TECHNISCHE UNIVERSITÄT
D O R T M U N D
FAKULTÄT FÜR INFORMATIK

technische universität
dortmund

Jens Möllmer

WCET Optimierung unter Beachtung der Speicherhierarchie

Bachelorarbeit

2. August 2011

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik XII

Gutachter:
Prof. Dr. Peter Marwedel
Dipl.-Inf. Sascha Plazar

GERMANY · D-44221 DORTMUND

Inhaltsverzeichnis

1 Einleitung

Im Jahr 1991 wurde vorausgesagt, dass sich die PC-Ära dem Ende neigt und sich daraus eine neue Ära entwickelt, in der ubiquitäre Systeme im Vordergrund stehen [Wei91]. Ubiquitäre Systeme sind in die Umgebung eingebettet und sie werden nicht mehr als Computer wahrgenommen. Dass der Trend in diese Richtung geht, zeigt die Verbreitung von eingebetteten Systemen, die meist als technische Grundlage für ubiquitäre Systeme dienen. [Mar07]

In einer Studie von BITKOM wurde der Markt der eingebetteten Systeme analysiert [Bit10]. Nach dieser hat der Markt der eingebetteten Systeme seit Jahren eine Zuwachsrate von bis zu 8% pro Jahr. Es wird auch erwähnt, dass die Hersteller davon ausgehen, dass die Preise für diese Produkte fallen werden, was der Verbreitung zugute kommen wird, da sich preiswerte Produkte schneller verbreiten. Vor allem der Mobiltelefon-Markt boomt. Schon im Jahr 2006 gab es in Deutschland mehr Mobiltelefone als Einwohner [dsl06]. Dies ist deshalb interessant, da ein Mobiltelefon auch als ein eingebettetes System gilt.

Als weiteres Paradebeispiel für eingebettete Systeme kann das Auto genannt werden: hier laufen viele Anwendungen, die das Autofahren sicherer und angenehmer gestalten sollen. Hierzu zählen zum Beispiel der Airbag, der vielen Fahrern das Leben rettet, der Fensterheber-Einklemmschutz oder der Tempomat, welche durch eingebettete Systeme gesteuert werden. Diese Anwendungen können zeitkritisch sein und unterliegen in diesem Fall Zeitschranken, in denen sie reagieren müssen. Bei diesen Echtzeitsystemen wird zwischen zwei Arten unterschieden: harte und weiche Echtzeitsysteme. Bei weichen Echtzeitsystemen existieren zwar Zeitschranken, die eingehalten werden sollen, bei denen ein Überschreiten aber unkritisch ist. Im Gegensatz dazu wird bei harten Echtzeitsystemen das Nicht-Einhalten der Zeitschranken als Fehler gewertet. Harte Zeitschranken müssen unbedingt eingehalten werden, da dies sonst zu Katastrophen führen kann (z.B. Airbag löst nicht rechtzeitig aus). Alle genannten Beispiele aus dem Auto sind also sogenannte *harte Echtzeitsysteme*. Der Airbag muss also in einer gewissen

Zeit auslösen und der Fensterheber muss rechtzeitig stoppen, um Unfälle zu vermeiden. Als Beispiel für ein weiches Echtzeitsystem können die Multimediaanwendungen in einem Auto genannt werden. Um zu überprüfen, ob die Zeitschranken eingehalten werden, wird die maximale Ausführungszeit (engl.: "Worst Case Execution Time", kurz: WCET) der Programme bestimmt, die auf solchen Systemen laufen.

Mithilfe der WCET kann also überprüft werden, ob ein System den zeitlichen Anforderungen entspricht. Hierbei ist es wichtig, dass die WCET möglichst genau bestimmt wird. Eine Unterabschätzung darf nicht vorkommen, da dann nicht gewährleistet ist, dass die Zeitschranken eingehalten werden. Eine zu starke Überabschätzung kann beim Entwurf eines Systems aufgrund der dadurch erhöhten Hardwareanforderungen zu unnötig hohen Systemkosten führen. Deshalb wird versucht die WCET von Programmen zu senken, um damit die Anforderungen an die Hardware des Systems zu reduzieren. Dies geschieht mit speziellen Optimierungen, welche auf verschiedene Weise dafür sorgen, dass die WCET sinkt. Zum Beispiel verschieben Speicheroptimierungen Programmteile in schnellere Speicher. Einige solcher Optimierungen wurden am Lehrstuhl 12 der TU Dortmund entwickelt, darunter auch drei Speicheroptimierungen (Cache Partitioning [PLM09],Scratchpad Memory Allocation [FK09], Memory Content Selection [PLM10]), die in dieser Arbeit auf ihre Zusammenarbeit überprüft werden sollen.

Der Rest des Kapitels ist wie folgt aufgebaut: In Kapitel 1.1 wird dargelegt, wieso das Thema von dieser Bachelorarbeit wichtig ist. Im Anschluss daran wird in 1.2 auf verwandte Arbeiten eingegangen und in 1.3 der Aufbau der Arbeit erläutert.

1.1 Motivation

Wie in der Einleitung erwähnt, befasst sich diese Arbeit mit drei Speicheroptimierungen (Scratchpad Memory Allocation, Memory Content Selection, Cache Partitioning). Jede Optimierung wurde einzeln entwickelt, wodurch sie nicht aufeinander abgestimmt sind. Das heißt, dass jede dieser Optimierungen die Änderungen der anderen Optimierungen nicht beachteten und sie sich gegebenenfalls gegenseitig behindern. Die Frage ist nun, inwieweit die WCET weiter sinkt, wenn zusätzlich zu einer Speicheroptimierung noch eine weitere Speicheroptimierung ausgeführt wird. Generell könnte angenommen werden, dass mehrere Optimierungen zusammen immer eine Verbesserung erzeugen.

Da die Optimierungen sich aber gegenseitig behindern könnten, muss dies nicht der Fall sein. Zusätzlich zu dieser Problematik könnte die Reihenfolge der Optimierungen eine Rolle spielen: Das Cache Partitioning beschleunigt einen Programmteil in dem es diesen in Partitionen in den cached Speicherbereich verschiebt. Bei der Scratchpad Memory Allocation wird der Programmteil in den sogenannten Scratchpad Speicher verschoben. Es kann nun passieren, dass ein Programmteil besser von der Scratchpad Memory Allocation verschoben wird, da dann die WCET stärker sinkt, jedoch durch die Reihenfolge der Optimierungen der Programmteil vom Cache Partitioning verschoben wird. Daher muss genau auf die Reihenfolge der Optimeirungen geachtet werden. Deshalb ist es interessant, inwiefern die drei Speicheroptimierungen zusammen benutzt werden können.

1.2 Verwandte Arbeiten

In [CA00], [ME04] ,[Mue95], werden zwar Techniken vorgestellt, die ein Cache Partitioning in Hardware oder in Software nutzen, jedoch beachten diese die WCET entweder gar nicht oder zumindest nicht als Hauptmerkmal. Die Idee des Cache Partitioning ist, durch Partitionierung des Caches die Vorhersagbarkeit von Cache Hits/Misses zu verbessern, wenn mehrere Tasks im System aktiv sind. Dazu wird jedem Task ein bestimmter Bereich des Caches zugewiesen. Jeder Task wird dann so über den Adressraum verteilt, so dass falls Teile von diesem Task in den Cache eingebracht werden, diese genau in dem zugewiesenen Bereich geladen werden. Dadurch wird sichergestellt, dass die Instruktionen, die von einem Task in den Cache geladen wurden, auch nach Kontextwechseln im Cache liegen. Die WCET wird bei der Arbeit in [CA00] nicht beachtet. Hier wird ein Cache Partitioning über sog. Columns realisiert. Der Cache arbeitet hierbei wie ein n-fach mengen-assoziativer Cache außer das die Platzierungs- und Ersetzungsstrategie geändert wurde. Zu einer Column werden jeweils die gleichen Cache-Zeilen von allen Cache-Sets zusammengefasst. Hierbei wird nur bestimmter Code in bestimmten Columns abgelegt.
In der Arbeit [ME04] wird das Cache Partitioning über einen n-fach mengen-assoziativen Cache erreicht. Hierbei bekommt jeder Task eine bestimmte Anzahl von Cache-Zeilen von jedem Cache-Set zugewiesen, in den nur Code von dem jeweiligen Task eingebracht werden kann. Dafür müssen auch Hardwareänderungen bei der Ersetzung von Cache-Zeilen vorgenommen werden, so dass hier nur die entsprechenden Cache-Zeilen des aktuellen Tasks überprüft werden.

In [Mue95] werden die Grundlagen für ein in software-basierendes Cache Partitioning mit Compiler Unterstützung dargestellt. Die dadurch entstandenen Ergebnisse sind, was die WCET betrifft, nicht optimal.
In [PLM09] haben Plazar et al. das WCET-Aware Software Based Cache Partitioning for Multi-Task Real-Time Systems vorgestellt, in der der Instruktion-Cache partitioniert wird. Um zu bestimmen, wie groß die Partitionsgröße für einen Task gewählt werden muss, wird die WCET Analyse genutzt. Für jeden Task wird für verschiedene Partitionsgrößen die WCET bestimmt und mit diesen Daten wird ein ganzzahlig lineares Programm (engl.: "Integer Linear Program", kurz: ILP) aufgestellt. Die Lösungen des ILP stellen dann die optimale Cachegröße für jeden Task dar, so dass die WCET des Gesamtsystems minimal wird.

Die WCET-driven Cache-aware Memory Content Selection wurde von Plazar et al. in [PLM10] vorgestellt. Diese Optimierung arbeitet mit cached und non-cached Speicherbereichen und berechnet für jede Funktion, ob sie in den cached oder non-cached Bereich verschoben werden soll, um so Programmcode zu beschleunigen. Vor dieser Optimierung sind alle Funktionen im non-cached Speicherbereich. Nach der Optimierung sind meist wenige Funktionen im cached Speicherbereich, was zur Folge hat, dass auch weniger Funktionen im Cache vorgehalten werden müssen, wodurch die Cache-Miss-Rate und dadurch auch die WCET sinkt.
In [SM08] wurde eine Arbeit vorgestellt, die mithilfe von Partitionieren und Lockdown des Caches ein besser vorhersagbares Systemverhalten liefert, wobei am Ende die WCET Reduktion analysiert wurde. Jedoch wird bei der Optimierung die WCET nicht herangezogen, sondern der Cache so verteilt, dass die Nutzung des Systems minimal wird. Deswegen sind die Ergebnisse in Bezug auf die WCET nicht optimal und können noch verbessert werden.

In [FK09] stellen Falk et al. die Optimal Static WCET-aware Scratchpad Allocation of Program Code vor. Bei dieser wird der sogenannte Scratchpad Speicher genutzt, um Programmcode zu beschleunigen. Der Scratchpad Speicher ist ein kleiner schneller Speicher, der oft nah an die CPU angebunden ist und über Software kontrolliert wird. Durch Verschieben von häufig genutzten Programmteilen in den Scratchpad Speicher wird versucht, die WCET zu senken. Das Besondere an dieser Variante ist die Beachtung des Pfades im Kontrollflussgraphen mit der größten Ausführungszeit (Worst Case Execution Path, kurz: WCEP) und das ILP gestützte Auswahlverfahren der zu verschiebenden Basisblöcke. Hierbei wird die Veränderung des WCEPs während

der Optimierung beachtet und auf diese entsprechend reagiert. Durch das ILP wird sichergestellt, dass immer die minimal mögliche WCET erreicht wird.
Zu diesem Thema wurden schon andere Arbeiten vorgestellt, unter anderem in [VS05], wo auch ein Ansatz gewählt wurde, der mithilfe eines ILPs die WCET reduzieren sollte. Jedoch kann hier kein Programmcode in den SPM verschoben werden sondern nur Daten und es eignet sich nicht zur Anwendung an realen Programmen, da einige Einschränkungen gemacht wurden. Es wird hierbei nicht darauf geachtet, ob der entsprechende Pfad im Kontrollflussgraphen überhaupt erreicht werden kann, womit es dann passieren kann das Daten allokiert werden, die nicht gebraucht werden.
Ein anderer Ansatz wird in [WM04] und [WM05] verfolgt, wo die Auswahl der zu verschiebenden Basisblöcke so getroffen wird, dass der Energieverbrauch minimiert wird. Für diese Auswahl wird anschließend die daraus resultierende Reduktion der WCET untersucht.

Die vorgestellten WCET-Optimierungen [PLM09], [FK09] und [PLM10] werden im **W**CET-aware **C** **C**ompiler (WCC) genutzt, der an dem Lehrstuhl 12 der TU Dortmund konzipiert und entwickelt wurde [FLT06]. Das Besondere an diesem Compiler ist, dass ein Modul zur WCET Analyse in den Übersetzungsvorgang integriert ist. Dadurch kann die WCET während des gesamten Übersetzungsvorgangs bestimmt und von speziellen Optimierungen beachtet werden. Zusätzlich dazu können vom Compiler viele Standard-Optimierung durchgeführt werden.

1.3 Ziele der Arbeit

Im Folgenden wird zunächst ein kurzer Überblick über die Ziele der Arbeit gegeben und anschließend näher auf die einzelnen Aspekte eingegangen.

- **Erweitern der SingleTask-Optimierungen**
  Zuerst sollen die Optimierungen, die nicht mit mehreren Tasks arbeiten können, erweitert werden, so dass sie auf mehrere Tasks angewendet werden können. Das ist deswegen sinnvoll, da in einem System meist nicht nur ein Task arbeitet, sondern mehrere und dadurch eine realistischere Umgebung geschaffen wird.

- **Festlegen der Reihenfolge**
  Es ist wichtig, eine sinnvolle Optimierungsreihenfolge festzulegen. Hierbei sollte die Optimierung mit dem größten Optimierungspotential zuerst ausgeführt werden

oder die bei der die geringsten Seiteneffekte auftreten. Es sollte auch auf die mögliche Zusammenarbeit der Optimierungen geachtet werden. Dadurch müssen die Optimierungen dann nur die Änderungen der Optimierungen beachten, die vor ihnen gelaufen sind.

- **Abstimmung der Optimierungen**
  Nachdem die Reihenfolge bestimmt wurde, muss für die einzelnen Optimierungen überprüft werden, welche Änderungen von anderen Optimierungen beachtet werden müssen. Dies trifft vor allem auf die Optimierungen zu, die den gleichen Speicher nutzen.

- **Evaluation**
  Es soll abschließend noch überprüft werden, inwiefern es sich lohnt, mehrere Speicheroptimierungen anzuwenden. Dies wird mithilfe von Benchmark-Ergebnissen überprüft. Hierzu werden alle Optimierungskombinationen auf mehrere Benchmarks-Sets angewendet und die Ergebnisse ausgewertet und analysiert.

1.4 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen, die für das Verständnis der Arbeit benötigt werden, erläutert. Danach werden in Kapitel 3 die bestehenden WCET-Optimierungen und die vorgenommenen Anpassungen vorgestellt. Die Benchmarks werden in Kapitel 4 erörtert und deren Ergebnisse analysiert. Das letzte Kapitel stellt eine Zusammenfassung der Arbeit dar und gibt einen Ausblick auf weitere Arbeiten.

2 Grundlagen

In diesem Kapitel wird auf die Grundlagen eingegangen, die für das Verständnis der Arbeit benötigt werden. Zuerst wird die verwendete Plattform vorgestellt. Als nächstes folgen die Begriffe WCET und WCEP in Kapitel 2.2 bzw. 2.3, da diese in der weiteren Arbeit oft verwendet werden und deshalb genau definiert werden müssen. Daraufhin wird der Forschungscompiler WCC in Kapitel 2.4 vorgestellt, da erst durch diesen die WCET Optimierungen möglich sind. Die verschiedenen Arten der Multitask-Repräsentation, die in dieser Arbeit genutzt werden, werden danach beschrieben. Zum Schluss folgt eine detaillierte Beschreibung der Funktionalität von Caches, da dieser von zwei in dieser Arbeit verwendeten Optimierungen genutzt wird.

2.1 Plattform

Für eingebettete Systeme gibt es viele verschiedene Prozessoren, die je nach Anwendungsgebiet mehr oder weniger Funktionen bieten. In dieser Arbeit konzentriere ich mich auf den TriCore TC1796 von Infineon [Inf08]. Der TriCore TC1796 wurde von Infineon für den Automotiv-Bereich entwickelt. Es handelt sich um einen High End Prozessor, der in diesem Bereich z.B. für Motorsteuerungen oder Sicherheitssysteme genutzt wird. Er vereint zugleich drei Architekturen in sich: DSP (Digitial Signal Processor, deutsch: Digitaler Signalprozessor), RISC (Reduced Instruction Set Computer, deutsch: Rechner mit reduziertem Befehlssatz) und CISC (Complex Instruction Set Computer, deutsch: Rechner mit komplexem Befehlssatz).

Ein DSP wird zur Echtzeitbearbeitung von digitalen Signalen eingesetzt. Zusätzlich zur Filterung von Signalen kann er auch andere Aufgaben übernehmen, wie z.B. Echounterdrückung und Spracherkennung. Der DSP besitzt dazu spezielle Befehle wie z.B. MAC (Multply Accumulate), welcher eine Addition und eine Multiplikation in einem Befehl vereint. Dieser wird bei der digitalen Filterung eingesetzt. Eine weitere Besonderheit ist die sogenannte Sättigungsarithmetik. Falls durch eine Operation der darstellbare Zahlenbereich in eine Richtung verlassen wird, wird das Ergebnis,

je nachdem auf welcher Seite der Zahlenbereich verlassen wird, auf die größte oder kleinste darstellbare Zahl gesetzt. Das heißt, bei der Rechnung 7+10 im Zahlenraum bis 15 ist das Ergebnis 15. Bei der RISC Architektur existieren nur wenige, einfache Befehle im Instruktionssatz. Hierdurch gibt es für eine Aufgabe meist nur einen Befehl. Allgemein ist durch die Einfachheit der Befehle die Anzahl der Takt-Zyklen, die das System für einen Befehl braucht, gering. Im Gegensatz dazu gibt es bei einer CISC Architektur viele Befehle, die meist komplex sind. Hierdurch gibt es für eine Aufgabe auch mehrere Instruktionen, die in unterschiedlichen Situationen Anwendung finden. Es werden dabei oft mehrere Takt-Zyklen für einen Befehl gebraucht, da ein Befehl den Prozessor möglichst lange beschäftigen soll.

Im Folgenden einige Features des TC1796:

- 32 Bit Prozessor, auch 16 Bit Befehle verfügbar
- vierstufige Pipeline
- 2 MB Program Flash Memory
- 128 KB Data Flash Memory
- 56 KB Data Memory SRAM
- 16 KB Stand-by Data Memory SBRAM
- 56 KB Local Data SRAM
- 48 KB Instruction Scratchpad Memory
- 16 KB Instruction Cache
  - 32 Byte Instruction Cache Line Size
  - LRU Ersetzungsstrategie
  - 2-fach Assoziativ

Da das Hauptaugenmerk auf drei Speicheroptimierungen liegt, ist vor allem der Speicherausbau interessant. In dem Program Flash Memory, dem Instruction Cache und dem Scratchpad Memory wird jeweils nur Programmcode abgespeichert. Daten werden in anderen Speichern gehalten, sind aber für diese Arbeit uninteressant, da die Optimierungen nur Programmcode verschieben. Allgemein gilt für Speicher: Umso größer die Speicher sind, desto langsamer sind diese. Das heißt, dass der Flash

Memory langsamer ist als der Instruction Cache/Scratchpad Memory und deswegen der Programmcode am besten in einen dieser Speicher verschoben werden sollte. Die weiteren genannten Features wie Pipeline Länge sind für die WCET Analyse von Belang. Das Tool zur WCET Analyse führt unter anderem eine Pipeline- sowie Cache-Analyse durch, wo die entsprechenden Informationen genutzt werden.

2.2 WCET

Ein grober Überblick über die WCET wurde schon in der Einleitung gegeben. Im Folgenden wird genauer erklärt, wieso die WCET wichtig ist und wie man diese berechnet. Bei dem Entwurf von harten Echtzeitsystemen, was viele eingebettete Systeme mit einschließt, ist die WCET wichtig, weil mithilfe dieser überprüft werden kann, ob die Zeitschranken eingehalten werden können. Zur genauen Bestimmung der WCET müssen alle möglichen Eingabedaten sowie Anfangszustände des verwendeten Prozessors beachtet werden. Die Berechnung kann auf zwei verschiedene Arten durchgeführt werden: dynamisch oder statisch.

Bei der dynamischen WCET Analyse wird für viele mögliche Eingaben die Ausführungszeit gemessen. Ein Problem hierbei ist, dass man nicht alle möglichen Kombinationen von Eingaben und Hardwarestartzuständen überprüfen kann, was an der Komplexität der zu untersuchenden Programme und Systeme liegt, die immer weiter zunimmt. Deswegen wird zu der maximalen Laufzeit von einigen Testdurchläufen ein gewisser Puffer addiert, um die WCET möglichst sicher abzuschätzen. Es kann jedoch passieren, dass die WCET bei der dynamischen Analyse unterabgeschätzt wird, da es eine Kombination von Eingabedaten und Anfangszuständen geben kann, die zu einer noch höheren Laufzeit führen kann. Die dynamische WCET Analyse eignet sich daher nicht zu sicheren Bestimmung von Zeitschranken in harten Echtzeitsystemen.

Bei der statischen WCET Analyse wird eine obere Schranke für die WCET abgeschätzt. Diese Schranke sollte, wie in der Einleitung schon erwähnt, möglichst nah an der echten WCET liegen, jedoch muss die Schranke sicher sein, d.h. die WCET darf nicht unterabgeschätzt werden. Eine solche Analyse wird von dem Analysetool aiT von AbsInt [Gmb11] erstellt. AiT kann weitgehend automatisch die WCET für ein Programm in Form einer Binärdatei ermitteln. Hierzu baut das Analysetool aiT aus dem Code den Kontrollflussgraphen auf, um diesen zu analysieren. Dies geschieht in mehreren Schritten, welche in [CFW01] vorgestellt wurden: Zuerst wird eine Werte-Analyse

durchgeführt. Hierbei werden unter anderem Variableninhalte und Schleifenobergrenzen bestimmt sowie indirekte Speicherzugriffe aufgelöst. Danach folgt eine kombinierte Cache/Pipeline-Analyse, welche jeweils in [Fer97] bzw. in [The04] vorgestellt wurde. Für Speicherzugriffe auf den gecachten Hauptspeicher wird dabei überprüft, ob die angefragten Daten im Cache liegen. Dies ist wichtig, da sich durch einen Cache-Miss Verzögerungen in der Pipeline bilden können. Aus den Ergebnissen der Analysen wird in der Pfad-Analyse eine obere Schranke für die WCET ermittelt, welche in [The02] vorgestellt wurde. Hierzu wird der Kontrollfluss in einem ILP dargestellt, das zur Berechnung des längsten Pfades und damit der WCET benutzt wird.

Für die statische WCET Analyse muss der Programmierer zusätzliche Informationen, wie maximale Schleifendurchläufe oder Rekursionsaufrufe angeben, da diese Informationen notwendig sind, um z.B. die Laufzeit von Schleifen zu bestimmen. Im weiteren Verlauf der Arbeit wird nur die statische WCET Analyse benutzt, da nur diese die WCET sicher bestimmen kann.

Allgemein wird zwischen $BCET$, $ACET$, $WCET_{real}$ und $WCET_{est}$ unterschieden. Als unterste Programmlaufzeit-Schranke gilt die Best Case Execution Time ($BCET$). Die Average Case Execution Time ($ACET$) gibt an, wie lange das Programm durchschnittlich läuft, die $WCET_{real}$ ist die größtmögliche Laufzeit für das Programm in einem realen System und die $WCET_{est}$ bezeichnet die maximale Laufzeit, welche durch die statische WCET Analyse bestimmt wurde.

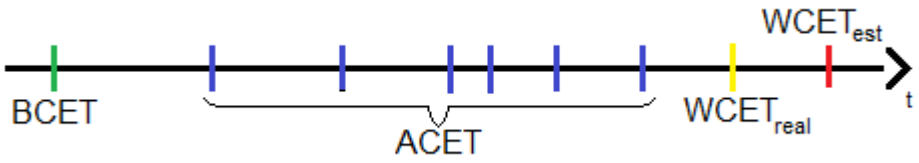

Abbildung 2.1: Vergleich der Ausführungszeiten

In Abbildung 2.1 ist ein Zeitstrahl mit verschiedenen Laufzeiten für ein Programm dargestellt. Ausgehend von links nach rechts: BCET, ACET, $WCET_{real}$, $WCET_{est}$. Die mögliche Ausführungszeit des Programms wird nach unten durch die BCET und nach oben durch die $WCET_{real}$ beschränkt. In diesem Intervall liegt die aufgetragene ACET, die die tatsächlich in mehreren Durchläufen gemessene Laufzeit des Programms angibt. Außerhalb dieses Intervalls liegt die $WCET_{est}$. Zwischen diesen Werten gilt

bei der statischen Analyse $WCET_{real} <= WCET_{est}$. Damit die WCET möglichst exakt ist, sollte $WCET_{est} - WCET_{real} \rightarrow 0$ gelten. Hier ist gut zu erkennen, dass, falls der geschätzte Wert $WCET_{est}$ unterhalb von $WCET_{real}$ liegt, für bestimmte Kontrollflusspfade die Zeitschranke nicht eingehalten werden kann. Wenn es sich bei dem System um ein zeitkritisches System handelt, könnte das die erwähnten fatale Folgen haben (z.B. Airbag löst nicht rechtzeitig aus).

2.3 WCEP

Der WCEP (Worst Case Execution Path) ist der längste Pfad im Kontrollflussgraphen eines Programmes, dessen Ausführungszeit die WCET repräsentiert. In diesem Kapitel wird genau darauf eingegangen wie er erkannt wird und warum er beachtet werden muss. Ein Programm besteht aus Funktionen, welche wiederum aus Basisblöcken bestehen. Ein Basisblock umfasst eine Menge von Instruktionen und kann nur bei der ersten Instruktion betreten und nur bei der letzten verlassen werden. Der Kontrollflussgraph ist ein gerichteter, zyklischer Graph, der alle möglichen Wege durch ein Programm darstellt. Für den Graph gilt: $G = (V, E)$, wobei V die Menge der Basisblöcke des Programms und E die Menge der Kanten ist, die zwei Basisblöcke verbinden, wenn diese hintereinander ausgeführt werden. Die Verzweigungen in dem Graphen werden durch bedingte Sprünge erzeugt. Für jeden Pfad in dem Graphen vom Start bis zu einem der Blätter, kann die maximale Ausführungszeit berechnet werden. Den Pfad mit der höchsten Laufzeit nennt man dann den WCEP. Optimierungen versuchen daher den aktuellen WCEP zu verkürzen, um damit die WCET zu reduzieren.

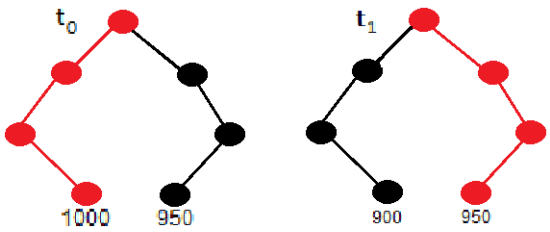

Abbildung 2.2: WCEP Beispiel

In Abbildung 2.2 sind schematisch zwei Kontrollflussgraphen aufgezeichnet. In diesen Graphen ist der Weg, der aktuell der WCEP ist, rot gekennzeichnet. Der Graph im linken Teil der Grafik stellt den Graphen für das unoptimierte Programm und der Graph

im rechten Teil für das optimierte Programm dar. Die Optimierung, die zwischendurch stattfindet, kann den WCEP verkürzen. Dadurch findet ein Pfadwechsel von links nach rechts statt. Wie im rechten Graph zu sehen, ist nach der Optimierung der rechte Pfad der WCEP. Nun muss der rechte Pfad weiter optimiert werden, wenn die WCET reduziert werden soll. Falls weiterhin der linke Pfad optimiert werden würde, würde die WCET nicht weiter sinken. Alle Optimierungen, die in dieser Arbeit betrachten werden, beachten mögliche Änderungen des WCEP.

2.4 WCC

In diesem Abschnitt wird der in dieser Arbeit verwendete **W**CET-aware **C** **C**ompiler (WCC) genauer erläutert. Ein Compiler wird häufig dazu genutzt, um Programmcode von einer Hochsprache in Maschinensprache zu übersetzten. Da die meiste Software in entsprechenden Hochsprachen wie C oder C++ geschrieben worden ist, lohnt sich dies auch bei Eingebetteten Systemen. Der Compiler kann bei dem Übersetzungsvorgang auch Optimierungen vornehmen. Diese kann er weitgehend automatisch durchführen, was sinnvoll ist, da die Optimierung von Hand aufwändig und fehleranfällig wäre. Die Optimierung werden benötigt, um den Programmablauf zu beschleunigen. Der WCC ist, wie in der Einleitung erwähnt, ein C Compiler für den Infineon TriCore TC1796/TC1797, der Programme unter Berücksichtigung der WCET optimieren und übersetzten kann. Die statische WCET Analyse, die während des Übersetzungsvorganges durchgeführt wird, wird von dem Tool aiT von der Firma AbsInt [Gmb11] erstellt. Dadurch, dass die WCET in jedem der Optimierungsschritte berechnet werden kann, kann die WCET als Metrik während der Optimierung miteinbezogen und durch spezialisierte Optimierungen verringert werden. Zusätzlich dazu werden Standard-Optimierungen durchgeführt. Hierzu zählen 23 High-Level und 20 Low-Level Optimierungen.

In Abbildung 2.3 wird der Informationsfluss durch den Compiler dargestellt.
Die Stationen, die durch die dicken schwarzen Pfeile verbunden sind, sind die Übersetzungsschritte, die auch andere Compiler durchführen, wie z.B. der *gcc* [Tea11]. Die weiteren Elemente sind Ergänzungen spezifisch für die WCET Optimierungen. Im Folgenden wird jeder Schritt des Compilers erklärt:

Die C Quelldateien und die Flow-Facts, welche Informationen über Anzahl der Schleifendurchläufe und Rekursionstiefen enthalten, werden vom ICD-C Parser in die High-Level Zwischendarstellung ICD-C übersetzt [Dor11]. Auf dieser können eine Reihe

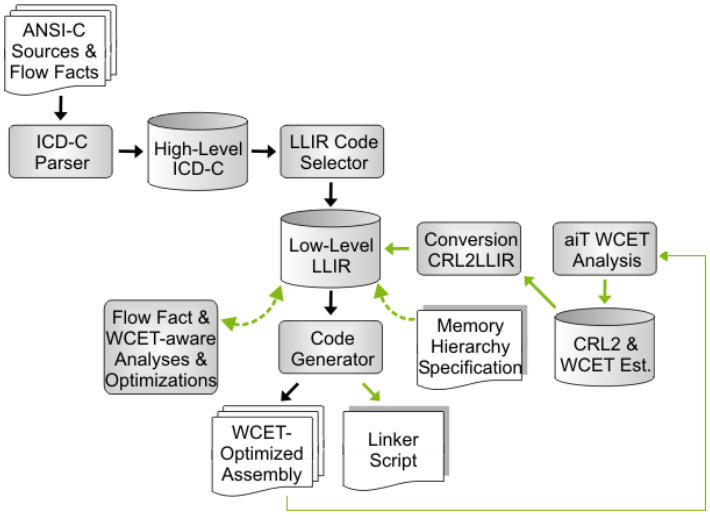

Abbildung 2.3: Compiliervorgang im WCC

von Standardoptimierungen angewendet werden. Danach wird über den LLIR Code Selector für den TriCore eine Low-Level IR erzeugt. Die LLIR ist eine assemblernahe Zwischendarstellung des Codes, an der sogenannte LLIR-Objektives hängen. Über diese kann man unter anderem die WCET oder die Codegröße auslesen. Zusätzlich wird die Speicher-Hierarchie-Beschreibung (Memory Hierarchy Specifications) in der LLIR benutzt, die die verschiedenen Speicher, deren Größen und Zugriffszeiten enthält. Die Flow Fact & WCET-beachtenden Analysen & Optimierungen arbeitet auf Basis der LLIR, da dort alle benötigten Informationen verfügbar sind (zum Beispiel die WCET). Auf dieser Abstraktionsebene arbeiten auch die Optimierungen, die in dieser Arbeit benutzt werden. Dann generiert der Code-Generator ein Linker Skript und die optimierten Assemblerdateien, aus denen ein Linker eine ausführbare Binärdatei erstellt und für die aiT die WCET bestimmen kann. Die ermittelte WCET wird in einer weiteren Zwischendarstellung abgespeichert, der sogenannten CRL2, welche wieder in die LLIR konvertiert werden muss. Diese oben erläuterte Analyse kann bei Bedarf beliebig oft wiederholt werden.

2.5 Multitask-Repräsentation

In Compilern ist die gleichzeitige Repräsentation von mehreren Tasks eines Systems keine Selbstverständlichkeit, daher wird in diesem Kapitel genauer darauf eingegangen, inwiefern dies im WCC realisiert ist. Allgemein kann durch die Multitask-Repräsentation

ein System besser simuliert werden, was wichtig ist, da sich Tasks in Bezug auf die WCET gegenseitig beeinflussen können und dieser Einfluss bei der Optimierung beachtet werden muss. Für die Repräsentation von mehreren Tasks im WCC gibt es zwei Varianten: sog. *Entrypoints* oder eine sog. *Task-Config*.

Die Task-Config besteht aus mehreren sogenannter Task-Entries. Ein Task-Entry umfasst alle Daten, die für die Repräsentation eines Tasks wichtig sind. Es werden unter anderem der Name des Tasks, die LLIR und die Konfiguration für den Task dort abgespeichert. Zusätzlich dazu können auch weitere Parameter, wie z.B. eine Deadline oder eine Priorität für das Scheduling, angegeben werden. Die Task-Entries sind zudem unabhängig voneinander. Das heißt, dass ein Task-Entry nur für einen Task steht und zwischen zwei Task-Entries keine Verbindung besteht.

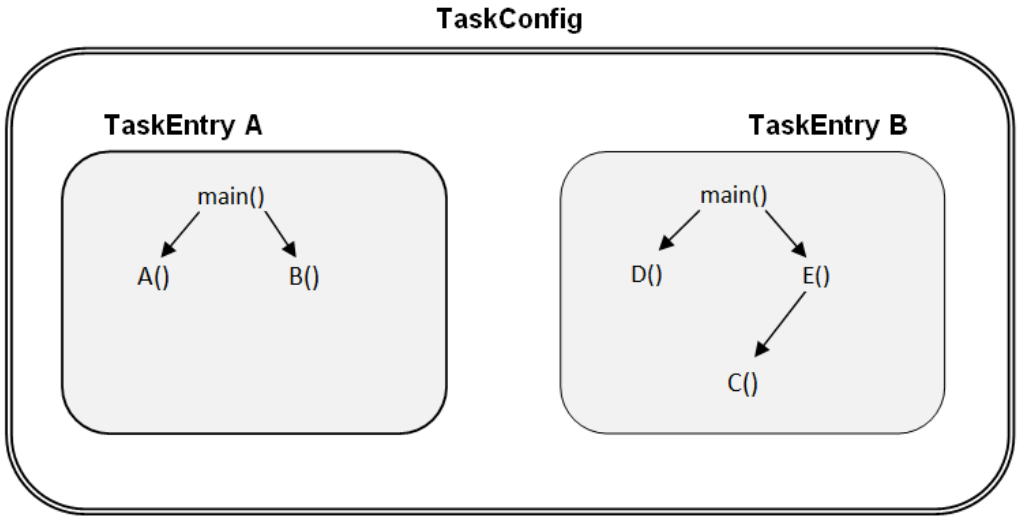

Abbildung 2.4: Taskconfig Beispiel

Dies wird auch in Abbildung 2.4 verdeutlicht. Hier ist schematisch eine Task-Config mit zwei Task-Entries abgebildet. In diesen Task-Entries sind die Funktionen dargestellt, die der Task aufruft, wobei die Pfeile Funktionsaufrufe darstellen. Hier ist deutlich zu erkennen, dass die Task-Entries unabhängig voneinander sind. Sie teilen auch keine Bibliotheken, selbst wenn mehrere Instanzen eines Task in einer Task-Config vorhanden sein sollten.

Im Gegensatz zur Task-Config, in der die Funktion `main` den Einsprungpunkt markiert, können ein oder mehrere Entrypoints die jeweiligen Einsprungpunkte des Programms markieren. Grundsätzlich kann jede beliebige Funktion ein solcher Einsprungpunkt sein. Von diesem Punkt aus wird dann der jeweilige Task ausgeführt. Dabei sind die Entrypoints nicht zwingend unabhängig voneinander: Ein Entrypoint kann eine Funktion aufrufen, die wiederum durch einen anderen Entrypoint als Startpunkt markiert wurde oder auch von einem anderen Entrypoint aus direkt oder indirekt aufgerufen werden kann.

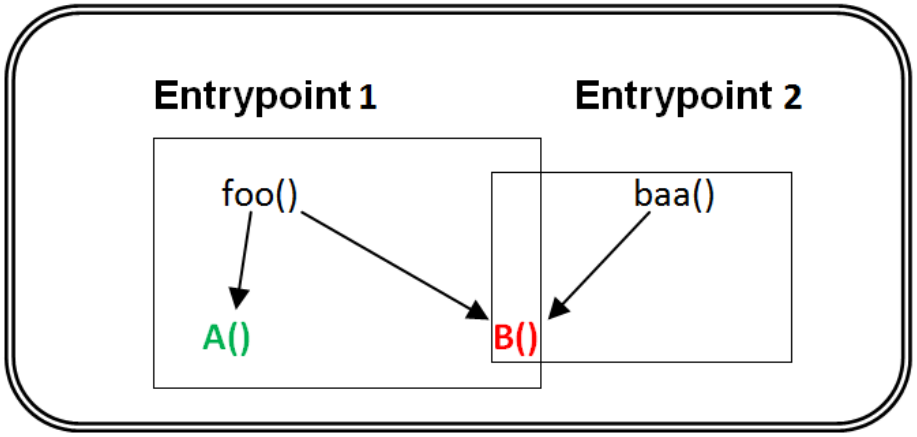

Abbildung 2.5: Entrypoints Beispiel

Verdeutlicht wird dies in Abbildung 2.5. Hier sind schematisch zwei Entrypoints dargestellt. Unter den Entrypoints ist jeweils die Funktion dargestellt, die den Start markiert. Die Pfeile deuten Funktionsaufrufe an. Hier ist zu erkennen, dass die Funktion `B()` von beiden Entrypoints benutzt wird, wobei die Funktion `A()` nur von einem Entrypoint aufgerufen wird. Die Entrypoints sind also erkennbar nicht unabhängig voneinander. Daher müssen die Optimierungen genau auf solche Abhängigkeiten achten, da z.B. die SPM Allocation die Funktion B() nur einmal in den SPM laden kann oder das Cache Partitioning die Funktion gar nicht beachten sollte, da sonst unter Umständen der Cache-Inhalt nicht mehr genau vorausgesagt werden kann.

2.6 Caches

Da die Geschwindigkeit von Prozessoren viel schneller steigt als die von Hauptspeichern, werden verschiedene Speicher unterschiedlicher Größe in sog. Speicherhierarchien eingesetzt, um dem entgegenzuwirken. Im Allgemeinen gilt dabei: Je kleiner ein Speicher ist, umso schneller arbeitet dieser und umso näher ist er an den Prozessor angebunden. Zwischen dem Hauptspeicher und dem Prozessor sitzt der Cache, welcher ein kleiner, schneller Zwischenspeicher ist. In diesem werden die Daten aus dem Hauptspeicher zwischengespeichert, da der Cache viel geringere Zugriffszeiten hat als der Hauptspeicher, wodurch die Ausführung von Programmen beschleunigt werden kann. Wenn nun auf ein Datum zugegriffen wird, kann es zu einem Cache-Hit oder einem Cache-Miss kommen. Bei einem Cache-Hit wurde das Datum im Cache gefunden und kann direkt an den Prozessor übergeben werden. Falls es zu einem Cache-Miss kommt, wurde das Datum nicht im Cache gefunden und muss aus dem Hauptspeicher geladen werden. Daraus folgt, dass vor allem wiederholte Zugriffe auf das selbe Element im Hauptspeicher beschleunigt werden können. Durch den Cache-Controller arbeitet der Cache völlig transparent und im Hintergrund. Bei einem Zugriff auf ein Datum, was im Cache liegt, sieht es von außen so aus, als ob dieses aus dem Hauptspeicher geladen wurde. Dadurch müssen keine Änderungen am Programmcode vorgenommen werden, da es irrelevant ist, ob ein Cache vorhanden ist oder nicht. In dieser Arbeit wird nur ein Instruktion-Cache genutzt, also ein Cache, in dem nur Programm-Instruktionen zwischengespeichert werden.

Der Cache besteht aus sogenannten Cache-Zeilen, welche nur komplett allokiert werden können. Die Cache-Zeilen werden über die Hauptspeicheradressen angesprochen, welche in Tag, Index und Offset-Bits eingeteilt werden. Die Index-Bits geben an, welche Cache-Zeile gemeint ist, während die Offset-Bits angeben, auf welches Wort in der Cache-Zeile zugegriffen wird. Um zu überprüfen, ob die Daten schon vorhanden sind, werden die Tag-Bits benötigt. Zusätzlich wird eine Modulo-Adressierung verwendet. Das heißt, dass die Adresse im Hauptspeicher Modulo einem bestimmten Wert die Cache-Zeile ergibt, in welche die Daten geladen werden. Dadurch werden mehrere Adressen vom Hauptspeicher auf eine Cache-Zeile abgebildet.

Man unterscheidet zwischen drei Cache-Organisationsarten: direct mapped, vollassoziativ und n-fach mengenassoziativ. In den Abbildungen 2.6, 2.7 und 2.8 sind jeweils der Hauptspeicher und der Cache schematisch dargestellt. Die Pfeile zeigen, welcher Hauptspeicherblock auf welche Cache-Zeile abgebildet wird.

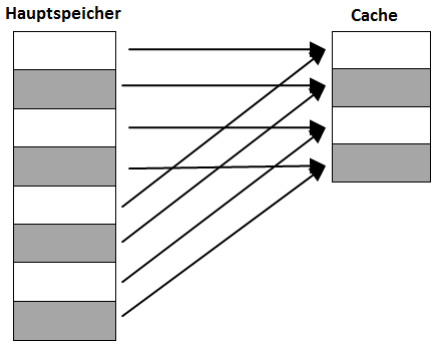

Abbildung 2.6: Direct mapped Cache

Bei der direct mapped Organisationsform wird jedem Hauptspeicherblock genau eine Cache-Zeile zugewiesen. Wenn nun ein Datum angefordert wird, wird über den Index-Teil der Adresse die entsprechende Cache-Zeile adressiert. Danach wird anhand der Tag-Bits überprüft, ob die Daten, die hier abgelegt sind, auch zu der angeforderten Adresse gehören. Falls dies der Fall ist, wird über die Offset-Bits bestimmt, welches Wort aus der Cache-Zeile gelesen werden soll. Wenn nicht, müssen die Daten aus dem Hauptspeicher geladen werden. Insgesamt gibt es mehr Hauptspeicherblöcke als Cache-Zeilen, so dass mehrere Hauptspeicherblöcke auf ein und die selben Cache-Zeile gemappt werden. Durch die Modulo-Adressierung mittels Index-Bits kann es passieren, dass obwohl andere Cache-Zeilen noch frei sind, eine Cache-Zeile ersetzt wird. Dies ist der Fall, wenn ein anderer Hauptspeicherblock, der auf die selben Cache-Zeile gemappt wurde, in den Cache geladen werden soll.

Bei der n-fach mengenassoziativen Organisation werden jeweils n Cache-Zeilen zu einem Cache-Set zusammengefasst; häufig zwei, vier oder acht Zeilen. Hierbei adressieren die Index-Bits eines Hauptspeicherblocks ein komplettes Cache-Set. Wenn nun ein Datum angefordert wird, werden aus der Adresse zuerst die Index-Bits ausgelesen und damit das Cache-Set identifiziert. Danach werden die Tag-Bits der Adresse mit den Tag-Bits aus allen Cache-Zeilen in dem Cache-Set verglichen und damit überprüft, ob eine Zeile die Daten der angeforderten Adresse enthält. Hierbei sind jeweils so viele

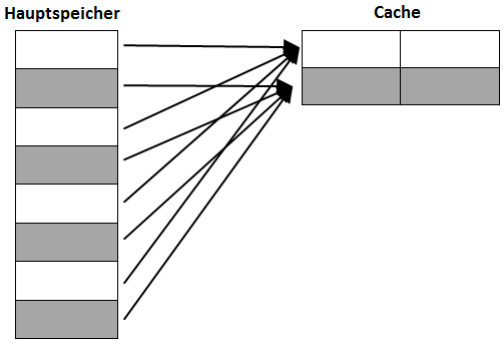

Abbildung 2.7: 2-fach mengenassoziativer Cache

Tag-Vergleiche nötig, wie Cache-Zeilen in einem Set sind, wobei die Tag-Vergleiche in Hardware realisiert werden müssen. Falls der Vergleich positiv ausfällt, wird über die Offset-Bits bestimmt, welches Wort aus der Cache-Zeile gelesen werden soll. Wenn nicht, müssen die Daten aus dem Hauptspeicher geladen werden. Wenn ein Cache-Set komplett belegt ist, also in jeder Cache-Zeile des Sets ein Datensatz liegt, jedoch neue Daten in dem Cache-Set abgelegt werden sollen, müssen die Daten von einer Cache-Zeile ersetzt werden. Hierzu muss eine Ersetzungsstrategie implementiert werden. Eine häufig eingesetzte Ersetzungsstrategie ist Least Recently Used (LRU), die auch vom TC1796 implementiert wird. Hierbei wird genau die Cache-Zeile ersetzt, deren Verwendung am längsten zurückliegt. Es können auch andere Strategien eingesetzt werden, wie z.B. immer den ältesten Eintrag oder einen zufällig ausgewählten Eintrag zu ersetzten.

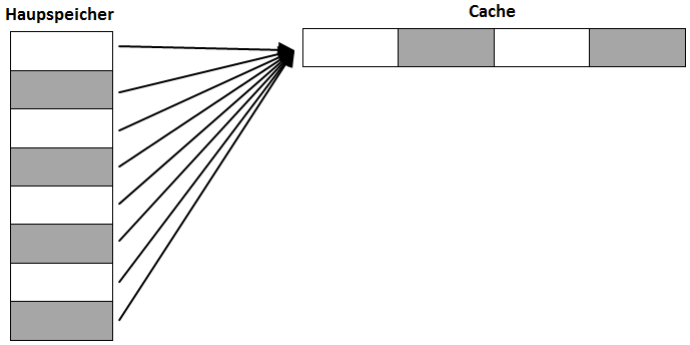

Abbildung 2.8: Vollassoziativer Cache

Bei der vollassoziativen Organisation werden alle Cache-Zeilen in einem Cache-Set zusammengefasst. Es gibt also nur ein Cache-Set, das alle Cache-Zeilen beinhaltet. Wenn nun ein Datum angefordert wird, muss nur das eine Cache-Set überprüft werden.

Es werden die Tag-Bits mit den Tag-Bits aus allen Cache-Zeilen in dem Cache-Set verglichen und damit überprüft, ob eine der n Zeilen die Daten der angeforderten Adresse enthält. Falls dies der Fall ist, wird über die Offset-Bits bestimmt, welches Wort aus der Cache-Zeile gelesen werden soll. Wenn nicht, müssen die Daten aus dem Hauptspeicher geladen werden. Hierbei sind so viele Tag-Vergleiche nötig, wie Cache-Zeilen insgesamt existieren. Bei dieser Organisationsform wird die Auslastung maximal, da erst Zeilen ersetzt werden, wenn der Cache voll ist. Jedoch ist die Suche nach Daten aufwendig, da immer viele Vergleiche nötig sind. Auch hier muss eine Ersetzungsstrategie implementiert werden. Dies erfolgt nach dem gleichen Schema wie bei der mengenassoziativen Organisationsform.

3 Anpassen der WCET-Optimierungen

Dieses Kapitel befasst sich mit den Anpassungen, die an den Optimierungen vorgenommen wurden. Zu Beginn wird dazu erläutert, wie die ursprünglichen Optimierungen arbeiten. Diese Informationen sind wichtig, denn mit ihnen kann daraufhin eine Reihenfolge der Optimierungen festgelegt werden. Je nachdem, wie die Reihenfolge gewählt wird, können bessere oder auch schlechtere Ergebnisse erzielt werden. Nachdem eine sinnvolle Reihenfolge festgelegt wurde, können die Optimierungen besser aufeinander abgestimmt werden, da für jede Optimierung der Vorgänger und die möglicherweise am Programm durchgeführten Änderungen bekannt sind. Im Anschluss daran werden die Änderungen, die an den Optimierungen vorgenommen worden sind, in Kapitel 3.2 dargestellt.

3.1 Die Optimierungen

Im Folgenden werden in den jeweiligen Unterkapiteln die Optimierungen einzeln vorgestellt. In dem ersten Unterkapitel wird die Scratchpad Memory Allocation vorgestellt, im nächsten die Memory Content Selection und zum Schluss das Cache Partitioning. Hierbei ist zu beachten, dass jeweils die ursprüngliche Optimierung vorgestellt wird, bevor diese für diese Arbeit erweitert wurden.

3.1.1 Scratchpad Memory Allocation

Die Optimal Static WCET-aware Scratchpad Allocation of Program Code [FK09] benutzt zur Verringerung der WCET den sogenannten Scratchpad Memory (SPM). Dies ist ein kleiner schneller Speicher, der eng an die CPU angebunden ist. Er ist mit dem L1-Cache einer CPU vergleichbar, wobei der Inhalt des SPM über Software kontrolliert wird. Das heißt, dass der Programmierer manuell oder ein Compiler bestimmen muss, wo welche Programmteile im SPM abgelegt werden. Da auf den SPM schneller als auf den normalen Speicher zugegriffen werden kann, wird durch das Verschieben von Programmcode in den SPM die Laufzeit der verschobenen Teile und damit in der Regel auch die WCET verringert. Dies passiert aber nur, wenn der Programmcode,

der verschoben wird, auf dem WCEP liegt, da nur der Programmcode auf diesem für die WCET entscheidend ist. Falls dies nicht der Fall ist, hat das Verschieben keinen Einfluss auf die WCET, aber unter Umständen trotzdem auf die ACET.

Zur Auswahl der zu verschiebenden Basisblöcke wird ein ILP benutzt. Hierbei werden zwei verschiedene Aspekte miteinbezogen: der WCEP und die Sprungstrafen. Um den WCEP in das ILP miteinzubeziehen, wird der Kontrollflussgraph modelliert. Hierfür wird für jeden Basisblock eine binäre Entscheidungsvariable x_i erzeugt, welche angibt, ob der Block im SPM liegt oder im Hauptspeicher. Zusätzlich dazu wird eine zweite Variable c_i für jeden Basisblock erzeugt. Diese gibt, je nachdem ob der Basisblock im SPM oder im Hauptspeicher liegt, die entsprechende WCET an. Des Weiteren wird für jeden Basisblock die WCET als Maximum der Summe seiner eigenen WCET und des direkten Nachfolgers jeder ausgehenden Kante berechnet. Daher genügt es, durch das Lösen des ILP die Kostenvariable für den Einstiegs-Basisblock zu minimieren. Hierfür müssen verschiedene Vereinfachungen vorgenommen werden.

Zuerst werden in dem Kontrollflussgraph für eine Funktion die Schleifen durch Super-Nodes ersetzt: Eine Schleife besteht aus einer Abfolge von einem oder mehreren Basisblöcken, die durch einen Sprung mehrmals ausgeführt werden können. Für die Berechnung der WCET eines Super-Node wird die WCET der Schleife für einen Durchlauf mit der Anzahl der maximalen Schleifendurchläufe multipliziert. Dadurch repräsentiert solch ein Super-Node genau die WCET, die die Schleife maximal erreichen könnte.

Als nächstes muss beachtet werden, dass, falls Basisblöcke in den SPM verschoben werden, Sprünge eingefügt werden müssen, um den Kontrollfluss zu erhalten. Da diese auch Instruktionen sind, müssen sie für die Bestimmung der WCET ebenfalls beachtet werden. Hierzu werden zunächst die verschiedenen Arten von Sprüngen erläutert, für die Sprungkosten modelliert werden müssen.

Es gibt drei Arten von Sprüngen: implizit, unbedingt und bedingt. Ein impliziter Sprung ist eine Kante im Kontrollflussgraphen von Basisblock b_i zu b_j ohne, dass eine Sprunginstruktion benötigt wird, da die Blöcke im Speicher nacheinander angeordnet sind. Bei einem unbedingten Sprung wird die Kontrolle von Basisblock b_i zu b_j mit einer Sprunginstruktion übergeben, wobei dies unabhängig von irgendwelchen Bedingungen geschieht. Ein bedingter Sprung liegt vor, wenn die Kontrolle von einem

Basisblock b_i im Kontrollflussgraphen entweder über die implizite Kante (Fall-Through Kante) zu Basisblock b_j oder über einen unbedingten Sprung (Pass-Through Kante) zu Basisblock b_k übertragen werden kann. Um den Kontrollfluss wiederherzustellen werden nur unbedingte Sprünge eingefügt, wobei es für jeden der eingefügt wurde eine entsprechende Strafe gibt. Hierbei ist zu beachten, dass durch häufigen Wechsel von Hauptspeicher in den SPM und zurück die WCET durch die eingefügten Sprunginstruktionen steigt. Deswegen bietet es sich an, Schleifen komplett in den SPM-Speicher zu verlegen, da dann nur innerhalb des selben Speichers gesprungen werden muss, oder Basisblöcke, die hintereinander ausgeführt werden.

Falls in einem Basisblock eine andere Funktion aufgerufen wird, werden die Kosten der Funktion auf die des Basisblockes addiert. Zusätzlich dazu muss eine Strafe für den Aufruf der Funktion hinzugefügt werden, ähnlich der Sprungstrafe. Als letzte Bedingung geht die maximale SPM-Größe in das ILP ein, da nur so viele Basisblöcke in den SPM verschoben werden dürfen, wie dort hineinpassen. Das ILP berechnet mit diesen Daten die Basisblöcke, die in den SPM verschoben werden sollen.

Die Optimierung kann durch das Verschieben von Programmcode in den SPM eine Verbesserung der WCET um durchschnittlich 7,4% erzielen, welche bei kleiner SPM-Größe von 10% relativ zur Codegröße des Benchmarks erreicht wurde. Wenn der SPM so groß gewählt wurde, so dass in diesen der komplette Benchmark verlegt werden kann, kann eine Reduktion der WCET von bis zu 40% beobachtet werden.

3.1.2 Memory Content Selection

In diesem Abschnitt wird die WCET-driven Cache-aware Memory Content Selection [PLM10] genauer betrachtet. Diese Optimierung benutzt zur Verringerung der WCET cached und non-cached Speicherbereiche. Also einen Bereich im Speicher, dessen Programmcode in den Cache geladen werden kann (cached) und einen Bereich, dessen Programmcode nicht in den Cache geladen werden kann (non-cached). Da die Optimierung nur auf einem Task arbeitet, wird angenommen, dass eine Verdrängung von Inhalten aus dem Cache nur durch andere Teile des zu optimierenden Task geschehen kann. Generell könnte angenommen werden, dass am besten alle Funktionen in den cached Speicherbereich eingebracht werden sollten. Jedoch kann dies zu einer hohen Cache-Miss-Rate führen, falls sich Programmteile immer wieder gegenseitig aus dem Cache verdrängen. Dies führst zu einer schlechte Vorhersagbarkeit des Cache-Inhaltes,

welche zu einer Steigerung der WCET führen kann. Daher versucht die Optimierung die Vorhersagbarkeit zu erhöhen in dem meist weniger Funktionen in den Cache eingebracht werden.

Für den zu optimierenden Task wird zu Beginn für alle Funktionen der Profit ausgerechnet, um so für jede Funktion zu bestimmen, ob es sich lohnt diese zu cachen oder nicht. Dies geschieht mit folgender Formel:

$$profit(f) = \frac{WCET(f) - WCET_{cached}(f)}{size(f)}$$

Dabei ist $WCET(f)$ die WCET der Funktion f , wenn sie ohne Cache ausgeführt wird, $WCET_{cached}(f)$ die WCET, wenn die Funktion mit Cache ausgeführt wird, und $size(f)$ gibt die Größe der Funktion in Bytes an. Für $WCET_{cached}(f)$ wird bei der WCET Analyse angenommen, dass alle Basisblöcke in den Cache passen und sie sich gegenseitig nicht verdrängen. $Profit(f)$ gibt an, wie stark eine Funktion von einer gecachten Ausführung profitiert. Falls $profit(f)$ negativ sein sollte, da die Funktion im Cache eine höhere Laufzeit hat, wird sie vom Cachen ausgeschlossen. Die WCET einer Funktion ist nur größer Null, wenn diese auf dem WCEP liegt. Dies gilt für $WCET(f)$ sowie $WCET_{cached}(f)$. Hierbei kann es z.B. auch passieren, dass eine Funktion f erst durch das Verschieben in den Cache auf dem WCEP landet, wenn sich durch das Verschieben der WCEP entsprechend ändert. Dadurch wäre $WCET(f) = 0$, aber $WCET_{cached}(f) > 0$. Daher gilt dann $profit(f) \leq 0$ und f gehört deswegen nicht zu den profitablen Funktionen und wird nicht in den gecachten Speicherbereich verschoben.

Wenn der Profit für alle Funktionen berechnet wurde, wird zuerst die profitabelste Funktion in den cached Speicherbereich verschoben, dann die zweit profitabelste Funktion usw. Dies geschieht solange, bis genau so viele Funktionen im Cache liegen, dass alle zur gleichen Zeit vorgehalten werden können. Danach wird für jede weitere Funktion überprüft, ob die WCET weiter sinkt, wenn diese Funktion in den cached Speicherbereich verschoben wird. Hierbei könnte es sein, dass die zusätzliche Funktion zwar eine andere aus dem Cache verdrängt, aber diese nicht mehr benötigt würde, dann würde die WCET sinken. Ist dies der Fall, bleibt die Funktion in dem cached Speicherbereich. Wenn nicht, dann wird sie wieder in den non-cached Speicherbereich verschoben. Dadurch, dass die unprofitablen Funktionen in dem non-cached Speicherbereich liegen, können sie die profitablen Funktionen nicht mehr aus dem Cache verdrängen, wodurch dann die Cache-Miss-Rate und die WCET sinkt.

Die WCET-Verbesserung, die diese Optimierung erzielen kann, hängt daher stark vom Speicherlayout und des Kontrollflusses des zu optimierenden Tasks ab. Durchschnittlich wird eine Reduktion der WCET um 8% bei 5% Cache-Größe relativ zur Codegröße des Gesamten Tasks erzielt. Bei höheren Cache-Größen von 10% und 20% relativ zur Codegröße des Gesamten Tasks liegt der Durchschnitt jeweils bei 6% bzw. 4% Verbesserung der WCET.

3.1.3 Cache Partitioning

In diesem Abschnitt geht es um das WCET-Aware Software Based Cache Partitioning for Multi-Task Real-Time Systems [PLM09]. Dabei wird, wie der Name schon vermuten lässt, der Cache partitioniert. Das heißt, dass jedem Task ein gewisser Teil des Caches zugewiesen wird. Dadurch wird die Analyse des Cacheinhaltes vereinfacht womit es dann zu einer Reduktion der WCET kommen kann.

Das Analysetool aiT kann die WCET nur für einen Task bestimmen. Da aber in einem System häufig mehrere Tasks parallel arbeiten und diese den Cache alle gleichzeitig benutzen, wird die Analyse des Cache-Verhalten kompliziert. Es kann nämlich durch einen Kontextwechsel passieren, dass der neue aktive Task genau die Funktionen aus dem Cache verdrängt, die der alte Task als nächstes benötigt hätte. Da dies theoretisch bei jedem Funktionsaufruf passieren kann, muss aiT für jeden Kontextwechsel den kompletten Cacheinhalt invalidieren. Dadurch wird die WCET viel höher abgeschätzt, als sie eigentlich ist (Überabschätzung). Aufgrund dieser Tatsachen werden oft Systeme eingesetzt, deren Caches deaktiviert wurden oder nicht vorhanden sind. Dadurch kann dann eine geringere WCET erreicht werden.

Um dem Abhilfe zu schaffen, kann der Cache partitioniert werden. Hierbei wird jedem Task eine Partition im Cache zugewiesen, in die nur Programmcode von diesem Task eingebracht werden kann. Deswegen können bei einem Kontextwechsel die Teile eines Tasks im Cache nicht verdrängt werden, wodurch die Vorhersagbarkeit des Cache-Inhalts erst möglich wird und die Cache-Hit-Rate steigt. Somit sinkt die WCET im Vergleich zum “always-miss Szenario”. Es gibt Architekturen, die eine Partitionierung des Caches in Hardware zulassen, dies wird aber beim TC1796 nicht unterstützt. Deshalb realisiert die Optimierung aus [PLM09] eine Partitionierung des Caches in Software.

Die Grundlagen zu Caches wurden in Kapitel 2.6 beschrieben. Durch die Modulo Adressierung werden mehrere Hauptspeicher-Adressen auf dieselbe Cache-Zeile abgebildet. Wenn dem Task nun ein Bereich des Caches zugewiesen wurde, muss der Task genau auf die Adressen im Hauptspeicher verteilt werden, die durch die Modulo Adressierung auf seinen zugewiesenen Bereich im Cache abgebildet werden. Die Optimierung realisiert dies mit Linker-Sections, in die Basisblöcke verschoben werden. Die Sections sind entsprechend der Adressen verteilt und werden dadurch auf die zugewiesenen Partitionen im Cache abgebildet. Dadurch wird ein Task in mehrere Blöcke geteilt, wodurch Sprünge zwischen diesen eingefügt werden müssen, um den Kontrollfluss wiederherzustellen. Die eingefügten Sprünge wirken sich jedoch negativ auf die WCET aus.

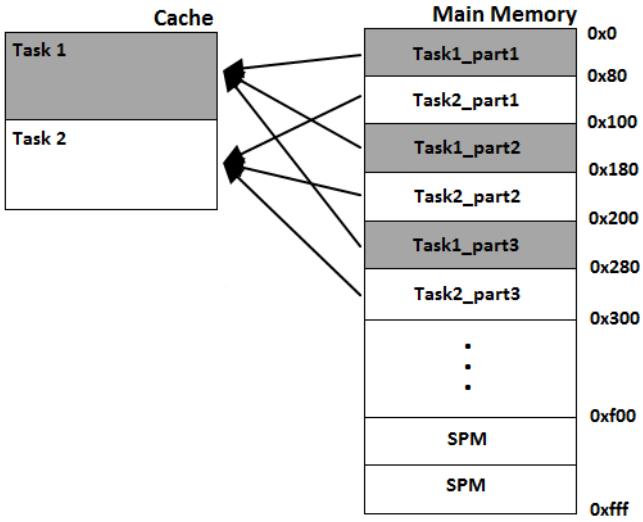

Abbildung 3.1: Cache Partitioning Beispiel

In Abbildung 3.1 wird die Verteilung der Adressen nochmal verdeutlicht. Auf der linken Seite der Grafik wird der Cache dargestellt und auf der rechten der Hauptspeicher. Im Cache liegen zwei Tasks (grau/weiß), welchen schon Partitionen zugewiesen worden sind. Um nun zu gewährleisten, dass nur der jeweilige Task in seine Partition geladen wird, muss der Task genau an die Adressen im Hauptspeicher verteilt werden, die auf die entsprechenden Cache-Zeilen seiner Partition abgebildet werden. Genau dies ist im Hauptspeicher zu beobachten. Hier wurden die Tasks in mehrere Teile aufgeteilt, wobei durch die Pfeile angedeutet wird, auf welche Partitionen im Cache die entsprechenden Adressen abgebildet werden. Hierbei ist zu beachten, dass die Blöcke im Cache und im

Hauptspeicher gleich groß sind. Gut zu erkennen ist, dass Task1 nur auf seine eigene Cachepartition abgebildet wird, respektive Task2 nur auf seine eigene. Unten rechts in der Abbildung ist in dem Adressraum auch der Scratchpad Memory (SPM) angedeutet. Zu diesen Adressen würde die Scratchpad Memory Allocation den Programmcode verschieben (siehe Kapitel 3.1.1). Da aber die Geschwindigkeit des SPMs ähnlich der des Caches ist, wird der SPM Inhalt nicht gecached.

Allgemein erfolgt die Optimierung in drei Schritten:

1. **Tasks partitionieren und WCET bestimmen**
   Hierbei werden die Tasks für jede vernünftige Cache-Größe (64...X Byte, wobei X die Größe des Tasksets aufgerundet auf 2^n ist) partitioniert und dann die WCET bestimmt und abgespeichert. Die WCET kann nach der Partitionierung mit aiT berechnet werden, da sich ein partitionierter Task wie ein Single-Task System verhält. Dadurch können andere Tasks nicht mehr auf das Cache Verhalten einwirken.

2. **ILP mit allen WCETs und Partitionsgrößen aufstellen und lösen**
   Es werden hierbei zwei Bedingungen beachtet. Jedem Task darf nur eine Partitionsgröße zugeordnet werden und die aufsummierten Partitionsgrößen dürfen die Cache-Größe nicht überschreiten. Die Zielfunktion beschreibt die Gesamt-WCET des Systems, welche minimiert werden soll. Dann wird das ILP gelöst.

3. **Gemäß der Lösungen aus 2. partitionieren**
   Den Tasks werden die optimalen Partitionsgrößen im Bezug auf die minimale WCET des Gesamtsystems zugewiesen und damit partitioniert.

Um zu überprüfen wie gut diese Optimierung ist, wurde sie mit einer Code-Größen-Heuristik verglichen. Bei dieser Heuristik hängt die Partitionsgröße eines Tasks von seinem Anteil an der Gesamt Taskset Größe ab. Die Optimierung erreicht im Vergleich zu der Heuristik eine Verbesserung der WCET um bis zu 30% für Tasksets kleiner Größe (5 Tasks). Bei größeren Tasksets (15 Tasks) sind sogar Verbesserungen von bis zu 33% möglich. Die Tasksets wurden für diese Ergebnisse mit zufällig ausgewählten Tasks aus einer Benchmarksuite gefüllt. Durchschnittlich werden für kleine Tasksets 12% und für große 19% Verbesserung generiert.

3.2 Änderungen an den Optimierungen

In diesem Abschnitt werden in den Unterkapiteln jeweils die Änderungen vorgestellt, die an den entsprechenden Optimierungen vorgenommen wurden. Zuerst geht es um die Reihenfolge, in der die Optimierungen angewandt werden. In Kapitel 3.2.2 werden die in dieser Arbeit entwickelten Erweiterungen und Änderungen an der Scratchpad Memory Allocation vorgestellt, danach folgt die Memory Content Selection und dann das Cache Partitioning. Im letzten Kapitel wird auf die Kombination des Cache Partitionings und der Memory Content Selection eingegangen.

3.2.1 Reihenfolge

In diesem Kapitel geht es um die Reihenfolge der Optimierungen und warum sie so gewählt wurde. Allgemein gilt, dass optimierende Compiler möglichst effektive Optimierungen bereitstellen sollen. Doch genauso wichtig wie die Effektivität der einzelnen Optimierungen ist die Reihenfolge, in der die Optimierungen angewandt werden. Wenn z.B. das *Loop-Unrolling*, wobei Schleifen abgerollt werden, erst zum Schluss einer Optimierungsreihenfolge angewandt würde, würden unter Umständen schlechtere Ergebnisse erzielt werden, da die Ausgangssituation für die anderen Optimierungen eine komplett andere wäre. Als Beispiel hierfür kann die *Common Subexpression Elimination* genannt werden. Hierbei werden Zwischenrechnungen, die öfters vorkommen, in einer temporären Variable gespeichert, so dass die Rechnung nur einmal ausgeführt werden muss. Es kann sein, dass erst durch das Loop-Unrolling klar wird, dass die *Common Subexpression Elimination* angewandt werden kann.

Um eine möglichst gute Reihenfolge von Optimierungen zu erstellen, ist viel Expertise des Compiler-Entwicklers von Nöten, da er weitreichende Kenntnis von der Funktionsweise der verwendeten Optimierungen haben muss. In [LPF+11] wird neuer Ansatz vorgestellt, der für ein zu übersetzendes Programm über einen mehr-kriteriellen Evolutionären Algorithmus eine verbesserte Optimierungsreihenfolge sucht. Als Kriterien gelten hierbei die WCET, die ACET und die Codegröße des erzeugten Programms. Zwar ist diese Optimierung sehr rechenintensiv, jedoch kann damit eine bessere Reihenfolge der Optimierungen in Bezug auf jeweils eines der Kriterien bestimmt werden.

Zusätzlich muss beachtet werden, dass bestimmte Optimierungen nur auf bestimmten Zwischendarstellungen angewandt werden können. Außerdem gilt für eine Reihenfolge von Optimierungen, dass diese nicht für alle möglichen Situationen optimal gewählt

werden kann. Es gibt immer Programme, die von einer anderen Reihenfolge der Optimierungen mehr profitieren würden. Jedoch kann eine Reihenfolge so gewählt werden, dass sie möglichst gute Ergebnisse für möglichst viele Programme liefert.

Wenn z.B. in der zu bestimmenden Reihenfolge die Scratchpad Memory Allocation nach dem Cache Partitioning und der Memory Content Selection ausgeführt wird, könnte dies dazu führen, dass nicht die größtmögliche WCET Reduktion erreicht wird: Das Cache Partitioning würde die Tasks in den cached Speicherbereich verschieben und dann partitionieren, wodurch alle Tasks über den Adressraum verteilt wären. Erst danach würde die SPM Allocation ausgeführt werden. Dies würde dazu führen, dass aus einigen Partitionen Programmcode in den SPM verschoben werden würde, wodurch Lücken in den Partitionen entstehen würden. Prinzipiell ist dies kein Problem, jedoch wird dadurch Cache-Speicher verschwendet, der andere Teile des Tasks cachen könnte. Es ist daher wahrscheinlich, dass niedrigere WCETs erzielt werden können, wenn die SPM Allocation vor den anderen Optimierungen ausgeführt wird.

Das Cache Partitioning und die Memory Content Selection versuchen beide, das Speicherlayout eines Tasks so zu verändern, dass das Cache-Verhalten verbessert wird. Um nun eine Reihenfolge dieser Optimierungen festzulegen, müssen zunächst die Auswirkungen dieser bedacht werden. Falls zuerst das Cache Partitioning und danach die Memory Content Selection ausgeführt wird, werden bei einem Task zuerst alle Funktionen partitioniert und dann werden bestimmte Funktionen vom Cachen ausgeschlossen. Hierbei entstehen auch Lücken in den Partitionen. Eine andere Möglichkeit wäre, dass die Memory Content Selection vor dem Cache Partitioning ausgeführt werden würde. Es würden also zuerst die profitabelsten Funktionen ausgesucht werden und diese würden dann partitioniert werden. Prinzipiell wäre dies kein Problem, jedoch kann sich die beste Auswahl der zu cachenden Funktionen für jede Partitionsgröße ändern, und daher kann eine höhere Reduktion der WCET erzielt werden, wenn die Auswahl der Funktionen für jede Partitionsgröße bestimmt wird. Um dies zu erreichen, wird eine Kombination der Optimierungen vorgenommen. Dies geschieht insoweit, dass das Cache Partitioning nicht mehr alle Funktionen verschiebt und partitioniert, sondern nun die Memory Content Selection integriert und für jeden Task und jede Partitionsgröße aufruft und dazu benutzt, Funktionen auszuwählen, die am meisten von gecachter Ausführung profitieren und dann nur diese partitioniert.

Daher werden die Optimierungen in folgender Reihenfolge angewendet:

- Scratchpad Memory Allocation
- Cache Partitioning mit/ohne Memory Content Selection

3.2.2 Erweiterung der Scratchpad Memory Allocation

Da die Scratchpad Memory Allocation als erste der Speicheroptimierungen durchgeführt wird, muss diese nicht auf vorhergehende Speicheroptimierungen achten. Die Optimierung wird in der ursprünglichen Version nur auf einem Task ausgeführt. Daher musste sie dahingehend erweitert werden, dass sie auf ein Taskset, was durch eine TaskConfig oder Entrypoints repräsentiert wird, angewandt werden kann. Eine grundlegende Idee für diese Umsetzung war, dass für jeden Task die SPM-Größe durch eine zu entwickelnde Heuristik bestimmt und dann der Task mit der entsprechenden Größe von der unveränderten Optimierung optimiert wird.

Für die SPM Größenbestimmung wurden in dieser Arbeit drei unterschiedliche Heuristiken entwickelt und implementiert: eine WCET-Heuristik, eine Codegrößen-Heuristik und eine Kombi-Heuristik. Jede Heuristik berechnet für jeden Task seinen zur Verfügung stehenden Teil vom SPM. Die erste Heuristik arbeitet auf Basis der WCETs der einzelnen Tasks und bestimmt darüber die SPM-Größen. Sie berechnet für Task $Task_j$ eines Tasksets mit n Tasks den Anteil am SPM nach der Formel:

$$size_{SPM}(Task_j) = \frac{WCET(Task_j)}{\sum_{i=1}^{n} WCET(Task_i)} * size(SPM)$$

Hier wird der genaue Anteil des Tasks an dem SPM über den prozentualen Anteil an der WCET des Tasksets ausgerechnet. Genau diese SPM-Größe wird der originalen SPM Allocation für die Optimierung des Tasks übergeben. Die Idee hierbei ist, dass durch diese Heuristik der Task mit der größten WCET auch den meisten Speicher zugeteilt bekommt, um dessen WCET am stärksten zu reduzieren.

Ähnlich wie die erste Heuristik, arbeitet die zweite Heuristik auf Basis der Codegrößen, um darüber die SPM-Größen zu bestimmen. Die Formel der Codegrößen-Heuristik, mit

der der Anteil an dem SPM von einem Task $Task_j$ aus einem Taskset mit n Tasks ausgerechnet wird, unterscheidet sich nur wenig von der der Laufzeit-Heuristik:

$$size_{SPM}(Task_j) = \frac{size(Task_j)}{\sum_{i=1}^{n} size(Task_i)} * size(SPM)$$

Bei dieser Formel wird der Anteil eines Tasks am SPM nicht über seinen prozentualen Anteil an der gesamten WCET berechnet, sondern über seinen prozentualen Anteil an der gesamten Codegröße des Tasksets. Bei dieser Heuristik ist die Idee, dass dem Task mit der größten Codegröße auch der größte Anteil am SPM zugewiesen wird.

Die letzte Heuristik ist eine Kombination der ersten Beiden. Bei der Kombi-Heuristik werden beide Formeln miteinander kombiniert und der Durchschnitt gebildet:

$$size_{SPM}(Task_j) = \left(\left(\frac{WCET(Task_j)}{\sum_{i=1}^{n} WCET(Task_i)} + \frac{size(Task_j)}{\sum_{i=1}^{n} size(Task_i)} \right) /2 \right) * size(SPM)$$

Hier wird jeweils der prozentuale Anteil an der Gesamt-WCET und an der Gesamt-Codegröße ausgerechnet und von diesen Werten der Durchschnitt gebildet. Dieser Prozentsatz bestimmt den Anteil an dem SPM, der der orginal SPM Allocation übergeben wird. Hierbei ist die Idee, dass die Kombination der Heuristiken in bestimmten Fällen bessere Ergebnisse liefern könnte als jede für sich.

Alle Heuristiken überprüfen zudem für jeden Task, ob die Task Codegröße kleiner ist als die ihm zugeteilte Speichermenge. Falls dies der Fall ist, wird dem Task nur soviel SPM zugeteilt, dass der gesamte Task in seinen Teil des SPM passt. Dadurch können andere Tasks solche nicht genutzten Speicherbereiche belegen. Zusätzlich wird der SPM zeilenweise vergeben. Es wird überprüft, ob die zugeteilte SPM-Größe des Tasks ein Vielfaches der Speicherzeilengröße ist. Falls dies nicht der Fall ist, wird auf ein Vielfaches aufgerundet. Dies kann natürlich nur passieren, wenn noch genug SPM zur Verfügung steht, um dies durchzuführen. Ist dies nicht der Fall, wird abgerundet.

Die Heuristiken können auch auf Entrypoints angewendet werden. Dabei kann es passieren, dass mehrere Entrypoints die gleiche Funktion aufrufen. Dann überprüfen die Heuristiken, dass diese Funktion nur einmal in den SPM verschoben und die Zuweisung

nicht durch die SPM Allocation für einen anderen Entrypoint verändert wird. Hierzu wird, nachdem ein Entrypoint eine SPM-Größe zugewiesen bekommen hat, dieser direkt von der SPM Allocation optimiert und erst danach wird der Rest SPM-Speicher an die anderen Entrypoints vergeben. Da nun die Basisblöcke, die schon in den SPM verschoben wurden, durch eine fixe WCET keinen Einfluss mehr auf das ILP der SPM Allocation haben, werden die Funktionen nur einmal in den SPM verschoben.

3.2.3 Änderungen der Memory Content Selection

Dieser Abschnitt beschäftigt sich mit den Anpassungen, die an der Memory Content Selection im Verlauf dieser Arbeit vorgenommen wurden. Da der Aufruf der Memory Content Selection nach der Scratchpad Memory Allocation passiert, müssen die Änderungen, die durchgeführt wurden, beachtet werden. Wenn nun eine Funktion verschoben werden soll, kann daher nicht die komplette Funktion verschoben werden, sondern es muss für jeden Basisblock überprüft werden, ob er in dem Hauptspeicher oder dem Scratchpad Speicher liegt. Falls ein Basisblock im Hauptspeicher liegt, kann dieser wie gewohnt zwischen gecachtem und ungecachtem Speicher verschoben werden. Ist dies nicht der Fall, hat die Scratchpad Memory Allocation diesen in den SPM verschoben und er muss dort belassen werden.

Zusätzlich kann es bei der Memory Content Selection passieren, dass die Reduktion der WCET größer ist, wenn, statt der ausgewählten Funktionen, alle Funktionen in den cached Speicherbereich verschoben werden. Dies kann durch die Anordnung der Funktionen im Cache passieren, welche dann eine andere ist wenn alle Funktionen verschoben werden. Um dem entgegenzuwirken, kann der Auswahl-Vorgang der zu cachenden Funktionen in umgekehrter Reihenfolge durchgeführt werden: Anstatt Funktionen in den Cache hinzufügen zu lassen, werden zuerst alle Funktionen in den cached Speicherbereich verschoben und dann die unprofitabelsten Funktionen nach und nach wieder in den non-cached Speicherbereich. Hierbei wird wie bei der anderen Reihenfolge jeweils die WCET mit und ohne diese Funktion ermittelt und damit überprüft, ob es sich lohnt diese Funktion zu cachen.

Im schlechtesten Fall müssen bei einer Menge von n Funktionen n WCET-Analysen durchgeführt werden. Um diese Anzahl zu verringern, können durch die Änderungen, die in dieser Arbeit vorgenommen wurden, nur so viele Funktionen vom Cachen ausgeschlossen werden, dass alle noch zu cachenden Funktionen im Cache gleichzeitig

vorgehalten werden können. Hierdurch sinkt die Anzahl der WCET-Analysen und damit auch die Laufzeit der Optimierung.

Die Memory Content Selection wurde in dieser Arbeit erweitert, so dass nun auch Entrypoints bearbeitet werden können. Einer der Unterschiede zwischen einer Task-Config und Entrypoints ist, dass bei Entrypoints ein und die selbe Funktion von zwei verschiedenen Entrypoints aufgerufen werden kann, wohingegen bei einer Task-Config die Tasks komplett unabhängig voneinander sind. Falls eine Funktion von mehreren Entrypoints genutzt wird, kann nicht vorhergesagt werden, wann diese Funktion Teile im Cache ersetzt (siehe Kapitel 2.5).
Deswegen muss eine Funktion F zwei Bedingungen erfüllen, damit sie bei der Optimierung eines Entrypoints in den gecachten Speicherbereich verschoben werden kann:

$$WCET(F) > 0 \text{ , für den aktuellen Entrypoint} \tag{3.1}$$

$$Funktionsaufrufe(F) = 0 \text{ , für alle anderen Entrypoints} \tag{3.2}$$

Die erste Bedingung stellt sicher, dass die Funktion für den Entrypoint, der aktuell bearbeitet wird, auch auf dem WCEP liegt, da dann die WCET der Funktion für diesen Entrypoint größer als Null ist. Nun darf die Funktion von keinem anderen Entrypoint benutzt werden, da sonst nicht klar ist, wann die Funktion Teile im Cache ersetzt und damit das Cache Verhalten nicht eindeutig bestimmt werden kann. Dies wird durch die zweite Bedingung sichergestellt, in der überprüft wird, ob die Funktion F von keinem anderen Entyrpoint aufgerufen wird.

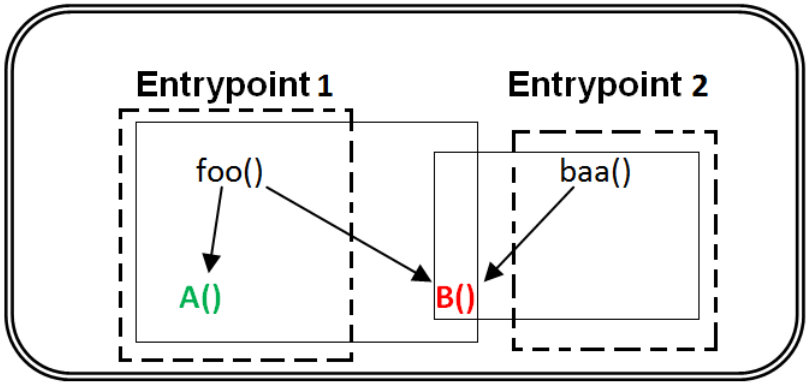

Abbildung 3.2: Entrypoints mit überschneidenden Funktionen

In Abbildung 3.2 wird dies nochmal verdeutlicht. Dort sind wie in Abbildung 2.5 zwei Entrypoints schematisch dargestellt. Hierbei wird die Funktion B() von beiden Entrypoints benutzt, wodurch nur der gestrichelt umrandete Teil der von den Entrypoints aus erreichbaren Funktionen für das Cachen infrage kommt, da die Funktion B() nicht beide Bedingungen erfüllt.

3.2.4 Änderungen des Cache Partitioning

Im Folgenden werden die Änderungen am Cache Partitioning näher erläutert. Da auch diese Optimierung nach der Scratchpad Memory Allocation aufgerufen wird, muss die Optimierung die Änderungen beachten, die durchgeführt wurden. Wenn also vor der Partitionierung eine Funktion in den cached Speicherbereich verschoben werden soll, um danach partitioniert zu werden, kann nun nicht die komplette Funktion verschoben werden, sondern es muss für jeden Basisblock überprüft werden, ob er im Hauptspeicher oder dem Scratchpad Speicher liegt. Falls ein Basisblock im Hauptspeicher liegt, kann dieser wie gewohnt zwischen gecachtem und ungecachtem Speicher verschoben werden. Liegt der Basisblock im Scratchpad Speicher, hat die Scratchpad Memory Allocation diesen in den SPM verschoben und er muss dort belassen werden.

Zusätzlich wurde in dieser Arbeit eine Methode entwickelt, die das Auswählen der Partitionsgröße, welche zwischen 64 und X Byte liegt, wobei X die Größe des Tasksets aufgerundet auf 2^n ist, für einen Task beschleunigt. Hierbei werden nicht alle Partitionsgrößen für jeden Task ausgewertet, sondern es wurde festgelegt, dass nur die Partitionsgrößen beachtet werden sollen, welche zwischen 10% und 100% der Codegröße eines Tasks liegen. Hierbei gibt es eine Ausnahme: Wenn der Task so klein ist, dass die kleinste Partitionsgröße von 64 Bytes bereits über 100% der Codegröße des Tasks liegt, wird dieser dennoch mit der kleinsten Partitionsgröße partitioniert. Durch den Wegfall von Partitionierungen und WCET-Analysen wird die Laufzeit der Optimierung reduziert, da diese den Großteil der Laufzeit in Anspruch nehmen.

Auch das Cache Partitioning wurde in dieser Arbeit so erweitert, dass es Entrypoints bearbeiten kann. Hierbei werden die selben Bedingungen wie bei der Memory Content Selection beachtet (siehe 3.1 und 3.2). Funktionen können nur gecacht werden, wenn die WCET für den aktuellen Entrypoint größer Null ist und von keinem anderen Entrypoint aufgerufen wird.

3.2.5 Kombination von Cache Partitioning und Memory Content Selection

Die Optimierungen werden, wie in Kapitel 3.2.1 beschrieben, kombiniert, da nur dadurch die maximale Reduktion der WCET erreicht werden kann. Das Cache Partitioning soll für jeden Task und jede Partitionsgröße die Memory Content Selection aufrufen, um die Funktionen zu bestimmen, die am meisten von gecachter Ausführung profitieren. Hierzu wurden in dieser Arbeit zwei Ideen verfolgt:

In der ersten Idee wird vor jedem Partitionieren für die aktuelle Partitionsgröße die Memory Content Selection aufgerufen, welche nur die profitabelsten Funktionen in den cached Speicherbereich verschiebt. Anschließend werden diese dann partitioniert. Im Gegensatz zum alleinigen Cache Partitioning kann hierbei jedoch kaum eine zusätzliche Reduktionen der WCET erreicht werden, da sich durch das Partitionieren der Tasks das Speicherlayout sehr stark verändert und die Memory Content Selection diese Änderungen bei der Auswahl der Funktionen nicht berücksichtigt.

Aus diesem Grund wurde die zweite Idee entwickelt: Hierbei wird ebenfalls die Memory Content Selection von dem Cache Partitioning für jede Partitionsgröße aufgerufen. Jedoch werden hier nach jeder Zuweisung einer Funktion zu einem Speicher durch die Memory Content Selection die Funktionen partitioniert und erst dann die WCET berechnet. So können die Änderungen des Speicherlayouts durch das Partitionieren und deren Einfluss auf die WCET berücksichtigt werden.

Der genaue Ablauf wird nochmal in Abbildung 3.3 dargestellt: Während das Cache Partitioning nach der optimalen Partitionsgröße für einen einzelnen Task sucht, wird die Memory Content Selection aufgerufen. Diese verschiebt zunächst alle Funktionen in den cached Speicherbereich, lässt diese durch das Cache Partitioning partitionieren und berechnet deren WCET. Danach wird der gleiche Vorgang ohne die unprofitabelste Funktion durchgeführt, dann ohne die zweit unprofitabelste usw. Hierbei wird jeweils die WCET mit dem vorherigen Lauf verglichen und falls diese geringer ist, wird die Funktion vom cachen ausgeschlossen und für diese Partitionsgröße nicht mehr beachtet. Dadurch, dass nach jeder Entscheidung der Memory Content Selection partitioniert und eine WCET-Analyse durchgeführt wird, steigt die Laufzeit der Optimierung an, jedoch wurden hierzu schon Verbesserungen implementiert (siehe Kapitel 3.2.3). Nachdem für alle Partitionsgrößen die WCETs und Funktionssets aller Tasks ermittelt worden sind, wird mit den Ergebnissen, wie bei dem normalen Cache Partitioning, ein ILP

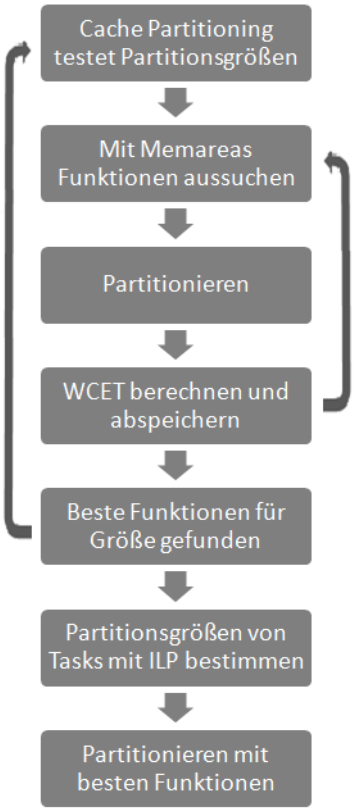

Abbildung 3.3: Ablauf des Cache Partitionings

aufgestellt. Die Lösung des ILPs bestimmt für jeden Task die optimale Partitionsgröße in Bezug auf die Gesamt-WCET des Systems. Nun muss vor dem Partitionieren der Tasks die durch die Memory Content Selection bestimmte Zuweisung von Funktionen zu gecachtem und ungecachtem Speicher wiederhergestellt werden.

4 Evaluation

In diesem Kapitel geht es um die Auswertung der Benchmarkergebnisse. Diese Auswertung ist wichtig, um zu überprüfen, ob die Optimierungen zusammen arbeiten können, inwiefern die Einsparungen der Optimierungen sich addieren und ob Seiteneffekte auftreten. Hierzu werden die Optimierungen auf mehrere Benchmark-Sets angewandt, um ein möglichst realistisches Ergebnis zu erhalten. Zusätzlich werden die verschiedenen Speichergrößen variiert, um zu überprüfen, ob z.B. die Cache-Größe einen höheren Einfluss auf die WCET Reduktion hat als die SPM-Größe. Zunächst werden hierfür die verwendeten Benchmarks in Kapitel 4.1 vorgestellt. Danach werden die Ergebnisse, die die Optimierungen für diese Benchmarks erreicht haben, in den Kapiteln 4.2 - 4.5 präsentiert. Hierbei werden zunächst die Einzel-Optimierungen getestet und dann die Kombinationen der Optimierungen. Die Ergebnisse werden jeweils mit den Werten eines System ohne Cache und ohne die Speicheroptimierungen verglichen, wobei das Optimierungslevel O2 genutzt wird.

4.1 Benchmarks

Es wurden vier verschiedene Benchmark-Sets verwendet, welche in Tabelle 4.1 aufgelistet werden. Hierbei wird für jeden Task zusätzlich noch die Codegröße in Bytes und die WCET ohne eine der Speicheroptimierungen angegeben.

Die Benchmarks wurden aus verschiedenen Benchmark-Suiten ausgewählt: Aus der MRTC Benchmark-Suite [GBEL10] wurde der *crc* Benchmark genutzt. Hier werden CRC-Check-Summen berechnet. Diese können z.B. bei einer Datenübertragen benutzt werden, um sicherzustellen, dass die empfangenen Daten den gesendeten Daten entsprechen. So kann ausgeschlossen werden, dass bei einer Übertragung über einen potentiell gestörten Kanal ein Bit flippt(von 0 auf 1 kippt oder andersherum).

Die Benchmarks *h264dec_ldecode_block* und *gsm_decode* stammen aus der Benchmark-Suite `MediaBench` [LPMS97]. Der *gsm_decode* kann zum Decodieren

Name	Benchmarks	Größe in Byte	WCET
Set1	g721_encode	4852	2278604
	edge_detect	770	39072467
	latnrm_32_64	418	311380
Set2	crc	518	252160
	g721_marcuslee_decoder	144	231114
	h264dec_ldecode_block	13994	325171
Set3	trellis	3086	1187540
	fft_1024	944	15187612
	gsm_decode	6532	17371046
	crc	518	252160
Set4	h264dec_ldecode_block	13994	325171
	g721_marcuslee_decoder	144	231114
	crc	518	252160
	fft_256	948	1041758
	latnrm_32_64	418	311380

Tabelle 4.1: Benchmark-Sets

eines GSM Signales in Software verwendet werden, wie z.B. in Handys. Beim *h264dec_ldecode_block* werden Videosignale, die mit dem h264 Codec [h2609] komprimiert wurden, decodiert, um sie z.B. auf einem Handy-Display wiedergeben zu können.

Der *g721_encode* ist aus der selbst erstellten `misc` Benchmark-Suite. Dies ist ein Audio-Encoder, mit dem Audiodaten komprimiert werden, z.B. im Handy.

Die übrigen Benchmarks stammen aus der UTDSP Benchmark-Suite [UTD11]: *edge_detect*, *latnrm_32_64*, *g721.marcuslee_decoder*, *trellis*, *fft_1024* und *fft_256*.

Die *fft_1024* und *fft_256* berechnen die Fast Fourier Transformation, welche für die digitale Signalverarbeitung wichtig ist. Diese findet in vielen mobilen Endgeräten Anwendung, wie z.B. im Handy bei bestimmten Audioprogrammen oder der WLAN-Nutzung. Die beiden Benchmarks unterscheiden sich jeweils nur in der Bitlänge. Der *g721.marcuslee_decoder* ist ein Audiodecoder, *latnrm_32_64* ein sogenannter Lattice Filter, welcher als digitaler Filter benutzt wird. *Trellis* ist ein Decoder, der in der digitalen Signalverarbeitung genutzt wird und *edge_detect* ist ein Bildfilter, welcher Kanten in Bildern erkennen kann.

Die Benchmark-Sets wurden so ausgewählt, dass für jedes Set mindestens ein praxisnahes Beispiel gefunden werden kann, wo diese angewandt werden. Zum Beispiel kann die Kombination der Benchmarks aus Set4 eine Videotelefonie simulieren, da hier Audio- und Videodecoder zum Einsatz kommen. Zu Set3 würde als Beispiel der GSM Signal Empfang passen, da hier das Signal decodiert und auf Korrektheit überprüft wird. Im Gegensatz zu Set4 könnte bei Set2 zwar keine Telefonie stattfinden, da hierzu Filter fehlen, jedoch könnte ein Video aus dem interenen Speicher oder als Stream aus dem Internet abgespielt werden. Set1 ist ein synthetisches Benchmark-Set, bei dem ein Task existiert, der eine geringe Codegröße, aber eine hohe WCET besitzt, um diesen Einfluss zu überprüfen.

4.2 Scratchpad Heuristik Ergebnisse

Die Multi-Task Scratchpad Memory Allocation wurde mit den verschiedenen in Kapitel 3.1.1 vorgestellten Heuristiken (WCET, Codegröße und Kombination der beiden) auf alle Benchmarks mit verschiedenen SPM-Größen angewendet. Für die aus den Ergebnissen erstellten Diagramme gilt: Auf der X-Achse ist die SPM-Größe in Prozent im Verhältnis zur Codegröße des Benchmark-Sets angegeben. Die Y-Achse stellt die prozentuale Reduktion der WCET des Bechmark-Sets dar. In den Diagrammen werden für die drei Heuristiken unterschiedliche Symbole verwendet, um die Ergebnisse darin zu markieren: Das Quadrat zeigt die Werte der WCET-Heuristik an, für die Codegrößen-Heuristik ist ein Raute eingezeichnet und das Dreieck markiert die Werte der Kombi Heuristik. Die Ergebnisse der Benchmarks werden auf ganze Zahlen gerundet und jeweils mit den Werten eines System ohne Cache und ohne die Speicheroptimierungen verglichen, wobei das Optimierungslevel O2 genutzt wird.

In Abbildung 4.1 ist ein Diagramm zu sehen, bei der die Scratchpad Memory Allocation mit allen Heuristiken auf das Benchmark-Set1 angewandt wurde. Die WCET-Heuristik erreicht schon bei einer kleinen SPM-Größe gute Ergebnisse: bereits bei 5% SPM-Größe wird eine Reduktion der WCET um 38% erreicht, was auch die Geringste zu erreichende Reduktion darstellt. Unterhalb dieser SPM-Größe kann keine Reduktion erreicht werden. Wenn nun mehr SPM-Speicher zur Verfügung steht (ab 20% der Codegröße), kann die Heuristik als maximale Reduktion 42% erzeugen. Die Codegrößen-Heuristik benötigt den meisten Speicher, um gute Ergebnisse zu erzielen. Es werden 20% SPM-Größe benötigt, um eine Reduktion der WCET von 41% zu erreichen. Hierbei ist zu beachten, dass dies auch der höchste Wert ist, den die Heuristik erzielt. Die minimale Reduktion ist anders

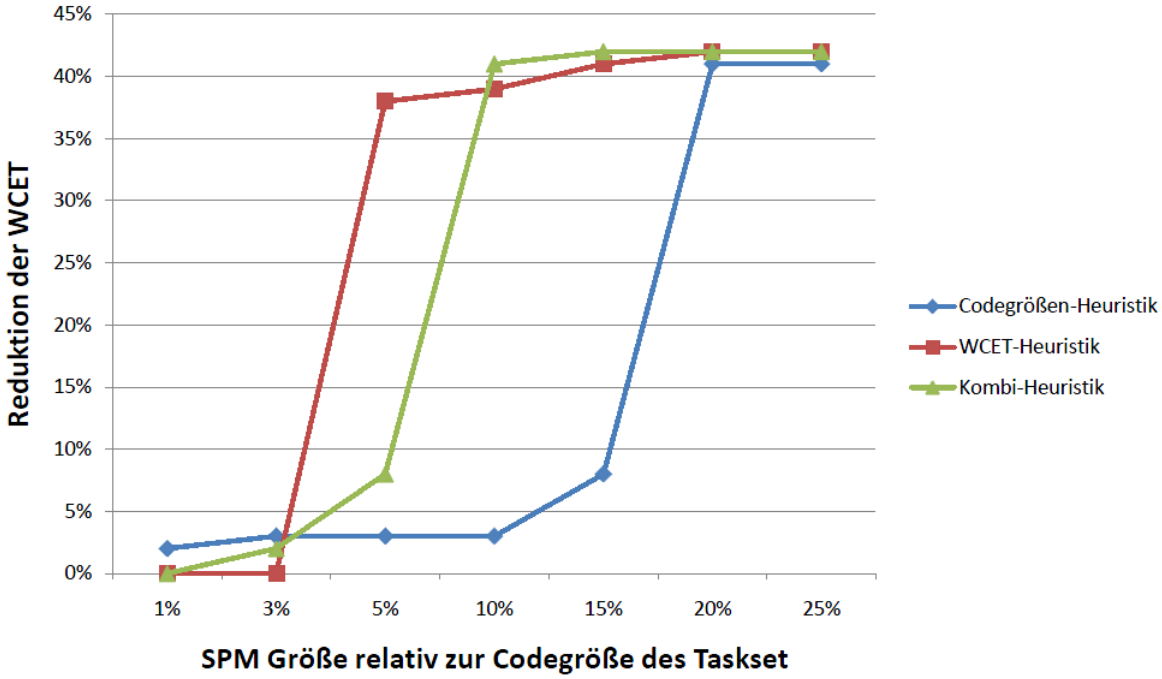

Abbildung 4.1: Multi-Task SPM Allocation mit Heuristiken angewandt auf Benchmark-Set1

wie bei den anderen Heuristiken: Es wird 2% Reduktion bei 1% SPM-Größe erzielt. Die Kombi-Heuristik benötigt im Gegensatz zur WCET-Heuristik etwas mehr Speicher um gute Ergebnisse zu erzielen. Sie erreicht bei einer SPM-Größe von 10% eine Reduktion der WCET um 41%. Dies ist knapp unter dem Höchstwert der zu erreichenden Reduktion der Heuristik von 42% ab 20% SPM-Größe. Den geringsten Wert von 1% Reduktion erlangt die Heuristik bei 3% SPM-Größe, unter dieser tritt keine Reduktion ein.
Der sprunghafte Anstieg von allen Heuristiken bei diesem Benchmark-Set kann dadurch erklärt werden, dass genau mit der entsprechenden SPM-Größe der Task edge_detect genug SPM-Speicher von der jeweiligen Heuristik zugewiesen bekommt, um die WCET dieses Tasks stark zu reduzieren.

Im Diagramm in Abbildung 4.2 wurde die Scratchpad Memory Allocation auf das Benchmark-Set2 angewandt. Hier erzielt die WCET-Heuristik schon bei 1% SPM-Größe eine Reduktion der WCET von knapp 18%, was in diesem Fall auch der geringste Wert der zu erreichenden Reduktion der Heuristik ist. Dies liegt daran, dass der Task *crc* selbst von kleinen SPM-Größen stark profitiert und die WCET-Heuristik diesem selbst bei 1% SPM-Größe 64 Byte SPM-Speicher zuweist. Der Unterschied zu den anderen Heuristiken

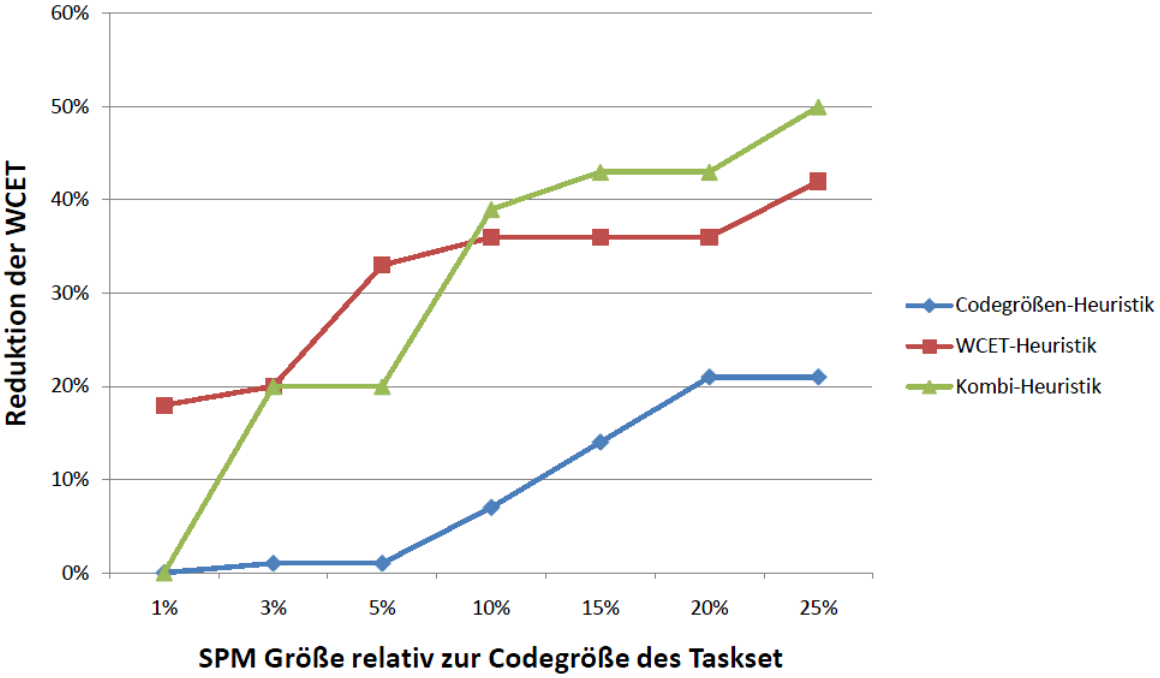

Abbildung 4.2: Multi-Task SPM Allocation mit Heuristiken angewandt auf Benchmark-Set2

in Bezug auf die Reduktion der WCET bei dieser SPM-Größe, ist mit knapp 18% entsprechend groß. Die anderen Heuristiken weisen dem Task *crc* keinen SPM-Speicher zu, da er eine sehr geringe Codegröße im Vergleich zum *h264dec_ldecode_block* hat, welchem sie stattdessen den SPM Speicher zuweisen. Als höchsten Wert der Reduktion werden von der WCET-Heuristik 42% erreicht. Die Codegrößen-Heuristik kann erst bei großen SPM-Größen gute Ergebnisse liefern. Hierbei wird eine Reduktion der WCET von 21% ab 20% SPM-Größe erzielt, was auch der maximal erreichbaren Reduktion entspricht. Die im Vergleich geringe WCET-Reduktion der Codegrößen-Heuristik liegt daran, dass die Tasks in diesem Benchmark-Set zwar alle ähnliche WCETs besitzen, jedoch nur der h264dec_ldecode_block eine sehr hohe Codegröße besitzt. Dieser macht 95% der Codegröße des Tasksets aus. Die Codegrößen-Heuristik weist dann nur diesem Task Speicher zu, was dazu führt, dass die Reduktion der WCET geringer ausfällt als bei den anderen Heuristiken. Die Kombi-Heuristik kann die maximale Reduktion der WCET-Heuristik übertreffen und erreicht eine Reduktion von 50% bei einer SPM-Größe von 25%. Als geringste Reduktion werden 20% bei 3% SPM-Größe erreicht, wobei unter dieser keine Reduktion eintritt.

In Abbildung 4.3 ist ein Diagramm zu sehen, bei der die Scratchpad Memory Allocation mit allen Heuristiken auf das Benchmark-Set3 angewandt wurde. Die WCET-Heuristik kann hier bei 5% SPM-Größe eine Reduktion der WCET von 22% erreichen. Wenn die SPM-Größe auf 25% steigt, kann die Heuristik sogar eine Reduktion von 37% erzielen, was die maximale Reduktion dieser Heuristik ist. Als minimale Reduktion werden 22% bei 5% SPM-Größe erreicht, unter dieser ist keine Reduktion möglich. Die Codegrößen-Heuristik kann eine minimale Reduktion von 24% bei 10% SPM-Größe erzielen, unter dieser SPM-Größe kann keine Reduktion erreicht werden. Ab 10% SPM-Größe kann die Codegrößen-Heuristik eine höhere Reduktion der WCET von 24% bei 10% SPM-Größe erzielen, wobei dieser knapp unter dem maximal zu erreichenden Wert von 25% ab 15% SPM-Größe liegt. Der Maximalwert der Reduktion der WCET, den die Kombi-Heuristik ab 10% SPM-Größe erzielen kann, beträgt 25%. Bei 1% SPM-Größe wird die minimale Reduktion von 1% erreicht. Auffällig an diesem Diagramm ist, dass bei geringen SPM-Größen Verschlechterungen der WCET eintreten. Hierbei bekommt der Task *gsm_decode* durch die Heuristiken einen geringen Anteil am SPM zugeteilt. Die Verschlechterung der WCET des Tasks entsteht durch die originale SPM Allocation, die eingesetzt wird, um die eigentliche SPM Allocation durchzuführen. Dies ist ein Fehler, welcher aus zeitlichen Gründen nicht im Rahmen dieser Arbeit behoben werden konnte.

Im Diagramm in Abbildung 4.4 wurde die Scratchpad Memory Allocation auf das Benchmark-Set4 angewandt. Die WCET-Heuristik erreicht hier bei 15% SPM-Größe eine Reduktion der WCET von 29%. Dieser Wert kann auf 32% erhöht werden, wenn die SPM-Größe auf 25% steigt. Als minimale Reduktion der WCET erreicht die Heuristik 4% bei 5% SPM-Größe. Die Codegrößen-Heuristik stagniert bei einer Reduktion von 10%, was auch dem Maximum der Reduktion entspricht. Dies passiert, da in dem Benchmark-Set4 der Task *h264dec_ldecode_block* eine sehr hohe Codegröße besitzt, welche 87% des Tasksets ausmacht, aber keine sehr hohe WCET, welche nur 15% der Gesamt-WCET ausmacht. Die Heuristik weist diesem Task fast den kompletten SPM zu, da dieser aber nur einen geringen Einfluss auf die WCET hat, ändert sich diese kaum. Als minimale Reduktion der WCET kann die Codegrößen-Heuristik 1% bei 10% SPM-Größe erreichen, unter dieser SPM-Größe wird keine Reduktion erzielt. Die Kombi-Heuristik kann ähnliche Werte wie die WCET-Heuristik erreichen. Das Maximum der Reduktion liegt auch hier bei 31% bei 25% SPM-Größe, wobei solch hohe Werte erst ab 20% SPM-Größe erzielt werden. Als kleinster Wert wird 5% Reduktion der WCET bei 10% SPM-Größe erreicht, wobei unter dieser SPM-Größe

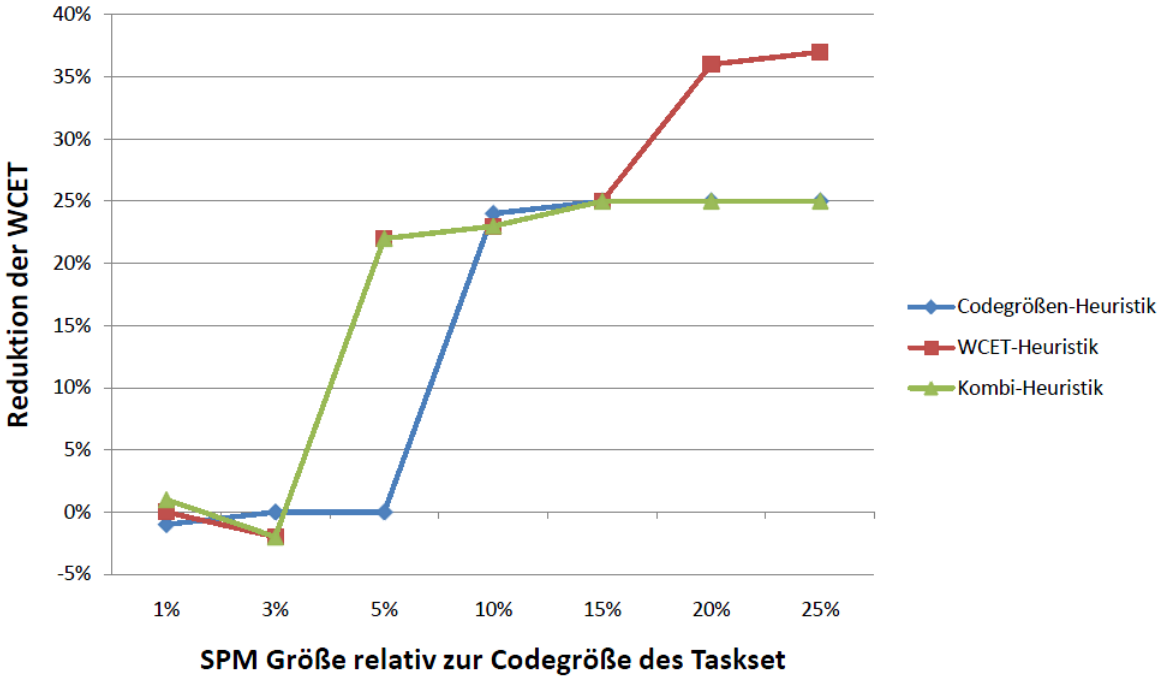

Abbildung 4.3: Multi-Task SPM Allocation mit Heuristiken angewandt auf Benchmark-Set3

keine Reduktion möglich ist. Besonders die Verschlechterungen der WCET stechen bei geringen SPM-Größen hervor. Bei allen Heuristiken wird in diesem Fall entweder dem Task *fft_256* oder dem Task *latnrm_32_64* ein geringer Anteil am SPM-Speicher zugewiesen. Die Verschlechterungen dieser Tasks entstehen auch hier wieder durch die originale SPM Allocation.

Um die Heuristiken vergleichen zu können, wird noch der Durchschnitt der maximalen Reduktionen für jede Heuristik über alle Benchmark-Sets berechnet. Die WCET-Heuristik ist die beste der drei Heuristiken. Sie erreicht im Durchschnitt mit 38,35% Reduktion der WCET das beste Ergebnis. Jedoch kann die Kombi-Heuristik mit 37% Reduktion auch ein sehr gutes Ergebnis erzielen und ist daher die zweitbeste Heuristik. Die Codegrößen-Heuristik kann im Durchschnitt nur 24,25% Reduktion erreichen und ist damit die schlechteste Heuristik. Zusätzlich kann die WCET-Heuristik im Gegensatz zur Kombi-Heuristik meist schon bei geringeren SPM-Größen gute Ergebnisse erzielen. Dass die Codegrößen-Heuristik ein so schlechtes Ergebnis erzielt, liegt daran, dass falls ein Task eine geringe WCET besitzt, aber eine hohe Codegröße, dieser durch die Heuristik viel SPM zugewiesen bekommt. Da die WCET dieses Tasks

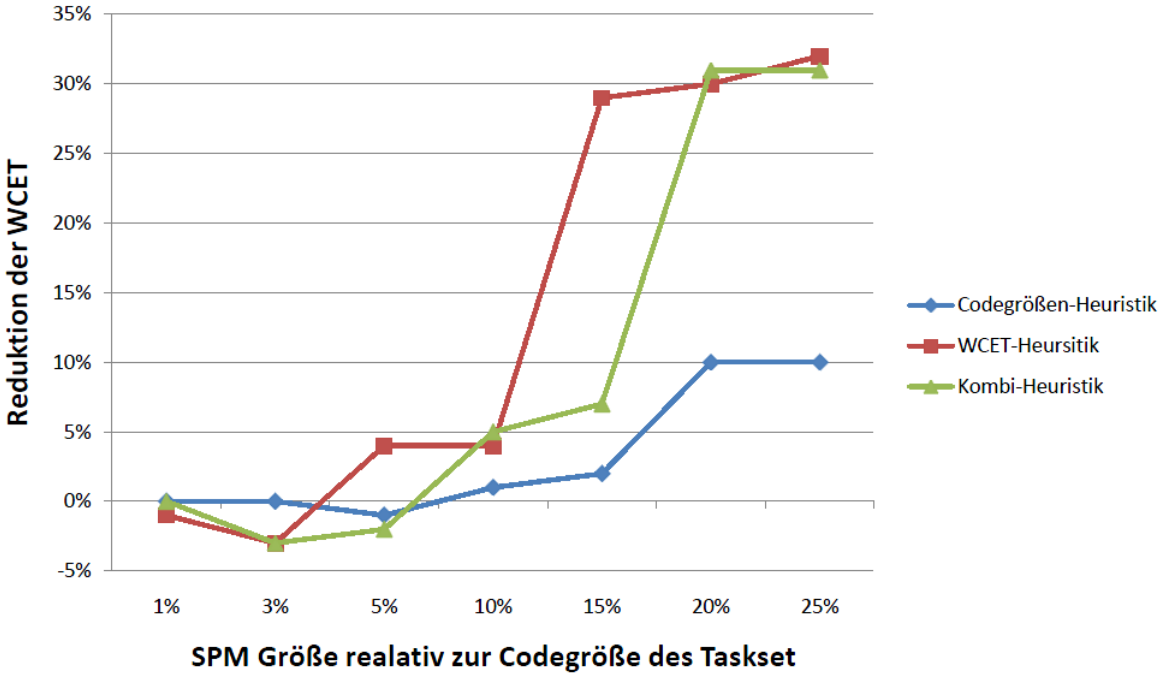

Abbildung 4.4: Multi-Task SPM Allocation mit Heuristiken angewandt auf Benchmark-Set4

aber nicht so eine große Auswirkung auf die Gesamt-WCET hat, sinkt diese oft nur marginal. Bei der WCET-Heuristik bekommt genau der Task am meisten SPM zugewiesen, der auch die höchste WCET besitzt. Dadurch sinkt die Gesamt-WCET stärker. Die Kombi-Heuristik besitzt die Vor- und Nachteile beider Heuristiken, womit in bestimmten Situationen die besten Ergebnisse erreicht werden, jedoch insgesamt nicht so gute wie bei der WCET-Heuristik.

4.3 Cache Partitioning und Memory Content Selection Ergebnisse

In diesem Abschnitt geht es darum herauszufinden, ob eine Vorauswahl der zu cachenden Funktionen durch die Memory Content Selection eine höhere Reduktion der WCET bringt, als wenn dies nicht geschieht. Hierzu wurde das Cache Partitioning einmal ohne Memory Content Selection und einmal mit Memory Content Selection auf die Benchmark-Sets angewandt und jeweils die Reduktion der WCET berechnet.

Für die aus den Ergebnissen erstellten Diagramme gilt: Auf der X-Achse ist die Cache-Größe in Prozent im Verhältnis zur Codegröße des Benchmark-Sets angegeben. Die Y-Achse stellt die prozentuale Reduktion der WCET des Benchmark-Sets dar. In den Diagrammen werden für die zwei Optimierungen verschiedene Symbole verwendet, um die erzielten Ergebnisse darin zu markieren. Die Werte für das Cache Partitioning (CP) werden durch das Quadrat dargestellt und die Raute zeigt die Werte an, wenn zusätzlich die Memory Content Selection (CP + MCS) ausgeführt wird. Die Ergebnisse der Benchmarks werden auf ganze Zahlen gerundet und jeweils mit den Werten eines System ohne Cache und ohne die Speicheroptimierungen verglichen, wobei das Optimierungslevel O2 genutzt wird.

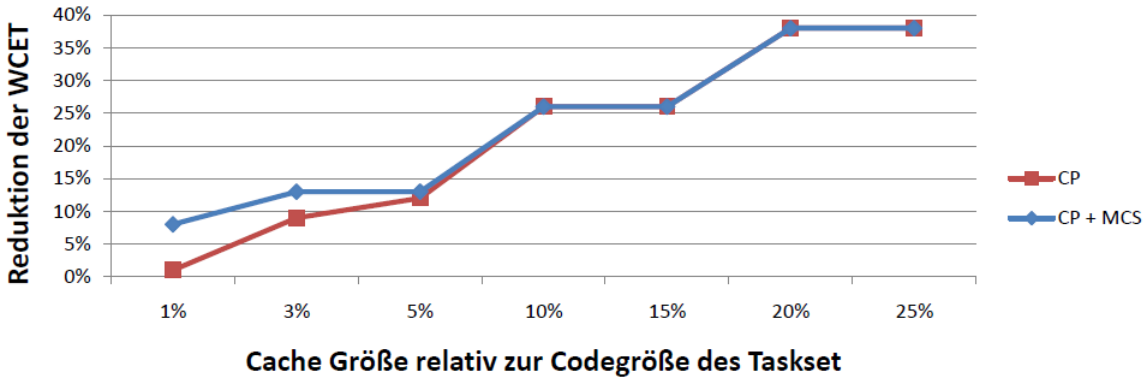

Abbildung 4.5: Cache Partitioning ohne/mit Memory Content Selection angewandt auf Benchmark-Set1

Im Diagramm in Abbildung 4.5 wurden die Optimierungen auf das Benchmark-Set1 angewandt. Das Cache Partitioning *ohne* Memory Content Selection erzielt hierbei eine minimale Reduktion der WCET von 1% bei 1% Cache-Größe. Als maximale Reduktion kann das Cache Partitioning alleine eine Reduktion der WCET von 37% ab 20% Cache-Größe. Wenn das Cache Partitioning *mit* Memory Content Selection ausgeführt wird, dann können selbst bei kleinen Cache-Größen höhere Reduktionen erzielt werden. Die minimale Reduktion beträgt 8% bei 1% Cache-Größe, wobei die maximale Reduktion auch 37% ab 20% Cache-Größe beträgt. Allgemein erzielen das Cache Partitioning allein und mit Memory Content Selection ab einer Cache-Größe von 10% die gleichen Ergebnisse. Die zusätzliche Ausführung der Memory Content Selection bringt bei kleinen Cache-Größen einen großen Vorteil: Bei 1% Cache-Größe wird eine um 7% höhere und bei 3% Cache-Größe eine um 4% höhere Reduktion erzielt.

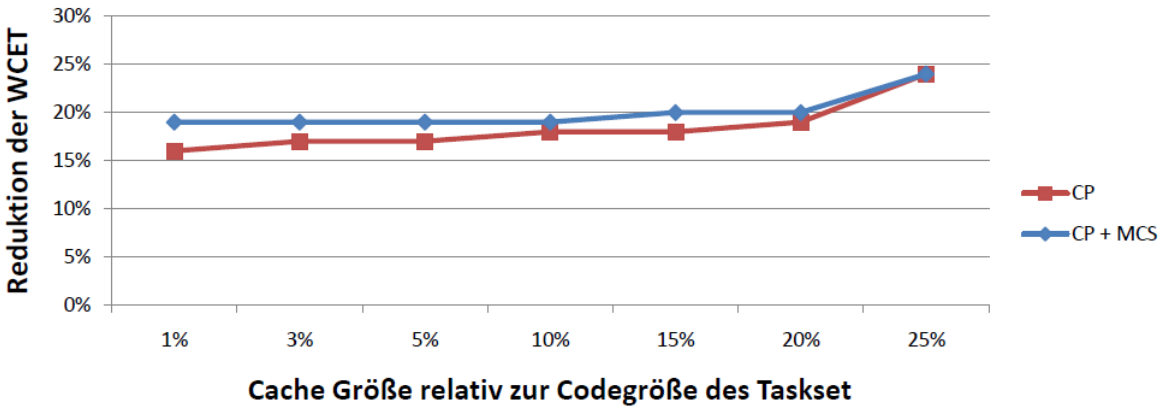

Abbildung 4.6: Cache Partitioning ohne/mit Memory Content Selection angewandt auf Benchmark-Set2

In Abbildung 4.6 ist ein Diagramm dargestellt, in dem die Optimierungen auf das Benchmark-Set2 angewandt wurden. Das Cache Partitioning *alleine* erreicht als minimale WCET Reduktion 16% bei 1% Cache-Größe und eine maximale Reduktion von 24% bei 25% Cache-Größe. Wenn zusätzlich die Memory Content Selection ausgeführt wird, wird die WCET Reduktion bei 1% Cache-Größe um 3% verbessert, wobei die maximale Reduktion gleich der Reduktion des Cache Partitioning alleine ist. Allgemein ist zu beobachten, dass die Reduktion der WCET um im Durchschnitt 2% höher ausfällt, wenn die Memory Content Selection zusätzlich ausgeführt wird, außer bei der maximal getesteten Cache-Größe von 25%.

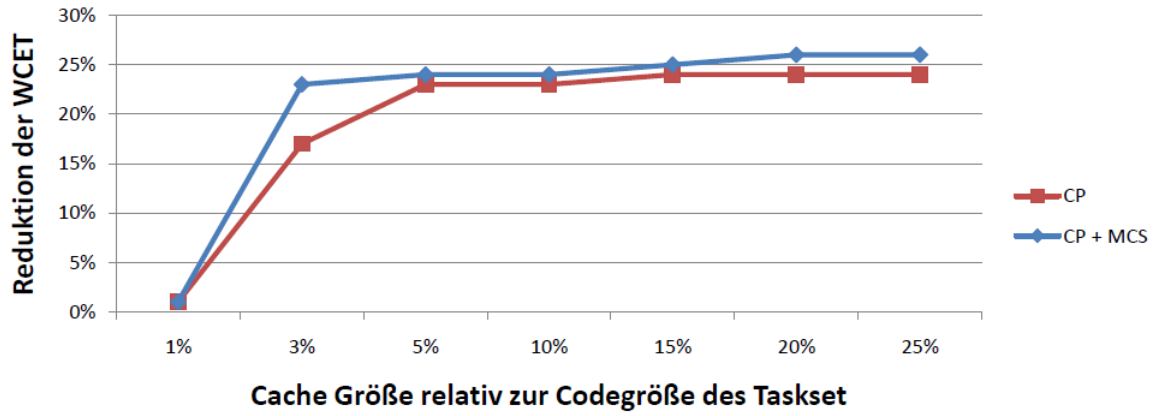

Abbildung 4.7: Cache Partitioning ohne/mit Memory Content Selection angewandt auf Benchmark-Set3

Im Diagramm in Abbildung 4.7 wurden die Optimierungen auf das Bechmark-Set3 angewandt. Wenn das Cache Partitioning *alleine* angewendet wird, erzielt es eine minimale Reduktion der WCET von 1% bei 1% Cache-Größe und eine maximale von 24% bei 25% Cache-Größe. Bei der zusätzlichen Anwendung der Memory Content Selection ist die minimale Reduktion gleich, jedoch ist die maximale Reduktion um 1% höher bei 25% Cache-Größe. Allgemein liefert auch in diesem Benchmark-Set das Cache Partitioning *mit* Memory Content Selection eine Reduktion die im Durchschnitt 3% besser ist, als *ohne* die Ausführung der Memory Content Selection.

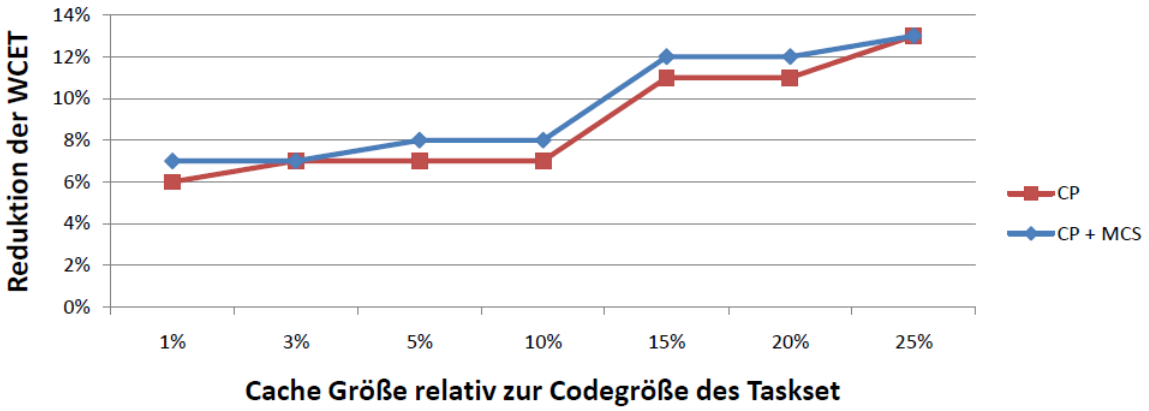

Abbildung 4.8: Cache Partitioning ohne/mit Memory Content Selection angewandt auf Benchmark-Set4

Die Abbildung 4.8 zeigt ein Diagramm, in dem die Optimierungen auf das Benchmark-Set4 angewandt wurden. Auch hier ist zu erkennen, dass, wenn zusätzlich die Memory Content Selection angewandt wird, meist bessere Ergebnisse erzielt werden. Das Cache Partitioning *alleine* erzielt eine minimale Reduktion der WCET von 6% bei einer Cache-Größe von 1% und eine maximale von 13% bei 25% Cache-Größe. Im Gegensatz dazu erzielt das Cache Partitioing mit Memory Content Selection eine minimale Reduktion der WCET von 7% bei einer Cache-Größe von 1% und eine maximale von 13% bei 25% Cache-Größe. Bei den Werten zwischen dem Minimum und dem Maximum kann das Cache Partitioning *mit* Memory Content Selection eine um durchschnittlich 1% höhere Reduktion der WCET erzielen, als das Cache Partitioning *alleine*.

Insgesamt wird durch die zusätzliche Ausführung der Memory Content Selection eine zusätzliche Reduktion der WCET von durchschnittlich 2% erzielt. Besonders bei kleinen Cache-Größen lohnt sich die Auswahl der zu cachenden Funktionen, da der wenige Platz dadurch effektiver genutzt werden kann.

4.4 Multi-Task Scrachtpad Memory Allocation und Cache Partitioning Ergebnisse

Dieser Abschnitt beschäftigt sich damit, ob die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning zusammen arbeiten können und inwiefern sich die Anwendung von beiden Optimierungen zusammen überhaupt lohnt. Dazu werden beide Optimierungen mit verschiedenen Parametern auf die Benchmark-Sets angewandt und die Reduktion der WCET berechnet. Diese Werte werden dann mit denen der einzelnen Optimierungen verglichen.

Für die aus den Ergebnissen erstellten Tabellen gilt: Auf der X-Achse ist die Cache-Größe in Prozent im Verhältnis zur Codegröße des Benchmark-Sets angegeben. Die Y-Achse stellt die SPM-Größe in Prozent im Verhältnis zur Codegröße des Benchmark-Sets dar. In den Zellen ist jeweils die Reduktion der WCET angegeben, die bei den entsprechenden Cache- bzw. SPM-Größen erreicht wird. Bei der Multi-Task Scratchpad Memory Allocation wird die WCET-Heuristik angewandt, da dies die Beste der drei Heuristiken ist. Die Ergebnisse der Benchmarks werden auf ganze Zahlen gerundet und jeweils mit den Werten eines System ohne Cache und ohne die Speicheroptimierungen verglichen, wobei das Optimierungslevel O2 genutzt wird.

In der Tabelle 4.2 sind Ergebnisse zu sehen, bei denen die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning auf das Benchmark-Set1 angewandt wurden. Die Multi-Task Scratchpad Memory Allocation alleine erzielt bei diesem Benchmark-Set bereits ab 5% SPM-Größe eine WCET Reduktion von mindestens 38% und eine maximale Reduktion der WCET von 42% ab 20% SPM-Größe. Die Reduktion der WCET nur durch das Cache Partitioning steigt etwas langsamer an. Es erreicht eine minimale Reduktion von 1% bei 1% Cache-Größe und steigt dann bis auf 37% Reduktion ab 20% Cache-Größe an. Es fällt auf, dass ab 5% SPM-Größe eine starke Reduktion der WCET eintritt, egal ob mit oder ohne Cache Partitioning. Falls das Cache Partitioning ausgeführt wird, kann es die WCET bei SPM-Größen unter 20%

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	1%	9%	12%	26%	26%	37%	37%
1%	0%	1%	11%	12%	27%	27%	37%	37%
3%	0%	1%	12%	12%	27%	27%	37%	37%
5%	38%	38%	40%	40%	40%	41%	41%	41%
10%	39%	40%	41%	42%	42%	42%	42%	42%
15%	41%	41%	42%	42%	42%	42%	42%	42%
20%	42%	42%	42%	42%	42%	42%	42%	42%
25%	42%	42%	42%	42%	42%	42%	42%	42%

Tabelle 4.2: Multi-Task SPM Allocation und Cache Partitioning angewandt auf Benchmark-Set1

geringfügig verbessern, ansonsten hat es keinen Effekt. Unter 5% SPM-Größe kann die SPM Allocation keinen Effekt erzielen, jedoch kann hier bei allen Cache-Größen das Cache Partitioning eine Reduktion der WCET erreichen, welche bis zu 5% unter der maximalen Reduktion liegt. Es zeigt sich, dass sowohl das Cache Partitioning als auch die Multi-Task SPM Allocation die Hotspots des Benchmark-Sets gut beschleunigen können und daher die Kombination nur bei geringen Speichergrößen (z.b. 3% SPM und 3% Cache) sinnvoll ist.

In der zweiten Tabelle 4.3 sind die Ergebnisse festgehalten, die erreicht werden, wenn die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning auf das Benchmark-Set2 angewandt werden. Die Multi-Task SPM Allocation kann hier selbst alleine bei geringen SPM-Größen gute Ergebnisse erzielen: schon bei 1% SPM-Größe wird eine Reduktion der WCET von 18% erreicht, welche bei größerem SPM auf bis zu 42% Reduktion anwächst. Das Cache Partitioning erreicht ohne die Multi-Task SPM Allocation eine minimale Reduktion der WCET von 16% bei 1% Cache-Größe, wobei diese sich auf bis zu 23% bei größerem Cache steigert. Bei diesem Benchmark-Set lohnt sich die Kombination der Optimierungen immer. Selbst bei kleinen SPM und

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	16%	17%	17%	18%	18%	19%	23%
1%	18%	22%	22%	29%	37%	37%	38%	38%
3%	20%	22%	22%	31%	37%	37%	38%	38%
5%	33%	35%	35%	42%	50%	50%	50%	51%
10%	36%	36%	36%	45%	53%	53%	54%	54%
15%	36%	42%	44%	45%	53%	53%	54%	54%
20%	36%	42%	44%	50%	53%	53%	54%	54%
25%	42%	42%	44%	50%	53%	53%	54%	54%

Tabelle 4.3: Multi-Task SPM Allocation und Cache Partitioning angewandt auf Benchmark-Set2

Cache-Größen wird durch die Kombination eine größere Reduktion der WCET erreicht: Bei 1% SPM-Größe und 1% Cache-Größe liegt die Reduktion bei 22%, was das Minimum der zu erreichenden Reduktion darstellt. Einzeln würden die Optimierungen jeweils nur 18% bzw. 16% Reduktion erzielen. Die Reduktionen der WCET addieren sich nicht, da es eine Überschneidung der zu optimierenden Hotspots des Benchmark-Sets gibt und diese nur von einer Optimierung beschleunigt werden können. Als maximale Reduktion der WCET kann die Kombination der Optimierungen 54% erreichen. Dieser Wert kann ab 10% SPM-Größe und ab 20% Cache-Größe erzielt werden.

In Tabelle 4.4 sind die Ergebnisse zu sehen, wo die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning auf das Bechnmark-Set3 angewandt wurden. Die Multi-Task SPM Allocation kann bei diesem Benchmark-Set erst ab 5% SPM-Größe eine größere Reduktion der WCET von 22% erzielen, welche sich auf bis zu 37% steigern lässt. Als minimale Reduktion der WCET wird 22% bei 5% SPM-Größe erreicht, unter dieser SPM-Größe kann keine Reduktion erreicht werden. Jedoch wird bei 3% SPM-Größe eine Verschlechterung der WCET erzielt, was an der originalen SPM Allocation liegt. Das Cache Partitioning alleine kann ab 3% Cache-Größe eine gute

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	1%	17%	23%	23%	24%	24%	24%
1%	0%	0%	0%	23%	23%	23%	24%	24%
3%	-2%	0%	1%	18%	22%	22%	22%	22%
5%	22%	23%	23%	23%	23%	23%	23%	23%
10%	23%	24%	24%	24%	24%	24%	24%	25%
15%	25%	25%	25%	25%	25%	25%	25%	25%
20%	36%	36%	36%	36%	37%	37%	37%	37%
25%	37%	37%	37%	37%	37%	37%	38%	38%

Tabelle 4.4: Multi-Task SPM Allocation und Cache Partitioning angewandt auf Benchmark-Set3

Reduktion der WCET von 17% erreichen, welche auf ihr Maximum bei 24% ab 15% Cache-Größe ansteigt. Mit einer Cache-Größe von 1% wird eine Reduktion der WCET von 1% erreicht. Die Kombination der Optimierungen bringt keinen großen Vorteil, da entweder das Cache Partitioning oder die Multi-Task SPM Allocation die Hotspots des Benchmark-Sets beschleunigen. Als maximale Reduktion erreicht die Kombination der Optimierungen 38% bei 25% SPM-Größe und ab 20% Cache-Größe. Die Ergebnisse der Multi-Task SPM Allocation können durch das Cache Partitioning meist nur um 1-2% verbessert werden, wohingegen die Ergebnisse die das Cache Partitioning alleine erreicht durch die Kombination um 14% gesteigert werden können.

In der vierten Tabelle 4.5 wurde die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning zusammen auf das Benchmark-Set4 angewandt. Die Multi-Task SPM Allocation erzielt hierbei ohne das Cache Partitioning eine Reduktion der WCET von bis zu 32% bei 25% SPM-Größe, wobei höhere Reduktionen erst ab 15% SPM-Größe erreicht werden. Das Cache Partitioning kann alleine eine Reduktion der WCET von 6% bei 1% Cache-Größe bis zu 13% bei 25% Cache-Größe erreichen. Die Kombination der beiden Optimierungen erreicht eine maximale Reduktion der WCET von 36% ab

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	6%	7%	7%	7%	11%	11%	13%
1%	-1%	4%	5%	5%	5%	9%	9%	11%
3%	-3%	3%	4%	5%	5%	9%	9%	11%
5%	4%	7%	7%	8%	11%	11%	11%	11%
10%	4%	7%	7%	8%	11%	11%	11%	11%
15%	29%	30%	30%	32%	35%	35%	35%	35%
20%	30%	30%	30%	32%	36%	36%	36%	36%
25%	32%	32%	32%	33%	36%	36%	36%	36%

Tabelle 4.5: Multi-Task SPM Allocation und Cache Partitioning angewandt auf Benchmark-Set4

20% SPM-Größe und ab 10% Cache-Größe. Allgemein wird das Ergebnis, was die Multi-Task SPM Allocation erreicht, durch das Cache Partitioning um durchschnittlich 7% verbessert. Demnach ergänzen sich die Optimierungen in diesem Benchmark-Set sehr gut.

Allgemein kommt es bei dieser Kombination der Optimierungen sehr auf die Benchmark-Sets an. Es kann passieren, dass die Multi-Task SPM Allocation schon alle Hotspots des Benchmark-Sets beschleunigt und das Cache Partitioning keine signifikante Verbesserung mehr erzielen kann. Dies kann jedoch auch andersherum sein: Falls die Multi-Task SPM Allocation keine oder nur geringe Reduktionen erzielen kann, kann das Cache Partitioning die Hotspots beschleunigen.

4.5 Multi-Task SPM Allocation, Cache Partitioning und Memory Content Selection Ergebnisse

In diesem Abschnitt geht es darum, ob die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning mit Memory Content Selection zusammen arbeiten

können und inwiefern sich die Anwendung von diesen drei Optimierungen zusammen überhaupt lohnt. Dazu wurden alle Optimierungen mit verschiedenen Parametern auf das Benchmark-Set angewandt und die Reduktion der WCET berechnet. Diese Werte werden dann mit denen der einzelnen Optimierungen verglichen und den Ergebnissen aus Kapitel 4.4. Für die aus den Ergebnissen erstellten Tabellen gilt: Die X-Achse zeigt die Cache-Größe in Prozent im Verhältnis zur Codegröße des Benchmark-Sets, wobei die Y-Achse die SPM-Größe in Prozent im Verhältnis zur Codegröße des Benchmark-Sets darstellt. In den Zellen ist jeweils die Reduktion der WCET angegeben, die bei den entsprechenden Cache- bzw. SPM-Größen erreicht wird. Bei der Multi-Task SPM Allocation wurde wieder die WCET-Heuristik benutzt. Die Ergebnisse der Benchmarks werden auf ganze Zahlen gerundet und jeweils mit den Werten eines System ohne Cache und ohne die Speicheroptimierungen verglichen, wobei das Optimierungslevel O2 genutzt wird.

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	7%	12%	12%	24%	24%	37%	37%
1%	0%	2%	12%	12%	14%	14%	37%	37%
3%	0%	2%	12%	12%	14%	14%	37%	37%
5%	38%	39%	40%	40%	40%	40%	41%	41%
10%	39%	41%	42%	42%	42%	42%	42%	42%
15%	41%	41%	42%	42%	42%	42%	42%	42%
20%	42%	42%	42%	42%	42%	42%	42%	42%
25%	42%	42%	42%	42%	42%	42%	42%	43%

Tabelle 4.6: Multi-Task SPM Allocation und Cache Partitioning mit Memory Content Selection angewandt auf Benchmark-Set1

In der Tabelle 4.6 wurde die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning mit Memory Content Selection zusammen auf das Benchmark-Set1 angewandt. Die Multi-Task SPM Allocation erzielt hierbei alleine eine Reduktion der

WCET von bis zu 42% ab 20% SPM-Größe, wobei höhere Reduktionen erst ab 5% SPM-Größe erreicht werden, davor kann die Optimierung das Benchmark-Set nicht beschleunigen. Das Cache Partitioning nur mit Memory Content Selection kann eine Reduktion der WCET von 7% bei 1% Cache-Größe und bis zu 37% bei 25% Cache-Größe erreichen. Die Kombination der Optimierungen kann eine maximale Reduktion der WCET von 43% bei 25% SPM und Cache-Größe erreichen, welche nur durch zusätzliche die Memory Content Selection möglich wird, ohne sie beträgt die Reduktion 42% (vlg Tabelle 4.2). Ähnlich gute Werte von 42% werden ab 10% SPM-Größe und 3% Cache-Größe erzielt. Die zusätzliche Anwendung der Memory Content Selection kann bei der Kombination der Optimierungen bei geringen Cache-Größen einen Vorteil erzielen. Im Gegensatz zum Cache Partitioning ohne die Memory Content Selection steigt die erreichte Reduktion der WCET um 2%. Auffällig hierbei ist, dass bei 1% Cache-Größe ohne die Multi-Task SPM Allocation 7% Reduktion der WCET erreicht wird, mit jedoch nur 2 bzw. 3% erzielt werden. Dies ist der Fall, da die Multi-Task SPM Allocation Programmcode in den SPM verschiebt, was zur Folge hat, dass durch das geänderte Speicherlayout wechselseitig ausgeführte Blöcke einzelner Tasks auf die selben Cache-Sets gemappt werden. Dadurch kommt es zu mehr Cache-Misses, wodurch die WCET steigt. Dieser Effekt tritt auch bei 1% SPM-Größe und 10% oder 15% Cache-Größe auf. Hierbei treten 4625 Cache-Misses auf(nicht in der Tabelle abgedruckt), wenn die SPM Allocation angewendet wird, im Gegensatz zu 1752 Cache-Misses, wenn nur das Cache Partitioning mit Memory Content Selection bei 15% Cache-Größe Anwendung findet. Allgemein werden die Hotspots des Benchmark-Sets jeweils entweder von der Multi-Task SPM Allocation oder vom Cache Partitioning mit Memory Content Selection verbessert. Im Gegensatz dazu bringt die Kombination der Optimierungen daher nur sehr geringfügige Verbesserungen.

In der Tabelle 4.7 wurde die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning mit Memory Content Selection zusammen auf das Benchmark-Set2 angewandt. Die Multi-Task SPM Allocation erzielt hierbei alleine eine Reduktion der WCET von bis zu 42% bei 25% SPM-Größe, wobei selbst geringe SPM-Größen schon eine hohe Reduktion der WCET von mindestens 18% erreichen. Das Cache Partitioning nur mit Memory Content Selection kann eine Reduktion der WCET von 19% bei 1% Cache-Größe und bis zu 24% bei 25% Cache-Größe erreichen. Die Kombination der Optimierungen kann eine maximale Reduktion der WCET von 54% bei mindestens 20% Cache-Größe und mindestens 10% SPM-Größe erreichen. Als minimale Reduktion der WCET durch die Kombination der Optimierungen werden 23% bei 1% SPM-Größe und

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	19%	19%	19%	19%	20%	20%	24%
1%	18%	23%	23%	29%	37%	38%	38%	38%
3%	20%	23%	23%	30%	37%	38%	38%	38%
5%	33%	35%	35%	42%	50%	50%	51%	51%
10%	36%	36%	36%	45%	53%	53%	54%	54%
15%	36%	36%	36%	45%	53%	53%	54%	54%
20%	36%	43%	44%	50%	53%	53%	54%	54%
25%	42%	43%	44%	50%	53%	53%	54%	54%

Tabelle 4.7: Multi-Task SPM Allocation und Cache Partitioning mit Memory Content Selection angewandt auf Benchmark-Set2

1% Cache-Größe erzielt. Allgemein wird durch die Kombination von den Optimierungen ein besseres Ergebnis erreicht, als wenn diese einzeln laufen. Die maximale Verbesserung der Reduktion der WCET, ausgehend von der alleinigen Multi-Task SPM Allocation, beträgt durchschnittlich 17%, wenn das Cache Partitioning nach der Multi-Task SPM Allocation ausgeführt wird. Allgemein werden die Ergebnisse im Gegensatz zur Kombination mit dem Cache Partitioning ohne Memory Content Selection bei geringen Cache-Größen von 1-3% um 1% verbessert. Bei diesem Benchmark-Set verteilt sich die Laufzeit über den gesamten Programmcode, so dass die Kombination der Optimierungen gute Ergebnisse erzielen. Einzeln können die Optimierungen zwar Hotspots beschleunigen, es bleibt jedoch genügend Optimierungspotential für darauf folgende Speicheroptimierungen.

In Tabelle 4.8 wurde die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning mit Memory Content Selection zusammen auf das Benchmark-Set3 angewandt. Die Multi-Task SPM Allocation erzielt hierbei alleine eine Reduktion der WCET von bis zu 37% bei 25% SPM-Größe, wobei gute Ergebnisse erst ab einer SPM-Größe von 5% erreicht werden. Unter 5% SPM-Größe kann keine signifikante

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	1%	23%	24%	24%	25%	26%	26%
1%	0%	0%	0%	16%	24%	24%	26%	26%
3%	-2%	0%	0%	22%	22%	22%	24%	24%
5%	22%	23%	23%	23%	23%	23%	24%	24%
10%	23%	24%	24%	24%	24%	24%	25%	25%
15%	25%	25%	25%	25%	25%	26%	26%	26%
20%	36%	36%	36%	36%	37%	37%	37%	38%
25%	37%	37%	37%	37%	38%	38%	38%	38%

Tabelle 4.8: Multi-Task SPM Allocation und Cache Partitioning mit Memory Content Selection angewandt auf Benchmark-Set3

Reduktion der WCET erzielt werden. Das Cache Partitioning nur mit Memory Content Selection kann eine Reduktion der WCET von 1% bei 1% Cache-Größe und bis zu 26% ab 20% Cache-Größe erreichen. Die Kombination der Optimierungen erzielt eine maximale Reduktion der WCET von 38% ab 10% Cache-Größe und 25% SPM-Größe und eine minimale Reduktion von 16% bei 1% SPM-Größe und 5% Cache-Größe. Die Hotspots des Benchmark-Sets können bei der Kombination entweder von der Multi-Task SPM-Allocation oder von dem Cache Partitioning mit Memory Content Selection verbessert werden, wobei die Multi-Task SPM Allocation bessere Werte erzeugt als das Cache Partitioning mit Memory Content Selection. Dadurch werden ausgehend von der Multi-Task SPM Allocation durch die Kombination nur geringfügig bessere Werte erreicht, wobei ausgehend von dem Cache Partitioning mit Memory Content Selection die Kombination viel bessere Werte erzielt (38% statt 26%).

In Tabelle 4.9 wurde die Multi-Task Scratchpad Memory Allocation und das Cache Partitioning mit Memory Content Selection zusammen auf das Benchmark-Set4 angewandt. Die Multi-Task SPM Allocation erzielt hierbei alleine eine Reduktion der WCET von bis zu 32% bei 25% SPM-Größe, wobei erst ab einer SPM-Größe von 15%

SPM Größe relativ zur Codegröße des Taskset \ Cache Größe relativ zur Codegröße des Taskset	**0%**	**1%**	**3%**	**5%**	**10%**	**15%**	**20%**	**25%**
0%	0%	7%	7%	8%	8%	12%	12%	13%
1%	-1%	5%	5%	6%	6%	10%	10%	11%
3%	-3%	4%	5%	6%	6%	9%	10%	11%
5%	4%	7%	7%	8%	11%	11%	11%	11%
10%	4%	7%	7%	8%	11%	11%	11%	11%
15%	29%	30%	30%	32%	35%	35%	35%	35%
20%	30%	30%	30%	33%	36%	36%	36%	36%
25%	32%	32%	30%	33%	36%	36%	36%	36%

Tabelle 4.9: Multi-Task SPM Allocation und Cache Partitioning mit Memory Content Selection angewandt auf Benchmark-Set4

gute Ergebnisse erzielt werden. Unter 15% SPM-Größe kann nur eine geringe Reduktion der WCET erzielt werden. Das Cache Partitioning nur mit Memory Content Selection kann eine Reduktion der WCET von 7% bei 1% Cache-Größe und bis zu 13% bei 25% Cache-Größe erreichen. Die Kombination der Optimierungen kann eine maximale Reduktion der WCET von 36% bei mindestens 10% Cache-Größe und mindestens 20% SPM-Größe erreichen. Als minimale Reduktion der WCET wird durch die Kombination 5% bei 1% SPM-Größe und 1% Cache-Größe erzielt. Durch die zusätzliche Ausführung der Memory Content Selection kann bei geringen Cache-Größen die Reduktion der WCET um 1% verbessert werden. Allgemein wird durch die Kombination von den Optimierungen ein besseres Ergebnis erreicht, wenn die SPM-Größe über 10% liegt, da bei geringen SPM-Größen die orginal SPM Allocation negative Werte erzielt und/oder das Cache Partitioning alleine größere Reduktionen erreicht. Die maximale durchschnittliche Verbesserung der Kombination der Optimierungen, ausgehend von der Multi-Task SPM Allocation bei einer SPM-Größe über 10%, beträgt 5%, wenn das Cache Partitioning mit Memory Content Selection ausgeführt wird. Die Kombination der Optimierungen bei diesem Benchmark-Set lohnt sich in der Hinsicht, dass die Teile, die nicht von der Multi-Task SPM Allocation beschleunigt werden können, durch

das Cache Partitioning mit Memory Content Selection beschleunigt werden. Wenn man von dem Cache Partitioning mit Memory Content Selection ausgeht, lohnt sich die Kombination besonders, da durch zusätzliche Anwendung der Multi-Task SPM Allocation die Reduktion der WCET von bis zu 13% auf bis zu 36% steigern kann.

Insgesamt gilt für diese Kombination das Gleiche wie für die SPM Allocation zusammen mit dem Cache Partitioning. Es kommt sehr auf das Benchmark-Set an, inwiefern die Kombination sich lohnt. Die zusätzliche Ausführung der Memory Content Selection bringt hier bei kleineren Cache-Größen einen Vorteil, so dass bei diesen bessere Ergebnisse erzielt werden. Hierbei werden bei kleinen Cache-Größen 1-2% zusätzliche Reduktion der WCET erreicht.

5 Zusammenfassung

In dieser Arbeit wurde überprüft, inwiefern die drei Speicheroptimierungen Scratchpad Memory Allocation, Cache Partitioning und Memory Content Selection zusammen arbeiten können und ob dies sich lohnt. Hierfür wurde zuerst die Scratchpad Memory Allocation auf mehrere Tasks erweitert. Dies wurde mithilfe von drei in dieser Arbeit entwickelten Heuristiken realisiert: der WCET-Heuristik, der Codegrößen-Heuristik und der Kombi-Heuristik. Danach wurde eine Reihenfolge der Optimierungen festgelegt, um die Änderungen, die an dem Cache Partitioning und der Memory Content Selection vorgenommen werden müssen, zu bestimmen, da diese zusammen arbeiten und auf die Änderungen der Scratchpad Memory Allocation achten müssen. Die Memory Content Selection wurde in das Cache Partitioning integriert, um die maximale Reduktion der WCET zu erzielen. Zum Schluss wurde mithilfe von Benchmarks überprüft, ob die Änderungen, die in dieser Arbeit vorgenommen wurden, funktionieren und ob die Kombination der Optimierungen bei der Reduktion der WCET bessere Werte erreichen kann, als jede Optimierung einzeln. Hierbei war auch die Frage, inwiefern sich die Reduktionen der WCET addieren oder ob Synergie-Effekte auftreten.

Für die in dieser Arbeit entwickelten Heuristiken für die SPM Allocation wurde festgestellt, dass die WCET-Heursitik die besten Ergebnisse liefert. Diese konnte einen Durchschnittswert der maximalen Reduktionen über alle Benchmark-Sets von 38,25% erzielen (im Vergleich zu 24,25% der Codegrößen-Heuristik und 37% der Kombi-Heuristik). Für das Cache Partitioning wurde überprüft, inwiefern sich die Integration der Memory Content Selection lohnt. Die Benchmarkergebnisse zeigen, dass sich nur bei kleinen Cache-Größen größere Unterschiede ergeben, d.h. dass sie sich vor allem bei kleinen Cache-Größen lohnt. Insgesamt konnte für die Kombination der Optimierungen festgestellt werden, dass je nach Benchmark-Set die Optimierungen besser oder schlechter zusammenarbeiten, da sich die Kombination nur gelohnt hat, wenn nicht schon durch eine Optimierung alle Hotspots des Benchmarksets beschleunigt wurden. Jedoch konnten im Vergleich meist bessere Ergebnisse erzielen werden als durch die einzelnen Optimierungen. Die Scratchpad Memory Allocation zusammen mit dem

Cache Partitioning kann durchschnittlich über alle Benchmarksets einen Maximalwert der Reduktion von 42,5% erreichen. Im Vergleich dazu kann die Scratchpad Memory Allocation durchschnittlich nur einen Maximalwert der Reduktion von 38,25% erzielen und das Cache Partitioning nur einen von 24,25%. Falls nun die Memory Content Selection zusätzlich ausgeführt wird, steigert sich dieser Wert bei der Kombination der Optimierungen auf 42,75%, da diese nur einen geringen Einfluss auf die maximale Reduktion hat.

Als Ausblick auf weiterführende Arbeiten wäre es interessant bei der Auswahl der Funktionen beim Cache Partitioning durch die Memory Content Selection zu überprüfen, ob eine Auswahl von Funktionen nicht aus allen Permutationen von Funktionen geschehen sollte. Hierbei könnte es sein, dass aufgrund von Synergieeffekten die Reduktion der WCET höher ausfällt.
Eine Möglichkeit, um die Multi-Task Scratchpad Memory Allocation zu verbessern, wäre statt der Heuristiken Anteile der einzelnen Tasks am SPM über ein ILP zu bestimmen, so dass eine optimale Auswahl von allen Basisblöcken getroffen wird.
Um das Zusammenspiel der Optimierungen zu verbessern, könnte auch nur ein ILP für das Cache Partitioning und die SPM Allocation zusammen erstellt werden, so dass eine optimale Auswahl für beide Optimierungen getroffen wird.

Literaturverzeichnis

[Bit10] Bitkom: *Eingebettete Systeme – Ein strategisches Wachstumsfeld für Deutschland.* `http://www.bitkom.org/files/documents/EingebetteteSysteme_web.pdf`, 2010. – [Online; accessed 05.06.2011]

[CA00] Chiou, Rudolph-L. Devadas S. D. ; Ang, B. S.: Dynamic Cache Partitioning via Columnization. In: *In Proceedings of Design Automation Conference (DAC).* Los Angeles, USA, 2000

[CFW01] C. Ferdinand, M. Langenbach F. Martin M. Schmidt H. Theiling S. T. R. Heckmann H. R. Heckmann ; Wilhelm, R.: Reliable and Precise WCET Determination for a Real-Life Processor. In: *In Proceedings of Embedded Software (EMSOFT)* . Tahoe City, USA, 2001

[Dor11] Dortmund, Informatik C.: *ICD-C Compiler Framework Developer Manual.* `http://www.icd.de/`, 2011. – [Online; accessed 17.05.2011]

[dsl06] dsltarife.net: *In Deutschland mehr als 82,8 Mio Handy-Anschlüsse.* `http://www.dsltarife.net/news/1716.html`, 2006. – [Online; accessed 5.06.2011]

[Fer97] Ferdinand, C.: *Cache Behavior Prediction for Real-Time Systems*, Saarland University, Germany, Diss., 1997

[FK09] Falk, H. ; Kleinsorge, J. C.: Optimal Static WCET-aware Scratchpad Allocation of Program Code. In: *In Proceedings of Design Automation Conference (DAC).* San Francisco, USA, 2009

[FLT06] Falk, H. ; Lokuciejewski, P. ; Theiling, H.: Design of a WCET-Aware C Compiler. In: *In Proceedings of Workshop on Worst Case Execution Time Analysis (WCET).* Dresden, Germany, 2006

[GBEL10] Gustafsson, J. ; Betts, A. ; Ermedahl, A. ; Lisper, B.: The Mälardalen WCET Benchmarks: Past, Present And Future. In: *In Proceedings of*

Workshop on Worst Case Execution Time Analysis (WCET). Brussels, Belgium, 2010, S. 136–146

[Gmb11] GmbH, AbsInt Angewandte I.: *aiT Worst-Case Execution Time Analyzers.* `http://www.absint.com/ait/`, 2011. – [Online; accessed 16.05.2011]

[h2609] h264encoder.com: *h264 encoder.* `http://www.h264encoder.com/`, 2009. – [Online; accessed 3.07.2011]

[Inf08] Infineon: *Technical Referenz Manual Tricore TC1796.* April 2008. – Version 1.0

[LPF+11] Lokuciejewski, P. ; Plazar, S. ; Falk, H. ; Marwedel, P. ; Thiele, L.: Approximating Pareto optimal compiler optimization sequences—a trade-off between WCET, ACET and code size. In: *Software: Practice and Experience* (2011), Mai. – DOI 10.1002/spe.1079

[LPMS97] Lee, C. ; Potkonjak, M. ; Mangione-Smith, W. H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In: *In Proceedings of International Symposium on Microarchitecture (MICRO).* Washington, DC, USA, 1997

[Mar07] Marwedel, P.: *Eingebettete Systeme.* 2007

[ME04] Molnos, Heijligers-M. Cotofana S. D. A. ; Eijndhoven, J: Cache Partitioning Options for Compositional Multimedia Applications. In: *Proceedings of Circuits, Systems and Signal Processing (ProRISC)*, 2004

[Mue95] Mueller, F.: Compiler Support for Software-Based Cache Partitioning. In: *In Proceeding of Special Interest Group on Programming Languages (SIGPLAN) .* Austin, USA, 1995

[PLM09] Plazar, S. ; Lokuciejewski, P. ; Marwedel, P.: WCET-Aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In: *In Proceedings of Workshop on Worst Case Execution Time Analysis (WCET).* Dublin, Ireland, 2009

[PLM10] Plazar, S. ; Lokuciejewski, P. ; Marwedel, P.: WCET-driven Cache-aware Memory Content Selection . In: *In Proceedings of the Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC).* Carmona, Spain, 2010

[SM08] Suhendra, V. ; Mitra, T.: Exploring Locking and Partitioning for Predictable Shared Caches on Multi-Cores. In: *In Proceedings of Design Automation Conference (DAC)*. Anaheim, USA, 2008

[Tea11] Team, GCC D.: *GNU Compiler Collection*. `http://gcc.gnu.org/`, 2011. – [Online; accessed 16.05.2011]

[The02] Theiling, H.: ILP-based Interprocedural Path Analysis. In: *Proceedings of the Workshop on Embedded Software (ES)*. Grenoble, France, 2002

[The04] Thesing, S.: *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*, Saarland University, Germany, Diss., 2004

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

[VS05] V. Suhendra, A. Roychoudhury et a. T. Mitra M. T. Mitra: WCET Centric Data Allocation to Scratchpad Memory. In: *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. Miami, USA, 2005

[Wei91] Weiser, M.: The computer for the 21st century. In: *Scientific American* 265 (1991), Nr. 3, S. 66–75

[WM04] Wehmeyer, L. ; Marwedel, P.: Influence of Onchip Scratchpad Memories on WCET Prediction. In: *Proceedings of Workshop on Worst Case Execution Time Analysis (WCET)*. Catania, Italy, 2004

[WM05] Wehmeyer, L. ; Marwedel, P.: Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. In: *Proceedings of Design, Automation and Test in Europe (DATE)* . Munich, Germany, 2005

Abbildungsverzeichnis

Tabellenverzeichnis