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

technische universität
dortmund

Hendrik Borghorst

WCET-bewusste Scratchpad-Speicherallokation von Code und Daten für Multi-Task Systeme

Bachelorarbeit

26. September 2011

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik 12
Fakultät für Informatik
Technische Universität Dortmund
Gutachter:
Prof. Dr. Peter Marwedel
Dipl.-Inf. Sascha Plazar

GERMANY · D-44221 DORTMUND

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Ziele der Arbeit 4

1.2 Verwandte Arbeiten 5

1.3 Aufbau der Arbeit 6

2 Grundlagen **7**

2.1 WCC-Übersetzer Infrastruktur 7

2.1.1 Beschreibung der Zielarchitektur 8

2.1.2 Ablauf der Übersetzung von C-Code zu Maschinencode 9

2.1.3 Multi-Task Repräsentation 10

2.1.4 Bestimmung der Worst-Case Laufzeit 11

2.2 Ganzzahlige lineare Programmierung 13

3 Scratchpad-Speicherallokation **15**

3.1 Allokation von Daten in Scratchpad-Speicher 15

3.2 Allokation von Code in Scratchpad-Speicher 16

4 Multi-Task Scratchpad-Speicherallokation **19**

4.1 Gain-Computation 21

4.2 Optimierung der gesamten Taskmenge 24

4.2.1 ILP-Formulierungen 24

5 Realisierung **33**

5.1 Implementierung der Gain-Computation 33

5.2 Implementierung der Multi-Task Speicherallokation 37

6 Evaluation **39**

6.1 Benchmark-Umgebung 39

6.2 Benchmark-Taskmengen 39

6.3 Durchführung der Benchmarks 42
6.4 Benchmark Ergebnisse 43
6.4.1 Separate Bewertung von DSPM und PSPM 44

7 Fazit und Ausblick **53**
7.1 Ausblick 53

1 Einleitung

In der heutigen Zeit ist die Informationsverarbeitung immer wichtiger geworden. Dabei haben eingebettete Systeme eine besonders wichtige Rolle, da diese in vielen Geräten zum Einsatz kommen, ohne dass sie dort bemerkt werden.

Es ist davon auszugehen, dass die Bedeutung der eingebetteten Systemen in Zukunft enorm zunehmen wird. So wird zum Beispiel mit der Einführung des neuen IPv6-Standards der Grundstein für eine stark zunehmende Vernetzung gelegt, wozu eine erhebliche Anzahl eingebetteter Systeme notwendig wird.

Von vielen eingebetteten Systemen wird erwartet, dass sie einige Grundbedingungen erfüllen [Mar07]. So ist es für viele der Systeme von großer Bedeutung, dass sie gewisse Aufgaben immer in festgelegten Zeitschranken erfüllen. Man stelle sich zum Beispiel die Steuereinheit eines Flugzeuges vor, die in einer kritischen Situation zu lange zum Rechnen benötigt. Dies könnte katastrophale Folgen haben. Dieser Zeitrahmen wird Worst-Case-Laufzeit (englisch: worst case execution time=WCET) genannt. Dies ist die maximale Laufzeit die ein Programm erreichen kann.

Ebenso wird von diesen Systemen erwartet, dass sie nicht ausfallen und so eine hohe Verfügbarkeit garantieren. Kosten- und Energieeffizenz sind für diese Systeme ebenso wichtig, da diese Systeme ihre Energie oft aus Akkus beziehen, aber auch, da durch einen zu hohen Energieverbrauch unnötige Wärme entsteht, die zu Problemen führen kann. Außerdem erwarten Kunden einen möglichst niedrigen Preis.

Möchte man nun diese Bedingungen in die Praxis umsetzen, so bietet es sich nicht an einen Übersetzer für normale Desktop- oder Serveranwendungen, wie zum Beispiel den GCC (Gnu Compiler Collection), zu verwenden, da diese Übersetzer diese Gegebenheiten in der Regel während der Optimierung nicht beachten. Dies geschieht durch das Betrachten der Durchschnitts-Laufzeit (englisch:average case execution time=ACET). [FL10]

Einen anderen Ansatz verfolgt die WCC-Übersetzer Infrastruktur [WCC11]. Dieser Übersetzer stellt eine Neuheit dar, da dies der erste Übersetzer ist, der bei den Optimierungen explizit auf die Worst-Case Laufzeit Rücksicht nimmt und versucht diese statt der ACET zu verbessern.

Eine mögliche Optimierung ist die Nutzung eines schnellen Speichers (sog. Scratchpad-Speicher). Durch die Nutzung dieses Speichers ist es möglich, die Zyklen pro Speicherzugriff zu minimieren, was einen Einfluss auf die WCET hat [Kle08].

Für diese Art der Optimierung existieren bereits mehrere Arbeiten, so beschreibt F. Rotthowe [Rot08] die Auslagerung von Daten in Scratchpad-Speicher (SPM) zur Minimierung der WCET. Dagegen beschreibt J. Kleinsorge [Kle08], wie sich die WCET minimieren lässt, indem Programmcode in den SPM ausgelagert wird. Diese Arbeiten betrachten jedoch nur ein einziges Programm. Da es jedoch immer wichtiger wird, dass Recheneinheiten nicht mehr auf die Ausführung eines einzigen Programmes begrenzt sind, entsteht die Motivation diese Optimierung auch für mehrere Programme auf einer CPU durchzuführen. Diesen Ansatz hat auch K. Risto[Ris11] verfolgt, der später noch weiter erläutert wird.

1.1 Ziele der Arbeit

Das Ziel dieser Arbeit ist es, die gegebenen Ansätze zur Minimierung der WCET durch Nutzung des Scratchpad-Speichers so zu erweitern, dass diese für Multi-Task Systeme funktionieren, dazu wird als Basis dieser Arbeit die Arbeit von K. Risto verwendet. Die Optimierung soll über eine Formulierung der ganzzahlig linearen Programmierung (ILP) erfolgen.

Diese Optimierung soll sowohl für Programmdaten als auch für Programmcode erfolgen. Realisiert wird die Optimierung innerhalb der WCC-Infrastruktur, da diese bereits das WCET-Analyse Werkzeug aiT [AIT11] integriert. Außerdem wurden innerhalb des WCC-Projektes auch schon die SPM-Optimierungen für einzelne Anwendungen realisiert, auf denen diese Arbeit basiert.

Die Optimierung soll speziell für den Infineon TriCore Prozessor, ein von der Automobilindustrie stark eingesetzter Mikrocontroller, realisiert werden. Desweiteren wird noch evaluiert werden, wie sich die SPM-Allokation auf die WCET auswirkt.

1.2 Verwandte Arbeiten

Diese Bachelorarbeit basiert zum Teil auf der Masterarbeit von Risto[Ris11], welche sich ebenfalls mit der Multi-Task Scratchpad-Speicherallokation beschäftigt. Allerdings wurden in dieser Arbeit Annahmen bei der Speicherallokation gemacht, welche zu einer nicht optimalen Worst-Case Laufzeit führen können.

In [Kle08] und [Rot08] wird von J. Kleinsorge bzw. F. Rotthowe eine Scratchpad-Speicherallokation für Programminstruktionen und Programmdaten vorgestellt, welche für einzelne Tasks die Worst-Case Laufzeit optimieren. Dabei werden zwei Strategien zur Allokation vorgestellt, zum einen die dynamische Allokation, bei der sich während der Ausführung die Speicherbelegung ändert, das heißt es liegen nicht dauerhaft die gleichen Variablen im Scratchpad-Speicher und zum anderen die statische Speicherallokation, bei der durchgehend die gleichen Variabeln im Scratchpad-Speicher liegen.

H. Kotthaus beschreibt in [Kot11], wie sich die Worst-Case Laufzeit durch Optimierungen des Caching-Verhaltens des TriCore Prozessors senken lässt. Es wird versucht die Anzahl der Cache-Misses zu reduzieren. Cache-Misses sind Zugriffe auf ein Datum, welches nicht im Cache liegt und so aus dem Hauptspeicher gelesen werden muss.

Eine andere Optimierungsstrategie verfolgt N. Schmitz in [Sch10], dort wird die Worst-Case Laufzeit reduziert, indem die Registerallokation durch ILP-Formulierungen (siehe Abschnitt 2.2) optimiert wird. Die Registerallokation verteilt die Variablen des Programmes auf die verfügbaren Register, dabei führt eine optimale Belegung zu einer niedrigeren Ausführungszeit.

1.3 Aufbau der Arbeit

Die Arbeit ist wie folgt strukturiert.

- **Kapitel 2** beschäftigt sich mit den theoretischen Grundlagen zur WCET-Analyse und der Optimierung durch Scratchpad-Speicher, sowie mit der Vorstellung der verwendeten Werkzeuge.
- **Kapitel 3** beschreibt die Optimierung durch Scratchpad-Speicher für einzelne Anwendungen.
- **Kapitel 4** stellt die Multi-Task Optimierung vor.
- **Kapitel 5** beschreibt, wie die Optimierung innerhalb des WCC-Frameworkes umgesetzt wurde und welche Änderungen notwendig waren.
- **Kapitel 6** befasst sich mit der Evaluation der gemachten Änderungen an der Multi-Task Allokation.
- **Kapitel 7** gibt einen Ausblick auf eventuelle Verbesserungen und eine abschließende Bewertung der Arbeit.

2 Grundlagen

Wie zuvor beschrieben, wird zur Realisierung der Optimierung dieser Arbeit die WCC-Infrastruktur verwendet. Im folgendem Kapitel werden daher die Arbeitsweise und die Einzelkomponenten des WCC-Übersetzers vorgestellt. Da der WCC-Übersetzer für viele Optimierungsprobleme, wie auch für die Multi-Task Optimierung, auf die ganzzahlige lineare Programmierung setzt, werden im Zuge dieses Kapitels noch die Grundlagen dafür erklärt.

2.1 WCC-Übersetzer Infrastruktur

Übersetzeroptimierungen sind schon seit langer Zeit ein fester Bestandteil der Forschung innerhalb der Informatik. Jedoch beschäftigen sich die Arbeiten auf diesem Gebiet meistens nur mit Optimierungen, die einen Einfluss auf die Durchschnitts-Laufzeit haben. Für eingebettete Systeme ist diese aber nicht von besonderer Relevanz. Für Echtzeit-Systeme mit „harten“ Echtzeitbedingungen, muss die Worst-Case Laufzeit bestimmt werden. Dies ist die maximale Laufzeit, die eine Anwendung jemals benötigt. Die Best-Case Laufzeit (BCET: best-case execution time) gibt an, wie schnell eine Anwendung im besten Fall läuft. Dieser Fall wird in der Regel jedoch nur selten erreicht und hat daher für eingebette Systeme keine Relevanz. Abbildung 2.1 zeigt, nach [Mar07], die verschiedenen Laufzeittypen im Vergleich. Wird die Best-Case Laufzeit bestimmt, so wird in der Regel eine niedrigere Laufzeit bestimmt, als die, die wirklich erreicht werden kann. Ebenso wird bei der Bestimmung der Worst-Case Laufzeit eine Überabschätzung vorgenommen, was zu einer höheren bestimmten Worst-Case Laufzeit als die reale führt. Die Durchschnittslaufzeit befindet sich zwischen der realen Best-Case Laufzeit und der realen Worst-Case Laufzeit.

Nach [FL10] ist es bis jetzt gängige Praxis, bei der Entwicklung von eingebetten Systemen grafische Entwicklungswerkzeuge zu verwenden. Diese setzen zum Beispiel einen endlichen Automaten in Maschinencode um. Ein Beispiel dafür stellt „Statemate“

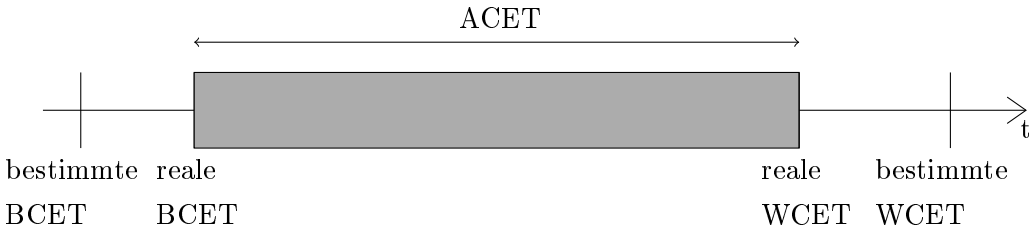


Abbildung 2.1: WCET im Vergleich zur ACET und BCET

dar [STA11]. Diese Werkzeuge wandeln in der Regel die grafische Darstellung zunächst in C-Programmcode um. Dieser wird in einem nächsten Schritt übersetzt, wobei keine weiteren Optimierungen, bezüglich der WCET, mehr stattfinden. Es können zwar normale Übersetzer-Optimierungen durchgeführt werden, die keinen direkten Einfluss auf die WCET haben, jedoch ist bei diesen die Auswirkungen auf die WCET ungewiss. Um von dieser Anwendung die WCET zu bestimmen muss in einem weiterem Schritt der Maschinencode mit anderen Werkzeugen analysiert werden. Dies ist ein sehr aufwändiger Prozess, da es nur sehr schlecht möglich ist, die WCET zu verbessern, normalerweise nur durch Ausprobieren von Änderungen an der grafischen Darstellung.

Der WCC-Übersetzer geht einen völlig neuen Weg. Er ist der erste Übersetzer, der eine nahtlose Integration von WCET-Analysewerkzeugen bietet und diese zur Optimierung einsetzt. So werden ILP-Formulierungen erstellt, mit Hilfe derer es möglich ist, den Einfluss einer Optimierung auf die WCET direkt zu bestimmen.

2.1.1 Beschreibung der Zielarchitektur

Der WCC-Übersetzer erzeugt Programmcode für die Infineon TriCore Prozessoren TC1796 bzw. TC1797. Jedoch besitzt der TC1797 keinen Daten-Scratchpad-Speicher [Inf09], so dass im Umfeld dieser Arbeit nur der Prozessor TC1796 betrachtet wird. Die Umsetzung der Optimierung für den Programm-Scratchpad-Speicher ist aber auf beiden Modellen anwendbar.

Bei den Infineon TriCore Prozessoren handelt es sich um Mikrocontroller die sehr stark von der Automobilindustrie eingesetzt werden. Der Name leitet sich aus den Fähigkeiten des Prozessors ab.

So vereint der TriCore Prozessor drei verschiedene Befehlssatztypen. Zum einen unterstützt der Befehlssatz die typischen Befehle eines DSP-Prozessors (Digitaler-Signal-

Prozessor, englisch: digital signal processing), zum anderen einen RISC Befehlssatz (reduced instruction set computer, engl. für Rechner mit reduziertem Befehlssatz), das heißt einen Befehlssatz, welcher nur wenige Befehle umfasst, so dass generell zur optimalen Nutzung bessere Übersetzer nötig sind. Ebenso enthält der TriCore Prozessor noch einen Mikrocontroller, mit dem es möglich ist verbundene Peripheriegeräte, wie Sensoren oder Aktoren, zu kontrollieren.

Innerhalb des TriCore Prozessors findet eine strikte Trennung zwischen Instruktionen und Programmdaten statt. Daher gibt es auch separate Scratchpad-Speicher für Code und Daten. Der TriCore Prozessor nutzt für den Zugriff auf den Speicher zwei separate Busse für Daten und Programminstruktionen. Ein Zugriff auf diese Busse erfolgt über ein **PMI** (Program Memory Interface) und ein **DMI** (Data Memory Interface). Diese beiden Interfaces sind direkt mit der eigentlichen Recheneinheit verbunden, so dass eine schnelle Kommunikation möglich ist.

Innerhalb des PMI befindet sich, bei dem TC1796 Prozessor, der Programm-Scratchpad-Speicher (PSPM) mit einer Kapazität von 48KB. Der Daten-Scratchpad-Speicher (DSPM) ist Teil des DMI und besitzt eine Kapazität von 64KB. Da ein Zugriff auf den Scratchpad-Speicher nicht über den Datenbus erfolgt, gibt es für die Recheneinheit keine Verzögerung, da sie direkt auf die Daten zugreifen kann. Im Falle eines Zugriffes auf den Hauptspeicher, müsste der Bus verwendet werden, wodurch Wartezyklen entstehen würden.

2.1.2 Ablauf der Übersetzung von C-Code zu Maschinencode

Zu Beginn wird ANSI-C99 Code durch einen Parser in das Datenformat **ICD-C** umgewandelt. Ebenfalls benötigt der Parser *Flow-Facts*, welche angeben wie oft eine Schleife zum Beispiel maximal ausgeführt wird oder wie tief eine Rekursion maximal ist. Die Flow-Facts werden per `_Pragma`-Operator mit übergeben. ICD-C ist eine Datenstruktur, die eine maschinenunabhängige Zwischendarstellung (intermediate representation) von C-Code darstellt. Auf dieser Ebene können schon maschinenunabhängige Optimierungen am Programmcode vorgenommen werden.

Durch einen *Code-Selector* findet im folgenden Schritt eine Übersetzung des ICD-C Codes in das **ICD-LLIR** Format statt. Dies ist eine Darstellungsart des Assemblercodes, welche später in Assemblercode für die Zielarchitektur umgewandelt wird. Innerhalb des ICD-LLIR Formates finden einige Optimierungsschritte für die Verbesserung der

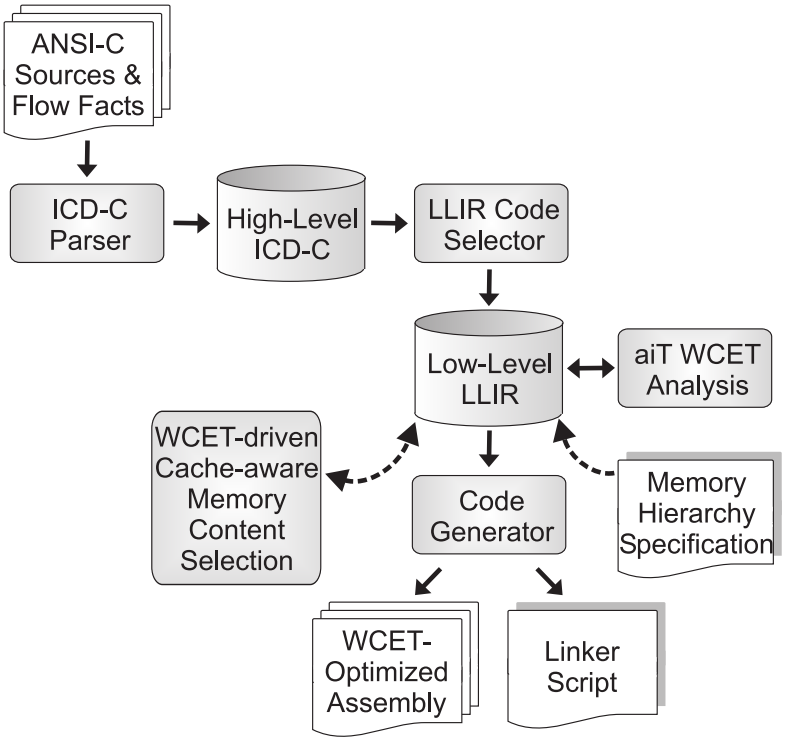


Abbildung 2.2: Aufbau des WCC-Übersetzers nach [FL10]

WCET statt, so auch die Scratchpad-Speicherallokation. Ebenso lässt sich das ICD-LLIR Format in das Format CRL umwandeln, welches durch das WCET-Analysewerkzeug „aiT“ vorausgesetzt wird. So ist es einfach möglich, innerhalb eines Optimierungsschrittes die WCET eines Programmes zu bestimmen.

Abschließend wird durch einen *Code-Generator*, der für die jeweilige Zielarchitektur entwickelt worden sein muss, der fertig optimierte Assemblercode erstellt. Der Aufbau des Übersetzers wird in Abbildung 2.2 dargestellt.

2.1.3 Multi-Task Repräsentation

Da es für Übersetzer nicht gewöhnlich ist mehrere Tasks gemeinsam zu optimieren, muss zunächst eine geeignete Repräsentation der Task-Menge definiert werden. Der WCC-Übersetzer verwendet für diesen Zweck eine eigene Repräsentation, die sich Taskconfig (deutsch: Taskkonfiguration) nennt. Eine Taskconfig beschreibt die verschiedenen Tasks die optimiert werden sollen und gibt für jeden Task die benötigten C-Dateien an. Zusätzlich werden noch Informationen zum Schedulingverfahren mitgegeben.

Die Tasks werden in einem ersten Schritt separat in das ICD-LLIR Format übersetzt, mit dem es anschließend möglich ist eine Optimierung mit gegenseitiger Rücksicht durchzuführen, wie z.B. die Multi-Task Scratchpad-Speicherallokation. Als Ausgabe liefert der WCC-Übersetzer für jeden Task eine separate Binärdatei, welche den Maschinencode enthält.

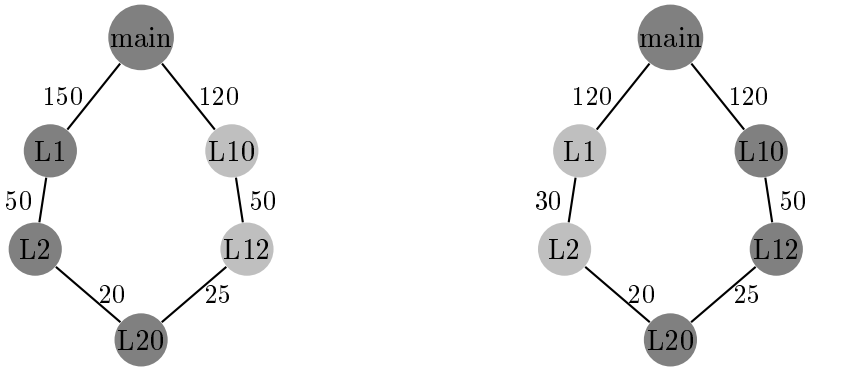
2.1.4 Bestimmung der Worst-Case Laufzeit

Um die Worst-Case Laufzeit zu bestimmen, gibt es prinzipiell zwei verschiedene Möglichkeiten. Zum einen die dynamische Analyse und zum anderen die statische Analyse. Im folgenden wird auf die Vor- und Nachteile beider Methoden eingegangen.

Die dynamische Worst-Case Laufzeit Analyse versucht die WCET zu bestimmen, indem verschiedene Eingabemöglichkeiten für ein Programm simuliert werden. Anhand dieser Eingaben wird versucht eine möglichst maximale Laufzeit zu ermitteln. Dieses Verfahren hat jedoch den Nachteil, dass nicht garantiert werden kann, dass eine Eingabe wirklich die maximale Laufzeit verursacht. Es kann immer möglich sein, dass es eine Eingabekombination gibt, welche eine noch höhere Laufzeit verursacht. Aus diesem Grund wird zwar eine gewisse Überabschätzung durchgeführt, jedoch sind die Angaben nicht verlässlich. Für Systeme, welche eine harte Echtzeit-Bedingung einhalten müssen, kann dieses Analyseverfahren daher auf keinen Fall verwendet werden, da dies mit einem zu großem Risiko verbunden wäre.

Das statische Worst-Case Laufzeit Analyseverfahren verfolgt einen anderen Ansatz. Dieses Verfahren analysiert den Maschinencode und beachtet dabei auch Faktoren wie Cache-Misses oder Speicherzugriffszeiten. Damit das statische Verfahren jedoch funktionieren kann, benötigt es weitere Informationen. Dies sind die „Flow-Facts“, welche Informationen enthalten, die einen direkten Einfluss auf die WCET eines Programmes haben. Dazu zählt die Anzahl der Schleifendurchläufe und die maximale Rekursionstiefe eines rekursiven Funktionsaufrufes. Außerdem müssen dem Analysewerkzeug Informationen über Speicherzugriffszeiten, Cache-Verhalten und der Pipeline der Zielarchitektur vorliegen, damit eine verlässliche Analyse stattfinden kann.

Zur Bestimmung der Worst-Case Laufzeit wird ein Kontrollflussgraph erstellt, der die verschiedenen Möglichkeiten des Programmflusses modelliert. Dabei handelt es sich um einen gerichteten Graphen, der auch Zyklen enthalten kann. Die Knoten des Graphen stellen dabei die Basisblöcke des Programmes dar. Eine gerichtete Kante zwischen zwei



vor WCET Optimierung nach WCET Optimierung

Abbildung 2.3: Problematik bei der Optimierung des WCEPs

Knoten steht für eine Ausführung der zwei Basisblöcke, die sie verbindet. Innerhalb dieses Graphen gibt es einen Pfad, der Worst-Case-Execution-Path (WCEP), der eine Ausführung des Programmes mit maximaler Ausführungsdauer beschreibt. Es gilt diesen Pfad zu finden und zu optimieren, dabei ist es wichtig, immer den gesamten Kontrollflussgraphen (Control flow graph=CFG) zu betrachten, da es vorkommen kann, dass nach einer durchgeführten Optimierung der Worst-Case-Execution-Path, nicht mehr der Pfad mit maximaler Laufzeit ist. Abbildung 2.3 zeigt wie sich die Pfade nach einer durchgeführten Optimierung beispielsweise ändern könnten. Vor der Optimierung ist der Pfad $main \rightarrow L1 \rightarrow L2 \rightarrow L20$ der WCEP, nach der Optimierung ist es $main \rightarrow L10 \rightarrow L12 \rightarrow L20$, da die Summe der Ausführungszeiten des rechten Pfades nun höher ist. Eine weitere Optimierung des alten Pfades würde zu keiner weiteren Verbesserung der Worst-Case Laufzeit führen, sondern nur die Average-Case Laufzeit eventuell beeinflussen. Um die WCET weiter zu optimieren muss der neue WCEP gefunden werden und optimiert werden.

Mit diesen Informationen ist es möglich eine Annäherung an die tatsächliche WCET zu bestimmen. Die bestimmte WCET liegt meistens über der realen WCET aber nie unter ihr, da eine Überabschätzung stattfindet. Aufgrund der hohen Zuverlässigkeit der statischen Worst-Case Laufzeitanalyse ist diese Methode das favorisierte Verfahren bei dem Entwurf von eingebetteten Systemen.

Der WCC-Übersetzer nutzt zur Bestimmung der Worst-Case Laufzeit das Analysewerkzeug „aiT“ der Firma AbsInt [AIT11].

2.2 Ganzzahlige lineare Programmierung

Die ganzzahlige lineare Programmierung ist ein Rechenmodell, welches häufig zur Optimierung eingesetzt wird. Eine ILP-Formulierung besteht aus Entscheidungsvariablen, welche nur ganzzahlige Werte annehmen dürfen, sowie einer Zielfunktion und Nebenbedingungen, welche bei der Lösung der Zielfunktion eingehalten werden müssen. Für die Zielfunktion kann zum Beispiel ein Maximum oder Minimum gesucht werden.

Für diese Arbeit wird zusätzlich die Einschränkung eingeführt, dass alle Entscheidungsvariablen binär sind:

$$x_i \geq 0 \text{ und } x_i \leq 1 \tag{2.1}$$

Eine Zielfunktion hat zum Beispiel folgende Form, wobei x_i die Entscheidungsvariablen sind und s_i eine Variable, wie zum Beispiel bestimmte Kosten.

$$\sum_{i \in I} x_i * s_i \tag{2.2}$$

Das Ziel ist es jetzt einen optimalen Wert zu finden für diese Funktion. Da in der Praxis häufig mehrere tausend Entscheidungsvariablen existieren, ist es nicht mehr möglich dies per Hand zu lösen. Innerhalb des WCC-Übersetzers ist es möglich das freie Lösungswerkzeug „lp-solve“ [LPS11] oder das kommerziell erhältliche Werkzeug „CPLEX“ der Firma IBM [CPL11] zu verwenden.

3 Scratchpad-Speicherallokation

In diesem Kapitel wird eine Optimierungsstrategie vorgestellt, welche gezielt die Worst-Case Laufzeit senkt. Dazu wird zuerst auf bestehende Arbeiten auf diesem Gebiet eingegangen, welche als Grundlage für diese Optimierung dienen. Zum einen beschreibt die Diplomarbeit von J. Kleinsorge [Kle08], wie sich durch Verschieben von Programmcode in den Scratchpad-Speicher die WCET senken lässt. Außerdem beschreibt F. Rotthowe [Rot08], in seiner Arbeit, die Scratchpad-Speicherallokation für Programmdaten. Auf diese Arbeiten wird im folgenden Abschnitt detailliert eingegangen.

Es gibt grundsätzlich zwei verschiedene Arten der Scratchpad-Speicherallokation. Zum einen gibt es die statische Allokation, bei der während der gesamten Ausführung die gleichen Datenblöcke bzw. Basisblöcke der Anwendung im Scratchpad-Speicher liegen. Dieses Verhalten hat den Nachteil, dass es vorkommen kann, dass zu einem Zeitpunkt während des Ausführung nicht mehr benötigter Code oder Daten im Scratchpad-Speicher liegen.

Zum anderen gibt es die dynamische Scratchpad-Speicherallokation. Dieses Verfahren kopiert zur Laufzeit des Programmes Code und Daten zwischen den verschiedenen Speichern. So ist es möglich, eventuell nicht weiter benötigte Basis- beziehungsweise Datenblöcke gegen welche auszutauschen, die noch benötigt werden. Die Informationen welche Blöcke wann eingelagert werden sollen, werden jedoch nicht zur Laufzeit bestimmt sondern sind, für diese Speicherallokation, vollständig während der Übersetzung bestimmt. Für das Kopieren zwischen den Speichern muss während des Übersetzungsvorganges zusätzlicher Programmcode erstellt werden, der diese Aufgabe zur Laufzeit übernimmt.

3.1 Allokation von Daten in Scratchpad-Speicher

Um Programmdaten möglichst effizient im Bezug auf die Worst-Case Laufzeit auf den Scratchpad-Speicher und den regulären Hauptspeicher aufzuteilen, stellt Rotthowe eine

Lösung vor, welche eine ILP-Formulierung verwendet. Bevor man diese Formulierung aufstellen kann, muss jedoch bekannt sein, durch welche Programmteile auf welchen Datenblock wie häufig zugegriffen wird, da selbstverständlich nicht jeder Programmteil auf alle Daten zugreift. Ist dies bekannt, so kann durch Laufzeitanalyse der Gewinn bezüglich der Worst-Case Laufzeit pro Datenblock, bestimmt werden. Dies wäre aber mit einer erheblichen Anzahl an Analyseschritten verbunden, so dass dies nicht praktikabel ist. Stattdessen werden die Kenntnisse der Zielarchitektur ausgenutzt und so ein Gewinn aus der Art und Anzahl der Zugriffe auf einen Datenblock berechnet.

Mit Hilfe dieser Werte, kann der Kontrollflussgraph für den Task aufgebaut, alle Pfade bestimmt und so der längste Pfad, der WCEP, gesucht werden. Aus diesen Informationen wird ein ILP-Modell erstellt, bei dem die Entscheidungsvariablen repräsentieren, ob ein Datenblock im Hauptspeicher oder im Scratchpad-Speicher liegt. Die Zielfunktion des ILPs stellt die modellierte WCET eines Tasks dar. So wird jedem Datenblock eine Entscheidungsvariable zugewiesen, die mit den Kosten, in Zyklen, des Datenblockes im Hauptspeicher multipliziert wird. Das Inverse der Entscheidungsvariablen wird mit den Kosten des Datenblockes im Scratchpad-Speicher multipliziert. So wird durch den Lösungsvorgang des ILPs eine optimale Belegung des Scratchpad-Speichers mit minimaler WCET bestimmt.

3.2 Allokation von Code in Scratchpad-Speicher

Die Scratchpad-Speicherallokation für Programmcode läuft ähnlich wie die für Daten. Jedoch müssen bei der Modellierung weitere Eigenschaften beachtet werden.

Die Allokation von Code erfolgt nicht, wie bei der DSPM-Allokation, über Datenblöcke, sondern über Basisblöcke des Programmes. Ein Basisblock ist eine Sequenz von Instruktionen, welche immer an der gleichen Stelle betreten und verlassen wird. Jedoch kann es vorkommen, dass ein Basisblock verzweigt und zwei mögliche nachfolgende Basisblöcke besitzt, was zum Beispiel der Fall bei einem bedingtem Sprung ist.

Wird ein Basisblock in den Scratchpad-Speicher verschoben und einer seiner Nachfolger nicht, so ist es notwendig den Kontrollflussgraphen an die geänderte Situation anzupassen, da ein Sprung in einen anderen Speicherbereich notwendig ist.

Diese Eigenschaft muss unbedingt bei der Modellierung beachtet werden, da ein Sprung in einen anderen Speicherbereich zusätzlichen Programmcode erfordert, wodurch die

Größe eines Basisblockes abhängig von seiner Position im Speicher nach der Optimierung noch anwachsen kann. Auf dieses Verhalten wird in Kapitel 5 detaillierter eingegangen.

Zur Modellierung dieser Eigenschaft werden zusätzlich zu den Kosten der Basisblöcke im ILP-Modell weitere Kosten eingeplant, welche von den Nachfolgern des Basisblockes und deren Position abhängig sind.

Im folgendem Abschnitt wird eine Erweiterung der oben genannten Speicherallokationen für Multi-Task Systeme vorgestellt. Dazu wurde als Grundlage die statische Speicherallokation verwendet.

4 Multi-Task Scratchpad-Speicherallokation

Diese Arbeit beruht auf der Arbeit von Kyrill Risto [Ris11], welche sich ebenfalls mit der Optimierung der Speicherallokation für eine Taskmenge beschäftigt. Jedoch hat sich herausgestellt, dass diese Optimierung nicht fehlerfrei funktioniert. In diesem Kapitel wird die Multi-Task Allokation beschrieben, und auf die Fehler eingegangen, welche eine optimale SPM Nutzung verhindern.

Das Kernproblem der Multi-Task Optimierung ist es, jedem Task die optimale Menge an Scratchpad-Speicher zur Verfügung zu stellen, so dass die Worst-Case Laufzeit maximal gesenkt wird. Zur Lösung dieses Problemes wird ein zwei-schrittiger Lösungsansatz verwendet. Der erste Schritt bestimmt für jeden Task der Taskmenge die Reduzierungen der WCET, welche durch unterschiedliche Belegung des Scratchpad-Speichers erzielt werden können. Dazu muss bestimmt werden, wie groß die Reduktion der Worst-Case Laufzeit ist, wenn einem Task t ein gewisser Anteil statischer und ein gewisser Anteil dynamischer Speicher zugewiesen wird. $S^{DSPM}_{statisch}$ und $S^{PSPM}_{statisch}$ geben die Größe des statischen SPM-Bereiches der gesamten Taskmenge an, sowie $S^{DSPM}_{dynamisch}$ und $S^{PSPM}_{dynamisch}$ für den dynamischen Bereich.

In einem zweitem Schritt werden diese Informationen verwendet, um eine optimale Kombination von statischen und dynamischen Anteilen aller Tasks am gesamten D/PSPM für eine optimale WCET Reduktion zu bestimmen.

Dazu werden in [Ris11] drei verschiedene Belegungsstrategien vorgestellt.

- **statische Allokation:** Bei der statischen Einteilung wird jedem Task eine feste Menge an Speicher zur Verfügung gestellt. Der Inhalt des Speichers bleibt während der gesamten Ausführung fest. Das heißt, dass sich alle Tasks den Speicher teilen und so oft nur eine geringe Kapazität erhalten können. Der Vorteil dieser Strategie liegt darin, dass bei einem Kontextwechsel keine Kopierkosten notwendig sind. Die Größe des statischen Bereiches wird durch $S^{DSPM}_{statisch}$ und $S^{PSPM}_{statisch}$ beschrieben,

welche auch in Abbildung 4.1 dargestellt ist, jedoch würde der statische Bereich dort den kompletten Speicher ausfüllen.

- **dynamische Allokation:** Die dynamische Allokation stellt jedem Task den gesamten Scratchpad-Speicher zur Verfügung. Ein Vorteil dieser Strategie ist, dass jedem Task eine größere Menge an Speicher zur Verfügung steht als bei der statischen Einteilung. Jedoch entstehen hierbei zusätzliche Kopierkosten während eines Kontextwechsels, da der Inhalt des SPM durch den Inhalt des neuen Tasks ersetzt werden muss. Während dieses Vorganges kann das System nicht mit der Programmausführung fortfahren, wodurch Rechenzyklen verschwendet werden. $S^{DSPM}_{dynamisch}$ und $S^{PSPM}_{dynamisch}$ geben die Größe des dynamischen Speicherbereiches jeweils für Daten- und Code-SPM an.

- **hybride Allokation:** Eine Kombination der beiden vorherigen Verfahren stellt die hybride Allokation dar. Der Speicher wird in einen statischen und in einen dynamischen Bereich eingeteilt. Innerhalb des statischen Bereiches bekommt jeder Task eine feste Menge Speicher zugewiesen, dieser kann auch eine Größe von 0 Bytes haben. Zusätzlich bekommt jeder Task einen zweiten Speicherbereich, der exklusiv für den zur Zeit laufenden Task bereit steht, dies ist der dynamische Bereich. Dadurch entstehen wie bei dem rein dynamischen Verfahren ebenfalls Kopierkosten, welche aber geringer sein können, als die des voll dynamischen Verfahrens. Für die Einteilung des SPM wird dieser in zwei Sektionen aufgeteilt. Die erste Sektion ist der statische Bereich am Anfang des SPM. Innerhalb dieses Bereiches liegen die statischen Daten der jeweiligen Tasks. Die zweite Sektion liegt am Ende des SPM und beinhaltet den dynamischen Speicherbereich des gerade aktiven Tasks. Abbildung 4.1 zeigt wie der Scratchpad-Speicher beispielhaft in zwei Sektionen geteilt wird, jedoch wäre bei optimaler Belegung kein Freiraum zwischen den Bereichen.

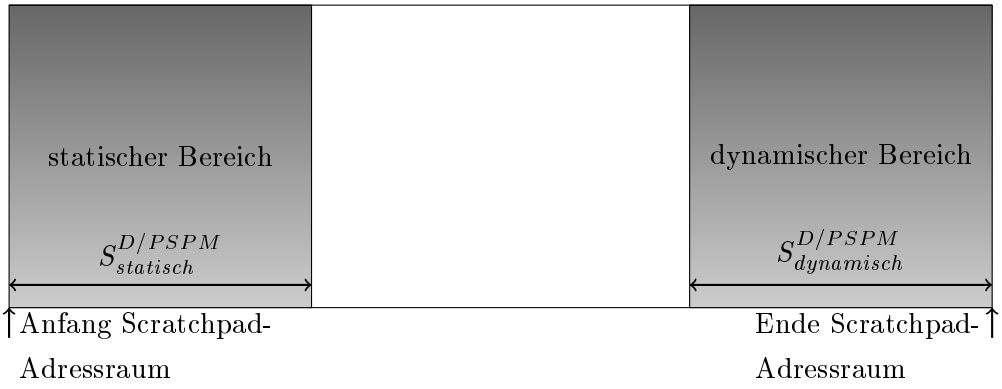


Abbildung 4.1: Hybride Aufteilung des Scratchpad-Speichers

Der folgende Abschnitt beschäftigt sich mit der Gain-Computation, durch diesen Schritt wird für alle Tasks für jede mögliche Partitionsgröße des P/DSPM die WCET berechnet. Danach wird beschrieben, wie diese Informationen genutzt werden, und für die gesamte Taskmenge eine optimale Kombination der Partitionsgrößen bestimmt wird.
Beide Schritte stellen zum Lösen des Optimierungproblems ILPs auf.

Dabei ist eine strikte Differenzierung zwischen den ILPs der Gain-Computation (Schritt 1) und der Multi-Task Scratchpad-Speicherallokation (Schritt 2) erforderlich. Die **Gain-Computation** erstellt ILP-Formulierungen, damit eine optimale Belegung des DSPM und PSPM sowohl für den statischen als auch dynamischen Bereich gefunden wird. Während der Gain-Computation wird immer nur ein Task betrachtet. Die **Multi-Task SPM-Allokation** erstellt die ILP-Formulierungen um eine optimale Belegung für die gesamte Taskmenge zu finden.

4.1 Gain-Computation

Damit die Multi-Task Optimierung die optimale Speicherbelegung für ein Multi-Task System finden kann, werden Informationen über die Worst-Case Laufzeiten der einzelnen Tasks benötigt.

Um diese Informationen zu bestimmen sind zahlreiche Laufzeitanalysen notwendig, die sehr rechenaufwändig sind. Damit die Laufzeit der Berechnungen nicht zu groß wird, wird die Worst-Case Laufzeit nur für Speichergrößen in einer bestimmten Schrittweite (z.B. 128 Bytes) berechnet.

Das hybride Allokationsverfahren setzt zwei Speicherbereiche voraus, weshalb alle möglichen Kombination, $d_i^{statisch}/d_i^{dynamisch}$ für den DSPM und $p_i^{statisch}/p_i^{dynamisch}$ für den PSPM, separat innerhalb der Schrittweite berechnet werden müssen. Dies geschieht separat für jeden Task aus der Taskmenge.

Für jede dieser Speicherkonfigurationen werden separat für DSPM und PSPM ein ILP aufgestellt, welches zweimal gelöst wird. Der erste Lösungsvorgang belegt optimal den statischen SPM-Bereich und der zweite Lösungsvorgang belegt den dynamischen Bereich für einen speziellen Task. Für die statische und die dynamische Multi-Task Scratchpad-Speicherallokation würde es ausreichen, das ILP während der *Gain-Computation* nur einmal zu lösen. Jedoch ist es für die hybride SPM-Allokation erforderlich das ILP zweimal pro Konfiguration zu lösen, wodurch hintereinander der statische und dynamische SPM-Bereich für einen speziellen Task belegt werden.

Daten-Scratchpad-Speicherallokation

Zu Beginn der Speicherallokation wird die ILP-Formulierung erstellt, welche notwendig ist, um eine optimale Verteilung der Datenblöcke zu finden, für einen Task t und einer bestimmten Partitionsgröße $d_i^{statisch}/d_i^{dynamisch}$.

Für die Formulierung des ILP-Problemes der Speicherallokation müssen zunächst die Variablen beschrieben werden. Die Entscheidungsvariable x_i gibt an, ob ein Datenblock db_i innerhalb des Scratchpad-Speichers oder im Hauptspeicher liegt:

$$x_i = \begin{cases} 1 & \text{Datenblock } db_i \text{ liegt in Daten-SPM} \\ 0 & \text{Datenblock } db_i \text{ liegt nicht in Daten-SPM} \end{cases} \tag{4.1}$$

Außerdem wird die Variable S_i eingeführt, welche die Größe des Datenblocks i angibt, sowie die Variable c_i, die modelliert, wie hoch die Kosten im Bezug auf die Worst-Case Laufzeit innerhalb des Scratchpad-Speichers sind. Dabei ist i ein Element aus der Indexmenge I der Datenblöcke. Die Variablen $d_i^{statisch}/d_i^{dynamisch}$ beschreiben die Partitionsgröße des statischen bzw. dynamischen DSPM-Bereiches.

Zur Sicherstellung, dass die Summe des Speicherbedarfs aller im Scratchpad-Speicher allozierten Blöcke nicht die Kapazität des statischen DSPM-Bereiches überschreitet, wird eine Nebenbedingung eingeführt:

$$\sum_{i \in I} x_i * S_i \leq d_t^{statisch} \tag{4.2}$$

Die Zielfunktion des ILPs ist die Kostenfunktion (Gleichung 4.3) des Hauptbasisblockes, welche minimiert werden soll [Rot08].

$$c_{main} \rightsquigarrow min \tag{4.3}$$

Der nächste Schritt der Optimierung ist das Lösen des ILPs, hierdurch erhält man eine optimale Verteilung der Datenblöcke auf den statischen Teil des Scratchpad-Speichers.

Nachdem das ILP zum Ersten mal gelöst wurde, beginnt der zweite Durchlauf der Allokation für diese Konfiguration, der den dynamischen Speicherbereich mit Datenblöcken füllt.

Während der zweiten Durchführung des Lösens, werden in [Ris11] die Entscheidungsvariablen der Blöcke, welche bereits während des ersten Lösungsvorganges in den statischen SPM-Bereich alloziert wurden, aus der Zielfunktion des ILPs entfernt.

Dieser Schritt ist jedoch nicht unproblematisch. Durch das Entfernen der Entscheidungsvariablen wird die Modellierung des Kontrollflusses des Programmes inkonsistent und es können Fehler bei der erneuten Lösung des ILPs entstehen, da keine Informationen über diese Blöcke berücksichtigt werden. Ebenfalls inkorrekt ist, dass die Entscheidungsvariablen weiterhin in der Nebenbedingung für die Scratchpad-Speicherkapazität exisitieren. Dies hat zur Folge, dass bei einem weiterem Lösen des ILPs keine weiteren Datenblöcke bzw. Basisblöcke mehr alloziert werden können, da dies durch die Kapazitätsnebenbedingung verhindert wird. Dies hat zur Folge, dass der dynamische Bereich leer bleibt.

Code-Scratchpad-Speicherallokation

Die Speicherallokation von Programminstruktionen verläuft sehr ähnlich wie die für Daten. Allerdings werden bei Programminstruktionen keine Datenblöcke alloziert, sondern Basisblöcke des Programmes. Die Berechnung der Gewinne durch eine Verschiebung in den schnellen Scratchpad-Speicher verläuft hier ebenfalls anders (siehe [Kle08]).

Nach [Ris11] werden auch hier die Entscheidungsvariablen, welche die Basisblöcke modellieren, vor dem zweiten Lösen des ILPs aus der Kostenfunktion entfernt, was die gleichen Folgen wie bei der Allokation des DSPM hat. Es werden keine zusätzlichen Basisblöcke mehr alloziert und der dynamische PSPM-Bereich bleibt leer.

4.2 Optimierung der gesamten Taskmenge

Nachdem für jeden Task die Worst-Case Laufzeiten der verschiedenen Speicherkonfigurationen berechnet wurden, kann der eigentliche Optimierungsschritt diese Informationen nutzen und eine optimale Belegung für die gesamte Taskmenge ermitteln.

Dazu werden für jedes der drei Verfahren (hybrides, dynamisches und statisches) zwei ILP-Formulierungen aufgestellt, jeweils separat für Datenblöcke und Code-Basisblöcke. Die folgenden ILP-Formulierungen basieren auf denen von K. Risto (s. [Ris11]).

4.2.1 ILP-Formulierungen

In einem Multi-Task System ist es notwendig, dass es einen Scheduler gibt, welcher verwaltet wie lange und wann ein bestimmter Task Rechenzeit zugewiesen bekommt. Durch diesen Scheduler entstehen zusätzliche Wartezeiten zwischen den Taskwechseln.

Dieser Scheduler-Overhead wurde durch K. Risto mit Hilfe des Realzeit Betriebssystemes „ERIKA“ ermittelt [Ris11]. Es wird angenommen, dass ein Round-Robin Schedulingverfahren eingesetzt wird [ERI11]. Mit Hilfe des in „ERIKA“ implementierten Schedulers ließ sich so eine Laufzeit von ungefähr 274 Zyklen pro Taskwechsel ermitteln.

Insgesamt sind die Kosten pro Taskwechsel definiert als:

$$schedCost(x) = nrp(x) * 274 \tag{4.4}$$

, wobei $nrp(x)$ für die Anzahl der Verdrängungen (englisch: preemptions) durch den Scheduler steht, welche sich wie in Gleichung 4.5 definieren lässt.

$$nrp(x) = \frac{1000 * x}{period_{sched} * clock_{cpu}} \tag{4.5}$$

Dabei steht x für die Ausführungszeit in Zyklen, $period_{sched}$ für die Dauer der Scheduling Zeitscheibe in Millisekunden und $clock_{cpu}$ für den CPU Takt in Hertz.

Die ILP-Formulierungen für Daten-Scratchpad-Speicher und Code-Scratchpad-Speicher sind in diesem Schritt weitestgehend analog, weshalb nur auf die Definition für den Code-Scratchpad-Speicher eingegangen wird und im Falle von Unterschieden nur auf diese eingegangen wird.

Statische Multi-Task Scratchpad-Speicherallokation

Wie zuvor erwähnt, hat jeder Task bei der statischen Scratchpad-Speicherallokation einen fest zugewiesenen Speicherbereich, welcher während der gesamten Ausführungszeit gleich bleibt. Durch einen Taskwechsel entstehen aus diesem Grund keine weiteren Kosten, als der Taskwechsel selbst. Die Daten der anderen Tasks verbleiben dabei an ihrer Stelle. Abbildung 4.2 zeigt schematisch wie der Speicher während der Ausführung mit einer statischen Allokation aussieht.

Die statische Allokation hat den Nachteil, dass sich die Tasks den Scratchpad-Speicher komplett teilen müssen, was, gerade bei geringen Speichergrößen, zu einer gegenseitigen Einschränkung und so nur zu geringen Reduktionen der WCET führt.

Die Entscheidungsvariablen dieser ILP-Formulierung haben die Form x_{it}^{P}, wobei i für die statische Speichergröße des Tasks t steht. P steht für Programmspeicher, wie zuvor erwähnt ist die Definition des Datenspeichers äquivalent, hier würde ein D an den Entscheidungsvariablen dies kennzeichnen. Im Folgendem gibt S^{PSPM} die Größe des Code-Scratchpad-Speichers an, sowie T die Menge der Tasks.

Damit die Allokation korrekt funktioniert, muss sichergestellt werden, dass jedem Task nur eine Konfiguration zugeteilt wird. Dies wird durch die Nebenbedingung (Gleichung 4.6) garantiert.

$$\forall t \in T : \sum_{i=0}^{S^{PSPM}} x_{it}^{D} = 1 \tag{4.6}$$

Damit die gesamte Kapazität des Scratchpad-Speichers nicht überschritten wird, wird eine weitere Nebenbedingung eingeführt (Gleichung 4.7). Diese stellt sicher, dass die Summe aller Konfiguration, aller Tasks, kleiner oder gleich der Speichergröße des Programm-Scratchpad-Speichers ist.

$$\sum_{t \in T} \sum_{i=0}^{S^{PSPM}} x_{it}^{P} * i \leq S^{PSPM} \tag{4.7}$$

Die Zielfunktion dieser ILP-Formulierung stellt die Worst-Case-Laufzeit der gesamten Taskmenge dar. Diese gilt es zu minimieren. Gleichung 4.8 modelliert, wie sich die Worst-Case Laufzeit der gesamten Taskmenge zusammensetzt.

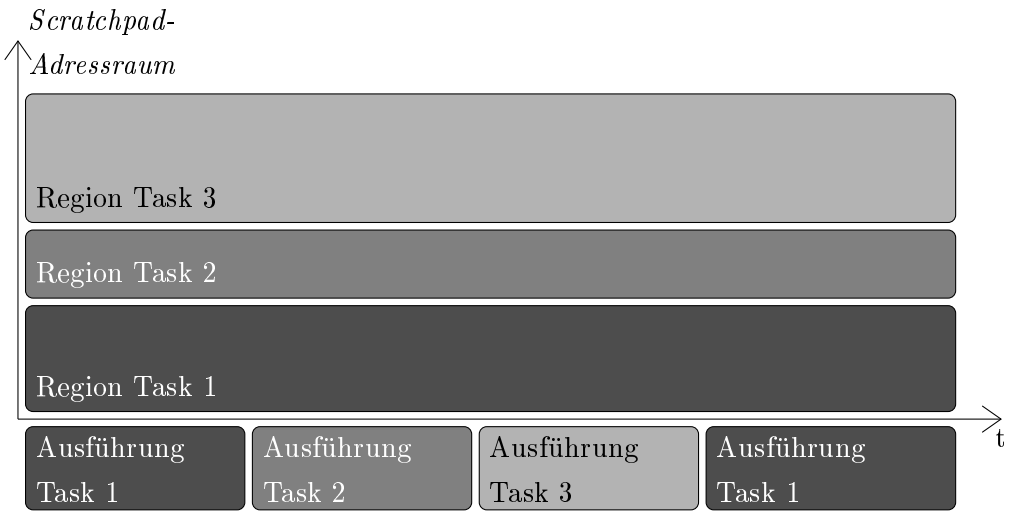


Abbildung 4.2: Schematische statische Allokation

$$wcet_P^{stat} = \sum_{t \in T} \sum_{i=0}^{S^{PSPM}} x_{it}^P * (wcet_P^t(i) + schedCost(wcet_P^t(i))) \tag{4.8}$$

Die Funktion $wcet_P^t(i)$ steht dabei für die Worst-Case Laufzeit von Task t mit der Speicherkonfiguration i in Bytes. Dies sind die Informationen, die im ersten Schritt, der „Gain-Computation“ berechnet wurden. $schedCost(wcet_P^t(i))$ (siehe Gleichung 4.4) gibt die zusätzlich benötigte Zeit für einen Taskwechsel an, die zuvor definiert wurde.

Das Ziel des ILPs ist es diese Funktion zu minimieren.

$$wcet_P^{stat} \rightsquigarrow min \tag{4.9}$$

Im folgenden Abschnitt wird mit der dynamischen Scratchpad-Speicherallokation eine weitere Möglichkeit zur Speichereinteilung vorgestellt.

Dynamische Multi-Task Scratchpad-Speicherallokation

Das dynamische Verfahren der Scratchpad-Speicherallokation arbeitet im Gegensatz zu dem statischen Verfahren nicht mit festen Speicherregionen pro Task. Stattdessen wird jedem Task der gesamte Scratchpad-Speicher zur Verfügung gestellt. Abbildung 4.3 zeigt wie eine Belegung beispielsweise aussehen könnte. Die Skizze zeigt, dass der gesamte

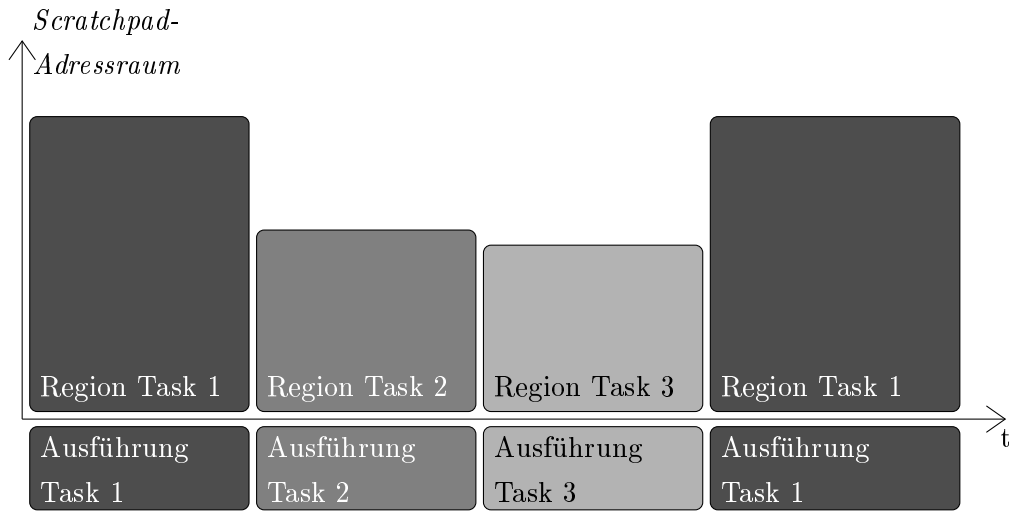


Abbildung 4.3: Schematische dynamische Allokation

Speicher immer nur durch einen Task belegt wird und nicht immer vollständig ausgelastet sein muss. Ein Nachteil dieses Verfahren ist, dass während eines Taskwechsels der gesamte Daten-Scratchpad-Speicher in den Hauptspeicher kopiert werden muss und anschließend der Inhalt des aktivierenden Tasks von dem Hauptspeicher in den Scratchpad-Speicher kopiert werden muss.

Dadurch entstehen zusätzliche Kosten während eines Taskwechsels, die man bei der Bestimmung der optimalen Speicherbelegung berücksichtigen muss. Dazu bezieht sich K. Risto in [Ris11] auf die Arbeit von Lutz Krumme [Kru10]. Für das Kopieren zwischen den beiden Speichern nutzt Krumme den DMA-Controller des TriCore Prozessors. Dabei handelt es sich um eine Einheit auf dem Prozessor, welche direkt auf den Speicher zugreifen kann, ohne dafür die CPU zu beanspruchen. Zur Bestimmung der Auswirkung auf die Worst-Case Laufzeit wurde in [Kru10] ebenfalls das Betriebssystem „ERIKA“ [ERI11] verwendet.

Der DMA-Controller des TriCore Prozessors ist in zwei Subblöcken strukturiert, welche jeweils acht Kanäle besitzen über die jeweils eine DMA-Transaktion möglich ist. Eine Transaktion kann aus bis zu 511 DMA-Transfers bestehen, welche aus 1, 2, 4, 8 oder 16 DMA-Moves bestehen können. Ein DMA-Move kann entweder ein 8, 16 oder 32 Bit

breites Datum enthalten. Die Laufzeittests von Krumme ergaben, dass sich die Laufzeiten der DMA-Aktionen wie in Gleichung 4.10 definieren lassen.

$$dmaCost(x, y) = 2 * x + y + 25 \tag{4.10}$$

Hier steht x für die Anzahl der DMA-Moves und y für die der DMA-Transfers. Es findet zusätzlich noch eine Korrektur um 25 Zyklen statt, da ansonsten der Wert der Zyklen laut Krumme für kleine x und y unter dem realen Wert liegt.

Für die ILP-Formulierung wird eine Funktion benötigt, die für eine zu kopierende Anzahl an Bytes jeweils die Laufzeit in Zyklen angibt. Diese wird in Gleichung 4.13 definiert. Diese Funktion ist abhängig von der Anzahl der DMA-Moves (Gleichung 4.11) und der Anzahl der DMA-Transfers (Gleichung 4.12). Die Anzahl der DMA-Transfers ist dabei abhängig von der Anzahl der DMA-Moves pro Transfer n_{mpt} und die Anzahl der DMA-Moves ist abhängig von der Wordbreite der CPU, in dem Falle des TriCore Prozessors 32 Bit.

$$n_{moves} = \lceil \frac{x}{word_{cpu}} \rceil \tag{4.11}$$

$$n_{transfers} = \lceil \frac{n_{moves}}{n_{mpt}} \rceil \tag{4.12}$$

$$cpCost(x) = dmaCost(n_{moves}, n_{transfers}) \tag{4.13}$$

Mit dieser Hilfsfunktion ist es nun möglich die ILP-Formulierungen für die dynamische Allokation, sowie für die hybride Allokation, zu erstellen.

Für die dynamische Scratchpad-Speicherallokation haben die Entscheidungsvariablen die Form x^P_{jt}. j gibt die Größe des dynamisch allozierten Bereiches an, t steht für einen Index aus der Taskmenge T und P für Programm-Scratchpad.

Für die Korrektheit der Speicherallokation, darf jedem Task nur eine Speicherkonfiguration zugewiesen werden, was durch Gleichung 4.14 gewährleistet wird.

$$\forall t \in T : \sum_{j=0}^{S^{PSPM}} x^P_{jt} = 1 \tag{4.14}$$

Da jedem Task bei der dynamischen Speicherallokation der gesamte Speicher zur Verfügung steht, ist die Nebenbedingung für die Kapazität des Speichers (vgl. Gleichung 4.7) nicht mehr die Summe der Konfigurationen über die gesamte Taskmenge, sondern es ist nur die Konfiguration der einzelnen Tasks relevant. Dadurch ergibt sich die Nebenbedingung:

$$\forall t \in T : \sum_{j=0}^{S^{PSPM}} x_{jt}^{P} * j \leq S^{PSPM} \tag{4.15}$$

Die Zielfunktion wird wieder durch die Summe der System-WCET bestimmt, welche durch Gleichung 4.16 definiert ist.

$$\begin{aligned} wcet_P^{dyn} = \sum_{t \in T} \sum_{j=0}^{S^{PSPM}} x_{jt}^{P} * (&wcet_P^t(j) + \\ &(cpCost(j) * npr(wcet_P^t(j))) + \\ &(schedCost(wcet_P^t(j)))) \end{aligned} \tag{4.16}$$

Auf die System-WCET haben die WCETs der einzelnen Tasks, die Kosten des Schedulers beim Taskwechsel und die Kopierkosten direkten Einfluss. Die Kopierkosten sind ebenfalls abhängig von der Anzahl der Unterbrechungen (siehe Gleichung 4.5). Für die Formulierung des Datenspeichers steht vor den Kopierkosten noch ein zusätzlicher Faktor von zwei, da der alte Speicherinhalt zusätzlich gesichert werden muss, da sich Programmdaten im Gegensatz zu Programminstruktionen ändern.

Auch bei der dynamischen Speicherallokation gilt es die WCET zu minimieren.

$$wcet_P^{dyn} \rightsquigarrow min \tag{4.17}$$

Abschließend wird mit der hybriden Scratchpad-Speicherallokation noch eine Alternative zu der dynamischen und statischen Scratchpad-Speicherallokation vorgestellt, die beide Verfahren kombiniert.

Hybride Multi-Task Scratchpad-Speicherallokation

Die hybride Scratchpad-Speicherallokation teilt den Scratchpad-Speicher in zwei Bereiche auf. Der eine Teil wird den Tasks statisch zugewiesen und ein zweiter Teil steht dem zur Zeit ausgeführtem Task exklusiv zur Verfügung. Abbildung 4.4 zeigt diese beiden

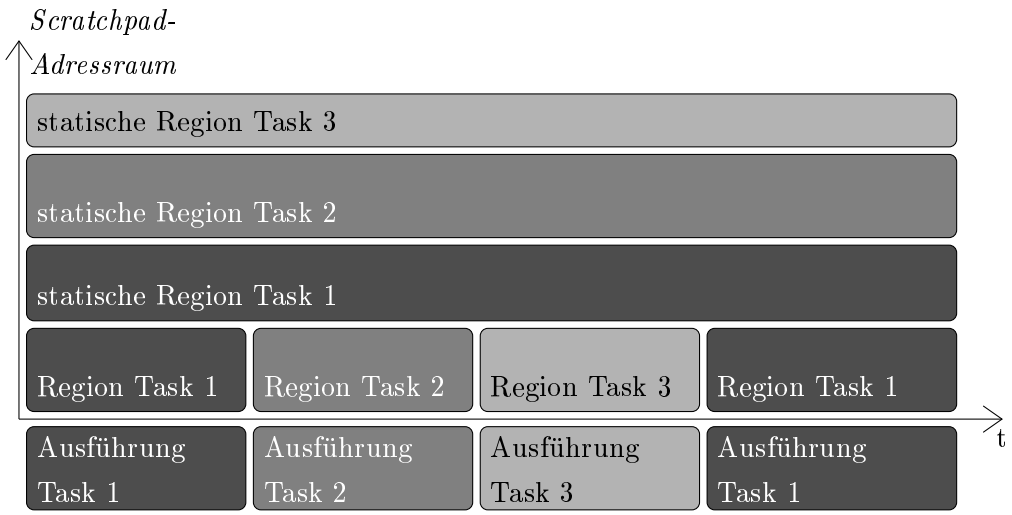


Abbildung 4.4: Schematische hybride Allokation

Bereiche, dabei ist der obere Teil in der Abbildung der statische Bereich und der untere der dynamische Bereich.

Für die ILP-Formulierungen haben die Entscheidungsvariablen die Form x^P_{ijt}, dabei steht i für die Größe des statischen Speicherbereiches, j für die des dynamischen, t für einen Index der Taskmenge T.

Damit jedem Task auch bei dieser Speicherallokation nur eine Konfiguration des Speichers zugeteilt wird, muss die Nebenbedingung (Gleichung 4.18) während des Lösens des ILPs beachtet werden.

$$\forall t \in T : \sum_{i=0}^{S^{PSPM}} \sum_{j=0}^{S^{PSPM}-i} x^P_{ijt} = 1 \tag{4.18}$$

Außerdem muss auch bei dieser ILP-Formulierung sichergestellt werden, dass niemals die Kapazität des Scratchpad-Speichers überschritten wird. Die Nebenbedingung (Gleichung 4.19) ist eine Kombination der Nebenbedingungen des statischen und des dynamischen Verfahrens (vgl. Gleichung 4.7 und Gleichung 4.15).

$$\forall t \in T : \sum_{t \in T} \sum_{i=0}^{S^{PSPM}} \sum_{j=0}^{S^{PSPM}-i} x^P_{ijt} * i + \sum_{i=0}^{S^{PSPM}} \sum_{j=0}^{S^{PSPM}-i} x^P_{ijt} * j \leq S^{PSPM} \tag{4.19}$$

Der vordere Teil der Nebenbedingung sichert, dass die Summe der statischen Speicheranteile aller Tasks in den Scratchpad-Speicher passen und der Hintere regelt diesen Umstand für die dynamischen Anteile der einzelnen Tasks.

Die Zielfunktion des ILPs ist auch hier die WCET des gesamten Systemes, welche wie folgt definiert ist:

$$wcet_P^{hybrid} = \sum_{t \in T} \sum_{i=0}^{S^{PSPM}} \sum_{j=0}^{S^{PSPM}-i} x_{ijt}^P * (wcet_P^t(i,j) + (cpCost(j) * npr(wcet_P^t(i,j))) + (schedCost(wcet_P^t(i,j)))) \tag{4.20}$$

Dabei handelt es sich um die Summe der spezifischen WCET der einzelnen Tasks, sowie die entstehende Zeit während eines Taskwechsels und den Kopierkosten für den dynamischen Speicherbereich. Für den Daten-Scratchpad-Speicher muss erneut der Faktor zwei zusätzlich vor den Kopierkosten stehen, da die Daten des alten Tasks vor dem Taskwechsel gesichert werden müssen.

Das folgende Kapitel beschreibt die notwendigen Änderungen an den ILP-Formulierungen sowie ihre Implementierung innerhalb des WCC-Übersetzers.

5 Realisierung

Die Realisierung der Multi-Task Scratchpad-Speicherallokation wurde innerhalb des WCC-Übersetzers, der von der „Design Automation for Embedded Systems Group“ der Fakultät für Informatik der technischen Universität Dortmund entwickelt wurde, umgesetzt. Dazu wurde auf bestehenden Programmcode zurückgegriffen und dieser auf die neue Multi-Task Optimierung angepasst. Durch die objektorientierte Struktur des WCC-Übersetzers ist es möglich andere Programmteile wiederzuverwenden. So konnte auf die statische Scratchpad-Speicherallokation für Code und Daten zurückgegriffen werden, die teilweise sehr ähnlich der *Gain-Computation* ist. Durch die Arbeit von K. Risto war ebenfalls schon ein großer Teil für die Multi-Task Optimierung implementiert, so dass direkt die Änderungen implementiert werden konnten.

Die Durchführung der Speicherallokation verläuft in zwei Ausführungsschritten. Im ersten Schritt werden durch die *Gain-Computation* die Laufzeittabellen der verschiedenen Tasks berechnet (siehe Abschnitt 4.1), welche dann in einem zweiten Ausführungsschritt eingelesen werden und zur Erstellung der ILP-Formulierungen (siehe Abschnitt 4.2) verwendet werden.

5.1 Implementierung der Gain-Computation

Die wichtigsten Änderungen an der Implementierung sind im dem ersten Schritt, der *Gain-Computation* notwendig, da es essenziell ist, dass die korrekten Zeitinformationen für die spätere Optimierung vorliegen. Die notwendigen Informationen sind alle WCETs für alle möglichen Speicherkonfigurationen von 0 Bytes bis zu einer bestimmten Größe SPM, die für die Optimierung verwendet werden soll.

Die größten Änderungen waren dazu an den ILP-Formulierungen notwendig. Die Implementierung aus [Ris11] ist dabei so vorgegangen, dass nach dem ersten Lösen des ILPs die Entscheidungsvariablen einfach aus der Zielfunktion entfernt wurden (siehe Abschnitt 4.1). Dieser Schritt wurde in der Implementierung entfernt und durch eine

neue Funktion ersetzt, welche eine Liste der Datenblöcke bzw. Basisblöcke erstellt, die während des ersten Lösungsvorganges alloziert wurden. Dies ist notwendig, damit während des nächsten Lösungsvorganges die Daten- bzw. Basisblöcke, welche bereits in die statische SPM-Partition eingelagert wurden, auch weiterhin korrekt betrachtet werden können.

Dazu wird eine Hilfsmenge B_{alloc} eingeführt, welche die Indizes der Daten- bzw. Basisblöcke enthält, die nach dem ersten Lösen des ILPs dem statischen Bereich des DSPM bzw. PSPM zugeordnet wurden.

Nebenbedingungen für die DSPM-Gain-Computation: Für den zweiten Lösungsvorgang muss die Nebenbedingung, die absichert, dass die allozierten Datenblöcke in die dynamische Partition des DSPM passen, so abgeändert werden, dass Datenblöcke, die bereits in die statische Partition des DSPM alloziert wurden, keinen Einfluss auf die Kapazitätsbeschränkung haben. Dazu wird die folgende Nebenbedingung eingeführt, welche nur die Datenblöcke betrachtet, die nicht gleichzeitig in der Menge B_{alloc} liegen.

$$\sum_{i \in I \setminus B_{alloc}} x_i * S_i \leq d_t^{dynamisch} \tag{5.1}$$

Desweiteren muss für den zweiten Durchlauf sichergestellt werden, dass die Entscheidungsvariablen des ersten Durchlaufes, für die gilt $x_i = 1$, garantiert so bleiben. Dafür wird mit Gleichung 5.2 eine zusätzliche Nebenbedingung eingeführt, die diese Anforderung sicherstellt.

$$\sum_{i \in B_{alloc}} x_i = |B_{alloc}| \tag{5.2}$$

Nebenbedingungen für die PSPM-Gain-Computation: Die *Gain-Computation* für den PSPM arbeitet prinzipell genau wie die des DSPM. Allerdings hat sich während des Testens gezeigt, dass es zu Programmabstürzen kommt, wenn man die Nebenbedingung Gleichung 5.1 ohne Änderung übernimmt. Die statische Scratchpad-Speicherallokation fügt für jeden Basisblock in das ILP eine Nebenbedingung ab, die abhängig von der Speicherposition der Nachfolger zusätzlich 12 Bytes einkalkuliert.

Diese zusätzliche Speicherkapazität wird für weitere Instruktionen benötigt, welche für einen Sprung zwischen zwei Speicherbereichen erforderlich sind.

Das Problem dieser Nebenbedingung, bezüglich der Scratchpad-Speicherallokation mit zwei Bereiche ist, dass dieser Fall nicht berücksichtigt wird und die 12 Bytes nur zusätzlich reserviert werden, falls die Nachfolger im Hauptspeicher liegen. Der Fall, dass ein Basisblock in der statischen Sektion liegt und ein Nachfolger in der dynamischen Sektion wird nicht berücksichtigt, da die Region für beide Sektionen die Gleiche ist (PSPM).

Um diesen Umstand zu umgehen, muss nach dem ersten Lösen des ILPs für jeden Basisblock, der nicht in die statische Sektion zugeordnet wurde, überprüft werden, ob der Basisblock Nachfolger besitzt und in diesem Falle überprüft werden ob die Nachfolger in dem statischem Bereich liegen. Falls nicht müssen für diesen Basisblock 12 Bytes zusätzliche Speichergröße einkalkuliert werden. Fügt man diesen Speicherbedarf nicht zusätzlich zu, führt dies zu einem Abbruch des Übersetzungsvorganges, da es notwendig ist, zusätzlichen Programmcode einzufügen, der die Sprünge zwischen zwei Speicherbereichen ausführt. Ohne den zusätzlichen Speicherbedarf passt dieser Programmcode nicht in den Scratchpad-Speicher und es führt zu einem Speicherüberlauf. Dabei sind die 12 Bytes eine Überabschätzung und nicht der exakt benötigte Speicherbedarf.

Dieser zusätzliche Speicherbedarf wird wie folgt als jc_i definiert.

$$jc_i = \begin{cases} 24 & \text{falls beide Nachfolger innerhalb einer anderen Sektion liegt} \\ 12 & \text{falls ein Nachfolger innerhalb einer anderen Sektion liegt} \\ 0 & \text{falls beide Nachfolger innerhalb der gleichen Sektion liegen} \end{cases} \tag{5.3}$$

Die Nebenbedingung für die eingeschränkte Speicherkapazität lässt sich dadurch wie folgt definieren:

$$\sum_{i \in I \setminus B_{alloc}} x_i * (S_i + jc_i) \leq p_t^{dynamisch} \tag{5.4}$$

.

Für die PSPM-Gain-Computation ist es ebenfalls wichtig, dass die Nebenbedingung Gleichung 5.2 gilt, welche sicherstellt, dass Basisblöcke die in die statische PSPM-Partition alloziert worden sind korrekt betrachtet werden.

Durch die Einführung einer Hilfsmenge und der korrekten Betrachtung bereits allozierter Blöcke werden auch bei der zweiten Durchführung der *Gain-Computation* einer Speicherpartitionierung Daten- bzw. Basisblöcke dem dynamischen Bereich zugeordnet.

Aufgrund von Optimierungen bezüglich der Laufzeit des Optimierungsschrittes der *Gain-Computation*, wird das ILP nur einmal erstellt und nicht mehr, wie zuvor, nach jeder Speichergröße. Dies spart mindestens zwei Laufzeitanalysen pro Speicherpartitionierung ein. Jedoch ist es daher notwendig, die entfernten Entscheidungsvariablen, nach dem zweiten Durchlauf, wieder der Nebenbedingung für die Scratchpad-Speicherkapazität hinzuzufügen, da sonst nach jeder Speicherpartitionierung Entscheidungsvariablen verloren gehen würden und dies so zu einer fehlerhaften Allokation führt. Außerdem muss die Nebenbedingung Gleichung 5.2 wieder entfernt werden, da diese für eine neue Speicherpartitionierung keine weitere Relevanz hat.

Da die Laufzeit der *Gain-Computation* äußerst hoch ist, wurde noch ein optionaler Parameter für den Übersetzer eingeführt, welcher es ermöglicht die Schrittweite der *Gain-Computation* einzustellen. Der Standardwert ist 128 Bytes, jedoch besteht die Möglichkeit diesen zu erhöhen, was für große Kapazitäten (ab ca. 4 KB aufwärts) unabdingbar ist, da die Laufzeitanalysen, welche für die *Gain-Computation* notwendig sind, ansonsten zu viel Arbeitsspeicher benötigen würden.

Mit den genannten Änderungen erstellt die *Gain-Computation* korrekte Laufzeittabellen, welche die absoluten Worst-Case Laufzeiten für die jeweiligen Tasks enthalten. Diese Tabellen werden in Dateien zwischengespeichert, welche durch den zweiten Schritt der Optimierung erneut eingelesen werden. Es ist möglich die *Gain-Computation* in mehrere Prozesse aufzuteilen. Dazu wird jedem Prozess ein Intervall als Parameter übergeben, welches angibt für welches SPM-Partitionierungen die WCETs der Tasks berechnet werden sollen. Dadurch werden für jeden Task mehrere Dateien erstellt, welche die Laufzeittabellen, das gewählte Intervall, sowie die Schrittweite enthalten. Dies ermöglicht die Umgehung von Speicherüberläufen bei der *Gain-Computation*, da die Laufzeitanalysen einen sehr hohen Speicherbedarf haben und nach jeder der Speicherbedarf des WCC-Übersetzers ansteigt. Daher kann es vorkommen, dass nach einigen Laufzeitanalysen der WCC-Übersetzer mehr als 4GB Arbeitsspeicher benötigen würde, was jedoch auf Grund seiner 32 Bit-Architektur nicht möglich ist und dies so einen Abbruch des Übersetzungsvorganges auslösen würde.

Der Vorteil der Zwischenspeicherung in Dateien ist, dass dadurch eine abgebrochene *Gain-Computation* wieder aufgenommen werden kann und so die bereits aufgewendete Rechenzeit nicht verschwendet wurde. Außerdem ermöglicht eine Speicherung in Dateien dieser Informationen, dass die *Gain-Computation* aufgeteilt wird und so eine Parallelisierung möglich ist, was die benötigte Rechenzeit auf einem heutigem Mehrkernsystem drastisch senkt.

5.2 Implementierung der Multi-Task Speicherallokation

Durch die Änderungen am Dateiformat für die Laufzeittabellen, ist es erforderlich, auch das Einlesen dieser Daten zu überarbeiten.

Für die zuvor genannten Funktion mussten die vorhandenen Einleseroutinen so geändert werden, dass diese nicht mehr nur eine Tabellendatei pro Task einlesen. Stattdessen ist es notwendig, dass der zweite Optimierungsschritt in einem bestimmten Ordner alle verfügbaren Laufzeit-Dateien einliest und daraus die notwendigen Informationen für die Multi-Task Optimierung erstellt. Es wird so vorgegangen, dass zu erst alle gefundenen Dateien eingelesen werden und jedesmal wenn der Übersetzer auf einen neuen Task trifft, für diesen eine neue interne Datenstruktur angelegt wird, welche später mit den Laufzeit-Informationen gefüllt wird. Zuvor muss aber noch ermittelt werden, bis zu welcher SPM-Kapazität zuvor die *Gain-Computation* aufgerufen wurde. Dies geschieht durch eine Suche des Maximums der oberen Grenze der SPM-Kapazität während der *Gain-Computation* aus den verfügbaren Laufzeit-Dateien. Ist das Maximum gefunden, so werden die Datenstrukturen der jeweiligen Tasks mit den Information der Dateien gefüllt.

Mit diesen Daten ist es möglich eine optimale Belegung für eine Taskmenge zu finden. Die Laufzeit dieser Operation ist im Vergleich zur *Gain-Computation* sehr gering, da nur sechs ILPs aufgestellt und gelöst werden müssen. Jeweils für die hybride, dynamische und statische Allokation und dies für den DSPM und PSPM separat.

Im folgenden Kapitel wird beschrieben wie die Multi-Task Optimierung getestet wurde und wie die Güte der Optimierung ist.

6 Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation der vorgestellten Multi-Task Optimierung. Die Evaluation ist wichtig, da überprüft werden muss, welchen Einfluss verschiedene Taskmengen auf die Qualität der Optimierung haben. So muss überprüft werden, ob sich alle Taskmengen gleich gut optimieren lassen oder ob Unterschiede existieren. Außerdem muss überprüft werden, wie groß die Auswirkungen verschiedener SPM-Kapazitäten sind. So gilt es festzustellen ob bereits kleine Mengen SPM zu einer bemerkbaren Reduktion der WCET führen oder ob nur große Kapazitäten einen Nutzen zeigen. Dazu wird zuerst die Testumgebung und die Benchmark Taskmengen vorgestellt. Des Weiteren werden die Testergebnisse mit denen aus [Ris11] verglichen.

6.1 Benchmark-Umgebung

Als System für die Benchmarks wurde ein Linux-Server mit der Linux-Distribution Debian 6 (Squeeze) verwendet. Es handelt sich bei dem Server um ein 8 Prozessor System mit der Möglichkeit, unter optimalen Bedingungen 2 Prozesse pro Prozessorkern auszuführen (Hyper-Threading). Dies ist aber nur selten der Fall, so dass Hyper-Threading nicht die gleiche Performance-Steigerung wie ein echter Prozessorkern erzielt. Es konnten also durchgängig theoretisch 16 Prozesse arbeiten, wobei beachtet werden musste, dass der Arbeitsspeicher, mit 20 GB, ausreichend war, da die Laufzeitanalysen sehr aufwändig sind und es nicht selten vorkommt, dass dazu mehr als 1 GB pro Prozess benötigt werden. Aus diesem Grund wurden in der Regel nur 12 Prozesse gleichzeitig gestartet.

6.2 Benchmark-Taskmengen

Um die Güte der Optimierung zu bestimmen wurden fünf verschiedene Taskmengen ausgewählt. Zwei Taskmengen davon sind direkt aus [Ris11] übernommen (Set1 und

Set2), um so einen Vergleich erstellen zu können. Die verwendeten Taskmengen sind in Tabelle 6.1 aufgeführt. Zusätzlich ist die Größe der Programminstruktionen und der Programmdaten in Bytes angegeben.

Die Benchmark Taskmengen Set2-Set5 wurden aus [Moe11] übernommen und können so mit den Optimierungen dieser Arbeit ebenfalls verglichen werden.

Die erste Taskmenge enthält den Task *adpcm_decoder*, dabei handelt es sich um einen Demodulator des ADPCM Verfahrens. Dies ist eine Modulationsart für digitale Signale. Der Benchmark ist Teil der MRTC-Benchmarksammlung [MRT11]. Die Benchmarks *edge_detect*, *latnrm_32_64*, *g723_marcuslee_decoder*, *trellis*, *fft_256* und *fft_1024* sind spezielle Benchmarks für DSP-Prozessoren aus der Sammlung UTDSP [UTD11].

edge_detect ist ein Filter für Bilder, der Kanten findet und *latnrm_32_64* ist ein Lattice-Filter. Dies ist eine in der Signalverarbeitung verbreitete Filterstruktur. *trellis* ist eine Implementierung eines Trellis-Code Dekodieres. Trellis-Code ist eine kodierte Modulationstechnik, welche in der Übertragungstechnik zum Einsatz kommt.

Der Task *md5* stammt aus der NetBench Sammlung [MMsH01]. Dieser Task berechnet eine MD5 Checksumme. Dies ist ein häufig eingesetztes Verfahren zur Validierung von Daten nach dem Datentransfer. Der Task *crc* ist auch ein Verfahren zur Verifizierung von Datenübertragungen und wird häufig im Netzwerkbereich verwendet, da es mit relativ geringem Aufwand verbunden ist.

Die Tasks *g721_encode*, *g723_encode* sowie *g723_marcuslee_decoder* sind Sprachcodecs, welche speziell für geringe Datenraten entwickelt wurden. Beispielsweise wird der Codec innerhalb des DECT-Standards verwendet, welcher sehr verbreitet unter Funktelefonen ist. Der Task *h264_ldecode_block* implementiert einen Dekoder für den h264 Videocodec, ein modernes Videokompressionsverfahren, welches unter anderem auch für Blu-ray Discs verwendet wird. Der Task ist Teil des MediaBench Benchmarks [LPMS97].

Die Tasks *fft_256* und *fft_1024* sind Realisierungen der schnellen Fourier-Transformation (englisch: fast fourier transform). Dies ist eine effiziente Realisierung der Fourier-Transformation, welche ein zeit-diskretes Signal in den Frequenzbereich transformiert. Dies ist eine für die digitale Signalverarbeitung unabdingbare Operation, welche zum Beispiel für Musikkompression und Signalanalysen eingesetzt wird.

Bei der Auswahl der Taskmengen wurde darauf geachtet, dass sich ein Beispiel aus der Praxis für jede Menge finden lässt, wo eine Kombination der Tasks eingesetzt werden

Taskmenge	Taskname	Code-Größe	Daten-Größe	WCET
T1	adpcm_decoder	2310	1296	631874
	edge_detect	770	12328	38927307
	g721_encode	4856	2512	2280401
	g723_encode	4856	2608	1706276
	latnrm_32_64	418	1024	311380
	md5	3980	72	49105437
Set1	edge_detect	770	12328	38927307
	g721_encode	4856	2512	2280401
	latnrm_32_64	418	1024	311380
Set2	crc	518	1408	252161
	g721_marcuslee_decoder	144	10048	231114
	h264_ldecode_block	13994	13128	327745
Set3	crc	518	1408	252161
	fft_1024	946	16384	15167802
	gsm_decode	6534	8112	17370290
	trellis	3086	984	1187540
Set4	crc	518	1408	252161
	fft_256	948	4096	1041758
	g721_marcuslee_decoder	144	10048	231114
	h264_ldecode_block	13994	13128	231114
	latnrm_32_64	418	1024	311380

Tabelle 6.1: Verwendete Benchmark Taskmengen mit Code- und Daten-Größe in Bytes sowie WCET in Zyklen

könnte. Bei *T1* könnte es sich um die Realisierung einer Sendeeinheit für z.B. Videotelefonie handeln, *Set1* könnte eine reine Videoübertragung realisieren, *Set3* könnte eine Empfangseinheit für einen Videostream über eine Datenleitung darstellen, *Set3* könnte zum Beispiel die Empfangseinheit eines GSM-Handys darstellen und *Set4* könnte genau wie *Set2* die Software für eine Videoempfangseinheit sein.

6.3 Durchführung der Benchmarks

Die Taskmengen wurden zur Optimierung mit der Übersetzeroptimierungsstufe `O2` übersetzt, so dass zusätzlich zu der Multi-Task Optimierung noch normale Übersetzeroptimierungen durchgeführt wurden. Für jeden Task wurde die *Gain-Computation* von 0 Bytes bis zu einer Größe von 4096 Bytes in einer Schrittweite von 128 Bytes durchgeführt. Um eine gute Parallelität der *Gain-Computation* zu erreichen, wurde nicht ein Prozess von 0 Byte bis 4096 Bytes für die gesamte Taskmenge gestartet, sondern viele Einzelprozesse, die jeweils für nur einen Task einen bestimmten Bereich der Gain-Tabellen berechnet hat (z.B. ein Prozess von 0 Bytes bis 256 Bytes und ein anderer von 384 Bytes bis 512 Bytes). Jedoch sollten wie zuvor erwähnt, nicht mehr Prozesse parallel gestartet werden als Prozessoren im System vorhanden sind. Durch die Parallelität war es möglich die gesamte *Gain-Computation* für alle Tasks aus Tabelle 6.1 innerhalb von einigen Stunden durchzuführen.

Im Normalfall können Übersetzeroptimierungen vergleichsweise leicht getestet werden, indem eine Anwendung einmal mit eingeschalteter Optimierung und einmal ohne übersetzt wird. Anschließend kann durch Laufzeitanalysen verglichen werden, wie sich die Optimierung auf die Laufzeit ausgewirkt hat. Dies ist für ein Multi-Task System jedoch nicht direkt anwendbar, da es sich um ein komplexeres System handelt, welches stark von dem Betriebssystem und dem dafür verwendeten Scheduling-Verfahren abhängt. Die Worst-Case Laufzeiten der Multi-Task Systeme stellen nur eine ungefähre Annäherung an die realen Laufzeiten dar, da es nicht möglich ist, ein ganzes Betriebssystem auf die Worst-Case Laufzeit zu analysieren. Für die gesamte Worst-Case Laufzeit der Systeme, wird im Folgenden für jedes Benchmark-Set nur die Summe der WCETs der einzelnen Tasks betrachtet.

Die Grenze von 4096 Bytes wurde gewählt, da einige vorangegangene Tests zeigten, dass eine größere Kapazität nur noch einen sehr geringen bzw. gar keinen weiteren Einfluss

mehr auf die Worst-Case Laufzeit hat. Dieses Verhalten wird durch Abbildung 6.12 und Abbildung 6.11 bestätigt.

Im folgendem Abschnitt werden die Ergebnisse, die während der Evaluation gewonnen wurden, vorgestellt. Dabei findet auch ein Vergleich der Qualität der Optimierung zwischen den Taskmengen statt. Ebenfalls wird auf die verschiedenen Allokationsformen (statisch, dynamisch und hybrid) eingegangen, sowie die unterschiedliche Auswirkung von DSPM und PSPM untersucht.

6.4 Benchmark Ergebnisse

Zu Beginn werden die Ergebnisse der Taskmenge „Set1“ vorgestellt. Abbildung 6.1 zeigt, wie stark sich die Worst-Case Laufzeit durch eine bestimmte Kapazität PSPM/DSPM, reduzieren lässt. Dabei gibt die X-Achse die zur Verfügung gestellte SPM-Kapazität an (DSPM- und PSPM-Kapazität gleich groß) und die Y-Achse stellt die prozentuale Abnahme der WCET gegenüber einer SPM-Kapazität von 0 Bytes dar.

Set1 ist mit 17 KB Codegröße und 19,8 KB Datengröße eine große Taskmenge. Es lässt sich jedoch erkennen, dass bereits 128 Bytes Scratchpad-Speicher, jeweils für Daten und Programmcode, eine Reduktion der WCET um bis zu 28% ermöglicht. Ebenfalls sieht man, dass das statische Verfahren bei kleinen SPM-Größen eine schlechtere Reduktion bringt als das dynamische bzw. hybride Verfahren. Verwendet man eine größere SPM-Kapazität, so nimmt auch für das statische Verfahren die Reduktion der WCET zu. Jedoch erfordert die statische Allokation 768 Bytes Scratchpad-Speicher, damit die gleiche Reduktion wie die dynamische und hybride Allokation bei 256 Bytes erreicht wird. Auffällig ist, dass eine weitere Erhöhung der SPM-Kapazität für die dynamische und hybride Allokation, ab einer SPM-Kapazität von 768 Bytes nur noch eine sehr geringe Auswirkung auf die WCET hat. Die Benchmark Ergebnisse aus [Ris11], für die gleiche Taskmenge, hatten eine Reduktion der WCET von ca. 30% . So lässt sich hier eine Verbesserung von ca. 20% feststellen.

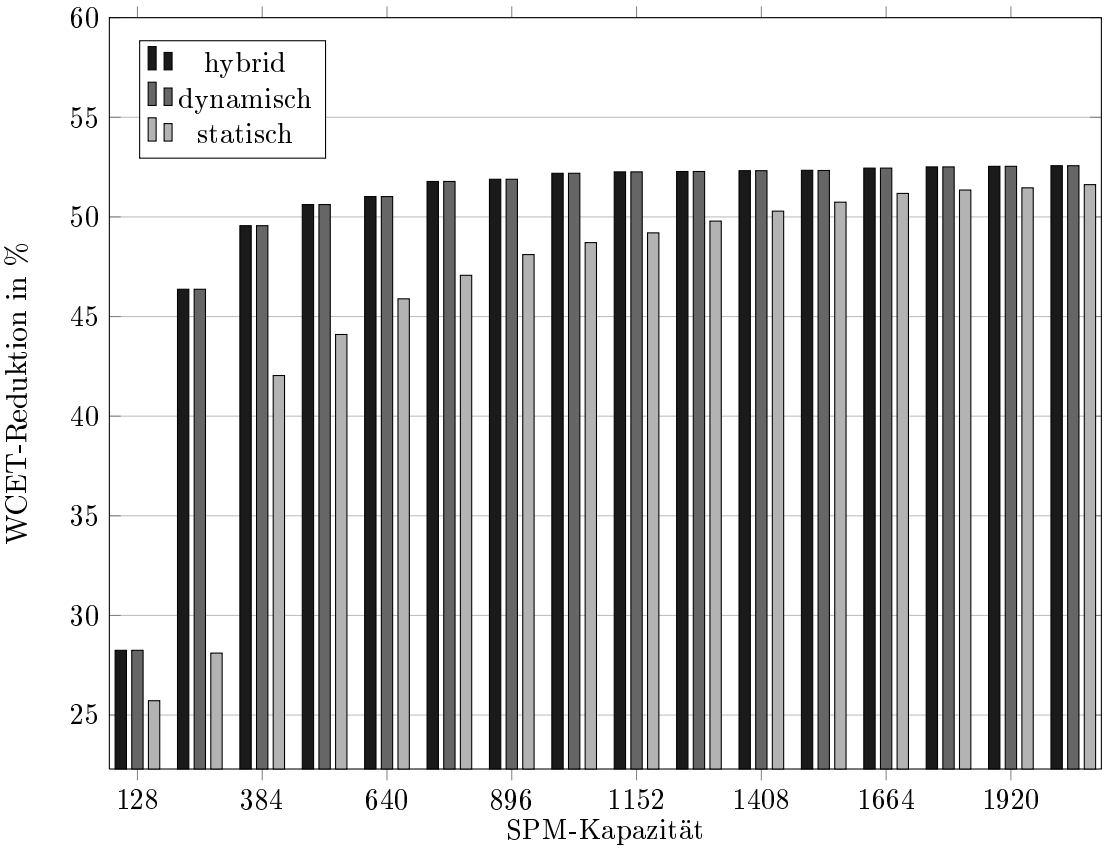


Abbildung 6.1: Reduktion der WCET des *Set1* durch Verwendung des DSPM und PSPM

6.4.1 Separate Bewertung von DSPM und PSPM

Im weiteren Verlauf werden die Benchmark Taskmengen Set2-Set5 getrennt nach DSPM und PSPM betrachtet, da sich die Auswirkung der Nutzung der beiden Scratchpad-Speicher stark unterscheidet.

Abbildung 6.2 zeigt für Set2 die Auswirkung des DSPM auf die Worst-Case Laufzeit. Es lässt sich erkennnen, dass nur eine sehr geringe Verbesserung erreicht wird, die weitgehend unabhängig von der Kapazität des DSPM ist. Der maximale Gewinn durch den DSPM liegt bei 3,16% ab 768 Bytes DSPM-Kapazität. Allerdings liegt der Gewinn für 128Bytes auch schon bei 3,15%. Auch für Set3 wird durch den DSPM nur eine sehr geringe WCET Reduktion von ca. 1,2% erreicht, wie man Abbildung 6.3 entnehmen kann.

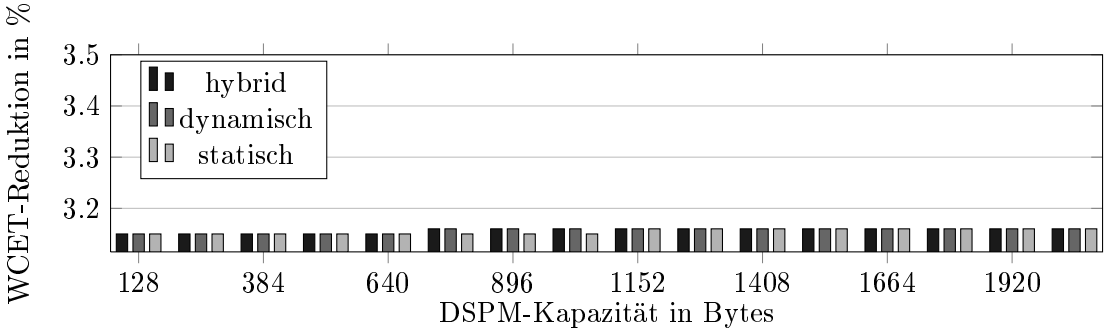


Abbildung 6.2: Reduktion der WCET des *Set2* durch Verwendung des DSPM

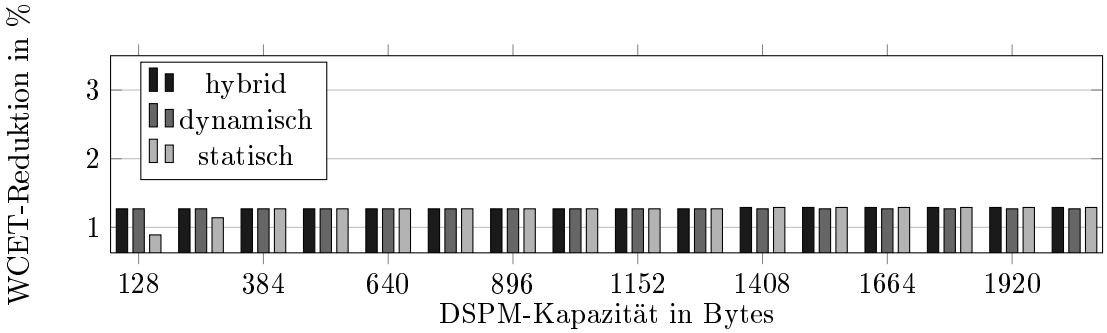


Abbildung 6.3: Reduktion der WCET des *Set3* durch Verwendung des DSPM

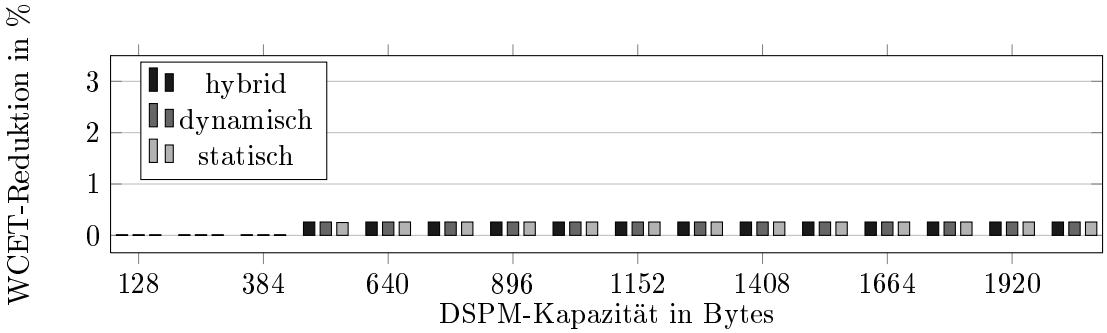


Abbildung 6.4: Reduktion der WCET des *Set4* durch Verwendung des DSPM

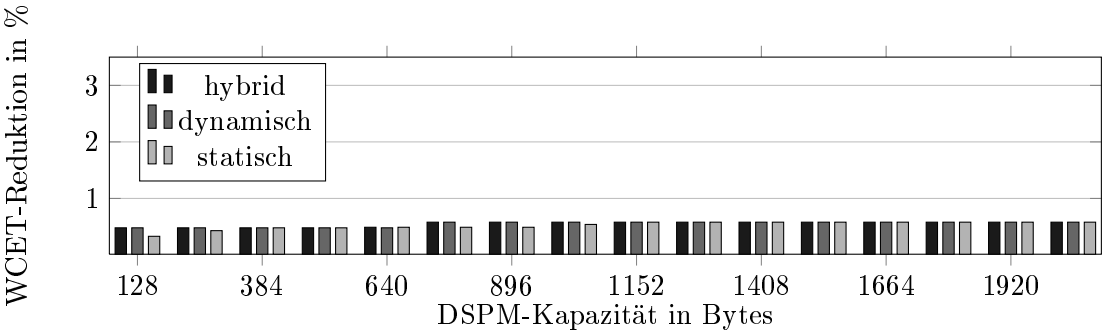


Abbildung 6.5: Reduktion der WCET des *Set5* durch Verwendung des DSPM

Für Set4 und Set5 wird durch eine Verteilung auf den DSPM sogar nur eine WCET Reduktion von $< 1\%$ erreicht (siehe Abbildung 6.4 und Abbildung 6.5).

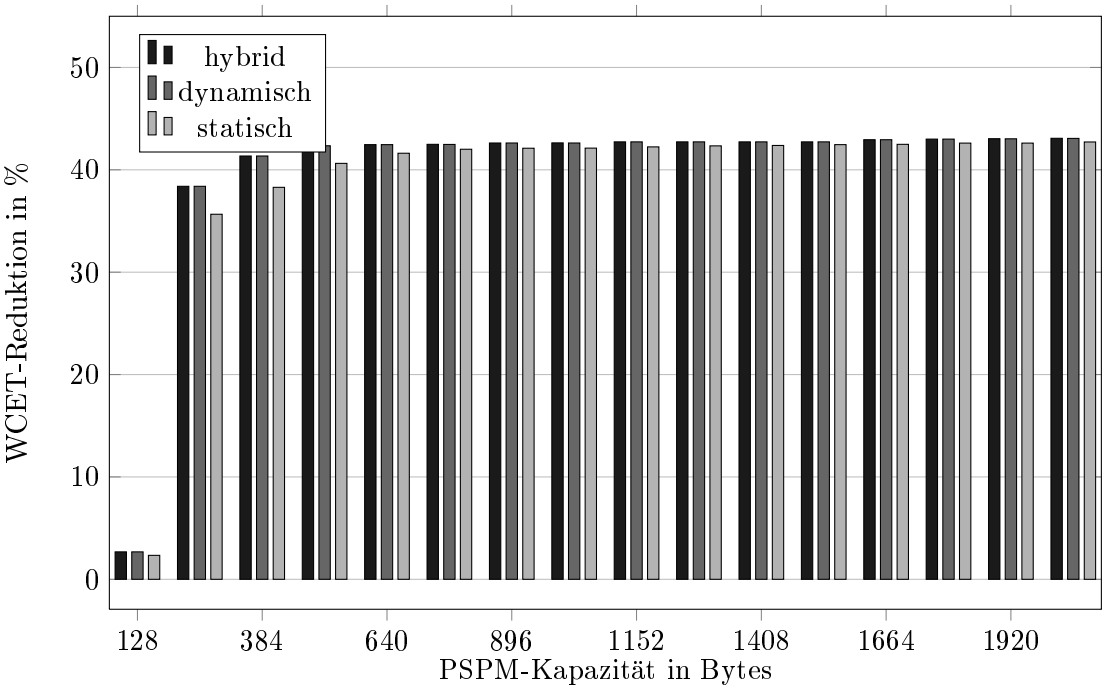


Abbildung 6.6: Reduktion der WCET des *Set2* durch Verwendung des PSPM

Die Auswirkung des PSPM für Set2 (siehe Abbildung 6.6) ist deutlich höher als die des DSPMs. Mit 256 Bytes PSPM ist bereits eine WCET Reduktion von fast 40% erreicht und bei 512 Bytes PSPM-Kapazität 42%.

Die Taskmenge Set3 hingegen benötigt eine größere PSPM-Kapazität als die Taskmengen Set1 und Set2 um eine vergleichbare Reduktion zu erreichen (siehe Abbildung 6.7). So wird erst mit 1024 Byte PSPM eine WCET Reduktion von 43% erreicht. Jedoch muss an dieser Stelle beachtet werden, dass 1024 Bytes nur ca. 10% der gesamten Code-Größe sind, so dass sich sagen lässt, dass die Reduktion von 43% durchaus nennenswert ist.

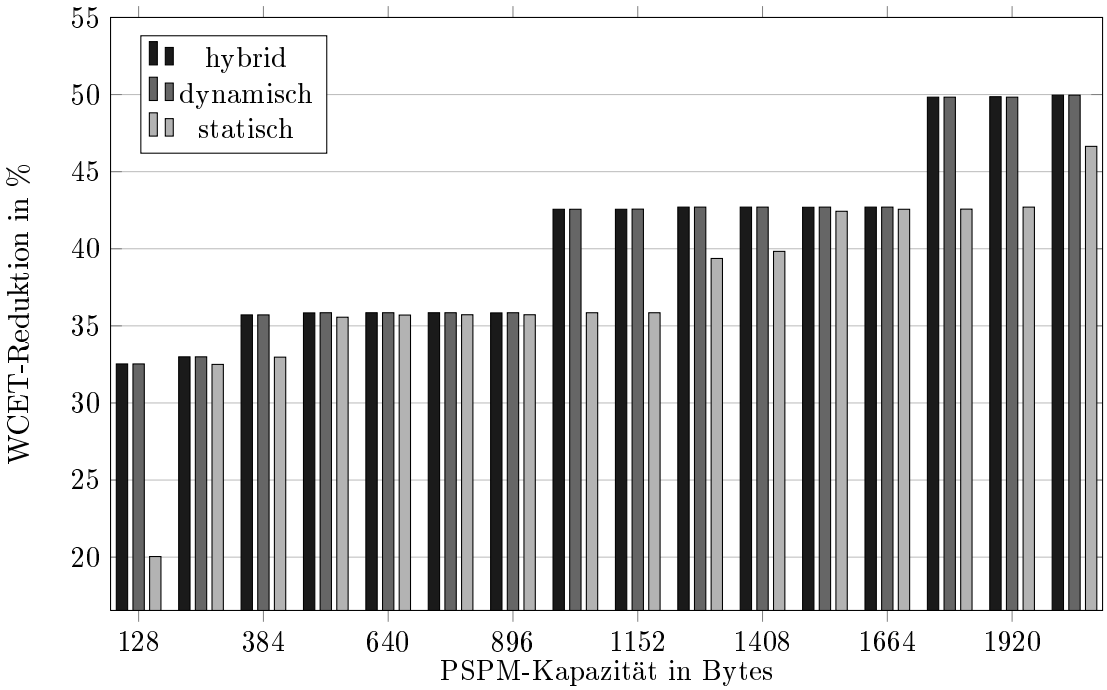


Abbildung 6.7: Reduktion der WCET des *Set3* durch Verwendung des PSPM

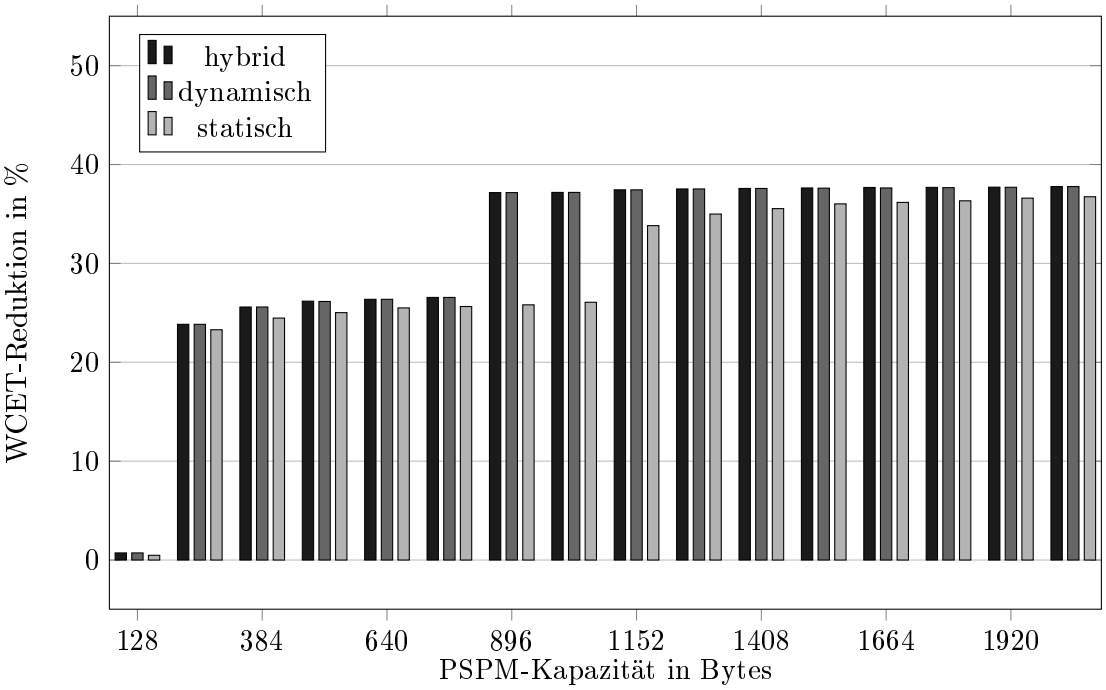


Abbildung 6.8: Reduktion der WCET des *Set4* durch Verwendung des PSPM

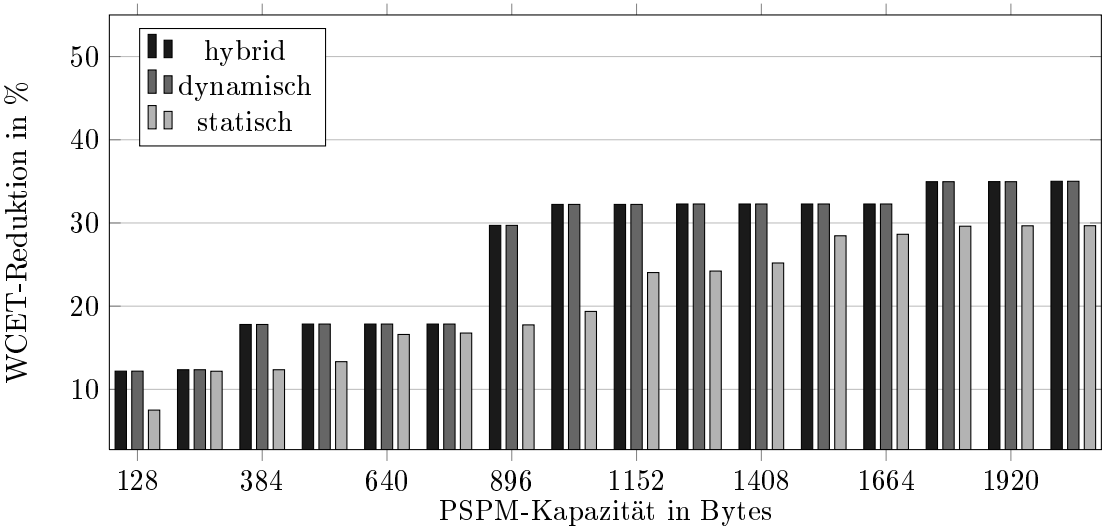


Abbildung 6.9: Reduktion der WCET des *Set5* durch Verwendung des PSPM

Durch die PSPM-Allokation kann für Set4 eine WCET Reduktion von maximal 37% erreicht werden und für Set5 maximal 35%. Die Taskmengen Set3 und Set5 verhalten

sich sehr ähnlich im Falle der PSPM-Allokation. Dies lässt sich durch die Ähnlichkeit der Taskmengen erklären, da Set3 eine Untermenge von Set5 ist.

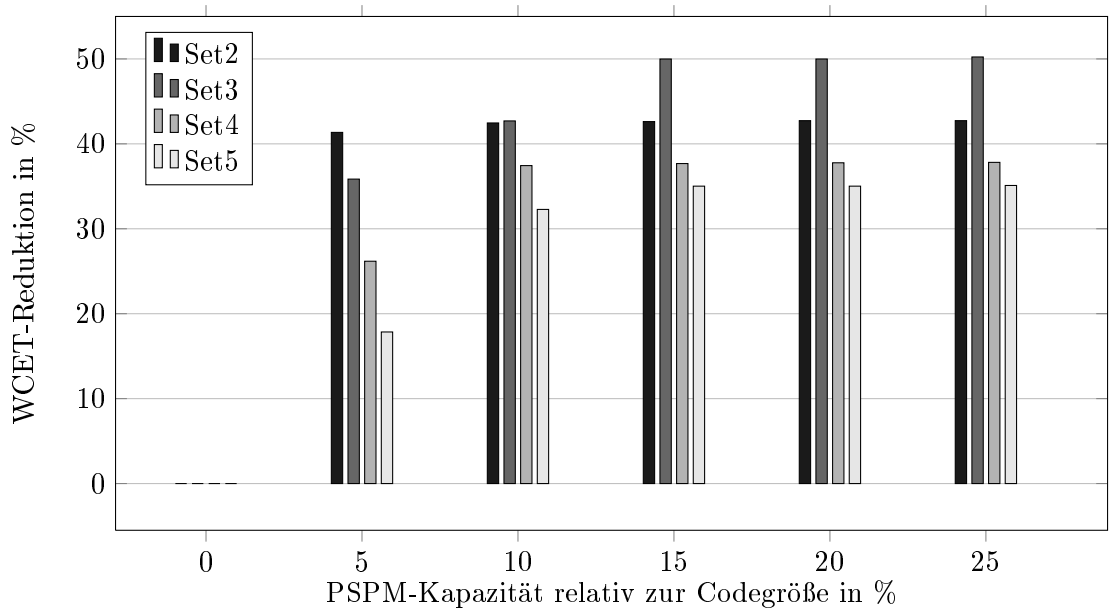


Abbildung 6.10: Reduktion der WCET von Set2-Set5 unter Angabe der relativen PSPM-Kapazität zur Codegröße unter Einsatz der hybriden Allokation

Abbildung 6.10 zeigt wie sich die WCET verhält, wenn einer Taskmenge eine relativ zur Codegröße große PSPM-Kapazität zugeordnet wird. Es lässt sich feststellen, dass bereits 10% PSPM-Kapazität eine Reduktion der WCET um 30% für alle Taskmengen erzielt. Für die Taskmengen Set2 und Set3 sogar schon eine Reduktion von 40%. Ebenfalls erkennbar ist, dass bereits ab 10% eine Sättigung für die Taskmengen Set2,Set4 und Set5 eintritt. Set3 hingegen erreicht bei einer PSPM-Kapazität von 15% noch eine Steigerung der Reduktion auf 50%.

Ein Grund für die unterschiedliche Reduktion, durch DSPM und PSPM, der WCET ist, dass sich die Basisblöcke eines Programmes deutlich leichter auf den PSPM verteilen lassen, da diese in der Regel nur einige Bytes groß sind. Dadurch ist es möglich, sehr fein die einzelnen Tasks auf den SPM zu verteilen. Die Datenblöcke des Programmes sind jedoch oft größere Arrays mit einer Größe von mehreren Kilobytes, was zur Folge hat, dass die Tasks nur sehr grob auf den DSPM aufgeteilt werden können.

Zur Überprüfung, ob die geringe Reduktion der WCET durch die DSPM-Allokation durch die geringen Kapazitäten des DSPM, von bis zu 4 KB, zu erklären ist, wurde die Taskmenge Set4 erneut optimiert bis zu einer DSPM-Kapazität von 28762 Bytes. Als Schrittweite für die *Gain-Computation* wurde 4096 Bytes gewählt, um die benötigte Rechenzeit auf ein Minimum zu halten. Dies hat den Nachteil, dass auch für den PSPM keine feine Einteilung mehr möglich ist und dadurch größere PSPM-Kapazitäten nötig sind, damit die gleiche Güte wie bei kleineren Schrittweiten erreicht wird.

Die DSPM-Allokation jedoch verhält sich sehr ähnlich wie bei einer geringen DSPM-Kapazität, was durch Abbildung 6.11 bestätigt wird. Es wird keine größere Verbesserung erzielt. Generell ist die Reduktion der WCET durch einen DSPM sehr gering. Der Task *fft_1024* besitzt vier Datenblöcke mit jeweils 4096 Bytes Größe. Werden alle vier Datenblöcke in den DSPM verschoben so erziehlt man lediglich eine Reduktion um 0,006%. Diese Ergebnisse decken sich jedoch mit denen aus [Rot08], da auch in dieser Arbeit festzustellen ist, dass die Reduktion der WCET durch einen DSPM stark abhängig von den gewählten Tasks ist.

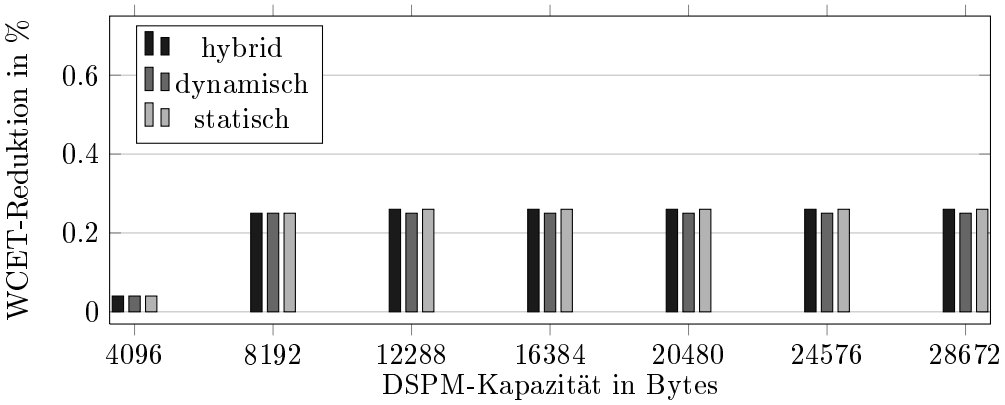


Abbildung 6.11: Reduktion der WCET des *Set4* durch Verwendung des DSPM (Schrittweite 4096 Bytes)

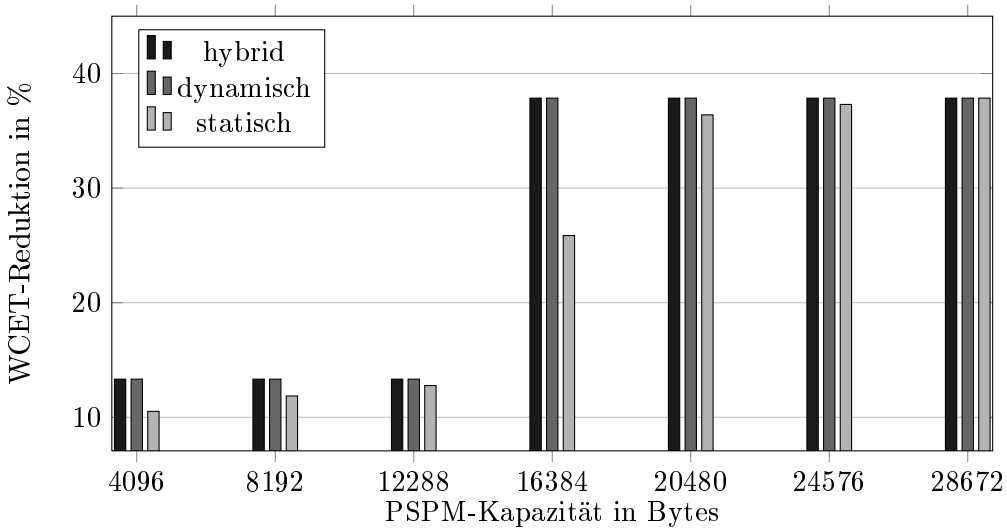


Abbildung 6.12: Reduktion der WCET des *Set4* durch Verwendung des PSPM (Schrittweite 4096 Bytes)

Für alle Benchmarks kann festgestellt werden, dass das statische Allokationsverfahren für kleine Speicherkapazitäten schlechter arbeitet als das dynamische und hybride Verfahren. Damit die statische Allokation eine vergleichbare Güte wie die dynamische und hybride Allokation erreicht, benötigt diese mehr Scratchpad-Speicher. Die Unterschiede zwischen dem dynamischen und dem hybriden Verfahren sind sehr gering, wobei in der Regel das hybride Verfahren eine geringfügige größere Verbesserung bringt.

Insgesamt lässt sich sagen, dass die Qualität der Optimierung sehr stark abhängig von den verwendeten Taskmengen ist. So lassen sich manche Tasks sehr fein auf den Scratchpad-Speicher verteilen und andere Tasks besitzen eine sehr grobe Struktur, was zur Folge hat, dass für eine Verbesserung der WCET eine große Kapazität SPM benötigt wird. Für die DSPM-Allokation ist es generell besser, wenn ein Programm nicht aus wenigen sehr großen Datenblöcken besteht. Es werden bessere Ergebnisse erzielt, wenn ein Programm viele kleine Datenblöcke, wie einzelne Variablen, besitzt. Große Datenstrukturen wie Arrays sind schlecht auf den Scratchpad-Speicher zu verteilen.

7 Fazit und Ausblick

Diese Arbeit beschäftigt sich mit der Überprüfung und Verbesserung der bestehenden Multi-Task Scratchpad-Speicherallokation von Kyrill Risto [Ris11], da sich herausgestellt hat, dass diese nicht fehlerfrei funktioniert. Dazu wurden zu erst die Grundlagen, die für die Optimierung notwendig sind, und anschließend die Arbeit von K. Risto vorgestellt. Zur Korrektur bestehender Fehler wurde im Folgendem eine Lösung vorgestellt, wozu die bestehenden ILP-Formulierungen und die Realisierung innerhalb des WCC-Übersetzers angepasst werden mussten. Wie diese Änderungen realisiert wurden, wird in Kapitel 5 beschrieben und welche weiteren Probleme aufgetreten sind. Dazu zählt unter anderem eine Problematik bezüglich der Sprunganpassung zwischen mehreren Speicherbereichen und dadurch entstandenen Abbrüchen des Übersetzungsvorganges. In Kapitel 6 wurde überprüft, wie stark sich die Multi-Task Scratchpad-Speicherallokation auf die Worst-Case Laufzeit auswirkt. Dabei wurde festgestellt, dass sich durch die Nutzung eines Programm-Scratchpad-Speichers die Worst-Case Laufzeit stark reduzieren ließ. Die Reduktion lag schon bei kleinen Speicherkapazitäten im Bereich von 30-40%. Allerdings hat sich auch gezeigt, dass die Auswirkung des DSPM sehr gering ausfiel. Es wurden nur Reduktionen der WCET von ca. 1-3% erreicht. Diese Ergebnisse decken sich jedoch mit anderen Arbeiten auf dem Gebiet der DSPM-Allokation. Es stellte sich heraus, dass die DSPM stark abhängig von den gewählten Tasks ist. Die PSPM-Allokation hingegen erreicht für alle Tasks dieser Arbeit mindestens eine Reduktion der WCET von ca. 30%, teilweise jedoch auch bis zu 50%.

7.1 Ausblick

Auf Grund der hohen Komplexität einer Multi-Task Optimierung lassen sich noch mehrere weitere Arbeiten auf dem Gebiet der Multi-Task Scratchpad-Speicherallokation vorstellen. Zum einen bietet die Daten-Scratchpad-Speicherallokation noch Potential für Verbesserungen, da sich in Kapitel 5 gezeigt hat, dass die Auswirkung der Ausnutzung eines DSPM Speichers nicht wie erwartet ist.

Eine andere Möglichkeit zur Verbesserung lässt sich in der Ausführungsgeschwindigkeit der *Gain-Computation* finden, denn diese ist trotz Optimierungen sehr rechenintensiv. Ein möglicher Ansatz zur Optimierung wäre, auf die Laufzeitanalysen zu verzichten und nur eine berechnete Laufzeit zu verwenden, die auch schon für das Lösen des ILPs verwendet wird. Dies würde zwar die Genauigkeit der Optimierung etwas senken aber die Laufzeit stark reduzieren. Eine erste Vorbereitung dafür wurde bereits innerhalb des WCC-Übersetzers umgesetzt. Die Gain-Computation kann bereits die Laufzeitinformationen für die Tasks mit Hilfe der berechneten Laufzeit erstellen, jedoch sind für die Speicherallokation weitere Änderungen notwendig, damit die bereitgestellten Informationen korrekt genutzt werden.

Außerdem wäre eine Implementierung der notwendigen Änderungen an einem Betriebssystem für die Multi-Task Scratchpad-Speicherallokation als Forschungsarbeit denkbar. Ein Betriebssystem müsste dazu in der Lage sein während eines Taskwechsels die verwendete Scratchpad-Speicherallokation zu erkennen und gegebenenfalls bei einer dynamischen Allokation die Speicherinhalte korrekt zu übertragen. Durch so eine Realisierung ließe sich die Optimierung praxisbezogener testen und es könnte eine Aussage über die Einsatzfähigkeit in einem realem Hardware- bzw. Softwareprojekt getroffen werden.

Literaturverzeichnis

[AIT11] *aiT Worst-Case Execution Time Analyzers*. `http://www.absint.com/ait/`. Version: August 2011

[CPL11] *IBM ILOG CPLEX Optimizer*. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`. Version: August 2011

[ERI11] *Erika Enterprise and RT-Druid*. `http://erika.tuxfamily.org/`. Version: August 2011

[FL10] Falk, Heiko ; Lokuciejewski, Paul: A compiler framework for the reduction of worst-case execution times. In: *Journal on Real-Time Systems* 46 (2010), oct, Nr. 2, S. 251–300. – DOI 10.1007/s11241-010-9101-x

[Inf08] Infineon Technologies: *TC1796 - 32-Bit Single-Chip Microcontroller, Data Sheet*, April 2008

[Inf09] Infineon Technologies: *TC1797 - 32-Bit Single-Chip Microcontroller, Data Sheet*, April 2009

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

[Kot11] Kotthaus, Helena: *Cache-bewusste Code-Positionierung zur Reduktion der maximalen Programmlaufzeit (WCET)*, Technische Universtität Dortmund, Diplomarbeit, January 2011

[Kru10] Krumme, Lutz: *Dynamische Scratchpad-Allokation von Code und Daten zur WCET-Minimierung*, Technische Universtität Dortmund, Diplomarbeit, August 2010

[LPMS97] Lee, Chunho ; Potkonjak, Miodrag ; Mangione-Smith, William H.: MediaBench: a tool for evaluating and synthesizing multimedia and

communicatons systems. In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 1997 (MICRO 30). – ISBN 0–8186–7977–8, 330–335

[LPS11] *lp_solve Homepage*. `http://lpsolve.sourceforge.net/`. Version: August 2011

[Mar07] Marwedel, Peter: *Eingebettete Systeme*. 1. Aufl. 2007. Korr. Nachdruck. Springer Berlin Heidelberg, 2007. – ISBN 9783540340485

[MMsH01] Memik, Gokhan ; Mangione-smith, William H. ; Hu, Wendong: NetBench: A benchmarking Suite for Network Processors. In: *In ICCAD*, 2001, S. 39–42

[Moe11] Moellmer, Jens: *WCET Optimierung unter Beachtung der Speicherhierarchie*. August 2011

[MRT11] *Mälardalen Real-Time Research Centre Benchmarks*. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. Version: August 2011

[Ris11] Risto, Kyrill: *Scratchpad-Allokation zur Reduktion der gröstmöglichen Laufzeit von Multitask-Systemen*, Technische Universtität Dortmund, Diplomarbeit, January 2011

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

[Sch10] Schmitz, Norman: *ILP-basierte Registerallokation zur Worst-Case Execution Time Minimierung*, Technische Universtität Dortmund, Diplomarbeit, June 2010

[STA11] *Rational Statemate*. `http://www-01.ibm.com/software/awdtools/statemate/`. Version: August 2011

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

[WCC11] *The WCET-aware C compiler WCC*. `http://ls12-www.cs.tu-dortmund.de/daes/de/forschung/wcet-aware-compilation/wcc-compiler-infrastructure.html`. Version: August 2011

Abbildungsverzeichnis

2.1 WCET im Vergleich zur ACET und BCET 8
2.2 Aufbau des WCC-Übersetzers nach [FL10] 10
2.3 Problematik bei der Optimierung des WCEPs 12

4.1 Hybride Aufteilung des Scratchpad-Speichers 21
4.2 Schematische statische Allokation 26
4.3 Schematische dynamische Allokation 27
4.4 Schematische hybride Allokation 30

6.1 Reduktion der WCET des *Set1* durch Verwendung des DSPM und PSPM 44
6.2 Reduktion der WCET des *Set2* durch Verwendung des DSPM 45
6.3 Reduktion der WCET des *Set3* durch Verwendung des DSPM 45
6.4 Reduktion der WCET des *Set4* durch Verwendung des DSPM 45
6.5 Reduktion der WCET des *Set5* durch Verwendung des DSPM 46
6.6 Reduktion der WCET des *Set2* durch Verwendung des PSPM 46
6.7 Reduktion der WCET des *Set3* durch Verwendung des PSPM 47
6.8 Reduktion der WCET des *Set4* durch Verwendung des PSPM 48
6.9 Reduktion der WCET des *Set5* durch Verwendung des PSPM 48
6.10 Reduktion der WCET von Set2-Set5 unter Angabe der relativen PSPM-Kapazität zur Codegröße unter Einsatz der hybriden Allokation 49
6.11 Reduktion der WCET des *Set4* durch Verwendung des DSPM (Schrittweite 4096 Bytes) 50
6.12 Reduktion der WCET des *Set4* durch Verwendung des PSPM (Schrittweite 4096 Bytes) 51