technische universität
dortmund

fakultät für
informatik

Masterarbeit

Cache-bewusste Code-Positionierung zur Reduktion der maximalen Programmlaufzeit (WCET)

Helena Kotthaus
19. Januar 2011

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

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Entwurfsautomatisierung für
Eingebettete Systeme

GERMANY · D-44221 DORTMUND

Inhaltsverzeichnis

1. Einleitung

Im Folgenden wird der Themenbereich der vorliegenden Arbeit motiviert (Abschnitt 1.1). Es werden die Ziele der Arbeit (Abschnitt 1.2) sowie die verwandten Arbeiten (Abschnitt 1.3) vorgestellt. Abschließend erfolgt eine Darstellung über den Aufbau der Arbeit (Abschnitt 1.4).

1.1. Motivation

Diese Arbeit ist dem Themengebiet der eingebetteten Systeme zuzuordnen. *Eingebettete Systeme* sind informationsverarbeitende Systeme, welche in ein übergeordnetes System integriert sind und für dieses komplexe Dienste leisten. Durch die Einbettung in ein größeres Produkt sind diese Systeme für den Benutzer meist transparent [Mar10]. Der Einsatz dieser Systeme erstreckt sich über eine Vielzahl von Anwendungsbereichen. Beispiele für eingebettete Systeme sind Antiblockiersysteme, Airbag-Systeme oder Flugzeugkontrollsysteme. Aber nicht nur Flugzeuge und Kraftfahrzeuge beinhalten eine Vielzahl von eingebetteten Systemen, auch in Fabriksteuerungen, medizinischen Geräten oder Verbraucherelektronik wie Mobiltelefonen oder Waschmaschinen sind eingebettete Systeme vorhanden. Durch den breiten Anwendungsbereich nimmt die Relevanz dieser Systeme stetig zu. So wurde der Begriff der Post PC Ära geprägt, da die Anzahl eingebetteter Systeme bereits weit über der Anzahl der klassischen PC-Systeme liegt.

Anforderungen, die an eingebettete Systeme gestellt werden, unterscheiden sich dabei stark von denen der konventionellen Softwaresysteme. Der Aufwand für die Entwicklung eines eingebetteten Systems ist daher oft größer. Während bei der Entwicklung von konventionellen Systemen z. B. Laufzeiteffizienz und geringe Kosten im Vordergrund stehen, liegen die Zielsetzungen bei der Entwicklung von eingebetteten Systemen auch auf der Effizienz des Energieverbrauches, der Codegröße, der physikalischen Größe und des Gewichtes. So besitzen diese Systeme meist nur einen kleinen Speicher und oft keine stationäre Stromversorgung.

Da eingebettete Systeme häufig Einsatz in sicherheitskritischen Bereichen finden, werden hier ebenfalls hohe Anforderungen an die Zuverlässigkeit und die Sicherheit gestellt. Die meisten dieser Systeme besitzen daher auch Eigenschaften von *Echtzeitsystemen*. Ein System ist echtzeitfähig, wenn es eine Antwort innerhalb einer fest vorgegebenen *Zeitschranke* garantieren kann. Es wird zwischen weichen und

harten Echtzeitsystemen unterschieden. Bei Systemen mit *weichen Echtzeitanforderungen* führt eine Überschreitung der vorgegebenen Zeitschranken lediglich zu einem Qualitätsverlust (z. B. bei Multimediaanwendungen). Bei Systemen mit *harten Echtzeitanforderungen* führt jedoch selbst eine einmalige Überschreitung der Zeitschranke zum Gesamtversagen des Systems und kann sogar eine Katastrophe verursachen (z. B. bei Steuerung des Airbags) [Kop00].

Um die Einhaltung von Zeitschranken garantieren zu können, ist die maximale Programmlaufzeit, die sogenannte *Worst Case Execution Time* (WCET) zu bestimmen. Jedoch stellt die Bestimmung der WCET eines Programms bei der steigenden Softwarekomplexität von eingebetteten Systemen ein Problem dar. So ist eine manuelle Bestimmung der WCET kaum möglich. Zwar könnten zur Einhaltung der Zeitschranken mehr Hardwareressourcen eingesetzt werden, dies würde jedoch im Bereich der eingebetteten Systeme keine optimale Lösung darstellen. Denn der Einsatz von zusätzlichen Speichern oder leistungsstärkeren Prozessoren führt unter anderem zu einem höheren Energieverbrauch und zusätzlichen Kosten. Da die WCET nur schwer zu bestimmen ist, werden statt der tatsächlichen WCET sichere obere Schranken für diese bestimmt. Diese Schranken können z.B. über eine statische Analyse gewonnen werden. Dabei ist eine möglichst präzise Bestimmung der WCET-Schranken von großer Bedeutung. Für klassische Softwaresysteme werden bereits seit langem Compiler-Optimierungen eingesetzt, um die durchschnittliche Laufzeit, die sogenannte *Average Case Execution Time* (ACET) zu reduzieren. Dies ist mit Hilfe von speziellen Compilern für eingebettete Systeme auch für die WCET möglich. Wird die WCET eines Programms reduziert, so werden Hardwareressourcen eingespart und damit gleichzeitig der Energieverbrauch und die Kosten des Systems gesenkt. Daher spielt die Optimierung der WCET im Bereich der eingebetteten Systeme eine tragende Rolle.

Bei klassischen Systemen werden Speicherhierarchien eingesetzt, die schnelle, kleine Speicher beinhalten, sogenannte *Caches*. Der Einsatz von Caches führt zu einer erheblichen Reduktion der ACET, da das sogenannte *Memory Wall Problem* reduziert wird. Das Memory Wall Problem beschreibt die immer größer werdende Lücke zwischen der Speicher- und der Prozessorgeschwindigkeit [WM95]. Der Cache dient hier als Puffer zwischen dem langsamen Hauptspeicher und dem Prozessor. Er nutzt die räumliche und zeitliche Lokalität von Speicherzugriffen aus und arbeitet zudem autonom, da er von der Hardware gesteuert wird. Somit sind Caches nicht nur einfach zu integrieren, sondern reduzieren neben der ACET ebenfalls den Energieverbrauch [VM07], da ein Hauptspeicherzugriff wesentlich mehr Energie benötigt als ein Zugriff auf einen kleineren Speicher wie den Cache.

Trotz der sehr guten Laufzeiteffizienz kann der Einsatz von Caches in eingebetteten Echtzeitsystemen zu Problemen führen. So ist die Zugriffszeit eines Datums abhängig vom Cacheinhalt. Der Cacheinhalt wird dabei zur Laufzeit von der Hardware bestimmt, dies erschwert die Vorhersagbarkeit des Zeitverhaltens. Befindet sich

das Datum beim Zugriff im Cache, so wird dies als *Cache-Hit* bezeichnet, liegt das Datum jedoch nur im Hauptspeicher, da es z. B. aufgrund der beschränkten Cachegröße verdrängt wurde, so wird dies als *Cache-Miss* bezeichnet. Liegt ein Cache-Miss vor, so ist das Datum zunächst aus dem Hauptspeicher in den Cache zu übertragen, dies führt zu einer wesentlich höheren Zugriffszeit und zu einem höheren Energieverbrauch als bei einem Cache-Hit. Ob ein Speicherzugriff zu einem Cache-Hit oder Cache-Miss führt, ist nicht immer vorhersagbar. Da im schlechtesten Fall von einem Cache-Miss auszugehen ist, kann dies bei der Bestimmung der WCET zu einer Überabschätzung führen. Für eine bessere Vorhersagbarkeit des Zeitverhaltens werden daher in eingebetteten Systemen häufig sogenannte *Scratchpad-Speicher* (engl. Sratch Pad Memory, kurz SPM) eingesetzt [WM04]. Diese sind ähnlich wie Caches kleine, schnelle Speicher, die in den Prozessor integriert sind. Dabei wird der Inhalt des Speichers nicht während der Laufzeit von der Hardware bestimmt, sondern manuell beim Entwurf des Systems vom Entwickler oder vom Compiler. Da diese Vorgehensweise jedoch im Gegensatz zur Verwendung von hardwaregesteuerten Caches sehr aufwändig ist, wird der Einsatz von Caches in eingebetteten Systemen dennoch angestrebt.

Gerade beim Einsatz von *Instruktionscaches* ist es möglich, das Verhalten des Speichers und damit die Vorhersagbarkeit des Zeitverhaltens zu verbessern. Dies kann über eine Optimierung des Speicherlayouts erreicht werden. Ein schlechtes Cacheverhalten, welches zu einer Überabschätzung der WCET führt, zeichnet sich durch eine hohe Anzahl an möglichen Cache-Misses aus. Wird die Anzahl dieser Cache-Misses reduziert, so kann einer Überabschätzung der WCET entgegengewirkt werden. Im Bereich der klassischen *Compiler-Optimierungen* gibt es bereits ein bekanntes Verfahren zur Verbesserung des Cacheverhaltens, dieses ist die sogenannte *Code-Positionierung*. Speicherobjekte, welche auf dieselben Bereiche im Cache abgebildet werden und eine hohe zeitliche Lokalität aufweisen, weil sie z. B. in einer Schleife aufgerufen werden, verdrängen sich häufig gegenseitig aus dem Cache. Dies führt zu einer hohen Anzahl an Cache-Misses. Diese Art der Cache-Misses wird auch als *Konflikt-Misses* bezeichnet, da die auslösenden Speicherobjekte im Konflikt stehen, wenn sich ihre Speicherbereiche im Cache überlappen. Diese Überlappung im Cache kann dabei durch eine kontinuierliche Anordnung der Objekte im Hauptspeicher aufgelöst werden. Somit nimmt gleichzeitig die Anzahl möglicher Konflikt-Misses ab. Eine Code-Positionierung kann dabei auf verschiedenen Ebenen arbeiten. So können z. B. *Basisblöcke*, dies sind Instruktionssequenzen (siehe Abschnitt 2.1.1), Funktionen oder ganze Tasks positioniert werden.

Im Bereich der Compiler-Optimierungen für eingebettete Systeme gibt es jedoch kaum Code-Positionierungsverfahren, die eine Reduktion der WCET bewirken und damit einen effizienten Einsatz von Caches ermöglichen. Genau hier setzt diese Arbeit an, im Folgenden werden nun die Ziele der Arbeit detailliert erläutert.

1.2. Ziele der Arbeit

Im Rahmen der Forschungsarbeiten des Lehrstuhl 12 für eingebettete Systeme der TU Dortmund wurde der *WCC* (WECT-aware C Compiler), ein WCET-optimierender Compiler, entwickelt [FL10]. Mit diesem ANSI C-Compiler ist es möglich, vollautomatisiert WCET-gesteuerte Optimierungen durchzuführen. Die WCET wird dabei während des Übersetzungsvorgangs durch das statische Analysewerkzeug *aiT* der Firma AbsInt Angewandte Informatik GmbH bestimmt [Abs11]. Durch den automatisierten Optimierungsprozess wird der überdimensionierte Einsatz von Hardware zur Einhaltung der WCET-Schranken oder eine zeitaufwändige manuelle Optimierung der WCET vermieden.

Innerhalb dieses Compilers existieren bereits zahlreiche *WCET-Optimierungen*, welche umfassend in [LM10] beschrieben werden. Im Rahmen dieser Arbeit wird eine weitere neue WCET-Optimierung für den WCC entwickelt. Diese Optimierung reduziert die Überabschätzung der WCET beim Einsatz von Instruktionscaches. Durch eine *Cache-bewusste Code-Positionierung* wird das Verhalten des Instruktionscaches verbessert. Dabei werden Speicherobjekte so angeordnet, dass die Anzahl möglicher Cache-Misses reduziert wird. Als Zielarchitektur dient hierbei der im automotiven Bereich weit verbreitete TriCore Prozessor *TC1797* der Firma Infineon. Die Realisierung der WCET-Optimierung gliedert sich in zwei Bereiche.

Cache-Analyse:
Um die Cache-Misses zu reduzieren ist es zunächst notwendig zu ermitteln, welche Speicherobjekte miteinander im Konflikt stehen. Diese Informationen werden anhand einer Analyse des Instruktionscaches gewonnen. Dabei soll Aussage darüber getroffen werden, welches Speicherobjekt S_i ein anderes Speicherobjekt S_j im Instruktionscache verdrängt, und wie häufig diese Verdrängung im schlechtesten Fall passiert. Ein Graph hilft hierbei, die Konflikte zwischen den Objekten zu modellieren. Als Speicherobjekte werden sowohl Basisblöcke innerhalb von Funktionen als auch die Funktionen selbst analysiert.

Optimierungsstrategie:
Für die Code-Positionierung ist eine Optimierungsstrategie erforderlich. Auf Basis der Informationen der Cache-Analyse, werden hierbei im Konflikt stehende Basisblöcke und Funktionen kontinuierlich im Hauptspeicher angeordnet. Dadurch soll die Anzahl der Cache-Konflikte verringert und somit die WCET reduziert werden.

Um die Ziele dieser Arbeit von anderen Arbeiten abzugrenzen und Arbeiten vorzustellen, die als Grundlage verwendet werden, folgt nun im nächsten Abschnitt eine Vorstellung verwandter Arbeiten.

1.3. Verwandte Arbeiten

Im Rahmen dieser Arbeit wird eine neue Cache-bewusste Code-Positionierung zur Reduktion der WCET entworfen, dabei werden mengenassoziative Instruktionscaches betrachtet. Die verwandten Arbeiten hierzu gliedern sich in unterschiedliche Bereiche. Viele Arbeiten befassen sich nur mit der Cache-bewussten Code-Positionierung zur Optimierung der ACET, ohne die WCET zu betrachten. Es gibt jedoch auch Arbeiten, die gezielt die WCET optimieren, jedoch ohne eine spezielle Cache-Analyse vorzunehmen. Des Weiteren sind im Bereich der Optimierung des Energieverbrauches von eingebetteten Systemen verwandte Arbeiten zu finden, da die Reduktion von Cache-Misses nicht nur zur Reduktion der WCET führt, sondern auch zur Optimierung des Energieverbrauches.

Die Code-Positionierung wurde bereits auf unterschiedlichen Ebenen erprobt. So gibt es Code-Positionierungen von Basisblöcken über Funktionen bis hin zu ganzen Tasks. In [ZWH05] wird eine Positionierung von Basisblöcken zur Reduktion der WCET vorgestellt. Dabei werden die Basisblöcke so positioniert, dass unbedingte Sprünge zwischen den Blöcken vermieden werden. Dies führt zur Reduktion von *Pipeline Delays* und damit auch zur Optimierung der WCET. Allerdings beschränkt sich diese Arbeit nur auf einfache Prozessorsysteme ohne Cache.

[GRB04] hingegen beschreibt eine Compiler-Optimierung, welche mit Caches arbeitet. Hier werden Funktionen mit Hilfe eines sogenannten *Call-Graphen* positioniert. Der Call-Graph modelliert dabei die Aufrufe zwischen den einzelnen Funktionen des Programms. Funktionen, die sich häufig gegenseitig aufrufen, werden im Speicher kontinuierlich angeordnet, so dass eine gegenseitige Verdrängung aus dem Cache vermieden wird. Diese Optimierung ist jedoch nur auf direkt-abgebildete Caches beschränkt und betrachtet lediglich die Reduktion der ACET. Außerdem werden die Informationen zum Aufbau des Call-Graphen über *Profiling* gewonnen, so dass die getesteten Programme mehrmals mit unterschiedlichen Eingabedaten auszuführen sind. Eine Positionierung von Funktionen, die auch mengenassoziative Caches unterstützt wird in [LMi10] beschrieben. Hier steht jedoch ebenfalls die Reduktion der ACET im Vordergrund. Außerdem werden die Informationen für die Positionierung wie in [GRB04] über Profiling gewonnen.

In [GA07] werden ganze Tasks positioniert, um die Anzahl der Cache-Misses zu reduzieren. Diese Arbeit konzentriert sich jedoch auf das *präemptive Scheduling* in Multitask Systemen. Sich unterbrechende Tasks werden so positioniert, dass ihnen unterschiedliche Teile des Caches zugeordnet werden. Dieses Verfahren optimiert die ACET und soll ebenfalls dazu dienen, die Abschätzung der WCET bei präemptivem Scheduling zu verbessern.

Die für diese Arbeit wichtigsten verwandten Arbeiten stellen [LFM08] und [VWM04]

dar. Dabei werden in [LFM08] Funktionen anhand eines Call-Graphen positioniert, um die Anzahl der Cache-Misses zu reduzieren. Diese Optimierung ist in den WCC des Lehrstuhls 12 für eingebettete Systeme der TU Dortmund integriert, und hat damit die Reduktion der WCET zum Ziel. Funktionen, die sich häufig gegenseitig aufrufen, werden ähnlich wie in [GRB04] im Hauptspeicher kontinuierlich angeordnet, so dass sie sich nicht gegenseitig aus dem Cache verdrängen, da sich ihre Cachezeilen nicht mehr überlappen. Die Optimierungsstrategie geht dabei nach einer *Greedy-Heuristik* vor. Sie unterstützt ebenfalls mengenassoziative Caches, es werden jedoch lediglich Funktionen anhand eines Call-Graphen positioniert.

Die zweite Arbeit [VWM04], welche ebenfalls am Lehrstuhl 12 für eingebettete Systeme der TU Dortmund entstanden ist, konzentriert sich auf die Reduktion des Energieverbrauches von eingebetteten Systemen. Hier wird eine Optimierung vorgestellt, welche mit einer Speicherhierarchie bestehend aus Hauptspeicher, SPM und Cache arbeitet. Durch den Einsatz des Scratchpad-Speichers wird nicht nur die Anzahl der Cache-Misses reduziert, sondern auch der Energieverbrauch. Der SPM wird dabei als Befehlspuffer eingesetzt. Die Interaktion der Speicherobjekte im Instruktionscache wird über einen *Konfliktgraphen* modelliert. Dieser Graph dient als Grundlage für das Energiemodell und den Allokationsalgorithmus. Der Konfliktgraph beinhaltet dabei Informationen über Basisblöcke, die sich gegenseitig aus dem Cache verdrängen und so zu einer hohen Anzahl an Cache-Misses führen bzw. zu einem hohen Energieverbrauch. Um die Konflikte aufzulösen, wird die optimale Menge an Basisblöcken in den SPM ausgelagert. Da der Konfliktgraph eine gute Möglichkeit darstellt, das Cacheverhalten zu modellieren, wird dieser unter kleinen Änderungen als Grundlage der Cache-Analyse dieser Arbeit verwendet werden.

Einige der verwandten Arbeiten haben wie diese Arbeit die Reduktion der WCET zum Ziel. Dabei arbeiten die Code-Positionierungsverfahren jedoch z. B. auf Basis eines Call-Graphen, statt eine Cache-Analyse einzusetzen. Nach der Vorstellung und Abgrenzung der verwandten Arbeiten folgt nun abschließend zu diesem Kapitel die Beschreibung des Aufbaus der Arbeit.

1.4. Aufbau der Arbeit

Kapitel 2: In diesem Kapitel werden die Grundlagen zu den Themengebieten der Arbeit vermittelt. Hierzu gehören die Methoden zur Analyse der WCET sowie die Funktionsweise des in der Arbeit verwendeten WCET-Analysewerkzeuges aiT. Außerdem werden der Aufbau und die Analyse von Caches sowie die Funktionsweise der Code-Positionierung vorgestellt.

Kapitel 3: Hier wird das am Lehrstuhl 12 entwickelte WCC (WCET-aware C Compiler) Compiler-Framework vorgestellt, welches die Grundlage dieser Arbeit

darstellt. Dabei wird besonders auf die interne Zwischendarstellung des Compilers eingegangen, die von der Optimierung dieser Arbeit genutzt wird. Abschließend wird die vom WCC unterstützte TriCore Plattform beschrieben, welche als Zielarchitektur verwendet wird.

Kapitel 4: Da eine Cache-bewusste Code-Positionierung Informationen über das Cacheverhalten benötigt, wird hier die in der Arbeit entworfene Cache-Analyse vorgestellt, welche Cache-Konflikte anhand eines Graphen modelliert.

Kapitel 5: Die Code-Positionierung der Arbeit erfolgt auf zwei Ebenen. Zum einen werden Basisblöcke innerhalb von Funktionen positioniert, und zum anderen werden die Funktionen selbst positioniert. Die zugehörigen Algorithmen der beiden Ebenen werden in diesem Kapitel vorgestellt.

Kapitel 6: Hier wird das entworfene Code-Positionierungsverfahren evaluiert. Dabei werden verschiedene Benchmarks in Bezug auf die Reduktion der WCET und der Cache-Misses ausgewertet. Außerdem wird auf die Laufzeit der Optimierung eingegangen.

Kapitel 7: Dieses Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick über mögliche Erweiterungen der entwickelten Optimierung.

2. Grundlagen

In diesem Kapitel werden die Grundlagen zu den Themengebieten der vorliegenden Arbeit vermittelt. Hierzu werden die Methoden zur Analyse der WCET vorgestellt (Abschnitt 2.1) und das in dieser Arbeit verwendete statische WCET-Analysewerkzeug aiT beschrieben (Abschnitt 2.2). Um die Abschätzung der WCET beim Einsatz von Caches zu verbessern, ist der Einsatz von Cache-Analysen unerlässlich. Es werden daher ebenfalls einige Methoden zur Cache-Analyse erläutert (Abschnitt 2.3). Da diese Arbeit eine Cache-bewusste Code-Positionierung zum Ziel hat, um so die Abschätzung der WCET beim Einsatz von Caches zu optimieren, wird abschließend auf die allgemeine Funktionsweise der Code-Positionierung eingegangen (Abschnitt 2.4).

2.1. WCET-Analyseverfahren

Eingebettete Realzeitsysteme müssen Zeitschranken einhalten können. Um die Einhaltung dieser Schranken zu garantieren, ist es notwendig die maximale Laufzeit eines Programms zu bestimmen, welche als *Worst Case Execution Time* (WCET) bezeichnet wird. In diesem Abschnitt wird zunächst die Bedeutung der WCET genauer erläutert (Abschnitt 2.1.1). Hierfür wird der Aufbau des Kontrollflussgraphen eines Programms beschrieben. Es wird auf die Problematiken eingegangen, die bzgl. der Bestimmung des längsten Programmpfades auftreten. Außerdem wird sowohl das dynamische (Abschnitt 2.1.2) als auch das statische WCET-Analyseverfahren (Abschnitt 2.1.3) vorgestellt.

2.1.1. WCET

Die Bestimmung der WCET eines Programms stellt jedoch ein Problem dar, denn sie ist von vielen Faktoren abhängig. Je nach Eingabedaten und Zielarchitektur kann die Programmlaufzeit stark variieren. Um die WCET zu ermitteln, müssten die Eingabedaten, die zur maximalen Laufzeit führen, bekannt sein. Die Problematik der Bestimmung der exakten WCET lässt sich auch auf das sogenannte Halteproblem reduzieren, welches im allgemeinen Fall unentscheidbar ist [Weg05].

Statt die reale WCET zu bestimmen, werden daher sichere obere Schranken für diese abgeschätzt. So wird im Allgemeinen zwischen der realen $WCET_{real}$ und der ge-

schätzten $WCET_{est}$ unterschieden. Dies gilt analog für die Bestimmung der kleinstmöglichen Ausführungszeit, der sogenannten *Best Case Execution Time* (BCET). Der Zusammenhang zwischen der $WCET_{real}$ und der $WCET_{est}$ wird in Abbildung 2.1 verdeutlicht. Hier sind die möglichen Ausführungszeiten eines Programms dargestellt, welche je nach Eingabedaten stark variieren. Die möglichen Ausführungszeiten eines Programms liegen im Intervall von $BCET_{real}$ und $WCET_{real}$.

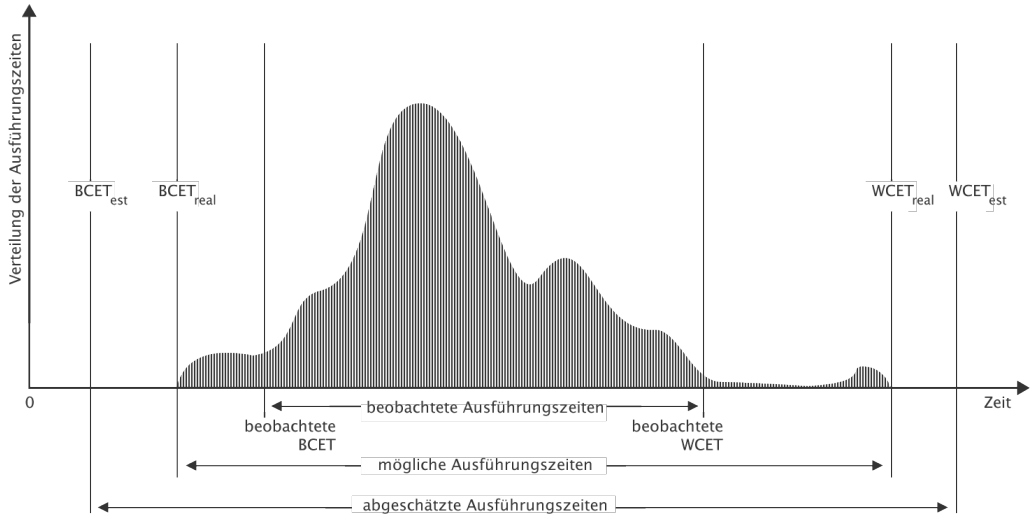

Abbildung 2.1.: WCET-Begriff nach [WEE08].

Bei der Abschätzung der realen WCET sind zwei wichtige Bewertungskriterien zu beachten. Die $WCET_{est}$ muss *sicher* und möglichst *präzise* sein. Sie ist sicher, wenn gilt:

$$WCET_{real} \leq WCET_{est} \tag{2.1}$$

Sie darf also niemals kleiner als die reale WCET sein, da sonst die Einhaltung von Zeitschranken nicht garantiert werden kann. Jedoch reicht eine sichere $WCET_{est}$ allein nicht aus, um ein eingebettetes System effizient zu entwerfen. Die Abschätzung der WCET sollte zusätzlich noch möglichst präzise sein. Je geringer die Differenz $WCET_{est} - WCET_{real}$ ist, desto präziser ist die $WCET_{est}$ und desto effizienter gestaltet sich der Ressourcenverbrauch des eingebetteten Systems. Denn eine zu große Überabschätzung der $WCET_{real}$ kann zu einer großen Verschwendung von Ressourcen führen. So würde der Einsatz von leistungsfähigerer Hardware benötigt, um die Überabschätzung zu kompensieren, obwohl dies bzgl. der realen WCET nicht notwendig wäre. So kommt es z. B. beim Einsatz von Caches häufig zu einer großen Überabschätzung der realen WCET. Dies liegt daran, dass das Zeitverhalten von Caches nur schwer vorhersagbar ist. Die in dieser Arbeit konzipierte Compiler-Optimierung versucht, dem Problem der Überabschätzung beim Einsatz von Caches entgegenzuwirken. Für eine präzise Abschätzung der $WCET_{real}$ gilt es also, die

$WCET_{est}$ so weit wie möglich zu reduzieren. Da die reale WCET nur abgeschätzt werden kann, wird im Folgenden der Begriff der WCET für die $WCET_{est}$ verwendet.

In [WEE08] werden verschiedenen Verfahren zur Bestimmung der WCET beschrieben und Werkzeuge zur WCET-Analyse vorgestellt und verglichen. So kann zur WCET-Analyse der *Kontrollflussgraph* eines Programms verwendet werden, um die Ausführungspfade des Programms zu untersuchen.

Definition 2.1 *Ein Kontrollflussgraph ist ein gerichteter Graph $G = (V, E)$ bestehend aus einer Knotenmenge $V = \{b_1, ..., b_n\}$, aller Basisblöcke eines Programms und einer Kantenmenge E von Kontrollflusskanten. Eine Kontrollflusskante $b_i \rightarrow b_j$ existiert, wenn der Basisblock b_j ein direkter Nachfolger von b_i ist und somit direkt nach b_j ausgeführt wird.*

Definition 2.2 *Ein Basisblock besteht aus einer Menge $I = \{i_1, ..., i_n\}$ von aufeinander folgenden Instruktionen. Er kann nur durch die erste Instruktion i_1 betreten und nur durch die letzte Instruktion i_n verlassen werden.*

Kontrollflussverzweigungen wie *bedingte Sprünge* sind somit nur am Ende eines Basisblocks möglich. Ein Pfad im Kontrollflussgraphen besteht aus einer Ausführungsreihenfolge von Basisblöcken. Er wird durch den ersten Basisblock (Quelle) der Startfunktion betreten und kann über einen Endknoten aus der Menge der Senken verlassen werden. Die Menge der Endknoten besteht aus den Basisblöcken, in denen das Programm terminieren kann. Dabei ist nicht jeder Pfad des Kontrollflussgraphen ausführbar. So können Basisblöcke mit einer Instruktion verlassen werden, die einen bedingten Sprung darstellt. Trifft die Bedingung niemals zu, so wird dieser Pfad auch als *unerreichbarer Pfad* (engl. Infeasible Path) bezeichnet, da er im Programm niemals zur Ausführung kommt. Werden diese Pfade bei der WCET-Analyse erkannt, können sie ausgeschlossen werden, was eine präzisere Abschätzung der WCET ermöglicht.

Für die Optimierung der WCET spielt das Wissen über den längsten Ausführungspfad eines Programms, den sogenannten *Worst Case Execution Path* (WCEP), eine wichtige Rolle. Wird die Länge des WCEPs verkleinert, so reduziert dies auch die WCET. Denn die Anzahl der Prozessorzyklen, die auf dem WCEP anfallen, entspricht der realen WCET. Um die WCET zu reduzieren ist es nötig, den Programmcode, der auf dem WCEP liegt, zu optimieren. Optimierungen von Programmcode außerhalb des WCEPs haben keinerlei Auswirkung auf die Reduktion der WCET. Eine Optimierung, welche den WCEP verkürzt, kann jedoch auch zu einem Wechsel des WCEPs führen. In Abbildung 2.2 wird diese *Instabilität* des WCEPs verdeutlicht.

In den dargestellten Kontrollflussgraphen ist die Anzahl der Prozessorzyklen der Basisböcke eingetragen. Im linken Graphen besteht der WCEP aus den Basisblöcken

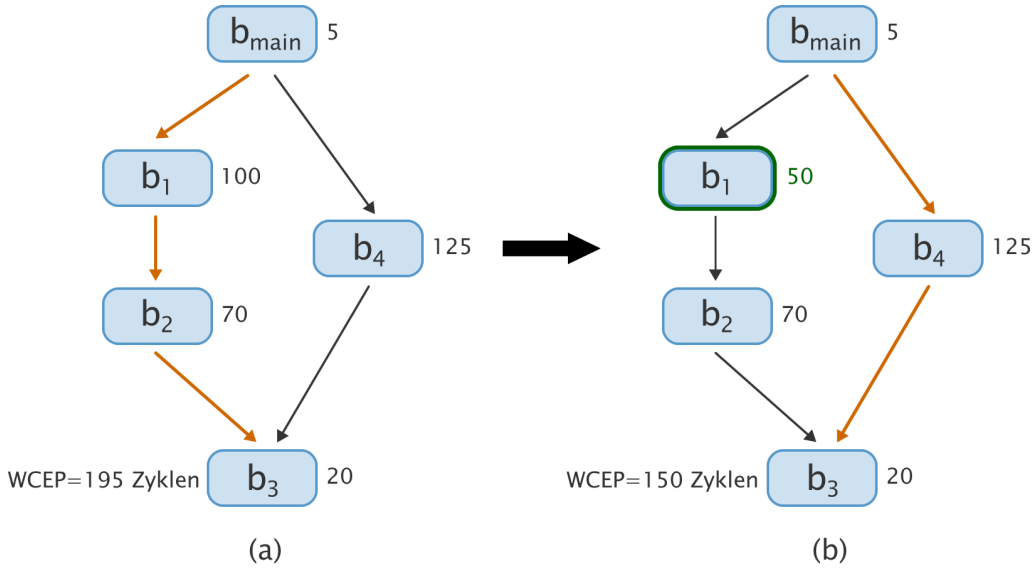

Abbildung 2.2.: Pfadwechsel bei der Optimierung auf dem WCEP [WEE08].

b_{main} und b_1 bis b_3. Hier wurde die Ausführungsdauer des Basisblocks b_1 über eine Optimierung verkürzt. Diese Optimierung hat jedoch zu einem Wechsel des WCEPs geführt, dies wird durch den rechten Graphen verdeutlicht. Der WCEP besteht nun aus den Basisböcken b_{main}, b_4 und b_3, da diese nach der Optimierung zusammen die längste Ausführungsdauer aufweisen. Eine weitere Optimierung des Basisblocks b_1 würde hier keine weitere Auswirkung auf die Reduktion der WCET haben, da dieser Block nun nicht mehr auf dem WCEP liegt. WCET-Optimierungen müssen diese Instabilität des WCEPs beachten, da sonst die Möglichkeit besteht, dass nach einem Pfadwechsel Programmcode optimiert wird, der nicht mehr auf dem WCEP liegt.

Zur Bestimmung des WCEPs können verschiedene Werkzeuge eingesetzt werden. Einige dieser werden in [WEE08] beschrieben. Die Werkzeuge arbeiten dabei mit unterschiedlichen Analyseverfahren. Im Folgenden werden die Unterschiede zwischen den dynamischen und den statischen WCET-Analyseverfahren erläutert.

2.1.2. Dynamische Analyse

Bei der *dynamischen WCET-Analyse* wird der Binärcode des zu untersuchenden Programms oder Teile davon auf der Zielarchitektur ausgeführt. Dabei wird das Programm mehrmals mit unterschiedlichen Eingabedaten ausgeführt. Unter den ermittelten Ausführungszeiten wird die maximale Laufzeit als Abschätzung der WCET verwendet. Der Vorteil dieses Verfahrens besteht darin, dass keine aufwändigen Analysen des Programms oder die Erstellung komplexer Prozessormodelle benötigt werden. Somit ist es in der Praxis sehr leicht und schnell umzusetzen. Jedoch stellt die

Auswahl der Eingabedaten ein Problem dar. Da es in endlicher Zeit nicht möglich ist, das Programm unter allen Eingabedaten zu testen, wird versucht eine repräsentative Menge an Daten zu generieren. Es kann dabei jedoch nicht garantiert werden, dass diese Menge auch die Eingaben enthält, die zur WCET führen. Somit ist die ermittelte WCET nicht sicher und stellt lediglich eine untere Schranke für die reale WCET dar.

Die dynamische WCET-Analyse eignet sich nicht für Systeme mit harten Echtzeitbedingungen. Sie kann jedoch eingesetzt werden, um die WCET, die über statische Analyseverfahren gewonnen wird, zu validieren. Da die dynamische WCET-Analyse keine sicheren oberen Schranken für die reale WCET liefert, wird im Folgenden die statische WCET-Analyse betrachtet.

2.1.3. Statische Analyse

Die *statische WCET-Analyse* ermöglicht im Gegensatz zur dynamischen Analyse die Berechnung von sicheren WCET-Schranken. Hier dient ebenfalls der Binärcode des zu untersuchenden Programms als Eingabe. Das Programm wird dabei jedoch nicht ausgeführt sondern statisch analysiert. Somit ist eine Ermittlung von möglichen Eingabedaten nicht notwendig. Stattdessen wird die Semantik des Programms genutzt, um dessen dynamisches Verhalten vorherzusagen. Für die Analyse werden ein abstraktes Modell der Zielarchitektur, der Kontrollflussgraph des Programms und *Flow Facts* verwendet. Flow Facts sind zusätzliche Informationen über den Kontrollfluss wie z. B. Schleifengrenzen und Rekursionstiefen, die vom Entwickler angegeben werden. Sie werden benötigt, da die Bestimmung der realen WCET unentscheidbar ist. Der Entwickler ist dabei für die Korrektheit der Angaben verantwortlich.

Das Modell der Zielarchitektur dient der Ermittlung des Zeitverhaltens von Prozessor, Pipelines und Speichern, wie z. B. Zugriffszeiten des Caches. Es wird zur Bestimmung der Ausführungszeiten des Programms benötigt. Um dabei eine zu große Überabschätzung der realen WCET zu verhindern, sollte das Modell der Zielarchitektur möglichst genau sein. Aus dem Programmcode bzw. der Binärdatei wird der Kontrollflussgraph generiert. Dieser beinhaltet die Informationen über mögliche Ausführungspfade. Der Graph wird zur Ermittlung der WCET mit den Informationen der Flow Facts angereichert. Es können hier auch Informationen über unerreichbare Pfade eingetragen werden. Der so mit zusätzlichen Informationen angereicherte Graph kann dann zur Bestimmung des WCEPs eingesetzt werden.

Zur Ermittlung des WCEPs und der zugehörigen WCET kann das sogenannte *IPET* Verfahren (Implicit Path Enumeration Technique) [LM95] verwendet werden. Hier wird die *ganzzahlige lineare Programmierung* (engl. Integer Linear Programming, kurz ILP) eingesetzt. Dabei wird jedem Basisblock des Programms eine maximale

Ausführungszeit c_i sowie eine maximale Ausführungshäufigkeit x_i zugeordnet. Über die Maximierung der folgenden Zielfunktion wird dann die WCET bestimmt.

$$WCET = \sum_{i \in B} c_i * x_i \rightarrow max \tag{2.2}$$

B steht hier für die Menge der Basisblöcke. Die Maximierung der Zielfunktion ist dabei noch an bestimmte Bedingungen geknüpft, welche direkt aus dem Kontrollflussgraphen sowie aus den Annotationen des Entwicklers abgeleitet werden können. Neben dem IPET Verfahren existieren noch andere Verfahren zur Bestimmung des WCEPs und seiner WCET wie z.B. das rein *pfadbasierte Verfahren* [SEE01] oder die unter anderem in [CP00] verwendete Analyse des *Syntaxbaums*. Jedoch besitzen diese Verfahren im Vergleich zur IPET einige Nachteile, welche genauer in [WEE08] beschrieben werden. Zudem wird das IPET Verfahren im WCET-Analysewerkzeug aiT verwendet [Abs11], welches in dieser Arbeit Einsatz findet. Daher erfolgt im nächsten Abschnitt eine genauere Betrachtung der WCET-Analyse unter aiT.

2.2. WCET-Analyse unter aiT

Zur Bestimmung der WCET wird im WCC das statische Analysewerkzeug aiT der Firma AbsInt Angewandte Informatik GmbH [Abs11] eingesetzt. AiT führt eine statische WCET-Analyse durch, die zu einer sicheren und präzisen WCET führt. Somit eignet es sich sehr gut für den Einsatz im Bereich der Entwicklung von eingebetteten Systemen mit harten Echtzeitbedingungen. Die Berechnung der WCET ist dabei in mehrere Analyseschritte aufgeteilt [RF08]. Der Aufbau dieser Analyseschritte ist in Abbildung 2.3 dargestellt. Dabei arbeiten sowohl die Werte-, die Cache-, und die Pipeline-Analyse auf dem Konzept der abstrakten Interpretation [CC77]. Die Pfadanalyse nutzt eine Erweiterung des in Abschnitt 2.1.3 beschriebenen IPET Verfahrens.

Als Eingabe der Analyse dient das Binärprogramm sowie eine Datei, die Flow Facts des Entwicklers enthält, wie z. B. Schleifengrenzen und Rekursionstiefen. Im ersten Schritt der Analyse erfolgt eine Rekonstruktion des Kontrollflussgraphen aus dem Binärprogramm. Der Kontrollflussgraph wird dann in eine Zwischendarstellung überführt. Das Format dieser Zwischendarstellung wird als *Control Flow Representation Language* (CRL) bezeichnet. Der im CRL-Format vorliegende Graph dient als Eingabe für die Werte-Analyse.

Die *Werte-Analyse* versucht, den Wertebereich von Registern und damit die Adressbereiche von Speicherzugriffen für jeden Ausführungskontext im Programm zu bestimmen. Ein Basisblock besitzt je nach Aufrufhistorie unterschiedliche Ausführungskontexte, die zur Bestimmung der WCET betrachtet werden müssen. Wird ein Basisblock z. B. innerhalb einer Schleife aufgerufen, so besitzt er je nach Schlei-

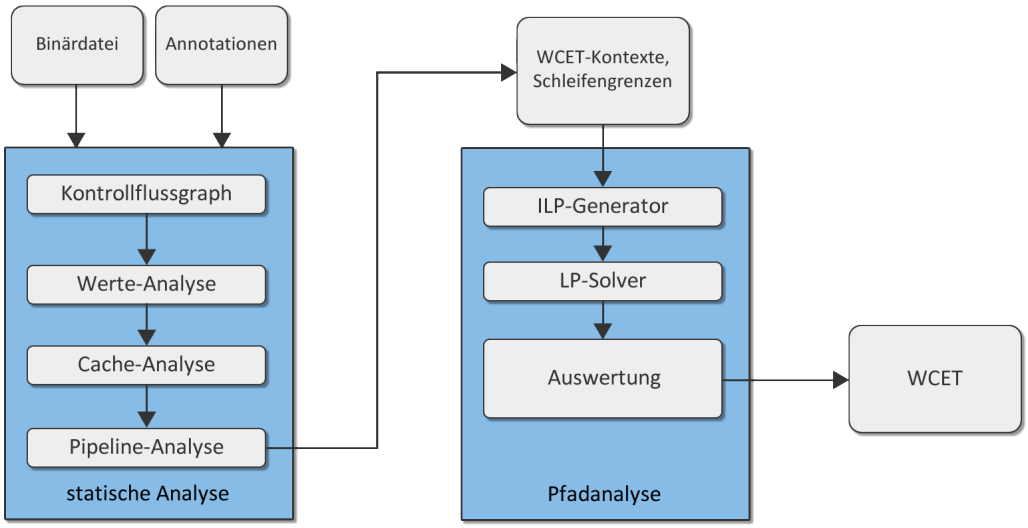

Abbildung 2.3.: Aufbau des WCET-Anaylsewerkzeugs aiT nach [Abs11].

fendurchlauf einen anderen Kontext. Beim ersten Durchlauf befindet sich der Basisblock z. B. noch nicht im Cache und besitzt somit eine andere WCET als in einem zweiten Durchlauf.

Um das zeitliche Verhalten eines Speicherzugriffs beim Einsatz von Caches abschätzen zu können, wird eine *Cache-Analyse* basierend auf [Fer97] durchgeführt. Hier werden die Speicherzugriffe in verschiedene Kategorien eingeteilt, wie z. B. *Always-Hit* oder *Always-Miss*. Es kann jedoch nicht für jeden Speicherzugriff bestimmt werden, ob dieser stets zu einem Cache-Hit oder zu einem Cache-Miss führt, weshalb es auch die Kategorie *Not-Classified* gibt. Auf die Methoden zur Einteilung von Speicherzugriffen in solche Kategorien wird in Abschnitt 2.3 genauer eingegangen.

Das Wissen über das zeitliche Verhalten der Prozessorpipeline spielt ebenfalls eine wichtige Rolle bei der Bestimmung der WCET. Daher erfolgt nach der Cache-Analyse eine *Pipeline-Analyse*. Diese dient dazu, die WCETs einzelner Instruktionen und Basisblöcke in den unterschiedlichen Ausführungskontexten zu bestimmen. Mit Hilfe der ermittelten WCETs für Basisblöcke und Kontrollflusskanten kann die *Pfadanalyse* den WCEP und damit die *globale WCET* des Programms bestimmen. Dabei wird das IPET Verfahren unter der zusätzlichen Betrachtung von Kontexten eingesetzt. Auf Basis der Maximierung der folgenden Zielfunktion wird die WCET bestimmt.

$$\sum_{\forall e,c} C(e,c) * T(e,c) \rightarrow max \tag{2.3}$$

$T(e,c)$ steht hierbei für die ermittelte WCET einer Kontrollflusskante e im Kontext c. $C(e,c)$ beschreibt die Ausführungshäufigkeit der Kante e im Kontext c. Wenn

für jede Kontrollflusskante in jedem Kontext die Ausführungshäufigkeit bekannt ist, kann die globale WCET ermittelt werden. Die Nebenbedingungen ergeben sich hierbei direkt aus dem Kontrollflussgraphen. So muss z. B. die Summe über die Ausführungshäufigkeiten aller eingehenden Kanten eines Basisblocks gleich der Summe der Ausführungshäufigkeiten der ausgehenden Kanten sein. Zusätzliche Bedingungen können über die Annotationen des Entwicklers abgeleitet werden.

Nachdem die Funktionsweise des WCET-Analysewerkzeugs aiT allgemein beschrieben wurde, wird im Folgenden näher auf die Cache-Analyseverfahren eingegangen. Werden Caches in der Zielarchitektur verwendet, führt dies ohne den Einsatz einer Cache-Analyse zu einer großen Überabschätzung der WCET. Für eine präzise Bestimmung der WCET besitzt daher auch aiT eine Cache-Analyse.

2.3. Cache-Analyseverfahren

In diesem Abschnitt wird das Themengebiet der Cache-Analyseverfahren behandelt. Dabei wird zunächst der Aufbau und die Funktionsweise von Caches erläutert (Abschnitt 2.3.1). Für die Vorhersage des Zeitverhaltens von Caches werden statische Analyseverfahren verwendet, daher werden die May- und Must-Analyse von Caches erläutert (Abschnitt 2.3.2).

2.3.1. Cache-Speicher

Um die Lücke zwischen der Geschwindigkeit von Prozessor und Hauptspeicher zu verringern, werden kleine schnelle Speicher wie z. B. Caches eingesetzt. Caches werden in Speicherhierarchien als Puffer zwischen Prozessor und Hauptspeicher eingesetzt. Dabei tragen sie nicht nur zu einer effizienten Laufzeit bei, sondern reduzieren auch den Energieverbrauch des Systems [VM07]. Der Inhalt des Caches wird dabei zur Laufzeit von der Hardware bestimmt. Dies führt beim Einsatz in eingebetteten Systemen aber zu Problemen, da das Zeitverhalten von Caches nur beschränkt vorhersagbar ist. Daher werden Cache-Analysen benötigt. Um zu verstehen, wie diese Analysen arbeiten, ist es notwendig, den Aufbau und die Funktionsweise des Caches zu verstehen.

Eine Speicherhierarchie kann mehrere hintereinander geschaltete Caches beinhalten. Diese unterscheiden sich anhand der Speicherkapazität und der Entfernung zum Prozessor. Je näher ein Cache am Prozessor liegt, desto schneller kann auf diesen zugegriffen werden. Diese Arbeit befasst sich mit der Optimierung des sogenannten *first-level* Caches, welcher direkt auf dem Chip des Prozessors liegt. Viele Prozessoren besitzen einen first-level Cache für Daten und einen für Instruktionen. Dabei hat sowohl der Datencache als auch der Instruktionscache eine eigene Speicherverwaltung. So können Daten- und Instruktionscache separat betrachtet werden. Die

Speicherverwaltung des Caches ist komplett in Hardware realisiert. Sie versucht, unter Ausnutzung der räumlichen und zeitlichen Lokalität immer das Datum vom Hauptspeicher in den Cache zu laden, auf welches der Prozessor als nächstes zugreift. Hierfür können verschiedene Strategien eingesetzt werden. Dabei wird versucht, so wenige Cache-Misses wie möglich zu erzeugen, um die Laufzeiteffizienz zu erhöhen.

Caches können unterschiedlich aufgebaut sein. Grundsätzlich wird ein Cache anhand von drei Parametern charakterisiert. Dies ist die *Kapazität*, die *Blockgröße* bzw. *Zeilengröße* und die *Assoziativität* [Smi82].

- **Kapazität**: Die Kapazität ist die Anzahl der Bytes, die der Cache beinhalten kann.
- **Blockgröße**: Die Blockgröße bzw. Zeilengröße bestimmt die Anzahl der Bytes, die auf einmal vom Hauptspeicher in den Cache übertragen werden. Die maximale Zeilenanzahl im Cache berechnet sich dabei wie folgt:

$$\text{Zeilenanzahl} = \frac{\text{Kapazität}}{\text{Zeilengröße}} \tag{2.4}$$

- **Assoziativität**: Ein Cache wird in sogenannte *Sets* (Mengen) eingeteilt, welche je nach Assoziativität eine bestimmte Anzahl an Speicherplätzen zur Verfügung stellen. Die Assoziativität beschreibt dabei die Anzahl an Speicherplätzen, die in einem Set zur Verfügung stehen, und damit die Plätze in die ein Block eingelagert werden kann. Dabei berechnet sich die Anzahl der Sets wie folgt:

$$\text{Anzahl der Sets} = \frac{\text{Zeilenanzahl}}{\text{Assoziativität}} \tag{2.5}$$

Anhand der Assoziativität werden drei Arten von Caches unterschieden, die Cache-Art bestimmt dabei die Platzierung eines Blocks (siehe Abbildung 2.4). Caches mit einer Assoziativität von *1* werden als *direkt-abgebildete* Caches bezeichnet. Dabei beinhaltet jedes Set nur eine Cachezeile, so dass die Anzahl der Sets gleich der Anzahl der Cachezeilen ist. Caches bei denen ein Block in einen beliebigen Speicherplatz eingelagert werden kann, werden als *vollassoziative Caches* bezeichnet. Hier ist die Assoziativität gleich n, wobei n für die Gesamtanzahl der Cachezeilen steht, somit besteht der Cache hier nur aus einem Set. Ist die Assoziativität größer *1* und kleiner n, so spricht man von *mengenassoziativen* Caches. Hierbei ist die Auswahl der Speicherplätze für einen Block auf die Anzahl der Cachezeilen beschränkt, die ein Set beinhaltet. Die Anzahl der Cachezeilen in einem Set ist dabei gleich der Assoziativität. Abbildung 2.4 veranschaulicht die Unterschiede, die bei der Zuordnung der Speicherplätze für einen Block je nach Cache-Art existieren.

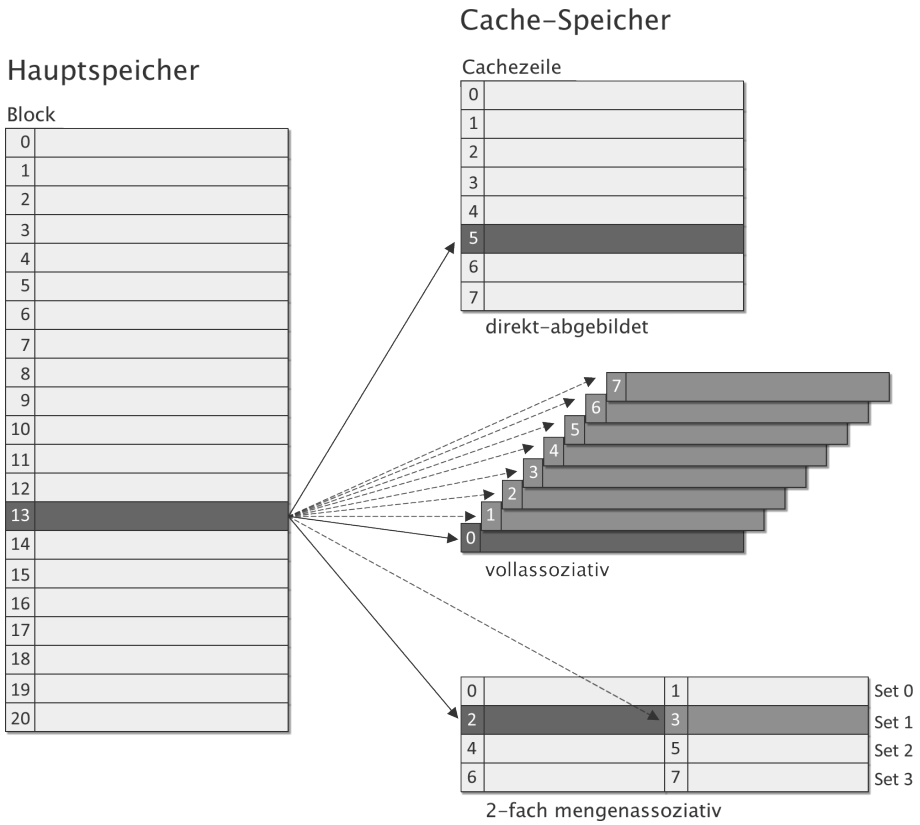

Abbildung 2.4.: Einlagerung eines Blocks bei unterschiedlichen Assoziativitäten [Mue00].

Die Zuordnung eines Blocks auf ein Set kann über die folgende Abbildungsfunktion beschrieben werden:

$$\text{(Blockadresse) mod (Anzahl der Sets)} \tag{2.6}$$

Die Cachezeilen bzw. Sets, auf die ein Block abgebildet werden kann, sind in Abbildung 2.4 grau hinterlegt.

Der Aufbau eines Caches ist anhand eines mengenassoziativen Speichers in Abbildung 2.5 dargestellt. Die Cacheadresse gliedert sich dabei in drei Teile. Sie besteht aus einem *Tag*, einem *Index* und einem *Offset*.

Der Offset, welcher auch als Wort-Adresse bezeichnet wird, wird für die Adressierung innerhalb einer Cachezeile benötigt. Die Zahl der Worte innerhalb einer Zeile beträgt dabei 2^{Offset}. Bei einer Zeilengröße von 32 Bytes benötigt der Offset somit fünf Bits. Der Index dient der Adressierung des Sets, somit ist seine Größe abhängig von der Anzahl der Sets. Der Tagbereich der Adresse wird zur Identifizierung eines Speicherbereichs genutzt. Hiermit wird festgestellt, ob sich das angeforderte Datum

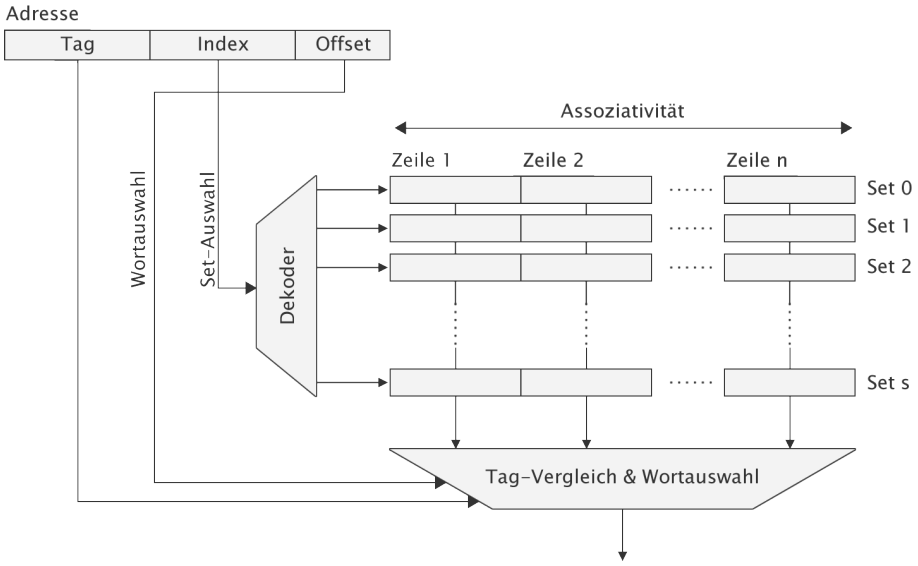

Abbildung 2.5.: Aufbau eines mengenassoziativen Caches nach [HP06].

im Cache befindet. Der Cache besitzt hierfür einen Tag-Speicher. So wird der Tagbereich der Blockadresse mit den Tags des zugehörigen Sets im Cache verglichen, der Aufwand hierfür steigt mit zunehmender Assoziativität.

Die am häufigsten eingesetzte Cache-Art ist die mengenassoziative Variante. Sowohl der direkt-abgebildete als auch der vollassoziative Cache weisen einige Nachteile auf. So ist der Hardwareaufwand bei einem vollassoziativen Cache sehr hoch, da er bzgl. der Tags eine hohe Anzahl an Vergleichern benötigt. Dies spiegelt sich nicht nur im Energieverbrauch wieder, sondern beschränkt auch die Kapazität. So sind vollassoziative Caches nur beschränkt realisierbar. Der direkt-abgebildete Cache besitzt zwar einen geringen Hardwareaufwand, hat jedoch eine schlechte Ausnutzung der Kapazität. Durch die direkte Abbildung eines Blocks aus dem Hauptspeicher auf eine Cachezeile wird bei einem Cache-Miss eine Ersetzung durchgeführt, obwohl noch Kapazität vorhanden ist. Zudem ist die Anzahl möglicher Cache-Misses hier im Vergleich zu mengenassoziativen Caches wesentlich höher, da ein Set nur aus einer Cachezeile besteht und keine alternativen Speicherplätze für einen Block zur Verfügung stehen. Somit stellen mengenassoziative Caches einen guten Kompromiss zwischen Hardwareaufwand und Laufzeiteffizienz bgzl. der Anzahl der Cache-Misses dar.

Die Anzahl möglicher Cache-Misses sinkt mit steigender Assoziativität [HS89]. Ein Cache-Miss kann dabei aus verschiedenen Gründen auftreten. Daher wird zwischen verschiedenen Arten von Cache-Misses unterschieden [HP06]. Ein Cache-Miss, der beim ersten Zugriff auf ein Datum passiert, wird als *Cold Start-Miss* bezeichnet. Zudem gibt es sogenannte Konflikt-Misses. Sie treten auf, wenn mehrere Blöcke aus dem Hauptspeicher auf dasselbe Set bzw. auf dieselbe Cachezeile abgebildet

werden. Da bei vollassoziativen Caches ein Block auf eine beliebige Cachezeile abgebildet werden kann, treten *Konflikt-Misses* nur bei direkt-abgebildeten und mengenassoziativen Caches auf. Bei vollassoziativen Caches findet eine Verdrängung nur aufgrund der begrenzten Kapazität statt.

Bei mengenassoziativen Caches werden sogenannte *Verdrängungsstrategien* eingesetzt, die in Hardware realisiert sind. Tritt ein Cache-Miss auf, bestimmen diese Strategien, welcher Block innerhalb eines Sets durch den neu zu ladenden Block verdrängt wird. Dabei wird die Strategie für jedes Set unabhängig durchgeführt. Das Wissen über die Funktionsweise der Verdrängungsstrategie ist dabei ausschlaggebend für eine präzise Bestimmung der WCET. Die Verdrängungsstrategien unterscheiden sich dabei stark bezüglich ihrer zeitlichen Vorhersagbarkeit. Eine der bekanntesten Strategien ist die *Least Recently Used (LRU)*-Verdrängungsstrategie. Hier wird der Block aus dem Set verdrängt, auf den am längsten nicht mehr zugegriffen wurde. Die LRU-Verdrängungsstrategie besitzt eine sehr gute zeitliche Vorhersagbarkeit [RDB07] und wird ebenfalls von dem statischen WCET-Analysewerkzeug aiT genutzt. Im Folgenden sollen daher zwei bekannte Cache-Analyseverfahren vorgestellt werden, die mit der LRU-Verdrängungsstrategie arbeiten und ebenfalls in aiT integriert sind.

2.3.2. May- und Must-Analyse

Mit Hilfe einer statischen Cache-Analyse kann die Präzision bei der Bestimmung der WCET erheblich erhöht werden. Die Analyse trägt dabei zur Verbesserung der zeitlichen Vorhersagbarkeit von Speicherzugriffen auf den Instruktionscache bei. Dazu werden die Speicherzugriffe in unterschiedliche Kategorien eingeteilt, wie z. B. die Kategorie Always-Hit, für Blöcke die bei jedem Speicherzugriff zu einem Cache-Hit führen, oder analog dazu die Kategorie Always-Miss. Je mehr Speicherzugriffe in solche Kategorien unterteilt werden können, desto präziser kann die WCET ermittelt werden. Zwei häufig verwendete Cache-Analysen sind die sogenannte *May-Analyse* und die *Must-Analyse* nach [FMW97] [FW99].

Über die Must-Analyse wird bestimmt, welche Blöcke sich beim Zugriff, in einem bestimmten Programmpunkt, mit Sicherheit im Cache befinden. Die May-Analyse bestimmt hingegen, welche Blöcke sich möglicherweise im Cache befinden, und kann somit Aussage darüber treffen, welche Blöcke sich in keinem Fall im Cache befinden. Beide Analysen arbeiten mit der LRU-Verdrängungsstrategie und basieren auf dem Konzept der abstrakten Interpretation. Hierbei wird die Semantik des Programms statisch analysiert, um Vorhersagen über das dynamische Verhalten zu ermitteln. Die Funktionsweise der Analysen wird folgend anhand eines mengenassoziativen Caches beschrieben. Ein Set besteht dabei aus einer Menge von Cachezeilen $L = \{l_1, ..., l_n\}$ und beinhaltet eine Menge $B = \{b_1, ..., b_n\}$ von Blöcken. Um Vorhersagen über den Inhalt des Caches zu ermöglichen, ist es notwendig, jeden abstrakten Cachezustand

bzw. Set-Zustand in jedem Programmpunkt zu betrachten. Abhängig vom Kontrollfluss des Programms beinhaltet ein Set dabei je nach Zustand eine andere Menge an Blöcken. Es wird hierbei mit sogenannten *abstrakten Cachezuständen* gearbeitet, die aus abstrakten Set-Zuständen bestehen. Der *Zustand eines Sets* wird mit einer Abbildungsfunktion beschrieben:

$$s : L \mapsto 2^B \tag{2.7}$$

Hier wird jede Cachezeile eines Sets auf eine Menge von Blöcken abgebildet. Die Menge aller möglichen Set-Zustände wird als S bezeichnet. Da die LRU-Verdrängungsstrategie immer den Block verdrängt, auf den in der Vergangenheit am wenigsten häufig zugegriffen wurde, ist ebenfalls über das Alter eines Blockeintrags Buch zu führen. Die Verdrängungsstrategie wird dabei für jedes Set separat angewendet. Ein Set-Zustand $s(l_x) = b_y$ beschreibt somit einen Zustand s, in dem der Block b_y bzgl. der LRU-Strategie ein relatives Alter von x besitzt. Das Alter wird dabei über die Position der Zeile im Set beschrieben, in der sich der Block befindet. Erfolgt ein Blockzugriff, so verändert sich der Zustand des Sets. Diese Zustandsaktualisierung wird durch eine sogenannte *Update-Funktion* beschrieben:

$$U : S \times B \mapsto S \tag{2.8}$$

Somit wird der ursprüngliche Set-Zustand auf einen neuen Zustand abgebildet, in dem der angeforderte Block enthalten ist. Bei einem Cache-Hit liegt dieser Block aber bereits im Set vor, so dass keine Verdrängung stattfindet. Hier ist für den neuen Zustand des Sets nur das Alter der Blöcke zu aktualisieren. Über eine Sequenz von Update-Funktionen kann so das Verhalten eines Sets im Kontrollfluss modelliert werden. Für jeden Basisblock im Kontrollflussgraphen ist dabei die Reihenfolge der Speicherzugriffe bekannt. Um das Verhalten eines Sets zu modellieren, wird die Update-Funktion mit einer Sequenz von Speicherzugriffen erweitert:

$$U(s, \langle b_{x_1}, ..., b_{x_y} \rangle) = U(...(U(s, b_{x_1}))..., b_{x_y}) \tag{2.9}$$

Die Set-Zustände eines Kontrollflusspfades können ermittelt werden, indem die Update-Funktion entlang eines Pfades angewendet wird. Dabei wird im Initialzustand von einem leeren Cache ausgegangen. Um die Zustände eines Sets bei einem Basisblock mit mehr als zwei Vorgängern zu verbinden, wird eine sogenannte *Join-Funktion* eingesetzt. Anhand der Join-Funktion und der Update-Funktion können die Must- und die May-Analyse unterschieden werden.

Bei der Must-Analyse ist die Position eines Blocks im Set eine obere Grenze für sein Alter. Der Zustand des Sets $s(l_x) = \{b_y, ..., b_z\}$ beschreibt somit eine Menge von Blöcken $b_y, ..., b_z$, die nach $n - x$ Speicherzugriffen aus dem Cache verdrängt werden. Die Speicherzugriffe müssen dazu auf andere Blöcke erfolgen, die noch nicht im Set enthalten oder älter als die im Set vorhanden Blöcke sind. Ein Block b_a ist

dabei älter als ein Block b_b, wenn $\exists l_x, l_y : b_a \in s(l_x), b_b \in s(l_y)$ mit $x \geq y$. Die Update-Funktion $U_{must}(s, b_x) = s'$ der Must-Analyse ist dabei wie folgt definiert:

$$s' = \begin{cases} [l_1 & \mapsto \{b_x\}, \\ & l_i \mapsto s(l_{i-1}) \mid i = 2...h-1, \\ & l_h \mapsto s(l_{h-1}) \cup (s(l_h) - \{b_x\}), \\ & l_i \mapsto s(l_i) \mid i = h+1...n]; \text{ wenn } \exists l_h : b_x \in s(l_h) \\ [l_1 & \mapsto \{b_x\}, l_i \mapsto s(l_{i-1}) \text{ für } i = 2...n]; \text{ sonst.} \end{cases} \tag{2.10}$$

Abbildung 2.6 veranschaulicht die Funktionsweise der Update-Funktion. Der angeforderte Block b_s befindet sich bereits im Set. Sein Alter wird dabei auf eins gesetzt, dies bedeutet, dass der Block nun auf Position l_1 im Set liegt. Alle Blöcke, die im Ausgangszustand des Sets jünger als der angeforderte Block sind, altern um eins. So wird b_x also um eine Position im Set nach hinten verschoben. Die Blöcke, die in der gleichen Zeile wie der angeforderte Block b_s stehen oder älter als b_s sind, werden nicht verschoben. Befindet sich der angeforderte Block beim Zugriff jedoch noch nicht im Set, so findet eine Verdrängung statt. Der Block wird dabei in die erste Zeile eingelagert und alle anderen Blöcke im Set altern um eins. Wenn im Ausgangszustand jede Cachezeile des Sets belegt war, so werden die Blöcke mit der höchsten Position verdrängt, da ihr Alter größer als die Anzahl der Cachezeilen im Set ist.

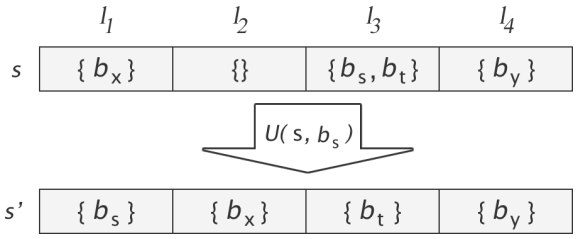

Abbildung 2.6.: Update-Funktion einer Must-Analyse nach [FW99].

Die Join-Funktion $J_{must}(s_1, s_2) = s$ der Must-Analyse bildet eine Schnittmenge über die im Kontrollfluss eingehenden Cache-Zustände eines Basisblocks, wobei:

$$s(l_x) = \{b_i \mid \exists l_a, l_b \text{ mit } b_i \in s_1(l_a), b_i \in s_2(l_b) \text{ und } x = max(a, b)\} \tag{2.11}$$

Nach der Ausführung der Join-Funktion ist ein Block nur im resultierenden Set enthalten, wenn er zuvor in beiden Sets s_1 und s_2 vorhanden war. Hatte er dabei unterschiedliche Alter, so wird das maximale Alter übernommen. Abbildung 2.7 veranschaulicht die Join-Funktion an einem Beispiel.

Die Must-Analyse kann somit Aussage darüber treffen, welche Blöcke bei einem

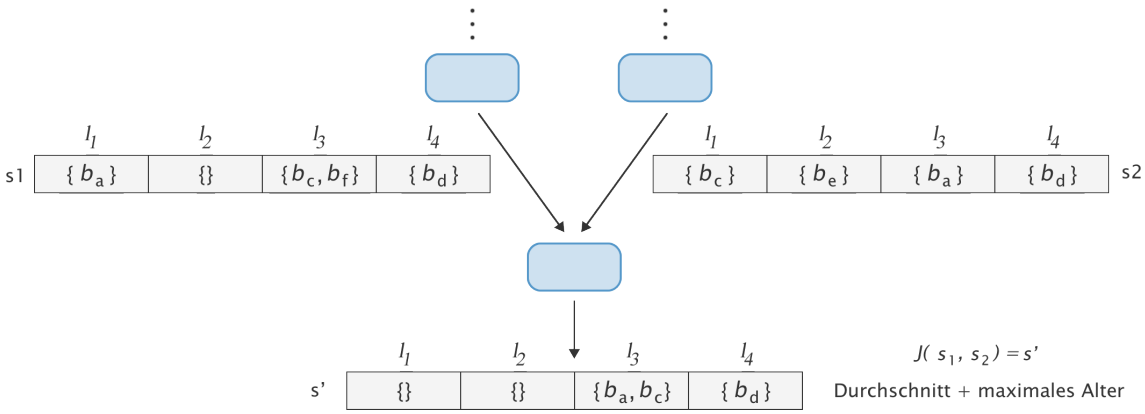

Abbildung 2.7.: Join-Funktion einer Must-Analyse nach [FW99]

Speicherzugriff stets zu einem Cache-Hit führen. Zusätzlich zur Must-Analyse kann auch die May-Analyse durchgeführt werden. Diese identifiziert die Blöcke, die sich möglicherweise im Cache befinden. Hierbei stellen die Positionen im Set-Zustand $s(l_x) = \{b_y, ..., b_z\}$ eine untere Grenze für das Alter der Blöcke dar. Ein Block $b_w \in \{b_y, ..., b_z\}$ wird dabei nach $n - x + 1$ Speicherzugriffen verdrängt. Die Speicherzugriffe müssen dabei auf Blöcke zugreifen die noch nicht im Set vorhanden sind, oder älter bzw. die gleiche Position wie b_w besitzen. Ein Block b_a ist dabei älter oder gleich alt wie ein Block b_b wenn $\exists l_x, l_y : b_a \in s(l_x), b_b \in s(l_y)$ und $x \geq y$. Dabei wird die folgende Update-Funktion $U_{may}(s, b_x) = s'$ verwendet:

$$s' = \begin{cases} [l_1 \mapsto \{b_x\}, \\ \quad l_i \mapsto s(l_{i-1}) \mid i = 2...h, \\ \quad l_{h+1} \mapsto s(l_{h+1}) \cup (s(l_h) - \{b_x\}), \\ \quad l_i \mapsto s(l_i) \mid i = h+2...n]; \text{ wenn } \exists l_h : b_x \in s(l_h) \\ [l_1 \mapsto \{b_x\}, l_i \mapsto s(l_{i-1}) \text{ für } i = 2...n]; \text{ sonst.} \end{cases} \tag{2.12}$$

Abbildung 2.8 veranschaulicht die Funktionsweise der Update-Funktion. Der angeforderte Block b_s befindet sich bereits im Set. Seine Position wird dabei auf eins gesetzt. Alle Blöcke, die jünger als der angeforderte Block sind, altern um eins. So wird b_x also um eine Position im Set nach hinten verschoben. Die Blöcke, die das gleiche Alter wie der angeforderte Block b_s besitzen, altern ebenfalls um eins. Blöcke, deren Position höher ist als b_y, werden jedoch nicht verschoben. Befindet sich der angeforderte Block beim Zugriff nicht im Set, wird seine Position auf eins gesetzt und alle anderen Blöcke im Set altern um eins, wie auch bei der Must-Analyse. Blöcke, die nach dem Update älter als die Anzahl der Cachezeilen im Set sind, werden verdrängt.

Die Join-Funktion $J_{may}(s_1, s_2) = s$ der May-Analyse bildet eine Vereinigung von

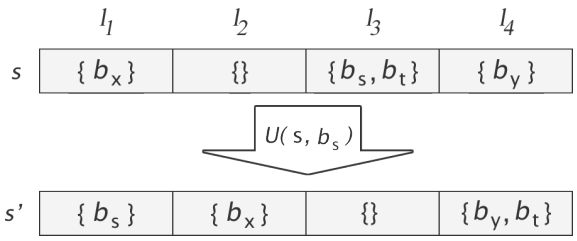

Abbildung 2.8.: Update-Funktion einer May-Analyse nach [FW99].

zwei alternativen Set-Zuständen. Wenn ein Block dabei zwei verschiedene Alter besitzt wird im resultierenden Set das minimale Alter verwendet:

$$\begin{aligned} s(l_x) = \{b_i \mid \exists l_a, l_b \text{ mit } b_i \in s_1(l_a), b_i \in s_2(l_b) \text{ und } x = min(a,b)\} \\ \cup \{b_i \mid b_i \in s_1(l_x) \text{ und } \nexists l_a, \text{ mit } b_i \in s_2(l_a)\} \\ \cup \{b_i \mid b_i \in s_2(l_x) \text{ und } \nexists l_a, \text{ mit } b_i \in s_1(l_a)\} \end{aligned} \tag{2.13}$$

In Abbildung 2.9 ist ein Beispiel für eine solche Join-Funktion dargestellt. Die May-Analyse kann somit Speicherzugriffe identifizieren, die stets zu einem Cache-Miss führen.

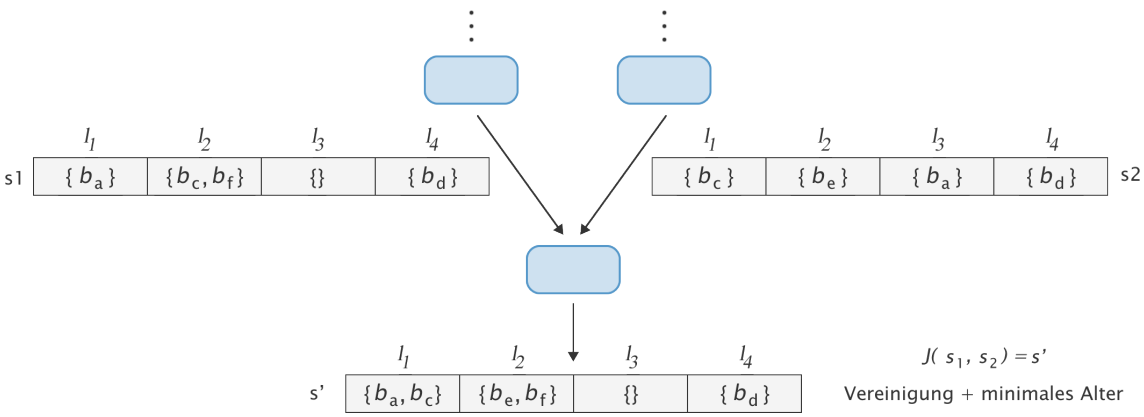

Abbildung 2.9.: Join-Funktion einer May-Analyse nach [FW99].

Die bei der statischen Cache-Analyse ermittelten Informationen werden nicht nur für die Bestimmung der WCET genutzt, sondern können auch für Compiler-Optimierungen, wie z. B. für die Code-Positionierung eingesetzt werden. Im folgenden Abschnitt wird daher das Verfahren der Code-Positionierung genauer erläutert.

2.4. Code-Positionierung

Die Code-Positionierung ist eine bekannte Compiler-Optimierung, die vor allem für Daten und für Instruktionscaches eingesetzt werden kann. Ziel ist es, das Verhalten des Cache-Speichers zu optimieren. Dabei wird die Lokalität von Speicherzugriffen auf Daten bzw. Instruktionen erhöht und so die Laufzeiteffizienz des Systems gesteigert.

Beim Daten-Cache gibt es Optimierungstechniken wie z. B. die Transformation von Schleifen. Dazu gehören das Vertauschen von Schleifen *(Loop Interchange)*, das Verschmelzen *(Loop Fusion)*, das Abrollen *(Loop Unrolling)* oder das Zerlegen von Schleifen *(Loop Tiling)* [Muc97]. Diese Verfahren arbeiten auf der *High Level*-Ebene, sie optimieren das Programm also direkt auf der Hochsprache. Im Folgenden soll jedoch nicht weiter auf diese Art von Optimierungen eingegangen werden, da sich diese Arbeit auf den Bereich der Code-Positionierung für Instruktionscaches konzentriert. Die Code-Positionierungsverfahren für Instruktionscaches arbeiten auf der *Low-Level*-Ebene. Dazu wird eine maschinencodeähnliche Zwischendarstellung des Compilers verwendet. Besitzt ein System keinen Cache, kann die Positionierung von Code trotzdem eingesetzt werden, da sie ebenfalls das Verhalten der Prozessor-Pipeline optimiert. Zusammenfassend können durch das Verfahren der Code-Positionierung für Instruktionen die folgenden Punkte optimiert werden [ACH01]:

- **Reduktion von Misses**: Durch die Code-Positionierung kann die Anzahl der Cache-Misses reduziert werden. Dabei wird die räumliche Lokalität von Speicherobjekten erhöht, auf die in einer hohen zeitlichen Frequenz zugegriffen wird. Außerdem können auch die Misses des *Translation Lookaside Buffer (TLB)* reduziert werden, da durch die Erhöhung der räumlichen Lokalität auch die Wahrscheinlichkeit für Seitenfehler reduziert wird [PH90].
- **Reduktion von Pipeline-Delays**: Sprunganweisungen im Code steigern die Anzahl der Pipeline-Delays. Die Code-Positionierung kann die Anzahl der Sprünge reduzieren, indem Speicherobjekte die eine hohe zeitliche Lokalität aufweisen, jedoch durch einen Sprung getrennt sind, hintereinander angeordnet werden.

Neben der Erhöhung der Laufzeiteffizienz wird auch gleichzeitig der Energieverbrauch reduziert. Dies ist vor allem für eingebettete Systeme von Bedeutung. Die meisten der Code-Positionierungsverfahren führen zur Reduktion der ACET. Im Bereich der eingebetteten Systeme kann diese Optimierung auch zur Reduktion der WCET eingesetzt werden. Hier trägt die Code-Positionierung besonders zu einer besseren Ausnutzung der Systemressourcen bei. Bei einigen Verfahren werden in den Code sogenannte *NOP(No Operation)*-Instruktionen eingefügt. Diese dienen dazu, Speicherobjekte auf eine bestimmte Position zu verschieben. Hierbei kommt es jedoch zu einer Codevergrößerung. Positionierungsverfahren, die in eingebetteten

Systemen eingesetzt werden, sollten jedoch die Vergrößerung des Codes vermeiden, da hier die Speicherkapazität häufig stark beschränkt ist.

Die Positionierung von Code ist auf verschiedenen Ebenen möglich. So können Instruktion, Basisblöcke, Funktionen oder auch ganze Tasks positioniert werden. Diese Arbeit konzentriert sich dabei auf die Positionierung von Basisblöcken und Funktionen zur Reduktion der WCET. Ziel ist es, das Cacheverhalten zu verbessern, indem die Anzahl der Cache-Misses bzw. Konflikt-Misses reduziert wird. Die Reduktion der Konflikt-Misses besitzt bei der Optimierung des Cacheverhaltens ein großes Potenzial. Daher gibt es neben der Code-Positionierung auch noch andere Verfahren, welche die Anzahl der Konflikt-Misses reduzieren [KW03]. So kann unter anderem z. B. die Kapazität einer Cachezeile vergrößert werden, um die räumliche Lokalität zu verbessern. Hierbei erhöht sich jedoch auch der Aufwand bei einem Cache-Miss. Es besteht auch die Möglichkeit, die Assoziativität zu erhöhen. Dies ist aber nur begrenzt möglich, da sich der Hardwareaufwand und die Zugriffszeit somit ebenfalls erhöhen. Ein weitere Technik stellt das sogenannte *Prefetching* dar. Durch spezielle Prefetching-Befehle werden z. B. besonders häufig genutzte Instruktionen vorab in den Cache geladen. Das Prefetching ist jedoch nur dann lohnenswert, wenn ungenutzte Speicherkapazitäten zur Verfügung stehen. Daher stellt die Code-Positionierung eines der besten Verfahren zur Reduktion der Cache-Misses im Bereich der eingebetteten Systeme dar. Dabei werden die Speicherobjekte so angeordnet, dass Konflikte umgangen werden. Instruktionen, auf die mit einer hohen zeitlichen Lokalität zugegriffen wird, werden kontinuierlich im Speicher angeordnet. Dadurch wird eine Überlappung der Bereiche im Cache vermieden und es findet keine gegenseitige Verdrängung mehr statt.

Da die Reduktion der Konflikt-Misses zu einer erheblichen Verbesserung des Cacheverhaltens beiträgt und somit zur Optimierung der Laufzeit (ACET und WCET) und des Energieverbrauches führt, wird in Abbildung 2.10 noch einmal veranschaulicht, wie ein Konflikt-Miss entsteht.

Fortlaufende Blockadressen werden in unterschiedliche Sets abgebildet. Wie in 2.6 definiert, wird, nachdem das letzte Set belegt wurde, beim ersten Set mit der Abbildung fortgefahren. Je nach Kapazität des Caches werden somit mehrere Blöcke des Hauptspeichers auf ein Set abgebildet. Der Indexteil der Blockadresse wiederholt sich nach einer bestimmten Anzahl an Bytes. Zur Berechnung dieses Abstands wird die Anzahl der Sets mit der Zeilengröße multipliziert. Blöcke, die innerhalb dieses Bereiches kontinuierlich angeordnet sind, können sich nicht gegenseitig verdrängen. Bei Blöcken, die jedoch den gleichen Indexteil besitzen, besteht die Möglichkeit der Verdrängung. Jedoch muss kein Konflikt entstehen, wenn die Blöcke in unterschiedliche Zeilen eines Sets eingelagert werden.

Um mit einer Code-Positionierung nicht nur die ACET sondern auch die WCET optimieren zu können, wird das Wissen über den WCEP und die WCET einzelner

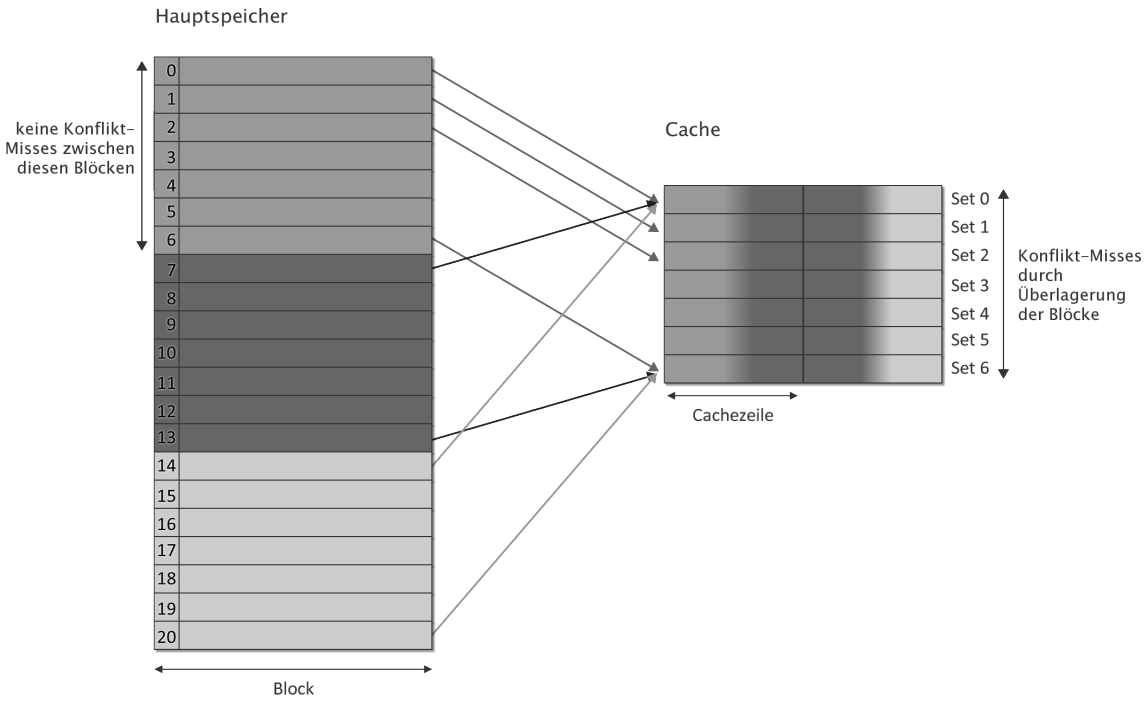

Abbildung 2.10.: Konflikt-Misses.

Speicherobjekte benötigt. Das Positionierungsverfahren, welches in dieser Arbeit entwickelt wird, nutzt dazu den WCET-optimierenden Compiler WCC, in dem das in Abschnitt 2.2 beschriebene Analysewerkzeug aiT integriert ist. Im Folgenden Kapitel wird daher der Aufbau und die Funktionsweise dieses Compilers beschrieben.

3. WCC (WCET-aware C Compiler)

Bei der Softwareentwicklung eingebetteter Systeme werden spezielle Anforderungen gestellt. Im Gegensatz zur Entwicklung von konventionellen Systemen stehen hier Kriterien wie die Codequalität, der Energieverbrauch und die Einhaltung von Zeitschranken im Vordergrund. Wird für den Übersetzungsvorgang ein Standard-Compiler gewählt, so sind die Optimierungsmöglichkeiten meist nur auf die ACET oder die Codegröße beschränkt. Die WCET wird hierbei nicht berücksichtigt. Für die Entwicklung eingebetteter Systeme werden daher Compiler benötigt, welche auf die speziellen Anforderungen der Systeme eingehen. Die Qualität des Codes ist hier von maximalem Interesse, um eine effiziente Nutzung der Systemressourcen zu ermöglichen. Dabei spielt vor allem die Einbeziehung der WCET eine ausschlaggebende Rolle. Im Rahmen der Forschungsarbeiten des Lehrstuhl 12 für eingebettete Systeme der TU Dortmund wurde der WCC (WCET-aware C Compiler), ein WCET optimierender Compiler, entwickelt [FL10]. Mit diesem ANSI C-Compiler ist es möglich, vollautomatisiert WCET-gesteuerte Optimierungen durchzuführen. Als Zielarchitektur wird hierbei die TriCore Plattform des TC1797 bzw. TC1796 der Firma Infineon unterstützt. Durch die Integration des Analysewerkzeugs aiT stehen während des Übersetzungsvorgangs die WCET-Informationen für die Optimierungen zur Verfügung.

Im Folgenden wird zunächst der Aufbau des WCC beschrieben (Abschnitt 3.1). Da die in dieser Arbeit entwickelte Optimierung auf der Low-Level Ebene arbeitet, erfolgt ebenfalls eine Beschreibung der ICD-LLIR, einer Zwischendarstellung des Compilers (Abschnitt 3.2). Abschließend wird auf die vom WCC unterstützte TriCore-Plattform eingegangen (Abschnitt 3.3).

3.1. Aufbau

Der Aufbau des WCC ist in Abbildung 3.1 dargestellt. Anhand der schwarzen Pfeile ist der Ablauf eines klassischen Compilers eingezeichnet. Die grauen Pfeile beschreiben die Erweiterungen des WCC.

Als Compiler-Front-End verwendet der WCC das *ICD-C Framework* [ICD11], welches vom Informatik Centrum Dortmund (ICD) entwickelt wurde. Der *ICD-C Parser* überführt den ANSI-C-Programmcode in eine maschinenunabhängige High-Level

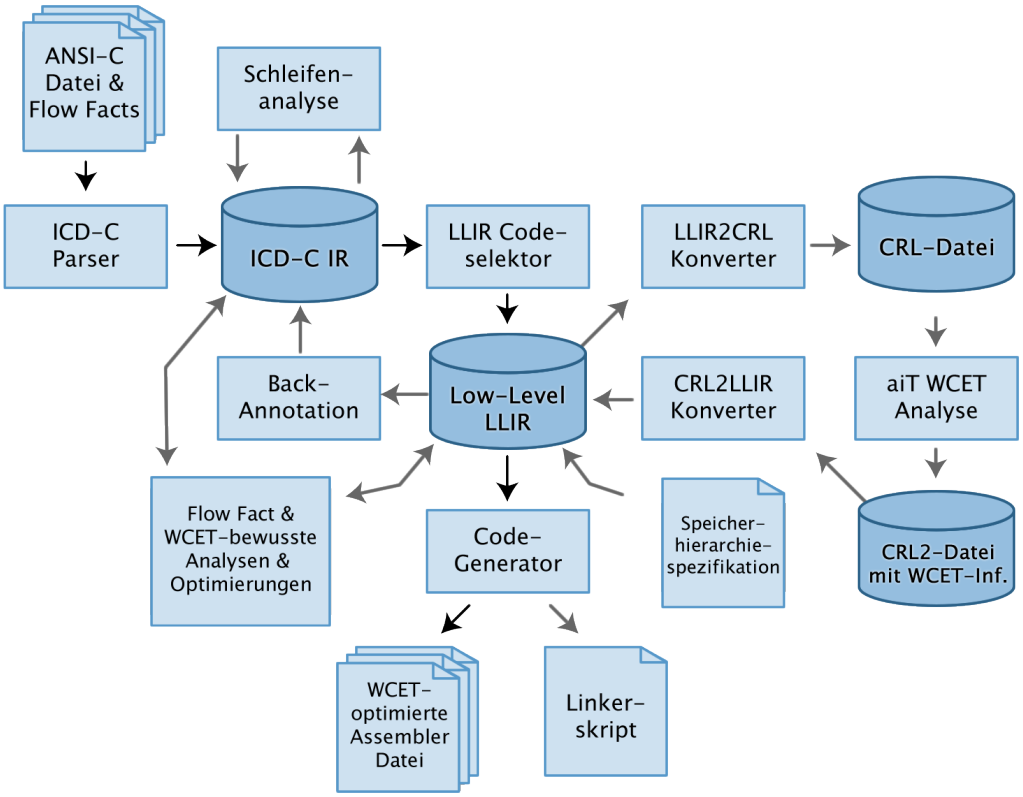

Abbildung 3.1.: Aufbau des WCET-aware C Compilers nach [FL10].

Zwischendarstellung namens *ICD-C IR*. Auf dieser Zwischendarstellung können High-Level-Optimierungen angewendet werden, welche die Eigenschaften der Hochsprache ausnutzten. Über den *Code-Selektor* wird die ICD-C IR in eine Low-Level Zwischendarstellung transformiert, der sogenannten *ICD-LLIR* [LIR11]. Auf dieser LLIR können hardwarenahe Optimierungen angewendet werden. Die in dieser Arbeit vorgestellte Code-Positionierung repräsentiert eine solche hardwarenahe Optimierung, da sie Informationen über die Abbildung der Speicherobjekte auf den Cache benötigt. Die Code-Positionierung wird dabei als eine der letzten Optimierungen nach der Registerallokation ausgeführt. Am Ende des Übersetzungsvorgangs erzeugt der *Code-Generator* aus der ICD-LLIR den Assembler-Code und ein Linker-Skript. Über einen Linker kann dann das Binärprogramm für die TriCore-Plattform erzeugt werden.

Die in Abbildung 3.1 dargestellten grauen Pfeile verweisen auf die Erweiterungen des klassischen Übersetzungsprozesses. Durch diese Erweiterungen werden die WCET Informationen in den Übersetzungsprozesses integriert. Die hierfür benötigten Komponenten werden im folgenden Abschnitt erläutert.

3.1.1. Integration der WCET-Analyse

Durch die Integration des aiT-Analysewerkzeugs werden die WCET Informationen zur Verfügung gestellt. Wie bereits in Abschnitt 2.2 beschrieben, liegen die WCET Informationen von aiT im sogenannten CRL-Format vor. Das Back-End des WCC wird für den Zugriff auf diese Informationen an aiT angebunden. Die ICD-LLIR und die CRL stellen beide Low-Level-Zwischendarstellungen dar. So können diese Formate ineinander überführt werden. Über den *LLIR2CRL-Konverter* wird daher die ICD-LLIR in das CRL-Format transformiert. Da auf der Ebene der ICD-LLIR das Programm bereits in Form eines Kontrollflussgraphen vorliegt, ist es unter aiT nicht mehr notwendig, diesen zu erzeugen. Stattdessen werden die einzelnen LLIR-Basisblöcke des Kontrollflussgraphen in CRL-Basisblöcke überführt. Nach der Analyse von aiT werden die im CRL-Format vorliegenden WCET-Informationen basisblockweise in die ICD-LLIR übertragen. Hierfür wird der *CRL2LLIR-Konverter* eingesetzt. Der Aufruf des aiT-Analysewerkzeugs ist somit für den Entwickler völlig transparent. Nach der aiT-Analyse stehen innerhalb der ICD-LLIR dann folgende Informationen zur Verfügung:

- Die WCET für Basisblöcke, Funktionen und für das Programm
- Die Worst-Case Ausführungshäufigkeit pro Funktion, Basisblock und Kontrollflusskante
- Die Anzahl der Instruktions-Cache-Misses pro Basisblock
- Informationen über unausführbare Kontrollflusskanten

Nach der Beschreibung der Integration der WCET-Informationen in den Übersetzungsprozess des WCC werden im nächsten Abschnitt zusätzliche Erweiterungen des WCC vorgestellt.

3.1.2. Erweiterungen

Der WCC besitzt zusätzlich einen *Flow Fact Manager* über den Annotationen von Schleifengrenzen sowie Rekursionstiefen angegeben werden können. Die Annotationen werden dabei über den Flow Fact Manager direkt in den C Quellcode eingetragen. Da die Flow Facts von der WCET-Analyse benötigt werden, sorgt der Flow Fact Manager zusätzlich für eine Übertragung der Annotationen in das CRL-Format. Zusätzlich zur manuellen Angabe von Annotationen steht eine *Schleifen-Analyse* zur Verfügung. Diese ermittelt automatisch die maximale Iterationsanzahl von Schleifen. Da die WCET-Informationen nach der Analyse von aiT nur vom Compiler Back-End aus erreichbar sind, müssen diese in die High-Level-Ebene übertragen werden. Dieser Vorgang wird über die *Back-Annotation* ermöglicht. Dabei werden die WCET-Informationen aus der ICD-LLIR in die ICD-C IR übertragen und sind damit auch für High-Level Optimierungen zugreifbar. Neben den Informationen aus der Schleifenanalyse und den Flow-Facts können der WCET-Analyse auch zusätzliche Detailinformationen über das *Speicherlayout* von Code und Daten mitgegeben

werden. Hierfür wird eine Speicherspezifikation angelegt, in der z. B. eingetragen wird, ob Code oder Daten in den Cache eingelagert werden. Damit der Linker den Binärcode gemäß der Speicherspezifikation erstellt, generiert der WCC hierfür automatisch ein Linkerskript. Die Informationen über das Speicherlayout werden zusätzlich auch von vielen Low-Level-Optimierungen benötigt.

Nach der Beschreibung der Funktionsweise des WCC wird im folgenden Abschnitt näher auf den Aufbau der ICD-LLIR eingegangen, da die in dieser Arbeit entwickelte Code-Positionierung auf der Low-Level Zwischendarstellung arbeitet.

3.2. ICD-LLIR

Die ICD-LLIR ist eine Klassenbibliothek, welche zur Repräsentation von Assemblercode im Back-End des WCC genutzt wird. Sie ist retargierbar und kann damit über eine Prozessorspezifikation auf verschiedene Prozessoren angepasst werden. Zudem beinhaltet sie eine Datenflussanalyse und eine Kontrollflussanalyse, welche von den hardwarenahen Low-Level Optimierungen des WCC genutzt werden können. Der Aufbau der Klassenhierarchie ist in Abbildung 3.2 dargestellt.

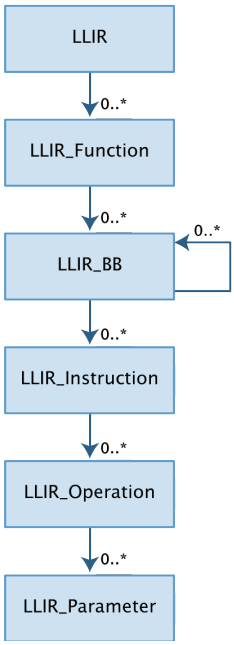

Abbildung 3.2.: Klassenhierarchie der ICD-LLIR nach [LIR11].

LLIR: Die LLIR stellt die Hauptklasse dar und repräsentiert eine komplette Quelldatei des Programms. Sie beinhaltet die Referenzen auf die Objekte der Klasse LLIR_Function.

LLIR_Function: Diese Klasse repräsentiert eine Funktion des Programms. Dabei verwaltet jedes Objekt der Klasse Referenzen auf die zugehörigen Basisblöcke der Klasse LLIR_BB

LLIR_BB: Die Basisblöcke eines Programms werden über die Klasse LLIR_BB repräsentiert. Jedes Objekt dieser Klasse besteht aus einer Ausführungssequenz von Instruktionen der Klasse LLIR_Instruction. Dabei besitzt jeder Basisblock Referenzen auf seine Vorgänger und Nachfolger im Kontrollfluss. So wird der Kontrollflussgraph einer Funktion über die Instanzen der Klasse LLIR_BB dargestellt.

LLIR_Instruction: Diese Klasse repräsentiert Maschineninstruktionen, die aus Instanzen der Klasse LLIR_Operation bestehen. Die Operationen werden parallel ausgeführt. Bei der TriCore-Plattform enthält jedes Objekt der Klasse LLIR_Instruction nur eine Operation.

LLIR_Operation: Die Klasse LLIR_Operation stellt Maschinen-Operationen dar in Form von Assembler-Opcode. Außerdem beinhaltet sie die von den Operationen benötigten Parameter der Klasse LLIR_Parameter.

LLIR_Parameter: Die Parameter einer Maschinen-Operation, wie z.B. Integer-Konstanten, Label und Register, werden über die Klasse LLIR_Parameter dargestellt.

Die aus der WCET-Analyse gewonnenen Informationen werden mit den Instanzen der LLIR-Klassenhierarchie verknüpft. Dies wird über die Klasse LLIR_Objective ermöglicht. So können WCET-Informationen über ein Objekt, welches von dieser Klasse abgeleitet wird, verwaltet werden. Über eine Handler Klasse werden dann die LLIR_Objective Instanzen mit den Klassen der LLIR verbunden. So ist es z. B. möglich, an einem Basisblock die Anzahl der Cache-Misses oder die Ausführungshäufigkeit abzufragen.

Nachdem nun der Aufbau der ICD-C LLIR erläutert wurde, wird im folgenden Abschnitt die vom WCC unterstütze TriCore-Plattform beschrieben.

3.3. TriCore-Plattform

Die in dieser Arbeit entwickelte Code-Positionierung benötigt als Low-Level Optimierung Wissen über das Speicherlayout der Zielarchitektur. Der WCC unterstützt dabei die TriCore Prozessoren TC1797 [Inf09] bzw. TC1796 [Inf07] der Firma Infineon, welche im automotiven Bereich weit verbreitet sind. In dieser Arbeit wird der TC1797 eingesetzt, daher wird im Folgenden ein Überblick über diese Zielarchitektur gegeben.

Der TC1797 ist ein 32-Bit Mikrocontroller, welcher speziell für die Bedürfnisse eingebetteter Systeme entworfen wurde. Er besitzt sowohl Eigenschaften eines *DSPs* (Digitaler Signalprozessor) als auch die eines *RISC* Prozessors. So verfügt er über eine superskalare Prozessorpipeline und einen Registersatz von 32 Allzweckregistern. DSP-typisch werden *MAC (Multiply-Accumulate)-Instruktionen* und *Packed-Operations* unterstützt. Zudem besitzt der TC1797 eine Sättigungsarithmetik und einen heterogenen Registersatz.

Die Prozessorpipeline des TC1797 besteht aus drei Teilen: der *Integer-Pipeline*, der *LS (Load/Store)-Pipeline* und der *Loop-Pipeline*. Die Integer-Pipeline verarbeitet arithmetische Instruktionen und bedingte Sprünge, während die Load/Store-Pipeline für Speicherzugriffe, Adress-Arithmetik, unbedingte Sprünge und Funktionsaufrufe zuständig ist. Schleifen-Instruktionen werden von der Loop-Pipeline verarbeitet, hier können *Zero-Overhead-Loops* realisiert werden. Im Idealfall können alle Pipelines unabhängig voneinander arbeiten. Ein *Stall* in der LS-Pipeline führt jedoch zu einem Stall in der Integer-Pipeline und umgekehrt. Der Befehlssatz besteht aus 32 und 16-Bit-Instruktionen. Instruktionen können meist in einem Zyklus abgearbeitet werden. MAC-Instruktionen oder Gleitkomma-Instruktionen benötigen jedoch mehr als einen Zyklus. Die Verarbeitung von Gleitkomma-Instruktionen erfolgt über eine dedizierte *FPU* (Floating Point Unit), welche direkt mit der CPU verbunden ist. Der heterogene Registersatz besteht aus separaten Adress- und Datenregistern sowie Spezialzweckregistern mit einer Breite von 32 Bit. Dabei können zwei benachbarte Register auch als 64 Bit Register verwendet werden. Der Registersatz ist dabei in einen oberen und einen unteren Kontext eingeteilt. Der obere Kontext wird bei einem Funktionsaufruf automatisch gesichert, somit ist die Sicherung mittels *mov*-Instruktionen überflüssig.

Die Speicherarchitektur des TC1797 besteht aus vielen unterschiedlichen On-Chip Speichern. Für Daten und Code stehen separate Cache bzw. Scratchpad-Speicher zur Verfügung.

Wie in Abbildung 3.3 dargestellt, kann der Prozessor über das sogenannte *PMI* (Program Memory Interface) direkt auf einen 16 KB großen Instruktionscache und einen 24 KB großen Sratchpad-Speicher zugreifen. Über einen lokalen Speicherbus ist das PMI mit dem Hauptspeicher verbunden. Analog zum PMI stehen für die Daten ebenfalls verschiedene Speicher zur Verfügung. Über das *DMI* (Data Memory Interface) kann der Prozessor auf einen 124 KB großen *LDRAM* (Local Data RAM) und einen 4 KB großen Datencache zugreifen. Der Hauptspeicher stellt ebenfalls separate Speicher für Code und Daten zur Verfügung, welche unter anderem aus *Flash*-Speicher bestehen. Der TC1797 bietet zudem die Möglichkeit, einige Speicher nach Bedarf anzupassen. So kann z. B. die Speicherkapazität des Scratchpad-Speichers auf 0 gesetzt werden, wenn dieser nicht benötigt wird. Aber auch die Größe des Instruktionscaches ist konfigurierbar. Der Instruktionscache ist ein mengenassozia-

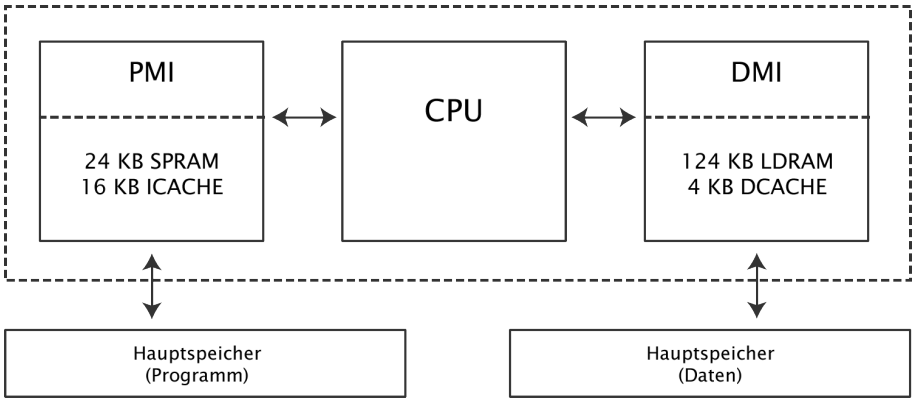

Abbildung 3.3.: Vereinfachte Darstellung der Cache-Architektur nach [Inf04].

tiver Cache, der die LRU-Verdrängungsstrategie verwendet. Mit einer Zeilengröße von 256 Bits können 4 Wörter in eine Cachezeile eingelagert werden. Durch die Konfigurierbarkeit der Speicher stellt die TriCore-Plattform eine flexible Architektur da, die für den Einsatz in eingebetteten Systemen sehr gut geeignet ist.

Nachdem die Grundlagen dieser Arbeit vermittelt und die verwendeten Werkzeuge sowie die Zielarchitektur vorgestellt wurden, soll nun im nächsten Kapitel der erste Teil des Lösungsansatzes für eine Cache-bewusste Code-Positionierung zur Reduktion der WCET vorgestellt werden.

4. WCET-bewusste Cache-Analyse

Ziel dieser Arbeit ist der Entwurf einer Cache-bewussten Code-Positionierung. Durch diese Optimierung wird die Überabschätzung der realen WCET beim Einsatz von Instruktionscaches reduziert. Speicherobjekte wie Basisblöcke und Funktionen werden dabei so positioniert, dass die Anzahl möglicher Konflikt-Misses und somit die WCET reduziert wird. Die Code-Positionierung benötigt hierfür Wissen über das Verhalten des Caches. So gliedert sich der Lösungsansatz dieser Arbeit in zwei Bereiche. Um die Anzahl der Konflikt-Misses zu reduzieren, ist es notwendig zu ermitteln, welche Speicherobjekte im Konflikt stehen. Diese Informationen werden anhand einer Cache-Analyse gewonnen, welche in diesem Kapitel vorgestellt wird. Der zweite Teil des Lösungsansatzes, welcher in Kapitel 5 vorgestellt wird, besteht aus einer Optimierungsstrategie zur Code-Positionierung. Diese arbeitet auf Basis der Informationen der Cache-Analyse. Im Konflikt stehende Speicherobjekte werden dabei kontinuierlich im Hauptspeicher angeordnet. Hierdurch werden die Cache-Konflikte verringert und die WCET reduziert.

Um die Cache-Analyse, welche folgend vorgestellt wird, innerhalb des WCET-Optimierungsprozesses dieser Arbeit einzuordnen, wird zunächst ein Überblick über den Gesamtaufbau der Optimierung gegeben (Abschnitt 4.1). Anschließend erfolgt eine Beschreibung der Modellierung von Cache-Konflikten (Abschnitt 4.2). Informationen über Cache-Konflikte werden hierbei innerhalb eines Graphen verwaltet. Dieser Graph enthält initial mehr Konflikte als real vorhanden. Damit Basisblöcke bzw. Funktionen nicht unnötig positioniert werden, wird die Konfliktmodellierung verfeinert. Dies geschieht anhand verschiedener Analysen wie der Kontrollfluss-Analyse und der May-Analyse (Abschnitt 4.3).

4.1. Aufbau der Optimierung

Der Aufbau der WCET-Optimierung dieser Arbeit ist in Abbildung 4.1 dargestellt. Der Optimierungsprozess besteht dabei aus zwei Bereichen. Über die Cache-Analyse werden *lokale Konfliktgraphen* und ein *globaler Konfliktgraph* modelliert. Die lokalen Konfliktgraphen werden für die Positionierung von Basisblöcken innerhalb einer Funktion benötigt. So wird pro Funktion ein lokaler Konfliktgraph erstellt, welcher die Cache-Konflikte zwischen Basisblöcken modelliert. Der globale Konfliktgraph beinhaltet die Informationen über die Cache-Konflikte zwischen den Funktionen eines Programms. Er wird somit für die Positionierung von Funktionen eingesetzt. Die

Code-Positionierung ordnet dabei zunächst die Basisblöcke mit Hilfe der Informationen aus den lokalen Konfliktgraphen an. Danach erfolgt die Positionierung der Funktionen anhand des globalen Konfliktgraphen.

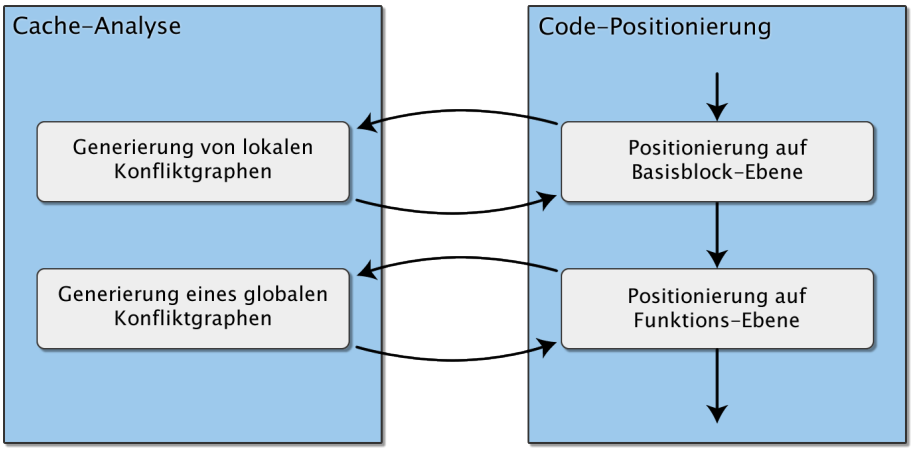

Abbildung 4.1.: Aufbau der WCET-Optimierung.

Die in Abbildung 4.1 dargestellten Pfeile zwischen der Code-Positionierung und der Cache-Analyse verdeutlichen, dass beide Bereiche während des Optimierungsprozesses im Austausch stehen. Nachdem die Cache-Analyse einen lokalen Konfliktgraph erstellt hat, ordnet die Code-Positionierung die im Konflikt stehenden Basisblöcke kontinuierlich im Hauptspeicher an. Da Basisblöcke und auch Funktionen, die sich in ihren Indexadressen überlagern, auf dieselben Sets im Cache abgebildet werden, können sie sich gegenseitig aus dem Cache verdrängen. Durch eine kontinuierliche Anordnung erhalten die Speicherobjekte, je nach Größe, unterschiedliche Indexadressen, so dass Konflikte aufgelöst werden können. Dabei wird nach jeder erfolgreichen Positionierung ein neuer lokaler Konfliktgraph aufgebaut. Dies ist notwendig, da eine Neuanordnung zu einer Änderung des Speicherlayouts führt. Anhand des neu erzeugten Konfliktgraphen erfolgt die nächste Positionierung. Nach der Abarbeitung aller lokalen Konfliktgraphen, folgt die Positionierung der Funktionen. Hierfür wird der globale Konfliktgraph erzeugt. Der Arbeitsablauf ist hier ähnlich zur Positionierung auf Basisblock-Ebene. Die Vorgehensweise der Code-Positionierung wird später in Kapitel 5 im Detail erläutert. Folgend soll die Cache-Analyse näher betrachtet werden.

Da die entworfene Optimierung WCET-bewusst arbeitet, erfolgen während der Cache-Analyse bzw. während der Code-Positionierung Aufrufe an das Analysewerkzeug aiT. Die Cache-Analyse integriert die daraus gewonnenen WCET-Informationen in die erzeugten Konfliktgraphen. Die Code-Positionierung hat so die Möglichkeit, zuerst die Basisblöcke bzw. Funktionen abzuarbeiten, die bzgl. der WCET das größte Optimierungspotenzial besitzen. Bevor die Konfliktgraphen als Basis der Code-

Positionierung verwendet werden können, werden diese in mehreren Arbeitsschritten innerhalb der Cache-Analyse erzeugt. Diese Analyse-Schritte sind in Abbildung 4.2 dargestellt.

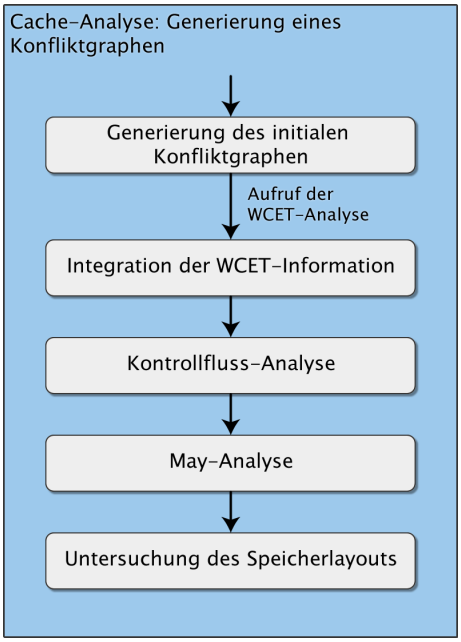

Abbildung 4.2.: Aufbau der Cache-Analyse.

Anhand des Speicherlayouts wird sowohl für die Erzeugung des globalen als auch für die der lokalen Konfliktgraphen jeweils zuerst ein *initialer Konfliktgraph* aufgebaut. Dieser beinhaltet Kanten zwischen Speicherobjekten, deren Sets sich im Cache überlagern. Nach dem Aufruf der WCET-Analyse kann ein initialer Graph in einem zweiten Schritt anhand der WCET-Informationen *gewichtet* werden. Hierbei werden unter anderem Informationen über Cache-Misses und maximale Ausführungshäufigkeiten in den Graphen integriert (siehe Abschnitt 4.2). Außerdem werden Kanten zwischen Speicherobjekten, die nicht auf dem *WCEP* liegen, aus einem Konfliktgraphen entfernt, denn eine Code-Positionierung außerhalb des WCEPs trägt nicht zur Optimierung bei. Da nach der WCET-Analyse Informationen über die Anzahl der Cache-Misses vorliegen, können zudem Kanten zwischen Speicherobjekten entfernt werden, in denen definitiv keine Cache-Misses auftreten. Über die nachfolgende *Kontrollfluss-* und *May-Analyse* und über die Untersuchung des *Speicherlayouts* wird ein Konfliktgraph weiter verfeinert. Dies ist notwendig, da ein Graph initial Kanten zwischen Speicherobjekten beinhaltet, die nicht zwingend im Konflikt stehen. Solche Konflikte werden als unreal bezeichnet und durch Objekte hervorgerufen, deren Sets sich zwar überlagern, die jedoch nicht im Konflikt stehen, da sie z. B. in unterschiedliche Cachezeilen eines Sets eingelagert werden. Zudem beinhaltet der Graph Konfliktkanten die kein Optimierungspotenzial bieten, da die zugehörigen Speicher-

objekte bereits vor der Code-Positionierung kontinuierlich im Hauptspeicher angeordnet sind. Solche Konflikte werden bei der Untersuchung des Speicherlayouts vor Beginn der Code-Positionierung entfernt.

Die Modellierung der Cache-Konflikte und damit der Aufbau eines Konfliktgraphen wird nun im folgenden Abschnitt genauer erläutert.

4.2. Modellierung der Cache-Konflikte

Mit Hilfe eines Konfliktgraphen können die Cache-Konflikte zwischen Funktionen und Basisblöcken der ICD-LLIR modelliert werden. In diesem Abschnitt wird zunächst der Aufbau des initialen Konfliktgraphen vorgestellt (Abschnitt 4.2.1). Danach erfolgt eine Beschreibung der Integration der WCET-Informationen über die Gewichtung des Graphen (Abschnitt 4.2.2).

4.2.1. Aufbau des Konfliktgraphen

Ein in der Cache-Analyse generierter Konfliktgraph baut auf einem Graphen aus [VWM04] auf. Hier wurde ein Konfliktgraph verwendet, um zu ermitteln, welche Speicherobjekte bei einer Verlagerung in einen Scratchpad-Speicher zur größten Energieeinsparung führen. Denn ein Speicherobjekt, welches eine hohe Anzahl an Konflikt-Misses aufweist, trägt auch zu einem höheren Energieverbrauch bei. Durch die Verlagerung in einen Scratchpad-Speicher wurden hier die Konflikte aufgelöst und so der Energieverbrauch reduziert (siehe Abschnitt 1.3). Die vorliegende Arbeit konzentriert sich jedoch auf den Einsatz von Caches in eingebetteten Systemen ohne Verwendung von Scratchpad-Speichern. Statt die Speicherobjekte aus dem Cache zu verlagern, werden diese innerhalb des Hauptspeichers neu angeordnet. Da der Konfliktgraph aus [VWM04] jedoch auch das Cacheverhalten modelliert, kann dieser unter einigen Anpassungen in dieser Arbeit verwendet werden. Der in dieser Arbeit für die Positionierung von Funktionen und Basisblöcken verwendete Konfliktgraph ist wie folgt definiert:

Definition 4.1 *Ein Konfliktgraph $G = (V, E, m, w)$ ist ein gerichteter Graph, dessen Kanten und Knoten gewichtet sind.*

$v_i \in V$: Ein Knoten v_i repräsentiert ein Speicherobjekt des Programms. Die Knotenmenge $V = \{v_1, ..., v_n\}$ beinhaltet einen Knoten v_i, wenn sein zugehöriges Speicherobjekt mit einem anderen Speicherobjekt v_j im Konflikt steht.

$e_{ij} \in E$: Die Kantenmenge E besteht aus gerichteten Kanten e_{ij} von Knoten v_i nach v_j, wenn eine Cachezeile, in die ein Teil von v_i eingelagert ist, durch eine Cachezeile von v_j ersetzt werden kann.

m_i : Das Knotengewicht m_i eines Knotens v_i stellt die Ausführungshäufigkeit dar, wenn das zugehörige Speicherobjekt ein Basisblock ist. Bei einer Funktion steht m_i für die Aufrufhäufigkeit.

w_{ij} : Das Kantengewicht w_{ij} einer Kante e_{ij} beschreibt die Anzahl möglicher Cache-Misses, die beim Zugriff auf v_i durch v_j verursacht werden, da v_j Cachezeilen von v_i verdrängt.

In Abbildung 4.3 ist der Aufbau des Konfliktgraphen noch einmal graphisch dargestellt. Eine Kante wie z. B. e_{ab} ist dabei auf den Knoten v_b gerichtet, wenn das zugehörige Speicherobjekt für die Cache-Misses des Quellknotens v_a verantwortlich ist.

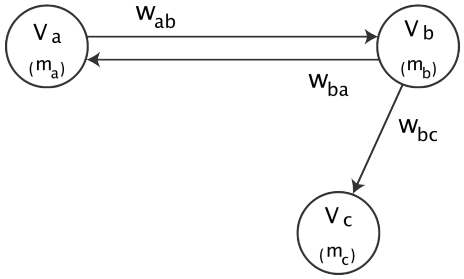

Abbildung 4.3.: Konfliktgraph nach Definition 4.1.

Die Knotenmenge V repräsentiert die Speicherobjekte. Je nach lokalem oder globalem Konfliktgraph, handelt es sich dabei um die Basisblöcke oder die Funktionen eines Programms. Die Kantenmenge E wird über die Adressräume der Speicherobjekte ermittelt. Dabei werden für jedes Paar von Speicherobjekten, deren belegte Sets bzw. Cachezeilen sich überlappen, zwei gerichtete Kanten erzeugt. Das Gewicht m_i eines Knotens v_i und das Gewicht w_{ij} einer Kante e_{ij} wird über eine WCET-Analyse bestimmt. Dabei ist zu beachten, dass das Kantengewicht w_{ij} in der oben stehenden Definition nur vereinfacht dargestellt ist. Die Anzahl der Cache-Misses, die über eine Verdrängung von v_i durch v_j verursacht wird, kann nicht über die WCET-Analyse von aiT bestimmt werden. Hier wird nur die Information über die Gesamtanzahl der Cache-Misses von v_i zur Verfügung gestellt. Das Speicherobjekt v_i kann jedoch auch neben v_j von anderen Speicherobjekten verdrängt werden. Aus diesem und anderen Gründen, die in Abschnitt 4.2.2 erläutert werden, wird die zur Verfügung stehende Gesamtanzahl der Cache-Misses von v_i nicht direkt als Kantengewicht verwendet, sondern mit zusätzlichen WCET-Informationen verrechnet. Eine genaue Beschreibung für die Zuordnung der WCET-Informationen erfolgt in Abschnitt 4.2.2. Folgend wird zunächst auf die Ermittlung der initialen Kantenmenge E eingegangen.

Um die Kantenmenge E zu ermitteln muss bekannt sein, welche Speicherobjekte sich im Cache überlappen und damit im Konflikt stehen. Hierfür wird zunächst eine

Abbildungsfunktion definiert. Diese ordnet jedem Speicherobjekt Sets im Cache zu. Da die ICD-LLIR Informationen über das Speicherlayout beinhaltet, kann der für die Abbildung benötigte Adressraum von Basisblöcken bzw. Funktionen bestimmt werden. Die Abbildungsfunktion definiert sich dabei entsprechend der Gleichungen (2.4) und (2.5) wie folgt:

$$Map(Addr) = (Addr \gg Off) \bmod \frac{K}{A * Z} \tag{4.1}$$

Addr steht dabei stellvertretend für eine der Hauptspeicheradressen des Speicherobjekts. Mit Hilfe des modulo-Operators der Funktion *Map(Addr)* kann über die Start- und Endadresse eines Speicherobjektes der Bereich der belegten Sets ermittelt und mit anderen Speicherobjekten verglichen werden. Abhängig von der Größe des Speicherobjektes kann dieses mehrere Sets belegen und damit einen hohe Anzahl an Konflikten auslösen. Die Gesamtanzahl der Sets des Caches ergibt sich aus der Division der Cache-Kapazität K mit dem Produkt aus Assoziativität A und Zeilengröße Z. Wie in Abschnitt 2.3.1 beschrieben, wird für die Adressierung eines Sets lediglich der vordere Teil der Hauptspeicheradressen benötigt. Der hintere Teil dient der Adressierung eines Wortes innerhalb einer Cachezeile und wird als Offset *Off* bezeichnet. Um die Speicherobjekte auf die Sets abzubilden, wird daher durch Rechts-Schieben der Offset-Teil der Hauptspeicheradresse *Addr* entfernt.

Zur Generierung des initialen Konfliktgraphen wird die beschriebene Abbildungsfunktion *Map(Addr)* verwendet. Abbildung 4.4 veranschaulicht dies anhand eines 2-fach mengenassoziativen Caches.

Der Hauptspeicher beinhaltet hier unter anderem drei Basisblöcke A, B und C, deren Startadressen an der linken Seite des Hauptspeichers eingetragen sind. Die nebenstehenden eingeklammerten Adressen sind die Binärdarstellungen der Hauptspeicheradressen. Der Offset-Teil, welcher für die Adressierung der Sets entfernt wird, ist hier grau hinterlegt. Die Kapazität des Caches beträgt 256 Byte. Eine Cachezeile ist hierbei 32 Byte groß. Werden die Kapazität, die Zeilengröße und die Assoziativität in den hinteren Teil der Abbildungsfunktion *Map(Addr)* eingesetzt, so ergibt sich die Anzahl der Sets von 4. Für die Adressierung eines Sets werden die niederwertigen 5 Bits der 32 Bit Adresse entfernt. Der Basisblock A wird somit über die Funktion *Map(Addr)* auf die ersten 3 Sets im Cache abgebildet. Über den Vergleich der belegten Sets mit denen der anderen Basisblöcke kann die Überlappung der Bereiche im Cache bestimmt werden. Basisblock B wird auf die Sets 1 und 2 abgebildet, seine Sets überlappen sich daher mit zwei der Sets von Basisblock A. Basisblock C wird auf die unteren beiden Sets abgebildet. Die Basisblöcke A, B und C überlappen sich somit alle im Set 2. Über den Vergleich der belegten Sets entsteht der in Abbildung 4.4 unten rechts dargestellte Konfliktgraph.

Für ein Paar von Speicherobjekten, deren Sets sich im Cache überlagern, werden

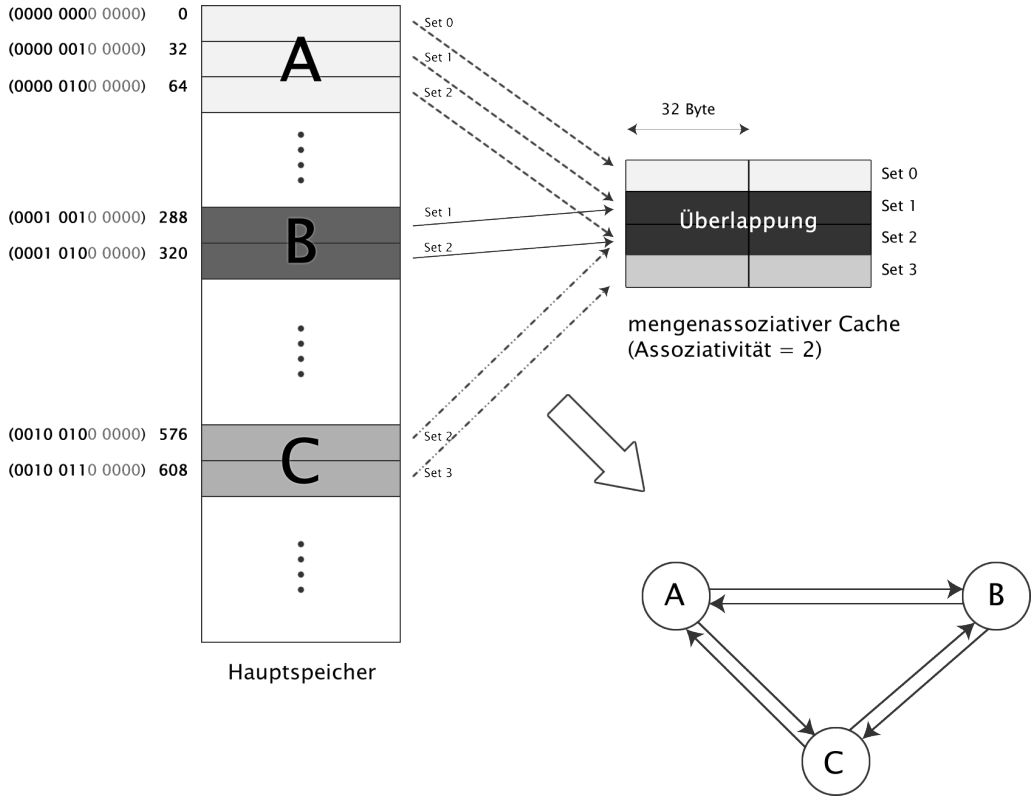

Abbildung 4.4.: Aufbau eines initialen Konfliktgraphen.

hier zwei gerichtete Kanten angelegt. Zu diesem Zeitpunkt ist jedoch noch nicht bekannt, ob wirklich eine gegenseitige Verdrängung zwischen A, B und C stattfindet. Um diese Informationen zu erhalten, sind weitere Analyse-Schritte notwendig, die im Abschnitt 4.3 beschrieben werden. Die Gewichte der erzeugten Kanten und auch der Knoten sind zunächst mit 0 belegt, da vor der Generierung eines initialen Konfliktgraphen noch keine WCET-Analyse stattgefunden hat. Wie die WCET-Informationen nach der WCET-Analyse in den Graphen integriert werden, wird im folgenden Abschnitt beschrieben.

4.2.2. Integration der WCET-Informationen

Um die Informationen für die Gewichtung des globalen und der lokalen Konfliktgraphen zu erhalten, erfolgt eine WCET-Analyse anhand von aiT. Dabei werden unter anderem folgende WCET-Informationen in die Graphen integriert:

- **Worst Case Execution Count** (WCEC): Die WCEC ist die Worst Case Ausführungshäufigkeit pro Basisblock.
- **Worst Case Call Frequency** (WCCF): Die WCCF ist die Worst Case Aufrufhäufigkeit pro Funktion.

- **Cache-Misses * WCEC**: Dies ist die Anzahl der Cache-Misses, die über alle Kontexte eines Basisblocks ermittelt wird, multipliziert mit der Ausführungshäufikeit des Basisblocks.

Die WCEC eines Basisblocks wird als Knotengewicht im lokalen Konfliktgraphen verwendet. Im globalen Konfliktgraphen, dessen Knoten Funktionen repräsentieren, wird die WCCF als Knotengewicht eingesetzt. Mit Hilfe dieser Gewichte kann bestimmt werden, welche Speicherobjekte auf dem WCEP liegen. Für die Gewichtung der Kanten wird unter anderem das oben beschriebene Produkt aus den Cache-Misses und der WCEC verwendet. Dieses wird folgend als *MissesSum* bezeichnet. Dabei stellt die MissesSum einen wesentlichen Bestandteil der Kantengewichte dar und ist später ausschlaggebend für eine effiziente Code-Positionierung. Ein Kantengewicht beinhaltet jedoch noch weitere Komponenten. Wie sich die Kanten- und Knotengewichte im Detail aus den WCET-Informationen zusammensetzten, wird später noch genauer erläutert. Nun soll zunächst auf die Bestandteile der MissesSum eingegangen werden, um zu verstehen, warum zusätzliche Komponenten neben der MissesSum für die Kantengewichtung benötigt werden.

Die MissesSum besteht, wie beschrieben, aus dem Produkt der Cache-Misses und der WCEC eines Basisblocks, summiert über alle seine Kontexte. Der in der ICD-LLIR vorliegende Kontrollflussgraph einer Funktion, welcher aus den Basisblöcken der Funktion besteht, enthält pro ausgehender Kontrollflusskante eines Basisblocks mehrere Kontexte. Ein Kontext beschreibt dabei eine mögliche Aufrufhistorie des Basisblocks und enthält, je nach Aufrufhistorie, eine unterschiedliche Anzahl an Cache-Misses und WCECs. Wird ein Basisblock z. B. innerhalb einer Schleife aufgerufen, so besitzt er je nach Schleifendurchlauf einen anderen Kontext. Beim ersten Durchlauf befindet sich der Basisblock z. B. noch nicht im Cache und besitzt somit in diesem Kontext eine höhere Anzahl an Cache-Misses als bei einem zweiten Durchlauf. Ein weiteres Beispiel stellt ein Basisblock dar, der mit einem bedingten Sprung endet. Durch den bedingten Sprung besitzt der Basisblock im Kontrollfluss zwei mögliche Nachfolger und damit zwei ausgehende Kanten. Die Anzahl der Cache-Misses und der WCEC ist dabei je nachdem, ob der Sprungbefehl ausgeführt wird oder nicht unterschiedlich. Für jede der ausgehenden Kanten gibt es wiederum unterschiedliche Kontexte, aus denen sich die Gesamtanzahl der Cache-Misses und die WCEC einer Kante zusammensetzt. Somit bildet sich die MissesSum über alle Kontexte der ausgehenden Kanten eines Blocks. Ein konkretes Beispiel für die Berechnung der MissesSum ist in Abbildung 4.5 dargestellt.

Der obere Teil der Abbildung stellt einen Ausschnitt aus einem ICD-LLIR Kontrollflussgraphen dar. A, B und C sind somit Basisblöcke. Der Basisblock A besitzt zwei ausgehende Kanten, da er wie im oben beschriebenen Beispiel mit einem bedingten Sprung endet. Wird die Sprunganweisung ausgeführt, so entstehen an der Kante von A nach B bei der Ausführung von A 20 Cache-Misses und eine WCEC von 0. Wird die Sprunganweisung nicht ausgeführt, entstehen hingegen nur 6 Cache-Misses und

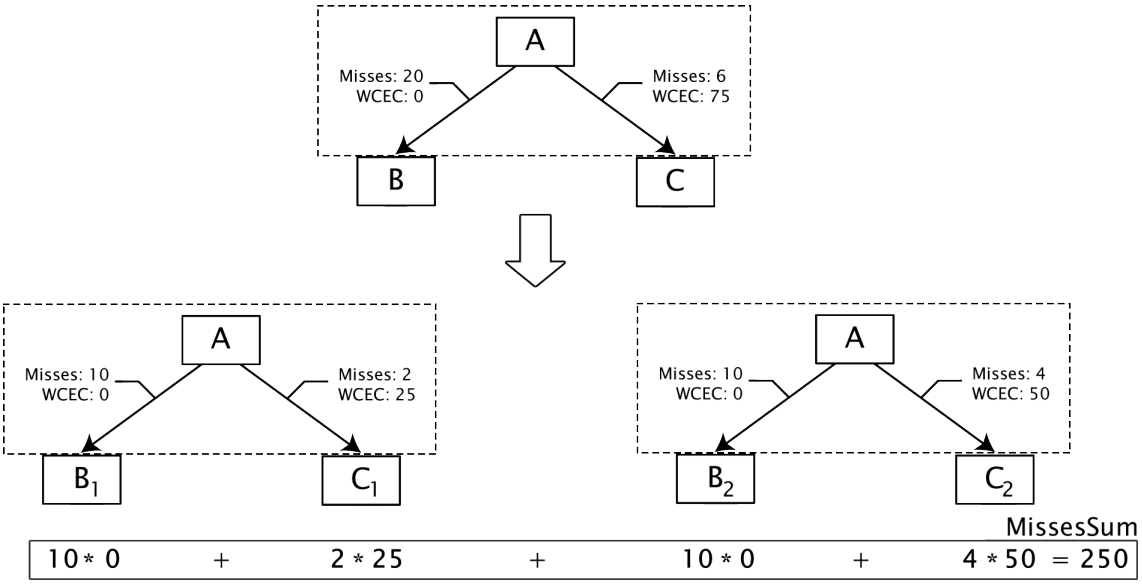

Abbildung 4.5.: Berechnung der *MissesSum* eines Basisblocks.

eine WCEC von 75. Diese Werte ergeben sich aus den einzelnen Kontexten einer Kante.

Im unteren Teil der Abbildung 4.5 sind die Kontexte der Kanten aufgeführt. So besitzt die Kante von A nach C zwei Kontexte. Diese sind im linken Teil durch die Kante von A nach C_1 und im rechten Teil durch die Kante von A nach C_2 abgebildet. Gleiches gilt für die Kante von A nach B. Hier sind pro Kontext die Cache-Misses und die WCEC eingetragen. Die Kontexte von A nach B_1 und nach B_2 besitzen eine WCEC von 0. Dies bedeutet, dass diese Kontexte nicht auf dem WCEP liegen und damit der Kontrollfluss von A nach B im Worst Case nicht ausgeführt wird. Da bei der MissesSum die Anzahl der Cache-Misses eines Kontextes mit seiner WCEC multipliziert wird, werden nur die Cache-Misses berücksichtigt, die auf dem WCEP liegen. Zusätzlich sind die Cache-Misses so über die WCEC gewichtet. Ein Basisblock, der eine niedrige Anzahl an Cache-Misses besitzt jedoch sehr häufig ausgeführt wird, kann so ebenfalls berücksichtigt werden. Das Ergebnis der MissesSum über die vier Kontexte von Basisblock A beträgt 250.

Die MissesSum gibt somit Auskunft darüber, wie viele Cache-Misses unter Berücksichtigung der WCEC im Worst Case bei der Ausführung eines Basisblocks auftreten. Auch wenn diese Informationen über die ausgehenden Kanten eines Basisblocks gewonnen werden, kann der Wert der Cache-Misses nicht direkt auf die Kanten im Konfliktgraphen abgebildet werden. Dies liegt daran, dass der Konfliktgraph Kanten zwischen Objekten beinhaltet, die nicht zwingend im Kontrollflussgraphen enthalten sein müssen. So sind z. B. Speicherobjekte, die im Kontrollflussgraphen nur indirekt über eine Pfad verbunden sind, im Konfliktgraph direkt über eine Kante verbunden.

Daher kann mit der MissesSum keine Aussage darüber getroffen werden, wie die Cache-Misses genau zustande kommen und welche Basisblöcke für die Cache-Misses verantwortlich sind. Um eine effiziente Code-Positionierung zu ermöglichen, müssen daher neben der *MissesSum* des Quellknotens einer Konfliktkante noch weitere Werte in die Berechnung des Kantengewichts einfließen.

Ein Speicherobjekt, welches durch v_i im Konfliktgraphen repräsentiert wird, kann mit mehreren anderen Speicherobjekten $v_j, ..., v_k$ im Konflikt stehen, wenn sich die zugehörigen Sets im Cache überlappen. Die Speicherobjekte $v_j, ..., v_k$ können zu verschiedenen Anteilen zur Gesamtanzahl der Cache-Misses von v_i beitragen. Je mehr ein Speicherobjekt v_j zur Gesamtanzahl der Cache-Misses von v_i beiträgt, desto stärker ist auch der Konflikt. Die Information über diese Anteile kann jedoch, wie beschrieben, nicht aus der MissesSum gewonnen werden. Um die Konflikte zwischen den Speicherobjekten trotzdem gewichten zu können, wird daher die Anzahl der gemeinsam belegten *Sets* von v_i und v_j zusätzlich in das Kantengewicht integriert. Je größer die Überlappung der Sets von v_i und v_j ist, desto höher ist somit auch das Kantengewicht w_{ij}. Beim lokalen Konfliktgraphen wird die reine WCEC des Quellknotens einer Kante ebenfalls in das Kantengewicht aufgenommen. Dies ist notwendig, da die *MissesSum* eines Basisblocks nur die WCECs der Kontexte beinhaltet, in denen Cache-Misses auftreten. Das Kantengewicht in einem lokalen Konfliktgraphen berechnet sich damit wie folgt:

$$w_{ij}^{L} = WCEC_i * MissesSum_i * Sets_{ij} \tag{4.2}$$

Dabei stellt $WCEC_i$ die Ausführungshäufigkeit des Quellknoten v_i dar. $MissesSum_i$ repräsentiert die Gesamtanzahl der Cache-Misses eines Quellknotes v_i unter Berücksichtigung der WCECs seiner Kontexte. Die Anzahl der gemeinsam belegten Sets zwischen dem Quellknoten v_i und dem Zielknoten v_j wird über $Sets_{ij}$ beschrieben.

Da eine Funktion aus Basisblöcken besteht, berechnen sich die Kantengewichte des globalen Konfliktgraphen ähnlich wie die des lokalen Konfliktgraphen:

$$w_{ij}^{G} = WCCF_i * BBMissesSum_i * Sets_{ij} \tag{4.3}$$

Die $WCCF_i$ stellt die Aufrufhäufigkeit des Quellknotens dar. $BBMissesSum_i$ ist die Summe über die *MissesSum* aller Basisblöcke der Funktion. $Sets_{ij}$ steht hier analog zu Gleichung (4.2) für die Anzahl der gemeinsam belegten Sets zwischen v_i und v_j.

Die Knotengewichte können im lokalen bzw. globalen Konfliktgraphen über die WCECs der Basisblöcke und die WCCFs der Funktionen einfach zugeordnet werden.

$$m_i^{L} = WCEC_i \tag{4.4}$$

$$m_i^G = WCCF_i \tag{4.5}$$

Nach der Gewichtung können die Graphen noch verfeinert werden. So ist über die WCET-Analyse bekannt, welche Basisblöcke bzw. Funktionen nicht auf dem WCEP liegen. Da die später folgende Code-Positionierung nur auf dem WCEP arbeiten sollte, werden alle Knoten, die nicht auf dem WCEP liegen, entfernt. Dies beinhaltet alle Knoten mit einem Gewicht von 0. Ebenfalls werden alle Knoten entfernt, deren ausgehende Kanten allesamt ein Gewicht von 0 aufweisen. Dies beinhaltet auch die Löschung von Knoten, die zwar auf dem WCEP liegen, jedoch keine Cache-Misses aufweisen.

Nach der Verfeinerung eines Konfliktgraphen über die Knoten- und Kantengewichte beinhaltet dieser jedoch noch Kanten, die keine realen Konflikte darstellen. So existieren Kanten mit einem Gewicht größer 0, obwohl keine Konflikt-Misses vorliegen. Dies liegt daran, dass die aus der WCET-Analyse ermittelte Anzahl der Cache-Misses nicht nur die Anzahl der *Konflikt-Misses*, sondern auch die der *Cold Start-Misses* enthält. Zudem existieren unreale Kanten zwischen Speicherobjekten, die z. B. in unterschiedliche Zeilen eines Sets eingelagert werden oder aufgrund des Kontrollflusses niemals in einen Konflikt geraten. Um diese Kanten ebenfalls zu entfernen, sind weitere Analyse-Schritte notwendig. Diese werden im folgenden Abschnitt erläutert.

4.3. Verfeinerung der Konfliktmodellierung

Für eine effiziente Code-Positionierung ist es notwendig, einen Konfliktgraphen nach der Gewichtung noch weiter zu verfeinern. Bei der Code-Positionierung erfolgt nach jeder neuen Anordnung zweier Speicherobjekte ein Aufruf an das WCET-Analysewerkzeug aiT. So wird ermittelt, wie sich eine Positionierung auf die WCET auswirkt. Bei einer erfolgreichen Positionierung wird aufgrund der Änderungen des Speicherlayouts ein neuer Konfliktgraph erstellt. Die Verfeinerung des Konfliktgraphen sorgt somit nicht nur für eine effiziente Code-Positionierung, sondern spart auch unnötige WCET-Analysen ein, da unreale Kanten entfernt werden und diese von der Code-Positionierung nicht betrachtet werden müssen.

Über einen ersten Analyse-Schritt (Abschnitt 4.3.1) wird ermittelt, welche Speicherobjekte aufgrund des Kontrollflusses nicht im Konflikt stehen können. In einem weiteren Analyse-Schritt (Abschnitt 4.3.2) werden dann Kanten zwischen Speicherobjekten entfernt, die aufgrund der Cachezustände im Kontrollfluss nicht im Konflikt stehen. Abschließend wird eine Untersuchung des Speicherlayouts durchgeführt, um weitere Kanten zu entfernen, die kein Optimierungspotenzial besitzen (Abschnitt 4.3.3).

4.3.1. Kontrollfluss-Analyse

Speicherobjekte, deren Sets sich überlappen, müssen nicht zwingend im Konflikt stehen. Wenn z. B. eines der Speicherobjekte in einem *if*-Statement aufgerufen wird, während das andere Objekt in einem *else*-Statement steht, ist eine Verdrängung aufgrund des Kontrollflusses nicht möglich, außer eine solche Verzweigung steht in einer Schleife.

Für jede Funktion und damit für jeden lokalen Konfliktgraphen steht, wie bereits erwähnt, ein Kontrollflussgraph über die Basisblöcke der ICD-LLIR zur Verfügung. Anhand dieses Graphen kann der Kontrollfluss analysiert werden. Damit ein Cache-Konflikt zwischen Basisblöcken entstehen kann, müssen die Basisblöcke in einer bestimmten Abfolge im Kontrollfluss enthalten sein. Eine Konfliktkante e_{ij} drückt aus, dass beim Zugriff auf v_i aufgrund von v_j ein Cache-Miss auftritt. Dies ist nur dann möglich, wenn v_i sowohl vor als auch nach v_j im Kontrollfluss vorhanden ist. Abbildung 4.6 veranschaulicht dies anhand eines Pfades.

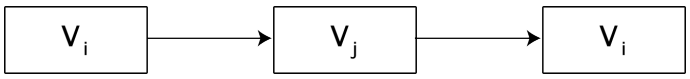

Abbildung 4.6.: Kontrollflusspfad einer Konfliktkante e_{ij}.

So muss v_i vor v_j ausgeführt werden, um von v_j verdrängt werden zu können. Denn durch die Ausführung von v_i werden die zugehörigen Speicherbereiche in den Cache eingelagert. Wird v_i nicht vor v_j ausgeführt, kann der Cache auch keine Bereiche des Speicherobjekts v_i beinhalten, die durch eine spätere Ausführung von v_j verdrängt werden. Zudem muss v_i auch nach v_j ausgeführt werden, damit es beim Zugriff auf v_i aufgrund von v_j zu Cache-Misses kommt. Beim Zugriff auf v_i werden dann die zugehörigen Cachezeilen aus dem Hauptspeicher nachgeladen, da diese bereits von v_j belegt sind. Wird v_i nicht nach v_j ausgeführt, so können zwar Bereiche von v_i durch v_j verdrängt werden, es werden jedoch keine Cache-Misses erzeugt, da nicht mehr auf v_i zugegriffen wird. Dabei müssen v_i und v_j nicht direkt hintereinander im Kontrollfluss enthalten sein. Es reicht aus, wenn v_i im Kontrollfluss von v_j aus erreichbar ist und umgekehrt.

Um unreale Konflikte aus einem lokalen Konfliktgraphen zu entfernen, überprüft die Kontrollfluss-Analyse für jede Konfliktkante, ob die zugehörigen Speicherobjekte in der oben beschriebenen Abfolge im Kontrollflussgraphen enthalten sind. Abbildung 4.7 veranschaulicht die Funktionsweise der Kontrollfluss-Analyse an einem konkreten Beispiel.

Auf der linken Seite ist ein lokaler Konfliktgraph dargestellt, die gestrichelten Kanten werden über die Kontrollfluss-Analyse entfernt. Auf der rechten Seite ist der

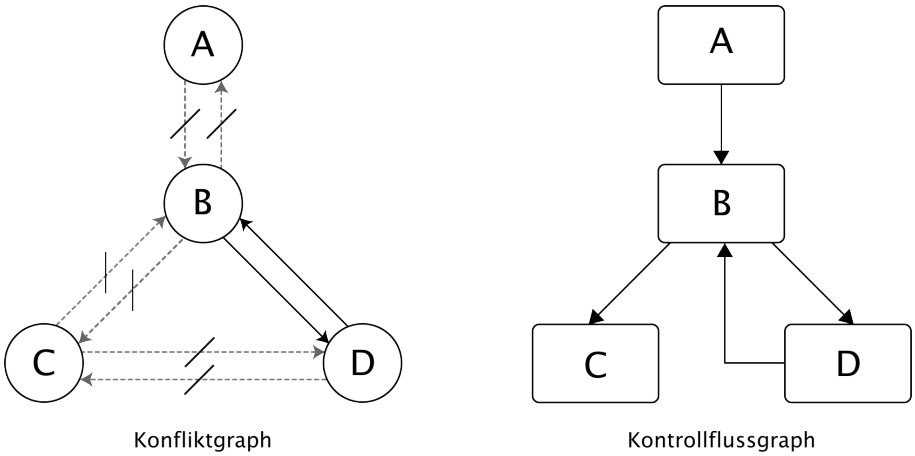

Abbildung 4.7.: Verfeinerung des Konfliktgraphen durch die Kontrollfluss-Analyse.

zugehörige Kontrollflussgraph dargestellt. Der Basisblock B besitzt zwei ausgehende Kanten, die eine Verzweigung des Kontrollflusses darstellen. So kann die Funktion entweder direkt über den Basisblock C verlassen werden oder zunächst den Basisblock D ausführen. Die Basisblöcke C und D können aufgrund der Verzweigung nicht im Konflikt stehen. Zwar könnte D von C aus dem Cache verdrängt werden, da sich laut Abbildungsfunktion die Sets von C und D überlappen. Da jedoch D nach C nicht mehr ausgeführt wird, können keine Cache-Misses aufgrund von C entstehen. Die Basisblöcke B und D werden innerhalb des Kontrollflussgraphen in einer Schleife ausgeführt. Somit können beim Zugriff auf D Cache-Misses durch B ausgelöst werden und umgekehrt. So werden Konfliktkanten zwischen Basisblöcken, die innerhalb einer Schleife ausgeführt werden, nicht entfernt.

Eine Konfliktkante wird demnach von der Kontrollfluss-Analyse wie folgt entfernt:

Definition 4.2 *Eine Konfliktkante e_{ij} wird entfernt, wenn es im Kontrollfluss keinen Pfad gibt, in dem v_i sowohl vor als auch nach v_j enthalten ist.*

Da die ICD-LLIR nur einen Kontrollflussgraph pro Funktion beinhaltet, jedoch keinen funktionsübergreifenden Kontrollflussgraphen zur Verfügung stellt, wird beim globalen Konfliktgraphen für die Kontrollfluss-Analyse ein zusätzlicher *Call-Graph* benötigt. Der Call-Graph steht bereits durch [LFM08] zur Verfügung. Er beinhaltet Kanten zwischen Funktionen, die bzgl. des Programm-Kontrollflusses hintereinander ausgeführt werden, da sie sich z. B. gegenseitig aufrufen. Da der Schwerpunkt dieser Arbeit auf der Code-Positionierung von Basisblöcken liegt, geht die Kontrollfluss-Analyse des globalen Konfliktgraphen etwas gröber vor als die der lokalen Konfliktgraphen. Bei dem globalen Konfliktgraphen werden alle Kanten zwischen Funktionen entfernt, die nicht im Call-Graphen vorhanden sind und sich somit nicht direkt hintereinander aufrufen. Das Konfliktpotenzial von Funktionen, die sich direkt hin-

tereinander aufrufen, ist größer als bei Funktionen, die nur indirekt über eine andere Funktion hintereinander ausgeführt werden. Daher wird trotz der gröberen Vorgehensweise eine effiziente Code-Positionierung ermöglicht.

Nach der Kontrollfluss-Analyse wird eine weitere Verfeinerung durchgeführt. So können Konfliktkanten zwischen Speicherobjekten entfernt werden, die aufgrund des vom Kontrollfluss abhängigen Cacheinhaltes nicht im Konflikt stehen. Dieser Analyse-Schritt wird im folgenden Abschnitt näher erläutert.

4.3.2. May-Analyse

Ein Speicherobjekt v_j kann ein anderes Speicherobjekt v_i nur verdrängen, wenn beim Zugriff auf v_j v_i in den gemeinsamen Sets der Speicherobjekte enthalten ist. Die Kontrollfluss-Analyse überprüft hierfür die Ausführungsreihenfolge der Speicherobjekte. Jedoch kann dabei keine Aussage darüber getroffen werden, ob v_i sich beim Zugriff von v_j wirklich noch im Cache befindet. So könnte v_i bereits vor dem Zugriff auf v_j durch ein anderes Speicherobjekt verdrängt werden. Damit wäre eine Konfliktkante e_{ij} unreal, da nicht aufgrund von v_j Cache-Misses erzeugt würden, sondern durch ein Speicherobjekt, welches vor v_j ausgeführt wird. Befindet sich v_i in keinem Fall vor dem Zugriff auf v_j im Cache, so kann die Konfliktkante e_{ij} entfernt werden.

Für das Entfernen solcher unrealen Konflikte wird beim lokalen Konfliktgraphen eine May-Analyse eingesetzt. Diese baut auf den Konzepten der in Abschnitt 2.3.2 beschriebenen May-Analyse auf, welche ebenfalls im WCET-Analysewerkzeug aiT integriert ist. Dabei werden für jeden Programmpunkt im Kontrollflussgraphen die abstrakten Cachezustände und damit die Zustände der einzelnen Sets bestimmt. Ein Cachezustand definiert sich hierbei über den Inhalt des Caches. Bei der May-Analyse beinhaltet ein Cachezustand die Speicherobjekte, die sich *möglicherweise* im Cache befinden. So kann Aussage darüber getroffen werden, welche Speicherobjekte sich an einem bestimmten Programmpunkt in keinem Fall im Cache befinden. Jedoch kann innerhalb der ICD-LLIR nicht auf die einzelnen Cachezustände des WCET-Analysewerkzeugs aiT zugegriffen werden. Nur die Anzahl der Cache-Misses eines Basisblocks ist bekannt. In dieser Arbeit wird daher eine May-Analyse entworfen, welche die einzelnen Cachezustände im Kontrollfluss einer Funktion modelliert. Über diese Zustände kann dann ermittelt werden, welche Konflikte aufgrund des Cacheinhaltes unreal sind.

Die eingehenden Cachezustände eines Basisblocks werden über die in den Gleichungen (2.12) bzw. (2.13) definierte Update- und Join-Funktion bestimmt. So kann bei einer Konfliktkante e_{ij} überprüft werden, ob vor dem Zugriff auf v_j v_i bereits im Cache enthalten war und somit von v_j verdrängt werden kann. Abbildung 4.8 veranschaulicht die Funktionsweise der May-Analyse.

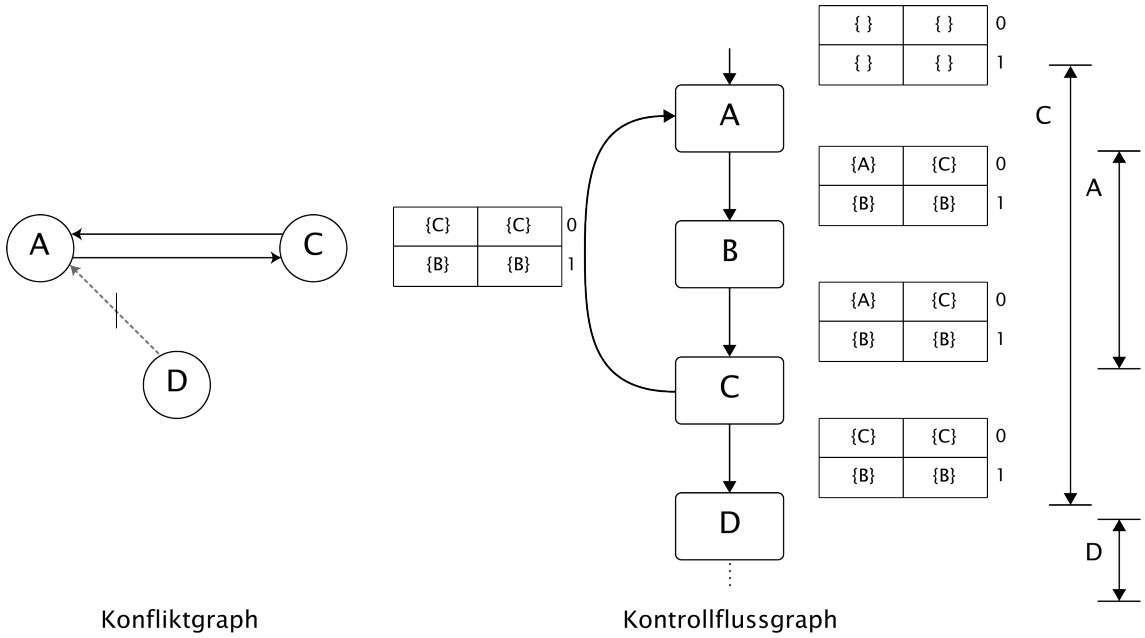

Abbildung 4.8.: Verfeinerung des Konfliktgraphen durch die May-Analyse.

Auf der linken Seite ist ein Ausschnitt aus einem Konfliktgraphen abgebildet. Er beinhaltet drei Knoten, welche den Basisblöcken des Kontrollflussgraphen auf der rechten Seite zugeordnet sind. Basisblöcke A, C und D belegen dabei dasselbe Set und stehen mit B nicht im Konflikt.

Die eingehenden Cachezustände der Basisblöcke sind an den Kontrollflusskanten eingezeichnet. Der Cache ist hier ein 2-fach mengenassoziativer Speicher, welcher zwei Sets beinhaltet. Dabei belegt Basisblock A nur eine Cachezeile im Set 0, während alle anderen Basisblöcke ihr zugehöriges Set vollständig belegen. Die Reihenfolge der Zugriffe auf die Basisblöcke ist über den Kontrollflussgraphen bekannt. So können anhand der in Gleichung (2.9) definierten Sequenz von Update-Funktionen und der LRU-Verdrängungsstrategie die Cachezustände entlang des Kontrollflusses modelliert werden. Wie in Abbildung 4.8 dargestellt, wird beim ersten Basisblock des Kontrollflusses einer Funktion von einem leeren Cachezustand ausgegangen. Über die Update-Funktion wird der Basisblock A in das Set 0 eingelagert. Besitzt ein Basisblock mehr als eine eingehende Kontrollflusskante, so werden die Cachezustände über die Join-Funktion für jedes Set zusammengeführt. A besitzt durch die Kontrollflussschleife zwei eingehende Cachezustände. Da A jedoch den ersten Basisblock im Kontrollfluss darstellt, erfolgt hier über die Join-Funktion lediglich eine Vereinigung mit leeren Set-Zuständen.

Bei einem Kontrollfluss, der Schleifen enthält, können sich die Cachezustände im ersten Schleifendurchlauf von denen im zweiten Schleifendurchlauf unterscheiden.

Daher werden die Zustände über die Update-Funktion solange entlang des Kontrollflusses aktualisiert, bis sie sich nicht mehr verändern. Sind alle Cachezustände stabil, so kann bestimmt werden, zu welchem Zeitpunkt sich die Basisblöcke im Cache befinden. Für die Basisblöcke A, C und D sind hierfür am Kontrollflussgraphen Intervalle eingezeichnet. Die Intervalle von A und D überschneiden sich dabei nicht, da Basisblock A bei Austritt aus der Schleife nicht mehr im Cache vorhanden ist, da er vom Basisblock C vollständig verdrängt wurde. Somit wird die Kante e_{DA} im Konfliktgraphen entfernt. Die Konfliktkanten, deren Quellknoten in den eingehenden Cachezuständen der Zielknoten enthalten sind, bleiben bestehen. Die Kanten des Konfliktgraphen können somit über die May-Analyse wie folgt entfernt werden:

Definition 4.3 *Ist bei einer Kante e_{ij} v_i nicht in den eingehenden Cachezuständen von v_j enthalten, so wird die Kante e_{ij} aus dem Konfliktgraphen entfernt.*

Dabei werden immer nur die Sets betrachtet, die von den zugehörigen Basisblöcken einer Kante gemeinsam belegt werden.

Jeder Basisblock belegt eine bestimmte Menge an Blöcken im Hauptspeicher. Bevor die May-Analyse jeden eingehenden Cachezustand eines Basisblocks berechnen kann, muss die Anzahl der Blöcke und damit die Anzahl der belegten Cachezeilen innerhalb eines Sets bekannt sein. Hierfür wird die Abbildungsfunktion *Map(Addr)* aus der Gleichung (4.1) verwendet. Werden die Basisblöcke aus Abbildung 4.4 betrachtet, so würde z. B. der Basisblock A in jedem seiner drei Sets jeweils eine Cachezeile belegen. Mit Hilfe dieser Information kann die Update-Funktion entlang des Kontrollflusses die Basisblöcke auf die Cachezeilen abbilden und damit die Set-Zustände aktualisieren. Die Aktualisierung erfolgt dabei pro Set. Der Zustand eines Sets wird gemäß der Funktion (2.7) beschrieben.

Zu Beginn der May-Analyse wird für den ersten Basisblock der eingehende Cachezustand initialisiert. Dieser setzt sich aus den Set-Zuständen gemäß der Abbildungsfunktion (2.7) zusammen. Besitzt eine Funktion eine Aufrufhäufigkeit von 1, so ist der eingehende Cachezustand leer. Wird die Funktion jedoch mehrmals ausgeführt, da sie z. B. innerhalb einer Schleife steht, so muss davon ausgegangen werden, dass sich bereits alle Basisblöcke der Funktion im Cache befinden. Dies ist notwendig, um zu verhindern, dass reale Konfliktkanten durch die May-Analyse entfernt werden. So wird der Startzustand des Caches über die Vereinigung der Set-Zustände erzeugt, wobei die Basisblöcke bereits auf die Sets abgebildet sind.

Jeder Basisblock verwaltet seine eingehenden Cachezustände und besitzt Zugriff auf die eingehenden Cachezustände seiner Nachfolger. Abbildung 4.9 veranschaulicht die Propagation dieser Zustände.

Nachdem der erste Basisblock A per Update-Funktion in den initialen Cachezustand

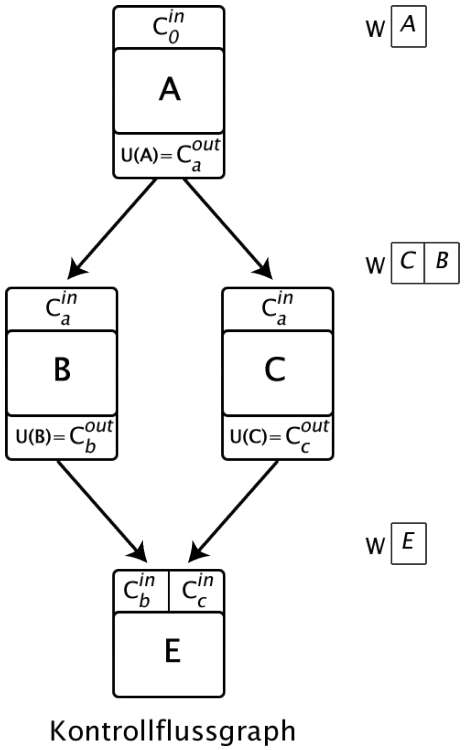

Abbildung 4.9.: Propagation der Cachezustände.

C_0^{in} eingefügt wurde, wird der aktualisierte Zustand C_a^{out} entlang des Kontrollflusses an seine Nachfolger B und C propagiert. C_a^{out} stellt zu Beginn den eingehenden Cachezustand C_a^{in} für die Basisblöcke B und C dar. Für die Abarbeitung der Basisblöcke werden diese in einer Liste verwaltet, die in Abbildung 4.9 mit W bezeichnet ist. Diese Liste ist zu Beginn der May-Analyse nur mit dem ersten Basisblock A besetzt, dieser wird wieder aus der Liste entfernt, wenn er den Cachezustand C_a^{out} für seine Nachfolger B und C erzeugt hat. Danach werden die Nachfolger B und C in die Abarbeitungsliste W eingefügt. Ein Basisblock fügt dabei seine direkten Nachfolger nur dann in die Liste ein, wenn die eingehenden Cachezustände der Nachfolger noch nicht stabil sind. Um festzustellen, ob ein eingehender Cachezustand stabil ist, vergleicht z. B. der Basisblock C seinen erstellten Cachezustand C_c^{out} mit dem zuvor erstellten eingehenden Zustand C_c^{in} von Basisblock E. Wird der Basisblock C zum ersten Mal besucht, so existiert noch kein zuvor erstellter Zustand C_c^{in}, daher wird E in die Abarbeitungsliste eingefügt. Wäre jedoch der von C erstellte Zustand C_c^{out} gleich einem zuvor erstellten C_c^{in}, so würde C den Basisblock E nicht in die Abarbeitungsliste eintragen, da der Cachezustand C_c^{in} in diesem Fall stabil ist. Somit ist die Liste, in welche die zu bearbeitenden Basisblöcke von ihren Vorgängern eingetragen werden, leer, wenn alle eingehenden Cachezustände stabil sind.

Hat ein Basisblock mehrere Vorgänger, so besitzt er auch mehrere eingehende Cachezustände. Diese werden per Join-Funktion vereinigt. Per Update-Funktion wird

der Basisblock dann in diesen vereinigten Cachezustand eingefügt. Der so erzeugte Zustand wird an die Nachfolger propagiert. Handelt es sich bei dem Basisblock jedoch um einen Schleifenkopf, wird ein solcher Cachezustand nicht direkt an die Nachfolger propagiert. Abbildung 4.10 veranschaulicht die Vorgehensweise bei der Propagation eines Cachezustandes in einer Schleife nach [FW99].

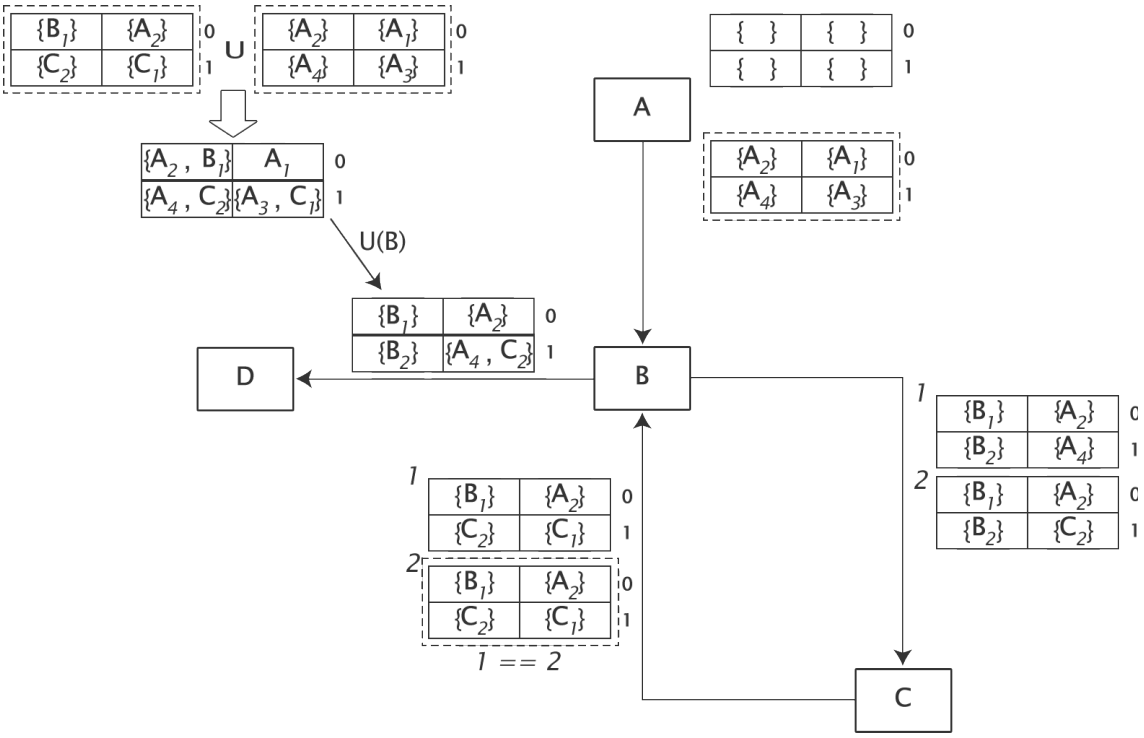

Abbildung 4.10.: Propagation von Cachezuständen einer Schleife.

Hier sind im Kontrollfluss vier Basisblöcke A, B, C und D enthalten. Der Cache hat eine Assoziativität von 2 und besitzt 2 Sets. Die von der May-Analyse erzeugten eingehenden Cachezustände sind an den Kanten eingezeichnet. Die einzelnen Hauptspeicherblöcke eines Basisblocks sind in den Cachezuständen jeweils nummeriert. Dabei belegt der Basisblock A bestehend aus den Blöcken A_1 bis A_4 in den Sets 0 und 1 jeweils zwei Cachezeilen. Basisblock B ist ebenfalls mit B_1 und B_2 auf die Sets 0 und 1 abgebildet. Basisblock C hingegen belegt mit C_1 und C_2 nur das Set 1. Da bei der Entfernung von unrealen Konfliktkanten nur die eingehenden Cachezustände relevant sind, wird die Belegung von Basisblock D nicht benötigt.

Nachdem der Basisblock A in den initial leeren Cachezustand eingefügt wurde, wird dieser an seinen Nachfolger B propagiert. Der Basisblock B stellt hier einen Schleifenkopf dar. Er besitzt zwei Vorgänger A und C und damit auch zwei eingehende Cachezustände. Jedoch werden nicht direkt alle eingehenden Cachezustände eines Schleifenkopfs über die Join-Funktion vereinigt. Nur die Zustände, die von Vorgän-

gern wie A außerhalb der Schleife erzeugt wurden, werden vereinigt. Die von Vorgängern wie C erzeugten Zustände werden erst nach ihrer Stabilisierung hinzugefügt. Der Basisblock B wird daher zunächst nur in den von A erzeugten Cachezustand eingefügt. Danach wird dieser Cachezustand nur an die Nachfolger innerhalb der Schleife propagiert. An den Kanten zwischen B und C sind jeweils zwei Cachezustände eingetragen. Dabei stellen die mit 1 nummerierten Zustände den Cacheinhalt im ersten Durchlauf der Schleife dar und die mit 2 nummerierten Zustände den Inhalt aller weiteren Schleifendurchläufe. Bereits beim zweiten Schleifendurchlauf hat sich der Cachezustand an der Rückkante von B stabilisiert. Der Basisblock C hat somit zweimal hintereinander den gleichen eingehenden Cachezustand für B erzeugt. Jetzt kann die Join-Funktion auf alle eingehenden Cachezustände des Schleifenkopfes B angewendet werden. Die Vereinigung der Zustände ist oben links in der Abbildung 4.10 dargestellt. Nachdem B in den vereinigten Zustand eingefügt wurde, wird dieser allerdings nur an die Nachfolger außerhalb der Schleife propagiert. Somit werden immer zunächst die Cachezustände innerhalb einer Schleife stabilisiert, bevor diese weiter propagiert werden.

Um die May-Analyse wie beschrieben durchzuführen, muss bekannt sein, welche Basisblöcke innerhalb bzw. außerhalb einer Schleife liegen. Zudem wird die Information benötigt, welche Basisblöcke einen Schleifenkopf repräsentieren. Hierfür wird eine bereits im WCC implementierte Lösung verwendet, die Informationen über Schleifen und ihre Basisblöcke zur Verfügung stellt.

Die May-Analyse terminiert, wenn alle eingehenden Cachezustände stabil sind. Dies ist garantiert, da sowohl die Update- als auch die Join-Funktion monoton sind und es nur eine endliche Anzahl an abstrakten Cachezuständen gibt, denn die Anzahl der Sets, der Cachezeilen und der Basisblöcke ist endlich [FW99]. Sind alle eingehenden Cachezustände stabil, wird der Konfliktgraph nach Definition 4.4 verfeinert.

Nach der Entfernung von unrealen Konflikten durch die Kontrollfluss- und die May-Analyse, wird eine weitere Verfeinerung des globalen und der lokalen Konfliktgraphen durchgeführt. Diese Verfeinerung erfolgt über eine Untersuchung des Speicherlayouts und wird im folgenden Abschnitt erläutert.

4.3.3. Untersuchung des Speicherlayouts

Die Code-Positionierung ordnet Speicherobjekte, die im Konflikt stehen, kontinuierlich im Hauptspeicher an. So wird die Überlappung der Indexadressen der Objekte verringert bzw. aufgelöst, und damit auch die Überlagerung der Sets. Die Konfliktgraphen enthalten jedoch auch Kanten zwischen Speicherobjekten, die bereits vor der Code-Positionierung kontinuierlich im Hauptspeicher angeordnet sind. Solche Konfliktkanten stellen somit kein Optimierungspotenzial dar.

Durch die Untersuchung des Speicherlayouts werden daher die Konfliktkanten in globalen und lokalen Graphen wie folgt entfernt:

Definition 4.4 *Ein Konfliktkante e_{ij} wird entfernt, wenn die zugehörigen Speicherobjekte v_i und v_j bereits kontinuierlich im Hauptspeicher angeordnet sind.*

Über die ICD-LLIR kann auf das Speicherlayout der Basisblöcke und Funktionen zugegriffen werden. So ist es möglich benachbarte Speicherobjekte zu ermitteln und die zugehörigen Kanten in den Konfliktgraphen zu entfernen.

Nach diesem letzten Analyse-Schritt der WCET-bewusste Cache-Analyse können nun die verfeinerten Konfliktgraphen als Grundlage für die Code-Positionierung verwendet werden. Der Entwurf der Code-Positionierung wird im folgenden Kapitel beschrieben.

5. WCET-gesteuerte Code-Positionierung

In diesem Abschnitt wird die Code-Positionierung der vorliegenden Arbeit vorgestellt. Nach einer kurzen Einführung (Abschnitt 5.1) folgt zunächst die Beschreibung der Code-Positionierung auf Basisblock-Ebene (Abschnitt 5.2). Danach wird der Algorithmus zur Code-Positionierung auf Funktions-Ebene erläutert (Abschnitt 5.3).

5.1. Einführung

Speicherobjekte, auf die mit einer hohen zeitlichen Lokalität zugegriffen wird und die im Hauptspeicher ungünstig angeordnet sind, führen zu einem schlechten Cacheverhalten. Eine ungünstige Anordnung besteht dann, wenn solche Objekte auf dieselben Bereiche im Cache abgebildet werden. Durch die Überlagerung im Cache stehen die Speicherobjekte im Konflikt, da sie sich häufig gegenseitig aus dem Cache verdrängen können. Daraus entsteht eine hohe Anzahl möglicher Konflikt-Misses. Ein schlechtes Cacheverhalten wie dieses führt zu einer starken Überabschätzung der WCET. Kann die Anzahl möglicher Konflikt-Misses reduziert werden, so führt dies ebenfalls zu einer Reduktion der Überabschätzung. Mit Hilfe einer Code-Positionierung ist es möglich, ein schlechtes Cacheverhalten zu verbessern und damit die WCET zu optimieren. Die in dieser Arbeit entworfene Code-Positionierung ordnet hierfür im Konflikt stehende Objekte kontinuierlich im Hauptspeicher an. Über eine solche Anordnung wird die Überlagerung von gemeinsam belegten Bereichen im Cache verringert bzw. aufgelöst. Abbildung 5.1 veranschaulicht, wie sich die Überlagerung zwischen Speicherobjekten über eine kontinuierliche Anordnung auflöst.

Im linken Teil der Abbildung 5.1 ist der Ausschnitt eines Konfliktgraphen dargestellt. Die Speicherobjekte A und B stehen dabei in einem gegenseitigen Konflikt. Die Kante von A nach B verdeutlicht, dass beim Zugriff auf das Speicherobjekt A Cache-Misses aufgrund von B entstehen. Zudem existiert eine Kante von B nach A. Somit wird das Speicherobjekt B ebenfalls von A aus dem Cache verdrängt. Im linken Teil der Abbildung 5.1 sind das Speicherlayout und die Belegung des Caches vor der Positionierung dargestellt. Der rechte Teil veranschaulicht die Belegung des Caches nach der Positionierung. Der Cache ist hier ein 2-fach mengenassoziativer Speicher mit 5 Sets. Die Cachezeilengröße beträgt 32 Byte. So werden die Blöcke

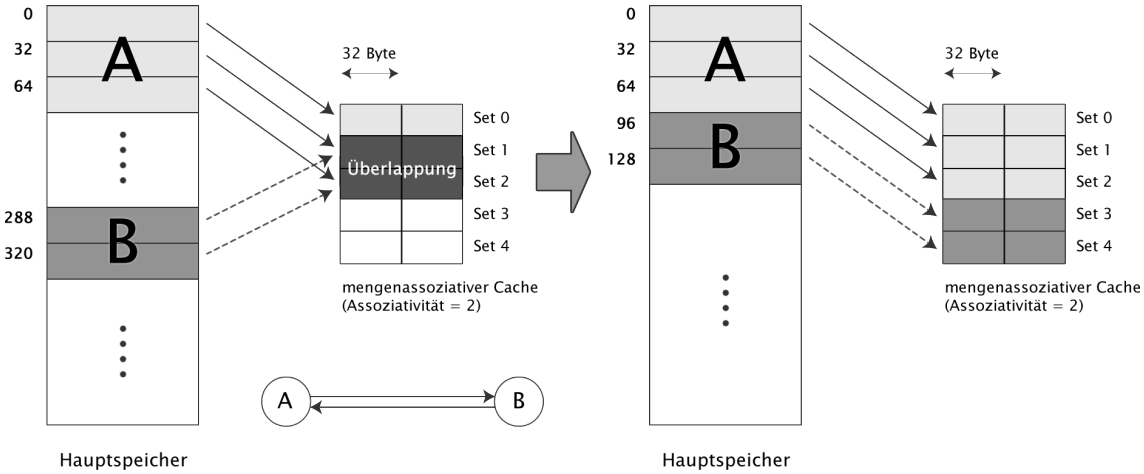

Abbildung 5.1.: Auflösung von Konflikten über eine kontinuierliche Anordnung.

des Speicherobjekts A vor der Neuanordnung auf die Sets 0 bis 2 abgebildet und die des Speicherobjekts B auf die Sets 1 bis 2. A und B sind vor der Positionierung so angeordnet, dass sich die Indexteile ihrer Hauptspeicheradressen überlagern. Somit müssen sie sich die Sets 1 und 2 teilen. Hierdurch kommt es zu einer gegenseitigen Verdrängung. Dabei werden besonders viele Konflikt-Misses erzeugt, wenn die Speicherobjekte eine hohe zeitliche Lokalität aufweisen, da sie z.B. gemeinsam in einer Schleife aufgerufen werden.

Die durch eine gegenseitige Verdrängung erzeugten Konflikt-Misses werden über eine kontinuierliche Anordnung der Speicherobjekte aufgelöst. Dies ist auf der rechten Seite der Abbildung 5.1 veranschaulicht. Die Speicherobjekte A und B sind nun unmittelbar nebeneinander im Hauptspeicher angeordnet. B erhält dadurch andere Indexadressen, die sich nicht mehr mit denen von A überlappen. Daraus resultiert auch eine neue Abbildung auf die Sets. Das Speicherobjekt B belegt nun statt der Sets 1 bis 2 die Sets 3 bis 4. Somit müssen sich A und B keine Sets mehr teilen und können sich somit auch nicht mehr gegenseitig aus dem Cache verdrängen.

Die Information über im Konflikt stehende Speicherobjekte erhält die Code-Positionierung mit Hilfe der im Kapitel 4 vorgestellten Cache-Analyse. Die Code-Positionierung findet dabei zunächst auf Ebene der Basisblöcke statt und in einem zweiten Schritt auf Ebene der Funktionen eines Programms. Die Informationen über die Konflikte auf Basisblock- und Funktions-Ebene werden dabei wie bereits beschrieben über die Cache-Analyse in Form von Konfliktgraphen zur Verfügung gestellt. Die Gewichte der Kanten dieser Graphen geben Aufschluss darüber, welche Speicherobjekte bei einer Neuanordnung das höchste Optimierungspotenzial bieten. So besitzt die Kante mit dem höchsten Gewicht das größte Optimierungspotenzial, da die zugehörigen Speicherobjekte eine hohe Anzahl an Konflikt-Misses erzeugen.

Sowohl die Code-Positionierung von Basisblöcken als auch die von Funktionen wird anhand eines Greedy-Algorithmus durchgeführt. Dieser positioniert iterativ die Objekte mit dem jeweils höchsten Kantengewicht im Hauptspeicher unmittelbar nebeneinander. Hierbei wird für jede Positionierung eine eigene WCET-Analyse durchgeführt, um die Instabilität des WCEPs zu beachten. Das durch eine Neuanordnung entstandene Speicherlayout wird dabei nur übernommen, wenn dieses zu einer Reduktion der WCET führt. Dieses Speicherlayout dient dann als Startpunkt für den nächsten Optimierungs-Schritt. Damit wird sichergestellt, dass die WCET des neuen Speicherlayouts niemals schlechter ist als die des vorherigen Speicherlayouts.

Im folgenden Abschnitt wird zunächst der Algorithmus zur Code-Positionierung auf Basisblock-Ebene erläutert. Danach erfolgt die Beschreibung der Code-Positionierung auf Funktions-Ebene.

5.2. Basisblock-Positionierung

Die Basisblock-Positionierung wird pro Funktion durchgeführt. So wird für jede Funktion des Programms ein lokaler Konfliktgraph gemäß der Cache-Analyse aus Kapitel 4 erstellt. Anhand dieses Graphen werden die im Konflikt stehenden Basisblöcke innerhalb einer Funktion neu angeordnet.

Bei der Basisblock-Positionierung wird zusätzlich eine Sprungkorrektur benötigt. Eine Sprungkorrektur kann notwendig sein, wenn zwei Basisblöcke A und B, die sowohl im Hauptspeicher als auch im Kontrollfluss direkt aufeinander folgen, getrennt werden. Endet der Basisblock A z. B. mit einem Sprung, dessen Bedingung nicht zutrifft, so wird der im Speicher nachfolgende Basisblock B ausgeführt. Wird B nun von A getrennt angeordnet, so müssen die Basisblöcke über eine zusätzliche Sprunganweisung wieder verbunden werden. Eine Trennung kann durch eine Basisblock-Positionierung verursacht werden, da hier eine Neuanordnung auf Basis der Konfliktgraphen durchgeführt wird. Durch eine kontinuierliche Anordnung von im Konflikt stehenden Basisblöcken kann jedoch auch eine zuvor benötigte Sprunganweisung von der Sprungkorrektur entfernt werden. Im WCC ist bereits eine solche Sprungkorrektur enthalten, die nach jedem Positionierungs-Schritt zum Einsatz kommt.

Die Vorgehensweise der Basisblock-Positionierung ist im Algorithmus 5.1 dargestellt. Als Eingabe dient das zu optimierende Programm P, welches in Form einer ICD-LLIR vorliegt. Ziel der Positionierung ist es, durch eine Reduktion der Konflikt-Misses von Basisblöcken die WCET des Programms P zu reduzieren. Hierbei wird zunächst die aktuelle $WCET_{cur}$ des Programms ermittelt (Zeile 1).

Der Ablauf der Basisblock-Positionierung geht dabei nach einem Greedy-Algorithmus

Eingabe : Programm P
Ausgabe : optimiertes Programm P

```
1  Ermittle WCET von P: WCET_cur = getWCET(P)
2  foreach Funktion f ∈ P do
3  |  Erzeuge lokalen Konfliktgraphen G = (V, E, w, m) für f
4  |  foreach Kante e_ij ∈ E aus G in absteigender Reihenfolge gemäß w do
5  |  |  Erstelle Kopie von P: P' = P
6  |  |  Positioniere v_i hinter v_j in P'
7  |  |  Führe Sprungkorrektur durch
8  |  |  Ermittle WCET von P': WCET_new = getWCET(P')
9  |  |  if WCET_new < WCET_cur then
10 |  |  |  Übernehme neues Speicherlayout: P = P'
11 |  |  |  Ermittle WCET von P: WCET_cur = getWCET(P)
12 |  |  |  Erzeuge neuen lokalen Konfliktgraphen G = (V, E, w, m) für f
13 |  |  end
14 |  |  if WCET_new == WCET_cur then
15 |  |  |  Lösche Kante e_ij aus E: E = E\{e_ij}
16 |  |  end
17 |  |  if WCET_new > WCET_cur then
18 |  |  |  Abbruch: Bearbeite nächste Funktion
19 |  |  end
20 |  end
21 end
```

Algorithmus 5.1: Greedy-Algorithmus der Basisblock-Positionierung.

vor. Für jede Funktion f des Programms P werden die im Konflikt stehenden Basisblöcke iterativ neu angeordnet. Hierfür wird für eine Funktion f zunächst ein lokaler Konfliktgraph $G = (V, E, w, m)$ gemäß der Cache-Analyse erzeugt. Anhand der Kantenmenge E des Graphen G werden dann die zugehörigen Basisblöcke positioniert. Die Kanten werden dabei in absteigender Reihenfolge gemäß ihrem Optimierungspotenzial bearbeitet, welches dem Kantengewicht w entspricht.

Die Positionierung zweier Basisblöcke wird dabei nicht direkt an dem Programm P durchgeführt, sondern an einer Kopie P'. Dies ist notwendig, da das Speicherlayout nach einer Positionierung nur übernommen werden soll, wenn es zu einer Reduktion der $WCET_{cur}$ beiträgt. Führt eine Positionierung jedoch zu einer Verschlechterung der $WCET_{cur}$ ist diese, wenn sie auf dem originalen Programm P durchgeführt wird, nur sehr schwer reversibel. Grund dafür ist die benötigte Sprungkorrektur, die, wie beschrieben, nach einer Positionierung durchgeführt werden muss. Werden die Basisblöcke an ihre alte Position gesetzt, muss auch eine erneute Sprungkorrektur erfolgen. Eine solche Rückpositionierung führt jedoch nicht exakt zum originalen Speicherlayout von P. Daher erfolgt die Positionierung zweier Basisblöcke immer zu-

nächst auf der Kopie P'.

Die Basisblöcke v_i und v_j der höchstgewichtigen Konfliktkante e_{ij} werden in P' unmittelbar nebeneinander positioniert (Zeile 6). Danach erfolgt eine Sprungkorrektur, und die $WCET_{new}$ des veränderten Programms P' wird ermittelt. Durch die erneute WCET-Analyse kann festgestellt werden, ob die Neuanordnung von v_i und v_j zu einer Reduktion der WCET führt (Zeile 9). Ist dies der Fall und die ermittelte $WCET_{new}$ von P' ist geringer als die $WCET_{cur}$ des originalen Programms P, so wird das optimierte Speicherlayout für P übernommen. Das neue Speicherlayout dient dann als Startpunkt für den nächsten Optimierungs-Schritt. So wird sichergestellt, dass die WCET des originalen Programms P nach einer Positionierung niemals schlechter als zuvor ist. Nach der Übernahme des neuen Speicherlayouts von P' nach P, wird eine zusätzliche WCET-Analyse für das Programm P durchgeführt (Zeile 11). Die aktuellen WCET-Informationen werden nach der Ermittlung der $WCET_{new}$ (Zeile 8) von aiT an das Programm P' angehängt, liegen jedoch noch nicht in P vor. Da die Neubildung des Konfliktgraphen $G = (V, E, w, m)$ von f (Zeile 12) anhand des originalen Programms P durchgeführt wird, müssen die WCET-Informationen von P aktualisiert werden. So stehen für die Gewichtung des Graphen stets die aktuellen WCET-Informationen zur Verfügung. Der Konfliktgraph der Funktion f muss hier erneut gebildet werden, da bei der Basisblock-Positionierung ein neues Speicherlayout entstanden ist. Ein einfaches Streichen der Konfliktkante e_{ij} ist hier nicht ausreichend, da es durch die Veränderung des Speicherlayouts auch zu völlig neuen Konflikten kommen kann, die es zu berücksichtigen gilt. Zudem können durch die Positionierung zweier Basisblöcke auch Konflikte zu anderen Basisblöcken aufgelöst werden, die dann unnötig betrachtet würden. Erst nach der Neubildung des Konfliktgraphen von f wird mit der Positionierung fortgefahren, dabei wird die höchstgewichtige Kante des neuen Graphen bearbeitet.

Ist die ermittelte $WCET_{new}$ von P' jedoch gleich der ursprünglichen $WCET_{cur}$ von P, so wird das Speicherlayout nicht übernommen (Zeile 14). Damit ist auch eine Neubildung des Konfliktgraphen von f nicht notwendig und die Kante e_{ij} wird einfach aus der Kantenmenge E des Graphen G entfernt. So kann mit der nächst höhergewichtigen Kante fortgefahren werden, bis der Graph keine Kanten mehr besitzt oder eine Verschlechterung der $WCET_{new}$ eintritt. Bei einer Verschlechterung der $WCET_{new}$ im Vergleich zur $WCET_{cur}$ (Zeile 17) wird ein Abbruch durchgeführt. Es wird also keine weitere Kante des Konfliktgraphen von f betrachtet, stattdessen werden die Konflikte innerhalb der nächsten Funktion des Programms P bearbeitet. Der Algorithmus terminiert, wenn alle Funktionen des Programms P anhand ihrer Konfliktgraphen bearbeitet wurden.

Nach der Basisblock-Positionierung erfolgt eine weitere Code-Positionierung auf Ebene der Funktionen des Programms. Diese Funktions-Positionierung wird im folgenden Abschnitt beschrieben.

5.3. Funktions-Positionierung

Ziel der Funktions-Positionierung ist es, die Konflikt-Misses von Funktionen zu reduzieren und damit die WCET des von der Basisblock-Positionierung optimierten Programms noch weiter zu reduzieren. Die Informationen über die im Konflikt stehenden Funktionen werden hier gemäß der Cache-Analyse durch einen globalen Konfliktgraphen zur Verfügung gestellt.

Die Funktions-Positionierung geht dabei ähnlich wie die Basisblock-Positionierung nach einem Greedy-Algorithmus vor. Hier wird jedoch keine Sprungkorrektur benötigt, da über die return-Instruktion einer Funktion automatisch zum Aufrufer zurückgekehrt wird. Ein Verhalten wie bei Basisblöcken, die mit einem bedingten Sprung enden, existiert somit nicht. So ist die Positionierung von Funktionen bei einem Misserfolg reversibel, weshalb keine Kopie des Programms angelegt werden muss. Die Vorgehensweise der Funktions-Positionierung ist im Algorithmus 5.2 dargestellt. Als Eingabe dient hier das Programm P, welches zuvor von der Basisblock-

Eingabe : Programm P
Ausgabe : optimiertes Programm P

1 Ermittle WCET von P: $WCET_{cur}$ = getWCET(P)
2 Erzeuge globalen Konfliktgraph $G = (V, E, w, m)$ für P
3 **foreach** *Kante* $e_{ij} \in E$ *aus* G *in absteigender Reihenfolge gemäß* w **do**
4 Merke Position von v_i: $pos^{v_i}_{orig} = pos^{v_i}$
5 Positioniere v_i hinter v_j in P
6 Ermittle WCET von P: $WCET_{new}$ = getWCET(P)
7 **if** $WCET_{new} < WCET_{cur}$ **then**
8 Erzeuge neuen globalen Konfliktgraphen $G = (V, E, w, m)$ für P
9 **end**
10 **if** $WCET_{new} == WCET_{cur}$ **then**
11 Positioniere v_i an alte Position: $v_i \rightarrow pos^{v_i}_{orig}$
12 Lösche Kante e_{ij} aus E: $E = E \backslash \{e_{ij}\}$
13 **end**
14 **if** $WCET_{new} > WCET_{cur}$ **then**
15 Positioniere v_i an alte Position: $v_i \rightarrow pos^{v_i}_{orig}$
16 Abbruch
17 **end**
18 **end**

Algorithmus 5.2: Greedy-Algorithmus der Funktions-Positionierung.

Positionierung optimiert wurde. Zunächst wird für die spätere Evaluierung einer Positionierung die aktuelle $WCET_{cur}$ des Programms P ermittelt. Danach erfolgt die Erzeugung des globalen Konfliktgraphen $G = (V, E, w, m)$ von P. Hiermit stehen die Informationen über die im Konflikt stehenden Funktionen des Programms zur Ver-

fügung. So können anhand der Kantenmenge E des Graphen nun die zugehörigen Funktionen positioniert werden. Die Kanten werden hier, wie bei der Basisblock-Positionierung, in absteigender Reihenfolge gemäß ihres Gewichtes w bearbeitet. So werden die Funktionen mit dem größten Optimierungspotenzial zuerst positioniert.

Die Funktions-Positionierung findet hier direkt auf dem Programm P statt. Führt eine Neuanordnung dabei nicht zur Reduktion der $WCET_{cur}$, so wird das Speicherlayout wieder in seine ursprüngliche Form gebracht. Hierfür ist es notwendig, sich die Position pos^{v_i} der Funktion v_i zu merken (Zeile 4). Die Position von v_j muss nicht abgespeichert werden, da für eine kontinuierliche Anordnung lediglich v_i eine neue Position hinter v_j erhält.

Nachdem die Position $pos^{v_i}_{orig}$ von v_i abgespeichert wurde, erfolgt die Neuanordnung der zur Kante e_{ij} gehörigen Funktionen v_i und v_j. Um die Instabilität des WCEPs zu berücksichtigen, ist eine erneute WCET-Analyse des modifizierten Programms P notwendig. Damit kann die Positionierung innerhalb von P bzgl. der $WCET_{new}$ evaluiert werden. Ist die $WCET_{new}$ geringer als die $WCET_{cur}$, welche vor der Positionierung ermittelt wurde, so dient das Speicherlayout von P als Startpunkt für den nächsten Optimierungs-Schritt (Zeile 7). Vor dem nächsten Optimierungs-Schritt ist es jedoch notwendig, einen neuen globalen Konfliktgraphen $G = (V, E, m, w)$ auf Basis des veränderten Speicherlayouts von P zu erzeugen. Dies ist wie bereits in Abschnitt 5.2 beschrieben notwendig, da die Neuanordnung von Funktionen zu einer Veränderung der ursprünglichen Konflikte führt und damit auch zu andern Kantengewichten w. So wird in der nächsten Iteration die höchstgewichtige Kante des neuen Konfliktgraphen G bearbeitet.

Ist die $WCET_{new}$ des modifizierten Programms P gleich der zuvor ermittelten $WCET_{cur}$ (Zeile 10), so wird die Positionierung von v_i und v_j zurückgeführt. Die Funktion v_i erhält so ihre ursprüngliche Position $pos^{v_i}_{orig}$. Danach wird die Kante e_{ij} aus dem Konfliktgraphen entfernt, um mit der nächst höhergewichtigen Kante fortzufahren. Führt das neue Speicherlayout jedoch zu einer Verschlechterung der WCET, so dass die $WCET_{new}$ größer als die ursprüngliche $WCET_{cur}$ ist (Zeile 14), wird die Positionierung zurückgeführt und der Algorithmus terminiert. Da das Speicherlayout von P nur bei einer Reduktion der $WCET_{cur}$ beibehalten wird, ist sichergestellt, dass es durch die Funktions-Positionierung niemals zu einer Verschlechterung der WCET kommt. Auch wenn die Funktions-Positionierung erst nach der Basisblock-Positionierung ausgeführt wird, ist die Reihenfolge der Positionierungen für die Reduktion der WCET nicht ausschlaggebend. So ist es durchaus möglich, die Funktions-Positionierung auch vor der Basisblock-Positionierung durchzuführen.

Nachdem die Cache-bewusste Code-Positionierung der vorliegenden Arbeit vollständig vorgestellt wurde, erfolgt im nächsten Abschnitt die Evaluierung dieser WCET-Optimierung.

6. Auswertung

Im Rahmen dieser Arbeit wurde eine Cache-bewusste Code-Positionierung zur Reduktion der WCET entworfen und in den WCC integriert. Um die Güte dieser WCET-Optimierung zu evaluieren, wurden verschiedene Benchmarks untersucht. In diesem Kapitel werden die Ergebnisse der Evaluierung präsentiert. Hierfür wird zunächst erläutert, in welchem Rahmen die Auswertungen stattfanden und welche Benchmarks genutzt wurden (Abschnitt 6.1). Danach folgt eine getrennte Vorstellung der Ergebnisse der Code-Positionierung auf Basisblock-Ebene (Abschnitt 6.2) und der Code-Positionierung auf Funktions-Ebene (Abschnitt 6.3). Hier werden die Auswertungen bzgl. der Verfeinerungsschritte der Cache-Analyse aus Kapitel 4, der Reduktion der Cache-Misses und der WCET präsentiert. Bei der Basisblock-Positionierung wird ebenfalls die Codegröße betrachtet. Abschließend werden die Auswertungen für die Kombination beider Positionierungstechniken präsentiert. Dabei wird nicht nur die Güte bzgl. der WCET und der Reduktion der Cache-Misses evaluiert, sondern auch die ACET und die CPU-Laufzeit der Optimierung betrachtet.

6.1. Verfahren zur Auswertung

Zur Evaluierung der entworfenen WCET-Optimierung werden verschiedene Parameter ausgewertet. Im Vordergrund steht hierbei die Betrachtung der WCET-Reduktion. Da diese anhand der Verringerung der Cache-Misses erreicht werden soll, wird ebenfalls die Anzahl der reduzierten Cache-Misses und der in Abschnitt 4.2.2 beschriebenen MissesSum ausgewertet. Das Optimierungspotenzial der Code-Positionierung hängt dabei im Wesentlichen von den Informationen der Cache-Analyse (siehe Kapitel 4) ab. Daher werden die einzelnen Analyse-Schritte, die in der Cache-Analyse zur Verfeinerung der Konfliktgraphen eingesetzt werden, ebenfalls betrachtet. Hier wird ausgewertet, wie viele unreale Konfliktkanten von dem jeweiligen Analyse-Schritt entfernt wurden. Je weniger unreale Konflikte ein Graph enthält, desto effizienter gestaltet sich die Code-Positionierung. Da Code-Positionierungstechniken allgemein auch zur Optimierung der ACET eingesetzt werden, soll ebenfalls die Auswirkung auf die durchschnittliche Laufzeit betrachtet werden. Abschließend wird zudem die CPU-Laufzeit der entworfenen WCET-Optimierung evaluiert.

Die Auswertungen werden insgesamt für direkt-abgebildete Instruktionscaches und für mengenassoziative Caches mit einer Assoziativität von 2, 4, 8 und 16 durch-

geführt. Eine detaillierte Beschreibung der Rahmenbedingungen, unter denen die Ergebnisse gewonnen werden, folgt im nächsten Abschnitt.

6.1.1. Rahmenbedingungen

Zur Auswertung der WCET-Optimierung werden die Benchmarks zunächst ohne die Cache-bewusste Code-Positionierung mit dem WCC kompiliert. Die hieraus gewonnen Ergebnisse dienen dann als Referenzwerte für einen erneuten Durchlauf mit der WCET-Optimierung. Als Zielarchitektur wird der TC1797 verwendet. Die Benchmarks werden unter der höchsten Optimierungsstufe (O3) getestet, um bereits hoch optimierten Code noch weiter zu verbessern. Die Cache-bewusste Code-Positionierung wird als eine der letzten Low-Level-Optimierungen ausgeführt. Nach ihr wird aufgrund von O3 nur noch ein Instruktions-Scheduling durchgeführt. Dieses wird jedoch deaktiviert, da lediglich die Ergebnisse, die von der Code-Positionierung erzeugt werden, auszuwerten sind und durch ein zusätzliches Instruktions-Scheduling verfälscht würden.

Für die WCET-Analyse wird das Werkzeug aiT in der Version a^3 verwendet, welches den ILP-Solver CPLEX nutzt. Der Programmcode der Benchmarks wird in den gecachten Flash-Speicher des TriCore geladen, um das Verhalten des Instruktionscaches analysieren zu können. Dabei werden verschiedene Assoziativitäten betrachtet, die von 1 (direkt-abgebildet) bis einschließlich 16 konfiguriert werden. Um ein realistisches Verhältnis zwischen Programmgröße und zur Verfügung stehender Cache-Kapazität zu schaffen wird die Cachegröße, wie auch in [PLM10] beschrieben, auf 10% der Programmgröße eines Benchmarks angepasst. Eine Code-Positionierung ist nur dann sinnvoll, wenn genügend Sets zur Verfügung stehen. Beinhaltet der auf 10% der Programmgröße angepasste Cache nur eine Cachezeile bzw. ein Set, kann eine Überlagerung zwischen im Konflikt stehenden Speicherobjekten nicht aufgelöst werden. Aus diesem Grund wird eine Mindestgröße festgelegt, die bei einem direkt-abgebildeten Cache der Größe von 2 Cachezeilen entspricht, und bei einem mengenassoziativen Cache der Größe von 2 Sets. Die Cachezeilengröße beträgt dabei stets 32 Byte.

Zur Bestimmung der ACET innerhalb des WCC wird das Entwicklungswerkzeug CoMET der Firma Synopsys verwendet. Dieses Werkzeug kann als Simulationsumgebung für verschiedene Hardwarearchitekturen eingesetzt werden und unterstützt hierbei auch den TriCore Prozessor.

Die Auswertungen werden auf einem Standard-PC-System mit einem Linux-Betriebssystem durchgeführt. Die CPU-Laufzeitmessungen der WCET-Optimierung finden dabei auf einem System statt, welches mit 8 Intel Xeon L5420 Prozessorkernen ausgestattet ist, die jeweils eine Taktfrequenz von 2,5 GHz besitzen.

Folgend werden die für die Auswertungen verwendeten Benchmarks betrachtet.

6.1.2. Verwendete Benchmarks

Um repräsentative Ergebnisse zu erzeugen, wurde die *WCETBENCH* des WCC genutzt. Diese setzt sich aus den Benchmark-Sammlungen DSPstone [ZMM94], MediaBench [LPM97], MiBench [GRE01], MRTC [Mae11], StreamIt [Str11], NetBench [MMH01], UTDSP [UTD11] und misc zusammen. Die misc Benchmark Sammlung enthält dabei Benchmarks, die von ICD-C stammen.

Die in der WCETBENCH vorhandenen Benchmarks decken das breite Anwendungsfeld der eingebetteten Systeme ab und eignen sich daher besonders gut für die Untersuchung von WCET-Optimierungen. Jedoch werden nicht alle der Benchmarks zur Evaluierung der in dieser Arbeit entworfenen Cache-bewussten Code-Positionierung eingesetzt. Die Auswahl der verwendeten Benchmarks ist in Anhang A enthalten, diese wurde anhand verschiedener Kriterien getroffen:

- Benchmarks, die nur eine sehr geringe Codegröße besaßen oder fast keine Cache-Misses enthielten, wurden aufgrund ihrer geringen Aussagekraft bzgl. der Code-Positionierung entfernt.
- Die Code-Positionierung arbeitet auf Basis von Konfliktgraphen. Da jedoch zur Generierung eines Konfliktgraphen z. B. eine gewisse Mindestanzahl an Basisblöcken vorhanden sein muss (siehe Abschnitt 4.2.1), wurden ebenfalls Benchmarks ausgeschlossen, anhand derer keine Konfliktkanten generiert werden konnten und somit auch keine Code-Positionierung möglich war.
- Benchmarks, die im Zusammenhang mit aiT zu einem Berechnungsabbruch führten, mussten ebenfalls entfernt werden.

Es wurden insgesamt 21 Benchmarks ausgewählt, die eine Codegröße von 160 bis 10656 Byte aufweisen. Die Cachegröße wurde wie beschrieben für alle verwendeten Benchmarks, so weit möglich, auf 10% ihrer Codegröße angepasst oder auf die Mindestgröße von 2 Sets gesetzt. Im Folgenden werden zunächst die Auswertungen der Code-Positionierung auf Basisblock-Ebene separat präsentiert.

6.2. Code-Positionierung auf Basisblock-Ebene

Zur Code-Positionierung auf Basisblock-Ebene werden die Informationen über im Konflikt stehende Basisblöcke benötigt. Diese Informationen werden in Form von lokalen Konfliktgraphen über die in Kapitel 4 beschriebene Cache-Analyse zur Verfügung gestellt. In diesem Abschnitt werden daher zunächst die Verfeinerungsschritte, die bei der Cache-Analyse zur Generierung der Graphen eingesetzt werden, betrachtet. Danach erfolgt eine Präsentation der Auswertungen zur Cache-Miss Re-

duktion. Aus dieser Reduktion folgt auch die Reduktion der WCET. Die Ergebnisse hierzu werden ebenfalls vorgestellt. Abschließend erfolgt eine Betrachtung der Codegrößenveränderung, welche durch die verwendete Sprungkorrektur beeinflusst wird. Die Vorstellung der Auswertungen bzgl. der ACET und der CPU-Laufzeit erfolgt später anhand der gesamten WCET-Optimierung, welche aus der Basisblock-Positionierung und der Funktions-Positionierung besteht.

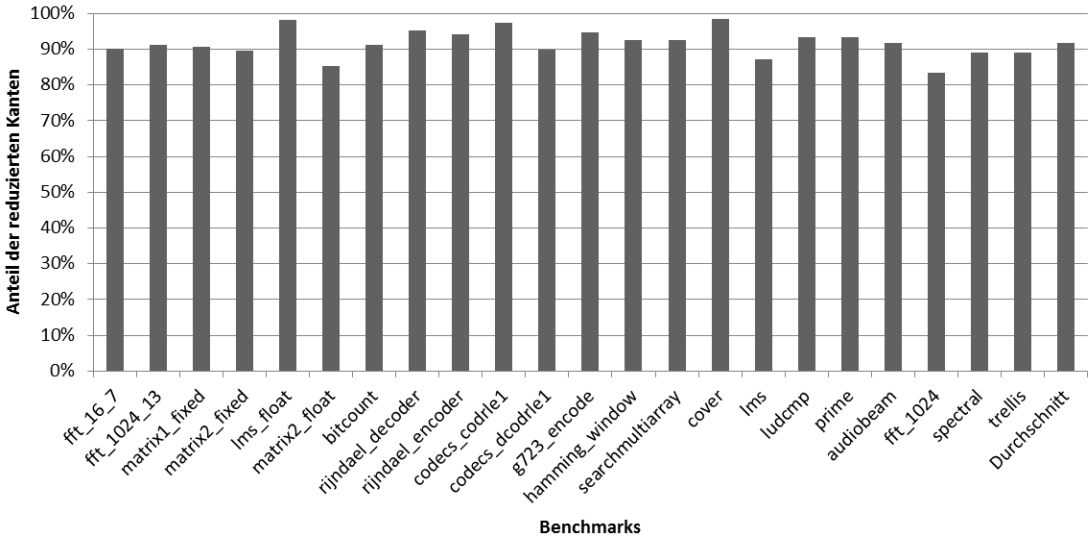

Abbildung 6.1.: Reduktion der Kantenanzahl durch die Cache-Analyse bei einer Assoziativität von 2 in den lokalen Konfliktgraphen.

Von der Cache-Analyse wird pro Funktion ein lokaler Konfliktgraph erzeugt, dessen Knoten die Basisblöcke der zugehörigen Funktion repräsentieren. Ein zunächst nur anhand der Abbildungsfunktion (4.1) erzeugter initialer Konfliktgraph wird anschließend über mehrere Analyse-Schritte verfeinert. Diese Verfeinerungen bestehen aus einer Kontrollfluss-Analyse, einer May-Analyse und der Untersuchung des Speicherlayouts. Durch diese Verfahren werden Kanten aus den Konfliktgraphen entfernt, die keine realen Konflikte darstellen bzw. kein Optimierungspotenzial bieten. Daher ist die Reduktion solcher Kanten ausschlaggebend für eine effiziente Code-Positionierung.

Abbildung 6.1 veranschaulicht die prozentuale Anzahl der Kanten, die insgesamt durch alle Verfeinerungsschritte der Cache-Analyse entfernt wurden. Hierbei wurde beispielhaft ein 2-fach mengenassoziativer Cache verwendet. Der Anteil der entfernten Kanten ist auf der y-Achse aufgetragen, während auf der x-Achse die jeweiligen Benchmarks zu sehen sind. Der Wert von 100% bedeutet, dass alle Kanten eines initialen Konfliktgraphen entfernt wurden. Dies ist nur dann der Fall, wenn der jeweilige Benchmark keine Konflikte zwischen Basisblöcken bezüglich des Caches

beinhaltet. So wurden minimal beim Benchmark fft_1024 83,3% der Kanten entfernt und maximale 98,4% beim Benchmark cover. Im Durchschnitt werden durch die einzelnen Analyse-Schritte insgesamt 91,7% der Kanten aus den initialen Konfliktgraphen entfernt. Somit wird eine effiziente Code-Positionierung ermöglicht, da ein sehr großer Anteil der Kanten, die kein Optimierungspotenzial darstellen, entfernt wird.

Tabelle 6.1.: Reduktion der Kantenanzahl bei separatem Einsatz der Analyse-Schritte für die lokalen Konfliktgraphen.

Analyse-Schritte	Kantenreduktion
Kontrollfluss-Analyse	15,9%
May-Analyse	83,2%
Untersuchung des Speicherlayouts	8,8%

In Tabelle 6.1 ist der gemittelte Anteil der entfernten Kanten bei separatem Einsatz der Analyse-Schritte über alle Benchmarks aus der Abbildung 6.1 aufgeführt. Da eine Kante jeweils von mehreren Analysen entfernt werden kann, ergibt sich eine Summe der Durchschnittswerte von über 100%. Ein Wert von 100% bedeutet hier also, dass alle Kanten der initialen Konfliktgraphen von einem Analyse-Schritt entfernt werden. So werden von der ursprünglichen Kantenanzahl der initialen Konfliktgraphen über die Kontrollfluss-Analyse im Durchschnitt 15,9% der initialen Kanten entfernt, während die Untersuchung des Speicherlayouts nur 8,8% zur Entfernung der Kanten beiträgt. Durch die May-Analyse wird mit 83,2% der größte Anteil der Kanten, die kein Optimierungspotenzial besitzen, entfernt. Dies ist auf die präzise Modellierung der Cachezustände im Kontrollfluss zurückzuführen.

Die Code-Positionierung ordnet die im Konflikt stehenden Basisblöcke kontinuierlich im Hauptspeicher an und reduziert somit die Cache-Misses. In Abbildung 6.2 sind die Werte der Cache-Misses aufgeführt, die durch die Code-Positionierung entstehen. Hier ist ebenfalls die MissesSum enthalten, welche sich aus der maximalen Ausführungshäufigkeit und der Anzahl der Cache-Misses aller Basisblöcke zusammensetzt. Der Vergleichswert von 100% an der y-Achse stellt die ursprüngliche Anzahl der Cache-Misses und der MissesSum vor der Code-Positionierung dar. Hier wurde beispielhaft wieder eine Assoziativität von 2 verwendet. Bei einigen Benchmarks wie z. B. spectral ist die Anzahl der Cache-Misses um 4,3% angestiegen. Dies begründet sich darin, dass die Cache-Misses der Basisblöcke mit einer hohen WCET gesenkt wurden, jedoch die Cache-Misses von Basisblöcken, die zu einem geringeren Anteil zur gesamt WCET beitragen, erhöht wurden. Daher ist auch der Wert der MissesSum angegeben, welcher dieses Verhältnis beinhaltet und beim spectral Benchmark somit um 4,5% reduziert wird. Bei den Benchmarks wie z. B. searchmultiarray tritt keine Veränderung der Cache-Misses bzw. MissesSum auf. Hier wurde aufgrund der geringen Programmgröße die Mindestgröße von 2 Sets gewählt. Da-

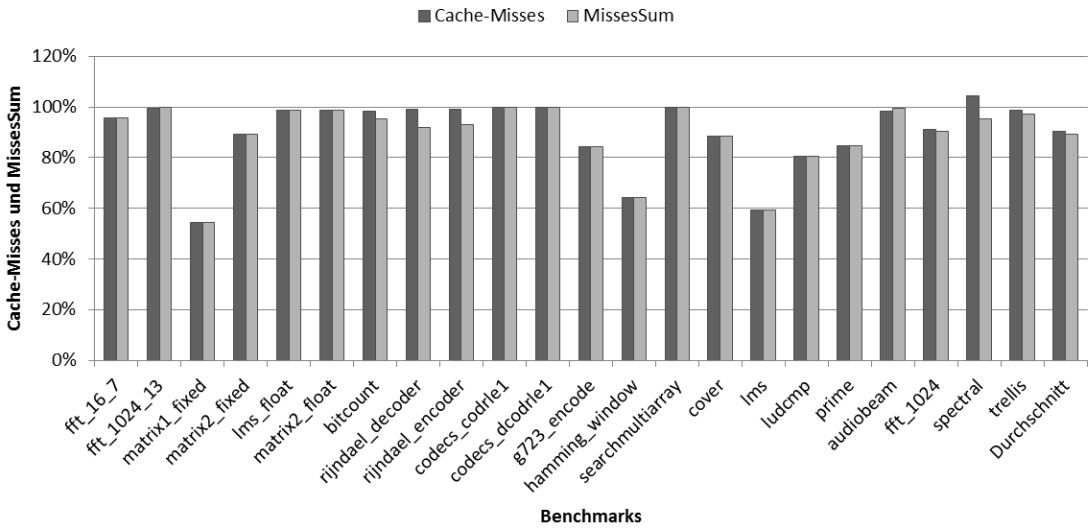

Abbildung 6.2.: Werte der Cache-Misses und MissesSum nach der Code-Positionierung auf Basisblock-Ebene für die Assoziativität 2.

durch beträgt die Cache-Kapazität hier weit mehr als 10% der Programmgröße, so dass nur ein geringes Konfliktpotenzial gegeben ist. Im Gegensatz dazu wird beim Benchmark matrix1_fixed für die Cache-Misses und die MissesSum eine Reduktion von 45,7% erzielt. Durch die Code-Positionierung konnte bei einer Assoziativität von 2 durchschnittlich eine Cache-Miss-Reduktion von 9,8% erreicht werden, wobei die MissesSum-Reduktion bei 10,9% liegt.

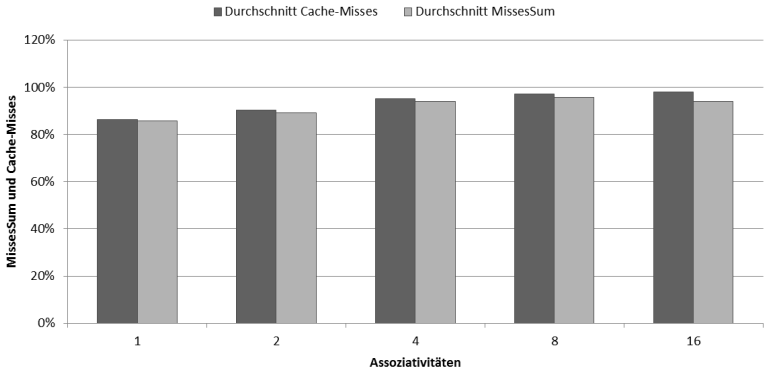

Abbildung 6.3.: Durchschnittswerte der Cache-Misses und MissesSum nach der Code-Positionierung auf Basisblock-Ebene über alle Assoziativitäten.

In Abbildung 6.3 sind die Werte der Cache-Misses und der MissesSum über alle konfigurierten Assoziativitäten dargestellt. Der Vergleichswert von 100% stellt

hier den Mittelwert der Cache-Misses und MissesSum aller Benchmarks dar. Hierbei ist auffällig, dass mit ansteigender Assoziativität auch das Reduktionspotenzial sinkt. Dies liegt daran, dass die Anzahl der Cache-Misses ebenfalls mit ansteigender Assoziativität sinkt, da mehr Cache-Zeilen pro Set vorhanden sind und somit weniger Verdrängungen stattfinden können. Dieses Verhältnis wirkt sich auch auf das Optimierungspotenzial der WCET aus. Die Auswertung der WCET unter einer Assoziativität von 2 ist in Abbildung 6.4 dargestellt.

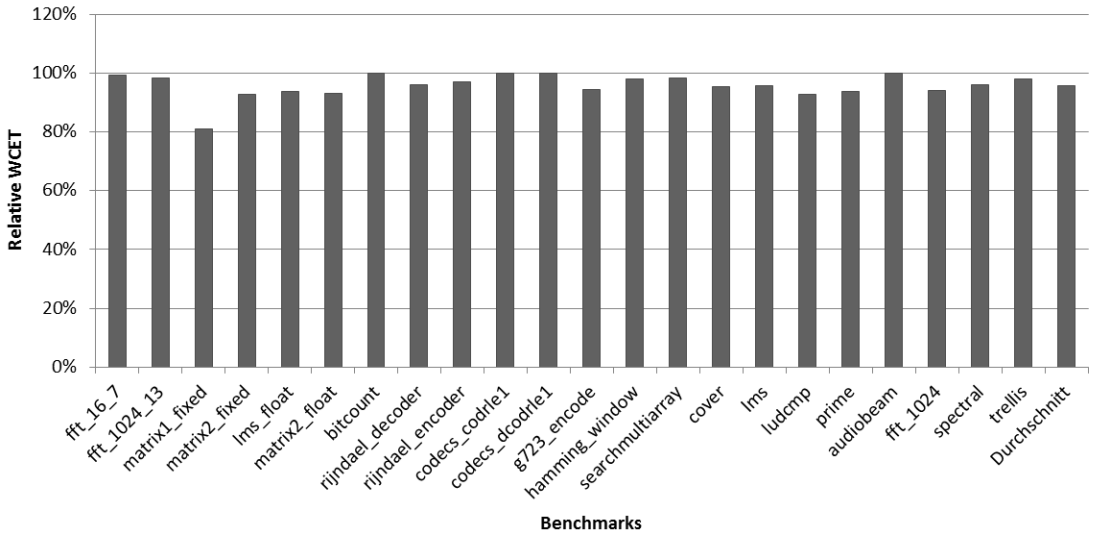

Abbildung 6.4.: Relative WCET-Werte nach der Code-Positionierung auf Basisblock-Ebene für die Assoziativität 2.

An der y-Achse ist die relative WCET aufgetragen wobei 100% hier stellvertretend für die WCET eines Benchmarks vor der Code-Positionierung steht. Die größte Reduktion von 18,9% konnte hier bei dem Benchmark matrix1_fixed erreicht werden. Dies spiegelt sich ebenfalls in der Reduktion der Cache-Misses bzw. MissesSum (siehe Abbildung 6.2) wieder. Im Durchschnitt über alle Benchmarks konnte durch die Code-Positionierung auf Basisblock-Ebene bei einer Assoziativität von 2 eine WCET-Reduktion von 4,3% erzielt werden.

Abbildung 6.5 zeigt den Mittelwert der WCET über alle Assoziativitäten. Bei einer Assoziativität von 16 wird nur eine sehr geringe WCET-Reduktion von 1,0% erreicht. Die Reduktion der WCET steigt dabei mit abnehmender Assoziativität. So konnte bei einem direkt-abgebildeten Cache (Assoziativität = 1) durchschnittlich eine Verbesserung von 7,5% erzielt werden. Dieses Verhältnis ist auf die in Abbildung 6.3 dargestellten Werte der Cache-Misses zurückzuführen, mit zunehmender Assoziativität ist hier die Reduktion der Cache-Misses geringer.

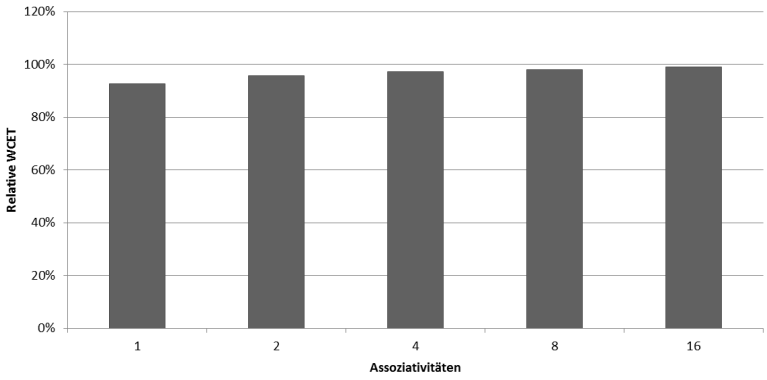

Abbildung 6.5.: Durchschnittswerte der WCET nach der Code-Positionierung auf Basisblock-Ebene über alle Assoziativitäten.

Abschließend soll noch die Codegrößenveränderung, welche durch die verwendete Sprungkorrektur beeinflusst wird, betrachtet werden. Abbildung 6.6 veranschaulicht die Auswirkungen der Code-Positionierung auf die Codegröße bei einer Assoziativität von 2.

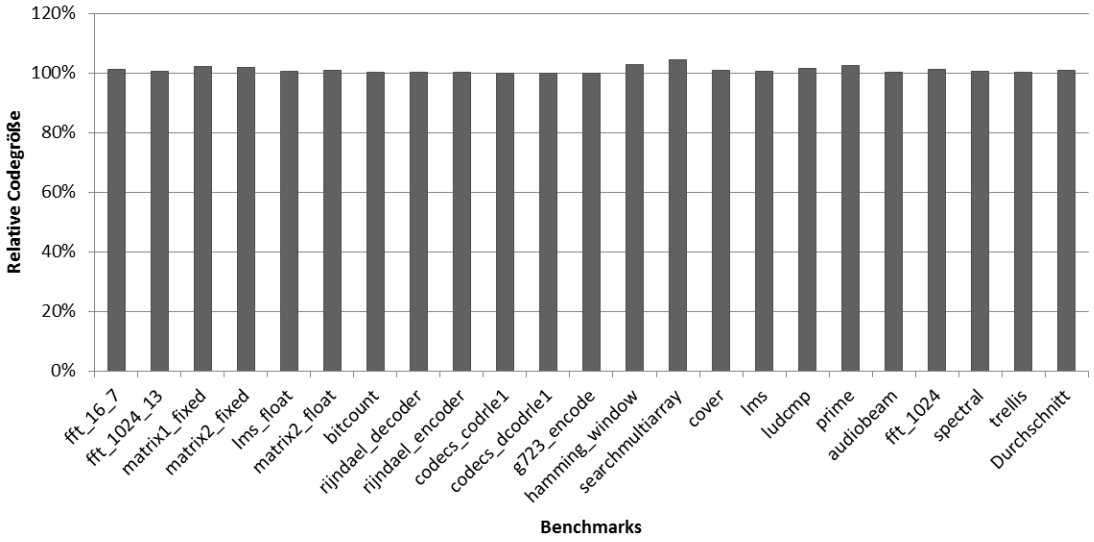

Abbildung 6.6.: Relative Codegröße nach der Code-Positionierung auf Basisblock-Ebene für die Assoziativität 2.

Auf der y-Achse ist die Codegröße der Benchmarks nach der Code-Positionierung angegeben. Der Wert von 100% stellt hierbei die Codegröße vor der Code-Positionierung dar. Die Sprungkorrektur, die bei jeder Iteration innerhalb der Code-Positionierung durchgeführt werden muss, führt hier nur zu einer unwesentlichen Vergrößerung. Die maximale Codevergrößerung liegt bei 4,3% bei dem Benchmark

searchmultiarray. Bei den meisten Benchmarks wie z. B. codecs_codrle1 oder audiobeam tritt jedoch keine Codevergrößerung auf. Durchschnittlich steigt die Codegröße über die zusätzlich eingefügten Sprunganweisungen nur um 1,0% bei einer Assoziativität von 2. Im Durchschnitt über alle Assoziativitäten liegt die Codevergrößerung bei 0,7%. Dieser geringe Wert begründet sich auch darin, dass bei einer kontinuierlichen Anordnung von zuvor getrennten Basisblöcken ebenfalls Sprunganweisungen entfernt werden.

Im folgenden Abschnitt werden die Evaluationsergebnisse der Code-Positionierung auf Funktions-Ebene dargestellt.

6.3. Code-Positionierung auf Funktions-Ebene

Zur Code-Positionierung auf Funktions-Ebene werden die Informationen über im Konflikt stehende Funktionen ebenfalls von der Cache-Analyse geliefert. Die Cache-Analyse erzeugt hierfür einen globalen Konfliktgraphen. In diesem Abschnitt werden daher zunächst die Verfeinerungsschritte, welche zur Erstellung des Graphen benötigt werden, betrachtet. Danach erfolgt die Auswertung bzgl. der Cache-Miss-Reduktion. Da aus dieser Reduktion wie auch bei der Basisblock-Positionierung die Reduktion der WCET folgt, werden die Ergebnisse hierzu ebenfalls vorgestellt. Die Codegröße wird im Vergleich zur Basisblock-Positionierung jedoch nicht betrachtet, da bei der Code-Positionierung auf Funktions-Ebene keine Sprungkorrektur benötigt wird.

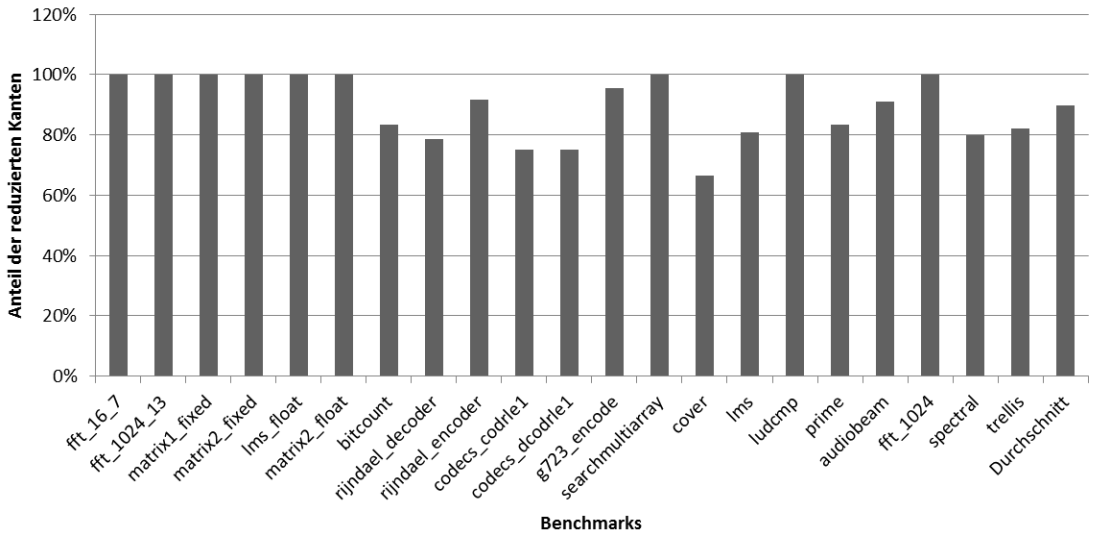

Abbildung 6.7.: Reduktion der Kantenanzahl durch die Cache-Analyse bei einer Assoziativität von 2 im globalen Konfliktgraphen.

Die Cache-Analyse erzeugt einen globalen Konfliktgraphen, dessen Knoten die Funktionen des Programms repräsentieren. Der Graph wird wie bei der Basisblock-Positionierung zunächst anhand der Abbildungsfunktion 4.1 erstellt. Danach wird dieser initiale Graph anhand einer Kontrollflussanalyse und der Untersuchung des Speicherlayouts verfeinert. Somit sollen hier ebenfalls Kanten aus dem Konfliktgraphen entfernt werden, die kein Optimierungspotenzial bieten. Eine May-Analyse wurde hier nicht durchgeführt, da die ICD-LLIR nur Kontrollflussgraphen für die Basisblöcke einer Funktionen zur Verfügung stellt, jedoch keinen funktionsübergreifenden Kontrollflussgraphen.

Abbildung 6.7 stellt den prozentualen Anteil der entfernten Kanten dar, welcher insgesamt durch beide Verfeinerungsschritte erzielt wurde, bei einer Assoziativität von 2. An der y-Achse ist der Anteil der entfernten Kanten dargestellt. Ein Wert von 100% bedeutet hier, dass alle Kanten aus dem initialen Konfliktgraphen entfernt wurden. Da die Benchmarks im Verhältnis zur Anzahl der Basisblöcke eine sehr geringe Anzahl an Funktionen besitzen (siehe Anhang A), wurden bei einem Großteil der Benchmarks bereits vor der Code-Positionierung alle Kanten entfernt. Somit konnte z. B. bei dem Benchmark ludcmp mit einer Entfernung der Kanten von 100% keine Funktions-Positionierung stattfinden. Bei dem Benchmark codecs_codrle1 hingegen sind nur 75,0% der Kanten entfernt worden. Im Durchschnitt über alle Benchmarks sind über die einzelnen Analyse-Schritte 90,5% der Kanten aus einem initialen Konfliktgraphen entfernt worden, wobei die Gesamtanzahl der Kanten hier wesentlich geringer war als bei der Basisblock-Positionierung. Die Tabelle

Tabelle 6.2.: Reduktion der Kantenanzahl bei separatem Einsatz der Analyse-Schritte für den globalen Konfliktgraphen.

Analyse-Schritte	Kantenreduktion
Kontrollfluss-Analyse	86,3%
Untersuchung des Speicherlayouts	13,6%

6.2 stellt den durchschnittlichen Anteil der Kantenreduktion bei separatem Einsatz der Kontrollflussanalyse und der Untersuchung des Speicherlayouts dar. Eine Kante kann hierbei sowohl von der Kontrollflussanalyse als auch von der Untersuchung der Speicherlayouts entfernt werden. Ein Wert von 100% bedeutet hier, dass alle Kanten im initialen Konfliktgraphen von einem Analyse-Schritt entfernt werden. So werden über die Untersuchung des Speicherlayouts 13,6% der Kanten entfernt, während die Kontrollflussanalyse ganze 86,3% der Kanten entfernt. Die Kontrollflussanalyse des globalen Konfliktgraphen entfernt alle Kanten zwischen Funktionen, die sich nicht direkt gegenseitig im Kontrollfluss aufrufen. Diese Vorgehensweise scheint jedoch etwas zu grob zu sein, da trotz vorhandener Cache-Misses bei vielen Benchmarks am Ende der Cache-Analyse der zugehörige Konfliktgraph keine Kanten mehr enthält. Folgend sollen nur noch die Benchmarks betrachtet werden, bei denen ein Kon-

fliktgraph vorhanden ist und zusätzlich auch eine erfolgreiche Positionierung von Funktionen stattgefunden hat.

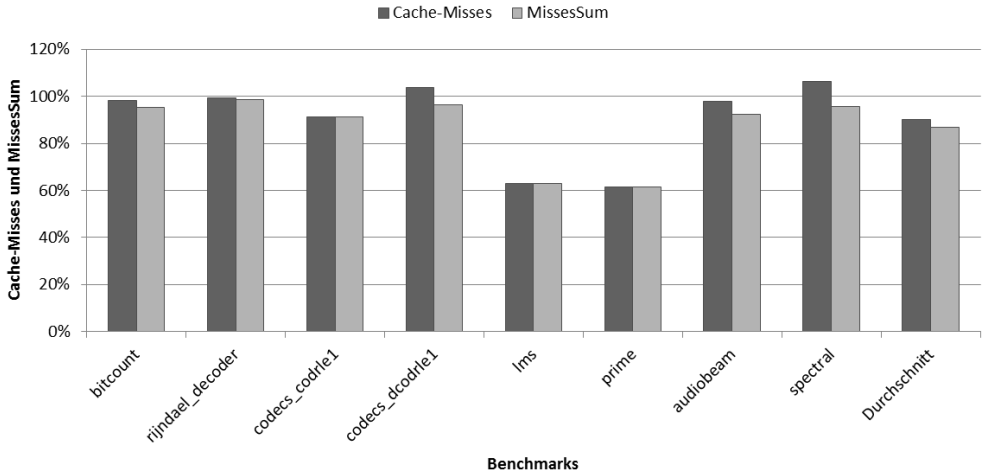

Abbildung 6.8.: Werte der Cache-Misses und MissesSum nach der Code-Positionierung auf Funktions-Ebene für die Assoziativität 2.

Abbildung 6.8 zeigt die Reduktion der Cache-Misses, die durch die Code-Positionierung auf Funktions-Ebene entsteht. Hierbei ist, wie auch bei der Basisblock-Positionierung, die MissesSum enthalten, um das Verhältnis von WCET und Cache-Misses zu verdeutlichen. Die Benchmarks, bei denen wie beschrieben kein Konfliktgraph vorhanden war, wurden entfernt. Der Vergleichswert von 100% steht für die ursprüngliche Anzahl der Cache-Misses und MissesSum. Die Code-Positionierung konnte hier bei dem Benchmark prime eine Reduktion von bis zu 38,5% erzielen. Durchschnittlich über alle der 8 hier betrachteten Benchmarks wurden die MissesSum um 14,2% und die Cache-Misses um 9,9% bei einer Assoziativität von 2 reduziert.

Die Ergebnisse der Konfigurationen der Assoziativität von 1 bis 16 sind in Abbildung 6.9 dargestellt. Die Reduktion der Cache-Misses und der MissesSum ist hier über alle Assoziativitäten veranschaulicht, wobei 100% den Mittelwert über die betrachteten 8 Benchmarks darstellt. Das Reduktionspotenzial der Cache-Misses und der MissesSum sinkt wie bei der Basisblock-Positionierung mit ansteigender Assoziativität. Bei einer Assoziativität von 1 wird im Durchschnitt eine Reduktion von 14,6% für die Cache-Misses und 20,6% für die MissesSum erreicht. Bei einer Assoziativität von 16 wird hingegen nur eine Reduktion von 4,0% für die Cache-Misses und 6,0% für die MissesSum erzielt. Dieses Verhältnis spiegelt sich auch in der Auswertung der WCET wieder.

Abbildung 6.10 veranschaulicht die relativen WCET-Werte, die bei einer Assoziati-

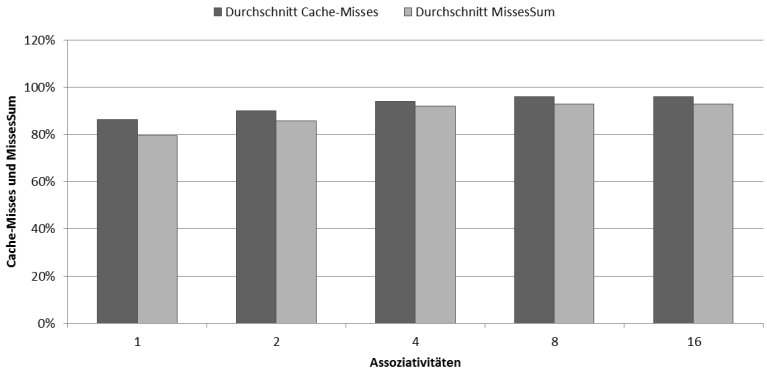

Abbildung 6.9.: Durchschnittswerte der Cache-Misses und MissesSum nach der Code-Positionierung auf Funktions-Ebene über alle Assoziativitäten.

vität von 2 durch die Code-Positionierung auf Funktions-Ebene erzeugt wurden. Der Wert von 100% steht hier für die WCET vor der Code-Positionierung. Dabei werden wie zuvor beschrieben nur die 8 Benchmarks betrachtet, bei denen eine erfolgreiche Funktions-Positionierung stattgefunden hat. Bei dem Benchmark codecs_dcodrle1 konnte eine WCET-Reduktion von 21,8% erzielt werden, wohingegen bei anderen Benchmarks wie z. B. prime nur eine WCET-Reduktion von 1,0% erreicht wurde. Durchschnittlich konnte die Code-Positionierung auf Funktions-Ebene eine WCET-Reduktion von 6,0% erzielen. Werden jedoch alle Benchmarks betrachtet, die auch bei der Basisblock-Positionierung zum Einsatz kamen liegt die Reduktion nur bei 2,3%. Dieser geringe Wert ist auf das zu grobe Vorgehen bei der Kontrollflussanalyse des globalen Konfliktgraphen zurückzuführen, jedoch auch auf die geringere Anzahl von Funktionen innerhalb der Benchmarks.

Abbildung 6.11 zeigt die Durchschnittswerte der WCET über alle Assoziativitäten, wobei hier wieder nur die 8 ausgewählten Benchmarks betrachtet werden. Je höher die Assoziativität desto geringer fällt auch die Reduktion der WCET aus. So bietet eine Assoziativität von 16 nur ein sehr geringes Optimierungspotenzial, da hier auch die Anzahl der Cache-Misses gering ist. Die WCET-Reduktion liegt hier bei 4,5%. Bei einer Assoziativität von 1 wird hingegen eine WCET-Reduktion von 7,6% erzielt.

Auch wenn die Code-Positionierung auf Funktions-Ebene bezüglich der WCET unter Betrachtung aller Benchmarks schlechtere Ergebnisse liefert als die Basisblock-Positionierung, so kann diese bei einer Kombination beider Positionierungsverfahren trotzdem zu einer zusätzlichen WCET-Reduktion beitragen. Im folgenden Abschnitt werden die Ergebnisse bezüglich der Kombination beider Code-Positionierungen präsentiert.

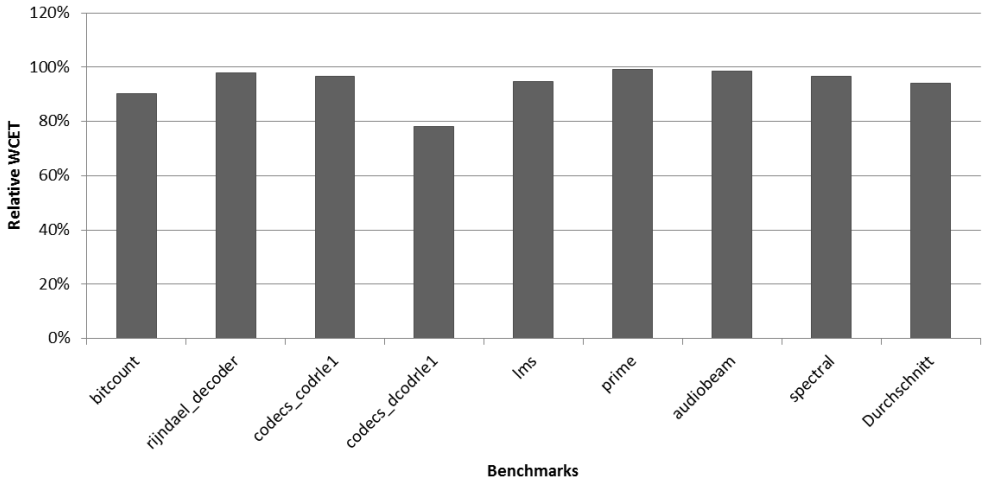

Abbildung 6.10.: Relative WCET-Werte nach der Code-Positionierung auf Funktions-Ebene für die Assoziativität 2.

6.4. Kombination der Code-Positionierungen

Die Cache-bewusste Code-Positionierung, die in dieser Arbeit entworfen wurde beinhaltet sowohl die Code-Positionierung auf Basisblock-Ebene als auch die auf Funktions-Ebene. Bevor die Kombination beider Verfahren evaluiert wird, sollen die Ergebnisse der Code-Positionierungen auf Basisblock- und Funktions-Ebene noch einmal im Vergleich betrachtet werden. In Tabelle 6.3 ist hierfür die erzielte Reduktion der

Tabelle 6.3.: Vergleich der Code-Positionierungen auf Basisblock- und Funktions-Ebene bei einer Assoziativität von 2.

Positionierung	*Cache-Misses*	*MissesSum*	*WCET*
Basisblock	9,8%	10,9%	4,3%
Funktion	3,8%	4,5%	2,3%

Cache-Misses, MissesSum und der WCET dargestellt. Die Werte beziehen sich hier auf den Durchschnitt über alle 21 Benchmarks, wobei eine Assoziativität von 2 verwendet wurde. Die Basisblock-Positionierung erzielt bei einer Cache-Miss-Reduktion von 9,8% und einer MissesSum-Reduktion von 10,9% eine WCET-Reduktion von 4,3%. Wohingegen die Funktions-Positionierung eine WCET-Reduktion von 2,3% erzielt und dabei die Cache-Misses um 3,8% und die MissesSum um 4,5% reduziert.

Über die Kombination beider Verfahren kann die Reduktion der WCET über die Reduktion der Cache-Misses bzw. MissesSum noch weiter gesteigert werden. In diesem Abschnitt wird daher die Kombination der Code-Positionierungsverfahren evaluiert. Dabei wird die Basisblock-Positionierung vor der Funktions-Positionierung ausge-

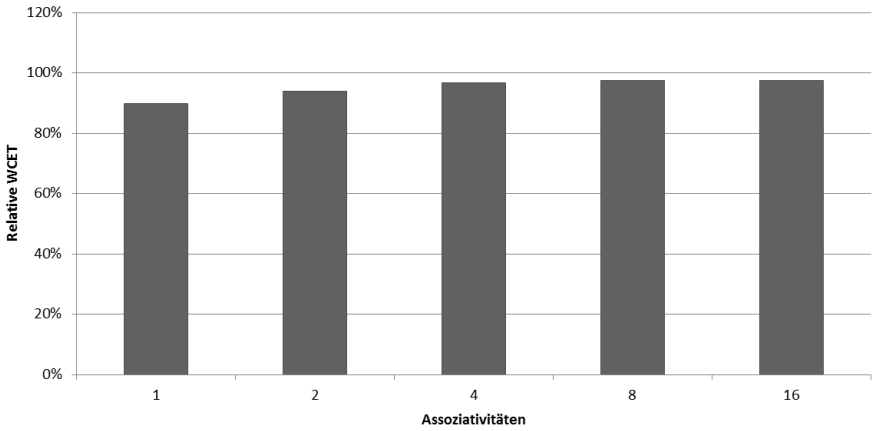

Abbildung 6.11.: Durchschnittswerte der WCET nach der Code-Positionierung auf Funktions-Ebene über alle Assoziativitäten.

führt. Die Reihenfolge hat hier jedoch keinen Einfluss auf die Auswertung, so dass ebenfalls die Funktions-Positionierung zuerst ausgeführt werden könnte. Neben den Werten der Cache-Misses und der MissesSum sowie der WCET werden in diesem Abschnitt auch die Auswirkungen der Optimierung auf die ACET und die CPU-Laufzeit betrachtet. Die Codegröße wird hierbei nicht weiter untersucht, da diese lediglich von der Basisblock-Positionierung beeinflusst wird und bereits im Abschnitt 6.2 beschrieben wurde.

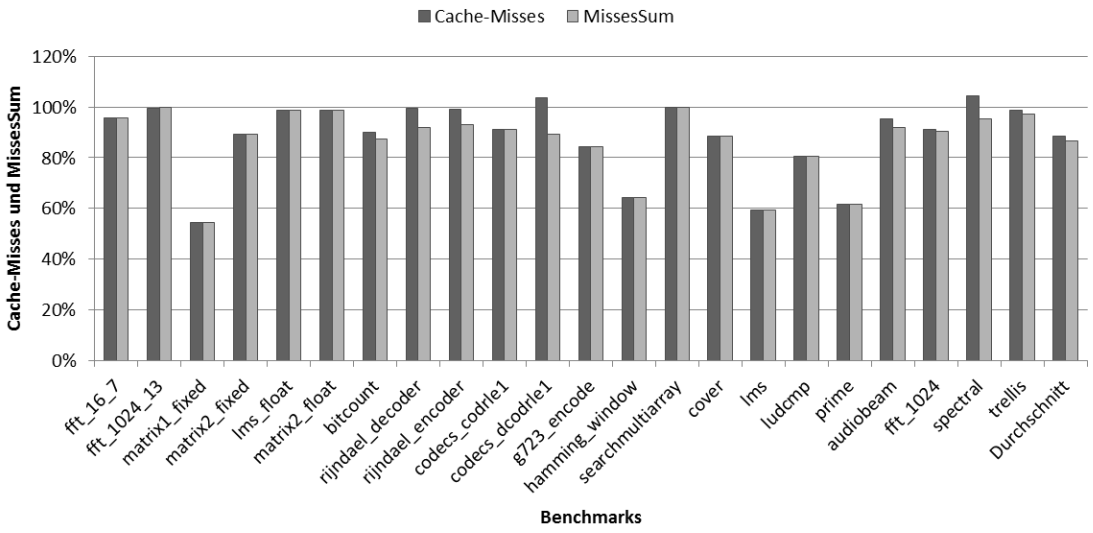

Abbildung 6.12.: Werte der MissesSum nach der Cache-bewussten Code-Positionierung bei einer Assoziativität von 2.

Da die Reduktion der WCET über die Reduktion der Cache-Misses erzielt wird, sollen hier zunächst die Auswertungen bezüglich der Cache-Misses präsentiert werden. In Abbildung 6.12 sind hierzu die Werte der Cache-Misses und der MissesSum, die durch die Cache-bewusste Code-Positionierung erzielt werden, dargestellt. Der Vergleichswert von 100% beschreibt dabei die Anzahl der Cache-Misses und der MissesSum vor der Positionierung. Die Assoziativität ist hier auf 2 konfiguriert. Die Werte werden vor allem durch die Basisblock-Positionierung beeinflusst (siehe Abbildung 6.2), da die Anzahl der Basisblöcke höher als die Anzahl der Funktionen ist und somit bei der Basisblock-Positionierung ein höheres Optimierungspotenzial besteht. Beim Benchmark matrix1_fixed wurde so eine Reduktion der Cache-Misses und der MissesSum von 45,8% erreicht. Im Gegensatz hierzu wurde bei dem Benchmark searchmultiarray keine Reduktion erzielt. Insgesamt konnte durchschnittlich über alle 21 Benchmarks eine Cache-Miss-Reduktion von 11,5% und eine MissesSum-Reduktion von 13,5% erreicht werden.

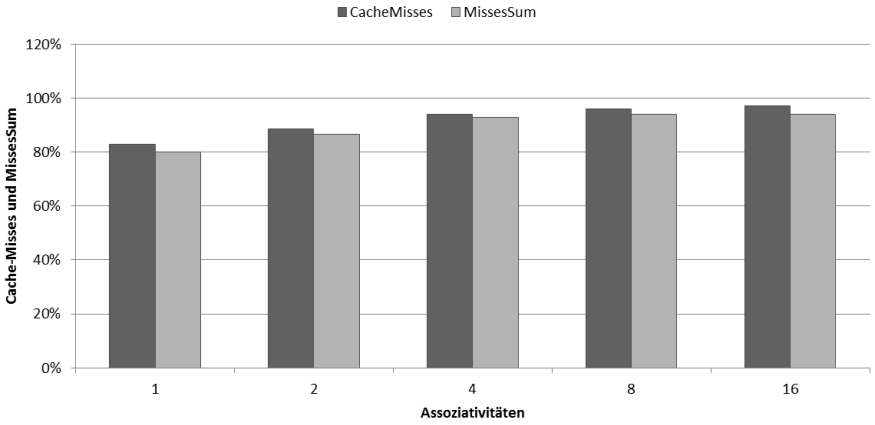

Abbildung 6.13.: Durchschnittswerte der Cache-Misses und MissesSum nach der Cache-bewussten Code-Positionierung über alle Assoziativitäten.

In Abbildung 6.13 sind die Durchschnittswerte der Cache-Misses und MissesSum, die durch die Cache-bewusste Code-Positionierung erzielt werden, über alle Assoziativitäten dargestellt. Bei einer Assoziativität von 16 wurde aufgrund des geringen Konfliktpotenzials nur eine Cache-Miss-Reduktion von 2,7% und eine MissesSum-Reduktion von 6% erzielt. Bei einer Assoziativität von 1 hingegen konnten die Cache-Misses um 17,0% und die MissesSum um 19,9% gesenkt werden. Wie bei der separaten Betrachtung der Funktions- und Basisblock-Positionierung spiegelt sich dieses Verhältnis auch hier in der Reduktion der WCET wieder.

Abbildung 6.14 veranschaulicht die relativen WCET-Werte, die bei einer Assoziativität von 2 durch die Cache-bewusste Code-Positionierung erzeugt wurden. Der

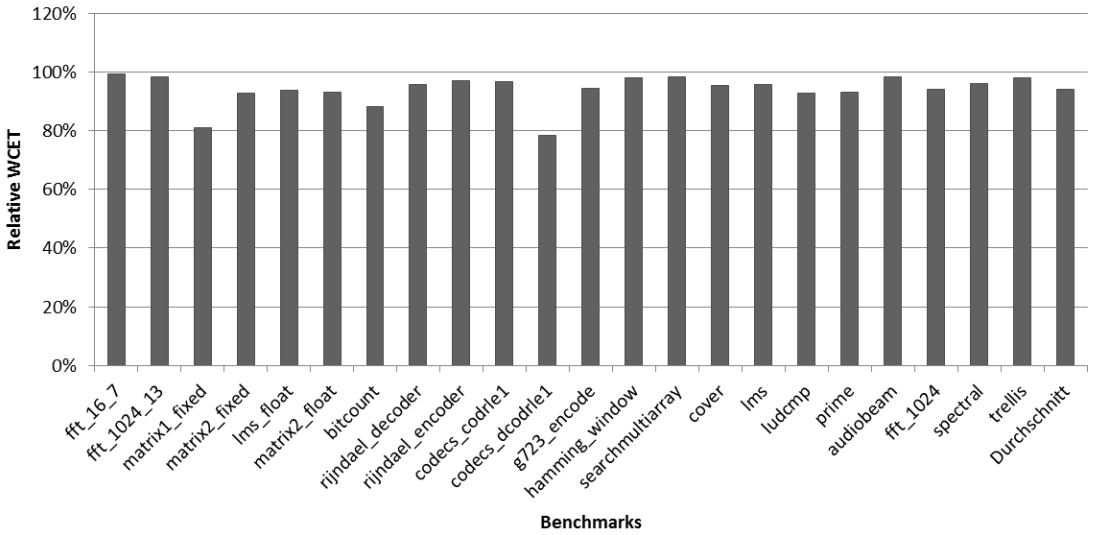

Abbildung 6.14.: WCET-Werte nach der Cache-bewussten Code-Positionierung bei einer Assoziativität von 2.

Wert von 100% steht für den Vergleichswert der WCET ohne die Cache-bewusste Code-Positionierung. Die größte Reduktion wurde hier wie bei der Funktions-Positionierung über den Benchmark codecs_dcodrle1 erreicht und liegt bei 21,8%. Diese Reduktion ist allein durch eine Neuanordnung von zwei im Konflikt stehenden Funktionen entstanden. Somit wird selbst durch eine sehr geringe Anzahl an Neuanordnungen eine hohe Reduktion der WCET erreicht. Der Benchmark searchmultiarray erzielt lediglich eine WCET-Reduktion von 1,7%. Hier wurde eine Neuanordnung von zwei Basisblöcken durchgeführt. Dieser Benchmark führt jedoch nicht wie in Abbildung 6.12 zu sehen zu einer Reduktion der MissesSum, weshalb diese geringe WCET-Reduktion auch auf die Sprungkorrektur zurückgeführt werden kann, welche durch die kontinuierliche Anordnung zweier Basisblöcke eine zuvor benötigte Sprunganweisung entfernt. Durchschnittlich wurde die WCET durch die Cache-bewusste Code-Positionierung um 6,1% reduziert. Somit konnte durch die Kombination beider Positionierungsverfahren eine zusätzliche Optimierung erreicht werden.

In Abbildung 6.15 sind die Durchschnittswerte der WCET über alle Assoziativitäten dargestellt. Hier gilt wieder das Prinzip des abnehmenden Optimierungspotenzials bei steigender Assoziativität. So wird bereits bei einer Assoziativität von 4 nur noch eine WCET-Reduktion von 3,3% erreicht. Bei einer Assoziativität von 1 hingegen wird die WCET um 10,2% gesenkt. Die Kombination der Positionierungsverfahren konnte im Vergleich zum separaten Einsatz der Positionierungen bei allen Assoziativitäten eine zusätzliche Reduktion der WCET erzielen und war somit lohnenswert.

Im Folgenden soll nun noch die ACET, betrachtet werden. Abbildung 6.16 ver-

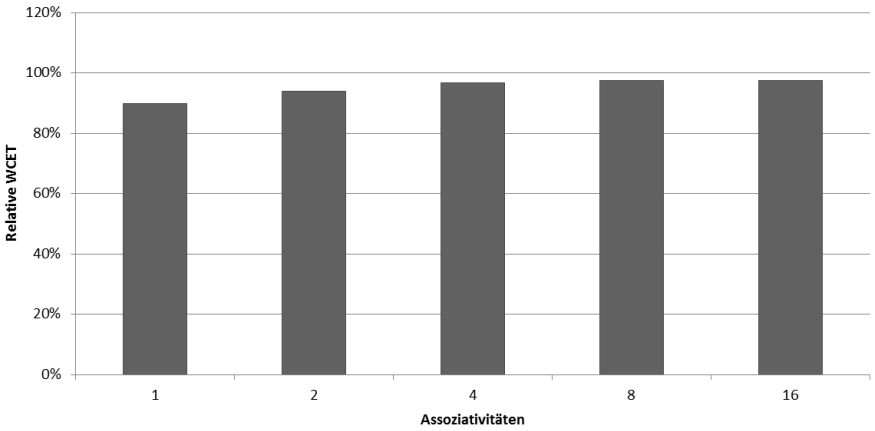

Abbildung 6.15.: Durchschnittswerte der WCET nach der Cache-bewussten Code-Positionierung über alle Assoziativitäten.

anschaulicht hierzu die ACET-Werte, die durch die Cache-bewusste Code-Positionierung bei einer Assoziativität von 2 erzeugt wurden. Die 100% Marke stellt hierbei die relativen ACET-Werte ohne die Cache-bewusste Code-Positionierung dar. So ist zu sehen, dass bzgl. der ACET fast keine Optimierung stattgefunden hat, sondern diese sogar um einen geringen Anteil erhöht wurde. Dies begründet sich darin, dass die Cache-bewusste Code-Positionierung eine WCET-Optimierung ist, welche nur entlang des WCEPs Änderungen vornimmt. So werden vor allem die Cache-Misses der Basisblöcke und Funktionen reduziert, die zum größten Anteil zur WCET beitragen. Durch diese Vorgehensweise kann sich die ACET verschlechtern, da z. B. die Basisblöcke, die zu einer geringen ACET beitragen nun mehr Cache-Misses erzeugen als zuvor. Die Anzahl der Cache-Misses verlagert sich somit. Hier erzielen lediglich die Benchmarks prime und searchmultiarray eine Reduktion der ACET. So wird eine maximale ACET-Reduktion von 7,0% über den Benchmark prime erzielt. Der Benchmark cover hingegen führte zu einer Erhöhung der ACET von 2,2%. Im Durchschnitt über alle Benchmarks wurde die ACET um 0,1% erhöht. Die Cache-bewusste Code-Positionierung hat somit kaum Einfluss auf die ACET. Dies spiegelt sich auch in den anderen Assoziativitäten wieder. Im Durchschnitt über alle Assoziativitäten wurde die ACET um 0,4% erhöht.

Abschließend soll nun noch die CPU-Laufzeit in Sekunden der Cache-bewussten Code-Positionierung betrachtet werden, welche über alle Benchmarks in Abbildung 6.17 dargestellt ist. Hier wurde wieder eine Assoziativität von 2 konfiguriert. Die Laufzeiten der einzelnen Benchmarks variieren je nach Anzahl der durchgeführten Positionierungen von Basisblöcken und Funktionen.

Bei jeder Positionierung wird ein Aufruf an das WCET-Analysewerkzeug durchgeführt, um die Instabilität des WCEPs zu berücksichtigen. Damit wird die meiste

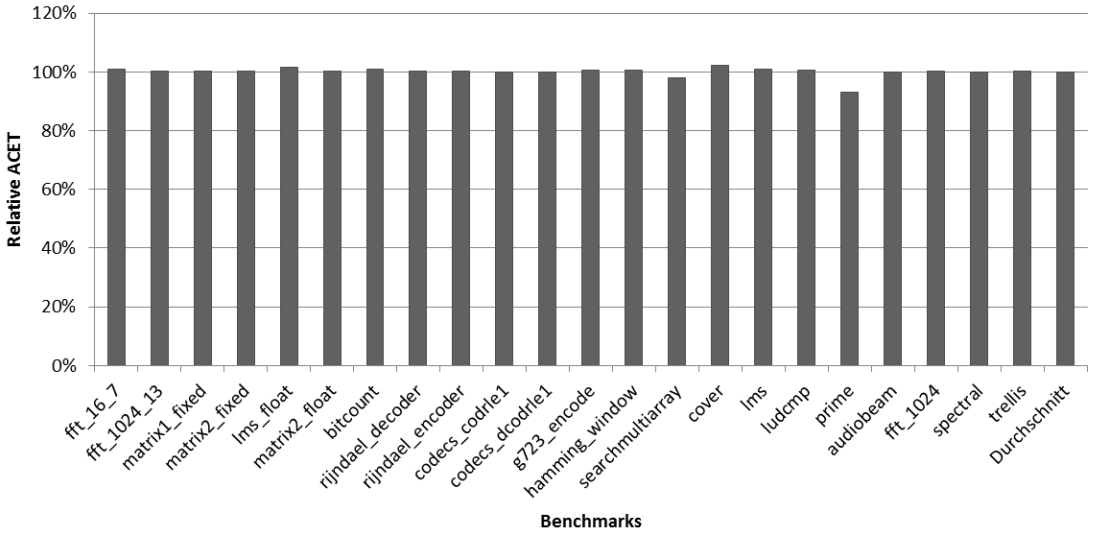

Abbildung 6.16.: Relative ACET-Werte nach der Cache-bewussten Code-Positionierung für die Assoziativität 2.

Zeit nicht vom Algorithmus der Cache-bewussten Code-Positionierung benötigt sondern von den WCET-Analysen. Beim Benchmark cover werden insgesamt 15 WCET-Analysen benötigt. Somit entsteht eine CPU-Laufzeit von 13 Minuten. Die geringste CPU-Laufzeit wird vom Benchmark lms_float benötigt und beträgt 3 Sekunden, bei 2 WCET-Analysen. Die CPU-Laufzeit, die für eine WCET-Analyse benötigt wird, kann je nach Komplexität des Benchmarks stark variieren. Im Durchschnitt über alle Benchmarks beträgt die CPU-Laufzeit der Cache-bewusste Code-Positionierung 2 Minuten und 30 Sekunden.

Abbildung 6.18 veranschaulicht die Durchschnittswerte der CPU-Laufzeiten über alle Assoziativitäten. Im Allgemeinen sinkt die CPU-Laufzeit mit steigender Assoziativität, da hier aufgrund des geringer werdenden Optimierungspotenzials weniger Positionierungen durchgeführt werden. Eine Ausnahme stellt hierbei die Assoziativität von 1 dar, hier beträgt die durchschnittliche CPU-Laufzeit 138 Sekunden. Da die Anzahl der WCET-Analysen bei der Assoziativität 2 nur um 10 ansteigt, die CPU-Laufzeit jedoch 151 Sekunden beträgt, wird davon ausgegangen, dass aiT für eine WCET-Analyse von mengenassoziativen Caches aufgrund des komplexeren Aufbaus mehr Zeit benötigt. Bei einer Assoziativität von 16 beträgt die CPU-Laufzeit lediglich 107 Sekunden, da hier nur wenige Positionierungen und damit WCET-Analysen stattfinden.

Nachdem die Ergebnisse der entworfenen WCET-Optimierung präsentiert wurden, folgen im nächsten Kapitel eine zusammenfassende Darstellung der Arbeit und ein Ausblick über Erweiterungsmöglichkeiten und Verbesserungen.

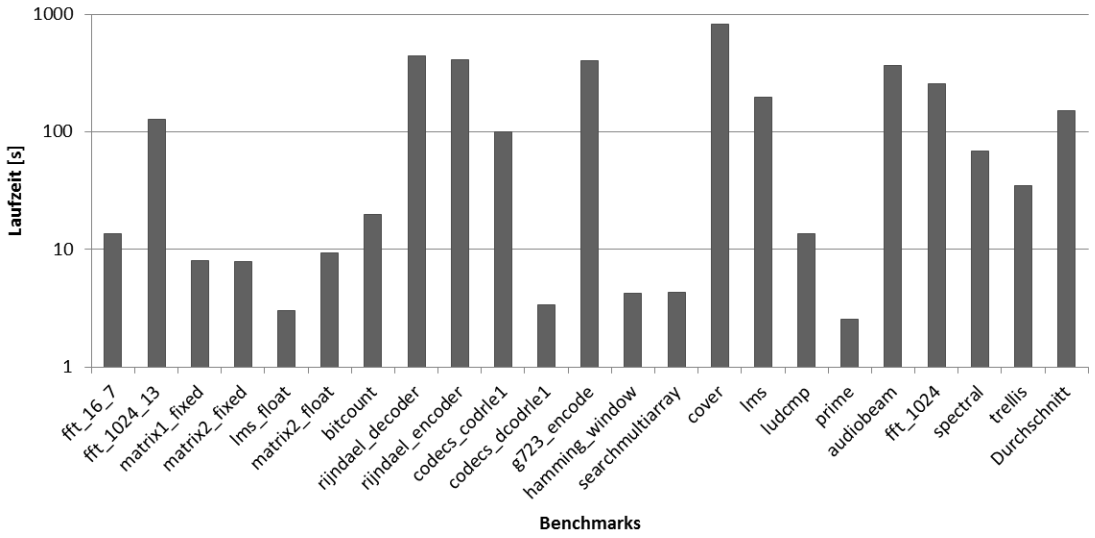

Abbildung 6.17.: CPU-Laufzeiten der Cache-bewussten Code-Positionierung für die Assoziativität 2.

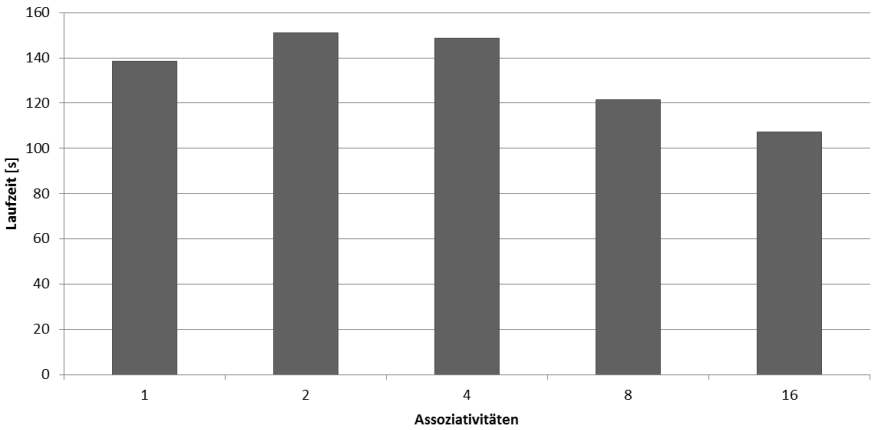

Abbildung 6.18.: Durchschnittswerte der CPU-Laufzeiten bei der Cache-bewussten Code-Positionierung über alle Assoziativitäten.

7. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine Cache-bewusste Code-Positionierung zur Reduktion der WCET entworfen, welche sowohl für direkt-abgebildete als auch für mengenassoziative Intruktionscaches eingesetzt werden kann. In diesem Kapitel werden in Abschnitt 7.1 die Ergebnisse der Arbeit zusammengefasst. In Abschnitt 7.2 wird ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben.

7.1. Zusammenfassung

In Kapitel 1 wurde der Themenbereich dieser Arbeit motiviert. Hier wurde die Relevanz der WCET für eingebettete Echtzeitsysteme verdeutlicht und die Problematiken beim Einsatz von Cache-Speichern dargestellt. Daraus ergab sich auch die Zielsetzung dieser Arbeit, welche über eine Code-Positionierung die Anzahl der Cache-Misses reduzieren soll und so die Vorhersagbarkeit des Cacheverhaltens bzgl. der WCET optimiert. Abschließend fand eine Abgrenzung zu den verwandten Arbeiten statt.

Die Grundlagen zu den Themengebieten dieser Arbeit wurden in Kapitel 2 vermittelt. Dabei sind die Methoden der WCET-Analyse sowie die Funktionsweise des verwendeten WCET-Analysewerkzeugs aiT vorgestellt worden. Zudem Erfolgte eine Beschreibung der Analyse-Methoden und des Aufbaus von Cache-Speichern, um die Vorgehensweise einer Cache-bewussten Code-Positionierung zu verdeutlichen.

Die in dieser Arbeit entworfene WCET-Optimierung wurde in den WCET-optimierenden Compiler WCC des Lehrstuhls 12 für eingebettete Systeme integriert. Daher erfolgte in Kapitel 3 eine Darstellung des Compiler-Frameworks, wobei besonders auf den Aufbau der internen Zwischendarstellung der ICD-LLIR eingegangen wurde, da diese als Grundlage der Cache-bewussten Code-Positionierung diente. Abschließend wurde die vom WCC unterstützte TriCore-Plattform des TC1797 beschrieben, welche als Zielarchitektur in dieser Arbeit zum Einsatz kam.

Kapitel 4 befasste sich mit dem Schwerpunkt dieser Arbeit. Hier wurden der Aufbau und die Methoden der entworfenen Cache-Analyse vorgestellt. Diese dient der Cache-bewussten Code-Positionierung als Grundlage und stellt damit Informationen über das Verhalten von Speicherobjekten zur Verfügung. So wurde die Vorgehensweise der Modellierung von Cache-Konflikten beschrieben, welche anhand von Graphen

die Informationen über Konflikte zwischen Basisblöcken und zwischen Funktionen eines Programms zur Verfügung stellen.

In Kapitel 5 wurde die Funktionsweise der Code-Positionierung dieser Arbeit vorgestellt. Diese findet auf zwei Ebenen statt, daher wurde hier sowohl der Algorithmus zur Positionierung von Basisblöcken als auch der zur Positionierung von Funktionen erläutert.

Um die Güte der Cache-bewussten Code-Positionierung zu evaluieren, wurden anhand verschiedener Benchmarks Auswertungen vorgenommen, die in Kapitel 6 präsentiert wurden. Da die Code-Positionierung über eine Reduktion der Cache-Misses das Verhalten des Instruktionscaches verbessert und damit auch zur Reduktion der WCET führen soll, bezogen sich die Auswertungen in erster Linie auf die Reduktion der Cache-Misses und der WCET. Darüber hinaus wurden ebenfalls die Ergebnisse bzgl. der Codegröße, der ACET und der CPU-Laufzeit der WCET-Optimierung vorgestellt.

Ziel dieser Arbeit war der Entwurf einer Cache-bewussten Code-Positionierung zur Reduktion der WCET, die sowohl Basisblöcke als auch Funktionen betrachtet. Durch diese WCET-Optimierung sollte das Verhalten des Instruktionscaches durch die Reduktion der Cache-Misses verbessert werden und somit die WCET-Überabschätzung beim Einsatz dieser Speicher in eingebetteten Echtzeitsystemen optimiert werden. Hierfür wurde der Entwurf in zwei Aufgabenbereiche eingeteilt. Um die Cache-Misses zu reduzieren war es zunächst notwendig, zu ermitteln welche Basisblöcke bzw. Funktionen miteinander im Konflikt stehen. Diese Information wurde anhand der in Kapitel 4 beschriebenen Cache-Analyse ermittelt. Der zweite Aufgabenbereich umfasste den Algorithmus zur Code-Positionierung. Auf Basis der Informationen der Cache-Analyse wurden hierbei im Konflikt stehende Basisblöcke und Funktionen kontinuierlich im Hauptspeicher angeordnet, um die Anzahl der zugehörigen Cache-Misses zu reduzieren.

Über die Cache-bewusste Code-Positionierung konnte eine Reduktion der Cache-Misses erzielt werden und somit auch eine Reduktion der WCET. Insgesamt konnte eine WCET-Reduktion von 10,2% bei direkt-abgebildeten Caches erzielt werden. Bei mengenassoziativen Caches (Assoziativitäten 2 bis 16) wurde eine WCET-Reduktion von 2,6% bei einer Assoziativität von 16, bis hin zu einer Reduktion von 6,1% bei einer Assoziativität von 2 erzielt. Somit konnten die in Abschnitt 1.2 beschriebenen Zielsetzungen der Arbeit erreicht werden.

7.2. Ausblick

Ein interessanter Punkt zur Erweiterung der Cache-bewussten Code-Positionierung dieser Arbeit wäre eine detailliertere Form der in Abschnitt 4.3.2 vorgestellten May-Analyse. Diese modelliert die Cachezustände anhand eines Kontrollflussgraphen, der sich über die Basisblöcke einer Funktion erstreckt. So können die Cacheinhalte vor Ausführung eines Basisblocks bestimmt und damit Informationen über Konflikte zur Verfügung gestellt werden. Für eine detailliertere May-Analyse könnte ein Graph aufgebaut werden, der nicht nur den Kontrollfluss von Basisblöcken innerhalb einer Funktion, sondern auch den Kontrollfluss zwischen den Basisblöcken aller Funktionen beschreibt. Somit könnte die May-Analyse auch die funktionsübergreifenden Cachezustände modellieren und damit detailliertere Informationen über Konflikte liefern. Die Code-Positionierung auf Funktions-Ebene könnte somit verfeinert und das Optimierungspotenzial bzgl. der WCET gesteigert werden.

Das WCET-Analysewerkzeug aiT liefert zu einem Basisblock lediglich die Gesamtanzahl der Cache-Misses. Diese besteht jedoch nicht nur aus Konflikt-Misses sondern zusätzlich aus Cold Start-Misses. Ein Basisblock kann mit mehreren anderen Basisblöcken im Konflikt stehen, die zu verschiedenen Anteilen zur Gesamtanzahl der Konflikt-Misses dieses Basisblocks beitragen. Jedoch liefert aiT keine Informationen über diese Anteile. In dieser Arbeit wurden daher weitere Komponenten mit der Gesamtanzahl der Cache-Misses verrechnet, um diesen Anteilen zumindest nahe zu kommen (siehe Abschnitt 4.2.2). Wenn die genaue Anzahl der Konflikt-Misses, die von einem Basisblock verursacht werden, zur Verfügung stünden, könnte das Optimierungspotenzial der Cache-bewussten Code-Positionierung noch weiter angehoben werden. Zudem könnte auch lediglich die Angabe über die Gesamtanzahl der Konflikt-Misses ohne die Anzahl der Cold Start-Misses, zur Verbesserung der entworfenen Code-Positionierung beitragen.

Neben der Auswertung bzgl. der WCET wäre ebenfalls die Auswertung bzgl. des Energieverbrauches interessant, da der Energieverbrauch in eingebetteten Systemen ebenfalls eine wichtige Rolle spielt.

A. Benchmarks

Tabelle A.1.: Verwendete Benchmarks

Benchmark	Basisblöcke	Funktionen	Codegröße
DSPstone_fixed_point			
fft_16_7	39	6	936
fft_1024_13	41	6	984
matrix1_fixed	16	2	496
matrix2_fixed	16	2	506
DSPstone_floating_point			
lms_float	10	2	894
matrix2_float	16	2	826
MiBench			
bitcount	57	9	950
rijndael_decoder	42	2	10656
rijndael_encoder	53	3	10370
misc			
codecs_codrle1	53	4	844
codecs_dcodrle1	81	7	1340
g723_encode	380	28	5244
hamming_window	7	1	218
searchmultiarray	13	2	184
MRTC			
cover	421	4	2780
lms	58	7	1488
ludcmp	39	2	1392
prime	14	3	160
StreamIt			
audiobeam	124	16	4950
UTDSP			
fft_1024	13	2	1110
spectral	169	10	5032
trellis	72	11	3114

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Abs11] AbsInt Angewandte Informatik GmbH: worst-case execution time analyzers. http://www.absint.com/ait, Januar 2011

[ACH01] Cilio, A. G. M., Corporaal, H.: Code Positioning for VLIW Architectures. In Proceedings of the 9th International Conference on High-Performance Computing and Networking (HPCN Europe), Springer-Verlag, 332-343, 2001

[CC77] Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL), Los Angeles, California, ACM, 238-252, 1977.

[CP00] Colin, A., Puaut I.: Worst Case Execution Time Analysis for a Processor with Branch Prediction. Real-Time Systems, 18(2/3), 249-274, 2000

[Fer97] Ferdinand, C.: Cache Behavior Prediction for Real-Time Systems. PhD thesis, Saarland University, 1997.

[FL10] Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. International Journal of Time-Critical Computing Systems (Real-Time Systems), Springer, 46(2), 251-300, 2010

[FMW97] Ferdinand, C., Martin F., Wilhelm, R.:Applying Compiler Techniques to Cache Behavior Prediction. ACM SIGPLAN Workshop Languages, Compilers, and Tools for Real-Time System (LCTRTS), 37-46, 1997

[FW99] Ferdinand, C., Wilhelm, R.: Efficient and Precise Cache Behavior Prediction for Real-Time Systems. Real-Time Systems, 17(2-3), 131-181, 1999

[GA07] Gebhard, G., Altmeyer, S.: Optimal task placement to improve cache performance. In Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT), Salzburg, Austria, ACM, 259-268, 2007

[GRB04] Guillon, C., Rastello, F., Bidault, T., Bouchez, F.: Procedure placement using temporal-ordering information: Dealing with code size expansion. In Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems (CASES), ACM, 268-279, 2004

[GRE01] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., Brown, R. B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE International Workshop on Workload Characterization (WWC), 2001, S. 3-14

[HP06] Hennessy, J. L., Patterson, D. A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 2006

[HS89] Hill, M. D.,Smith, A. J.: Evaluating associativity in CPU caches. In Readings in computer architecture, Morgan Kaufmann Publishers Inc., 82-100, 2000

[ICD11] ICD-C Compiler Framework Developer Manual. ICD-Informatik Centrum Dortmund, http://www.icd.de/es/icd-c, 2011

[Inf04] Infineon Technologies AG: TriCore 1 Guidelines of Cache Management V1.1., 2004

[Inf07] Infineon Technologies AG: TC1796 32-Bit Single-Chip Microcontroller User's Manual, V2.0, 2007

[Inf09] Infineon Technologies AG: TC1797 32-Bit Single-Chip Microcontroller User's Manual, V1.1, 2009

[Kop00] Kopetz, H.: Software engineering for real-time: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE), Limerick, Ireland, ACM, 201-211, 2000

[LFM08] Lokuciejewski, P., Falk, H., Marwedel, P.: WCET-driven Cache-based Procedure Positioning Optimizations. In Proceedings of the 2008 Euromicro Conference on Real-Time Systems (ECRTS), IEEE Computer Society, 321-330, 2008

[KW03] Kowarschik, M., Weiß, C.: An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms.In Algorithms for Memory Hierarchies. Springer, 2003.

[LIR11] ICD Low Level Intermediate Representation backend infrastructure (LLIR) Developer Manual. ICD-Informatik Centrum Dortmund, 2011

[LM95] Li, Y.-T. S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In Proceedings of the 32nd annual ACM/IEEE Design Automation Conference (DAC), San Francisco, California, United States, ACM, 456-461, 1995

[LMi10] Liang, Y., Mitra, T.: Improved procedure placement for set associative caches. In Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems (CASES). Scottsdale, Arizona, USA, ACM, 147-156, 2010

[LM10] Lokuciejewski, P., Marwedel, P.: Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems. Springer-Verlag, 2010

[LPM97] Lee, C., Potkonjak, M., Mangione-Smith, W. H.: MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In Proceedings of the 30th Annual ACM/IEEE international Symposium on Microarchitecture (MICRO), 330-335, 1997

[Mar10] Marwedel, P.: Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems. Springer-Verlag, 2010

[Mae11] Mälardalen WCET Research Group: MRTC Benchmark Suite. http://www.mrtc.mdh.se/projects/wcet, Januar 2011

[MMH01] Memik, G., Mangione-Smith, W. H., Hu, W.: NetBench: A Benchmarking Suite for Network Processors. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD). Piscataway, USA, 2001

[Muc97] Muchnick, S. S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[Mue00] Mueller, F.: Timing Analysis for Instruction Caches. Real-Time Systems, 18(2/3), 217-247, 2000

[PH90] Pettis, K., Hansen, R. C.: Profile guided code positioning. In Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI). ACM, 16-27, 1990

[PLM10] Plazar,S., Lokuciejewski, P., Marwedel P.: WCET-driven Cache-aware Memory Content Selection. In Proceedings of The 13th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC), Carmona, Seville, 107-114, 2010

[RDB07] Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Systems, 37(2), 99-122, 2007

[RF08] Reinhold, H., Ferdinand, C.: Worst-Case Execution Time Prediction by Static Program Analysis, AbsInt Angewandte Informatik GmbH, White paper, 2008

[SEE01] Stappert, F., Ermedahl A., Engblom, J.: Efficient longest executable path search for programs with complex flows and pipeline effects. In Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems (CASES), Atlanta, Georgia, USA ACM, 132-140, 2001

[Smi82] Smith, A. J.: Cache Memories. ACM Computing Surveys (CSUR), ACM, 14(3), 473-530, 1982

[Str11] Stream It Entwickler: StreamIt Benchmark Suite. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml, Januar 2011

[UTD11] UTDSP Benchmark Suite: http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html, Januar 2011

[VM07] Verma, M., Marwedel, P.: Advanced Memory Optimization Techniques for Low-Power Embedded Processors. Springer, 2007

[VWM04] Verma, M., Wehmeyer, L., Marwedel, P.: Cache-Aware Scratchpad Allocation Algorithm. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2 (DATE), IEEE Computer Society, 2004

[WEE08] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem-overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS), 7(3), 36:1-36:53, 2008

[Weg05] Wegener, I.: Theoretische Informatik - eine algorithmenorientierte Einführung. 3. Auflage, Teubner Verlag, 21-26, 2005

[WM04] Wehmeyer, L., Marwedel, P.: Influence of Onchip Scratchpad Memories on WCET prediction. In Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis, Catania, Sicily, Italy, 2004

[WM95] Wulf, W., McKee, S.: Hitting the memory wall: Implications of the obvious. ACM Computer Architecture News, 23(1), 20-24, 1995

[ZMM94] Živojnović, C. S., Martinez, J., Meyr, H.: DSPStone: A DSP-oriented Benchmarking Methodology. In Proceedings of the International Conference on Signal Processing and Technology (ICSPAT), Dallas, USA, 1994

[ZWH05] Zhao, W., Whalley, D., Healy, C., Mueller, F.: Improving WCET by applying a WC code-positioning optimization. ACM Transactions on Architecture and Code Optimization (TACO), ACM, 2(4), 335-365, 2005