christopher Kleinsorge und Felix Rotthowe auf, die statische und dynamische Allokationsmodelle für Code [Kle08] und Daten [Rot08] vorgestellt haben, die auf ganzzahliger linearer Programmierung aufbauten

(*Integer Linear Progamming/ILP*).

Suhendra et al. [SMRC05] haben ebenfalls ein ILP-Modell für eine statische Datenallokation gezeigt. Darüber hinaus haben sie noch einen Branch-and-Bound-Algorithmus sowie eine Greedy-Heuristik vorgestellt, die unausführbare Pfade erkennen und entsprechend engere WCET-Schranken liefern.

Die Greedy-Heuristik wurde dann von Jiang et al. [JHH09] verfeinert, indem die Pfade mit der höchsten WCET gemeinsam betrachtet werden und diejenigen Variablen für den Scratchpad-Speicher ausgewählt werden, die die höchste Einsparung für beide Pfade liefern.

Deverge und Puaut [DP07] haben ein ILP-Modell für dynamische Datenallokation auf Basisblockebene gezeigt, welches auf den maximalen Gewinn des Kontrollpfades mit der höchsten WCET hin optimiert. Nach einer solchen Optimierung wird der ggf. neue WCET-Pfad durch einen iterativen Algorithmus bestimmt und das ILP entsprechend angepasst.

Wu [Wu09] hat einen iterativen Algorithmus zur Allokation von Stack-Daten und globalen Variablen gezeigt. Dieser Algorithmus kann sowohl WCET-basiert als auch ACET-basiert arbeiten.

Ein weiterer Ansatz zur WCET-Reduzierung ist das Cache-Locking. Hierbei wird der Zustand des Daten- oder Befehls-Caches zu für bestimmte Zeiträume eingefroren.

Arnaud und Puaut [AP06] haben einen Algorithmus vorgestellt, der ein Cache-Locking so durchführt, dass die Anzahl der Cache-Misses in der Regel in der selben Größenordnung bleibt, wie ohne Anwendung des Algorithmus. Allerdings ist eine höhere Vorhersagbarkeit des Cache-Inhalts gegeben, mit der dann eine engere WCET-Schranke ermittelt werden kann.

Puaut [Pua06] hat außerdem noch eine Kombination zweier Algorithmen vorgestellt: Beim ersten handelt es sich um einen Greedy-Algorithmus, der basisblockweise arbeitet und den Block mit der maximalen WCET-Einsparung im Scratchpad-Speicher platziert. Weiterhin wird ein genetischer Algorithmus vorgestellt, dessen Individuen Paare von Ladezeitpunkten und Cacheinhalten sind. Als Fitnessfunktion gilt die WCET. Bei einer zufälligen Initialpopulation benötigt der Algorithmus mehrere Stunden, um das Ergebnis des Greedy-Algorithmus zu erreichen. Wird allerdings die Ausgabe des Greedy-Algorithmus als Initialpopulation verwendet, kann mit dem genetischen Algorithmus eine weitere Verbesserung erreicht werden.

Vera et al. [VLX03] haben einen iterativen Algorithmus zum Cache-Locking für Daten präsentiert. Die Datenzugriffe werden in zwei Klassen aufgeteilt. Die erste Klasse bilden konkrete Zugriffe auf einzelne Variablen bzw. Felder. Die zweite Klasse bilden Zugriffe, deren Bereiche abhängig von anderen Daten sind. Zunächst werden die Daten aus der ersten Klasse je nach Zugriffshäufigkeit im Cache platziert, bei Feldern nur die Bereiche, auf die zugegriffen wird. Der verbleibende Platz wird dann auf die Daten der zweiten Klasse verteilt. Hierbei werden die Felder jedoch komplett berücksichtigt. Die ACET steigt teilweise um bis zu 40%, die WCET-Abschätzungen sinken hingegen deutlich.

Neben den Arbeiten von Kleinsorge und Rotthowe existieren offensichtlich keine rein ILP-basierten Ansätze der dynamischen SPM-Allokation zur WCET-Minimierung. Darüber hinaus wurde das DMA-gestützte parallele Kopieren innerhalb der dynamischen

Ansätze noch nicht untersucht.

1.3 Ziele der Diplomarbeit

Die Ziele dieser Arbeit sind, die dynamischen Allokationen von Daten und Code im Rahmen des *WCET-aware C Compiler (WCC)* weiterzuentwickeln. Bei der Datenallokation soll zunächst der bestehende Algorithmus von [Rot08] auf Basisblockebene verfeinert werden.

Aufbauend auf den dynamischen Allokationen von Code und Daten wird versucht, die Kopieraktionen in eine *Direct Memory Access*-Einheit auszulagern und damit parallel zum übrigen Code auszuführen. Die ILP-Modelle der Code- und Daten-Allokation werden dabei um Mechanismen erweitert, die eine gute Positionen zum Einfügen des DMA-Codes festlegen.

Abschließend sollen dann die Ergebnisse der DMA-basierten Allokation mit der statischen Speicherallokation verglichen werden. Zusätzlich soll untersucht werden, ob und inwieweit es generell bei gemeinsam durchgeführter Code- und Datenallokation zu Synergieeffekten kommt.

1.4 Aufbau der Diplomarbeit

Die Diplomarbeit ist wie folgt strukturiert:

Kapitel 2 liefert die formalen Grundlagen für die Entwicklung der weiteren Optimierungen. Zum einen wird das Grundkonzept der WCET-Analyse vorgestellt. Darüber hinaus werden die Grundlagen von *Integer Linear Programming* (ILP) gezeigt.

In **Kapitel 3** wird der am Informatik-Lehrstuhl 12 der TU Dortmund entwickelte und eingesetzte WCET-optimierende Compiler *WCC* und seine Toolchain vorgestellt. Des Weiteren wird die Zielplattform, der Infineon TriCore TC1796 insbesondere hinsichtlich der Scratchpad-Speicher und der DMA-Einheit erläutert.

In **Kapitel 4** wird die verfeinerte dynamische Datenallokation auf Basisblockebene vorgestellt. Im Einzelnen werden das neue ILP-Modell im Detail und die Einbindung der Ergebnisse in den bestehenden WCC erläutert. Die Platzierung des Spillcodes wird gesondert beleuchtet, da je nach Art des Kontrollflusses eine gesonderte Platzierung erforderlich ist.

Kapitel 5 stellt einige Ansätze zur Reduktion der Kopierkosten innerhalb einer dynamischen Scratchpad-Allokation durch. Im Detail wird die Ansteuerung der DMA-Einheit des TC1796 erläutert. Außerdem wird die Analyse der DMA-Einheit hinsichtlich der Dauer eines bzw. mehrerer Kopiervorgänge mit ihren Ergebnissen dargestellt.

Die DMA-unterstützte Scratchpad-Allokation für Code und Daten wird dann schließlich in **Kapitel 6** mit ihren ILP-Modellen vorgestellt. Die Einbindung der neuen Funktionalität in den WCC wird wie in Kapitel 4 auch hier dargestellt.

Im **Kapitel 7** werden dann die Resultate der drei neuen ILP-Modelle anhand einzelner Benchmarks vorgestellt. Hier werden auch die eingangs erwähnten Untersuchungen zur gemeinsamen Code- und Daten-Allokation dargestellt.

Das abschließende **Kapitel 8** fasst schließlich die Ergebnisse der Diplomarbeit zusammen und liefert einen Ausblick auf weitergehende Forschungsmöglichkeiten.

2 Grundlagen

In diesem Kapitel sollen die formalen Grundlagen der WCET und ihrer Analyse dargestellt werden. Im Weiteren wird die ganzzahlige lineare Optimierung formal vorgestellt.

2.1 Worst case execution time

Eingebettete Systeme, die unter Echtzeitbedingungen arbeiten müssen, unterliegen dabei Deadlines. Falls es sich um harte Echtzeitbedingungen handelt, muss jeder Programmdurchlauf unabhängig von der Eingabe zu einem korrekten Ergebnis führen, da das Überschreiten dieser Zeitschranke im schlimmsten Fall zu einer Katastrophe führen kann.

Das zentrale Problem für die exakte Berechnung der WCET eines Programms ist das fehlende Wissen über diejenige Eingabe(n), die zur maximalen Ausführungszeit führen. Voraussetzung für die Lösung dieses Problems wäre die Lösung des Halteproblems [WEE+08] und damit ist diese Problemstellung unentscheidbar. Damit bleibt lediglich die Möglichkeit, das Problem zu entschärfen, also von der exakten Berechnung zu Gunsten einer möglichst guten Schranke abzurücken. Eine Übersicht über die verschiedenen Werte bzw. Schranken liefert Abbildung 2.1.1.

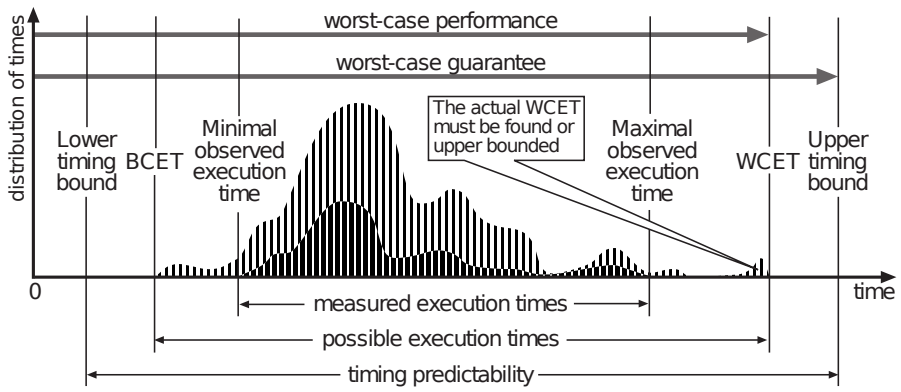


Abbildung 2.1.1: Schranken der Timing-Analyse eines einzelnen Programms. Die untere, dunklere Kurve stellt ein Histogramm der Laufzeiten des Programms mit verschiedenen Eingaben dar. Die obere Kurve stellt die Verteilung über alle möglichen Eingaben dar und ist durch die minimale (BCET) und maximale Laufzeit (WCET) begrenzt. Außerhalb dieses Intervalls liegen dann die Schranken, innerhalb derer die tatsächliche Laufzeit dann liegt. (Abb. aus [WEE+08])

Es gibt zwei grundsätzliche Ansätze, um die WCET eines Programms näherungsweise festzustellen. Eine Möglichkeit ist ein messungsbasiertes Vorgehen. Hier werden Laufzeitmessungen auf verschiedenen Eingaben als Grundlage für die WCET-Kalkulation genommen. Diese Verfahren werden meist für weiche Realzeit-Systeme bzw. für Systeme verwendet, deren Deadline-Einhaltung durch eine Wahrscheinlichkeitsschranke definiert wird [BCP02].

Auf der anderen Seite stehen Verfahren, die die WCET mittels statischer Analyse approximieren. Hier wird vom Binärcode ein Modell abstrahiert, anhand dessen in Verbindung mit einem Modell der Zielarchitektur dann die WCET-Abschätzung getroffen wird. Der Abstraktionsgrad des Modells liefert ein wichtiges Kriterium für die Berechenbarkeit einerseits und die Güte der Abschätzung andererseits [WEE+08].

2.1.1 Analyseansätze

Frühe Ansätze der simulationsbasierten WCET-Analyse sind von atomaren Laufzeiten einzelner Maschinenbefehle ausgegangen und haben dann mit Hilfe der Ausführungshäufigkeit der einzelnen Befehle die WCET eines Programms abgeschätzt [Sha89, PS91]. Die seinerzeit bestimmten Schranken lagen nah an den Messungen, die für verschiedene Eingaben erzielt worden sind. Einer der wenigen Faktoren, die über die dokumentierten atomaren Laufzeiten hinaus beachtet werden musste, waren seinerzeit die Refresh-Zyklen des Hauptspeichers.

Bei heutigen Prozessorarchitekturen mit heterogenen Pipelines, Caches und Branch Prediction ist dieses Modell nicht mehr anwendbar, da die Masse der möglichen Seiteneffekte nicht mehr auf ein solch einfaches Modell heruntergebrochen werden kann. Als Basis für die WCET-Abschätzung dient der *Worst Case Execution Path*, also derjenige Pfad, dessen Ausführung zur maximalen Laufzeit führt.

Dieser Pfad wird aus dem Kontrollflussgraphen des Programms so gebildet, dass schließlich eine Folge von Basisblöcken den WCEP definiert.

Definition 2.1. Ein Basisblock ist die längstmögliche unmittelbare Folge von Instruktionen, die immer gemeinsam von der ersten bis zur letzten Instruktion ausgeführt werden. [Muc04]

Daraus folgt, dass der erste Befehl der Beginn einer Funktion, ein Ziel eines bedingten oder unbedingten Sprungs oder der erste Befehl nach einem nicht genommenen bedingten Sprung ist. Es kann allerdings auch ein Befehl sein, der an der Rücksprungadresse einer Funktion steht. Für die letzte Instruktion heißt dies, dass es sich hierbei um irgendeine Form eines Sprunges handeln muss.

Definition 2.2. Ein Kontrollflussgraph (CFG) ist ein gerichteter Graph $G = (V, E)$, in dem die Knoten die einzelnen Basisblöcke einer Prozedur wiedergeben und genau dann mit einer Kante verbunden werden, wenn ein Ausführungspfad existiert, in dem sie aufeinander folgen. Anschließend werden noch zwei virtuelle Knoten, ein *entry vertex* und ein *exit vertex* definiert, die einerseits mit dem ersten Basisblock und andererseits mit allen die Prozedur verlassenden Basisblöcken verbunden werden. [Muc04]

Um die maximale Laufzeit eines Programms nicht über die verschiedenen Prozeduren gestückelt ermitteln zu müssen, wird ein interprozeduraler Kontrollflussgraph verwendet, der sich über den kompletten Programmcode erstreckt.

Definition 2.3. Ein Programm $PROG = \{P_o, \ldots, P_n\}$ ist die Menge aller möglichen Pfade durch den interprozeduralen Kontrollflussgraph $G = (V, E)$, die in einem eindeutigen Startknoten $v_S \in V$ beginnen und in einem eindeutigen Endknoten $v_T \in V$ enden, so dass gilt $\forall P_i \in PROG : P_i = (v_S, \ldots, v_T)$. [Kle08]

Die eigentliche statische WCET-Analyse verläuft dann nach [BEGL05] grob in drei Schritten. Im ersten Schritt wird der Kontollfluss des zu überprüfenden Programms analysiert. Im Rahmen dieser Analyse werden unter anderem Daten über die Iterationshäufigkeit von Schleifen, über Funktionsaufrufe und über Datenabhängigkeiten des Kontrollflusses gesammelt. Aus den Datenabhängigkeiten des Kontrollflusses können beispielsweise bereits Informationen über unausführbare Pfade extrahiert werden. In einigen Fällen sind dabei manuelle Angaben notwendig. Gerade bei datenabhängigen Schleifengrenzen ist die statische Analyse auf die Annotation von oberen Grenzen angewiesen, um letztendlich eine sinnvolle WCET-Abschätzung liefern zu können.

Im zweiten Schritt wird für atomare Bausteine des Programms, meist sind dies Basisblöcke, das Laufzeitverhalten analysiert. Während der erste Schritt noch weitgehend unabhängig vom Zielsystem war, kommt an dieser Stelle das abstrahierte Modell der Hardware zum Tragen. Hier werden zunächst die Auswirkungen der einzelnen Befehle auf die reine Prozessorlast untersucht. Darüber hinaus werden, je nach Detaillierungsgrad des Modells, die Einflüsse von Hardwareoptimierungen, wie Caches, Pipelines oder auch der Sprungvorhersage berücksichtigt.

Im letzten Schritt werden die in den ersten Schritten gewonnenen Daten kombiniert und derjenige Pfad $P_i \in PROG$ gesucht, der die höchste Ausführungszeit hat und damit den WCEP bildet. Für diese Kombination eignet sich die Formulierung als Problem der ganzzahligen linearen Programmierung, die im folgenden Abschnitt erläutert wird [LM95].

2.2 Integer linear programming

Die WCET-Optimierung unterliegt in seiner Eigenschaft als Optimierungsproblem einer Reihe von Einschränkungen, deren Einhaltung für eine korrekte Optimierung notwendig ist. Als Beispiel sei hier die Größenbeschränkung eines Speicherbereiches zu nennen. Es gilt nun, innerhalb der Lösungen, die den Einschränkungen entsprechen, diejenige zu identifizieren, die das gewünschte Optimierungskriterium weitestgehend erfüllt. In diesem Fall ist dies eine möglichst minimale WCET.

2.2.1 Lineare Optimierung

Eine zentrale Klasse der Optimierungsprobleme sind die linearen Optimierungsprobleme, deren Einschränkungen durch lineare Ungleichungen dargestellt werden können. Weiter-

hin ist die Zielfunktion ebenfalls linear zu wählen. Formal sind i lineare Einschränkungen

$$\sum_j a_{ij} x_j \leq b_i \tag{2.2.1}$$

und eine zu minimierende Zielfunktion z mit

$$\begin{aligned} z &= \sum_j c_j x_j \\ z &\rightarrow min \end{aligned} \tag{2.2.2}$$

mit $a_{ij}, b_i, c_j, x_j, z \in \mathbb{R}$ und a_{ij}, b_i, c_j konstant gegeben. Soll z maximiert werden, kann dies durch Umkehr der Vorzeichen von c_j erreicht werden.

Ein solches lineares Programm (LP) kann drei verschiedene Zustände haben:

1. Keine der Variablenbelegungen von x_j erfüllt alle Einschränkungen $\Rightarrow$ Es gibt keine Lösung.

2. Es gibt Variablenbelegungen von x_j, die alle Einschränkungen erfüllen, allerdings existiert kein Infimum für $z \Rightarrow$ Das Problem ist unbeschränkt.

3. Es gibt eine Variablenbelegung von x_j, für die z minimal wird $\Rightarrow$ Das Problem ist lösbar

Am Beispiel in Abbildung 2.2.1 ist ein zweidimensionales Optimierungsproblem geometrisch dargestellt. Es existieren fünf einschränkende Ungleichungen der Form $\sum_j a_i x_{ij} \leq b_i$, die den grau markierten Bereich umschließen. Sämtliche Werte innerhalb dieses Bereichs liegen im sogenannten Zulässigkeitsbereich. Innerhalb dieses Bereichs sind dann die Werte für x_i zu finden, die z minimal werden lassen. Auf den in den Abbildungen dargestellten Linien der Zielfunktion haben alle Punkte den selben Wert für z. In diesem Fall ist das LP offensichtlich lösbar. Ein Beispiel für ein unbeschränktes LP ist in Abbildung 2.2.2 zu sehen. Ein LP ohne Lösung wäre derart gestaltet, dass sich kein Zulässigkeitsbereich innerhalb der Einschränkungsgeraden findet.

Für allgemeine lineare Optimierungsprobleme existieren seit langem effiziente Algorithmen [Kar84].

2.2.2 Ganzzahlige lineare Optimierung

Ganzzahlige lineare Programme (*integer linear programs*) sind eine Einschränkung des Ergebnisraums der allgemeinen linearen Optimierung auf ganze Zahlen sowohl bei den Konstanten als auch bei den Variablen. Für Formeln und Zielfunktion gelten weiterhin die Formeln 2.2.1 und 2.2.2; nun allerdings mit $x_j \in \mathbb{Z}$ und weiterhin $a_{ij}, b_i, c_j, z \in \mathbb{R}$.

Die Einschränkung des Zulässigkeitsraums auf abzählbar viele Lösungen vereinfacht die Suche nach dem Optimum allerdings nicht – das Gegenteil ist der Fall. Dantzig hat bereits 1957 gezeigt, dass das Rucksack-Problem als ILP-Modell beschrieben werden kann [Dan57]. Diese Reduktion soll hier nun an Hand des kleinen Beispiels von Dantzig umrissen werden. Sei a_j das Gewicht des $j$ten Objekts und sei b_j der relative Nutzen des

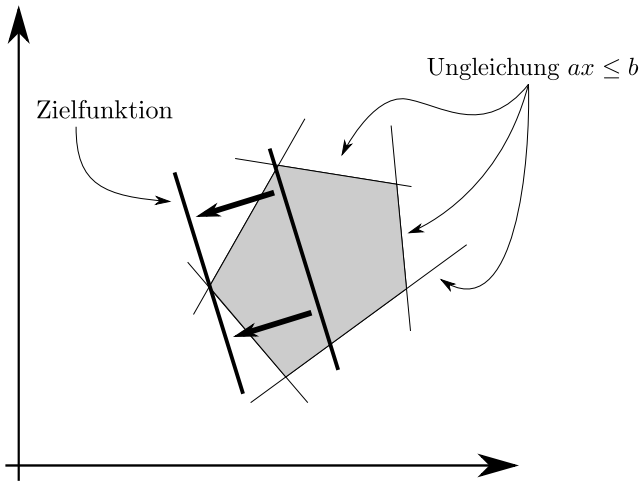


Abbildung 2.2.1: Geometrische Darstellung eines zweidimensionalen linearen Optimierungsproblems

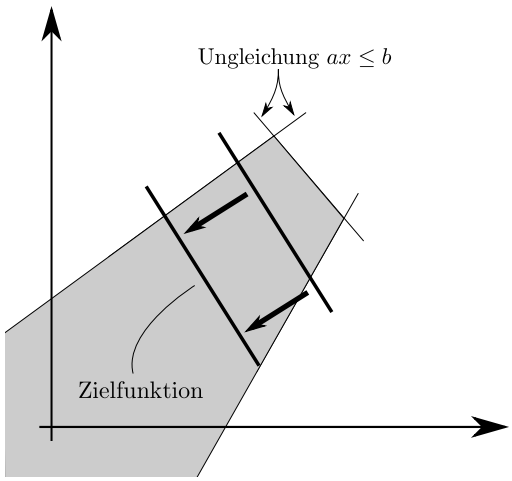


Abbildung 2.2.2: Geometrische Darstellung eines unbeschränkten zweidimensionalen Optimierungsproblems

Objekts im Vergleich zu den übrigen Objekten, die zur eventuellen Mitnahme ausstehen. Nun steht $x_j = 1$ dafür, dass das $j$te Objekt zur Mitnahme ausgewählt wird und $x_j = 0$ dafür, dass es nicht mitgenommen wird. Eine Gewichtsschranke von 30 kg können wir nun mit

$$\sum_j a_j x_j \leq 30$$

mit $x_j \in \{0, 1\}$

darstellen. Als Zielfunktion wählen wir

$$\sum_j b_j x_j = z$$

und versuchen z zu maximieren. Rein formal haben wir an dieser Stelle binäre Variablen mit eingeführt. Allerdings kann man diese auch mit ganzzahligen Variablen und zusätzlichen Einschränkungen modellieren, ohne dem generellen Aufbau zu widersprechen.

Das Rucksackproblem selbst ist NP-vollständig [Weg03] und daraus folgt, dass die Optimierung eines ILPs ebenfalls NP-vollständig ist. Nichtsdestoweniger existieren eine Reihe von approximativen Algorithmen, die in der Lage sind, ein ILP in angemessener Zeit näherungsweise zu lösen [AP01].

Der im Rahmen des Optimiervorgangs eingesetzte ILP-Optimierer *IBM ILOG CPLEX* implementiert selbst neben dem namensgebenden Simplex-Algorithmus eine Reihe dieser Algorithmen, wie zum Beispiel den Barrier-Algorithmus oder einen *Branch-and-Cut*-Ansatz [IBM10].

3 Toolchain & Zielplattform

Die im Rahmen dieser Diplomarbeit entstandenen Optimierungen wurden in eine bereits bestehende Compiler-Umgebung eingebettet. Diese Umgebung soll in diesem Kapitel vorgestellt werden. Zunächst wird aber der Zielprozessor, der Infineon TC1796, ein Chip mit TriCore-Architektur vorgestellt. Dieser ist durch die Compiler-Umgebung des *WCET-aware C Compilers* (WCC) vorgegeben.

3.1 Der TC1796

Der TC1796 ist ein Prozessor aus der TriCore-Serie von Infineon Technologies. Charakteristisch für die TriCore-Serie ist die Kombination eines 32-Bit-Microcontrollers mit einem RISC-Mikroprozessor und einem Digitalen Signalprozessor (DSP). Sein Haupteinsatzgebiet ist der Automobilbereich.

Der TC1796 hat drei parallele Pipelines:

- Die Integer-Pipeline, die logische und arithmetische Operationen auf Daten einschließlich datenabhängiger konditionaler Sprünge ausführt.
- Die Load/Store-Pipeline, die sämtliche lesenden und schreibenden Speicherzugriffe ausführt. Darüber hinaus behandelt sie Adressarithmetik, unbedingte Sprünge, Funktionsaufrufe und Kontextwechsel.
- Die Loop-Pipeline, eine sekundäre Pipeline, die in erster Linie die *loop*-Instruktion behandelt und *zero-overhead-loops* ermöglicht.

Unter bestimmten Umständen können die ersten beiden Pipelines zeitgleich mit einem Befehl versorgt werden und arbeiten diese dann parallel ab. Damit können dann im Idealfall zwei Instruktionen pro Prozessorzyklus verarbeitet werden. Im Regelfall werden beide Pipelines angehalten, falls Wartezyklen notwendig werden sollten.

Der TriCore-Befehlssatz zeichnet sich durch eine 32-Bit-Architektur aus. Er bietet 32 Universalregister mit einer Breite von 32 Bit. Diese sind aufgeteilt in 16 Daten- und 16 Adressregister. Die Register können paarweise als 64-Bit-Register angesprochen werden.

Neben den Universalbefehlen, wie Lade- und Speicherbefehlen bietet er auch Befehle aus dem DSP-Bereich, wie beispielsweise gesättigte Arithmetik, *Multiply-Accumulate*-Befehle oder sog. *Zero Overhead Loops*. Viele Befehle existieren neben ihrer 32-Bit-Version auch in einer 16-Bit-Version mit reduzierter Parameterauswahl, um sowohl die Codegröße als auch den Energieverbrauch zu reduzieren.

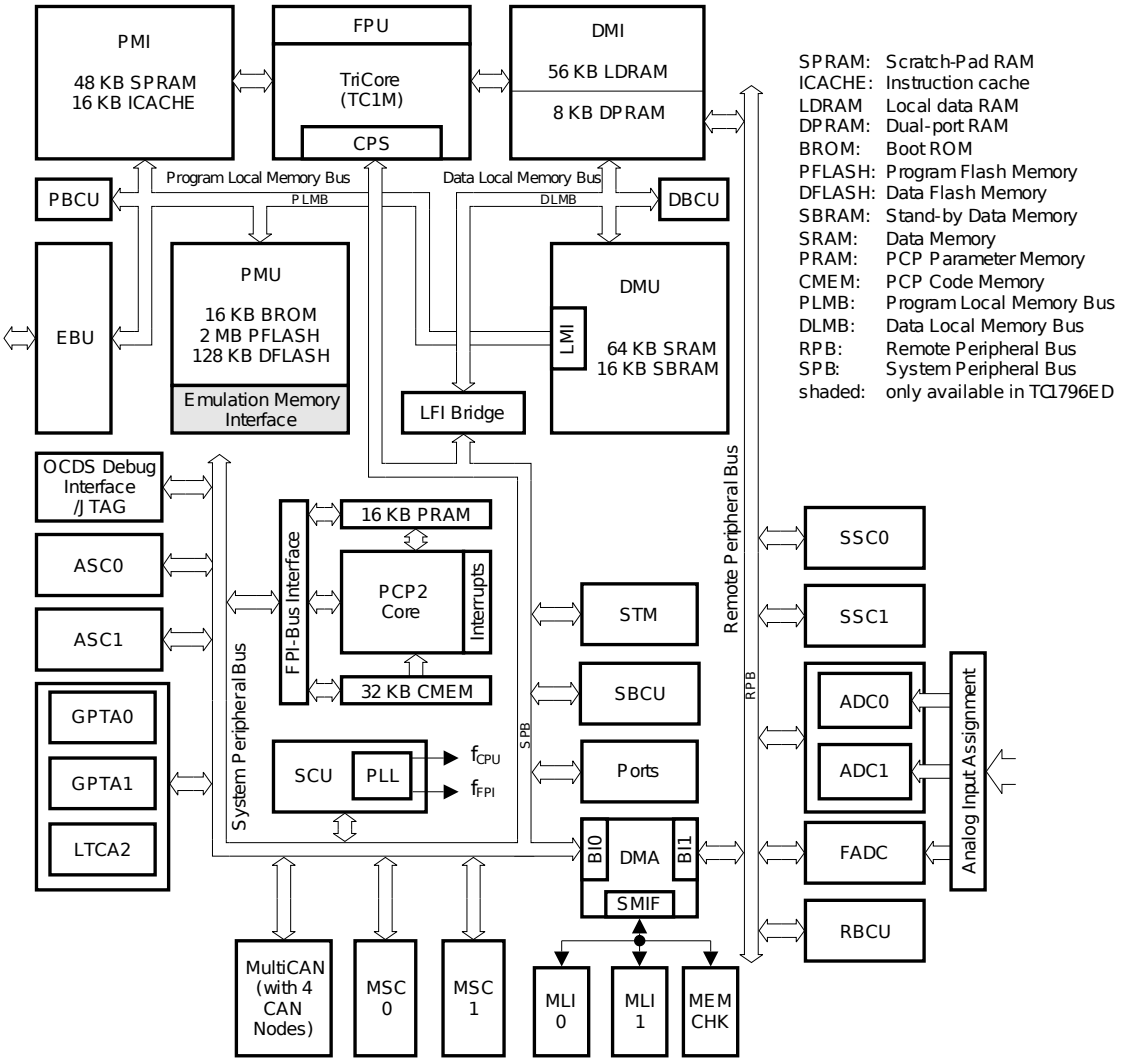


Abbildung 3.1.1: Blockdiagramm des TC1796 aus [Inf08a]

Der TC1796 hat auf dem Chip bereits eine Reihe von Speichern integriert. Die logische Anordnung dieser Speicher kann dem Blockdiagramm Abbildung 3.1.1 entnommen werden.

- 2 MB Program Flash Memory (PFLASH)
- 128 KB Data Flash Memory (DFLASH)
- 136 KB Data Memory (LDRAM, SRAM, SBRAM)
- 8 KB Dual-Ported Memory (DPRAM)
- 48 KB Code Scratchpad Memory (SPRAM)
- 16 KB Instruction Cache (ICACHE)
- 15 KB BootROM (BROM)

Für die in dieser Arbeit vorgestellten Optimierungen ist in erster Linie der Scratchpad-Speicher (SPRAM, LDRAM) von Belang.

3.1.1 Scratchpad-Speicher

Unmittelbar mit dem Prozessorkern verbunden sind das *Program Memory Interface* (PMI) und das *Data Memory Interface* (DMI). Im PMI sind insgesamt 64 KB Speicher enthalten, die sich in 16 KB Instruction Cache und 48 KB Scratchpad-Speicher aufteilen. Dieser Speicher kann innerhalb eines einzelnen Prozessorzyklus ausgelesen werden, und die Pipeline kann ohne angehalten zu werden weiterlaufen. Im Falle eines Cache-Misses oder eines Zugriffs auf externe Speicher muss die Pipeline so lange angehalten werden, bis das erforderliche Datum zur Verfügung steht.

Im DMI sind ebenfalls 64 KB Scratchpad-Speicher enthalten, hiervon können 8 KB zusätzlich über den externen Peripheriebus angesprochen werden. Die Zugriffszeiten liegen, analog zum Scratchpad-Speicher für Code, bei einem Prozessorzyklus. Sowohl DMI als auch PMI dienen dem Prozessorkern neben ihrer Eigenschaft als schnelle Speicher als Schnittstelle zu den Bussen, über die auf die weiteren Daten- bzw. Instruktionsspeicher zugegriffen werden kann.

Beide Scratchpad-Speicher sind normal im linearen Adressraum eingebettet. Für den Zugriff können daher die üblichen Instruktionen der Load/Store-Hierarchie verwendet werden.

3.1.2 DMA-Einheit

Der TC1796 besitzt eine DMA-Einheit mit 16 Kanälen, die über den System Peripheral Bus auf Daten- und Instruktionsspeicher zugreifen kann. Wie aus dem Blockdiagramm in Abbildung 3.1.2 ersichtlich, teilen sich die 16 Kanäle auf zwei unabhängige Subblöcke mit jeweils 8 Kanälen auf.

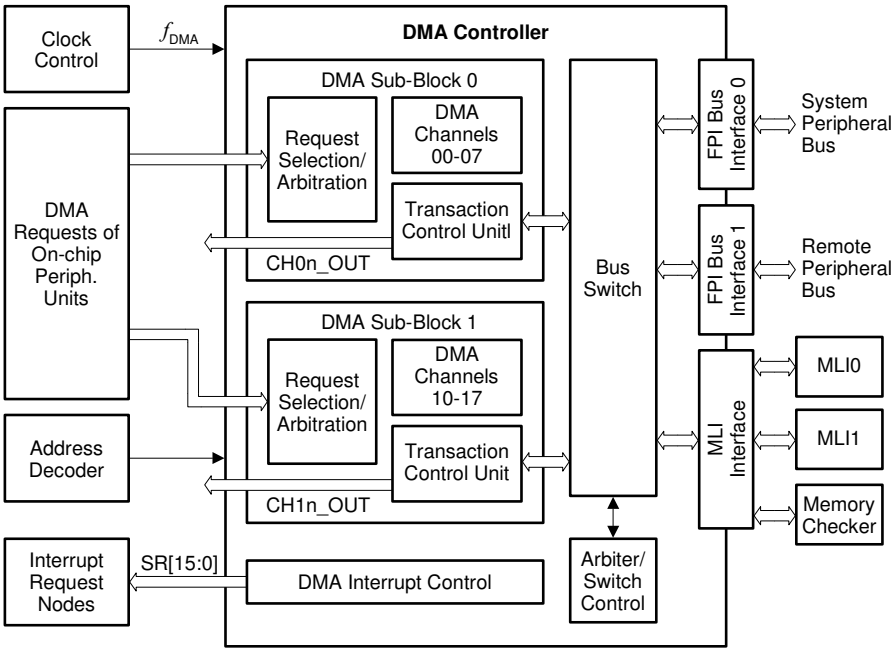


Abbildung 3.1.2: Blockdiagramm der DMA-Einheit des TC1796 aus [Inf08a]

Über jeden Kanal lässt sich genau eine DMA-Transaktion durchführen. Eine DMA-Transaktion besteht aus bis zu 511 DMA-Transfers. Ein DMA-Transfer besteht aus 1, 2, 4, 8 oder 16 DMA-Moves, und ein dieser besteht aus der Kopie eines 8, 16 oder 32 Bit breiten Datums von einer Speicheradresse zu einer anderen. Die Maximalgröße einer einzelnen DMA-Transaktion ist damit auf die Kopie von 32 704 Byte beschränkt. Adressiert werden kann der komplette TriCore-Adressraum mit 4 GB.

Die DMA-Einheit wird über einen ihr zugeordneten Registersatz gesteuert. Dieser ist, ebenso wie die Scratchpad-Speicher, im linearen Adressraum eingebettet. Im Gegensatz zu den Universalregistern können diese also nur über die Speicheradresse ausgelesen oder beschrieben werden. Ein Lese- oder Schreibvorgang auf diese Register kann ohne ein Anhalten der Pipeline durchgeführt werden. Im Sinne des Timings verhalten sie sich also wie normale Universalregister.

3.2 aiT

Im Rahmen der vorhandenen Compilerumgebung wird zur Analyse der WCET das Tool *aiT* der Angewandte Informatik GmbH, Saarbrücken verwendet [Abs10]. Es führt eine statische Analyse des gegebenen Binärcodesauf einem sehr detaillierten Modell der Zielarchitektur durch. Die Analyse schließt das Verhalten des Stacks, der Pipelines und des Caches mit ein. Darüber hinaus wird der Ausführungskontext des Codes durchweg in die Analyse miteinbezogen. Dadurch kann eine hohe Güte der durch aiT getätigten WCET-Abschätzungen erreicht werden.

Die Analyse wird schrittweise durchgeführt. Ausgangspunkt ist der von einem Compiler erstellte Binärcode.

1. Aus dem Binärcode wird der ursprüngliche Kontrollfluss wiederhergestellt. Dieser liegt dann in Form eines Kontrollflussgraphen vor.

2. Auf diesem Kontrollflussgraphen wird eine Wertanalyse durchgeführt, die einerseits die Ziele von Speicherzugriffen bestimmten Adressen oder Adressbereichen zuordnen. Ebenfalls in diesem Schritt werden Iterationsobergrenzen für die erkannten Schleifen berechnet.

3. Mit den in der Wertanalyse berechneten Speicherzugriffszielen kann nun eine Cache-Analyse durchgeführt werden. Für diese Cache-Analyse werden Schleifen und rekursive Funktionen mittels *virtual inlining, virtual unrolling* virtuell abgerollt und die Cache-Analyse kann abhängig vom einzelnen Ausführungskontext arbeiten. Dies führt dazu, dass eventuelle Cache Misses in der ersten Iteration sich nicht auf die WCET aller weiteren Iterationen auswirkt.

4. Mit den in der Cache-Analyse gewonnenen Daten kann dann in der Pipeline-Analyse für jeden Basisblock in jedem Ausführungskontext die WCET berechnet werden.

5. Abschließend wird aus den Ergebnissen der Kontrollfluss in ein ILP-Modell synthetisiert. Die Lösung dieses ILP-Modells liefert dann die Gesamt-WCET des Programms.

Alle Zwischenschritte nutzen zur Ein- bzw. Ausgabe die *Control-Flow Representation Language* (CRL). In dieser werden je nach Analysefortschritt unter anderem der Kontrollfluss, globale Daten (z.B. Einstellungen der Zielarchitektur), Cache- und Pipeline-Zustände gespeichert. Diese liegen im Klartext vor und können zwischen den einzelnen Analyseschritten von externen Tools gelesen oder bearbeitet werden. Da die CRL erweiterungsoffen ist, können externe Tools, wie zum Beispiel der WCC eigene Informationen in dieser Sprache formulieren, ohne die Analysekette zu stören.

Für die Handhabung der CRL-Dateien steht seitens AbsInt eine Bibliothek zur Verfügung, die in eigene Tools eingebunden werden kann [Abs].

3.3 WCC

Der *WCET-aware C Compiler* (WCC) ist ein Compiler, der seit 2005 am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelt wird. Zielarchitekturen des Compilers sind der Infineon TC1796 und TC1797. Hauptzielsetzung der Optimierungen innerhalb des Compilers ist die Reduktion der WCET. Der Compiler ist eng mit dem statischen Analysetool aiT, welches im Abschnitt 3.2 beschrieben wurde, verzahnt. Durch diese Verzahnung stehen den Optimierungen bereits vor Abschluss der Übersetzung nicht nur die Gesamt-WCET, sondern auch die Laufzeiten einzelner Abschnitte bis hinunter auf Basisblockebene zur Verfügung.

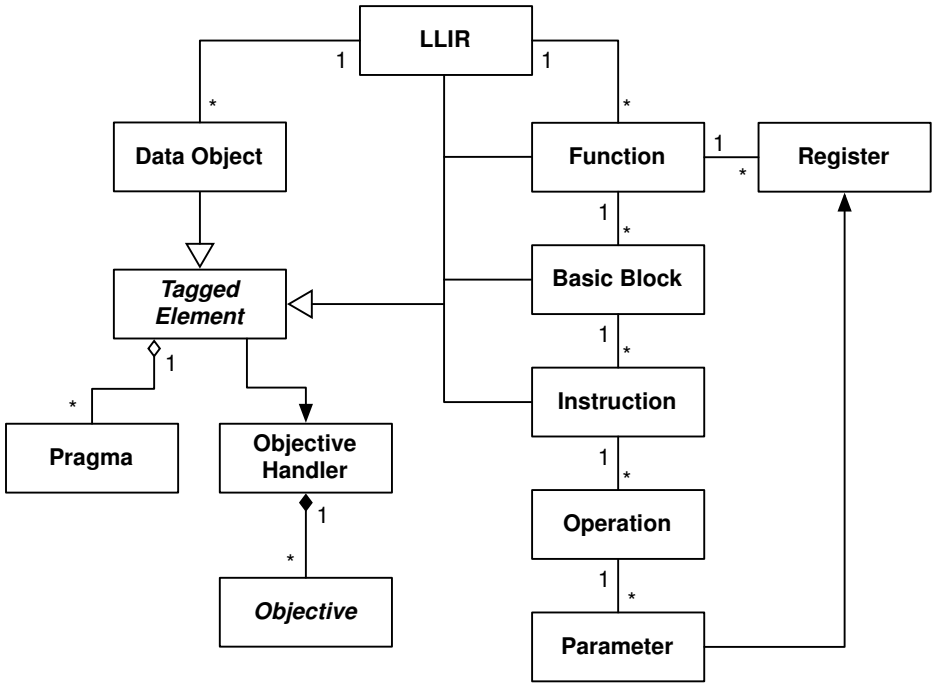


Abbildung 3.3.1: Klassendiagramm der LLIR aus [Rot08]

Der Compiler arbeitet mehrstufig [FL10]. Der Parser übersetzt den ANSI-C-Code der Eingabedateien in eine High-Level-Zwischendarstellung für das ICD-C-Framework [Inf10a]. Diese Zwischendarstellung ermöglicht erste Optimierungen, die noch unabhängig von der gewählten Zielarchitektur sind, wie zum Beispiel Klonen oder Positionierung von Funktionen. Nach diesen High-Level-Optimierungen wird die Zwischendarstellung vom Code-Selector in Assembler-Code übersetzt, der dann in einer Low-Level-Zwischendarstellung, der ICD-LLIR [Inf10b], eingebettet ist.

Diese Datenstruktur wird vom WCC in eine CRL2-Datei konvertiert, mit der dann aiT eine erste WCET-Berechnung durchführen kann. Diese ist für sämtliche WCET-gerichteten Optimierungen, wie die hier vorgestellten Scratchpad-Optimierungen, notwendig, da in ihr das Modell für den zu optimierenden WCEP enthalten ist. Nach Abschluss der Low-Level-Optimierungen wird schließlich aus der ICD-LLIR ausführbarer Maschinencode generiert.

3.3.1 Low-Level Intermediate Representation

Die *Low-Level Intermediate Representation* (LLIR) des ICD bildet die Datenstruktur, auf der die in dieser Arbeit vorgestellten Optimierungen durchgeführt werden. In ihr wird der durch den Code Selector generierte Assembler-Code von Funktionsebene bis auf Instruktionsparameterebene heruntergebrochen. Das Klassendiagramm in Abbildung 3.3.1 zeigt die für die Scratchpad-Optimierungen notwendigen Teil der LLIR.

Seien nun die Code-Bestandteile der LLIR kurz erklärt. Eine *Function* repräsentiert

eine logische Einheit innerhalb des Assembler-Codes. Insbesondere werden Funktionen auf Quellcode-Ebene auf dieser Ebene repräsentiert. Umgekehrt ist diese Verknüpfung jedoch nicht gegeben, wie später in Abschnitt 5.1 deutlich wird. Jede *Function* enthält einen Zeiger auf den jeweiligen ersten *Basic Block*.

Die Basisblöcke einer Funktion bilden eine doppelt verkettete Datenstruktur, das heißt, jeder *Basic Block* enthält sowohl Zeiger auf seine Vorgänger als auch auf seine Nachfolger im Kontrollfluss. Innerhalb der Basisblöcke werden die einzelnen *Instructions* als Listen verwaltet. Die Reihenfolge innerhalb der Listen spiegelt auch die spätere Platzierung im Maschinencode wieder.

Da es sich bei der TriCore-Architektur um eine RISC-Architektur handelt, ist die Beziehung zwsichen *Instruction* und *Operation* eine 1:1-Beziehung. Mehrere Operationen innerhalb einer Instruktion werden lediglich bei VLIW-Architekturen benötigt, die innerhalb einer Instruktion mehrere Operationen parallel bearbeiten können. Damit bleiben nur noch die einzelnen *Parameter* einer Operation, die dann aus Registern, Konstanten, Labels oder Operatoren bestehen können. Operatoren kennzeichnen dabei die genaue Variante eines Maschinenbefehls, beispielsweise Postinkrementoperatoren oder spezielle Adressierungsarten.

4 Dynamische Datenallokation auf Basisblock-Ebene

Die bisher bestehende dynamische Allokation von Daten für Scratchpad-Speicher von Rotthowe [Rot08] war hinsichtlich der Platzierung des Kopiercodes auf Grenzen von Schleifen und Funktionen beschränkt. Bei Programmen, die starken Gebrauch von *Function Inlining* betreiben, geht die Flexibilität der dynamischen Datenallokation deutlich zurück. Auch im Hinblick auf die im weiteren Verlauf geplanten DMA-gestützten Allokation erscheint es sinnvoll, das Allokationsmodell auf Basisblockebene zu verfeinern. Als Vorlage für das nun vorgestellte ILP-Modell dient das Allokationsmodell von Kleinsorge [Kle08].

4.1 ILP-Formulierung

Als grundlegende Datenstruktur für das folgende ILP-Modell dient ein interprozeduraler Konstrollflussgraph $IPCFG = (V, E)$, dessen Knoten $v_k \in V$ einzelne Basisblöcke und dessen Kanten $e_j \in E$ bestehende Kontrollflüsse zwischen den einzelnen Basisblöcken darstellen.

Aus dem IPCFG wird dann ein gerichteter, azyklischer Graph $G_{DAG} = (V, E \setminus E_b)$ entwickelt, mit Hilfe dessen dann die WCET der einzelnen Pfade berechnet werden kann. E_b kennzeichnet hier die Menge der Kanten, die eine Schleife symbolisieren und den IPCFG zyklisch machen, den sogenannten Back-Kanten. Zur Identifikation dieser Back-Kanten wird wie folgt nach [Muc04] verfahren.

Definition 4.1. Ein Knoten $v_a \in V$ dominiert einen Knoten $v_b \in V$ genau dann, wenn jeder Pfad vom Eingangsknoten zu v_b den Knoten v_a enthält.

Hieraus folgt, dass aus dem Graphen ein Dominanz-Baum, ausgehend vom Eingangsknoten, gebildet werden kann. In Abbildung 4.1.2 ist der Dominanzbaum des Graphen in Abbildung 4.1.1 zu sehen.

Definition 4.2. Eine Back-Kante $e_b \in E_b$ ist eine Kante, deren Ziel $v_t \in V$ den Ursprung $v_s \in V$ dominiert.

Im hier gegebenen Beispiel dominiert der Knoten v_4 den Knoten v_6, und damit ist die Kante $e = (v_6, v_4)$ eine Back-Kante. Ein Programm ist mit dem folgenden ILP genau dann analysierbar, wenn der zugehörige Kontrollflussgraph reduzierbar ist.

Definition 4.3. Ein Flussgraph $G = (V, E)$ ist genau dann reduzierbar, wenn E so in die disjunkten Teilmengen der Forward-Kanten E_f und der Back-Kanten E_b zerlegt werden

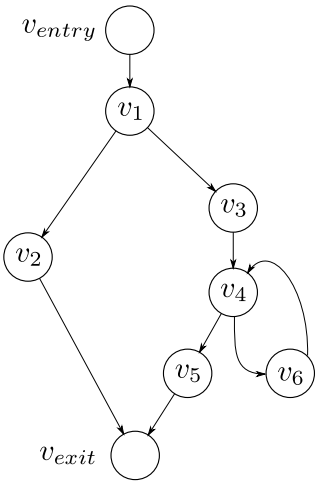


Abbildung 4.1.1: Ausgangskontrollflussgraph, der noch nicht azyklisch ist.

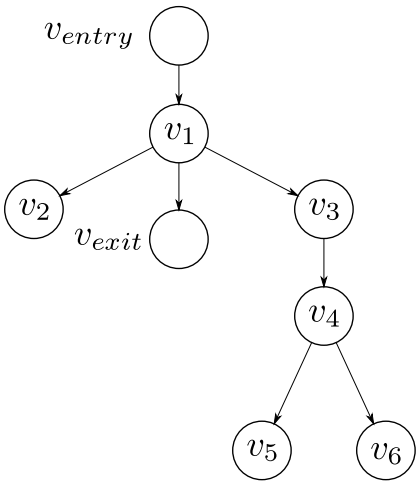


Abbildung 4.1.2: Dominanzbaum des CFG aus Abbildung 4.1.1.

kann, dass (V, E_f) einen gerichteten, azyklischen Graph darstellt, in dem jeder Knoten vom Eingangsknoten aus erreicht werden kann und die Kanten aus E_b der Definition 4.2 entsprechen.

Sollte der Graph im Sinne dieser Definition nicht reduzierbar sein, handelt es sich um einen Graphen, der Schleifen mit mehreren Eintrittspunkten besitzt. Derlei Konstruktionen sind mit C *goto*-Statements zwar möglich, werden jedoch sehr selten genutzt und gelten seit langem als unglücklich und umgehbar [Dij68].

Schleifen werden in einer dedizierten Art in das ILP eingefügt. Schleifen können als Teilgraph des IPCFG aufgefasst werden. Sie haben einen Anfangs- und einen Endknoten, die durch eine Back-Kante vom End- zum Anfangsknoten im Sinne der Definition 4.2 verbunden sind. Sämtliche Schleifen eines Programms sind über ihren Eingangsknoten, den Schleifenkopf, identifizierbar. Die restlichen Knoten einer Schleife bis zum Endknoten werden als Schleifenrumpf bezeichnet.

In der folgenden Aufschlüsselung des ILP-Modells kennzeichnen sich sowohl die binären als auch die ganzzahligen Variablen mit einem Kleinbuchstaben und einem bzw. zwei Indizes: a_c^b. Konstanten, deren Wert im Vorfeld bestimmt wird, werden in Funktionsschreibweise mit einem bzw. zwei Parametern dargestellt: $a(b, c)$.

4.1.1 Konstanten

Für die Bildung des ILP-Modells müssen zunächst einige Konstanten festgelegt werden, auf deren Basis dann die WCET-Schranke berechnet werden kann. Zum einen spielt die Größe der einzelnen Datenobjekte $d \in \mathit{data\_objects}$ eine Rolle bei der Auslastung des SPM.

Definition 4.4. Sei $\mathit{size}(d_i)$ die Größe der Variable d_i in Byte.

Damit die WCET des Gesamtprogramms aus dem ILP hervorgeht, müssen die Laufzeiten der einzelnen Basisblöcke ebenfalls im ILP-Modell enthalten sein:

Definition 4.5. Sei $c(v_k)$ die WCET des durch v_k repräsentierten Basisblocks ohne Variablen im Scratchpad-Speicher.

Für die Auswahl der im SPM zu lagernden Datenobjekte spielt der Nutzen in Form von Zeiteinsparung die Hauptrolle. So wird für alle möglichen Kombinationen von Knoten v_k und Variablen d_i der Gewinn $\mathit{gain}(d_i, v_k)$ im Voraus berechnet.

Definition 4.6. Sei $\mathit{gain}(d_i, v_k)$ der erzielte Laufzeitgewinn bei einer Auslagerung von d_i während der Ausführung von v_k.

Dieser Gewinn definiert sich aus der Differenz zwischen $c(v_k)$ und der WCET des durch v_k repräsentierten Basisblocks mit der Variable d_i im SPM. Um den WCET-Beitrag von Schleifen gesondert korrekt berechnen zu können, muss zu jeder Schleife die maximale Anzahl an Iterationen im ILP bekannt sein.

Definition 4.7. Falls v_k ein Schleifenkopf ist, dann sei $\mathit{loopcount}(v_k)$ die maximale Anzahl an Schleifeniterationen. Sonst sei $\mathit{loopcount}(v_k) = 1$.

Die Schleifeniterationsvariable wird ausschließlich im Zusammenhang mit Schleifen verwendet, weswegen sie auch in Abhängigkeit des Schleifenkopfes festgehalten werden kann. Der Standardfall eines Knotens, der kein Schleifenkopf ist, sei hier nur der Vollständigkeit halber mit aufgeführt. Schließlich muss noch der Einfluss der Kopieroperationen auf die WCET mitberücksichtigt weden. Hierfür werden die Konstanten $c_{load}(d_i)$ und $c_{store}(d_i)$ mit definiert.

Definition 4.8. Sei $c_{load}(d_i)$ die Dauer eines Kopiervorgangs des Datums d_i in den Scratchpad-Speicher.

Definition 4.9. Sei $c_{store}(d_i)$ die Dauer eines Kopiervorgangs des Datums d_i in den Hauptspeicher.

Die Werte für c_{load} und c_{store} lassen sich auf Grund der Ergebnisse der Pipeline-Simulation von Rotthowe wie folgt abschätzen:

	c_{load}	c_{store}
Konstanten (read-only-Speicher)	$15 \cdot size(d_i) + 20$	0
Variablen (SRAM)	$6 \cdot size(d_i) + 20$	$4 \cdot size(d_i) + 20$

Tabelle 4.1: Berechnung der Kopierkosten

Die Werte setzen sich zusammen aus den größenabhängigen Bestandteilen, die die Speicherzugriffe enthalten. Die übrigen 20 Zyklen ergeben sich aus der Laufzeit für den Funktionaufruf, den Rücksprung und die Vorbereitung der eigentlichen Kopierschleife.

4.1.2 Variablen

Für die basisblockweise Datenallokation werden zunächst vier verschiedene Entscheidungsvariablen definiert, die nach Optimierung des ILPs die Speicherkonfiguration wiederspiegeln. Die zentrale Entscheidungsvariable s_k^i definiert den Speicherort einer Variable an einem bestimmten Programmpunkt.

$$s_k^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ im Knoten } v_k \text{ im SPM liegt} \\ 0, & \text{sonst} \end{cases}$$

Um die Konsistenz der Speicherorte über den Programmverlauf vollständig sicherzustellen, müssen noch eine Reihe von Entscheidungsvariablen definiert werden, die den Zustand auf den Kanten repräsentieren.

$$x_j^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ auf der Kante } e_j \text{ im SPM liegt} \\ 0, & \text{sonst} \end{cases}$$

$$y_j^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ auf der Kante } e_j \text{ ins SPM geladen wird} \\ 0, & \text{sonst} \end{cases}$$

$$z_j^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ auf der Kante } e_j \text{ in den Hauptspeicher} \\ & \text{geschrieben wird} \\ 0, & \text{sonst} \end{cases}$$

4.1.3 Constraints

Zunächst muss für alle Knoten des Graphen sichergestellt werden, dass die Größe des Scratchpad-Speichers stets größer und gleich der Größe der Variablen ist, die in ihm ausgelagert werden sollen:

$$\forall v_k \in V : \sum_i s_k^i \cdot size\,(d_i) \leq size\,(SPM) \tag{4.1.1}$$

Weiterhin kann eine Variable d_i auf jeder Kante e_j nur in genau einem Zustand sein:

1. Sie liegt im SPM, also $x_j^i = 1$
2. Sie wird in den SPM kopiert, also $y_j^i = 1$
3. Sie wird in den Hauptspeicher kopiert, also $z_j^i = 1$
4. Sie liegt im Hauptspeicher $x_j^i = y_j^i = z_j^i = 0$

Dies wird durch folgenden Constraint sichergestellt:

$$\forall d_i \in data\_objects,\, e_j \in E : \; x_j^i + y_j^i + z_j^i \leq 1 \tag{4.1.2}$$

Ferner muss die Konsistenz des SPM-Inhalts noch gesichert werden. Das heißt, dass ein Datum d_i an einem Knoten v_k nur dann im Scratchpad-Speicher sein kann, wenn es entweder auf den eingehenden Kanten bereits dort ist oder dort kopiert wird (4.1.3). Wenn in einem Knoten v_k ein Datum d_i im Scratchpad-Speicher liegt, dann muss es auf allen ausgehenden Kanten ebenfalls dort liegen oder zurück in den Hauptspeicher kopiert werden (4.1.4).

$$\forall d_i \in data\_objects \forall v_k \in V \setminus \{v_{entry}\}\, \forall e_j \in In(v_k) : \; s_k^i = x_j^i + y_j^i \tag{4.1.3}$$

$$\forall d_i \in data\_objects \forall v_k \in V \setminus \{v_{exit}\}\, \forall e_j \in Out(v_k) : \; s_k^i = x_h^i + z_h^i \tag{4.1.4}$$

Beim Wurzel- und Senken-Knoten des IPCFG wird dieser Constraint mangels vorhandener Kanten jeweils wechselweise nicht eingefügt.

4.1.4 Flow-Constraint und Zielfunktion

Die Bildung der WCET anhand eines IPCFG geschieht über eine Kette von Ungleichungen, die sich entlang des WCEP aufreihen. Sei nun v_0 der Startknoten und v_t der Endknoten eines Kontrollflussgraphen. Dann sei $P = (v_0, \ldots, v_t)$ ein Pfad ohne Schleifen oder Funktionsaufrufe durch das Programm. Die Flussvariable, die sich durch den WCEP zieht ist w_k. In ihr ist die kumulierte WCET vom Knoten v_k bis zum Endknoten v_t gespeichert. Die WCET baut sich über diese Ungleichungen vom Endknoten entgegen der Ausführungsreihenfolge bis zum Startknoten hin auf.

$$\begin{aligned} w_k \quad \geq \quad & \underbrace{w_{k+1} + c\left(v_k\right)}_{(1)} - \underbrace{\sum_i gain\left(d_i, v_k\right) \cdot s_k^i}_{(2)} + \\ & \underbrace{\sum_i c_{load}\left(d_i\right) \cdot y_e^i + \sum_i c_{store}\left(d_i\right) \cdot z_e^i}_{(3)} \end{aligned} \tag{4.1.5}$$

Eine solche Ungleichung wird für jeden v_k nachfolgenden Knoten im reduzierten CFG in das ILP eingefügt. Durch die Formulierung als Ungleichung wird sichergestellt, dass im Falle mehrerer Ungleichungen an einem Knoten die kumulierte WCET das Maximum einnimmt. Die in w_k kumulierte WCET des Knotens v_k ist damit mindestens so groß, wie die Summe aus der Laufzeit des atomaren Knotens $c\left(v_k\right)$ und der kumulierten WCET des Nachfolgeknotens (1). Von dieser wird dann der durch die SPM-Allokationen erzielte Laufzeitgewinn abgezogen (2). Wieder hinzugezählt werden die Kosten von Kopien, die auf der Kante zwischen den betrachteten Knoten durchgeführt werden (3).

Für den Endknoten gilt mangels ausgehender Kante dann

$$w_t = c\left(v_t\right) - \sum_m gain\left(d_m, v_t\right) \cdot s_t^m \tag{4.1.6}$$

Dies führt dazu, dass die Gesamt-WCET schließlich in w_0 festgehalten wird und die Zielfunktion dann

$$w_0 \rightarrow min \tag{4.1.7}$$

Für den Fall, dass im Programmfluss Schleifen oder Funktionsaufrufe enthalten sind, müssen diese gesondert berücksichtigt werden und in das Gesamt-ILP eingebunden werden. Das Vorgehen wird in den folgenden beiden Abschnitten erläuert.

Schleifen

Schleifen innerhalb des Kontrollflussgraphen erfordern eine gesonderte Betrachtung, da es vorkommen kann, dass der Schleifenrumpf oder Teile davon keine Verbindung zum Endknoten des Programms v_{exit} über Forward-Kanten haben. Für die Lösung dieses Problems werden dem Schleifenkopf nun zwei Formeln zugewiesen. Einerseits wird die

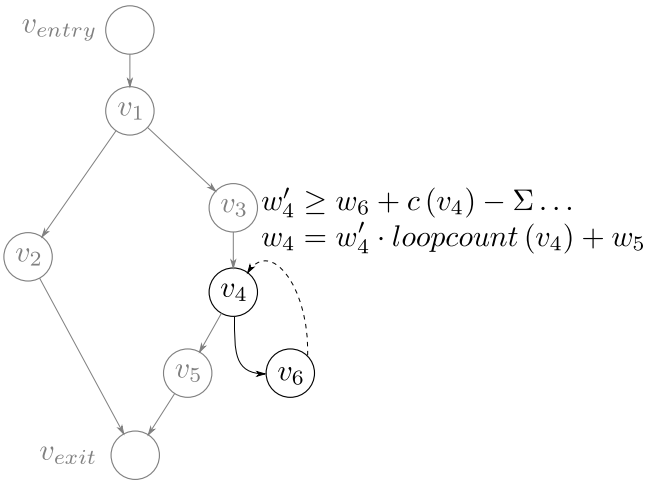


Abbildung 4.1.3: Beispiel für die Einbindung von Schleifen in das ILP

WCET einer einzelnen Schleifeniteration analog zu Formel 4.1.5 in einer zusätzlichen Variable w'_i berechnet:

$$w'_{head} \geq w_{body} + c(v_{head}) - \sum_i gain(d_i, v_{head}) \cdot s^i_{head} + \sum_i c_{load}(d_i) \cdot y^i_e + \sum_i c_{store}(d_i) \cdot z^i_e \tag{4.1.8}$$

Dabei ist v_{head} der Schleifenkopf, v_{body} der erste Knoten des Schleifenrumpfs und $e_e = (v_{head}, v_{body})$.

Die Einbindung der Schleife in die Gesamt-WCET des Programms wird mit

$$w_{head} = w'_{head} \cdot loopcount(v_{head}) + w_{ext} \tag{4.1.9}$$

hergestellt. Der Knoten v_{ext}, dessen kumulierte WCET hier verwendet wird ist der Knoten, der unmittelbar auf die Schleife folgt.

Im Beispiel aus Abbildung 4.1.3 ist v_4 der Schleifenkopf und der Knoten v_6 alleiniger Schleifenrumpf. Zunächst wird die WCET einer einzelnen Schleifeniteration w'_4 mit Hilfe der Ungleichung

$$w'_4 \geq w_6 + c(v_4) - \sum_m gain(d_m, v_4) \cdot s^m_4 + \sum_m c_{load}(d_m) \cdot y^m_e + \sum_m c_{store}(d_m) \cdot z^m_e$$

$$\text{mit } e_e = (v_4, v_6)$$

berechnet. Diese Einzel-WCET wird dann mittels

$$w_4 = w'_4 \cdot loopcount(v_4) + w_5$$

in die Berechnung der Gesamt-WCET miteinbezogen.

Funktionsaufrufe

Funktionsaufrufe bedürfen ebenfalls einer gesonderten Behandlung, die ähnlich der Einbindung von Schleifen vonstatten geht. Bei mehrfachen Aufrufen einer Funktion ausgehend von verschiedenen Stellen innerhalb eines Programms kann es dazu kommen, dass Back-Kanten entstehen. Um diese zu umgehen, wird jede einzelne Funktion f analog zum Gesamtprogramm behandelt und bekommt je einen virtuellen Eintritts- und Austrittsknoten v_{f_entry} bzw. v_{f_exit}. Für jede Funktion wird die WCET in w_{f_entry} berechnet.

Aus der Definition eines Basisblocks geht hervor, dass ein Funktionsaufruf grundsätzlich am Ende eines Basisblocks stattfindet. Sei nun v_{caller} der Knoten, an dessen Ende die Unterfunktion aufgerufen wird. Weiterhin sei v_{return} der Knoten, in den der Kontrollfluss nach Abarbeitung der Funktion zurückkehrt. Dann wird die WCET des Aufrufknotens w_{caller} wie folgt berechnet:

$$\begin{aligned} w_{caller} \quad \geq \quad & w_{f\_entry} + w_{return} + c\left(v_{caller}\right) - \sum_i gain\left(d_i, v_{caller}\right) \cdot s^i_{caller} + \\ & \sum_i c_{load}\left(d_i\right) \cdot y^i_e + \sum_i c_{store}\left(d_i\right) \cdot z^i_e \end{aligned} \tag{4.1.10}$$

mit $e_e = \left(v_{caller}, v_{f\_entry}\right)$

Das heißt, dass die WCET einer Funktion nur einmalig berechnet wird und der Wert anschließend an den aufrufenden Knoten mit eingebunden wird. Die Knoten v_{return}, zu denen der Kontrollfluss zurückkehrt bekommen zusätzlich zu den Kopierkosten auf der ausgehenden Kante noch die Kopierkosten auf der Return-Kante $e_{ret} = \left(v_{f\_exit}, v_{return}\right)$ addiert:

$$\begin{aligned} w_{return} \quad \geq \quad & w_{return+1} + c\left(v_{return}\right) - \sum_i gain\left(d_i, v_{caller}\right) \cdot s^i_{caller} + \\ & \sum_i c_{load}\left(d_i\right) \cdot y^i_{out} + \sum_i c_{store}\left(d_i\right) \cdot z^i_{out} + \\ & \sum c_{load}\left(d_i\right) \cdot y^i_{ret} + \sum c_{store}\left(d_i\right) \cdot z^i_{ret} \end{aligned} \tag{4.1.11}$$

4.2 WCC-Einbindung

Für die Einbindung des ILPs in den WCC wurde der schon vorhandene Code von Rotthowe [Rot08] als Coderahmen verwendet. Für den IPCFG wurde die Datenstruktur von Kleinsorge [Kle08] in diesen Code übernommen. Innerhalb des Coderahmens wurde die Generierung des ILPs komplett neu erstellt. Aus dem Ergebnis des ILP-Solvers wird dann basisblockweise eine Belegung des Scratchpad-Speichers nach einem First-Fit-Algorithmus berechnet.

Dabei wird beginnend mit dem Startknoten des Programms für jeden Knoten eine Aufteilung des Scratchpad-Speichers berechnet. Dabei werden neu hinzugekommene Datenobjekte an der ersten ausreichend großen Stelle im SPM beginnend mit der niedrigsten SPM-Adresse platziert.

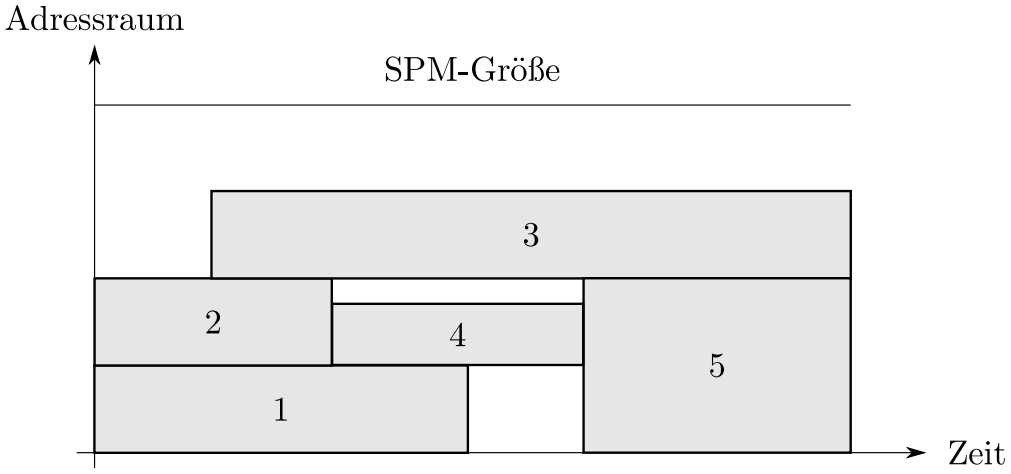


Abbildung 4.2.1: Beispiel zur SPM-Belegung nach dem First-Fit-Algorithmus

Dabei kann es dazu kommen, dass aus Fragmentierungsgründen für ein Datenobjekt im Scratchpad-Speicher kein Platz mehr vorhanden ist. In diesem Fall wird die Variable im betroffenen Basisblock nicht im SPM liegen. Falls im weiteren Verlauf Platz zur Verfügung steht, wird die Allokation nachgeholt.

In Abbildung 4.2.1 wird ein Diagramm für ein einfaches Beispiel für dieses Vorgehen dargestellt. Auf der x-Achse ist die Zeit aufgetragen, auf der y-Achse der Adressraum beginnend mit der SPM-Startadresse. Darüber hinaus ist noch die Größe des SPM als Schranke eingetragen. Dieses Beispiel legt einen linearen Programmablauf zu Grunde. Dabei sollen folgende SPM-bezogene Operationen durchgeführt werden:

- Objekte 1 und 2 sind zum Programmstart im Scratchpad-Speicher und werden an dessen Beginn platziert.
- Objekt 3 wird ins SPM an der erstmöglichen Stelle, d.h. nach Objekt 2 eingelagert.
- Objekt 2 wird in den Hauptspeicher zurückgeschrieben und Objekt 4 an der erstmöglichen Stelle, also unmittelbar oberhalb von Objekt 1 eingelagert.
- Objekt 1 wird in den Hauptspeicher zurückgeschrieben und Objekt 5 soll an der erstmöglichen Stelle eingelagert werden. Obwohl grundsätzlich genug Platz im SPM vorhanden ist, ist dieser in diesem Fall nicht am Stück vorhanden. Deswegen bleiben die Objekte 3 und 4 zunächst allein im SPM.
- Objekt 4 wird in den Hauptspeicher zurückgeschrieben und damit ist nun genug zusammenhängender Platz für Objekt 5 vorhanden, welches in diesen freien Bereich eingeordnet wird.

Sämtliche Datenobjekte bekommen einen festen Platz im Hauptspeicher zugewiesen, auch wenn sie in der Startallokation, also zu Beginn der Programmausführung, im SPM liegen.

4.2.1 Platzierung des Spillcodes

Die Platzierung des Spillcodes ist abhängig von der Struktur des Graphen an den jeweiligen Kanten mit positiven Kopierentscheidungen. Prinzipbedingt müssen die Kopiervorgänge vom SPM in den Hauptspeicher vor den Kopiervorgängen vom Hauptspeicher ins SPM durchgeführt werden, da ansonsten nicht sichergestellt werden kann, dass zu jedem Zeitpunkt ausreichend Platz zur Verfügung steht, alle vorgesehenen Variablen im SPM unterzubringen. Um dieser Tatsache Rechnung zu tragen, wird Code zur Auslagerung von Daten aus dem SPM jeweils an das Ende von Basisblöcken bzw. im Sinne des Graphen auf den Beginn ausgehender Kanten gelegt. Umgekehrt wird der Code zur Einlagerung von Daten in den SPM jeweils zu Beginn der Basisblöcke bzw. auf das Ende eingehender Kanten gelegt.

Für die Ein- und Auslagerungen sind grundsätzlich jeweils zwei verschiedene Möglichkeiten der Graphenstruktur gegeben: Es handelt sich entweder um eine einzelne oder um mehrere Kanten. Die generelle Platzierung für alle Varianten ist in Tabelle 4.2 detailliert erklärt.

Eine Ausnahme von dieser Codeplatzierung bilden Rücksprungkanten aus Unterfunktionen. Hier wird der Spillcode immer am Zielknoten platziert, da innerhalb einer Funktion der Aufrufer zur Laufzeit und damit die Zielbelegung des Scratchpad-Speichers nach dem Rücksprung nicht bekannt ist.

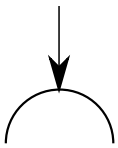
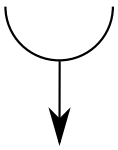
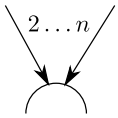
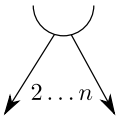
Eingehende Kanten	Ausgehende Kanten
 Nur eine eingehende Kante: Der Spillcode kann zu Anfang des zugehörigen Basisblocks platziert werden.	 Nur eine ausgehende Kante: Der Spillcode kann am Ende des zugehörigen Basisblocks platziert werden. Ist die letzte Instruktion allerdings eine Sprunganweisung, muss der Spillcode direkt davor platziert werden.
 Zwei bis n eingehende Kanten: Zunächst werden die für alle Kanten notwendigen Kopieraktionen ermittelt. Dann wird ausgewertet, welche bei allen Kanten notwendig sind. Diese werden dann zunächst am Anfang des Basisblocks platziert. Für einzelne Kanten gesondert notwendige Kopieroperationen werden wiederum davor platziert mit einem unbedingten Sprung an den Anfang des gemeinsamen Spillcodes (dieser entfällt, falls der gemeinsame Code unmittelbar auf den separaten folgt). Die Sprungziele der letzten Befehle der vorhergehenden Basisblöcke müssen gegebenenfalls auf den Start des für die Kante eingefügten Spillcodes umgebogen werden. Falls eine der Kanten keinen Sprung beinhaltet, muss dieser eventuell eingefügt werden.	 Zwei bis n ausgehende Kanten: Zu behandeln ist hier nur der Fall $n = 2$, da jeder andere Fall nur dann auftreten kann, wenn es sich um das Ende einer Funktion handelt. Dieser Fall wird bereits gesondert behandelt. Mehrere ausgehende Kanten können ansonsten nur bei einer bedingten Sprunganweisung auftreten. Auch hier wird ähnlich wie bei den eingehenden Kanten verfahren. Zunächst werden für beide Kanten die notwendigen Kopieroperationen ermittelt. Aus diesen wird dann die Schnittmenge gebildet, und dieser gemeinsame Teil vor der bedingten Sprunganweisung platziert. Falls für den genommenen Sprung ebenfalls Kopieroperationen notwendig sind, wird für diese ein zusätzlicher Basisblock eingefügt und als Ziel für den bedingten Sprung eingesetzt. Ans Ende des zusätzlichen Basisblocks wird gegebenenfalls noch ein unbedingter Sprung an die ursprüngliche Zieladresse eingefügt. Hinter der bedingten Sprunganweisung wird ggf. der separate Spillcode für diese Kante eingefügt

Tabelle 4.2: Verschiedene Graphenstrukturen und das jeweilige Vorgehen zur Codeplatzierung

5 Reduktion der Kopierkosten

Ein zentrales Problem der dynamischen Allokation von Scratchpad-Speicher, insbesondere für Daten, sind die zwangsläufig anfallenden Kopierkosten. Diese liegen häufig in einem Bereich, der die Einsparpotenziale durch Nutzung des Scratchpad-Speichers zu weiten Teilen zunichte macht. Im Rahmen dieser Arbeit wurde deswegen nach Möglichkeiten gesucht, diese Kosten zu minimieren. Im Vorfeld kristallisierten sich vier verschiedene Ansatzpunkte heraus, die im Verlauf der Arbeit näher untersucht wurden.

Der Kopiercode der dynamischen Allokationen wurde bislang direkt als Inline-Code an den erforderlichen Stellen eingefügt. Untersucht wurde hierbei, ob es sich lohnt, den Kopiercode in eigene Funktionen auszulagern und damit die Laufzeit des Programms zu verringern. Im Rahmen dessen wurde untersucht, ob die Änderung der Breite der Kopierinstruktionen einen Einfluss auf die Laufzeit hat.

Ein grundsätzlich anderer Ansatz war, die Kopieraktionen als parallelisierbar anzusehen und damit die Laufzeit von den Kosten des Kopiercodes zu entlasten. Ein Teilansatz war die Modifikation der Annotationen für den WCET-Analyzer aiT, ein weiterer Teilansatz war die Nutzung der prozessoreigenen DMA-Einheit, um den eigentlichen Kopiervorgang aus der Laufzeit zu entfernen.

5.1 Spezialisierung der Kopierfunktionen

Der ursprüngliche Assemblercode, der von den dynamischen Allokationen generiert wurde, wurde an den von der ILP-Lösung vorgesehenen Stellen direkt eingefügt. Durch die Auslagerung der einzelnen Spill-Code-Snippets in eigene Funktionen können sowohl Laufzeit als auch Codegröße gesenkt werden.

Der ursprüngliche Kopier-Code, beispielhaft in Listing 5.1 dargestellt, sah zunächst das Retten derjenigen Register auf den Stack vor, die im Rahmen der Kopieraktion verwendet wurden. Daraufhin wurde die Schleife vorbereitet, das heißt es wurden sowohl Start- und Zieladressen, als auch die Iterationsgrenzen gesetzt. Darauf folgte eine einfache Zero-Overhead-Schleife, und der Kopiercode wurde vom Zurückschreiben der Register aus dem Stack abgeschlossen.

In der neuen Variante aus Listing 5.2 kann der Code zum Retten und Zurückschreiben der Register entfallen, da der TC1796 beim Aufruf einer Funktion mittels `call` den oberen Kontext automatisch sichert. Umgekehrt wird der obere Kontext beim Verlassen der Funktion mittels `ret` wiederhergestellt [Inf08b].

Listing 5.1 Ursprünglicher Kopiercode

```
_tospm:                          # Retten der genutzten Register
sub.a %a10, 20
st.w [%a10] 16, %d0
st.a [%a10] 12, %a3
st.a [%a10] 8, %a2
st.a [%a10] 4, %a1
movh.a %a1, HI:dec_nbh           # Vorbereitung der Schleife
lea %a1, [%a1] LO:dec_nbh
movh.a %a2, 53248
lea %a2, [%a2] 16548
movh.a %a3, 0
lea %a3, [%a3] 1
_tospm_loop:                     # Kopierschleife
ld.w %d0, [%a1+] 4
st.w [%a2+] 4, %d0
loop %a3, _tospm_loop
_tospm_end:                      # Register zurückschreiben
ld.a %a1, [%a10] 4
ld.a %a2, [%a10] 8
ld.a %a3, [%a10] 12
ld.w %d0, [%a10] 16
lea %a10, [%a10] 20
```

Listing 5.2 Kopiercode in spezialisierter Funktion

```
call _tospm                      # Aufruf der Kopierfunktion
[...]
_tospm:                          # Schleife vorbereiten
movh.a %a12, HI:dec_nbh
lea %a12, [%a12] LO:dec_nbh
movh.a %a13, 53248
lea %a13, [%a13] 16548
movh.a %a14, 0
lea %a14, [%a14] 1
_tospm_loop:                     # Kopierschleife
ld.w %d8, [%a12+] 4
st.w [%a13+] 4, %d8
loop %a14, _tospm_loop
_tospm_end:                      # Rückkehr
ret
```

5.2 Breite der Kopieraktionen modifizieren

Wie in den vorangegangenen Codebeispielen zu sehen ist, werden die Daten mit `ld.w` und `st.w` in 32-Bit-Worten kopiert. Da der TC1796, wie in 3.1 erwähnt, in der Lage ist, auch mit 64 Bit breiten Werten umzugehen, besteht die Möglichkeit, die Anzahl der ausgeführten Kopierinstruktionen näherungsweise zu halbieren.

Hierbei kann es allerdings zu einer Vergrößerung des Codes kommen. Von der Kopierschleife kann dann nur der Teil der zu kopierenden Daten erfasst werden, dessen Größe sich noch restfrei durch 8 Byte teilen lässt. Der verbleibende „Verschnitt“ wird dann mit einzelnen 8, 16 oder 32 Bit breiten Kopierinstruktionen kopiert.

5.3 Annotationen modifizieren

Der vom WCC eingesetzte WCET-Analyzer aiT kann bei seiner Binärcode-Analyse mit Annotations-Dateien unterstützt werden. Diese Dateien enthalten unter anderem Informationen über die Anzahl an Schleifeniterationen, die Ziele von Speicherzugriffen oder auch die Taktgeschwindigkeit des Prozessors [Abs08]. Mit Hilfe dieser Annotationen sollte ein Verhalten dargestellt werden, in dem das Kopieren von Code bzw. von Daten aus dem und in den Scratchpad-Speicher keinen Einfluss auf die Gesamtlaufzeit eines optimierten Programms haben soll. Der Grundgedanke dahinter sah vor, dass die Kopieroperationen als parallel zur Ausführung des übrigen Codes anzusehen seien.

Vielversprechend erschien dabei die Annotation von Code-Stücken, die nicht analysiert werden sollen. Diesen kann dann eine feste Ausführungszeit vorgegeben werden, die aiT dann in der weiteren Analyse verwendet. Als Ausführungszeit wurde in den Annotationen dann entsprechend ein Wert von 0 eingesetzt. Die Analyse von aiT lieferte allerdings höhere WCET-Werte für die getesteten Programme, als ohne eine derartige Annotation. Dies ist darauf zurückzuführen, dass aiT nach einer nicht analysierten Funktion von einem unbekannten Pipeline- und Cache-Zustand ausgeht. Letzteres führt dazu, dass die Speicherzugriffe im weiteren Programmablauf zunächst als Cache-Misses bewertet werden und damit die WCET deutlich ansteigen lassen.

Darüber hinaus unterstützt aiT in den Annotations-Dateien die künstliche Steigerung der Laufzeit einelner Instruktionen. In diesem Fall wird der betroffene Code-Block weiterhin analysiert und in die WCET-Berechnung mit einbezogen. Der Versuch, über diese Möglichkeit eine zusätzliche negative Laufzeit zu annotieren, scheiterte allerdings, da aiT die Annotations-Datei nicht mehr akzeptierte.

Als Fazit lässt sich damit ziehen, dass die Modifikation der Annotationen nicht geeignet ist, um die Kopierkosten, wenn auch nur virtuell, zu reduzieren.

5.4 Nutzung der DMA-Einheit

Ein weiteres Konzept zur Reduktion der Kopierkosten lag in der Nutzung der DMA-Einheit des TC1796, um die notwendigen Kopien durchzuführen. Die DMA-Einheit unterstützt das nebenläufige Kopieren von Daten, ohne dass das laufende Programm nach

Start der Transaktion aktiv eingreifen muss. Die DMA-Einheit muss vor Beginn mit den notwendigen Informationen des Kopiervorgangs versorgt werden und führt diesen dann eigenständig durch. Sowohl nach Abschluss dieses Vorgangs als auch nach Abschluss von Teiloperationen kann eine Interrupt-Behandlung gestartet werden, es kann allerdings auch durch eine manuelle Abfrage festgestellt werden, ob eine Transaktion bereits abgeschlossen ist.

Für den Fall der WCET-Analyse mit aiT ist die Interrupt-Variante nicht analysierbar, da aiT keine Modellierung der DMA des TC1796 mit sich bringt. Die manuelle Abfrage, die schlussendlich auf eine Busy-Waiting-Schleife hinausläuft, kann aus dem selben Grund nicht angewendet werden. Damit muss auf anderem Wege sichergestellt werden, dass ein DMA-gestützter Datentransfer in einem ausreichend großen Zeitrahmen durchlaufen kann. Dies wird im Abschnitt 5.4.2 im Detail erläutert. Zunächst wird allerdings erklärt, wie die DMA-Einheit genau gesteuert wird.

5.4.1 Ansteuerung der DMA-Einheit

Die DMA-Einheit des TC1796 wird durch insgesamt 135 Register gesteuert, denen jeweils eine Adresse zwischen `0xf0003c00` und `0xf0003eff` zugeordnet ist [Inf07]. Im Folgenden werden die für die geplanten Kopieraktionen notwendigen Register vorgestellt, beginnend mit denjenigen, die für jeden der 16 DMA-Kanäle einzeln verfügbar sind. Doch zunächst eine Begriffsdefinition

Definition. Eine *DMA transaction*, die ein Datenobjekt kopieren soll, setzt sich zusammen aus *DMA moves*, die 8, 16 oder 32 Bit breit sind. Bis zu 16 *DMA moves* bilden einen *DMA transfer*, von denen bis zu 511 eine *DMA transaction* ausmachen.

- Das *Channel Control Register* (DMA_CHCRmx) teilt sich in mehrere Bitfelder auf. Für alle nun vorgestellten Register mit Bitfeldern gilt, dass nicht erläuterte Bits nicht verwendet werden und konstant 0 sind:

Feld	Bits	Beschreibung
TREL	[8 : 0]	**Transfer Reload Value**: In diesem Bitfeld wird die Anzahl der Transfers je Transaktion gespeichert.
PRSEL	[15 : 13]	**Peripheral Request Select**: Steuert Hardware-Anfragen, hier konstant 000_2
BLKM	[18 : 16]	**Block Mode**: 2^{value} =Anzahl der Moves je Transfer, $000_2 \leq value \leq 100_2$
RROAT	19	**Reset Request Only After Transaction**: Gesetzt, falls eine Transaktion am Stück laufen soll, hier konstant 1_2
CHMODE	20	**Channel Operation Mode**: Steuert Hardware-Anfragen, hier konstant 0_2
CHDW	[22 : 21]	**Channel Data Width**: Breite eines Moves – $00_2\hat{=}$8 Bit; $01_2\hat{=}$ 16 Bit; $10_2\hat{=}$ 32 Bit
PATSEL	[25 : 24]	**Pattern Select**: *Pattern detection* wird hier nicht verwendet, hier konstant 00_2
CHPRIO	28	**Channel Priority**: Da keine Priorisierung vorgesehen ist, hier konstant 1_2
DMAPRIO	30	**DMA Priority**: Regelt die Prioritätsebene auf dem FPI-Bus; um Zeitschranken einzuhalten, hier konstant 1_2

- Das *Channel Interrupt Control Register* (DMA_CICRmx) teilt sich ebenfalls in mehrere Bitfelder auf; da hier nicht zirkulär kopiert oder mit Interrupts gearbeitet wird, ist der Registerinhalt stets $0000\,0000_{16}$.

Feld	Bits	Beschreibung
WRPSE	0	**Wrap Source Enable**: Steuert, ob die Quelle zirkulär gelesen werden soll, hier konstant 0_2
WRPDE	1	**Wrap Destination Enable**: Steuert, ab das Ziel zirkulär geschrieben werden soll, hier konstant 0_2
INTCT	[3 : 2]	**Interrupt Control**: Steuert die Generierung von Interrupts, hier konstant 00_2
WRPP	[7 : 4]	**Wrap Pointer**: Speichert Interruptbehandlungsroutinen, hier nicht verwendet, konstant 0000_2
INTP	[11 : 8]	**Interrupt Pointer**: Speichert Interruptbehandlungsroutinen, hier nicht verwendet, konstant 0000_2
IRDV	[15 : 12]	**Interrupt Raise Detect Value**: Kennzeichnet Interruptauslöser, hier nicht verwendet, konstant 0000_2

- Das *Address Control Register* (DMA_ADRCRmx) enthält Informationen, in welchen Schritten und in welcher Richtung die Speicherbereiche durchlaufen werden sollen:

Feld	Bits	Beschreibung
SMF	[2 : 0]	**Source Address Modification Factor**: Enthält die Schrittgröße beim Traversieren, hier konstant 000_2
INCS	3	**Increment of Source Address**: Kennzeichnet, ob aufsteigend traversiert wird, hier konstant 1_2
DMF	[6 : 4]	**Destination Address Modification Factor**: analog zu SMF, hier konstant 000_2
INCD	7	**Increment of Destination Address**: analog zu INCS, hier konstant 1_2
CBLS	[11 : 8]	**Circular Buffer Length Source**: Kennzeichnet die Größe des Ringpuffers (falls verwendet), hier konstant 1111_2
CBLD	[15 : 12]	**Circular Buffer Length Destination**: Analog zu CBLS, hier konstant 1111_2
SHCT	[17 : 16]	**Shadow Control**: Zeigt an, ob und wie das Schattenadressregister verwendet wird, hier konstant 00_2

- Das *Source Address Register* (DMA_SADRmx) enthält die 32 Bit breite Startadresse der DMA-Transaktion, das *Destination Address Register* (DMA_DADRmx) enthält die 32 Bit breite Zieladresse der DMA-Transaktion.

Schließlich gibt es noch drei allgemeine DMA-Register die für eine einfache Kopieroperation von Belang sind:

- Das *Interrupt Clear Register* (DMA_INTCR) ist ein „write-only"-Register, das heißt, dass hier hineingeschriebene Werte nicht gespeichert, sondern nur direkt verarbeitet werden. Hier können eventuell gesetzte Interrupt-Flags gelöscht werden. Da das Modell ohne Interrupts arbeitet, werden alle etwaigen Bits gelöscht und das Register auf $FFFFFFFF_{16}$ gesetzt.

- Das *Software Transaction Request Register* (DMA_STREQ) ist ebenfalls ein „write-only"-Register. Erst wenn hier das Bit für den jeweiligen Kanal gesetzt wird, wird die DMA-Transaktion die über die o.g. Register definiert wird, gestartet. Die Kanäle sind anhand ihrer Nummer sortiert. Das heißt, in Bit 0 wird Kanal 0 gesteuert und in Bit 15 wird Kanal 15 gesteuert. Die Bits 16–31 werden nicht verwendet und sind konstant 0.

Der komplette Ablauf einer einzelnen beispielhaften DMA-Transaktion im Kanal 0 wäre hier folgender:

```
DMA_INTCR = 0xffffffff;
DMA_CHCR00 = 0x50080000 | codierter Transaktionsumfang;
DMA_CHICR00 = 0x00000000;
DMA_ADRCR00 = 0x0000ff88;
DMA_SADR00 = Quelladresse;
DMA_DADR00 = Zieladresse;
DMA_STREQ = 0x00000001; // Startet Transaktion
```

Die Laufzeit des Assembler-Codes, der einen solchen Auftrag initiiert, konnte mit aiT analysiert werden und lieferte bei jedem Auftreten einen Wert von weniger als 50 Zyklen.

5.4.2 Dauer eines Kopiervorgangs

Da in der TriCore-Modellierung von aiT die DMA nicht mit modelliert ist, darf der zu analysierende Quellcode keine Überprüfungen enthalten, ob eine DMA-Transaktion bereits fertiggestellt wurde. Eine solche Busy-Waiting-Schleife könnte von aiT als Endlosschleife aufgefasst werden und die Berechnung einer WCET-Schranke verhindern. Daher ist über andere Mechanismen sicherzustellen, dass bei Verwendung einer mittels der DMA-Einheit kopierten Variable bzw. eines Basisblocks die zugehörige DMA-Transaktion abgeschlossen wurde. Hierzu ist es notwendig, die Dauer echter DMA-Transaktionen zu messen und anhand der Messwerte eine geeignete Schranke zu finden.

Evaluationsaufbau

Zur Messung der Dauer eines Kopiervorgangs wurde das TriCore-Evaluationsboard *TriBoard TC1796 V4.1* verwendet. Um ein Testprogramm auf dem TC1796 auszuführen, wird ein Betriebssystem benötigt, welches mindestens eine rudimentäre Laufzeitumgebung zur Verfügung stellt. Die Wahl fiel hier auf *Erika Enterprise*, ein Minimal-Echtzeit-Kernel für Single- und Multi-Core-Umgebungen [Evi09]. Die Portierung auf TriCore-Umgebungen wurde bereits im Vorfeld von Kleinsorge durchgeführt [Kle10].

Durch prozessorspezifische Header-Dateien besteht die Möglichkeit, einzelne Registerinhalte über C-Code zu steuern. Neben dem Zugriff auf die DMA-Register besteht auch die Möglichkeit über ein Timing-Register die Anzahl der vergangenen Zyklen zu bestimmen. Um möglichst flexibel verschiedene Konfigurationen zu testen, wurde eine Funktion implementiert, die für die gegebenen Kombinationen an DMA-Aktionen den notwendigen Code generiert. In diesen Code ist die Zeitmessung und die Ausgabe der Kopierdauer mit integriert. Der Code ist in 5.3 dargestellt.

Der Code wurde per Remote-Debugger auf dem Evaluationsboard ausgeführt und die gemessenen Werte zur weiteren Ausführung als CSV-Datei abgespeichert.

Moves	Transfers	Dauer
16	1	42
128	8	270
256	16	529
384	24	788
512	32	1050
640	40	1309
768	48	1569
896	56	1831
1024	64	2090
1152	72	2349
1280	80	2608
1536	96	3130
1792	112	3651
2048	128	4169
2304	144	4691
2560	160	5209
2816	176	5730
3072	192	6248
3328	208	6770
3584	224	7291
3840	240	7809
4096	256	8330
4352	272	8848
5120	320	10410
5632	352	11449
6144	384	12489
6656	416	13528
7168	448	14571
7680	480	15611
8176	511	16619

Tabelle 5.1: Exemplarische Messwerte der DMA-Analyse

Listing 5.3 Generische Funktion zur Messung der Dauer einer DMA-Transaktion

```
void call_dma(int from, to, chWidth, movesPerTransfer, transfers)
{
  // Bildung der Bitfelder im Channel Control Register
  chcr = transfers | chcr;
  switch ( movesPerTransfer ) {
    case 16: chcr = ( 4 << 16 ) | chcr; break;
    case 8: chcr = ( 3 << 16 ) | chcr; break;
    case 4: chcr = ( 2 << 16 ) | chcr; break;
    case 2: chcr = ( 1 << 16 ) | chcr; break;
  }
  switch ( chWidth ) {
    case 32: chcr = ( 2 << 21 ) | chcr; break;
    case 16: chcr = ( 1 << 21 ) | chcr; break;
  }
  chcr = 0x50080000 | chcr;

  // Setzen der Register
  DMA_INTCR.reg = 0xffffffff;
  DMA_HTREQ.reg = 0xffff0000;
  DMA_CHCR00.reg = chcr;
  DMA_CHICR00.reg = 0x00000008;
  DMA_ADRCR00.reg = 0x0000ff88;
  DMA_SADR00.reg = from;
  DMA_DADR00.reg = to;

  // Aktuellen Zeitpunkt sichern
  start = STM_TIM0.reg;

  // Transaktion starten
  DMA_STREQ.reg = 0x00000001;

  // Busy waiting, bis Transaktion abgeschlossen
  while( DMA_INTSR.reg == 0x00000000 );

  // Aktuellen Zeitpunkt sichern und Dauer berechnen
  stop = STM_TIM0.reg;
  result = stop - start;

  // Ausgabe
  printf("Duration:␣%d\n", stop-start);
}
```

Messwerte und Bildung einer oberen Schranke

Für eine hinreichend große Datenbasis wurde der Code mit verschiedenen Einstellungen bezüglich Quelladresse, Zieladresse, Kanalbreite etc. ausgeführt. Als Start bzw. Ziel wurden jeweils Adressen im Hauptspeicher bzw. im Scratchpad-Speicher gewählt. Die Messwerte zeigten, dass die Dauer eines Kopiervorgangs sowohl von der Anzahl der durchgeführten DMA-Moves, als auch von der Anzahl der DMA-Transfers abhängig ist. Die Kanalbreite selbst hat offensichtlich keinen Einfluss auf die Dauer eines Kopiervorgangs.

Ebenso keinen Einfluss auf die Kopierdauer hat die Anordnung von Quelle und Ziel einer Transaktion. Das heißt, dass der Kopiervorgang eines bestimmten Datums vom Haupt- in den Scratchpad-Speicher genau so viel Zeit benötigt, wie der Kopiervorgang in umgekehrter Richtung. Wie aus Tabelle 5.1 ersichtlich ist, kann als Grundlage von folgendem Zusammenhang ausgegangen werden:

$$\begin{aligned} 1 \text{ Move} &\approx 2 \text{ Zyklen} \\ 1 \text{ Transfer} &\approx 1 \text{ Zyklus} \\ \rightsquigarrow 2 \cdot \#(Moves) + \#(Transfers) &\approx \#(Zyklen) \end{aligned}$$

Bei einer niedrigen Anzahl von Transfers bzw. Zyklen kann es bei dieser Vorschrift dazu kommen, dass die berechnete Anzahl an Zyklen unter der gemessenen liegt. Für die exemplarisch dargestellten Messwerte würde ein Aufschlag von 10 Zyklen reichen. Dabei wäre der Puffer zwischen berechnetem und gemessenem Wert im schlimmsten Fall bei lediglich einem Zyklus. Ein einzelner Zyklus kann durch unerwartete Seiteneffekte schnell aufgebraucht werden. Um bei der Einbindung in die ILP-Modelle der DMA-gestützten Scratchpad-Allokation mit einem größeren Sicherheitspuffer zu rechnen, wird der Puffer um 15 Zyklen vergrößert. Damit wird die Dauer eines DMA-Transfers wie folgt abgeschätzt:

$$dur = 2 \cdot moves_{count} + transfers_{count} + 25 \tag{5.4.1}$$

Parallele Nutzung mehrere DMA-Kanäle

Die DMA-Einheit ermöglicht die Nutzung von 16 Kanälen, in denen jeweils eine Kopieraktion durchgeführt werden kann. Da sämtliche Kanäle über den selben Bus auf die verschiedenen Speicherbereiche zugreifen, können diese nur sequentiell abgearbeitet werden. Die Evaluation hat gezeigt, dass keine Seiteneffekte durch die konsekutive Durchführung mehrerer DMA-Transaktionen mehrerer DMA-Kanäle ergeben. Deswegen kann für diesen Fall die Addition der einzelnen Werte für die Kopierdauer als sicher erachtet werden.

6 Dynamische Allokationen mit DMA-Nutzung

In diesem Kapitel werden die ILP-Modelle zur DMA-gestützten Allokation des Scratchpad-Speichers vorgestellt. Als Grundlage dient erneut das ILP-Modell von Kleinsorge [Kle08], welches allerdings einige Erweiterungen erhält. Diese sind notwendig geworden, da der Kopiervorgang der DMA-Einheit als nebenläufiger Prozess mitmodelliert werden muss. Zusätzlich muss die Konsistenz beider Speicherbereiche (Hauptspeicher und Scratchpad-Speicher) auch während der Kopierprozesse sichergestellt werden.

Die in diesem Kapitel beschriebenen ILP-Modelle basieren wie das in Kapitel 4 vorgestellte ILP-Modell für die dynamische Datenallokation auf einem interprozeduralen Kontrollflussgraphen $G = (V, E)$. Analog gelten die in Abschnitt 4.1 aufgeführten Definitionen, um Schleifen sicher zu erkennen.

6.1 Datenallokation

Um die Komplexität des ILP-Modells für die DMA-gestützte Scratchpad-Allokation von Daten einzuschränken, wurden einige vereinfachende Annahmen getroffen:

- Parallel stattfindende DMA-Transaktionen werden immer zum selben Zeitpunkt gestartet und gelten zu einem gemeinsamen Zeitpunkt als beendet, da über die Reihenfolge der Kopieroperationen innerhalb der DMA-Einheit keine Aussage getroffen werden kann. Im weiteren Verlauf wird eine solche Gruppierung von Transaktionen als DMA-Block bezeichnet.
- Ein DMA-Block enthält entweder nur Kopiervorgänge in den Scratchpad-Speicher oder nur in den Hauptspeicher. Es ist zwar grundsätzlich möglich, die Kopien aus dem SPM in den Hauptspeicher über die DMA-eigenen Prioritätsstufen in zwei Klassen zu unterteilen. Allerdings kann dieses Verhalten auch durch die konsekutive Anordnung der DMA-Blöcke näherungsweise erreicht werden. Das ILP hingegen kann damit vereinfacht werden, da insbesondere die Berechnung der Dauer eines DMA-Blocks nur noch von einem Start- bzw. Zielzeitpunkt abhängt.
- Die Kopie einer Variable wird in nur einem Kanal durchgeführt.

6.1.1 ILP-Modell

Analog zum ILP-Modell aus Abschnitt 4.1, werden auch hier Variablen und Konstanten über die Notation differenziert. Variablen sind an einer Index-Notation (Beispiel: a_b^c) zu erkennen, während Konstanten als Funktion notiert werden (Beispiel: $a\,(b, c)$).

Konstanten

Die Konstanten sind zu weiten Teilen gleich mit denen des ILP-Modells der Datenallokation ohne DMA-Unterstützung. Auch in diesem Modell die Größe eines Datenobjekts $d_i \in data\_objects$ und die des Scratchpad-Speichers selbst enthalten:

Definition 6.1. Sei $size\,(d_i)$ die Größe der Variable d_i in Byte.

Definition 6.2. Sei $size\,(SPM)$ die Größe des Daten-SPM in Byte.

Auch hier wird die Laufzeit der einzelnen Basisblöcke ohne diese Optimierung benötigt, da von ihnen bei der Optimierung des ILPs der Laufzeitgewinn abgezogen werden muss:

Definition 6.3. Sei $c\,(v_k)$ die WCET des durch v_k repräsentierten Basisblocks ohne Variablen im Scratchpad-Speicher.

Der Aufruf einer DMA-Transaktion verursacht unabhängig von der Größe der Transaktion eine konstante zusätzliche Laufzeit. Diese ließ sich mit Hilfe von aiT auf 50 Zyklen nach oben abschätzen. Dieser Wert wird in einer Konstante festgehalten, die bei einer eventuellen Änderung dieser Schranke angepasst werden kann.

Definition 6.4. Sei $c\,(DMA) = 50$.

Der Nutzen der Einlagerung eines Datenobjets in den Scratchpad-Speicher wird wieder über die Konstante $gain\,(d_i, v_k)$ zur bereitgestellt. Die erzielbaren Werte werden auch hier im Voraus berechnet, da sie unabhängig von der optimalen Lösung des ILP sind:

Definition 6.5. Sei $gain\,(d_i, v_k)$ der erzielte Laufzeitgewinn bei einer Einlagerung von d_i in den Scratchpad-Speicher während der Ausführung von v_k.

Auch in diesem Modell werden Schleifen gesondert betrachtet. Deswegen muss zu jeder Schleife auch wieder die maximale Iterationsanzahl zur Verfügung stehen.

Definition 6.6. Falls v_k ein Schleifenkopf ist, dann sei $loopcount\,(v_k)$ die maximale Anzahl an Schleifeniterationen. Sonst sei $loopcount\,(v_k) = 1$.

Da die Dauer einer DMA-gestützten Kopieroperation unabhängig von Quelle und Ziel der Kopie ist, muss für jedes Datenobjekt nur noch eine Konstante eingeführt werden.

Definition 6.7. Sei $dur\,(d_i)$ die Dauer einer Kopieroperation von d_i, die unter Ausnutzung der DMA-Einheit durchgeführt wird.

Die Werte für $dur\,(d_i)$ werden im Vorfeld mit Hilfe der Formel 5.4.1 berechnet. Die

Anzahl der Moves und die Anzahl der Transfers werden folgendermaßen berechnet:

$$
\begin{aligned}
size\,(d_i) \mod 2 \neq 0 &\Rightarrow move_{count} = size\,(d_i) \wedge transfer_{count} = size\,(d_i) \\
size\,(d_i) \mod 4 \neq 0 &\Rightarrow move_{count} = \frac{size\,(d_i)}{2} \wedge transfer_{count} = \frac{size\,(d_i)}{2} \\
size\,(d_i) \mod 8 \neq 0 &\Rightarrow move_{count} = \frac{size\,(d_i)}{4} \wedge transfer_{count} = \frac{size\,(d_i)}{4} \\
size\,(d_i) \mod 16 \neq 0 &\Rightarrow move_{count} = \frac{size\,(d_i)}{4} \wedge transfer_{count} = \frac{size\,(d_i)}{8} \\
size\,(d_i) \mod 32 \neq 0 &\Rightarrow move_{count} = \frac{size\,(d_i)}{4} \wedge transfer_{count} = \frac{size\,(d_i)}{16} \\
size\,(d_i) \mod 64 \neq 0 &\Rightarrow move_{count} = \frac{size\,(d_i)}{4} \wedge transfer_{count} = \frac{size\,(d_i)}{32} \\
\text{sonst} &\Rightarrow move_{count} = \frac{size\,(d_i)}{4} \wedge transfer_{count} = \frac{size\,(d_i)}{64}
\end{aligned}
$$

Hierbei ist nur anhand der ersten zutreffenden Bedingung zu entscheiden. Die Zusammensetzung dieser Werte ergibt sich aus der Kanalbreite von 8, 16 oder 32 Bit sowie der Anzahl Moves pro Transfer. Es wird jeweils versucht diese Werte zu maximieren, wobei der Kanalbreite dabei eine höhere Priorität zufällt.

Als Hilfsvariable bei der Bildung einer oberen Schranke für ein Reihe von Variablen wird noch die Summe der Dauer aller Kopiervorgänge benötigt:

Definition 6.8. Sei $dur\,(all) = \sum_i dur\,(d_i)$.

Um die Konsistenz der Variableninhalte zu sichern, darf während eines DMA-Transfers einer Variable kein lesender oder schreibender Zugriff auf die Variable durchgeführt werden, da zur Übersetzungszeit nicht bekannt ist, ob eine bestimmte Variable bereits kopiert wurde oder nicht. Dafür muss der Variablenzugriff für das ILP als Konstante mitdefiniert werden:

Definition 6.9. Sei $usage\,(d_i, v_k) = 1$, falls im Knoten v_k auf die Variable d_i zugegriffen wird. Sonst sei $usage\,(d_i, v_k) = 0$.

Die Zugriffsinformationen werden im Vorfeld der LLIR entnommen. Abschließend sei nun noch die Anzahl der nutzbaren DMA-Kanäle festgelegt. Diese ist durch den Prozessor auf einen Höchstwert von 16 begrenzt. Bei einer parallelen DMA-gestützten Allokation von Code und Daten müssen die Kanäle getrennt voneinander verwaltet werden, da die ILPs getrennt voneinander arbeiten.

Definition 6.10. Sei dma_count die Anzahl der verfügbaren DMA-Kanäle.

Variablen

Bei den Entscheidungsvariablen werden zwei Variablen aus dem ILP-Modell ohne DMA-Unterstützung verwendet. Die erste ist die zentrale Entscheidungsvariable für die Berechnung des Laufzeitgewinns:

$$s_k^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ im Knoten } v_k \text{ im SPM liegt} \\ 0, & \text{sonst} \end{cases}$$

Um die Speicherinhalte konsistent zu halten, wird das ILP-Modell um den aktuellen Zustand jeder Variable an jeder Kante erweitert:

$$x_j^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ auf der Kante } e_j \text{ im SPM liegt} \\ 0, & \text{sonst} \end{cases}$$

$$t_j^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ auf der Kante } e_j \text{ per DMA} \\ & \text{in den SPM kopiert wird} \\ 0, & \text{sonst} \end{cases}$$

$$f_j^i = \begin{cases} 1, & \text{falls die Variable } d_i \text{ auf der Kante } e_j \text{ per DMA} \\ & \text{aus dem SPM kopiert wird} \\ 0, & \text{sonst} \end{cases}$$

Um die Nebenläufigkeit der DMA-Transaktionen im ILP-Modell mitberücksichtigen zu können, müssen deren Start und Ende ebenfalls gekennzeichnet werden:

$$tStart_k^i = \begin{cases} 1, & \text{falls zu Beginn des Knotens } v_k \text{ ein DMA-Transfer} \\ & \text{in den SPM für die Variable } d_i \text{ beginnt} \\ 0, & \textit{sonst} \end{cases}$$

$$tEnd_k^i = \begin{cases} 1, & \text{falls zu Beginn des Knotens } v_k \text{ ein DMA-Transfer} \\ & \text{in den SPM für die Variable } d_i \text{ endet} \\ 0, & \textit{sonst} \end{cases}$$

$$fStart_k^i = \begin{cases} 1, & \text{falls am Ende des Knotens } v_k \text{ ein DMA-Transfer} \\ & \text{aus dem SPM für die Variable } d_i \text{ beginnt} \\ 0, & \textit{sonst} \end{cases}$$

Die Start-Variablen markieren später den Ort im Code, an welchem der Kopiercode hinterlegt wird. Die Variablen $tEnd_m^i$ und $fStart_m^i$ werden für die Konsistenz des SPM-Inhalts benötigt. Eine Variable $fEnd_m^i$ wird nicht benötigt.

Allerdings werden noch eine Reihe von Indikatorvariablen benötigt, die in diesem Zusammenhang noch bestimmte Ereignisse und Zustände kennzeichnen sollen:

$$tInd_j = \begin{cases} 1, & \text{falls auf der Kante } e_j \text{ mind. ein DMA-Kanal} \\ & \text{in den SPM belegt ist} \\ 0, & \text{sonst} \end{cases}$$

$$fInd_j = \begin{cases} 1, & \text{falls auf der Kante } e_j \text{ mind. ein DMA-Kanal} \\ & \text{aus dem SPM belegt ist} \\ 0, & \text{sonst} \end{cases}$$

$$tEndInd_k = \begin{cases} 1, & \text{falls am Knoten } v_k \text{ mind. ein DMA-Transfer} \\ & \text{in den SPM endet} \\ 0, & \text{sonst} \end{cases}$$

$$fStartInd_k = \begin{cases} 1, & \text{falls am Knoten } v_k \text{ mind. ein DMA-Transfer} \\ & \text{aus dem SPM beginnt} \\ 0, & \text{sonst} \end{cases}$$

Um die Dauer eines nebenläufigen DMA-Transfers im ILP modellieren zu können, müssen dafür ebenfalls Variablen deklariert werden.

$rTo_j^i \;\hat{=}\;$ Abgelaufene Zeit eines DMA-Transfers der Variable d_i an der Kante e_j in den SPM

$rFrom_j^i \;\hat{=}\;$ Verbleibende Zeit eines DMA-Transfers der Variable d_i an der Kante e_j aus dem SPM

Damit die Dauer eines solchen Transfers zuverlässig festgestellt werden kann, wird eine weitere Variable eingeführt, die die Laufzeit eines Basisblocks dynamisch darstellen zu können-

$cDyn_k \;\hat{=}\;$ Laufzeit des Knotens v_k in Abhängigkeit von der SPM-Allokation

Um die WCET eines Programms mit diesem ILP-Modell optimieren zu können, muss diese ebenfalls in Form einer Variable vorliegen. Dazu dient die Flussvariable w_i:

$w_k \;\hat{=}\;$ Laufzeit des WCEP vom Knoten v_k zur Senke des Graphen

Constraints

Die in diesem Modell verwendeten Constraints lassen sich in vier Gruppen unterteilen:

1. Constraints zur Einhaltung der Systemeigenschaften
2. Constraints zur Sicherstellung der Speicher- und Datenkonsistenz
3. Constraints zur Modellierung der Dauer eines DMA-Kopiervorgangs
4. Constraints, die Hilfsvariablen für die ersten beiden Gruppen definieren

Einhaltung der Systemeigenschaften Die Constraints zur Einhaltung der Systemeigenschaften sind ähnlich zu denen aus dem ILP-Modell ohne DMA-Unterstützung. So wird auch hier ein Constraint zur Einhaltung der Größe des Scratchpad-Speichers benötigt:

$$\forall v_k \in V : \sum_i s_k^i \cdot size\,(d_i) \quad \leq \quad size\,(SPM) \tag{6.1.1}$$

Als weitere Systemeigenschaft darf die Anzahl der verfügbaren DMA-Kanäle nicht überschritten werden:

$$\forall e_j \in E : \sum_i t_j^i + f_j^i \quad \leq \quad dma\_count \tag{6.1.2}$$

Sicherstellung der Konsistenz Die folgenden Constraints stellen die Konsistenz des Scratchpad-Speichers und der Daten sicher. Die Speicherkonsistenz wird auch hier ähnlich zum bisherigen ILP-Modell festgelegt. Ein Datum kann in einem Knoten nur dann im Scratchpad-Speicher liegen, wenn es auf der eingehenden Kante im SPM liegt oder ein DMA-Transfer am Anfang des Knotens endet. Eine Analoge Vorschrift gilt für die ausgehenden Kanten:

$$\forall d_i \forall v_k \in V \setminus \{v_{entry}\}\, \forall e_j \in In\,(v_k) : \; s_k^i \quad = \quad tEnd_k^i + x_j^i \tag{6.1.3}$$

$$\forall d_i \forall v_k \in V \setminus \{v_{exit}\}\, \forall e_j \in Out\,(v_k) : \; s_k^i \quad = \quad fStart_k^i + x_j^i \tag{6.1.4}$$

Die Vorschrift, dass ein Datenobjekt zu einem bestimmten Zeitpunkt nur einen Zustand haben darf (analog zu Formel 4.1.2) ist hier nicht explizit notwendig, da sie sich aus den zuletzt genannten Formeln und der Einschränkung, dass eine Gruppe von DMA-Transaktionen entweder nur in den SPM verläuft oder umgekehrt, implizit ergibt:

$$\forall e_j \in E : \; tInd_j + fInd_j \leq 1 \tag{6.1.5}$$

Damit wäre die Konsistenz des Scratchpad-Speichers im Modell fixiert. Als nächstes wird verhindert, dass auf Daten, die zur Ausführungszeit eines Knotens transferiert werden,

zugegriffen wird. Dazu werden für jede Kombination von $v_k \in V \setminus \{v_{entry}, v_{exit}\}$ und $d_i \in data\_objects$ mit $usage\,(d_i, v_k) = 1$ folgende Constraints eingetragen:

$$\forall e_{in} \in In\,(v_k)\, \forall e_{out} \in Out\,(v_k) :$$

$$t^i_{in} + t^i_{out} \leq 1 \tag{6.1.6}$$
$$f^i_{in} + f^i_{out} \leq 1 \tag{6.1.7}$$

Mit diesen Constraints ist dann sowohl die Konsistenz des Scratchpad-Speichers als auch die Konsistenz der Daten über die gesamte Laufzeit hinweg gesichert.

Dauer der Kopiervorgänge Die nächste Constraint-Gruppe modelliert die Dauer der Kopiervorgänge. Hierbei war zu beachten, dass die Dauer eines Kopiervorgangs der kumulierten Dauer aller innerhalb des betreffenden DMA-Blocks kopierten Objekte entspricht. Damit wird dir Vereinfachung Rechnung getragen, dass eine Gruppe von Kopieroperationen einen gemeinsamen Beginn und Schluss haben. Für alle nun folgenden Constraints dieser Gruppe sei nun $d_i \in data\_objects$, $e_j \in E$, $v_k \in V \setminus \{v_{entry}, v_{exit}\}$, $e_{in} \in In\,(v_k)$ sowie $e_{out} \in Out\,(v_k)$. Alle Ungleichungen werden für alle möglichen Kombinationen dieser Variablenmengen erstellt. Zunächst werden die Ungleichungen zur Bildung des abgelaufenen Zeitraums einer Kopie in den Scratchpad-Speicher gebildet. Die Werte der Variablen rTo^i_j bilden sich über eine Randbedingung (Ende des DMA-Blocks) und einen Flow-Constraint. Darüber hinaus sind noch zwei Nebenbedingungen einzuhalten. Während eine Kopie stattfindet, muss der Wert größer Null sein. Findet keine Kopie statt, muss der Wert genau Null sein:

$$rTo^i_j \geq t^i_j \tag{6.1.8}$$
$$rTo^i_j \leq dur\,(all) \cdot t^i_j \tag{6.1.9}$$

Mit dem Flow-Constraint wird der Wert der Variable um die Laufzeit des betreffenden Blocks modifiziert:

$$rTo^i_{in} \geq rTo^i_{out} - t^i_{out} \cdot cDyn_k \tag{6.1.10}$$

Die Randbedingung am Ende eines Transfers setzt den Wert auf die kumulierte Dauer aller hier endenden Kopieraktionen:

$$rTo^i_{in} \geq tEnd^i_k \cdot \left(\sum_n tEnd^n_k \cdot dur\,(d_n) \right) \tag{6.1.11}$$

Die letzten beiden Ungleichungen entsprechen allerdings nicht der Definition eines ILP, da eine Variable mit einem nicht-statischen Wert multipliziert wird. Eine solche Konstruktion kann allerdings durch vier weitere gültige Constraints ersetzt werden [Bis10]. Der Aufbau dieser Constraints wird in Abschnitt 6.3.1 erläutert.

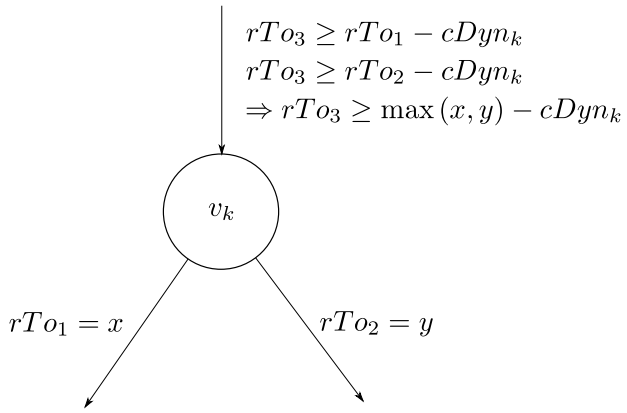


Abbildung 6.1.1: Bei Verzweigungen wird für rTo der „kürzere“ Weg bevorzugt. Dieser ist durch einen höheren Wert von rTo an der ausgehenden Kante erkennbar.

Das Zusammenspiel dieser Ungleichungen soll hier nun für die verschiedenen auftretenden Knotenkonfigurationen erläutert werden. Die Constraints sollen die notwendige Mindestdauer, während der die Entscheidungsvariable t_j^i bei einer laufenden DMA-Transaktion den Wert eins annehmen muss, sicherstellen. Die Modellierung der Variablenmenge rTo_j^i sieht vor, dass deren Werte während einer DMA-Operation in Kontrollflussrichtung monoton steigen, bis das Ende der Transaktion bei $tEnd_k^i = 1$ erreicht ist. Außerhalb einer DMA-Transaktion sollen die Werte konstant bei null stehen. Dies wird durch die Ungleichung 6.1.8 gesichert. Die Ungleichung 6.1.9 stellt sicher, dass innerhalb einer DMA-Transaktion die Werte nie null erreichen dürfen.

Die Bildung der Werte von rTo_j^i verläuft mittels Flow-Constraints ähnlich der Bildung der kumulierten WCET-Werte entgegen der Kontrollflussrichtung. Unterschiedlich ist, dass die Kopierdauer auf Kantenvariablen gespeichert wird, und dass die Kosten der Basisblöcke abgezogen und nicht addiert werden. Dadurch wird bei Verzweigungen der „kürzere“ Weg bevorzugt. Dies ist notwendig, um auch bei Durchläufen des Programms im Best Case die Kopieroperation im Rahmen der berechneten Zeit abzuschließen. Der kürzere Weg zeichnet sich durch den höheren Wert von rTo an der ausgehenden Kante aus, da dort bis dahin weniger Zeit abgezogen werden konnte. Dieses Verhalten wird in Abbildung 6.1.1 deutlich. Da die Laufzeiten der Basisblöcke auf Grund möglicher Allokationsentscheidungen nicht statisch sind, wird auf die Hilfsvariable $cDyn_k$ zurückgegriffen, die gemäß Gleichung 6.1.33 die Blocklaufzeit in Abhängigkeit von der aktuellen Allokation enthält.

Schließlich muss noch geklärt werden, wie die insgesamt benötigte Kopierdauer ihren Weg in die Variable findet. Dieser Höchstwert wird am Ende der eigentlichen Transaktion erreicht. Dieser Zeitpunkt wird im ILP mit der Variable $tEnd_k^i = 1$ für ein bestimmtes Datum d_i an einem Knoten v_k gekennzeichnet. Da die Variablenwerte die gemeinsame Dauer aller Transaktionen innerhalb eines DMA-Blocks modelliert, muss der Variablenwert auf die Summe der Dauer aller durchgeführten DMA-Transaktionen innerhalb des

Blocks gesetzt werden. Dies ist natürlich nur der Fall, wenn die Variable d_i selbst auch kopiert wird. Dieses Verhalten wird von Ungleichung 6.1.11 modelliert.

Die Constraints für die Werte der Variablen $rFrom_j^i$ werden analog gebildet. Lediglich die Rollen der ein- und ausgehenden Kanten wird vertauscht:

$$rFrom_j^i \geq f_j^i \tag{6.1.12}$$

$$rFrom_j^i \leq dur\,(all) \cdot f_j^i \tag{6.1.13}$$

$$rFrom_{out}^i \geq rFrom_{in}^i - f_{in}^i \cdot cDyn_k \tag{6.1.14}$$

$$rFrom_{out}^i \geq fStart_k^i \cdot \left(\sum_n fStart_k^n \cdot dur\,(d_n) \right) \tag{6.1.15}$$

Hilfsvariablen Die letzte Gruppe der Constraints definiert die Werte der notwendigen Hilfsvariablen. Auch für die Constraints dieser letzten Gruppe gelten folgende Variablenmengen: $d_i \in data\_objects$, $e_j \in E$, $v_k \in V \setminus \{v_{entry}, v_{exit}\}$, $e_{in} \in In\,(v_k)$ sowie $e_{out} \in Out\,(v_k)$. Alle Ungleichungen werden für alle möglichen Kombinationen dieser Variablenmengen erstellt. Diese werden unter anderem dafür genutzt, den gleichzeitigen Beginn und das gleichzeitige Ende eines DMA-Transaktions-Blocks zu sichern. Dazu werden zunächst die Indikatorvariablen, die die generelle DMA-Nutzung anzeigen, wie folgt belegt:

$$tInd_j \leq \sum_i t_j^i \tag{6.1.16}$$

$$tInd_j \geq t_j^i \tag{6.1.17}$$

$$fInd_j \leq \sum_i f_j^i \tag{6.1.18}$$

$$fInd_j \geq f_j^i \tag{6.1.19}$$

Darüber hinaus werden die Indikatorvariablen, die den Start oder das Ende einer DMA-Transaktion anzeigen benötigt und folgendermaßen belegt:

$$tEndInd_k \leq \sum_i tEnd_k^i \tag{6.1.20}$$

$$tEndInd_k \geq tEnd_k^i \tag{6.1.21}$$

$$fStartInd_k \leq \sum_i fStart_k^i \tag{6.1.22}$$

$$fStartInd_k \geq fStart_k^i \tag{6.1.23}$$

Mit Hilfe der folgenden Constraints werden dann das gemeinsame Ende bzw. der gemeinsame Beginn eines DMA-Blocks festgelegt:

$$tInd_{in} + tInd_{out} + tEndInd_k \leq 2 \tag{6.1.24}$$

$$fInd_{in} + fInd_{out} + fStartInd_k \leq 2 \tag{6.1.25}$$

Diese besagen, dass rund um einen Knoten v_k maximal 2 der angegebenen Indikatorvariablen den Wert eins annehmen dürfen. Durch die Ungleichungen nach Formel 6.1.24 sind nur noch folgende Kombinationen möglich:

- Sowohl auf der ein- und ausgehenden Kante e_{in} und e_{out} läuft ein Kopiervorgang ($tInd_{in}$ und $tInd_{out}$ haben den Wert eind, $tEndInd_k$ hat den Wert null)
- Auf der eingehenden Kante läuft ein Kopiervorgang, der am Knoten v_k beendet ist ($tInd_{in}$ und $tEndInd_k$ haben den Wert eins, $tInd_{out}$ hat den Wert null)
- Die dritte mögliche Kombination, nach der $tInd_{out}$ und $tEndInd_k$ den Wert eins annehmen und $tInd_{in}$ den Wert null annimmt, ist durch die Definition dieser Variablen ausgeschlossen:
  - Voraussetzung 1: Damit $tEndInd_k$ den Wert eins annimmt, muss nach den Formel 6.1.21 mindestens eine Variable aus $tEnd_k^i$ ebenfalls diesen Wert annehmen.
  - Voraussetzung 2: Damit $tInd_{out}$ den Wert eins annimmt, muss nach Formel 6.1.17 mindestens eine Variable aus t_{out}^i den Wert eins annehmen.
  - Voraussetzung 3: Schließlich darf gemäß Formel 6.1.16 keine der Variablen t_{in}^i den Wert eins annehmen, damit $tInd_{in}$ den Wert null annehmen kann.
  - Widerspruch: Da nach Voraussetzung 3 alle Variablen $t_{in}^i = 0$ sind, gilt nach Formel 6.1.27, dass alle für Variablen $tEnd_k^i = 0$ gilt. Dies widerspricht sich mit Voraussetzung 1.
- Die damit verbleibenden beiden validen Kombinationen besagen schließlich, dass eine DMA-Transaktion nur gemeinsam mit den übrigen laufenden als beendet gilt.

Dieser Nachweis lässt sich analog für die Ungleichung 6.1.25 führen.

Die Variablen $tEnd_k^i$, die das Ende eines spezifischen Datentransfers in den Scratchpad-Speicher anzeigen, bilden sich aus den Belegungen der Entscheidungsvariablen t_j^i:

$$tEnd_k^i \geq t_{in}^i - t_{out}^i \tag{6.1.26}$$

$$tEnd_k^i \leq t_{in}^i \tag{6.1.27}$$

$$tEnd_k^i + t_{in}^i + t_{out}^i \leq 2 \tag{6.1.28}$$

Die Variable $tEnd_k^i$ hat nach Ungleichung 6.1.26 den Wert eins, falls auf der eingehenden Kante eine Transaktion läuft ($t_{in}^i = 1$) und auf der ausgehenden nicht ($t_{out}^i = 0$). Sie hat nach Ungleichung 6.1.27 den Wert null, falls auf der eingehenden Kante keine Transaktion läuft ($t_{in}^i = 0$). Falls auf beiden Kanten eine Transaktion läuft ($t_{in}^i = t_{out}^i = 1$), hat $tEnd_k^i$ nach Ungleichung 6.1.28 ebenfalls den Wert null.

Die Variablen $fStart_k^i$ werden analog dazu gebildet:

$$fStart_k^i \geq f_{out}^u - f_{in}^i \tag{6.1.29}$$

$$fStart_k^i \leq f_{out}^i \tag{6.1.30}$$

$$fStart_k^i + f_{in}^i + f_{out}^i \leq 2 \tag{6.1.31}$$

Abschließend soll noch die letzte Variablenmenge $tStart_k^i$, die den Beginn eines spezifischen Datentransfers in den Scratchpad-Speicher anzeigt, definiert werden. Die Variablen aus dieser Menge wurden bislang nicht verwendet. Allerdings kennzeichnen sie für den Flow-Constraint die Stellen, an denen Kosten für den DMA-Aufruf entstehen.

$$tStart_k^i = t_{out}^i \wedge \neg t_{in}^i \tag{6.1.32}$$

Auch dieser Constraint entspricht nicht der allgemeinen ILP-Notation, allerdings kann auch sie durch regelgerechte Ersetzung in ein ILP eingefügt werden. Das Vorgehen hierzu ist in Abschnitt 6.3.2 erklärt.

Die Laufzeit eines durch den Knoten v_k repräsentierten Basisblocks in Abhängigkeit von der SPM-Allokation wird über die Entscheidungsvariable s_k^i modelliert:

$$cDyn_k = c(v_k) - \sum_i gain(d_i, v_k) \cdot s_k^i \tag{6.1.33}$$

WCET-Constraints und Zielfunktion

Die Bildung der Flow-Constraints zur Ermittlung der Gesamt-WCET verläuft dann wieder analog zur konventionellen dynamische Allokation. Sei daher v_0 erneut der Startknoten und v_t der Endknoten eines Kontrollflussgraphen. O.B.d.A. sei $P = (v_0, \ldots, v_t)$ ein Pfad, dessen Knoten weder einen Schleifenkopf repräsentieren, noch einen Funktionsaufruf enthalten. Dann lässt sich die WCET mit folgender Gleichung abschätzen:

$$w_k \geq w_{k+1} + cDyn_k + \sum_i tStart_k^i \cdot c(DMA) + \sum_i fStart_k^i \cdot c(DMA) \tag{6.1.34}$$

Diese Variable wird für alle Knoten $v_k \in V \setminus \{v_t\}$ und alle jeweiligen Nachfolgerknoten v_{k+1} in das ILP eingefügt, sofern v_k keinen Schleifenkopf repräsentiert.

Für den Endknoten v_t wird wieder eine gesonderte Vorschrift eingefügt, da keine ausgehenden Kanten existieren:

$$w_t = cDyn_t$$

Die Gesamt-WCET kulminiert schließlich wieder in der Variable w_0. Diese bildet damit auch erneut die Zielfunktion

$$w_0 \rightarrow min$$

Schleifen Schleifen und Funktionsaufrufe werden nach den selben Prinzipien wie in Kapitel 4 behandelt. Für einen Knoten v_{head}, der einen Schleifenkopf repräsentiert, werden an Stelle der Ungleichung 6.1.34 zwei (Un-)Gleichungen in das ILP eingefügt. Zunächst wird die WCET eines Schleifendurchlaufs in w'_{head} erfasst:

$$w'_{head} \geq w_{body} + cDyn_{head} + \sum_i tStart_{head}^i \cdot c(DMA) + \sum_i fStart_{head}^i \cdot c(DMA) \tag{6.1.35}$$

Der Knoten v_{body} ist dabei der erste Knoten des Schleifenrumpfs, der innerhalb der Schleif unmittelbar auf v_{head} folgt. Die WCET der einzelnen Schleifeniteration wird dann folgendermaßen in die Gesamt-WCET eingebunden:

$$w_{head} = w'_{head} \cdot loopcount\,(v_{head}) + w_{ext} \tag{6.1.36}$$

Die hier enthaltene kumulierte WCET w_{ext} repräsentiert die WCET des ersten auf die Schleifen folgenden Knotens v_{ext}.

Funktionsaufrufe Schließlich müssen noch die Funktionsaufrufe behandelt werden, damit die Flow-Constraints den kompletten IPCFG abdecken können. Zunächst wird, ähnlich wie bei Schleifen, die WCET eines einzelnen Funktionsdurchlaufs ermittelt. Der Funktion werden dann in der Graphenstruktur noch je ein virtueller Ein- und Austrittsknoten v_{f_entry} bzw. v_{f_exit} zugeordnet. Die WCET der Funktion ist damit in w_{f_entry} gespeichert. Die WCET der Funktion wird dann mit dieser Ungleichung in die WCET des aufrufenden Knotens übernommen:

$$\begin{aligned} w_{caller} \quad \geq \quad & w_{f\_entry} + w_{return} + cDyn_{caller} + \\ & \sum_i tStart^i_{caller} \cdot c\,(DMA) + \sum_i tStart^i_{caller} \cdot c\,(DMA) \end{aligned} \tag{6.1.37}$$

Dabei ist v_{caller} der Knoten, an dessen Ende die Funktion f aufgerufen wird. Der Knoten v_{f_entry} ist der virtuelle Einstiegsknoten der Funktion und nach Abarbeitung der Funktion wird der Kontrollfluss in v_{return} fortgesetzt. Die WCET des virtuellen Endknotens der aufgerufenen Funktion wird fix auf null gesetzt:

$$w_{f\_exit} = 0 \tag{6.1.38}$$

Im Gegensatz zur Allokation ohne DMA-Unterstützung ist für den Return-Knoten v_{return} keine gesonderte Behandlung notwendig, die die Platzierung des Kopier-Codes an die Knoten und nicht an die Kanten gebunden ist. Hier kann der übliche Flow-Constraint aus Formel 6.1.34 verwendet werden.

6.2 Codeallokation

Die DMA-gestützte Scratchpad-Allokation von Code ähnelt im Aufbau stark dem Modell der Datenallokation. Sofern anwendbar, werden die vereinfachenden Annahmen aus der Datenallokation mit übernommen. Parallel stattfindende DMA-Transaktionen beginnen und enden im Sinne des ILP-Modells jeweils zum selben Zeitpunkt. Diese Einschränkung ist erforderlich, da die Reihenfolge, in der parallel gestartete DMA-Transaktionen ausgeführt werden, nicht bekannt ist. Außerdem wird ein zu kopierender Basisblock ausschließlich über einen einzelnen DMA-Kanal durchgeführt.

Die Einschränkung, dass ein DMA-Block entweder in den SPM oder in den Hauptspeicher kopiert, ist in der Code-Allokation nicht erforderlich. Da der Code als statisch angesehen wird, ist ein Zurückkopieren des Codes in den Hauptspeicher ohnehin nicht notwendig.

6.2.1 ILP-Modell

Da das ILP-Modell zur Scratchpad-Allokation von Code in erster Linie eine Teilmenge des Modells zur Datenallokation ist, werden die einzelnen Vorschriften nur dann im Detail erläutert, falls es sich um Elemente handelt, deren Eigenschaften von ihren Gegenstücken im Daten-Modell abweichen.

Konstanten

Diejenigen Konstanten, die sich im Daten-Modell (unter anderem) auf ein bestimmtes Datenobjekt bezogen, beziehen sich im Code-Modell an Stelle dessen auf einen konkreten Basisblock b_i. Auch im Code-Modell werden die Größenkonstanten benötigt:

Definition 6.11. Sei $size\,(b_i)$ die Größe des Basisblocks b_i in Byte.

Definition 6.12. Sei $size\,(SPM)$ die Größe des Code-SPM in Byte.

Die Kostenkonstanten verändern sich im Vergleich zur Datenallokation nicht:

Definition 6.13. Sei $c\,(v_k)$ die WCET des durch v_k repräsentierten Basisblocks im Hauptspeicher.

Ebenso bleibt die Laufzeit zum Start einer DMA-Transaktion analog zu Definition 6.4 unverändert, da der Aufrufcode sich nur in den Adressbereichen, die als Start und Ziel der Kopieroperation eingetragen werden unterscheidet.

Definition 6.14. Sei $c\,(DMA) = 50$.

Die *gain*-Konstante ist nicht mehr von zwei Faktoren, sondern nur noch von einem Faktor, dem Basisblock b_i abhängig. Eine Abhängigkeit mit einem CFG-Knoten v_k ist deswegen nicht mehr gegeben, weil die Einlagerung eines Basisblocks in den Scratchpad-Speicher nur zur Laufzeit des Blocks selbst einen Einfluss auf die WCET hat. Zur Laufzeit aller übrigen Blöcke wird kein Gewinn erzielt.

Definition 6.15. Sei $gain\,(b_i)$ der erzielte Laufzeitgewinn bei Einlagerung von b_i in den Scratchpad-Speicher zum Zeitpunkt seiner Ausführung.

Die Definition der übrigen verwendeten Variablen ist deckungsglich mit denen der Datenallokation:

Definition 6.16. Falls v_k ein Schleifenkopf ist, dann sei $loopcount\,(v_k)$ die maximale Anzahl an Schleifeniterationen. Sonst sei $loopcount\,(v_k) = 1$.

Definition 6.17. Sei $dur\,(b_i)$ die Dauer eine Kopieroperation von b_i, die unter Ausnutzung der DMA-Einheit durchgeführt wird.

Die Berechnung der Werte von $dur\,(b_i)$ wird analog zur Berechnung der Werte von $dur\,(d_i)$ durchgeführt.

Definition 6.18. Sei $dur\,(all) = \sum_i (b_i)$.

Definition 6.19. Sei dma_count die Anzahl der verfügbaren DMA-Kanäle.

Variablen

Die Variablen bilden ebenfalls einen Ausschnitt der Variablen der Datenallokation ab. Aus diesem Grund wird auch hier auf eine detaillierte Beschreibung verzichtet. Zunächst die Haupt-Entscheidungsvariablen:

$$s_k^i = \begin{cases} 1, & \text{falls } b_i \text{ im Knoten } v_k \text{ im SPM liegt} \\ 0, & \text{sonst} \end{cases}$$

$$x_j^i = \begin{cases} 1, & \text{falls } b_i \text{ auf der Kante } e_j \text{ im SPM liegt} \\ 0, & \text{sonst} \end{cases}$$

$$t_j^i = \begin{cases} 1, & \text{falls } b_i \text{ auf der Kante } e_j \text{ per DMA in den SPM kopiert wird} \\ 0, & \text{sonst} \end{cases}$$

Die Hilfsvariablen werden ebenfalls analog übernommen. Die Variablen, die sich auf DMA-Transfers aus dem SPM in den Hauptspeicher beziehen, fallen weg:

$$tStart_k^i = \begin{cases} 1, & \text{falls zu Beginn des Knotens } v_k \text{ eine DMA-Transaktion des} \\ & \text{Basisblocks } b_i \text{ beginnt} \\ 0, & \text{sonst} \end{cases}$$

$$tEnd_k^i = \begin{cases} 1, & \text{falls zu Beginn des Knotens } v_k \text{eine DMA-Transaktion des} \\ & \text{Basisblocks } b_i \text{ endet} \\ 0, & \text{sonst} \end{cases}$$

$$tInd_j = \begin{cases} 1, & \text{falls auf der Kante } e_j \text{ mind. ein DMA-Kanal belegt ist} \\ 0, & \text{sonst} \end{cases}$$

$$tEndInd_k = \begin{cases} 1, & \text{falls am Knoten } v_k \text{ mind. ein DMA-Block endet} \\ 0, & \text{sonst} \end{cases}$$

Die Flussvariablen für die Laufzeiten der DMA-Transaktionen und die WCET haben folgende Bedeutung:

$$rTo_j^i \quad \hat{=} \quad \text{Abgelaufene Zeit eines DMA-Transfers des Basisblocks } b_i \text{ an der Kante } e_j$$

$w_k \hat{=}$Laufzeit des WCEP vom Knoten v_k zur Senke des Graphen

Die allokationsabhängige Laufzeitvariable $cDyn_k$ ist wie in der Datenallokation definiert:

$cDyn_k \hat{=}$Laufzeit des Knotens v_k in Abhängigkeit von der SPM-Allokation

Neu hinzu kommen die Kostenvariablen $cJump$ und $cCall$, die eventuell entstehende Zusatzkosten für vorher nicht vorhandene Sprünge zwischen Speicherbereichen modellieren:

$$\begin{aligned} cJump_k^l &= \text{Zusätzliche Sprungkosten, die durch unterschiedliche} \\ &\quad \text{Speicherorte von } v_k \text{ und } v_l \text{ auftreten.} \\ cCall_k^{f_n} &= \begin{cases} w_{f_n\_entry} + cJump_k^{f_n\_entry}, & \text{falls in } v_k \text{die Funktion } f_n \text{ aufgerufen wird} \\ 0, & \text{sonst} \end{cases} \end{aligned}$$

Dabei wird der Tatsache Rechnung getragen, dass Sprünge vom Hauptspeicher in den Scratchpad-Speicher und umgekehrt eine höhere Ausführungszeit haben, als Sprünge innerhalb eines Speicherbereichs. Da die vergleichsweise komplexe Berechnung dieser Kosten kein zentrales Element dieses ILPs sind, wird für die Berechnungsvorschriften auf [Kle08] verwiesen.

Constraints

Die benötigten Constraints lassen sich wieder in die zuvor erwähnten vier Gruppen unterteilen. Für die Constraints gilt ebenso wie für Konstanten und Variablen, dass sie im Prinzip nur eine Teilmenge der Constraints der Datenallokation darstellen.

Einhaltung der Systemeigenschaften Auch hier müssen die Größe des Scratchpad-Speichers und die Anzahl der verfügbaren DMA-Kanäle als einschränkende Faktoren in das ILP aufgenommen werden:

$$\forall e_j \in E : \sum_i t_j^i \cdot size\,(b_i) + \sum_i x_j^i \cdot size\,(b_i) \leq size\,(SPM) \tag{6.2.1}$$

$$\forall e_j \in E : \sum_i t_j^i \leq dma\_count \tag{6.2.2}$$

Sicherstellung der Konsistenz Um den Inhalt des Scratchpad-Speichers konsistent zu halten, ist in diesem Modell lediglich ein Constraint notwendig:

$$\forall b_i \forall v_k \in V \setminus \{v_{entry}\}\, \forall e_j \in In\,(v_k) :\ s_k^i = tEnd_k^i + x_j^i \tag{6.2.3}$$

Dauer der Kopiervorgänge Die Berechnungsvorschriften für die Variablenmenge rTo_j^i werden für alle möglichen und auf die einzelnen Constraints anwendbaren Kombinationen aus $b_i \in basic\_blocks$, $e_j \in E$, $v_k \in V \setminus \{v_{entry}, v_{exit}\}$, $e_{in} \in In\,(v_k)$ und $e_{out} \in Out\,(v_k)$ gebildet:

$$rTo_j^i \geq t_j^i \tag{6.2.4}$$

$$rTo_j^i \leq dur\,(all) \cdot t_j^i \tag{6.2.5}$$

$$rTo_{in}^i \geq rTo_{out}^i - t_{out}^i \cdot cDyn_k \tag{6.2.6}$$

$$rTo_{in}^i \geq tEnd_k^i \cdot \left(\sum_n tEnd_k^n \cdot dur\,(d_n) \right) \tag{6.2.7}$$

Die letzten beiden Ungleichungen müssen bei der eigentlichen ILP-Konstruktion gemäß Abschnitt 6.3.1 ersetzt werden.

Hilfsvariablen Die Vorschriften zur Belegung der Hilfsvariablen lassen sich direkt aus den Constraints der Datenallokation ableiten:

$$tInd_j \leq \sum_i t_j^i \tag{6.2.8}$$

$$tInd_j \geq t_j^i \tag{6.2.9}$$

$$tEnd_k^i \geq t_{in}^i - t_{out}^i \tag{6.2.10}$$

$$tEnd_k^i \leq t_{in}^i \tag{6.2.11}$$

$$tEnd_k^i + t_{in}^i + t_{out}^i \leq 2 \tag{6.2.12}$$

$$tEndInd_k \leq \sum_i tEnd_k^i \tag{6.2.13}$$

$$tEndInd_k \geq tEnd_k^i \tag{6.2.14}$$

$$tStart_k^i = t_{out}^i \wedge \neg t_{in}^i \tag{6.2.15}$$

Diese Constraints sind für alle anwendbaren Kombinationen aus $b_i \in basic\_blocks$, $e_j \in E$, $v_k \in V \setminus \{v_{entry}, v_{exit}\}$, $e_{in} \in In(v_k)$ und $e_{out} \in Out(v_k)$ zu bilden. Die Vorschrift zur Bildung von $tStart_k^i$ wird bei der endgültigen Aufstellung dann gemäß Abschnitt 6.3.2 ersetzt.

Die Laufzeit eines Basisblocks ist abhängig vom aktuellen SPM-Zustand wird analog zur Datenallokation definiert.

$$cDyn_k = c(v_k) - gain(v_k) \cdot s_k^i \tag{6.2.16}$$

WCET-Constraints und Zielfunktion

Der grundlegende Flow-Constraint für die DMA-gestützte Scratchpad-Allokation von Code lautet, leicht abgewandelt gegenüber der Datenallokation:

$$w_k \geq w_{k+1} + c(v_k) - gain(v_k) \cdot s_k^i + cJump_k^{k+1} + cCall_k + \sum_n tStart_k^n \cdot c(DMA) \tag{6.2.17}$$

mit $e = (v_k, v_{k+1})$ und b_i ist Basisblock zu v_k

Die Einbindung von Schleifen funktioniert wieder wie in beiden bisher vorgestellten ILP-Modellen mit Hilfe zweier Constraints:

$$w'_{head} \geq w_{body} + c(v_{head}) - gain(v_{head}) \cdot s_{head}^i + cJump_{head}^{body} + \sum_n tStart_{head}^n \cdot c(DMA)$$

(6.2.18)

Dabei ist v_{head} der Schleifenkopf, v_{body} der erste Basisblock innerhalb des Schleifenrumpfs und b_i der durch v_k repräsentierte Basisblock.

$$w_{head} = w'_{head} \cdot loopcount\,(v_{head}) + w_{ext} + cJump^{ext}_{head} \tag{6.2.19}$$

Der Knoten v_{head} repräsentiert weiterhin den Schleifenkopf und v_{ext} repräsentiert den Basisblock, der nach Verlassen der Schleife erreicht wird.

Die Zielfunktion, die die zu minimierende Gesamt-WCET enthält, lautet:

$$w_0 \rightarrow min$$

6.3 Spezielle ILP-Konstruktionen

In den vorgestellten ILP-Modellen zur DMA-gestützten Scratchpad-Allokation wurden jeweils Vorschriften verwendet, die der allgemeinen Form eines ILPs

$$\sum_j a_{ij} x_j \leq b_i$$

mit a_{ij} und b_i konstant widersprachen. Allerdings können derartige Konstruktionen durch das zusätzliche Einfügen regelgerechter Constraints ersetzt werden. In den gezeigten Modellen sind zwei Fälle zu unterscheiden: ein Produkt zweier ILP-Variablen und die logische Verknüpfung zweier Variablen. Für beide Fälle gibt es Möglichkeiten, durch Einführung einer neuen Variable, die das Ergebnis des gewünschten Terms enhält, das ursprüngliche Ziel zu erreichen.

6.3.1 Produkt zweier Variablen

Bisschop hat gezeigt, dass sich beliebige Produkte zweier ILP-Variablen durch eine Reihe neuer Constraints ersetzen lassen [Bis10]. In diesem Modell wird das Produkt einer Binär- und einer ganzzahligen Variable benötigt. Für diesen Sonderfall existiert eine deutlich vereinfachte Lösungsmethode, die hier vorgestellt werden soll. Zusätzlich zu den beiden zu multiplizierenden Variablen wird für die ganzzahlige Variable eine obere Schranke benötigt. Sei nun b die Binärvariable, i positiv und ganzzahlig mit einer oberen Schranke u. Dann gilt $r = b \cdot i$ durch folgende Ungleichungen:

$$\begin{aligned} r &\leq u \cdot b \\ r &\geq 0 \\ r &\leq i \\ r &\geq i - u \cdot (1 - b) \end{aligned}$$

Die Korrektheit lässt sich einfach zeigen: Falls $b = 0$, lauten die ersten beiden Gleichungen $r \leq 0$ und $r \geq 0$. Daraus folgt unmittelbar $r = 0$. Im Fall $b = 1$ lauten die letzten beiden Gleichungen $r \leq i$ und $r \geq i$. Daraus folgt dann unmittelbar $r = i$.

6.3.2 Logische Verknüpfungen

Die aussagenlogischen Operatoren $\{\wedge, \vee, \neg\}$ können jeweils durch Ungleichungen ersetzt werden, die der allgemeinen Form eines ILPs genügen. Damit können dann sämtliche zweiwertigen aussagenlogischen Terme durch eine entsprechende Anzahl an ILP-Ungleichungen ersetzt werden.

Seien a, b Binärvariablen und r das jeweilige Resultat. Dann lässt sich die Gleichung $r = a \wedge b$ durch folgende Ungleichungen ersetzen:

$$\begin{aligned} r &\leq a \\ r &\leq b \\ r &\geq a + b - 1 \end{aligned}$$

Die Gleichung $r = a \vee b$ lässt sich durch diese Ungleichungen ersetzen:

$$\begin{aligned} r &\geq a \\ r &\geq b \\ r &\leq a + b \end{aligned}$$

Und abschließend lässt sich $r = \neg a$ durch eine einzelne Gleichung ersetzen:

$$r = 1 - a$$

Die Korrektheit dieser Vorschriften lässt sich einfach an Hand einer Wahrheitstafel nachweisen.

6.4 WCC-Integration

Da beide vorgestellten ILP-Modelle in der Grundstruktur auf dem vorhandenen (Code) bzw. hier vorgestellten (Daten) dynamischen Allokations-Modell beruhen, wurden die Erweiterungen ebenfalls in die bestehenden Strukturen eingebettet. Die DMA-gestützte Allokation lässt sich nun mit einem einfachen Kommandozeilenschalter des Compilers ein- oder ausschalten. Dabei kann die Anzahl der nutzbaren DMA-Kanäle mit angegeben werden. Da für DMA-Operationen insgesamt 16 Kanäle zur Verfügung stehen, wird bereits bei der Auswertung der Kommandozeile überprüft, ob dieser Wert eingehalten wurde.

Die ILP-Modelle zur DMA-gestützten Scratchpad-Allokation wurden in die bereits vorhandenen konventionellen dynamischen Allokationen eingebettet. Die neu hinzugekommenen Variablen und Constraints, darunter insbesondere die Modellierung der Kopierdauer sowie der DMA-Blöcke, werden im Rahmen der bisherigen ILP-Erstellung mitgeneriert. Als grundlegende Datenstruktur bleibt der *IPCFG* aus der Scratchpad-Allokation von Kleinsorge erhalten.

Der grundsätzliche Ablauf der Optimierungen blieb damit unverändert. Die Auswertung der Optimierung des ILP-Solvers musste hingegen an einigen Stellen angepasst werden. Ursache dafür ist die Tatsache, dass Änderung der Speicherallokation und der

Ort der Spillcode-Platzierung nun getrennt sind. Je nach Struktur des Programms kann es dabei auch dazu kommen, dass es für eine Allokationsänderung mehrere Spillcode-Platzierungen gibt. Ebenso ist der umgekehrte Fall zu berücksichtigen, dass ein Aufruf des DMA-Codes an mehreren Punkten innerhalb des Kontrollflusses die Speicherallokation ändert.

Der generierte DMA-Code war zunächst für aiT nicht analysierbar, da der Speicherbereich, in dem die Register abgebildet sind, nicht in der Speicherkonfiguration aufgeführt waren. Ebenso waren keine Informationen über die Zugriffszeit hinterlegt. Diese konnte dann mit dem Evaluationsboard an der Hardware überprüft werden. Die Messungen ergaben, dass die Schreibzugriffe auf die Register ohne Zeitverzug durchgeführt werden konnten. Nach Annotation dieser Werte war aiT dann in der Lage, den DMA-Code mitzuanalysieren.

Die Code-Generierung, sowie die Abschätzung der Laufzeit der DMA-Transaktionen wurden in eine eigene Bibliothek ausgelagert.

6.4.1 DMA-Bibliothek

Die DMA-Bibliothek erfüllt zwei Aufgaben. Zum einen berechnet sie die voraussichtliche Dauer einer DMA-Transaktion anhand des ihr übergebenen Objekts. Darüber hinaus enthält sie die komplette Code-Generierung, um eine oder mehrere DMA-Transaktionen zu starten.

In ihr ist ein einfaches Modell integriert, um mögliche parallele DMA-Transaktionen von Code und Daten auf die verfügbaren Kanäle zu verteilen. Die Kanäle werden implizit gemäß ihrer dem Compiler über die Kommandozeile angegebenen Aufteilung fest zugeordnet. Die zahlenmäßige Aufteilung wird durch die jeweiligen ILP-Modelle sichergestellt, die konkrete Zuweisung wird dadurch getrennt, dass Transaktionen von Daten die Kanäle aufsteigend, beginnend mit Kanal 0 zugeteilt werden. Code-Transaktionen werden dagegen die Kanäle absteigend, beginnend mit Kanal 15 zugeteilt.

Wird in der Kommandozeile eine Aufteilung von sechs Kanälen für Datentransfers und zehn Kanälen für Codetransfers angegeben, so werden die Kanäle 0 bis 5 der Datenallokation und die Kanäle 6 bis 15 der Codeallokation zugeteilt.

7 Resultate

In diesem Kapitel werden die Resultate, die sich mit den einzelnen Optimierungen erzielen ließen, vorgestellt. Als Testgrundlage dienten eine Reihe von Benchmarks, die unterschiedliche Einsatzgebiete von eingebetteten Systemen abdecken sollen. Diese entstammten der WCC-eigenen Benchmark-Suite und waren damit für die WCET-Analyse dahingehend vorbereitet, dass beispielsweise die minimalen und maximalen Iterationszahlen von Schleifen im Quellcode annotiert waren und so direkt für die WCET-Analyse durch aiT nutzbar waren. Ursprünglich entstammen die einzelnen Benchmarks unter anderem der MediaBench [LPMS97], den MRTC Real-Time Benchmarks [MRT06], der NetBench [MMSH01], der DSPstone-Suite [ZVSM94], der StreamIt-Suite [MIT] und der UTDSP-Suite [Lee98].

Zunächst werden hier die Ergebnisse der Minimierung der Kopierkosten dargestellt. Danach werden die Ergebnisse der dynamischen SPM-Allokationen diskutiert und schließlich wird ein Überblcik über die WCET-Werte bei gemeinsamer SPM-Allokation von Code und Daten gegeben.

7.1 Verringerung der Kopierkosten

Im Kapitel 5 wurden insgesamt fünf verschiedene Ansätze zur Verringerung des Einflusses der Kopierkosten auf die WCET eines Programms vorgestellt. Zwei dieser Ansätze, die die Modifikation der Annotationen beinhalteten (s. Abschnitt 5.3), konnten nicht durchgeführt werden. Im ersten Fall hat aiT die Annotation von negativen Zusatzlaufzeiten nicht akzeptiert und damit überhaupt keine statische WCET-Analyse durchgeführt. Im zweiten Fall, der Annotation fester Laufzeiten einzelner Funktionen, wurde die Gesamtanalyse zwar durchgeführt, aber bedingt durch die Tatsache, dass aiT nach einer so annotierten Funktion von einem unbekannten Cache- und Pipeline-Zustand ausgeht, wird die WCET wesentlich höher eingeschätzt als ohne diese Annotationen.

Die Spezialisierung der Kopierfunktionen aus Abschnitt 5.2 hat in erster Linie dafür gesorgt, dass die Dauer einer solchen Kopieroperation in ihrer Gesamtheit besser abschätzbar wurde. In der bisherigen Variante war lediglich die Laufzeit der Kopierschleife von aiT dediziert abschätzbar, da der Kopiercode als Inline-Code eingefügt wurde. Die Befehle, die die Kopierschleife vorbereiten, waren durch aiT dann jeweils dem vorhergehenden Basisblock zugeordnet. Ebenso wurden die Befehle, die sich an die Kopierschleife anschlossen, wie zum Beispiel die Wiederherstellung der Register vom Stack, dem nachfolgenden Basisblock zugeschlagen.

Die Abschätzungen der Laufzeiten für die Kopierschleifen mit den in Tabelle 4.1 angegebenen Werten konnten bestätigt werden. Ebenso konnte in den Messungen ein Wert

von 20 Zyklen als obere Schranke für die Laufzeit des Rahmencodes um die eigentliche Kopierschleife herum ermittelt werden.

Der Ansatz aus Abschnitt 5.2, die Kopierbreite zu erweitern, lieferte mehrdeutige Ergebnisse. Ohnehin war mit keiner großen Steigerung des Laufzeitgewinns zu rechnen, da Speicherzugriffe im TC1796 mit einer Breite von 32 Bit durchgeführt werden. Allenfalls eine Verringerung der Zyklen in der *Instruction Fetch*-Phase war zu erwarten, da die Anzahl der Iterationen der Kopierschleife halbiert werden kann. Die Ausführungsdauer der 64 Bit breiten Varianten der Lade- und Speicherbefehle entspricht im wesentlichen der doppelten der 32-Bit-Variante.

Getestet wurde die Datenallokation an der Benchmark *ndes* mit spezialisierten Kopierfunktionen, um die Laufzeit der einzelnen Kopierfunktionen exakt bestimmen zu können. Bei einer gegebenen SPM-Größe von 20% der Gesamtdatengröße, dies entspricht 252 Byte, wurden im Laufe der Ausführung zwei Variablen jeweils in den Scratchpad-Speicher und den Hauptspeicher kopiert. Die Größe der Datenobjekte lag jeweils bei 49 Byte. Der Kopiercode bestand also aus zwölf 32 Bit und einer 8 Bit breiten Kopieroperation. Für die Kopien in den Scratchpad-Speicher wurde eine Laufzeitschranke von 272 bzw. 273 Zyklen festgestellt. Für die Kopie zurück in den Hauptspeicher betrugen die Werte 94 bzw. 100 Zyklen.

Wird die Breite der Kopieroperationen auf 64 Bit erweitert, besteht der Kopiercode aus sechs 64 Bit und einer 8 Bit breiten Kopieroperation. Die Werte veränderten sich nun uneinheitlich. Für die Kopien in den Scratchpad-Speicher wurden nun 88 Zyklen, also weniger als ein Drittel des ursprünglichen Wertes, berechnet. Für die Kopien in den Hauptspeicher stieg der Wert um ca. zwei Drittel auf 167 Zyklen.

Die Ursache für dieses Verhalten ist unklar. Es wird vermutet, dass es sich um Seiteneffekte handelt, die durch die veränderten Pipelinezustände während des Kopiervorgangs auftreten.

Der fünfte und letzte Ansatz zur Reduktion der Kopierkosten war die DMA-gestützte Scratchpad-Allokation. Die Resultate dieser Optimierung werden in Abschnitt 7.3 dargestellt.

7.2 Dynamische Datenallokation (ohne DMA)

Das in Kapitel 4 vorgestellte Modell zur dynamischen Scratchpad-Allokation von Daten stellt eine Verfeinerung des Modells von Rotthowe dar, welches eine Änderung des SPM-Inhalts nur an Funktions- und Schleifengrenzen erlaubte. Das hier vorgestellte Modell ermöglicht, die SPM-Allokation an allen Basisblockgrenzen zu verändern. Es wurde erwartet, dass die Allokation im Vergleich zum statischen Modell zu einer niedrigeren bzw. mindestens gleichwertigen WCET-Abschätzung kommt.

Als Test-Benchmark wurde erneut *ndes* verwendet, da die ansonsten getesteten Benchmarks entweder kein Potenzial für eine dynamische Allokation hatten oder die Übersetzung nicht komplett durchlief. In vielen Benchmarks gab es nur wenige Datenobjekte, die von der Allokation berücksichtigt werden können. Wenn diese dann parallel genutzt werden, wie es häufig bei Ein- und Ausgabevariablen der Fall ist, kann die dynami-

SPM-Größe	statische Allokation	dynamische Allokation
0	154972	154972
8	150872	149939
16	150869	161989
32	150613	155373
64	147031	158151
128	143191	157992
256	134520	144057
512	132185	132185
1024	131929	137660
2048	131161	137844

Tabelle 7.1: Ergebnisse der dynamischen Datenallokation für *ndes*

sche Allokation nicht greifen. Nur bei nacheinander und nicht unmittelbar wechselweise genutzten Variablen ist eine Verbesserung der WCET durch eine dynamische Datenallokation erwartbar.

Das Ergebnis der Analyse zeigt allerdings in den meisten Fällen keine Verbesserung der WCET. Lediglich bei einer Scratchpad-Größe von 8 Byte wird eine WCET erreicht, die unter dem der statischen Allokation liegt. Darüber hinaus wird noch einmal bei einer Scratchpad-Größe von 512 Byte ein zur statischen Allokation äquivalenter Wert erreicht. Auffällig ist zudem, dass bei den Größen von jeweils einschließlich 16 bis 128 Byte eine Allokation berechnet wird, deren WCET über der WCET ohne Optimierung liegt.

Die Werte aus Tabelle 7.1 zeigen, dass das Optimierungsmodell in einigen Konfigurationen offensichtlich fehlerhaft arbeitet, während es in anderen das erwartete Verhalten zeigt. Der Fehler konnte im Rahmen der Anfertigung dieser Arbeit leider nicht mehr lokalisiert werden, er liegt vermutlich u.a. bei der Berücksichtigung der Kopierkosten. Diese werden nach wie vor mit einem Modell berechnet, in welchem alle an einer Stelle notwendigen Kopieroperationen innerhalb einer Funktion liegen. Allerdings werden die notwendigen Kopieroperationen an Verzweigungen bzw. Vereinigungen gemäß Tabelle 4.2 innerhalb des Kontrollflussgraphen aufgeteilt und es kann dazu kommen, dass zusätzliche Sprünge eingebaut werden, deren Beitrag zur Laufzeit nicht in ausreichendem Maße gewürdigt wird bzw. Seiteneffekte hervorrufen, die im Voraus nicht absehbar waren. Erschwerend kommt hinzu, dass der Einbau dieser zusätzlichen Sprünge ausschließlich von der am Ende gewählten Allokation abhängt.

Grundsätzlich scheint das Allokationsmodell in dieser auf Basisblöcke verfeinerten Variante allerdings zu funktionieren, da der Spillcode nicht nur an Schleifen- und Funktionsgrenzen eingefügt wurde.

Benchmark	Ergebnis
adpcm_g721_board_test (DSPstone)	manueller Abbruch
adpcm_g721_verify (DSPstone)	manueller Abbruch
fir2dim (DSPstone)	statisch
fir (DSPstone)	statisch
iir_biquad_N_sections (DSPstone)	statisch
iir_biquad_one_section (DSPstone)	statisch
lms (DSPstone)	statisch
matrix (DSPstone)	statisch
h264dec_ldecode_macroblock (MediaBench)	statisch
md5 (NetBench)	statisch
bitonic (StreamIt)	statisch
filterbank (StreamIt)	statisch

Tabelle 7.2: Ergebnisse der Datenallokation auf den übrigen Benchmarks

7.3 DMA-gestützte Allokationen

Die DMA-gestützten Scratchpad-Allokationen von Code und Daten, wie sie in Kapitel 6 vorgestellt wurden, galten als der vielversprechendste Ansatz zur Reduktion der Kopierkosten. Durch die Parallelisierbarkeit der Kopien zur Ausführung des übrigen Codes, der zuverlässig reproduzierbaren Kopierdauer und dem konstanten WCET-Aufschlag des Starts einer DMA-Transaktion, erschien eine Modellierung dieser Allokation erfolgversprechend.

Die Allokationsmodelle lieferten jedoch durchweg statische Allokationen als Resultat, sofern die Optimierung terminierte. In einigen Fällen wurde die ILP-Optimierung nach einer Laufzeit von teilweise mehr als 24 Stunden ergebnislos abgebrochen. Beide Allokationsmodelle wurden mit allen Benchmarks der MRTC-Suite getestet. Die Ergebnisse sind in Tabelle 7.3 dargestellt. Da die dynamische Datenallokation, um wirkungsvoll arbeiten zu können, mehrere Datenobjekte benötigt, die nicht parallel verwendet werden, wurde die Datenallokation noch auf die DSPstone-Suite, die MediaBench, die NetBench und die StreamIt-Suite angewendet. Diese Ergebnisse sind in Tabelle 7.2 angegeben. Die in den Tabellen nicht aufgeführten Benchmarks lieferten entweder Compilerfehler außerhalb der Allokationsmodelle oder andere Teilprozesse des Compilers, wie der Code Selctor oder die WCET-Analyse mussten nach langer Laufzeit manuell abgebrochen werden. Bei der Datenallokation wurde jeweils eine Scratchpad-Größe von 50 % der Größe der vorhandenen Datenobjekte gewählt, bei der Code-Allokation wurde ein Wert von 256 Byte fest vorgegeben. Die in den Tabellen nicht aufgeführten Benchmarks lieferten entweder Compilerfehler außerhalb der Allokationsmodelle oder andere Teilprozesse des Compilers mussten nach langer Laufzeit manuell abgebrochen werden. Eine manuelle Analyse der gebildeten ILPs ergab, dass diese eine Verschiebung von Objekten prinzipiell zulassen. Dazu wurde eine *tEnd*-Variable fest auf den Wert eins gesetzt und überprüft, ob das ILP noch gelöst werden kann. Somit konnte eine fehlerhafte Generierung der ILPs

ausgeschlossen werden.

Die dahinterliegende Problematik lässt sich grob in drei Felder unterteilen:

1. Fehlendes Wissen über die *Best Case Execution Time* (BCET) für die Abschätzung der Kopierdauer im CFG.

2. Ein (zu) hoher statischer WCET-Aufschlag für kleine zu kopierende Objekte.

3. Die hohe Komplexität des ILP-Modells.

Das erste Problemfeld äußert sich in einer teilweise deutlichen Überschätzung des Bereiches einer Kopieraktion im CFG. Um sicherstellen zu können, dass eine per DMA durchgeführte Kopie zu einem bestimmten Zeitpunkt bzw. an einem bestimmten Knoten beendet ist, muss diese Kopie auch dann beendet sein, wenn das Programm über den Pfad mit der kürzesten Laufzeit, den *Best Case Execution Path* (BCEP), läuft. Um dies sicherzustellen, wird bei jeder Verzweigung bzw. Vereinigung des Kontrollflusses der kürzere Weg zum Anfang bzw. Ende des DMA-Blocks bei der Abbildung der Kopierdauer auf den Graphen berücksichtigt. Damit wird eine äußerst schlechte Schranke für die BCET im Sinne der Abbildung 2.1.1 angesetzt.

So kommt es dazu, dass natürliche Schleifen beispielsweise nur mit einer Iteration berücksichtigt werden und While-Schleifen unter Umständen nur mit der Laufzeit des Schleifenkopfs in die Kalkulation mit einfließen. Da während der zugehörigen DMA-Transaktion der jeweilige Speicherbereich im SPM bereits geblockt ist, ohne einen Laufzeitgewinn zu ermöglichen, muss das Ziel einer DMA-gestützten Allokation in möglichst engen Schranken für die Dauer einer Gruppe von DMA-Transaktionen liegen. In Abbildung 7.3.1 ist ein Beispiel für dieses Problemfeld dargestellt.

Anhand dieses Ausschnitts aus einem Kontrollflussgraphen kann das Problem zur Abschätzung der DMA-Blöcke innerhalb des CFG gut nachvollzogen werden. Angenommen, zu Beginn des untersten Knoten soll ein DMA-Block enden, der eine Laufzeit von 200 Zyklen hat. Mit jedem Schritt entgegen der Ausführungsreihenfolge durch den CFG wird der Wert von rTo um die Laufzeit des Knotens (hier innerhalb der Knoten angegeben) reduziert.

Da im ILP-Modell keine Informationen über die minimale Anzahl an Iterationen vorliegen, wird davon ausgegangen, dass die Schleife in diesem Beispiel nur eine Iteration hat. Damit hat die Variable rTo am Anfang des dargestellten CFG-Fragments für die verbleibende Laufzeit noch einen Wert von 80. Würde die minimale Iterationszahl der Schleife, hier zehn, allerdings zur Verfügung stehen, würde die Kopierdauer in diesem Graphen nur die beiden Schleifenknoten umfassen und nicht aus dem Fragment herausragen.

Das zweite Problemfeld ist eher kleinerer Natur. Während die konstante Anzahl von 50 Zyklen für die Initialisierung einer Kopie großer Felder oder Strukturen deutlich unter der Kopierdauer liegt, ist dieser Wert für die Kopie beispielsweise eines einzelnen Integer-Werts im Vergleich zu einem normalen Kopiervorgang innerhalb des Kontrollflusses deutlich zu hoch.

Benchmark	Ergebnis (Daten)	Ergebnis (Code)
adpcm_decoder	manueller Abbruch	manueller Abbruch
adpcm_encoder	manueller Abbruch	manueller Abbruch
binarysearch	statisch	statisch
compressdata	statisch	statisch
crc	statisch	statisch
edn	keine glob. Datenobjekte	statisch
expint	keine glob. Datenobjekte	statisch
fac	keine glob. Datenobjekte	statisch
fdct	statisch	statisch
fft1	statisch	statisch
fibcall	keine glob. Datenobjekte	statisch
fir	statisch	statisch
insertsort	statisch	statisch
janne_complex	keine glob. Datenobjekte	statisch
jfdctint	statisch	statisch
lcdnum	statisch	statisch
lms	statisch	statisch
ludcmp	statisch	statisch
matmult	statisch	statisch
minver	statisch	statisch
ndes	statisch	statisch
petrinet	statisch	statisch
prime	keine glob. Datenobjekte	statisch
qsort_exam	statisch	statisch
qurt	statisch	statisch
select	keine glob. Datenobjekte	statisch
st	statisch	statisch

Tabelle 7.3: Ergebnisse der Allokationsmodelle in der MRTC-Suite

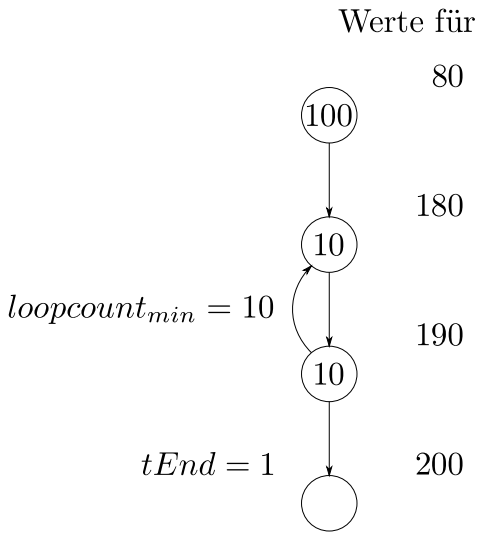


Abbildung 7.3.1: Beispiel für die Abschätzung der Kopierdauer im CFG

Während der Implementierung der DMA-gestützten Allokation war noch kein Wert für die Dauer der Initialisierung der DMA-Transaktion in Ermangelung passender Mess- oder Analysewerte im ILP-Modell enthalten. Damit war die Initialisierung einer DMA-Transaktion im Sinne des ILPs nicht mit einem Laufzeitaufschlag verbunden. Dies machte sich im Resultat dahingehend deutlich, dass hier dynamische Allokationen von kleinen Objekten auftraten.

Im endgültigen ILP, in dem diese Dauer berücksichtigt wird, kommt es allerdings nicht mehr zu Veränderungen in der SPM-Allokation.

Das letzte und vermutlich größte Problemfeld ist die hohe Komplexität des gebildeten ILP-Modells. Dies ist in erster Linie der Modellierung der Parallelität von DMA-Einheit und Prozessorkern geschuldet. Bei der Datenallokation führt es zum Beispiel dazu, dass die Anzahl der Variablen knapp dem 20-fachen des Produkts aus Basisblöcken und Datenobjekten entspricht. Für die Anzahl der Constraints lässt sich keine derartige Abschätzung abgeben, da diese noch mit der Anzahl der Kanten innerhalb des Kontrollflussgraphen zusammenhängen.

Als Beispiel für die Komplexität dienen die Benchmarks *ndes* und *adpcm_ decoder*: *ndes* hat 86 Basisblöcke und 13 Datenobjekte, deren Produkt liegt bei 1 118. Diese resultieren in einer Variablenanzahl von 21 761. Die Anzahl der Constraints ist mit 79 362 noch einmal deutlich höher. Bei *adpcm_ decoder* liegen die Werte für Basisblöcke und Datenobjekte bei 102 und 73. Das Produkt liegt damit bei 7 446. Das ILP enthält nun bereits 148 273 Variablen und 421 557 Constraints.

Die Lösungszeit eines solchen ILPs kann Stunden, wenn nicht Tage betragen und ist angesichts der an sich geringen Komplexität der Benchmarks nicht mehr akzeptabel. Auch wenn die Problemfelder selbst unabhängig voneinander sind, wird die Lösung eines

einzelnen nur wenig Einfluss auf die Gesamtproblematik haben. Während das zweite Problemfeld noch mit relativ geringem Aufwand zu lösen ist, wird die Einarbeitung der BCET in das Optimierungs-Modell unweigerlich einen Einfluss auf die schlussendliche Komplexität der generierten ILPs und damit auf das dritte Problemfeld haben, da bei der Kalkulation der Werte für $rFrom$ und rTo dann zusätzliche Constraints für die Behandlung von Schleifen eingebaut werden müssen.

7.4 Gemeinsame Durchführung der Allokationen

Die in der Literatur bisher bekannten SPM-bezogenen Optimierungen hatten immer entweder die Allokation von Daten oder die Allokation von Code zum Ziel. Bisher wurde offenbar noch nicht untersucht, in wie weit sich eine gleichzeitige Allokation von Code und Daten auf die WCET eines Programms auswirkt.

Es besteht die Annahme, dass sich bei einer gemeinsamen Allokation von Code und Daten Synergieeffekte bilden, so dass der reale Laufzeitgewinn über der Summe der einzelnen liegt. Diese Vermutung ist naheliegend, weil Programmcode, der aus dem Scratchpad-Speicher heraus ausgeführt wird, durch Zugriffe auf den langsamen Hauptspeicher über Gebühr ausgebremst werden kann. Umgekehrt kann der Zugriff auf ein Datum im Scratchpad-Speicher keinen Laufzeitgewinn ausspielen, wenn die zugehörige Instruktion noch nicht zur Ausführung gekommen ist.

Als Benchmark-Suite wurde die MRTC-Suite ausgewählt, da hier strukturell deutlich unterscheidbare Benchmarks in einer noch übersichtlichen Komplexität zusammengestellt sind. Die WCET-Analyse wurde für alle möglichen Kombinationen aus den SPM-Größen 0, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 und 4096, sowie die Optimierungsstufen `-O1` bis `-O3` durchgeführt. Untersucht wurde jeweils die statische Allokation, da die dynamischen Allokationen auf vielen Benchmarks nicht fehlerfrei laufen. Dabei hat sich gezeigt, dass die auffälligsten Effekte bei der maximalen Kombination, das heißt bei jeweils 4 KB Scratchpad-Speicher, erkennbar waren.

Dabei kam es nicht immer zu den erwarteten Synergieeffekten. Bei einigen Benchmarks, wie zum Beispiel bei den Benchmarks *countnegative, fir, ludcmp, sqrt* und *st* kam es bei einigen Optimierungsstufen sogar zu gegenteiligen Effekten. Allerdings bleiben diese in einem Bereich, der sich über Seiteneffekte erklären lässt. Die Analysewerte dieser Benchmarks sind in Diagramm 7.4.1 dargestellt. Die meisten Benchmarks unterliegen Laufzeitverbesserungen gegenüber der Addition der Einzelallokationen im meist niedrigen einstelligen Prozentbereich der nichtoptimierten Gesamtlaufzeit.

Eindeutig am stärksten ausgeprägt sind die Synergieeffekte bei Benchmarks, die in verschachtelten Schleifen häufig auf Daten zugreifen. In diesem Zusammenhang stechen die Sortieralgorithmen *Bubble Sort* und *Insertion Sort*, sowie die Matrix-Inversions-Benchmark *minver* heraus. Je nach Optimierungsstufe lassen sich maximale Laufzeitverbesserungen gegenüber den addierten Einzeloptimierungen im zweistelligen Prozentbereich erreichen. Die Benchmarks *bsort100, insertsort* und *minver* erreichen zusätzliche WCET-Verbesserungen zwischen 16 und 18 Prozent. Die Benchmarks mit einem zusätzlichen Laufzeitgewinn von über zehn Prozent werden in Diagramm 7.4.2 im einzelnen

dargestellt.

Angesichts der hier über die Einzeloptimierungen teilweise deutlich hinaus gehenden Gesamtergebnisse eröffnet sich an dieser Stelle ein neues Forschungsfeld, welches eine gemeinsame Optimierung der Scratchpad-Allokationen von Code und Daten analysiert.

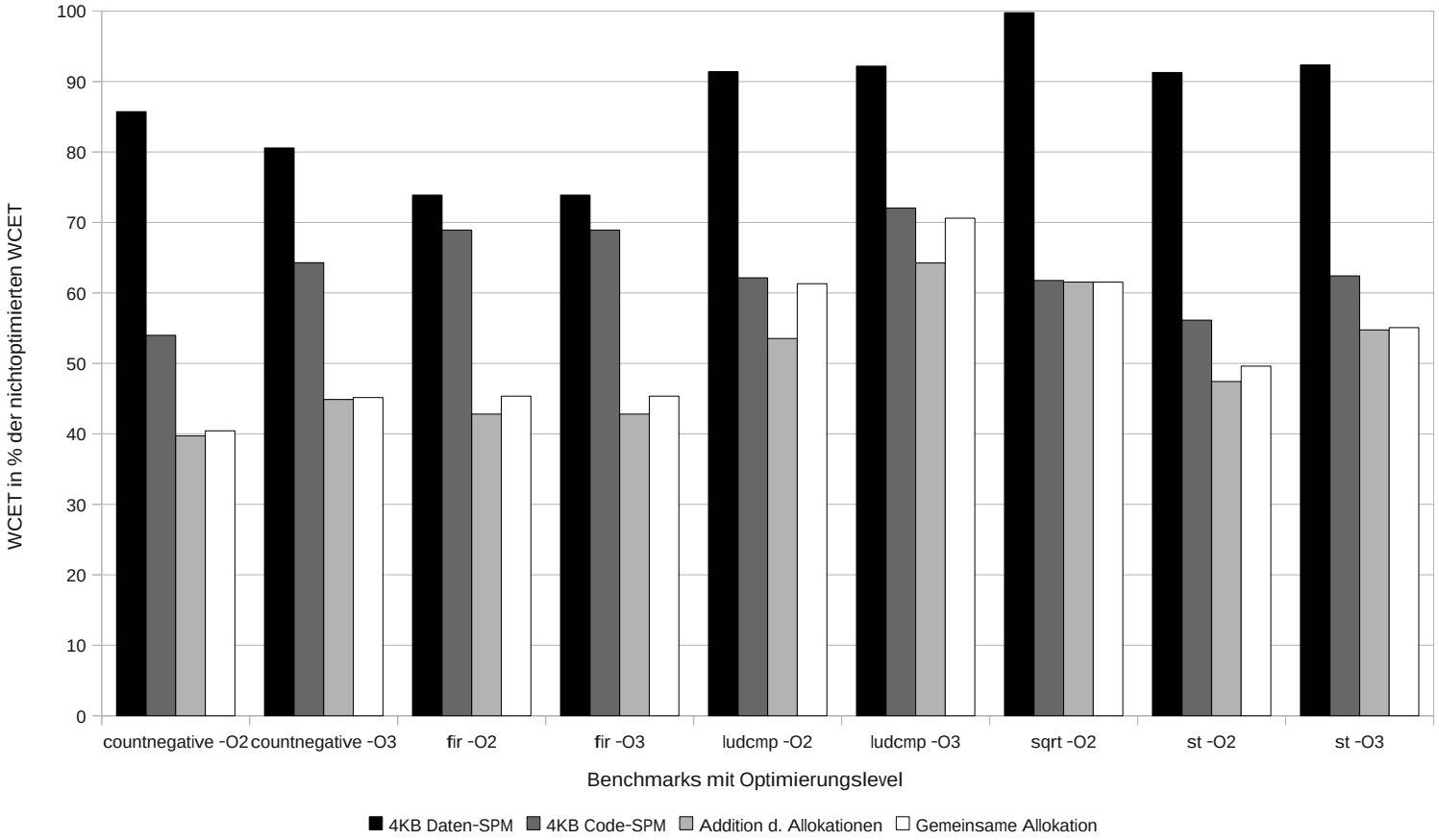


Abbildung 7.4.1: WCET-Vergleich bei gemeinsamer Allokataion von Code und Daten. Die in diesem Diagramm dargestellten Benchmarks lieferten WCET-Schranken, die über den erwarteten addierten Laufzeitverbesserungen lagen. Die Benchmark *ludcmp*, die das Lösen einer linearen Gleichung enthält, liefert Werte, die um etwa 7 Prozent über den erwarteten liegen. In absoluten Zahlen sind dies jedoch nur 528 bzw. 681 Zyklen, die auch durch Seiteneffekte innerhalb der Pipeline erklärbar sind.

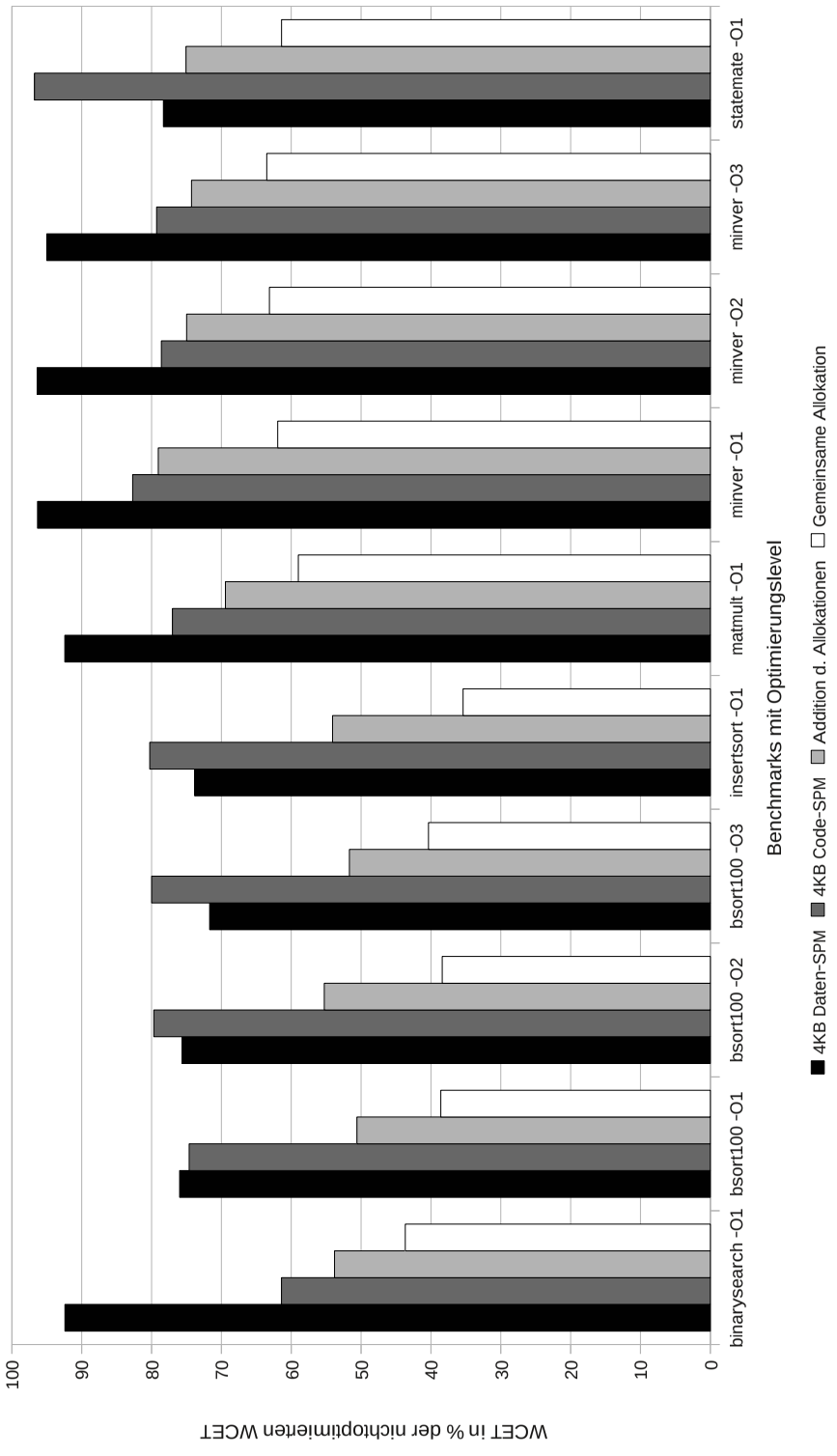


Abbildung 7.4.2: WCET-Vergleich bei gemeinsamer Allokation von Code und Daten. Hier sind diejenigen Benchmarks dargestellt, deren über den zu erwartenden Gewinn hinaus erzielte Laufzeitverbesserung im zweistelligen Prozentbereich lag.

8 Zusammenfassung/Ausblick

In diesem Kapitel werden die im Rahmen dieser Diplomarbeit erstellten Optimierungsansätze zusammengefasst und Vorschläge für weitere Ansätze zur Erschließung der Thematik gegeben.

8.1 Zusammenfassung

Das Hauptziel dieser Arbeit war es, neue Ansätze zur dynamischen Scratchpad-Allokation von Code und Daten zur WCET-Minimierung aufzustellen. Diese Ansätze sollten im Rahmen des am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelten *WCET-aware C Compiler* WCC implementiert werden. Hierbei galt es, Mittel und Wege zu finden, die Kopierkosten innerhalb einer dynamischen Allokation zu senken, da diese die durch dynamische Allokationen erzielbaren WCET-Einsparungen weitgehend amortisieren.

Zunächst wurde ein ILP-Modell präsentiert, welches mit konventionellen Kopierfunktionen arbeitet, die innerhalb des normalen Kontrollflusses liegen. Die Granularität der Allokationsentscheidungen sollte im Gegensatz zu den bekannten Modellen auf Basisblockebene liegen, so dass für jeden einzelnen Basisblock eine unterschiedliche Scratchpad-Belegung möglich ist. Innerhalb dieses WCET-Modells wurden einerseits die Scratchpad-Inhalte innerhalb jedes Basisblocks des zu übersetzenden Programms modelliert. Andererseits musste im ILP-Modell ebenso der *Worst Case Execution Path*, also der Pfad durch den Kontrollfluss eines Programms, dessen Laufzeit maximal ist, mit modelliert werden.

Da mit der im Vergleich zu Rotthowe wesentlich höheren Flexibilität der SPM-Belegungen auch die Variabilität der Stellen wächst, an denen der Kopiercode platziert werden muss, musste dieser Aspekt gesondert berücksichtigt werden. Im bisherigen Modell war die Änderung der Scratchpad-Belegung nur an Grenzen von Funktionen und Schleifen vorgesehen. Die Konstruktion der Kontrollflussgraphen ist an diesen Grenzen recht einheitlich und so existierten nur wenige Varianten, die für die Platzierung des Spillcodes berücksichtigt werden mussten.

Im hier vorgestellten Modell wird die Platzierung des Kopiercodes auf fünf verschiedene Varianten aufgeteilt. Vier davon sind ausschließlich von der Anzahl der ein- und ausgehenden Kanten eines Knotens des CFG abhängig. Die letzte Variante betrifft Rücksprungkanten aus Unterfunktionen. Da innerhalb der Unterfunktion zur Laufzeit nicht bekannt ist, von wo sie aufgerufen wurde, kann am Ende der Funktion keine Fallunterscheidung getroffen werden, welcher SPM-Zustand herzustellen ist. Deswegen wird der Kopiercode in solchen Fällen grundsätzlich am Anfang des Rücksprungknotens in der aufrufenden Funktion platziert.

Für die Reduktion der Kopierkosten wurden eine Reihe von Ansätzen untersucht, die dem Problem auf unterschiedliche Art und Weise begegnen. Die beiden Ansätze, die Kopierkosten nur mittels Annotationen zu reduzieren, galten als reine Machbarkeitsuntersuchung um herauszufinden, wie sich die WCET-Schranken bei minimierten Kosten für die Kopieroperationen verhalten. Da im Rahmen dieser Ansätze keine sinnvolle bzw. im Falle der negativen Annotationen sogar überhaupt keine WCET-Analyse mit Hilfe von aiT möglich war, kann diese Überlegung im derzeitigen Stand von aiT nicht weiter verfolgt werden.

Weitere Ansätze zur Kostenreduktion beschäftigten sich mit der Modifikation der Kopieroperationen selbst. Während die Spezialisierung der Kopierfunktionen lediglich die Abschätzbarkeit der Kosten vereinheitlichte, lieferte die Modifikation der Breite der Kopieroperationen von 32 auf 64 Bit ein ambivalentes Ergebnis. Für einige Kopieroperationen stieg die Höhe der WCET-Schranke, für andere hingegen sank sie. Mit einer signifikanten Verbesserung der Laufzeit wurde ohnehin nicht gerechnet, da die interne Struktur des TC1796 auf 32 Bit breiten Operationen basiert. Dieses ambivalente Ergebnis war allerdings in dieser Form unerwartet.

Der letzte Ansatz zur Kostenminimierung erschien am vielversprechendsten. Der als Zielsystem verwendete TC1796 verfügt über eine *Direct Memory Access*-Einheit, die es ermöglicht, parallel zum normalen Kontrollfluss Speicherbereiche zu kopieren. Für die Nutzung der DMA-Einheit wurden sowohl für die Daten- als auch für die Code-Allokation ILP-basierende Modelle vorgestellt. Innerhalb dieser Modelle musste neben der SPM-Konsistenz und dem WCEP-Aufbau auch die Dauer der einzelnen DMA-Transaktionen modelliert werden. Dazu mussten zunächst reale Messwerte gewonnen werden, um dafür Kostenabschätzungen vornehmen zu können.

Die Abbildung der Dauer einer DMA-gestützten Kopie auf einen Bereich im IPCFG führte zu einer deutlichen Erhöhung der Komplexität des ILP-Modells. Diese hohe Komplexität einerseits und die, mangels vorhandener Informationen über die Laufzeit im Idealfall, unzureichende Abbildung der Kopierdauer auf den IPCFG führten dazu, dass die erreichten Ergebnisse der DMA-gestützten Analyse die statische Analyse nicht übertreffen können.

Zu diesem Punkt kommt die Problematik des konstanten Zeitaufwands für jeden Kopier-Auftrag. Dieser ist für große zu kopierende Objekte zwar recht klein, allerdings leiden diese am meisten unter der unzureichenden Abbildung der Kopierdauer auf den Kontrollflussgraphen. Umgekehrt ist dieser Aufwand für besonders kleine Objekte im Vergleich zum eigentlichen Kopieraufwand überproportional groß, so dass der erzielbare Laufzeitgewinn in der Regel kompensiert wird.

Untersuchungen, in denen der Zeitaufwand für einen Kopierauftrag nicht im ILP vorhanden war, lieferten demzufolge auch dynamische Allokationen. Da der Kopiercode in der WCET-Analyse wiederum berücksichtigt wird, kam es dabei erwartungsgemäß nicht zu Optimierungen.

Schließlich ist die hohe Komplexität des ILP-Modells vermutlich Ursache dafür, dass das ILP-Modell bei etwas komplexeren Benchmarks nicht in akzeptabler Zeit lösbar war

und in der Regel nach 24 Stunden reiner Prozessorlaufzeit abgebrochen wurde.

Abschließend wurden noch die Auswirkungen einer gemeinsamen statischen Scratchpad-Allokation von Code und Daten analysiert. Hierbei wurde festgestellt, dass die Summe der Einzeloptimierungen als erwarteter Gewinn in den meisten Fällen teilweise sogar deutlich überschritten wurde. Für einzelne Benchmarks konnte die erwartete WCET-Optimierung von etwa 25 % mit Werten von 36 und 39 % signifikant übertroffen werden.

8.2 Ausblick

Der Großteil der vorgestellten Optimierungen und Ergebnisse kann als Grundlage für weitere Forschungsarbeiten dienen. Von diesen werden nun einige vorgestellt:

Die Nutzung von 64 Bit breiten Kopieroperationen lieferte mehrdeutige Werte. Die Kopien ins SPM lieferte niedrigere WCET-Schranken, die Kopien in den Hauptspeicher höhere. Hier kann untersucht werden, ob diese Unterschiede systematisch sind. Falls dies der Fall ist, kann die Breite der Kopierfunktionen vom Ziel der Kopie abhängig gemacht werden.

Die konventionelle dynamische Scratchpad-Allokation von Daten liefert derzeit Modelle, die die WCET-Schranke in einigen Fällen gegenüber der statischen Optimierung nicht senken. Dies ist vermutlich auf Probleme in der Abschätzung der Kopierkosten zurückzuführen. Es bietet sich in diesem Rahmen an, die Platzierung des Spillcodes nach anderen, als den hier vorgestellten Kriterien durchzuführen. Beispielsweise könnte an Stelle des hier vorgestellten knotenbasierten Modells eine kantenbasierte Platzierung eingeführt werden, die mit einer niedrigeren Anzahl an zusätzlichen Sprüngen auskommt. In diesem Rahmen könnte auch eine Modifikation der bedingten Sprünge durchgeführt werden, um die Zahl der zusätzlichen unbedingten Sprünge innerhalb der Fallunterscheidungen zu verringern.

Die DMA-gestützten Allokationsmodelle liefern auf Grund von Problemen in drei Bereichen keine dynamischen Allokationen. Um die bislang unzureichende Abbildung der Dauer eines DMA-Blocks auf den Kontrollflussgraphen näher an den realen Verhältnissen auszurichten, fehlendem ILP-Modell derzeit Informationen über den BCEP beziehungsweise die BCET des zu übersetzenden Programms. Mit den Informationen über Iterationsuntergrenzen einzelner Schleifen, wie sie in den meisten Benchmarks bereits annotiert sind, kann ein Modell entwickelt werden, welches eine realistischere Abbildung der Kopierdauer auf den Graphen enthält.

Das zweite weiter zu bearbeitende Feld von Interesse für weitere Untersuchungen betrifft die anfallenden Kosten eines DMA-Transfers. Diese sind zwar konstant, liegen aber für kleine Objekte in einem Bereich, der deutlich über denen einer gewöhnlichen Inline-Kopie läge. Zum einen kann versucht werden, die Kosten über eine Optimierung des generierten Assembler-Codes zu senken. Zusätzlich kann analysiert werden, bei welcher Größe des zu kopierenden Objekts die Kosten für DMA- und Inline-Kopie identisch sind und entsprechend DMA- oder Inline-Code platzieren.

Das letzte Feld ist eher grundsätzlicher Natur. Das hier vorgestellte Modell erreicht für einige Benchmarks eine Komplexität, die dazu führt, dass die notwendige Zeit, um das ILP zu optimieren mehrere Stunden oder sogar ganze Tage umfasst. Da eventuelle

Lösungsansätze für die beiden ersten Bereiche zwangsläufig einen Einfluss auf die Komplexität des gesamten Modells haben, erscheint es sinnvoll, einen Ansatz zu wählen, der die verschiedenen Aspekte der Probleme gleichermaßen berücksichtigt. Eventuell ist es möglich, die Dauer eines DMA-Blocks nicht mit in das ILP-Modell aufzunehmen, sondern einen Hybrid-Ansatz zu entwickeln, der bezüglich des ILPs eine geringere Komplexität aufweist.

Schließlich haben die gemessenen Werte für gemeinsame Allokationen von Code und Daten gezeigt, dass sich für den Großteil der untersuchten Benchmarks teils deutliche Synergieeffekte gezeigt haben. Es erscheint naheliegend, ein Modell zu entwickeln, welches eine kombinierte Analyse und Optimierung durchführt, um diejenigen Allokations-Kombinationen zu identifizieren, die den größten Realgewinn versprechen.

Abbildungsverzeichnis

Literaturverzeichnis

[Abs] AbsInt Angewandte Informatik GmbH. CRL version 2. `http://www.absint.com/artist2/doc/crl2/`. abgerufen am 15. August 2010.

[Abs08] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time and Stack Analysis : aiT/StackAnalyzer for TriCore - Documentation for Linux/Solaris, Juni 2008. Version 2.0.

[Abs10] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzers. `http://www.absint.com/ait/`, August 2010. abgerufen am 15. August 2010.

[AP01] Dimitris Alevras und Manfred W. Padberg. *Linear optimization and extensions : problems and solutions*. Universitext. Springer, Berlin [u.a.], 2001.

[AP06] Alexis Arnaud und Isabelle Puaut. Dynamic instruction cache locking in hard real-time systems. In *14th International Conference on Real-Time and Network Systems, 2006. RTNS '06.*, pages 179–188, 2006.

[BCP02] Guillem Bernat, Antoine Colin, und Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. *IEEE International Real-Time Systems Symposium*, pages 279–288, 2002.

[BEGL05] Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, und Björn Lisper. Applying static WCET analysis to automotive communication software. pages 249 – 258, jul. 2005.

[Bis10] Johannes Bisschop. *AIMMS : Optimization Modeling*. Paragon Decision Technology B.V., Haarlem, 2010.

[Dan57] George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–277, 1957.

[Dij68] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commununications of the ACM*, 11(3):147–148, 1968.

[DP07] J.-F. Deverge und I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *19th Euromicro Conference on Real-Time Systems, 2007. ECRTS '07.*, pages 179–190, 4-6 2007.

[Evi09] Evidence srl. Erika Enterprise and RT-Druid. `http://erika.tuxfamily.org/`, 2009. abgerufen am 17. August 2010.

[FL10] Heiko Falk und Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, pages 1–50, 2010. 10.1007/s11241-010-9101-x.

[IBM10] IBM. IBM ILOG CPLEX data sheet, Juni 2010.

[Inf07] Infineon Technologies. TC1796 : User's Manual - Volume 1 (of 2): System Units, Juli 2007.

[Inf08a] Infineon Technologies. TC1796 : Data Sheet, V1.0, April 2008.

[Inf08b] Infineon Technologies. TriCore 1 User's Manual, Volume 2 - Instruction Set, V 1.3 & V 1.3.1 Architecture, Januar 2008. V 1.3.8.

[Inf10a] Informatik Centrum Dortmund e.V. ICD-C Compiler Framework. `http://www.icd.de/es/icd-c/icd-c.html`, 2010. abgerufen am 15. August 2010.

[Inf10b] Informatik Centrum Dortmund e.V. ICD-LLIR Low-Level Intermediate Representation. `http://www.icd.de/es/icd-llir/icd-llir.html`, 2010. abgerufen am 15. August 2010.

[JHH09] Xiangtao Jiang, Zhigang Hu, und Jianbiao He. WCET-guided optimal data allocation to scratchpad memory. In *WASE International Conference on Information Engineering, 2009. ICIE '09.*, volume 2, pages 297–301, 10-11 2009.

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, New York, NY, USA, 1984. ACM.

[Kle08] Jan C. Kleinsorge. WCET-centric code allocation for scratchpad memories. Diplomarbeit, Technische Universität Dortmund, September 2008.

[Kle10] Jan C. Kleinsorge. Infineon TriCore - ErikaWiki. `http://erika.tuxfamily.org/wiki/index.php?title=Infineon_Tricore`, März 2010. abgerufen am 17. August 2010.

[Lee98] Corinna G. Lee. UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, Mai 1998. abgerufen am 21. August 2010.

[LM95] Yau-Tsun Steven Li und Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Notices*, 30(11):88–98, 1995.

[LPMS97] Chunho Lee, Miodrag Potkonjak, und William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[Mar07] Peter Marwedel. *Eingebettete Systeme*. eXamen.press. Springer, Berlin u.a., 2007.

[MIT] MIT Computer Architecture Group Home Page. StreamIt - Benchmarks. `http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml`. abgerufen am 23. August 2010.

[MMSH01] Gokhan Memik, William H. Mangione-Smith, und Wendong Hu. Netbench: a benchmarking suite for network processors. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 39–42, Piscataway, NJ, USA, 2001. IEEE Press.

[MRT06] MRTC - Mälarden Real-Time Research Center. MRTC Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, September 2006. abgerufen am 21. August 2010.

[Muc04] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, San Francisco, 2004.

[PS91] Chang Yun Park und Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5):48–57, 1991.

[Pua06] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *18th Euromicro Conference on Real-Time Systems, 2006*, pages 217–226, 0-0 2006.

[Rot08] Felix Rotthowe. Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung. Diplomarbeit, Technische Universität Dortmund, August 2008.

[Sha89] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[SMRC05] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, und Ting Chen. WCET centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium, 2005. RTSS 2005.*, pages 223–232, 8-8 2005.

[VLX03] Xavier Vera, Björn Lisper, und Jingling Xue. Data cache locking for higher program predictability. *SIGMETRICS - Performance Evaluation Review*, 31(1):272–282, 2003.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, und Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, 7(3):1–53, 2008.

[Weg03] Ingo Wegener. *Komplexitätstheorie : Grenzen der Effizienz von Algorithmen*. Springer-Lehrbuch. Springer, Berlin, 2003.

[Wu09] Cheng-Ying Wu. A stack-optimized scratchpad memory allocator for reducing either the average-case or the worst-case execution time. Master's thesis, National Sun Yat-sen University, Kaohsiung, Taiwan, R.O.C., 2009.

[ZVSM94] Vojin Zivojnovic, Juan M. Velarde, Christian Schlager, und Heinrich Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology*, 1994.