**UNIVERSITÄT DORTMUND**
FACHBEREICH INFORMATIK

Diplomarbeit

# Einfluss von statischem Cache-Locking auf Worst-Case Execution Times

Sascha Plazar

05. Januar 2007

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik XII
(Technische Informatik und Eingebettete Systeme)
Fachbereich Informatik
Universität Dortmund

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

GERMANY · D-44221 DORTMUND

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Die Geschwindigkeit von Prozessoren hat sich im Vergleich zur Geschwindigkeit der Speichertechnik in den letzten Jahren rasant weiterentwickelt. Caches sind in Computersystemen inzwischen sehr beliebt, um die immer größer werdende Schere zwischen der Geschwindigkeit des Prozessors und der des vergleichsweise langsamen Speichers auszugleichen. Für die durchschnittliche Ausführungszeit führt der Einsatz von Daten- und Instruktionscaches nachweislich zu kürzeren Ausführungszeiten vor allem bei hoher Code-Lokalität. Ein weiterer Grund für den breiten Einsatz von Caches ist die völlige Transparenz und die Tatsache, dass keine Anpassungen des Programmcodes für einen Einsatz nötig sind.

In Realzeit-Systemen mit harten Zeitschranken ist der Einsatz von Caches bisher eher hinderlich gewesen, da die Geschwindigkeit durch zwangsläufig auftretende Cache-Hits und -Misses nur schwierig und in Multitasking-Systemen z.T. nicht vorhersagbar gewesen ist. Um trotzdem Angaben für obere Schranken für die Ausführungszeit machen zu können, sind bisher Systeme ohne Caches eingesetzt worden oder bei der Analyse der Worst-Case Execution Time von Programmen eventuelle Gewinne durch vorhandene Caches nicht betrachtet worden. Daher werden vermehrt kleine schnelle on-Chip Speicher eingesetzt (engl. *Scratchpad*), die in den Prozessor integriert sind und meist mit vollem Prozessortakt bei einer Zugriffsverzögerung von nur einem Takt betrieben werden. Der Inhalt des Speichers ist in der Vergangenheit meist statisch gewesen – ist also beim Entwurf des Systems durch den Entwickler oder den Compiler bestimmt worden. Deshalb können sichere Vorhersagen über die Zugriffgeschwindigkeit für Inhalte des Scratchpads und daher auch über die WCET eines Programms getroffen werden.

Neuere Cache-Architekturen bieten die Möglichkeit, Instruktionen oder Daten in den Cache zu locken, also vor Verdrängung zu schützen, und so dauerhaft vorzuhalten. Dadurch ist es möglich, die Zugriffsgeschwindigkeit auf Befehle und Daten, die in den Cache gelockt werden, sicher vorherzusagen und somit verbindliche Aussagen über die Worst-Case-Laufzeit von

Programmen zu treffen. Durch die Mechanismen zum Lockdown von Inhalten lassen sich die Vorteile des Scratchpads bei der Vorhersagbarkeit mit denen von Caches bzgl. Transparenz und Code-Kompatibilität kombinieren.

Bisher stehen die Forschungen im Bereich von Caches in Realzeit-Systemen noch am Anfang und es gibt wenig Ergebnisse und Erfahrungen, wie sich der Einsatz von gelockten Caches auf die WCET eines Programms auswirkt. Außerdem stellt sich zudem die Frage, welche Teile des Programms durch ein Locking in den Cache besonders große Einsparungen in der Laufzeit ermöglichen. Eine möglichst optimale Auswahl von Elementen zum Locking ist schwierig und je nach verwendetem Verfahren auch zeitaufwändig, da die Berechnung der WCET einer Binärdatei wesentlich aufwändiger ist als die Messung oder Berechnung der ACET.

## 1.1 Related Work

Bisher gibt es wenig Forschungsprojekte, die sich mit der Optimierung der Worst-Case-Laufzeit von Programmen beschäftigen, und noch weniger, die den Einfluss von Caches auf die Worst-Case Execution Time untersuchen. Deshalb sollen an dieser Stelle verwandte Arbeiten vorgestellt werden, die direkt oder indirekt die Optimierung der WCET eines Programmes durch unterschiedliche Techniken erreichen.

Erste Ansätze und Analysen, die gelockte D-Caches betrachten, werden in [VLX03a] vorgestellt. Lisper u.a. bestimmen die Datenobjekte, bei denen eine statische Cache-Analyse fehlschlägt, und sichern eine hohe Worst-Case Speichergeschwindigkeit, indem diese Objekte in den D-Cache gelockt werden. Außerdem wird in [VLX03b] der Einfluss von gelockten D-Caches und Cache Partitoning auf die Vorhersagbarkeit von Multitasking-Systemen untersucht. Zum einen wird dort der Cache mit den Daten geladen, die wahrscheinlich als nächstes benutzt werden, und zum Anderen wird versucht, mit Compiler Optimierungen wie Loop-Tiling und -Padding ein Verdrängen von Inhalten des Caches zu minimieren.

Die Forschergruppe um Chiou beschäftigt sich mit Column-Caches, bei denen Teile des Caches dynamisch bestimmten Speicherbereichen zugewiesen werden [CRD00]. So kann durch Software sichergestellt werden, dass sich Programme oder auch Teile davon nicht gegenseitig aus dem Cache verdrängen.

Campoy u.a. untersuchen wie diese Arbeit statisches Cache-Locking und die Auswirkungen auf die WCET von Tasks sowie die Worst-Case Auslastung des Prozessors. Dafür sind in [CPIM05] zwei Algorithmen zur Selektion des Cache-Inhaltes entwickelt worden.

Puaut und Decotigny betreiben in [PD02] ähnliche Forschungen, indem sie zwei Algorithmen entwickeln, die in Realzeitsystemen mit mehreren Tasks

zum einen die Auslastung des Systems und zum anderen die Interferenz der Tasks untereinander minimieren sollen.

Am Informatik-Lehrstuhl 12 der Universität Dortmund werden seit 2001 Compiler und Techniken entwickelt, die Optimierungen durch den Einsatz von Scratchpad Speichern [Mar06] ermöglichen. Verschiedene Arbeiten untersuchen die möglichen Einsparungen an Energie, bei der Average-Case Execution Time und bei der Worst-Case Execution Time. In [WM05] wird der Einfluss von Scratchpads auf die WCET Vorhersage betrachtet, und als Auswahlstrategie für Elemente, die in das Scratchpad verschoben werden, deren Energieverbrauch zugrunde gelegt. Außerdem ist ein Algorithmus entwickelt worden, der Teile eines Programms, die besonders häufig aus dem Cache verdrängt werden, in das Scratchpad verschiebt [VWM04].

## 1.2 Ziel der Diplomarbeit

In dieser Arbeit soll der Einfluss von statischem Cache-Locking auf die Worst-Case Execution Time von Programmen untersucht werden. Im Folgenden wird von statischem Locking gesprochen, wenn der Inhalt des Caches zur Entwurfszeit des Systems bzw. der Programme bestimmt wird. Dieser Inhalt wird dann beim Systemstart oder vor der eigentlichen Programmfunktionalität in den Cache geladen und gelockt und verbleibt dort bis das System abgeschaltet bzw. das Programm beendet wird. Dazu sollen folgende Punkte umgesetzt werden:

- **Erzeugung von Lockdown-Code**
  Damit überhaupt Instruktionen in den Cache gelockt werden können, muss zuerst eine Referenzimplementierung erzeugt werden, die es ermöglicht, manuell ausgewählte Programmteile zu locken.

- **Aufbau einer Toolchain**
  Es soll ein Softwarewerkzeug erstellt werden, das es ermöglicht, halbautomatisch ein gegebenes C-Programm zu übersetzen und bzgl. der Worst-Case Execution Time durch Locking einzelner Funktionen zu optimieren.

- **Entwicklung von Selektionsalgorithmen**
  Für diese Toolchain sollen Algorithmen entwickelt werden, die eine möglichst optimale Auswahl von Elementen zur Entwurfszeit eines Systems ermöglichen. Diese Auswahl von Funktionen soll durch ein statisches Lockdown in den I-Cache die WCET des zu optimierenden Programms möglichst weit senken.

- **Verifizierung der Ergebnisse**
  Alle implementierten Algorithmen müssen an Benchmarks getestet

werden, um zu überprüfen, welche Einsparungen in der Worst-Case-Laufzeit eines Programms durch Locking einzelner Teile in den Cache möglich sind. Dies soll einen Vergleich der Algorithmen untereinander und mit bisherigen Techniken ermöglichen.

## 1.3 Aufbau der Diplomarbeit

An dieser Stelle soll auf den Aufbau dieser Arbeit eingegangen und kurz erläutert werden, was in den einzelnen Kapiteln behandelt wird.

Kapitel 2 befasst sich mit den Grundlagen für diese Arbeit und dem statischen Lockdown von Instruktionen in den I-Cache. Dazu wird die untersuchte Hardware und die Arbeitsweise der Caches sowie die Techniken des Cache-Locking im Detail erläutert. Außerdem wird der benutzte Compiler, die Software für die ACET-Messungen und das Werkzeug für die WCET-Analyse vorgestellt.

In Kapitel 3 wird der typische Workflow für die Übersetzung und anschließende Optimierung eines C-Programms mit Hilfe der entwickelten Toolchain dargestellt. Zuerst werden die Voraussetzungen der einzelnen Softwarekomponenten für eine Integration in die Toolchain erläutert und anschließend der eigentliche Übersetzungsvorgang vorgestellt. Daraufhin werden die Techniken der Analyse des übersetzten Programms für eine Auswahl der zu lockenden Funktionen besprochen. Zum Schluss wird der implementierte Startup-Code besprochen, der die Routinen zum Locking von Funktionen implementiert.

Die eigentlichen Optimierungsalgorithmen, die für diese Arbeit entwickelt worden sind, werden in Kapitel 4 vorgestellt. Es werden zunächst die Grundlagen für Optimierungsprobleme allgemein und die Besonderheiten der WCET-Optimierung besprochen, bevor die drei entwickelten Selektionsalgorithmen für die Auswahl der in den Cache zu lockenden Funktionen vorgestellt werden.

Kapitel 5 präsentiert die Ergebnisse der erzielten Optimierungen und verwendet dazu fünf Benchmarks, an denen die Algorithmen getestet werden. Außerdem wird ein Vergleich zwischen bestehenden und den hier eingeführten Auswahlverfahren für die Menge der zu lockenden Elemente angestellt. Abschließend wird noch die benötigte Rechenzeit für eine Optimierung auf dem Entwicklungssystem ermittelt.

Das letzte Kapitel 6 fasst die erreichten Ziele und Einsparungen der entwickelten Techniken noch einmal zusammen und gibt einen Ausblick auf mögliche Weiterentwicklungen und wünschenswerte Verbesserungen.

# Kapitel 2

# Grundlagen

Bei hardwarenahen Optimierungen wie dem Lockdown von Cache-Inhalten ist es unumgänglich, sich mit der Zielplattform, also der verwendeten Hardware wie Prozessor-, Speicher- und Cache-Architektur, vertraut zu machen. Kapitel 2.1 geht deshalb näher auf die untersuchte ARM-Familie und deren Besonderheiten ein.

Es ist ebenso wichtig, den eingesetzten Compiler genau auszuwählen, um optimale Ergebnisse im Hinblick auf hardwarespezifischen Code und die Verwendbarkeit für die hier vorgestellten Optimierungsansätze sicherzustellen. Auf die Besonderheiten und Voraussetzungen bezüglich der verwendeten Werkzeuge geht Kapitel 2.2 ein.

## 2.1 Zielplattform

Von den existierenden Systemen mit einer Cache-Architektur unterstützen momentan nur einige ausgewählte, vornehmlich im Embedded-Bereich anzutreffende Systeme, überhaupt die Möglichkeit des Lockdowns von Cache-Inhalten. Neben einigen Mitgliedern der PowerPC-Familie sind dies hauptsächlich ARM-Abkömmlinge der ARMv4T Generation. Die ARM-Prozessoren, die das Cache-Locking unterstützen, verwenden verschiedene Arten, um ein Locking von Daten oder Instruktionen zu ermöglichen:

- **Way-Adressing**
  Spezielle Register ermöglichen inkrementelles Locking mit einer Granularität von einem Way. Diese Technik unterstützten die Prozessoren ARM920T, ARM922T und ARM940T.
- **Lock-Bits**
  Ein Bit pro Way steuert das Locking für jedes Way unabhängig voneinander bei den Prozessoren ARM926EJS sowie ARM1026EJS.
- **Special Allocate Commands**
  Befehle steuern das Lockdown des Caches und ermöglichen es u.a.,

den D-Cache als direkt adressierbaren zusätzlichen On-Chip-RAM zu konfigurieren. Der Intel™ XScale unterstützt solche Instruktionen mit seinem Befehlssatz.

Um zu verstehen, wie das eigentliche Cache-Locking funktioniert, wird im nächsten Abschnitt zunächst die verwendete Architektur und anschließend das damit verbundene Speicher-, sowie Cache-Interface vorgestellt. In Kapitel 2.1.3 wird gezeigt, wie ein derartiges Lockdown von Cache-Inhalten durch Software gesteuert wird.

### 2.1.1 Untersuchte Prozessor-Architektur

Im diesem Kapitel werden der Prozessor ARM920T und seine Eigenschaften vorgestellt. Alle Algorithmen, Benchmarks und Optimierungen die vorgestellt werden, stützen sich auf diesen Prozessor mit der gezeigten Architektur. Der Prozessor besitzt folgende Eigenschaften, die für diese Diplomarbeit wichtig sind:

- 32-Bit RISC Von-Neumann-Architektur
- Unterstützt 32-Bit ARM- und 16-Bit Thumb-Instruktionen
- 8-, 16- oder 32-Bit DRAM-Interface
- 16 KB Daten-Cache
- 16 KB Befehls-Cache
- Virtuelle 64-fach assoziative Caches
- Jeder Cache ist aufgeteilt in 8 Segmente
- Eine Cachezeile fasst 32 Bytes
- Replacement-Policy: Round-Robin oder pseudo Random
- Lockdown durch Way-Adressing

### 2.1.2 Verwendete Speicher-/Cache-Architektur

Wie bereits weiter oben erwähnt, besitzt der ARM920T eine Von-Neumann-Architektur und die damit verbundenen Vor- und Nachteile. Durch das gemeinsame DRAM-Speicherinterface für Daten und Instruktionen müssen für viele Befehle zwei oder mehr Leseanforderungen über den Speicherbus übertragen werden. Der erste Lesezugriff holt den eigentlichen Befehl aus dem Speicher, die darauf folgenden Zugriffe holen Operanden, die aufgrund ihrer Größe nicht mit ins Befehlswort kodiert werden können.

Für Analysen, Messungen und Benchmarks wird eine Konfiguration mit 33 MHz Speichertakt und 32-Bit breitem Zugriff mit einer Latenz von 4 Wartezyklen für einen Lese- oder Schreibvorgang angenommen.

Die Nachteile der Speicherarchitektur und die immer größer werdende Differenz zwischen Speicher- und Prozessorleistung versucht man durch den Einsatz von Caches auszugleichen. Die Caches sind im Gegensatz zum Speicher als Harvard-Architektur ausgelegt - also getrennte Caches für Daten und Befehle. Jeder dieser virtuellen Caches fasst 16 KB und ist 64-fach assoziativ ausgelegt (siehe Kapitel 2.1.1), das heißt, es können 64 Speicherzeilen mit konkurrierenden Index-Bits gespeichert werden, ohne dass ein Zeile verdrängt wird. Die Abbildung 2.1 verdeutlicht die von der normalen Literatur abweichende Darstellung der ARM-Caches, die in [Adv02] verwendet wird: Cache-Zeilen sind übereinander angeordnet und von 0 bis 63 nummeriert. Die 8 Segmente sind als Flächen mit je 64 Cache-Zeilen hintereinander angeordnet. Ein Way entspricht einer Cache-Zeile über alle acht Segmente, in die ein zusammenhängender Speicherbereich von 256 Bytes gelockt werden kann.

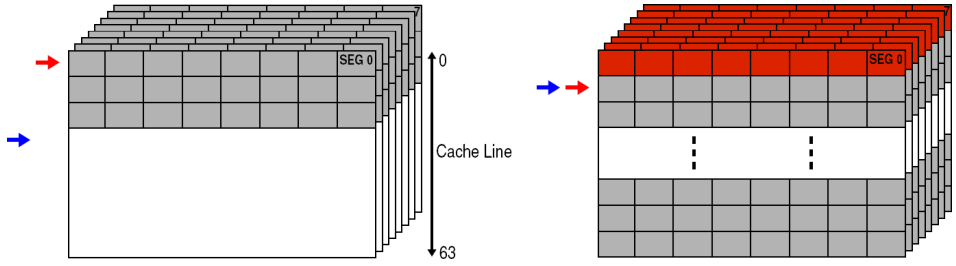



**Abbildung 2.1:** Links: ungelockter Cache mit drei gefüllten Zeilen. Rechts: eine Zeile gelockt.

### 2.1.3 Cache-Locking

Nachdem die Eigenschaften der verwendeten Hardware bekannt sind, wird nun gezeigt, wie ein Lockdown von Instruktionen durchgeführt wird und welche Voraussetzungen dafür erfüllt sein müssen.

Damit überhaupt vorhersehbar ist, welche Zeile des Caches als nächste ersetzt wird, muss die Replacement-Policy auf Round-Robin über das Coprozessor-Register `CP15` gesetzt werden. Es muss sichergestellt werden, dass die Lockdown-Befehle selbst in einem non-cached Speicherbereich liegen, damit nicht beim Ausführen der Lockdown-Code selbst in den Cache geladen und anstelle der zu beschleunigenden Instruktionen gelockt würde. Wie die Seitentabelle manipuliert wird und der Cache sowie die richtige Replacement-Policy eingeschaltet werden, zeigt [Adv98d]. Außerdem dürfen zu lockende Instruktionen nicht bereits im Cache vorhanden sein, weil diese nicht erneut in den Cache geladen werden würden. Sollen Elemente im

D-Cache gelockt werden, so ist es sinnvoll, den Write-Back-Modus einzuschalten, damit Schreiboperationen ebenfalls vom gelockten Cache profitieren.

Da Daten bzw. Instruktionen vor ihrer ersten Verwendung erst einmal in den Cache geladen werden müssen, entsteht zusätzlicher Overhead in Form von Prozessor- und Speicherzyklen. Das Locking einer Cache-Zeile mit 32 Bytes benötigt 47 Prozessorzyklen, was die Kostenfunktion $C$ in Prozessorzyklen, abhängig von der Größe $S$ in Bytes ergibt:

$$C = \frac{\lfloor S \rfloor^{32}}{32} * 47$$

Der Ausdruck $\lfloor S \rfloor^{32}$ bedeutet, dass die Angabe in Bytes auf das nächste Vielfache von 32 aufgerundet werden soll, da immer nur ganze Zeilen mit 32 Bytes in den Cache gelockt werden können.

Für den Fall, dass ein gelockter Cache geflusht werden sollte, sind die darin enthaltenen Daten verloren und der von gelocktem Code oder gelockten Daten belegte Platz zunächst einmal verloren. Durch ein Zurücksetzen der Register auf normalen Cache-Betrieb steht der Platz dann erneut zur Verfügung.

Ein typischer Lockdown-Vorgang läuft folgendermaßen ab:

1. Interrupts abschalten, damit der Lockdown-Vorgang nicht unterbrochen wird und dadurch Interruptroutinen in den Cache geladen würden
2. Ggf. Cache einschalten
3. Round-Robin Replacement-Policy einschalten
4. Cache flushen
5. Daten- bzw. Instruktions-Block laden
6. Neuen Eintrag locken
7. Zurück zu 5. bis keine weiteren Daten/Instruktionen mehr zu locken sind oder Cache komplett gelockt ist
8. Interrupts wieder einschalten

Zwei Register steuern die Cache-Operationen, mit deren Hilfe Daten bzw. Instruktionen im Cache gelockt werden können. Das erste heißt Victim Base Pointer Register und gibt die Cache-Zeile für alle Segmente an, ab der überhaupt Zeilen verdrängt werden dürfen. Das zweite Register enthält die Victim Pointer Adresse, die angibt, welche Zeile als nächstes verdrängt wird.

Nach der Initialisierung des Caches zeigen beide Register auf die erste Cache-Zeile. Das bedeutet, dass alle Cache Zeilen für normale Cache-Operationen zur Verfügung stehen und als nächstes die erste Cache-Zeile verdrängt wird. Wird eine Zeile mit neuem Inhalt gefüllt, dann wird die Adresse des Victim Pointers um eins erhöht. Der linke Teil von Abbildung 2.1 zeigt den Cache bereits mit einigen gefüllten Zeilen und den Victim Pointer, der auf die nächste Zeile mit bisher ungültigem Inhalt zeigt (blauer Pfeil deutet die Pointer-Adresse an). Der Victim Base Pointer zeigt noch immer auf die erste Cache Zeile, deshalb steht der gesamte Cache für den normalen Betrieb zur Verfügung (roter Pfeil deutet die Pointer-Adresse an).

Soll jetzt ein Way mit Daten gefüllt und gelockt werden, so wird der Victim Pointer auf die Adresse des Victim Base Pointer gesetzt und ein Bereich von 256 Bytes an Instruktionen über Coprozessor-Befehle in das selektierte Way geladen. Sollen Daten in den Cache geladen werden, benutzt man LOAD-Operationen, da die MMU den Cache automatisch mit nicht vorhandenen Blöcken füllt. Um das Way abschließend gegen Verdrängung zu schützen, wird zum Schluss die Victim Base Pointer und Victim Pointer Adresse um ein Way inkrementiert. Der rechte Teil von Abbildung 2.1 zeigt den Cache mit einem gelockten Way, dementsprechend positionierten Victim Base Pointer und dem restlichen gefüllten Cache.

## 2.2 Werkzeuge

Die ARM-Prozessorarchitektur ist inzwischen weit verbreitet, und so gibt es für einen Entwickler eine entsprechende Auswahl an Softwarewerkzeugen. Viele Werkzeuge sind auf spezielle Einsatzzwecke wie energiesparenden oder besonders schnellen Code zugeschnitten und lassen sich daher nicht gleich gut für die gewünschten Optimierungen und Analysen einsetzen.

Bei den Compilern hat man schon bei der GNU Compiler Collection (GCC) die Wahl zwischen der Standard Version [GCC] und einer auf Realzeitsysteme spezialisierten Version [OAR04]. Auf der kommerziellen Seite sind u.a. der Imagecraft ICCARM, der Keil CARM sowie die ARM-Compiler aus dem ARM Software-Development-Toolkit [Adv98a] zu nennen, die sich in den gebotenen Features und erzeugtem Code teilweise deutlich unterscheiden. Ergänzend sind noch Ergebnisse aus der Forschung wie der ENCC des Lehrstuhls Informatik 12 der Universität Dortmund zu nennen.

Um Vorhersagen über das Laufzeit- und Energieverhalten von erzeugtem Code auf einem Eingebettetem System zu machen, benutzt man vermehrt Softwareprodukte, die eine Analyse oder Simulation ohne Vorhandensein des Zielsystems ermöglichen. So können Entwickler von Systemen bereits beim Entwurf prüfen, ob ihr Produkt die gegebenen harten Zeitschranken einhält und überhaupt auf der Zielplattform lauffähig ist, oder vor dem Einsatz mögliche Auswirkungen auf den Energieverbrauch vorhersagen. Auch

in dieser Kategorie von Software gibt es eine große Auswahl an unterschiedlichen Produkten für die Analyse von Average-Case Execution Times sowie Worst-Case Execution Times, von denen an dieser Stelle nur einige erwähnt werden.

In der Kategorie Average-Case Execution Time- und Energieanalyse finden sich Produkte wie der ARM Sourcelevel Debugger (siehe [Adv98a] Kapitel 7) oder MPARM [BBB+05, MPA], die Informationen über das analysierte Programm durch Simulationsdurchläufe sammeln. Der Sourcelevel Debugger bietet im Gegensatz zu MPARM kein Energiemodell und eignet sich deshalb nur zur Analyse von Average-Case Execution Times.

Die wesentlich aufwändigeren Analysen fallen in den Bereich der Realzeit-Systeme und der damit unumgänglichen Berechnung von harten Zeitschranken, was die Auswahl an geeigneten Produkten wesentlich verkleinert. Momentan ist AbsInts aiT Worst-Case Execution Time Analyzer das am weitesten entwickelte und meist verbreitete Produkt für die ARM-Familie. Deshalb wird in Kapitel 2.2.2 nur auf aiT eingegangen, das auch für die Validierung zeitkritischer Avionik-Software bei der Entwicklung des Airbus A380 eingesetzt worden ist [Zie05]

### 2.2.1 Compiler

Um Code zu übersetzen, der auf der Zielplattform ARM920T läuft, eignen sich alle oben aufgelisteten Compiler, jedoch lassen sich die wenigsten weit genug konfigurieren, um für das Cache-Locking optimierten Code zu erzeugen. Außerdem müssen Informationen durch den Compiler bzw. Linker bereitgestellt werden, die bestimmte Eigenschaften des erzeugten Programms für eine Verarbeitung des in Kapitel 3 vorgestellten Workflows enthalten. Dort wird anhand eines exemplarischen Optimierungsdurchlaufs gezeigt, wofür die unten aufgelisteten Informationen und Besonderheiten des Compilers benötigt werden.

Von den oben aufgeführten Compilern erfüllt lediglich die GNU Compiler Collection folgende erforderlichen Randbedingungen für die Übersetzung von Code für gelockte Caches und die Ausgabe von Linker-Informationen:

- **Erzeugung von 32-Bit Code**
  Die Erzeugung von 32-Bit Befehlen ist unumgänglich, um den für Cache-Manipulationen benötigen Coprozessor überhaupt ansprechen zu können.

- **Linker-Konfiguration**
  Es ist notwendig, besondere Linker-Skripte zu erzeugen, um den Lockdown-Code in ein spezielles Segment zu linken, das auf einer eigenen non-cachable Page der MMU liegt.

- **Linker-Informationen**
  Ausführliche Linker-Informationen werden benötigt, die Auskunft über die Startadresse und die Größe einer Funktion während der Ausführung des zu optimierenden Programms geben.

Die für diese Arbeit eingesetzte Version der Realzeit-Version der GCC mit der Versionsnummer 3.2.3 hat den Namen ARM-RTEMS. Sie ist vollständig kompatibel zur Standard-Version der gewöhnlichen ARM-GCC und ist um Funktionen zur Entwicklung von Echtzeit-Applikationen erweitert worden.

### 2.2.2 ACET-/WCET-Analyzer

Analysen, die die Laufzeit von Programmen betreffen, sind in der Vergangenheit bereits häufig behandelt worden und sind inzwischen gängige Mittel, um Realzeit-Systeme zu planen und zu optimieren. Bisher ist bei solchen Systemen der Cache und die damit verbundenen Leistungssteigerungen im Hinblick auf die Ermittlung der Worst-Case Execution Time nicht betrachtet oder einfach weggelassen worden. Erst jüngere Forschungen betrachten das Verhalten von Caches im ungelockten Betrieb und die damit verbundenen Auswirkungen auf die Pipeline durch Cache-Hits und -Misses. Nachfolgend werden die beiden Programme vorgestellt, mit denen die in dieser Diplomarbeit entwickelten Optimierungen getestet worden sind.

#### ACET-Analyzer

Der ARM Sourcelevel Debugger entstammt dem bereits oben erwähnten ARM Software-Development-Toolkit und bietet neben Debugging-Funktionen auch die Möglichkeit, komplette Simulationen der geladenen Programme auszuführen. Als Eingabe dient ihm das zu simulierende Programm, eine Datei mit Speicherinformationen und wahlweise Kommandozeilenoptionen oder eine Konfigurationsdatei für Einstellungen zum Prozessor sowie Takt. Zu jeder Zeit lassen sich Breakpoints setzen und Informationen über Prozessor-, Speicher- und Wartezyklen ausgeben. Mit Hilfe dieser Informationen sind die ACET-Benchmarks in Kapitel 5.2 durchgeführt worden.

#### WCET-Analyzer

Die Ermittlung der WCET ist natürlich aufwändiger und nicht durch Simulation zu ermitteln. Es ist notwendig, den Kontrollflussgraphen des zu betrachtenden Programms zu analysieren und die Laufzeit auf dem längsten Pfad zu ermitteln. Um eine noch genauere obere Schranke für die WCET zu bestimmen, können noch zahlreiche Einflüsse wie Kontexte, Datenabhängigkeiten und Einflüsse auf die Pipeline eines Prozessors betrachtet werden, worauf hier aber nicht weiter eingegangen werden soll. Der eingesetzte aiT

Worst-Case Execution Time Analyzer analysiert genau diese genannten Einflüsse und errechnet mit Hilfe von anzugebenden Obergrenzen für Schleifendurchläufe und Rekursionen eine möglichst genaue obere Zeitschranke für die Ausführung eines Programms. Neuere, noch experimentelle Versionen unterstützen auch die Analyse von einfachen Caches, jedoch nicht oder nur indirekt die Analyse von gelockten Cache-Inhalten. Wie trotzdem die WCET auf Systemen mit gelockten Caches ermittelt werden kann, zeigt Kapitel 5.1.

Die für alle Messungen eingesetzte Version von aiT ist die zur Verfassungszeit aktuelle Version 1.6r5, die weitgehend kompatibel ist zur in [Abs04] beschriebenen Version. Momentan werden die Prozessoren ARM7 (und Abkömmlinge wie ARM9xx), Motorola Star12/HCS12, PowerPC 555 und Infineons TriCore unterstützt. Die am Lehrstuhl Informatik 12 eingesetzte Version für Infineons TriCore unterscheidet sich trotz gleicher Versionsnummer von der Ausführung für die ARM-Familie. Die aiT-Versionen für den TriCore bietet Schnittstelle durch Bibliotheken zu den einzelnen Modulen, die für einen kompletten Analysevorgang und die anschließende Berechnung der WCET eingesetzt werden. Da die ARM-Version von aiT diese Schnittstellen nicht bietet, bleibt nur die Möglichkeit, alle für eine Optimierung nötigen Informationen aus den Reports und generierten Graphen eines Analysevorgangs durch Parsen zu extrahieren.
Damit aiT die Analyse eines Programms überhaupt durchführen kann, benötigt es bestimmte Angaben durch den Benutzer, um die oben angegebenen Werte möglichst genau zu bestimmen. Dies sind z.B. Spezifikationen, die die Hardware beschreiben, oder Informationen über die Struktur des zu analysierenden Programms, die es ermöglichen, den Kontrollfluss überhaupt erst zu bestimmen oder die Präzision der bestimmten Zeitschranken zu erhöhen. Die bereitzustellenden Informationen können bei der ARM-Version von aiT in zwei Dateien geschrieben werden, die sich ebenfalls in Informationen die Hardware betreffend und Software beschreibend aufteilen.

Hardware-Spezifikationen werden in eine `AIP` genannte Datei geschrieben und umfassen u.a. folgendes:

- **Speicherausstattung**
  Es können beliebig große Speicherbereiche mit jeweils unterschiedlicher Busbreite und Zugriffszeiten angegeben werden, die das Speicherlayout des untersuchten Systems widerspiegeln. Es können auch Informationen über die Speicherart wie RAM oder ROM angegeben werden oder Segmente, die Daten und/oder Instruktionen enthalten, spezifiziert werden. Dadurch werden Speicherzugriffe wie auf einem realen System mit unterschiedlichen Verzögerungen in die Analyse miteinbezogen.

- **Taktfrequenz**
  Um die WCET als Zeiteinheit berechnen zu können, wird die Taktfrequenz des untersuchten Systems angegeben.

Angaben den Kontrollfluss betreffend werden in eine Datei geschrieben (`AIS`-Datei), die folgende Informationen enthalten kann:

- **Loop-Bounds**
  Zu Beginn der Analyse versucht aiT, die Anzahl von Durchläufen für jede Schleife zu bestimmen, was jedoch meist nur bei einfachen Schleifen und konstanten Indizes gelingt. Ist die Anzahl von Durchläufen abhängig vom Programmzustand bzw. Variableninhalten, so muss der Benutzer durch eigene Analysen die obere Grenze für diese Schleifen angeben.

- **Rekursionstiefen**
  Genau wie bei den Angaben zu den Loop-Bounds müssen Rekursionstiefen u.U. von Hand angegeben werden, damit eine Analyse durch aiT möglich ist.

- **Funktionsaufrufe und Sprünge**
  Im Idealfall bestimmt aiT die Sprungziele z.B. für Funktionsaufrufe selber. Bei komplizierteren Konstrukten, wie Sprüngen mit registerindirekten Zielen, müssen oft die Ziele von Hand bestimmt und spezifiziert werden.

- **Lese-/Schreibbefehle**
  Die Geschwindigkeit von LOAD- und STORE-Befehlen hängt direkt von der Geschwindigkeit des gerade benutzten Speichersegments ab. Deshalb müssen bei Speicheroperationen meist die Quell- bzw. Zieladressen angegeben werden, damit aiT mit Hilfe des Speicherlayouts die individuelle Verzögerung der Speicherzugriffe bestimmen kann.

Die bisher genannten Annotationen sind nur ein kleiner Auszug der von aiT unterstützten, um das Zielsystem so gut wie möglich zu beschreiben, damit man eine möglichst genaue Zeitschranke für die Ausführung von Programmen erhält. Hier sind nur die für die hier entwickelten Optimierungen benutzten beschrieben worden, weshalb für eine genaue Beschreibung des vollen Funktionsumfanges auf [Abs04] verwiesen wird.

Um die WCET für ein Programm zu berechnen, kombiniert aiT zwei Techniken, die in Kapitel 4 beschriebene *Lineare Programmierung* zusammen mit der *Implicit Path Enumeration Technique* (IPET), die in [BR06] ausführlich behandelt wird. Welche Analysen aiT im einzelnen mit den zahlreichen verwendeten Modulen ausführt, ist in anderen Arbeiten bereits ausreichend beschrieben worden und wird deshalb an dieser Stelle nicht mehr aufgeführt. Einen kleinen Einblick bietet das Handbuch [Abs04], für weitergehende Erklärungen bietet [HF04] eine gute Basis.

Ist der Analysevorgang beendet, gibt es eine Reihe von Dateien unterschiedlichen Typs, die Informationen über die einzelnen Analyseschritte beinhalten. Ein paar für den weiteren Verlauf interessante Dateien sind:

- **GDL**-Datei
  Die GDL-Datei enthält eine Beschreibung des von aiT bestimmten Callgraphen einer analysierten Binärdatei. Dargestellt sind die Graphen in der *Graph-Description-Language*, die u.a. Informationen über Basisblöcke, Sprungbefehle und Ausführungszeiten wiedergeben kann.

- **Report**-Datei
  aiT schreibt Nachrichten über die verarbeiteten Eingabedaten aus den AIS- und AIP-Dateien sowie Informationen über die einzelnen Analyseschritte in eine Report-Datei. Diese Report-Datei enthält auch die berechneten Laufzeiten der einzelnen Funktionen, unterteilt in Methodenrumpf und enthaltene Schleifen, sowie die gesamte Worst-Case Execution Time der analysierten Binärdatei.

Diese Dateien werden für die Weiterverarbeitung durch die Algorithmen aus Kapitel 4 herangezogen. Welche Informationen benötigt werden, wird an den entsprechenden Stellen weiter erläutert.

# Kapitel 3

# Workflow

Viele gängige Optimierungsverfahren ändern den Quelltext von Programmen z.B. durch Loop-Unrolling, Loop-Tiling oder Verwendung von besonders performanten Befehlen und Inline-Assembler, was z.T. durch größeren Code erkauft wird. Ähnliche und andere Ansätze findet man im Compiler, wo Gewinne in der Laufzeit z.B. durch Selektion von besonders schnellen Befehlsfolgen und Entfernung von überflüssigen Operationen erzielt werden. Neu hinzugekommen sind Optimierungen für Scratchpads, bei denen einzelne performancekritische Basisblöcke in schnelle Onchip-Speicher verschoben werden.

Die meisten bisherigen Optimierungen für Average-Case Execution Times und Worst-Case Execution Times finden also vor oder während des Übersetzungsvorganges statt und sind somit indirekt abhängig von der verwendeten Programmiersprache oder dem Compiler-Backend.

Der hier vorgestellte Workflow für die Anwendung der in Kapitel 4 vorgestellten Algorithmen ist weitgehend unabhängig von der verwendeten Programmiersprache und dem dazugehörigen Compiler. Dies wird erreicht durch die Generierung eines Startup-Codes, der den eigentlichen Lockdown-Code enthält. Dieser Code wird den Objektdateien des zu optimierenden Programms hinzugelinkt und wird bei Programmstart angesprungen. So werden erst die Speicherbereiche initialisiert und Instruktionsblöcke im Cache gelockt, bevor zur Hauptroutine des eigentlichen Programms verzweigt wird.

Der Workflow für einen Übersetzungsvorgang eines C-Programms mit Optimierungen der WCET durch statisches Cache-Locking ist in Abbildung 3.1 dargestellt. Ausgehend von dem Quellcode des Programms wird zunächst ein ausführbares Programm erzeugt, das von der GCC übersetzt und durch die Option `-Os` in der Größe optimiert wird. Das Binärprogramm wird vom Linker mit generischen Startup-Code ohne Lockdown-Instruktionen versehen. Die Daten des Link-Vorganges, die Informationen über Funktions- sowie Objektgrößen bzw. Startadressen des Binärprogramms im Adressraum enthalten, werden von WCETOPT verarbeitet. WCETOPT ist der Teil der

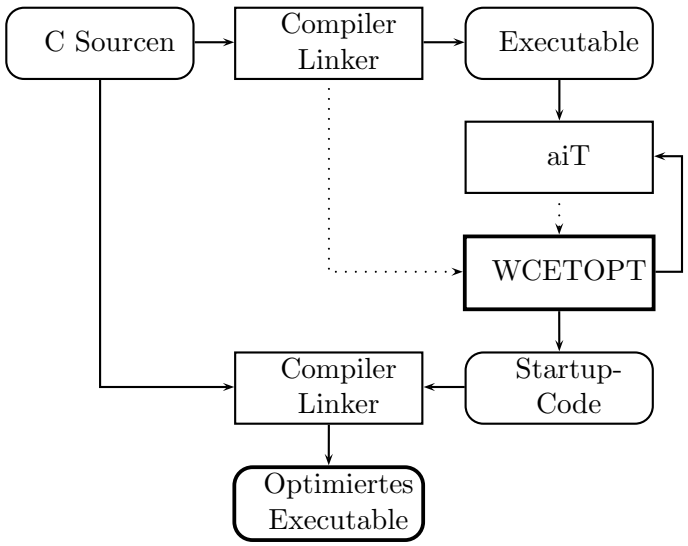



**Abbildung 3.1:** Workflow für statisches Cache-Locking

benutzten Toolchain, in dem die Selektionsalgorithmen für die Optimierung der Worst-Case Execution Time durch statisches Cache-Locking implementiert worden sind. Es enthält auch den Großteil der Routinen, um nach der Erzeugung der Binärdatei diese von aiT analysieren zu lassen und die Ergebnisse auszuwerten. Je nach Struktur des zu analysierenden Programms werden ggf. durch weitere aiT-Analysen die zu lockenden Funktionen bestimmt. WCETOPT erzeugt einen Startup-Code, der den eigentlichen Lockdown-Code enthält und das Locking der Funktionen in den Cache durchführt. Dieser Startup-Code wird bei einem erneuten Übersetzungsvorgang dem C-Programm hinzugelinkt und so das optimierte Maschinenprogramm erzeugt. Im Folgenden wird kurz auf die Voraussetzungen für die Optimierung der WCET eines Programmes durch statisches Cache Locking eingegangen, danach auf den eigentlichen Ablauf von Übersetzung und Analyse bis hin zur Erzeugung des optimierenden Startup-Codes. Das Hinzulinken des Startup-Codes wird hier nicht behandelt, da es zum einen sehr compilerspezifisch ist und zum anderen lediglich dem gewöhnlichen Linker-Aufruf zur Generierung der Binärdatei aus den einzelnen Objektdateien entspricht.

## 3.1 Voraussetzungen

Damit der hier vorgestellte Arbeitsablauf funktioniert, müssen die Informationen über den Link-Prozess aus Kapitel 2.2.1 verfügbar sein. Außerdem soll an dieser Stelle noch auf Grenzen des vorgestellten Verfahrens sowie sinnvolle Benchmarks eingegangen werden, um die Selektionsverfahren zur Optimierung der WCET durch statisches Cache Locking zu testen.

### Granularität

Die weiter oben erwähnten Scratchpad Optimierungen arbeiten auf der Ebene von Basisblöcken und vermögen so, besonders kleine, aber kritische Teile des Programmcodes zu optimieren. Die Selektionsalgorithmen aus dem nächsten Kapitel arbeiten alle auf Funktionsebene, d.h. es können als atomare Einheit nur ganze Funktionen gelockt werden, was mehrere Gründe hat. Zuerst einmal müssen die Startadresse und die Länge des zu lockenden Bereichs bekannt sein, was der GCC-Linker nur für Funktionen ausgibt. Außerdem gibt es momentan keine Möglichkeit die WCET von Basisblöcken als kleinsten Einheiten durch aiT zu bestimmen, was im nächsten Kapitel noch einmal ausführlich begründet wird.

### Sinnvolle Benchmarks

Bei Optimierungsproblemen bzgl. der WCET eines Programmes stellt sich die Frage, wann es sich lohnt, den Nutzen eines Algorithmus durch Benchmarks zu testen und deren Funktion zu verifizieren. Bei Funktionen als atomaren Einheiten macht es keinen Sinn, komplizierte Selektionsalgorithmen durch Benchmarks mit wenigen Funktionen auf Verbesserungen zu testen. Bei den schnell wachsenden Cache-Größen sollte zudem ein zu optimierendes Stück Code viel größer als der verwendete Cache sein, denn sonst wäre der Einsatz eines schnellen Onchip-Speichers, der das gesamte Programm fasst, oder das komplette Laden des Programms in den Cache naheliegend.

Die in Kapitel 5.2 untersuchten Benchmarks bestehen aus oben genannten Gründen aus mindestens 15 Funktionen, und die erzeugten Binärdateien sind zwischen 45 KB und 595 KB groß.

## 3.2 Übersetzung

Vor jeder Analyse und Optimierung müssen erst einmal die einzelnen Quelldateien übersetzt und zu einem ausführbaren Programm mit generischem Startup-Code gelinkt werden, da sich alle Analysen auf den bereits übersetzten Binärcode beziehen. Für den Übersetzungsvorgang wird der Kommandozeilenschalter `-Os` benutzt, der den Compiler anweist, das erzeugte Binärprogramm in der Größe zu optimieren. Der Inhalt des Startup-Codes ist zu diesem Zeitpunkt noch ohne Bedeutung, weshalb die Standard-Initialisierungsroutinen der GCC verwendet werden.

Alle hier vorgestellten Programme und Benchmarks sind in der Programmiersprache C implementiert worden. Durch die Verwendung der GCC ist es theoretisch möglich, alle unterstützten Sprachen durch die Benutzung der entsprechenden Frontends zu übersetzen und anschließend zu optimieren. Als Auswahl an verfügbaren Frontends wären C, C++, Java, Objective-

C, Fortran(95), Treelang und Ada zu nennen. Jedoch ist der beschriebene Workflow nur für die Sprache C getestet und durchgeführt worden.

Ein Vorteil der Analyse von fertig übersetztem Code ist die Möglichkeit der Einbeziehung von Bibliotheken in die Optimierung. Die meisten bisher verwendeten Verfahren setzen das Vorhandensein des Quelltextes von zu optimierenden Programmen voraus, damit die zu Beginn des Kapitels genannten Optimierungen überhaupt möglich sind. Werden von Programmen fertige Bibliotheken wie die Standard C Library oder hinzugekaufte Bibliotheken verwendet, hat man meist keinen Zugriff auf den Quellcode und muss besonders performancekritische Funktionen aufwändig nachimplementieren. Die hier vorgestellten Analyseverfahren benötigen für bereits existierende Bibliotheken oder Objektdateien nur Angaben zu Loop-Bounds und Rekursionstiefen, um diese in den Optimierungsprozess miteinzubeziehen. Diese oberen Schranken für den Programmablauf müssen dann durch Reverse-Engineering oder durch Debuggen ermittelt werden.

## 3.3 Analyse

Die hier besprochene Analyse der Binärdatei besteht aus zwei Teilen, der Auswertung von Compiler- bzw. Linker-Informationen sowie der eigentlichen Laufzeitanalyse durch einen der in Kapitel 4 vorgestellten Algorithmen in Zusammenarbeit mit dem WCET-Analyzer aiT.

Die Auswertung des Link-Vorganges dient dazu, die Aufteilung des Binär-Programms in Daten und Funktionen mit Adressangaben in einer Datenstruktur zu speichern. Zu diesem Zweck extrahieren Skripte die benötigten Informationen aus der Ausgabe des Programms arm-rtems-readelf. Das Programm liest u.a. Informationen über die Aufteilung in einzelne Sections, Segmente, Startadressen und die Länge von Funktionen sowie Objekten aus den Symbolen der Assemblerdirektiven .section bzw. .size einer ausführbaren Datei aus. Den genauen Aufbau eines Binärprogramms im *Executable and Linkable Format* (ELF) beschreibt [TIS93]. Die Erweiterungen der ARM Implementierung, die vom ARM Software-Development-Toolkit umgesetzt werden, werden in [Adv98c] gezeigt. Als Datenstruktur für die gewonnenen Informationen dienen u.a. verkettete Listen, die jeweils für Funktionen und Objekte deren Startadresse, Adresse des letzten Bytes sowie den Namen speichern und in Kapitel 4 genauer beschrieben werden. Zusätzlich können später Informationen darüber gespeichert werden, wie die Laufzeit einer Funktion in Takten ist, und ob diese Funktion gelockt werden soll.

Der aufwändigere Teil der Analyse ist die Lösung des Selektionsproblems. Die drei im nächsten Kapitel vorgestellten Algorithmen verarbeiten dazu die Informationen aus der erzeugten Datenstruktur und benutzen aiT, um die Wors-Case-Laufzeit der einzelnen Funktionen oder des gesamten Programms zu bestimmen. Ein derartiger Analysevorgang wird normalerwei-

se über ein graphisches Benutzerinterface gesteuert, mit dessen Hilfe man die zahlreichen Optionen ändern kann. Für eine automatisierte Optimierung eines Programms sind manchmal über 100 aiT-Durchläufe nötig, so dass eine Benutzerinteraktion nicht in Frage kommt. Deshalb werden alle aiT-Projektdateien mit Optionen und Spezifikationen vor jedem Durchlauf automatisch erzeugt und aiT durch eine Kommandozeilenoption für eine komplette WCET-Analyse ohne grafische Anzeige gestartet. Die Ausgabedateien mit den Informationen über die Analyse und die erzeugten Graphen werden auch ohne GUI erzeugt und lassen sich über die Projektdateien steuern. Das Extrahieren der Informationen aus jedem aiT-Durchlauf geschieht durch Skripte, die die relevanten Informationen aus den Ausgabedateien filtern und in separate Eingabedateien für die WCETOPT-Toolchain schreiben. Die Informationen, die aus den Dateien extrahiert werden, sind:

- **Laufzeit**
  Die Laufzeit der einzelnen Funktionen wird aufgeteilt in den Funktionsrumpf und die enthaltenen Schleifen von aiT ausgegeben. Die einzelnen Laufzeiten werden dann zu der Gesamtlaufzeit der betrachteten Funktion aufsummiert.

- **Funktionsaufrufe**
  Findet ein Funktionsaufruf statt, so gibt aiT neben dem Namen der aufgerufenen Funktion auch die Hex-Adresse des Sprungbefehls an. Anhand dieser Adresse wird aus der Liste der Funktionen und deren Länge die aufrufende Funktion bestimmt. Mit den beiden Informationen wird dann in Kapitel 4.5.1 der Callgraph des zu analysierenden Programms bestimmt.

Durch diese gewonnenen Informationen wird versucht, eine möglichst optimale Kombination an Funktionen zu finden, die später in den Cache geladen und gelockt wird. Eine optimale Lösung ist die Kombination, die mit möglichst wenig benutztem Cache-Speicher die geringste Gesamtlaufzeit für das optimierte Programm erzielt. Vorgehensweisen, Probleme und Randbedingungen werden in Kapitel 4 erläutert.

## 3.4 Erzeugung von Startup-Code

Haben die in Kapitel 4 vorgestellten Selektionsalgorithmen die Menge der in den Cache zu lockenden Funktionen ermittelt, muss die erzeugte Binärdatei um Lade- und Lockdown-Befehle ergänzt werden. Diese Befehle füllen den Cache mit den ausgewählten Funktionen, Schützen den Inhalt durch ein Lockdown vor Verdrängung und werden in den Startup-Code integriert. Als Startup-Code bezeichnet man den Teil eines Programms, der vor Ausführung der eigentlichen Programm-Funktionalität – i.d.R. die Routine mit

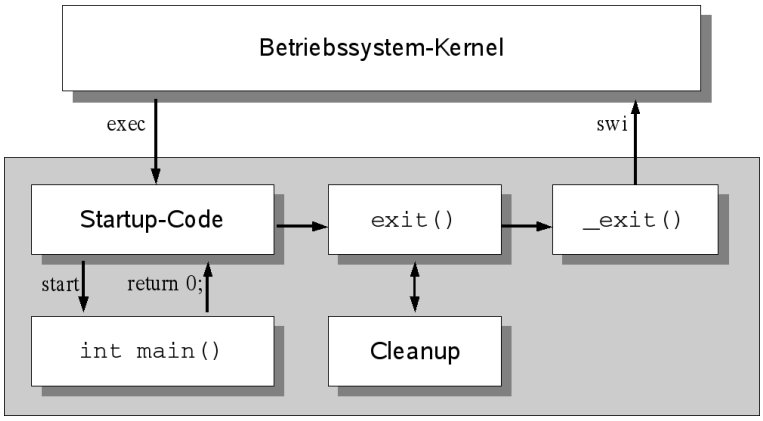



**Abbildung 3.2:** Kontrollfluss beim Programmstart

dem Namen *main* – abgearbeitet wird. Normalerweise ist der Startup-Code für Initialisierungszwecke vor Programmausführung und anschließende Aufräumarbeiten zuständig. Die Aufräumarbeiten werden nach Beendigung des Hauptprogramms durch den Aufruf der Funktion exit() ausgeführt und geben z.B. durch Variablen belegten Speicher wieder frei. Sind die Aufräumarbeiten erledigt, wird die Funktion _exit() ausgeführt, die dem Betriebssystem durch einen Software-Interrupt das Programmende signalisiert. Abbildung 3.2 zeigt den Ablauf bildlich, weitere Informationen gibt [Wol06].

Soll ein Teil des Codes in den Cache geladen und gelockt werden, so muss der Startup-Code um geeignete Routinen ergänzt werden. Der Ablauf eines Lockdown-Vorgangs ist in Kapitel 2.1.3 bereits erläutert worden, weshalb an dieser Stelle die wichtigen Teile des nötigen Codes besprochen werden sollen. Die folgenden Funktionen sind für das Locking nötig und werden anschließend besprochen:

- Disable/Enable Interrupts
- Enable Cache
- Enable Round-Robin Replacement-Policy
- Flush Cache
- Lock I-Cache

### Disable/Enable Interrupts

Die Funktion *disable_int* aus Listing 3.1 schaltet die Interrupts vor dem Lockdown von Cache-Inhalten aus und ist vom Aufbau der Funktion zum Einschalten der Interrupts sehr ähnlich. Die zweite Zeile und die letzte Zeile sind wie bei allen folgenden Listings für das Sichern der Rücksprungadresse

auf den Stack und das Rückspringen nach Beendigung der Funktion zuständig und werden deshalb nur einmal besprochen. Zeile 3 liest das Coprozessor-Statusregister in Register r1, damit in der nächsten Zeile das Bit für das Abschalten der Interrupts im selben Register gesetzt werden kann. Bevor der Rücksprung erfolgt, wird das Coprozessor-Statusregister noch zurückgeschrieben. Der einzige Unterschied im Code, um die Interrupts einzuschalten, ist, dass das bereits erwähnte Bit in Zeile 4 gelöscht anstatt gesetzt wird.

```
1 disable_int:
2         STMFD r13!, {lr}
3         MRS r1, CPSR
4         ORR r1, r1, #0x80
5         MSR CPSR_c, r1
6         LDMFD r13!, {PC}
```

**Listing 3.1:** Funktion zum Abschalten von Interrupts

### Enable Cache/Round-Robin Replacement Policy

Die Routine zum Einschalten des Caches und der Round-Robin Replacement Policy ist in Listing 3.2 dargestellt und benutzt wie alle Funktionen zur Cache Manipulation das Coprozessor-Register CP15. Dazu liest Zeile 3 den Inhalt über einen Coprozessor-Befehl in Register r0, damit anschließend das Bit für die Ersetzungsstrategie des Cache-Inhaltes gesetzt werden kann. Zeile 5 setzt das Bit, das den Befehls-Cache einschaltet, falls dieser noch nicht benutzt wird. Die vorletzte Zeile schreibt den manipulierten Inhalt zurück und wendet so die Änderungen an.

```
1 enable_cache:
2         STMFD r13!, {lr}
3         MRC p15, 0, r0, c1, c0, 0
4         ORR r0, r0, #0x4000
5         ORR r0, r0, #0x1000
6         MCR p15, 0, r0, c1, c0, 0
7         LDMFD r13!, {PC}
```

**Listing 3.2:** Funktion zum Einschalten des Caches und der Replacement Policy

### Flush Cache

Damit keine Instruktionen, die in den Cache gelockt werden sollen, an anderer Stelle bereits darin enthalten sind, wird vor dem Lockdown-Vorgang der Cache geflusht. Im Gegensatz zum D-Cache, wo alle geänderten Daten zuerst zurückgeschrieben werden müssen, kann der I-Cache mit einzigen Befehl invalidiert und geflusht werden. Dieser 5-Zeiler ist in Listing 3.3 abgebildet.

```
1 flush_cache:
2         STMFD r13!, {lr}
3         MOV r0, #0
4         MCR p15, 0, r0, c7, c5, 0       @; Flush I-Cache
5         LDMFD r13!, {PC}
```

**Listing 3.3:** Flush-Cache Routine

### Lock I-Cache

Der Programmcode aus Listing 3.4 erwartet die Startadresse des zu lockenden Bereichs in Register r0 und die Endadresse in Register r1. Es findet allerdings keine Abfrage statt, ob der zu lockende Bereich auch wirklich in den vorhandenen Cache-Speicher passt. Dies wird durch die Selektionsalgorithmen sichergestellt, die für die Offline-Einteilung des Caches zuständig sind.

```
1 lock_icache
2         BIC r0, r0, #31
3         MOV r2, #0
4         MCR p15, 0, r2, c9, c0, 1
5 li_loop:
6         MCR p15, 0, r0, c7, c13, 1
7         ADD      r0,r0,#32
8
9         AND      r3,r0,#0xE0
10        CMP      r3,#0x0
11        ADDEQ    r2,r2,#0x1<<26
12        MCREQ    p15,0,r2,c9,c0,1
13
14        CMP      r0,r1
15        BLE      li_loop
16
17        CMP      r3,#0x0
18        ADDNE    r2,r2,#0x1<<26
19        MCRNE    p15,0,r2,c9,c0,1
```

**Listing 3.4:** Lockdown-Code

In Zeile 2 wird die Startadresse an einem Vielfachen von 32 Bytes ausgerichtet, danach das Register r2, das den Victim Base Pointer zwischenspeichern soll, auf Way 0 gesetzt. Zeile 4 setzt die Victim Base Pointer und Victim Pointer Register durch einen Coprozessor-Befehl.

Die Schleife von Zeile 5 bis Zeile 15 sorgt für den eigentlichen Lockdown-Vorgang. In Zeile 6 wird zuerst eine Cache-Zeile durch einen Coprozessorbefehl geladen und anschließend in Zeile 7 der Adresszeiger auf die nächste Cache-Zeilenadresse für das nächste Segment erhöht. Die Zeilen 9 bis 12 überprüfen, ob ein Überlauf in den Adressbits vom letzten zum ersten Segment erfolgt ist. Ist das der Fall, wird die Victim Base Pointer und die

Victim Pointer Adresse um ein Way erhöht und in die Register geschrieben. In Zeile 14 wird überprüft, ob bereits genügend Daten in den Cache geladen worden sind. Falls dies nicht der Fall ist, springt Zeile 15 wieder zurück an den Schleifenkopf.

Die Zeilen 17-19 sorgen dafür, dass falls die benutzten Adressbits nicht schon wieder auf Segment 0 zeigen, der Victim Pointer um eins erhöht wird und so eine normale Cache-Operation mit den verbleibenden Zeilen gesichert ist. Anhand des Codebeispiels sieht man bereits, dass ein Lockdown-Vorgang immer ab dem ersten Way beginnt und momentan nur ein statisches Locking des Caches unterstützt wird, da keine Befehle vorgesehen sind, um den Inhalt des Caches zur Programmlaufzeit dynamisch zu verändern.

# Kapitel 4

# Selektionsalgorithmen für statisches Cache-Locking

Dieses Kapitel beschreibt die eigentlichen Algorithmen zur Optimierung der Worst-Case Execution Time von Programmen durch statisches Cache-Locking. Dazu werden im folgenden Abschnitt zuerst die Grundlagen für Optimierungsprobleme der Laufzeit von Programmen betreffend und Besonderheiten bei der Betrachtung von WCET-Optimierungen erläutert. Anschließend folgen die drei für diese Diplomarbeit entwickelten Selektionsalgorithmen für statisches Cache-Locking. Diese sollen eine möglichst optimale Menge an Funktionen für ein Lockdown in den Cache auswählen, um die WCET des zu optimierenden Programms zu minimieren.

Die vorgestellten Algorithmen benutzen unterschiedliche Vorgehensweisen, um möglichst gute Lösungen zu finden. Die Kapitel 4.3 und 4.4 stellen zwei Ansätze vor, die vom Aufwand der Implementierungen relativ einfach sind, dafür aber weniger gute Resultate liefern oder von der Zeitkomplexität her schlecht sind. Der Algorithmus in Kapitel 4.5 ist von der Umsetzung im Vergleich zu den anderen sehr aufwändig, liefert dafür aber bessere Resultate und ist von dem nötigen Zeitaufwand für die Optimierungen beiden überlegen.

## 4.1 Grundlagen

In den vorigen Kapiteln ist bereits ausführlich beschrieben worden, was Cache-Locking eigentlich ist, und anhand von Quelltexten ist verdeutlicht worden, wie ein Lockdown-Vorgang funktioniert. Jetzt stellt sich die Frage, welche Instruktionen in den Cache geladen und gelockt werden sollen, um einen möglichst großen Gewinn im Hinblick auf die WCET eines Programms zu erreichen. Die im Folgenden vorgestellten Selektionsalgorithmen versuchen die optimale Menge an Funktionen zu finden, die mit möglichst

geringem Einsatz an Cache-Speicher die größten Verbesserungen der Worst-Case Execution Time bewirken.

Typische Ansätze für Scratchpad-Optimierungen, die versuchen die Average-Case Execution Time zu senken oder Energie zu sparen, verwenden für die Auswahl von Elementen die lineare Programmierung, die eine lineare Zielfunktion zu optimieren versucht. Die Zielfunktion ist durch eine Menge von linearen Gleichungen und Ungleichungen eingeschränkt. Um das Problem formulieren zu können, werden folgende Definitionen eingeführt:

- $x_i$ : Ist eine Funktion
- $X$ : Ist die Menge der Funktionen mit $x_i \in X$
- $U$ : Ist die Menge der ungelockten Funktionen
- $L$ : Ist die Menge der gelockten Funktionen
- $T$ : Ist die Menge der temporär gelockten Funktionen
- $n = |X|$ : Ist die Mächtigkeit der Menge $X$
- $a(x_i)$ : Ist die Ausführungshäufigkeit der Funktion $x_i$
- $r(x_i)$ : Ist der Gewinn durch das Locking der Funktion $x_i$
- $s(x_i)$ : Ist die Größe der Funktion $x_i$ in Bytes
- $w(x_i)$ : Worst-Case-Laufzeit der Funktion $x_i$ in Takten, falls $x_i \in U$
- $w_l(x_i)$ : Worst-Case-Laufzeit der Funktion $x_i$ in Takten, falls $x_i \in L$
- $S_c$ : Ist die Größe des Caches in Bytes
- $F_c$ : Ist die Größe des freien Caches in Bytes
- $R(L)$ : Ist die Gewinnfunktion für das Lockdown der Funktionen mit $x_i \in L$

Der Gewinn $r(x_i)$ durch das Locking der Funktion $x_i$ wird im Folgenden als Reward bezeichnet und in CPU-Takten gemessen. Der Reward ist wie folgt definiert:

$$r(x_i) = w(x_i) - w_l(x_i) \tag{4.1}$$

Wird eine Funktion mehr als einmal ausgeführt, muss deren Ausführungshäufigkeit mit einbezogen werden:

$$r_A(x_i) = (w(x_i) - w_l(x_i)) * a(x_i) \tag{4.2}$$

Mit den obigen Definitionen lässt sich das Selektionsproblem durch eine Zielfunktion formulieren:

$$R(L) = \sum_{i=1}^{n} l_i * r_A(x_i) \tag{4.3}$$

Dabei ist $l_i$ ein Vektor, der für jede Funktion angibt, ob diese gelockt wird oder nicht. Ist $l_i = 0$, so ist $x_i \in U$, also nicht gelockt, und wenn $l_i = 1$, dann wird die Funktion $x_i$ im Cache gelockt.
Die Beschränkung für die Optimierung der Zielfunktion 4.3 ist die Einhaltung der Größe des vorhandenen Caches und ergibt sich zu:

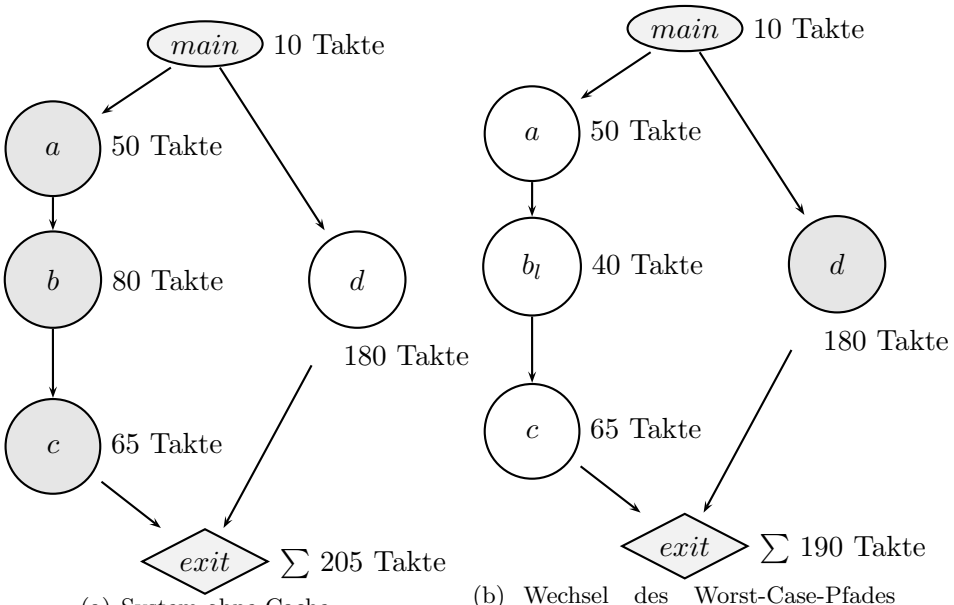
$$S_c \geq \sum_{i=1}^{n} l_i * s(x_i) \tag{4.4}$$

Unter Beachtung der Beschränkung aus Gleichung 4.4 lässt sich durch einen IP-Solver die Zielfunktion maximieren und so der optimale Vektor $l$ bestimmen. So lässt sich z.B. die Menge der Elemente bestimmen, die in den Scratchpad-Speicher verschoben werden, um eine möglichst große Energiemenge zu sparen oder die Average-Case Execution Time von Programmen zu senken.

Für die Optimierung von Programmlaufzeiten im Worst-Case eignen sich solche Algorithmen nur bedingt, und wenn überhaupt für Programme mit sequentiellem Programmfluss. Besitzt ein Programm viele bedingte Verzweigungen, lässt sich das Selektionsproblem zur WCET-Minimierung nicht einfach in Gleichungen für den IP-Solver fassen.

Um die Problematik bei WCET-Optimierungen zu verstehen, sind in Abbildung 4.1(a) die möglichen Kontrollflüsse eines einfachen Programms mit fünf Funktionen dargestellt. An jeder Funktion ist deren Worst-Case-Laufzeit annotiert, die Gesamt-WCET ist als Summe an das symbolische Programmende mit dem Namen *exit* annotiert. Die Abbildung zeigt ein System ohne Cache und der Ausführungspfad mit der längsten Laufzeit ist grau hinterlegt. Die Laufzeit des linken Pfades ausgehend von *main* mit den Knoten $a$, $b$ und $c$ hat eine Laufzeit von 205 Takten und ist um 25 Takte länger als der rechte Pfad. Wird nun die Funktion $b$ in den Cache geladen und gelockt, so verkürzt sich ihre Laufzeit von 80 auf 40 Takte. Die Länge des linken Pfades verkürzt sich ebenfalls um 40 Takte auf 165 Takte. Nun ist nicht mehr der linke Ausführungspfad der längste, denn allein die Laufzeit der Funktion $d$ ist bereits länger als der ganze linke Pfad. Der nach dem Locking von $b$ aktuell längste Ausführungpfad $main \rightarrow d$ ist in Abbildung 4.1(b) dargestellt.

Es findet also durch das Lockdown einer Funktion u.U. ein Wechsel des Worst-Case-Pfades statt. Dieser mögliche Wechsel des Worst-Case-Pfades




(a) System ohne Cache

(b) Wechsel des Worst-Case-Pfades durch Locking von $b$

**Abbildung 4.1:** Wechsel des Worst-Case-Pfades durch Locking eines Elementes

macht es schwierig, wenn nicht sogar unmöglich, eine Zielfunktion und die dazu gehörigen Gleichungen für Beschränkungen aufzustellen. Es wäre zwar möglich, Gleichungen für einen IP-Solver aufzustellen, die die Elemente auf dem Worst-Case-Pfad miteinbeziehen, jedoch könnten die Gleichungen nach Selektion und Locking eines einzelnen Elementes bereits ungültig sein. Da die WCET eines Programms nur durch die Ausführungszeit des längsten Pfades bestimmt wird, ist es sinnlos, Elemente zu optimieren, die nicht auf dem Worst-Case-Pfad liegen. So könnten diese Gleichungen Programmteile optimieren, die auf die WCET des Programms keinen Einfluss haben.

Es gibt bereits Selektionsalgorithmen, die entlang eines einmal ermittelten Worst-Case-Ausführungspfades optimieren. Diese Ansätze arbeiten alle mit der Annahme, dass das betrachtete Programm wenig Verzweigungen im Kontrollfluss hat bzw. die nebenläufigen Pfade sehr kurz sind im Vergleich zum Worst-Case-Pfad. Diese "Einzelpfadanalyse" nutzt der in Kapitel 4.2 vorgestellte Algorithmus sowie die Arbeiten [CPIM05] und [PD02], die sich ebenfalls mit dem Lockdown von Caches beschäftigen.

Abhilfe schaffen die Selektionsalgorithmen aus Kapitel 4.4 und 4.5, die auch Wechsel von Ausführungspfaden berücksichtigen. Da diese Algorithmen aufgrund der oben genannten Schwierigkeiten das Selektionsproblem nicht mehr durch Lineare Programmierung lösen können, muss die Berechnung des Rewards einer Funktion angepasst werden. Der Gewinn $r(x_i)$ durch

das Locking der Funktion $x_i$ wird nun auf den Gewinn in Takten pro Byte normiert und wird für alle folgenden Abschnitte wie folgt abgeändert:

$$r(x_i) = \frac{w(x_i) - w_l(x_i)}{s(x_i)} \tag{4.5}$$

Wird eine Funktion mehr als einmal ausgeführt, muss deren Ausführungshäufigkeit mit einbezogen werden:

$$r_A(x_i) = \frac{w(x_i) - w_l(x_i)}{s(x_i)} * a(x_i) \tag{4.6}$$

Durch die Normierung des Gewinns pro gelocktem Byte kann nun durch den Vergleich der Rewards direkt entschieden werden, für welche Funktionen es besonders lohnenswert ist, sie in den Cache zu locken. Also bewertet der Reward einer Funktion, wieviel Nutzen in Takten pro genutztem Byte Cache-Speicher das Lockdown der Funktion erbringt.

## 4.2 Einzelpfadanalyse

Die Einzelpfadanalyse ist eine der einfachsten Möglichkeiten, um Funktionen zu ermitteln, die eine möglichst große Verringerung der Worst-Case Execution Time durch das Locking in den Cache erreichen. Dabei wird nicht immer zwangsläufig der Worst-Case-Pfad ermittelt, um den Gewinn einer Funktionen durch die Optimierung zu bestimmen. Vielmehr werden Wechsel des längsten Ausführungspfades durch das Locking von Elementen in den Cache nicht berücksichtigt, weshalb an dieser Stelle von "Einzelpfadanalyse" gesprochen wird. Es gibt verschiedene Möglichkeiten eine solche Analyse durchzuführen, wobei der schematische Ablauf immer derselbe ist:

1. Eine Bewertungsfunktion bestimmt den Nutzen jeder Funktion durch ein Lockdown in den Cache. Als Grundlage können beispielsweise die Anzahl der ausgeführten Befehle im Average-Case oder die Funktion zur Reward-Berechnung (Gleichung 4.1 bzw. 4.5) dienen.

2. Bestimmung einer möglichst optimalen Menge $L$, deren Funktionen eine möglichst große Einsparung an Takten der WCET ermöglicht.

Für die Bestimmung der Menge der zu lockenden Funktionen $L$ bietet sich zum einen die Lineare Programmierung aus dem vorigen Abschnitt mit einer Zielfunktion und Ungleichungen zur Beschränkung des Problems an. Zum anderen ist es mit dem folgenden Verfahren, dessen Pseudocode in Algorithmus 1 abgebildet ist, möglich, die Elemente der Menge $L$ auszuwählen:

Zu Beginn wird bei dem Einzelpfad-Algorithmus einmal die Laufzeit aller Funktionen auf dem Worst-Case-Ausführungspfad sowohl in einem System ohne Cache (Zeile 1) als auch in den Cache geladen und gelockt (Zeile 3)

mit Hilfe eines Softwarewerkzeuges wie aiT bestimmt. Anschließend wird der Reward jeder Funktion durch das Locking (Zeile 5) in den Cache mit Hilfe der Gleichung 4.5 berechnet. Die Schleife in den Zeilen 6 - 8 wählt immer das Element mit dem größten Reward aus der Menge der ungelockten Funktionen aus, falls es in den freien Cache passt, und lockt es dauerhaft. Abgebrochen wird die Schleife, falls alle Elemente in den Cache gelockt sind oder kein ungelocktes Element mehr in den freien Cache passt. Der freie Cache wird bei jedem Lockdown einer Funktion in Zeile 8 aktualisiert.

**Eingabe** : Menge der Funktionen $X$
**Ausgabe** : Menge der zu lockenden Funktionen $L$

```
   /* Start des Algorithmus                                    */
1  WCET aller Funktionen messen:  w(x) ∀x ∈ U
2  Locke alle Funktionen temporär (siehe S. 34):  T = X,  L = ∅
3  WCET aller Funktionen messen:  w_l(x) ∀x ∈ T
4  Alle Funktionen unlocken:  T = L = ∅
5  Berechne Reward jeder Funktion im Vergleich zu 2):  r(x) ∀x ∈ U
6  while  (∃x ∈ U | s(x) ≤ F_c) do
7  |  Locken des Elements mit dem größten
   |  Reward:  L = L ∪ x | max r(x)
8  |  Freien Cache aktualisieren:  F_c = F_c − s(x)
```

**Algorithmus 1** : Pseudocode des Einzelpfad-Algorithmus

Problematisch wird der Fall, bei dem ein Wechsel im Kontrollfluss durch das Locking eines Elementes stattfindet (siehe Seite 29). An dieser Stelle ist eine weitere Anwendung von Optimierungen durch den Algorithmus u.U. sinnlos, wenn die Ausführungszeit durch den neuen Ausführungspfad dominiert wird.

Die Anzahl der benötigten aiT-Aufrufe ist für den Algorithmus konstant, was zugleich den einzigen Vorteil darstellt:

$$A(n) = 2$$

Die Auswahl von Funktionen entspricht dem klassischen Rucksackproblem, für das der Algorithmus 1 immer das Element mit dem größten Nutzen (hier Reward) als erstes auswählt. Dabei wird nicht berücksichtigt, ob es Kombinationen aus Elementen mit geringerem Einzelreward gibt, die den zur Verfügung stehenden Platz besser ausnutzen und einen höheren Gesamtreward aufweisen. Die oben angeführte Lineare Programmierung findet im Gegensatz zum vorgestellten Algorithmus für den einmal ermittelten Worst-Case-Ausführungspfad immer die optimale Lösung für die Menge $L$ der zu lockenden Funktionen.

Aufgrund der bisher aufgeführten Nachteile der Einzelpfadanalyse wird der vorgestellte Algorithmus nur der Vollständigkeit halber aufgezählt und

für einen Vergleich in Kapitel 5.3 herangezogen. Dieser Vergleich stellt die Optimierung entlang des zu Beginn ermittelten längsten Ausführungspfades den in Kapitel 4.3 bis 4.5 vorgestellten Verfahren gegenüber.

## 4.3 Greedy-Algorithmus

Der Greedy-Algorithmus ist eine Erweiterung, die versucht, die Nachteile der Einzelpfadanalyse durch weitere Analysen auszugleichen. Der Fortschritt ist die Ermittlung der WCET des zu analysierenden Programms nach jedem Lockdown einer Funktion in den Cache, so dass Änderungen des Kontrollflusses bei der Auswahl der Elemente berücksichtigt werden.

Der Algorithmus beginnt damit, die in Kapitel 3.3 beschriebenen Informationen der GCC über den Übersetzungsvorgang einzulesen. Mit diesen Informationen wird eine Datenstruktur gefüllt, die Informationen über jede im Programm enthaltene Funktion speichert. Folgende Informationen werden berücksichtigt:

- **Name der Funktion**
  Der Name wird durch Auswertung der Informationen über den Linkvorgang bestimmt.
- **Startadresse**
  Die Adresse, an der die betreffende Funktion in den Speicher geladen wird, wird ebenfalls aus den Informationen über den Linkvorgang ermittelt.
- **Größe**
  Mit Hilfe der Größe der Funktion, die von arm-rtems-readelf über den Linkvorgang ausgegeben wird, wird der Reward sowie die Adresse des letzten Bytes der Funktion im Speicher berechnet.
- **WCET-Takte**
  Dies sind die CPU-Takte, die die Funktion für eine einmalige Ausführung in einem System ohne Cache benötigt. Sie werden durch den Selektionsalgorithmus mit Hilfe von aiT ermittelt.
- **Opt. WCET-Takte**
  Dies sind die CPU-Takte, die die Funktion für eine einmalige Ausführung benötigt, falls die Funktion vor ihrer Ausführung in den I-Cache geladen und gelockt wird. Die benötigten Takte werden durch den Selektionsalgorithmus mit Hilfe von aiT ermittelt.
- **Reward**
  Der Reward ist der Gewinn in eingesparten Takten pro Byte einer Funktion und wird vom Selektionsalgorithmus mit Hilfe der Gleichung 4.5 berechnet.

Der Pseudocode des Greedy-Algorithmus für die folgenden Schritte ist in Algorithmus 2 abgebildet. Zuerst wird ein Analysedurchgang ausgeführt, bei dem alle Funktionen in den Cache gelockt werden (Zeile 1 und 2). Die Messung mit komplett gelocktem Cache, also $T = X$, wird im Folgenden Referenz-Messung genannt und bleibt während des ganzen Optimierungsvorgangs erhalten[1]. In der ersten Iteration der Schleife ab Zeile 4 wird in Zeile 5 ein zweiter Analysevorgang für ein System ohne bzw. mit abgeschaltetem Cache durchgeführt. aiT berechnet nur die Laufzeiten von Elementen, die auf dem ermittelten Worst-Case-Pfad liegen, weshalb die erwähnte Datenstruktur auch nur Laufzeiten und den Reward für diese Funktionen enthält. Alle Funktionen auf nebenläufigen Kontrollfluss-Pfaden haben deshalb eine Laufzeit von Null.

**Eingabe** : Menge der Funktionen $X$
**Ausgabe** : Menge der zu lockenden Funktionen $L$

`/* Start des Algorithmus */`
1 Locke alle Funktionen temporär: $T = X,\ L = \emptyset$
2 WCET aller Funktionen messen: $w_l(x)\ \forall x \in T$
3 Alle Funktionen unlocken: $T = L = \emptyset$
4 **while** $(\exists x \in U \mid s(x) \leq F_c)$ **do**
5 WCET-Messung mit allen bisher gelockten Elementen: $w(x)\ \forall x \in U$
6 Berechne Reward jeder Funktion im Vergleich zu 2): $r(x)\ \forall x \in U$
7 Locken des Elements mit dem größten Reward: $L = L \cup y \mid max\ r(y)$
8 Freien Cache aktualisieren: $F_c = F_c - s(y)$

**Algorithmus 2** : Pseudocode des Greedy-Algorithmus

Sind die Informationen über die Laufzeit berechnet, wird zunächst für jede Funktion in Zeile 6 der Reward berechnet (siehe Gleichung 4.5), wobei Funktionen auf nebenläufigen Pfaden automatisch einen Reward von Null bekommen. Im ersten Iterationsschritt wird in Zeile 7 die Funktion $x$ mit dem größten Reward $r(x)$ in den Cache gelockt ($L = L \cup x$). Anschließend wird im zweiten Iterationsschritt wieder eine WCET-Messung mit dem einen gelockten Element durchgeführt (Zeile 5) und der Reward für das Locking jeder Funktion im Vergleich zur Referenz-Messung neu berechnet (Zeile 6). Nun wird wieder aus der Menge $U$ der ungelockten Funktionen das Element mit dem größten Reward ausgewählt und gelockt (Zeile 7). Die WCET für das Programm mit den dauerhaft gelockten Funktionen wird erneut vermessen und die Rewards berechnet. Es finden solange weitere Iterationsschritte statt, bis keine Funktion auf dem Worst-Case-Pfad mehr ungelockt ist oder

[1] Wie ein Cache simuliert wird, der groß genug ist, um eine beliebige Anzahl Funktionen zu speichern, zeigt Kapitel 5.1.

der freie Cache nicht mehr ausreicht, um weitere Funktionen zu locken (Abbruchbedingung in Zeile 4).

**Tabelle 4.1:** Ermittelte Messwerte für das Beispiel aus Abb. 4.1(a)

| Iterat. Schritt | Name | Größe | WCET | Opt.WCET | Reward |
|---|---|---|---|---|---|
| 1 | main | 16 | 10 | 6 | 0.25 |
| | a | 20 | 50 | 27 | 1.15 |
| | b | 10 | 80 | 40 | 4 |
| | c | 36 | 65 | 40 | 0.69 |
| | d | 24 | 0 | 0 | 0 |
| 2 | main | 16 | 0 | 6 | -0.375 |
| | a | 20 | 0 | 27 | -1.35 |
| | c | 36 | 0 | 40 | -1.11 |
| | d | 24 | 180 | 0 | 7.5 |

Um die Funktionsweise des Greedy-Algorithmus zu verdeutlichen, kann nun folgendes Beispiel angeführt werden: Tabelle 4.1 enthält fiktive Werte für das Beispiel aus Abb. 4.1(a), die nach der Analyse der Laufzeiten durch aiT in der Datenstruktur gespeichert sind. Anschließend beginnt der Selektionsdurchlauf für die Elemente, die gelockt werden sollen. Hier wird als erstes die Funktion $b$ gelockt, die den größten Reward aufweist, anschließend wird die WCET mit gelockter Funktion $b$ für alle ungelockten Funktionen neu berechnet. Im zweiten Iterationsschritt würde die Funktion $d$ gelockt, für die ein unberechtigt hoher Reward von 7.5 berechnet worden ist, da für die WCET der gelockten Funktion in Iterationsschritt 1 kein Wert bzw. Null ermittelt worden ist. Die WCET der Funktion $d$ ist zu Beginn deshalb Null, weil $d$ vor locken von $b$ nicht auf dem Worst-Case-Pfad liegt und aiT für Funktionen auf nebenläufigen Pfaden keine Laufzeit berechnet. Bereits im zweiten Iterationsschritt sieht man, dass einige Elemente negative Rewards aufweisen. Der Fehler entsteht, da bei der Referenzmessung die Funktionen auf dem Worst-Case-Ausführungspfad gelegen haben und eine Laufzeit größer Null ermittelt worden ist. In Iterationsschritt 2 liegen diese Funktionen aber auf einem nebenläufigen Pfad und bekommen eine Laufzeit von Null, die in die Formel 4.5 eingesetzt einen negativen Reward ergibt.

Der vorgestellte Algorithmus liefert die besten Lösungen für den Fall, dass sich der Worst-Case-Pfad auch dann nicht ändert, wenn beliebige darauf liegende Funktionen gelockt werden. In manchen Fällen wird eine Änderung des Pfades nicht immer fehlerfrei erfasst und es werden u.U. Funktionen gelockt, die durch den Wechsel des Ausführungspfads mit der längsten Laufzeit einen unberechtigt hohen Reward zugewiesen bekommen.

Dieser Umstand resultiert aus der Bewertungsfunktion $r(x)$ und den beiden dafür nötigen WCET-Messungen. Die erste Messung (Referenzmessung)

misst das System mit allen Funktionen gelockt, die zweite mit abgeschaltetem Cache. Da in vielen Fällen alle Funktionen gleichermaßen beschleunigt werden, werden für die Funktionen auf dem nebenläufigen Pfad in beiden Fällen Null Takte ermittelt. Das Locking eines Elementes auf dem Worst-Case-Pfad könnte diesen jedoch in der Laufzeit verkürzen, so dass ein alternativer Pfad jetzt länger ist. Eine Vergleichsmessung mit den bereits gelockten Elementen für die Berechnung des Reward würde nun für Funktionen auf dem alternativen Pfad eine Laufzeit größer Null ergeben, und der Reward der Funktionen auf dem neuen Worst-Case-Pfad wäre folgerichtig zu hoch angesetzt. Funktionen, die vorher auf dem Worst-Case-Pfad gelegen haben, bekämen einen negativen Reward.

Es wäre ebenfalls problematisch, wenn bei den ersten beiden überhaupt durchgeführten Messungen, also der Referenz-Messung und der Messung mit abgeschaltetem Cache, ein Wechsel des Worst-Case-Pfades stattfindet. Dies ist möglich, wenn Funktionen mit vielen Daten operieren, die weiterhin im vergleichsweise langsamen DRAM liegen. Dadurch wäre es möglich, dass beim kompletten Lockdown des Caches die Ausführungszeit eines nebenläufigen Pfades die WCET bestimmt, da Funktionen auf nebenläufigen Pfaden nicht um den gleichen Faktor beschleunigt werden wie die Funktionen auf dem Worst-Case-Pfad bei abgeschaltetem Cache. So wäre die Laufzeit einiger Funktionen bei abgeschaltetem Cache u.U. recht hoch, und bei komplett gelocktem Inhalt die ausgegebene Laufzeit gleich Null, da die Funktionen dort auf einem nebenläufigen Ausführungspfad liegen. Der Reward dieser Funktionen wäre dadurch viel besser, da die gesamte Laufzeit der ungelockten Funktion als Gewinn in Takten durch die Funktion $r(x)$ beim Lockdown interpretiert würde.

Für einen Optimierungsdurchlauf mit dem Greedy-Algorithmus sind mindestens zwei aiT-Analysen nötig, um eine Funktion zu locken. Für jede weitere der $n-1$ Funktionen ist eine weitere WCET-Analyse erforderlich, was maximal

$$\begin{aligned} A(n) &= 2 + n - 1 \\ &= n + 1 \end{aligned}$$

aiT-Aufrufe nötig macht, um die Funktionen zu bestimmen, die in den Cache gelockt werden sollen.

## 4.4 $n^2$-Ansatz

Der $n^2$-Algorithmus ist eine Weiterentwicklung des Greedy-Algorithmus, um auch Änderungen des Worst-Case-Pfades durch Locking von Elementen korrekt berücksichtigen zu können. Er geht nach dem selben einfachen Prinzip vor, immer das Element mit dem höchsten Reward zu locken, weshalb man

ihn auch in die Klasse der Greedy-Algorithmen einstufen kann. Die eigentliche Erweiterung gegenüber des Greedy-Algorithmus ist die Neuberechnung des WCET-Pfades, nachdem ein Element gelockt worden ist. Dazu sind die Reward-Berechnung abgeändert worden, indem nicht mehr das ganze Programm gelockt und anschließend vermessen wird, sondern immer nur einzelne Funktionen testweise in den Cache gelockt werden. So wird die Änderung der WCET durch das Lockdown eines Elementes für alle möglichen Kombinationen ausgetestet.

**Eingabe** : Menge der Funktionen $X$
**Ausgabe** : Menge der zu lockenden Funktionen $L$

```
   /* Start des Algorithmus                                    */
 1 T = ∅
 2 L = ∅
 3 while (∃x ∈ U | s(x) ≤ F_c) do
 4 |   WCET-Analyse (Referenzmessung) mit gelockten Funktionen: L
 5 |   forall (y ∈ U | s(y) ≤ F_c) do
 6 |   |   Locke Element y temporär: T = y
 7 |   |   WCET-Analyse mit gelockten Elementen: L ∪ T
 8 |   |   Merke WCET für temporär gelockte Funktion: w_l(y) | y ∈ T
 9 |   Berechne Reward für jede ungelockte Funktion: r(x) | x ∈ U
10 |   Dauerhaftes Locking des Elements mit besten
   |   Reward: L = L ∪ z | max r(z)
11 |   Freien Cache aktualisieren: F_c = F_c − s(z)
```

**Algorithmus 3** : Pseudocode des $n^2$-Algorithmus

Der Pseudocode für den $n^2$-Ansatz ist in Algorithmus 3 dargestellt und wird im Folgenden näher erläutert. Der Teil des Algorithmus, der für die eigentliche Auswahl zuständig ist, besteht aus zwei ineinander verschachtelten Schleifen. Die äußere in Zeile 3 startende Schleife läuft so lange durch, bis es keine Funktion mehr gibt, die ungelockt ist und in den verbleibenden Cache passt. In Zeile 4 wird das ganze Programm wie beim Greedy-Ansatz mit den bisher im Cache gelockten Funktionen analysiert und als Referenz-Messung gespeichert. In der inneren Schleife (Zeilen 5-8) wird jede Funktion $x_i$, die ungelockt ist, einzeln vorübergehend gelockt (Zeile 6), und mit Hilfe von aiT in Zeile 7 die WCET für das Programm bzw. den Worst-Case-Pfad berechnet. Der Gewinn für eine temporär gelockte Funktion in Takten, im Vergleich zur Referenzmessung aus Zeile 4, wird nach jedem Schritt gespeichert (Zeile 8). Sind alle möglichen Einsparungen der bisher ungelockten Funktionen berechnet, wird die innere Schleife abgebrochen und in Zeile 9 für alle ungelockten Funktionen der Reward berechnet. Das Element mit dem größten Reward wird dauerhaft gelockt (Zeile 10). Die Messung mit

**Tabelle 4.2:** Optimierungsdurchläufe für Programm aus Abb. 4.1(a)

| **Optimierungs-durchlauf** | **Gelockte Funktionen** | **Name** | **Gewinn [Takte]** | **Reward** |
|---|---|---|---|---|
| 1 | $\emptyset$ | main | 4 | 0.25 |
| | $\emptyset$ | a | 15 | 0.75 |
| | $\emptyset$ | b | 15 | 1.5 |
| | $\emptyset$ | c | 15 | 0.42 |
| | $\emptyset$ | d | 0 | 0 |
| 2 | $b$ | main | 0 | 0 |
| | $b$ | a | 0 | 0 |
| | $b$ | c | 0 | 0 |
| | $b$ | d | 25 | 1.04 |
| 3 | $b, d$ | main | 4 | 0.25 |
| | $b, d$ | a | 23 | 1.15 |
| | $b, d$ | c | 25 | 0.69 |
| 4 | $a, b, d$ | main | 4 | 0.25 |
| | $a, b, d$ | c | 25 | 0.69 |
| 5 | $a, b, c, d$ | main | 4 | 0.25 |
| 6 | $a, b, c, d,$ $main$ | | | |

dem nun dauerhaft gelockten Element wird jetzt als Referenz-Messung für den nächsten Durchgang zugrunde gelegt.

Mit den verbleibenden $n-1$ Funktionen wird im zweiten Optimierungsdurchlauf wieder genauso verfahren, d.h. alle Funktionen werden einzeln gelockt – zusätzlich zu der bereits dauerhaft gelockten. Anschließend wird wieder der Reward für jede Funktion berechnet, das Element mit dem größten Reward dauerhaft gelockt und dessen Messung als Referenz verwendet. Auch hier werden weitere Durchgänge gemacht, bis entweder alle Funktionen gelockt sind oder der Cache keine weiteren Funktionen mehr fasst.

Die Tabelle 4.2 zeigt die einzelnen Optimierungsdurchläufe, Messergebnisse und ausgewählten Elemente für das Beispielprogramm aus Abbildung 4.1(a). Die erste Spalte gibt den jeweiligen Optimierungsdurchlauf an, die zweite gibt die bisher zum Lockdown ausgewählten Funktionen, also die Menge $L$, an. Die Spalte mit dem Titel "Name" gibt die im jeweiligen Durchlauf temporär gelockte Funktion und damit den im Schritt aktuellen Inhalt der Menge $T$ an. Die letzten beiden Spalten enthalten die durch das temporäre Locking gewonnenen Takte sowie den entsprechenden Reward.

Im ersten Optimierungsdurchlauf bringt das Locking von Funktionen auf dem linken Ausführungspfad, also *main*, $a$, $b$ und $c$, einen Gewinn, und es wird das Element mit dem höchsten Reward (Funktion $b$) zum dauerhaften Locking ausgewählt. Ein Lockdown der Funktion $d$ bringt keinen Gewinn, da

diese auf einem nebenläufigen Ausführungspfad liegt. Beim zweiten Durchgang ist die Funktion $b$ schon gelockt, und es werden die verbleibenden Funktionen vorübergehend in den Cache geladen und gelockt. Hier kann nur die Funktion $d$ einen Gewinn erzielen, da ein Wechsel des Worst-Case-Pfades stattgefunden hat, und die Funktion wird dauerhaft in den Cache gelockt. In den Durchgängen drei, vier und fünf werden nacheinander die übrigen Elemente ihrem Reward nach absteigend zum Lockdown ausgewählt. Der Gewinn für das Locking von Funktionen kann in den einzelnen Durchgängen durchaus unterschiedlich sein, da die WCET des Programmes u.U. durch die Länge eines anderen Ausführungspfades nach unten beschränkt wird. Dann muss für weitere Optimierungen zuerst der neue Worst-Case-Pfad optimiert werden. So ein Fall tritt bei Durchlauf eins auf, wo die Laufzeit des Programms beim Locking einer der Funktionen $a, b$ oder $c$ durch die Worst-Case-Laufzeit vom Pfad $main \rightarrow d$ nach unten auf 190 Takte begrenzt wird. Deshalb ist der Gewinn in Takten der drei Funktionen auf 15 Takte beschränkt und es muss im zweiten Durchgang erst der rechte Ausführungspfad optimiert werden.

Der Vorteil des vorgestellten Verfahrens ist, dass es mögliche Wechsel des Worst-Case-Ausführungspfades berücksichtigt, da in jedem Durchgang alle bisher ungelockten Funktionen testweise einzeln gelockt und die Auswirkungen auf die Laufzeit gemessen werden. Das ist zugleich auch der Nachteil, da in jedem Durchgang alle verbleibenden Funktionen durch einen separaten aiT-Aufruf gemessen werden müssen, egal ob sie auf dem Worst-Case-Pfad liegen oder nicht. Im ersten Schritt sind also $n+1$ Aufrufe nötig, im zweiten noch $n$. Insgesamt sind also maximal

$$A(n) = (n+1) + n + \cdots + 2 \tag{4.7}$$
$$= (n+3)(n/2) \tag{4.8}$$
$$= \frac{n^2 + 3n}{2} \tag{4.9}$$

aiT-Aufrufe nötig, um die Elemente für den Lockdown-Vorgang auszuwählen. Man sieht schnell, dass bei Programmen mit 10 Funktionen bereits über 60 rechenintensive aiT-Aufrufe nötig sein können, weshalb der $n^2$ Algorithmus hauptsächlich für kleinere Probleme in vertretbarer Zeit anwendbar ist.

## 4.5 Graph-Algorithmus

Bisher sind zwei Algorithmen vorgestellt worden, die mit steigender Gesamtanzahl der Funktionen für die Auswahl von Funktionen zum Locking in den Cache verhältnismäßig viele WCET-Analysen durch aiT benötigen. Dieser Nachteil resultiert daraus, dass für die Auswahl eines Elementes zum Locking ausprobiert worden ist, welche Funktion den besten Reward bringt. Deshalb sind maximal so viele aiT-Aufrufe wie Funktionen im Programm

existieren, mindestens aber zwei, nötig. Der in diesem Abschnitt vorgestellte Graph-Algorithmus geht einen anderen Weg, so dass er die von aiT bereitgestellten Informationen besser nutzt, indem er selbst den Kontrollfluss des betrachteten Programms analysiert und immer entlang des längsten Ausführungspfades optimiert.

Dazu ist es jedoch erst einmal nötig, den Kontrollfluss des Programms aus den gegebenen Informationen zu ermitteln und in einer geeigneten Datenstruktur darzustellen. Als Datenstrukturen für alle folgenden Algorithmen werden gerichtete Graphen benutzt, die durch eine Knotenmenge $V$ und eine Kantenmenge $E$ dargestellt werden. Die Knotenmenge repräsentiert die Menge der Funktionen eines Programms und wird zu Beginn einmal erzeugt. Die Knoten speichern eine Reihe von Informationen, die z.T. beim Greedy-Algorithmus bereits Verwendung finden und noch einmal aufgelistet werden:

- **Index**
  Der Index ist der Knotenname, der den Knoten in der Menge der Knoten eindeutig identifiziert, da eine Funktion auf einem Ausführungspfad auch mehrmals aufgerufen werden kann. Er wird automatisch fortlaufend nummeriert.

- **Name**
  Der Name ist der reale Name der durch den Knoten repräsentierten Funktion, wie er vom Linker ausgegeben wird.

- **Startadresse**
  Die Adresse, an der die betreffende Funktion in den Speicher geladen wird, ist die Startadresse.

- **Size**
  Mit Hilfe der Größe wird der Reward, der verwendete Cache-Speicher, sowie die Adresse des letzten Bytes der Funktion im Speicher berechnet.

- **WCET**
  Die Variable WCET speichert die CPU-Takte, die die Funktion für eine einmalige Ausführung im Worst-Case in einem System ohne Cache benötigt.

- **OptWCET**
  Die CPU-Takte, die die Funktion für eine einmalige Ausführung im Worst-Case benötigt, falls die Funktion vor ihrer Ausführung in den I-Cache geladen und gelockt wird, werden in der Variable OptWCET gespeichert.

- **Reward**
  Der Reward ist der Gewinn in eingesparten Takten pro Byte einer

Funktion und wird vom Selektionsalgorithmus durch Gleichung 4.6 berechnet.

- **Locked**
  Hier wird gespeichert, ob eine Funktion bei einem vorhergehenden Optimierungsdurchlauf bereits in den Cache gelockt worden ist.

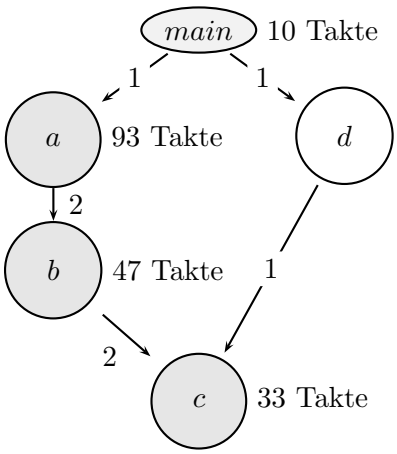
Die Kanten repräsentieren Funktionsaufrufe und werden dargestellt durch Tripel, bestehend aus zwei Knoten-Indizes und einem Gewicht. Start- und Zielknoten entsprechen Caller und Callee, das Gewicht je nach angewendetem Algorithmus den Ausführungshäufigkeiten oder einer Entfernung. Durch die Mengen der Knoten und Kanten ist es möglich, den Graphen vollständig zu beschreiben und alle zur Selektion von Elementen für das Cache-Locking benötigten Algorithmen darauf auszuführen. Bis jedoch ein Element durch die Analyse des längsten Ausführungspfades ausgewählt werden kann, muss zuerst der Callgraph des Programms generiert werden. Dieser wird dann in einen abgewandelten Kontrollflussgraphen transformiert, in dem schließlich der längste Ausführungspfad bestimmt wird. Auf diesem Worst-Case-Pfad kann dann das Element mit dem größten Reward zum Lockdown ausgewählt werden.

Wie die beschriebenen Vorgänge funktionieren, soll in den folgenden vier Abschnitten beschrieben werden. Danach werden noch auftretenden Probleme des Verfahrens besprochen und mögliche Verbesserungen vorgestellt.

### 4.5.1 Callgraph

Die Analyse eines Programms setzt Kenntnis von dessen Struktur voraus. Für die hier durchgeführten Optimierungen sind Funktionen die atomaren Elemente dieser Struktur. Wie die Funktionen eines Programms miteinander zusammenarbeiten, wird in einem so genannten *Callgraphen* dargestellt. Die Knoten des Graphen sind die einzelnen Funktionen; die gerichteten Kanten stellen dar, welche Funktion eine andere aufruft. Das Gewicht eines Knotens repräsentiert die Laufzeit der entsprechenden Funktion, und die Kantengewichte geben die Ausführungshäufigkeiten wieder. Ein Beispiel für einen einfachen Callgraphen zeigt Abbildung 4.2 zusammen mit einem möglichen Quelltext. Knoten, die auf dem längsten Ausführungspfad liegen, sind grau schattiert.

Das Beispielprogramm prüft beim Start in der *main* Funktion, ob ein Parameter mit angegeben worden ist. Ist das der Fall, so wird die Funktion $a$ aufgerufen, ansonsten $d$. Die Funktion $a$ ruft irgendwann eine weitere Funktion $b$ zweimal auf, und die wiederum ruft im Programmverlauf $c$ auf. Die andere vom Hauptprogramm aufgerufene Funktion $d$ ruft ebenfalls die Funktion $c$ in jedem Fall auf. Ob die einzelnen Pfade, die vom Hauptprogramm ausgehen, wie im Beispiel konkurrierend sind oder rein sequentiell – also nacheinander – aufgerufen werden, ist am Callgraphen nicht zu erkennen.




```
1  int main(int argc,
2       char *argv[]) {
3  if (argc > 1) a();
4    else d();
5  }
6  void a(){
7    ...
8    b();
9    b();
10   ... }
11 void b(){
12   ...
13   c();
14   ... }
15 void d(){
16   ...
17   c();
18   ... }
```

**Abbildung 4.2:** Callgraph eines Programms mit mehreren Ausführungspfaden

Dies lässt sich nur durch Analyse des Quelltextes bzw. der dazugehörigen Binärdatei klären.

Um den Callgraph eines Programms zu erzeugen, gibt es mehrere Möglichkeiten. Zum einen bietet die GCC den Kommandozeilenschalter `-dr` an, der beim Übersetzen eines Programms den für interne Optimierungen nötigen Dump des Register-Transfer-Levels ausgibt. Der erzeugte Dump enthält Informationen über Funktionsaufrufe, die mit Hilfe des Perl-Skripts `egypt` [Gus] extrahiert und für Analysezwecke weiterverarbeitet werden können. Ein entscheidender Nachteil ist jedoch, dass Funktionsaufrufe von externen Bibliotheken nicht dargestellt werden können, so dass nur der übersetzte Quelltext mit in die Analyse einfließt.

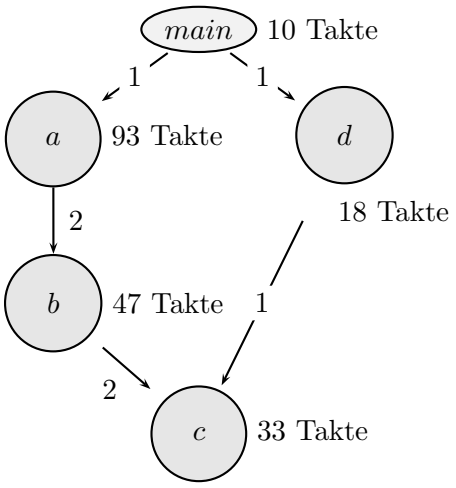
Die andere Möglichkeit ist, Funktionsaufrufe aus den von aiT durchgeführten Analysen zu extrahieren und damit den Callgraphen zu erzeugen. Für die TriCore-Version von aiT existieren Bibliotheken, die benutzt werden können, um die gewünschten Informationen aus den unterschiedlichen Schritten des Analyseprozesses auszulesen. Bei der ARM-Version bleibt nur die Möglichkeit, die Funktionsaufrufe aus der Datei mit dem von aiT erzeugten Graphen zu extrahieren (sog. *GDL* Datei, die auf Seite 16 näher beschrieben wird). Dies wird durch Skripte erledigt, die relevante Informationen durch Stringmanipulation extrahieren und in eine Datei schreiben, die dann vom Optimierungsprogramm eingelesen wird. Eine Informationseinheit, die über einen einzelnen Funktionsaufruf extrahiert werden kann, sieht z.B. so aus:

```
symbol:iszero
callfrom:0x2c14
max_count:100
max_count:64
```

Das Symbol ist die aufgerufene Funktion, die von der Hex-Adresse `0x2c14` im Arbeitsspeicher angesprungen wird. Welche Funktion der Aufrufer ist, muss anhand der Startadressen und Größen der einzelnen Funktionen aus den weiter oben erwähnten Knoteninformationen berechnet werden. Die Informationen mit dem Titel `max_count` geben die einzelnen Aufrufhäufigkeiten für unterschiedliche Kontexte an. Wie oft eine Funktion insgesamt aufgerufen wird, muss dann bei Bedarf über alle Aufrufe einer Funktion und alle Kontexte aufsummiert werden. Um den Callgraphen eines Programms aufzubauen, muss nur die Datei mit den vom Skript extrahierten Informationen eingelesen werden und die Kantenmenge mit je einer Kante pro eingelesener Informationseinheit ergänzt werden.

Da die eingelesenen Daten noch keine Informationen über die Laufzeiten der Funktionen enthalten, muss zusätzlich die Report-Datei (siehe Seite 16) von aiT ausgewertet werden. Diese Datei enthält u.a. Angaben über die Laufzeit jeder Funktion und die Gesamtlaufzeit des Programms. Die von aiT berechnete Laufzeit einer Funktion bezieht sich immer auf alle Kontexte, also die Gesamtlaufzeit aller Aufrufe durch Elemente auf dem Worst-Case-Pfad. Die Laufzeit für eine einmalige Ausführung muss daher mittels Division der Gesamtlaufzeit der betreffenden Funktion durch die Anzahl der Aufrufe der Funktion bestimmt werden. Die Anzahl der Aufrufe ist hier wieder die Summe aller Funktionsaufrufe (eingehende Kanten im Graphen) durch Funktionen auf dem längsten Ausführungspfad. Bei der Berechnung der einmaligen Worst-Case-Ausführungszeit einer Funktion wird eine gewisse Ungenauigkeit in Kauf genommen, weil Dinge wie das Füllen der Pipeline so nicht berücksichtigt werden können. Außerdem können unterschiedliche Kontexte von aiT ausgewertet werden, so dass für einzelne Kontexte die obere Grenze für die Laufzeit einer Funktion evtl. eingeschränkt werden kann. Dies findet natürlich in der Normierung auf genau eine Laufzeit für jede Funktion im Callgraphen keine Berücksichtigung. So entspricht die berechnete Laufzeit einer Funktion eher der Durchschnittslaufzeit über alle Kontexte.

Das Beispielprogramm in Abbildung 4.2 zeigt bereits die aus der Graphenbeschreibung von aiT gewonnenen Informationen über die Takte einer Funktion und die Aufrufhäufigkeiten, die an den Kanten annotiert sind. Hier ist es von Vorteil, dass aiT nur die Laufzeit von Funktionen auf dem längsten Ausführungspfad berechnet, denn so lässt sich bestimmen, welche Elemente sequentiell aufgerufen werden und wo konkurrierende Pfade entstehen. Im Beispiel ist der Worst-Case-Pfad $main \rightarrow a \rightarrow b \rightarrow c$, und ein dazu konkurrierender Ausführungspfad ist $main \rightarrow d \rightarrow c$. Weil für $d$ kei-




```
1  int main(int argc,
2        char *argv[]) {
3    a();
4    d();
5  }
6  void a(){
7    ...
8    b();
9    b();
10   ... }
11 void b(){
12   ...
13   c();
14   ... }
15 void d(){
16   ...
17   c();
18   ... }
```

**Abbildung 4.3:** Callgraph eines Programms mit einem Ausführungspfad

ne Ausführungszeit berechnet worden ist, ist das ein Zeichen, dass dort ein konkurrierender Pfad startet.

Für das nächste Beispiel wird der Quellcode des Hauptprogramms leicht modifiziert wie in Abbildung 4.3 ersichtlich wird. Dort werden bei jedem Durchlauf des Programms sowohl $a$ als auch $d$ ausgeführt, was sich im Callgraphen durch die Annotation von Laufzeiten für beide Funktionen darstellt. Nun liegen alle Elemente des Programms auf dem schlimmsten Ausführungspfad, was sich in einem Callgraphen jedoch nicht ausreichend darstellen lässt. Deshalb werden zur Darstellung von Nebenläufigkeiten die Kontrollflussgraphen des nächsten Abschnitts benutzt.

### 4.5.2 Erzeugung eines Kontrollflussgraphen

Die Laufzeit eines Programms hängt von dessen Kontrollfluss ab, der aber vor Ausführung meist noch nicht bekannt ist. Aus diesem Grund müssen alle möglichen Kontrollflüsse betrachtet werden, die dann in einem Kontrollflussgraphen (*KFG*) dargestellt werden können. Dies geschieht durch Betrachtung des Callgraphen, der bereits alle Informationen über die Funktionen und deren Abhängigkeiten enthält. Einen anderen Weg, um den KFG aus einer Binärdatei zu generieren, zeigt [The02], wohingegen [CMW00] Analysen bzgl. der Ausführungsgeschwindigkeit von Programmen direkt am Callgraphen durchführt.

Um aus dem im letzten Kapitel erzeugten Callgraphen einen KFG zu erzeugen, findet eine abgewandelte Tiefensuche (*Depth-First Search - DFS*)

auf dem Callgraphen Anwendung, die immer bei *main* startet und endet, wenn keine weiteren Elemente mehr verbleiben. Der ursprüngliche Algorithmus zur Tiefensuche in einem Graphen wird in [CLRS01] vorgestellt. Alle Funktionen, für die eine Laufzeit bestimmt worden ist, liegen auf dem Worst-Case-Pfad und werden deshalb im KFG auf einem gerichteten Pfad abgebildet. Alle Knoten ohne Laufzeit liegen auf nebenläufigen Pfaden, für die Verzweigungen im KFG eingefügt werden müssen. Der Pseudocode für die Transformation eines Callgraphen in einen Kontrollflussgraphen ist in Algorithmus 4 abgebildet.

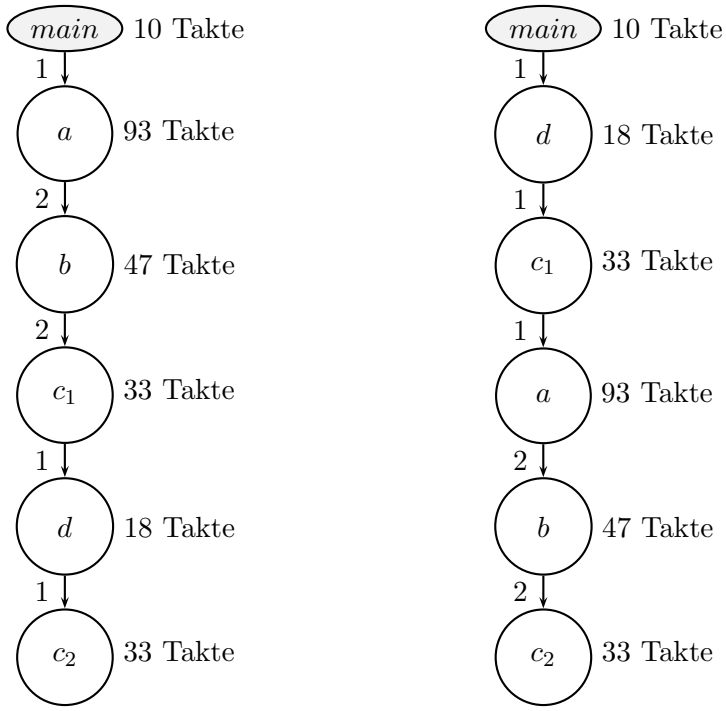
```
Eingabe : Callgraph (CG), (Sub-) Kontrollflussgraph (KFG),
          Startknoten s, Knotenindex i, LetzteKnoten lk
Ausgabe : Callgraph (CG), (Sub-) Kontrollflussgraph (KFG)
/* Start des Algorithmus                                    */
1  DFS
   (CG C, KFG G, Startknoten s, Knotenindex i, LetzteKnoten lk)
2    if G = ∅ then
3      i = G → InsertIndexedNode(s);
4      lk = (i, s);
5    if C → IstSenke(s) then
6      return (i, s);
7    forall Knoten v ∈ (C → Kind(s)) do
8      Knotenindex j = G → InsertIndexedNode(v);
9      if C → w(v) == 0 then
10       G → InsertEdge(i, j, C → g(s, v));
11       MesseWCETab(v);
12       lk = lk ∪ DFS(C, G, v, j, j);
13     else
14       forall ((Knotenindex k, Knotenname n) ∈ lk) do
15         G → InsertEdge(k, j, C → g(n, v));
16       lk = DFS(C, G, v, j, j);
17 return lk;
```

**Algorithmus 4** : Pseudocode des Tiefensuch-Algorithmus

Die Graphen, die dem DFS-Algorithmus als Parameter übergeben werden, sind der vorher erzeugte Callgraph sowie der Graph, in dem der Kontrollflussgraph erzeugt werden soll. Sowohl der Callgraph als auch der Kontrollflussgraph werden wie oben erwähnt durch die Menge $V$ der Knoten und die Menge $E$ der Kanten beschrieben. Vor dem ersten Aufruf muss der Callgraph wie in Kapitel 4.5.1 bestimmt werden, der KFG wird nicht initialisiert. Beim ersten Aufruf wird der Algorithmus mit *main* als Parameter für den

Startknoten aufgerufen. Der Rückgabewert für einen Aufruf ist die Menge der zuletzt besuchten Knoten. Im Callgraphen werden die Elemente über die dazugehörigen Funktionsnamen angesprochen, die im Kontrollflussgraphen nur als eine der Knoteneigenschaften gespeichert werden. Im KFG wird jeder Knoten über einen einmaligen Schlüssel (Knotenindex) identifiziert, da eine Funktion durchaus mehrmals auf dem selben oder auch auf unterschiedlichen Pfaden aufgerufen werden kann. Die Kantenmenge des Kontrollflussgraphen wird dementsprechend auch durch Paare von Knotenindizes repräsentiert, und daher werden beim Einfügen von neuen Kanten auch Indizes anstatt Funktionsnamen übergeben.

Nun zum eigentlichen Algorithmus: In den Zeilen 2-4 werden der KFG und die Menge der zuletzt besuchten Knoten mit dem Startknoten initialisiert, falls der Kontrollflussgraph noch keine Elemente enthält. Die Methode $InsertIndexedNode$ fügt die Funktion mit dem Namen aus $s$ in den KFG unter einem automatisch inkrementierten Index ein und liefert diesen Index als Rückgabewert. Zeile 5 überprüft, ob der Startknoten keine weiteren Kinder enthält und übergibt für diesen Fall das Paar aus den Knotenindex und dem realen Namen der Funktion $s$ als Rückgabewert für den zuletzt besuchten Knoten. Ist der Startknoten keine Senke, so wird ab Zeile 7 die Menge seiner Kinder im Callgraphen betrachtet und das jeweils betrachtete Kind auch als Knoten mit einem neuen Index in den KFG eingefügt (Zeile 8). Alle Eigenschaften des neuen Knoten wie Knotengewicht (hier die WCET-Laufzeiten), Größe etc. werden aus dem Callgraphen mit in den KFG übernommen. In der nächsten Zeile wird überprüft, ob das betrachtete Kind im Callgraphen eine Laufzeit besitzt und deshalb auf dem Worst-Case Ausführungpfad liegt. Ist das überprüfte Kind der erste Knoten auf einem alternativen Pfad (Laufzeit von Null), so wird von dem Startknoten zum Kind eine Kante als Verzweigung in den KFG eingefügt (Zeile 10). Das Gewicht der neuen Kante entspricht dem Gewicht der Kante des Callgraphen, über die das Kind besucht worden ist. Die WCET für den von dort startenden Callgraphen wird anschließend durch aiT in Zeile 11 bestimmt. Danach wird die Tiefensuche rekursiv mit dem Kind als Startknoten sowie als Menge der zuletzt besuchten Knoten aufgerufen (Zeile 12) und die Rückgabemenge zur Menge $lk$ der zuletzt besuchten Knoten hinzugefügt. Falls für das Kind jedoch eine Laufzeit ermittelt worden ist und es somit auf dem längsten Ausführungspfad liegt, so werden in den Zeilen 14 und 15 die Enden aller konkurrierenden Ausführungpfade mit dem Worst-Case-Pfad an dem neuen Knoten vereint. Dazu wird für jedes Element aus der Menge der zuletzt besuchten Knoten eine Kante zu diesem neuen Kind eingefügt, deren Gewicht wieder dem Gewicht der Kante des Callgraphen entspricht, über die das Kind besucht worden ist. Anschließend wird in Zeile 16 wieder die Tiefensuche mit dem aktuell betrachteten Kind als Startknoten sowie als Menge der zuletzt besuchten Knoten aufgerufen und der Rückgabewert als neue Menge $lk$ der zuletzt besuchten Knoten gesetzt.



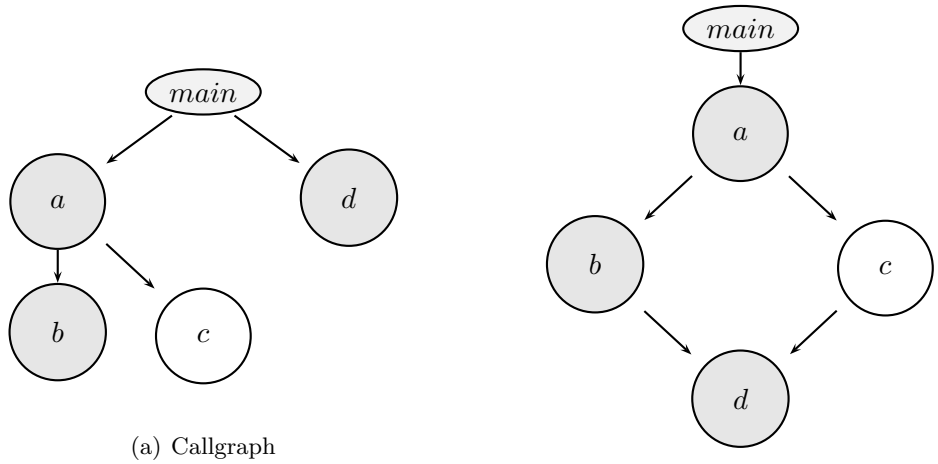

(a) Variante 1 (b) Variante 2

**Abbildung 4.4:** Kontrollflussgraph für Beispiel 4.3

Die Zeilen 9-12 behandeln also beginnende Nebenläufigkeiten, indem eine Verzweigung am aktuellen Knoten des Worst-Case-Pfades eingefügt wird, und fügen das Ende eines solchen Pfades mit in die Menge der zuletzt besuchten Knoten ein. Die Zeilen 13-16 dagegen fügen einen weiteren Knoten an den Worst-Case Ausführungspfad an und verbinden auch eventuelle Enden von nebenläufigen Pfaden mit dem neuen Knoten.

In dem erzeugten KFG können die einzelnen Pfadsegmente in falscher Reihenfolge auftreten, da die Analyse des Callgraphen keine Rückschlüsse auf die Ausführungsreihenfolge zulässt. Jedoch ist für eine WCET-Analyse die Ausführungsreihenfolge der einzelnen Knoten auf einem Pfad unerheblich, da die Länge des Pfades allein durch seine Elemente und deren Ausführungshäufigkeit bestimmt wird, und nicht durch die Reihenfolge. Die Gesamtlaufzeit eines Pfades berechnet sich aus der Summe der Laufzeiten aller darauf liegenden Knoten. Die Laufzeit eines Knotens ist die Anzahl der Aufrufe (Summe der Gewichte der eingehenden Kanten von Elementen auf dem Worst-Case-Pfad) multipliziert mit dem Knotengewicht, also der benötigten Zeit für eine einzelne Ausführung.

Eine Tiefensuche für Beispiel 4.3 erzeugt einen Kontrollflussgraphen mit einem einzelnen gerichteten Pfad, wie in Abbildung 4.4(a) zu sehen ist. Selbst




(a) Callgraph

(b) Kontrollflussgraph

**Abbildung 4.5:** Transformation von CG zu KFG

wenn die Reihenfolge der Aufrufe von $a$ und $d$ im *main* vertauscht wäre, könnte ein solcher Pfad entstehen, da der Sourcecode bei der Erzeugung des Kontrollflussgraphen nicht betrachtet wird. Man sieht schnell, dass die Länge des alternativ entstehenden Pfades $main \rightarrow d \rightarrow c_1 \rightarrow a \rightarrow b \rightarrow c_2$ wie in Abbildung 4.4(b) die gleiche ist. Der Index an den Knoten, die die Funktion $c$ repräsentieren, dient nur der Übersichtlichkeit, um die einzelnen Aufrufe besser auseinander zu halten.

In Abbildung 4.5(a) ist ein Callgraph abgebildet, wo die jeweiligen Knoten des Worst-Case-Pfades grau schattiert sind. Von Knoten $a$ startet ein nebenläufiger Pfad, dessen einziger Knoten $c$ weiß dargestellt ist. Die abgewandelte Tiefensuche, um den KFG in Abbildung 4.5(b) zu erzeugen, startet bei *main*. Es wird das linke Kind $a$ gewählt und danach dessen linkes Kind $b$, die beide als Pfad in den Kontrollflussgraphen eingefügt werden. Da $b$ keine Kinder hat, wird es der Menge der zuletzt besuchten Knoten hinzugefügt und der Durchlauf eine Ebene höher mit dem nächsten Kind von $a$ fortgesetzt. An dieser Stelle merkt der Algorithmus, dass das rechte Kind $c$ auf einem alternativen Ausführungspfad liegt, da aiT keine Laufzeit für diese Funktion berechnet hat, und fügt deshalb $c$ als weiteres Kind von $a$ in den KFG ein. Anschließend wird aiT mit $c$ als Start-Funktion erneut aufgerufen, um die Laufzeiten der Funktionen für den von dort startenden Sub-Callgraphen zu erhalten und diese in den Kontrollflussgraphen zu integrieren. Die zurückgelieferte Menge der zuletzt besuchten Knoten enthält nur $c$, das mit der Menge der zuletzt besuchten Kinder vereinigt wird zu $\{b, c\}$. Da $c$ keine Kinder und $a$ keine weiteren Kinder mehr hat, wird wieder eine Ebene höher fortgefahren. Als nächstes wird das rechte Kind $d$ von *main* besucht und als Nachfolger der zuvor gespeicherten Menge der zuletzt

besuchten Knoten eingesetzt, da es eine Laufzeit aufweist und somit auf dem längsten Ausführungspfad liegt. Die Menge der Kinder enthält dann nur noch $d$, da keine nebenläufigen Pfade mehr vorhanden sind. Da $d$ keine Kinder hat und auch keine höher liegenden Knoten mehr unbesuchte Kinder haben, ist $d$ das letzte Element und der KFG damit komplett erzeugt.

Der Algorithmus zur Tiefensuche hat eine Zeitkomplexität von

$$O(|V| + |E|)$$

die der Komplexität der normalen Tiefensuche entspricht. Allerdings werden im Worst-Case maximal

$$A(n) = 2n$$

aiT-Analysevorgänge angestoßen, wenn alle Funktionen auf zueinander nebenläufigen Pfaden liegen. Dann muss jede Funktion separat einmal in einem System ohne Cache und einmal im Cache gelockt analysiert werden.

### 4.5.3 Längster Weg

Der oben erzeugte Kontrollflussgraph spiegelt alle möglichen Ausführungspfade in einem Programm wieder. Um jetzt eine Optimierung der WCET durch Lockdown von Elementen durchzuführen, können nur Funktionen auf dem längsten Pfad gelockt werden. Deshalb gilt es, diesen längsten Pfad vor jeder Auswahl einer Funktion zum Locking in den Cache zu bestimmen.

In der Literatur gibt es viele Lösungen, um den kürzesten Weg von einem Punkt zu einem oder mehreren Zielen zu bestimmen, aber wenige, die den längsten Weg finden. Zur Bestimmung des längsten Weges im KFG von der Quelle eines Programms (i.d.R. die Funktion *main*) zu seinen Senken ist deshalb der Algorithmus von Dijkstra modifiziert worden und in Algorithmus 5 als Pseudocode dargestellt. Der Dijkstra-Algorithmus zur Berechnung der kürzesten Pfade erzeugt in seiner ursprünglichen Form einen Spannbaum, der alle kürzesten Entfernungen der Knoten zum Startknoten enthält. Die formale Definition für die Erzeugung eines Spannbaumes mit den längsten Wegen durch den abgewandelten Dijkstra lautet:

- $G$ : Ist der zu analysierende gerichtete, gewichtete KFG
- $V$ : Ist die Menge der Knoten des Graphen $G$
- $E$ : Ist die Menge der Kanten des Graphen $G$
- $g(v, w)$ : Ist eine Kostenfunktion, die das Gewicht der Kante $(v, w)$ angibt
- $L$ : Ist die Menge der gelockten Funktionen
- $s$ : Ist der Startknoten

- $Q$ : Ist die Prioritätswarteschlange der noch zu bearbeitenden Knoten. Sie speichert Paare von Knoten und deren Entfernungen zur Quelle.
- $D[v]$ : Ist ein Vektor, der die errechneten Abstände aller Knoten zu $s$ enthält
- $\Pi$ : Ist ein Spannbaum, der alle zur Laufzeit besuchten maximalen Wege von $s$ enthält

Um das gewünschte Ergebnis, den längsten Weg, zu erhalten, muss zuerst der Spannbaum $\Pi[v]$ aufgebaut werden. Anschließend wird ausgehend vom Zielknoten der längste Weg rekonstruiert. Da hier nicht bekannt ist, an welchem Knoten der längste Weg endet, werden alle Wege von den Senken zur Quelle rekonstruiert. Die mathematische Definition und Durchführung des ursprünglichen Dijkstra-Algorithmus wird in [Cor04] ausführlich beschrieben. Deshalb wird hier auf die umgesetzte Implementierung und die Änderungen für das Finden längster Wege eingegangen.

Im Kontrollflussgraphen werden die Ausführungshäufigkeiten als Kantengewichte und die Worst-Case-Laufzeit einer Funktion neben den Knoten annotiert. Die Worst-Case-Laufzeit einer Funktion $x$ ist abhängig davon, ob die Funktion gelockt ist, also $x \in L$, oder nicht. Während der Durchführung des Dijkstra-Algorithmus wird in dem Entfernungsvektor $D[v]$ für jeden Knoten seine maximale Entfernung zur Quelle, also die maximale Ausführungszeit der bisher ermittelten Wege zu diesem Knoten, annotiert. Im erzeugten Spannbaum werden nur Kantengewichte benutzt, die die Worst-Case-Laufzeit des Weges zum Zielknoten über die entsprechende Kante angeben. Da die Laufzeit einer Funktion von der Menge $L$ abhängig ist, bestimmt diese Menge der gelockten Funktionen auch die Länge der Wege. Wie der Algorithmus funktioniert, wird nun anhand des Pseudocodes erklärt:

### Initialisierung

Der Aufbau des Spannbaums beginnt damit, dass die Prioritätswarteschlange mit dem Paar $(main, w_l(main))$ initialisiert wird (Zeile 4-6), falls $main$ in den Cache gelockt worden ist, ansonsten mit $(main, w(main))$. Dieses Paar besteht aus dem Start-Knoten und der Worst-Case-Laufzeit der Quelle als Entfernung zu sich selbst. Außerdem wird der Spannbaum in Zeile 7 mit dem ersten Element, der Quelle $main$, initialisiert. Nach der Initialisierung wird iterativ verfahren, indem immer das erste Element der Schlange $Q$ betrachtet und entfernt wird, bis $Q$ keine weiteren Elemente mehr enthält.

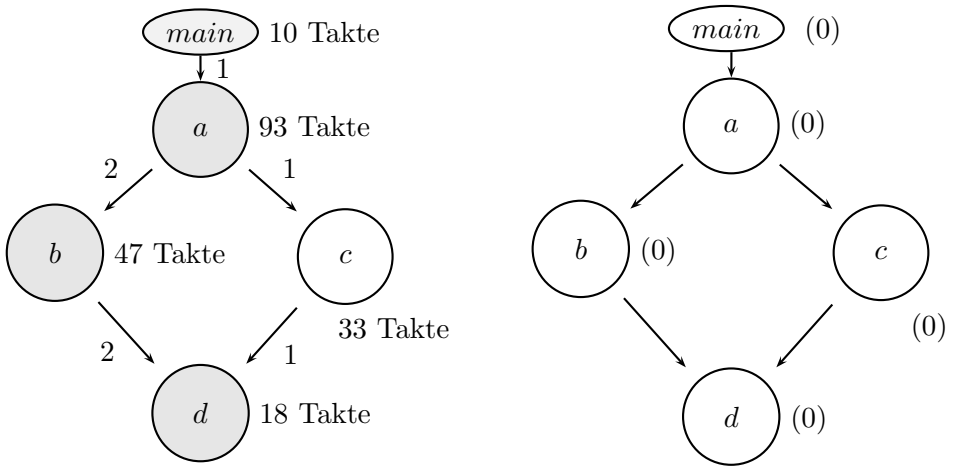
**Eingabe** : Kontrollflussgraph $G$: Knotenmenge $V$, Kantenmenge $E$;
Menge der gelockten Funktionen $L$, Startknoten $s$
**Ausgabe** : Spannbaum $\Pi$, Abstandsvektor $D[v]$

```
   /* Start des Algorithmus                                          */
 1 Dijkstra (KFG G, SpB Π, AV D[v], Start s)
 2     Queue : Q;
 3     Pair[Name, Abstand] : this, next;
 4     this.Name = s;
 5     this.Abstand = (this.Name ∈ L) ? w_l(s) : w(s);
 6     Q.push(this);
 7     Π → InsertNode(s);
 8     while Q != ∅ do
 9         this = Q.pop;
10         D[this.Name] = max(D[this.Name], this.Abstand);
11         forall v ∈ (G → V) : (this.Name, v) ∈ (G → E) do
12             next.Name = v;
13             if (next.Name ∈ L) then
14                 next.Abstand = G → g(this.Name, next.Name) *
                   w_l(next.Name) + D[this.Name];
15             else
16                 next.Abstand = G → g(this.Name, next.Name) *
                   w(next.Name) + D[this.Name];
17             If not (Π → HasEdge(next.Name, this.Name)) then
18                 Q.push(next);
19                 Π → InsertNode(next.Name);
                   /* Rückwärtskante in Spannbaum einfügen       */
20                 Π → InsertEdge(next.Name, this.Name);
21                 Π → g(next.Name, this.Name) = next.Abstand;
```

**Algorithmus 5** : Pseudocode des abgewandelten Dijkstra

### Iterationsschritt

In jedem Iterationsschritt wird immer das erste Paar der Warteschlange $Q$ mit dem Knoten *this* und seiner Entfernung *this.Abstand* zur Quelle geholt (Zeile 9). In der darauf folgenden Zeile wird die Entfernung des geholten Knotens über den neu gefundenen Weg aktualisiert, falls dieser länger ist. Hier wird im Gegensatz zum originalen Dijkstra die Entfernung für einen Ort zur Quelle durch die größte gefundene Entfernung für diesen Ort zur Quelle ersetzt. Für alle Knoten $v$, die mit *this* durch ausgehende Kanten in $G$ verbunden sind (Zeile 11), werden separat folgende Arbeiten durchgeführt, in denen die Instanz des Knotens $v$ als *next* angesprochen wird (Zeile 12):




(a) Kontrollflussgraph

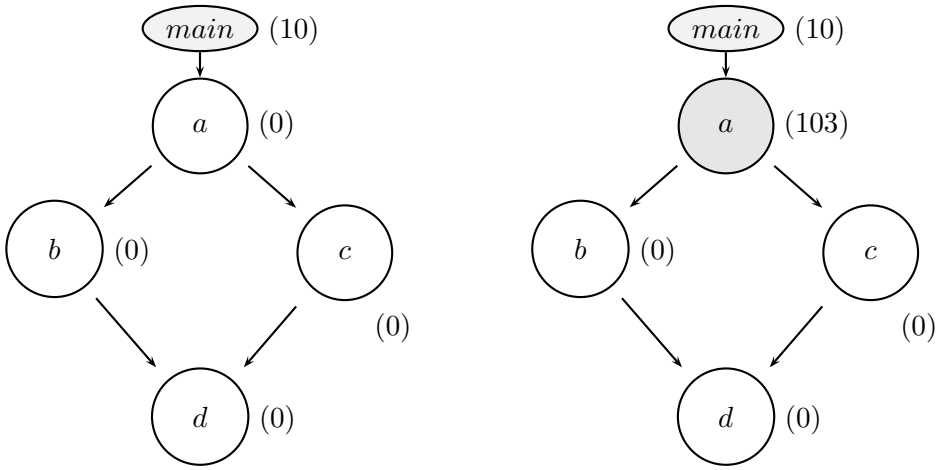
(b) Dijkstra Initialisierung

**Abbildung 4.6:** Dijkstra: Einfacher KFG und Initialisierung

- Berechne die Entfernung vom Knoten *next* zur Quelle über den Knoten *this* durch Multiplikation des Kantengewichts $G \rightarrow g(this, next)$ mit der einmaligen Ausführungszeit $w(next)$ der Funktion bzw. der gelockten Funktion $w_l(next)$ und Addition des Abstandes von *this* zur Wurzel (Zeile 13-16). Das Kantengewicht entspricht der Ausführungshäufigkeit der Funktion.
- In Zeile 17 wird geprüft, ob die in G gefundene Kante in $\Pi$ bereits exisitert, um Zyklen in $G$ nicht zu folgen und führt ansonsten folgende Arbeiten durch:
  - Zeile 18 fügt das Paar *next* ans Ende von $Q$ an
  - Füge einen neuen Knoten $next.Name$ in $\Pi$ ein (Zeile 19)
  - Zeile 20 fügt eine neue Kante $(next, this)$ in $\Pi$ ein
  - Die letzte Zeile setzt das Kantengewicht auf $next.Abstand$, also die Entfernung von *next* über *this* zur Quelle

Jeder Iterationsschritt holt also ein Element $x$ aus der Warteschlange und aktualisiert dessen Entfernung zur Quelle in $D[x]$, falls der neue Weg länger ist. Danach werden alle von dort aus direkt erreichbaren Knoten $v$ betrachtet und deren Entfernung zur Quelle über $x$ zu $v$ berechnet. Die Entfernung des Vektors $D[v]$ speichert die Entfernung über den längsten gefundenen Weg zu $v$. Jeder besuchte Knoten wird zusammen mit einer anders herum gerichteten Kante $(v \rightarrow x)$ in den Spannbaum eingefügt.

Das Beispiel in den Abbildungen 4.6(a) bis 4.10 zeigt ein Programm mit einem einfachen Kontrollflussgraphen, auf den der Dijkstra-Algorithmus angewendet wird. Abbildung 4.6(a) zeigt den KFG des Programms, Abbildung

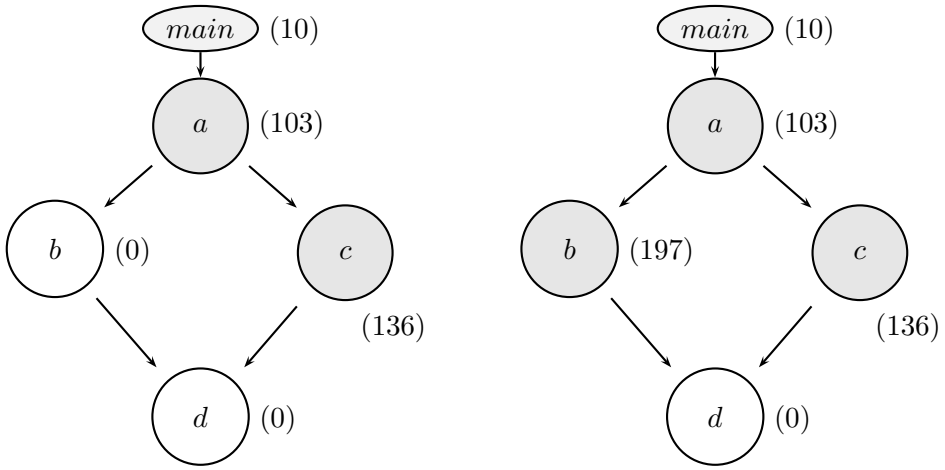



(a) Schritt 1 (b) Schritt 2

**Abbildung 4.7:** Dijkstra: Schritte 1 und 2

4.6(b) zeigt den Graphen nach der Initialisierung mit den Werten des Vektors $D[v]$ annotiert an die jeweiligen Knoten. Der Spannbaum enthält nach der Initialisierung nur die Quelle $s = main$, die auch zusammen mit der Laufzeit 10 als erstes Paar in die Warteschlange $Q$ geschrieben wird. Die Entfernungen $D[v]$ zur Quelle sind zu Beginn alle noch unbekannt und daher Null. Die folgenden Iterationsschritte sind noch einmal in Tabelle 4.3 übersichtlich zusammengefasst.

Im ersten Iterationsschritt (Bild 4.7(a)) wird das erste Element ($main$) aus der Warteschlange geholt und dessen Entfernung $D[main] = 10$ aktualisiert. Für das Kind $a$ wird die Entfernung $d(a) = 10 + 1 * 93$ berechnet und das Paar $(a, d(a))$ in die Warteschlange gestellt. Im zweiten Schritt wird das Paar wieder aus $Q$ geholt und dessen Entfernung $D[a] = d(a)$ zur Quelle gesetzt (Abbildung 4.7(b)). Außerdem wird für beide Kinder $c$ und $b$ die Entfernung $d(c) = D[a] + 1 * 33 = 136$ bzw. $d(b) = D[a] + 2 * 47 = 197$ von der Quelle über $a$ berechnet und die Paare in die Warteschlange geschrieben. Der dritte Schritt holt das rechte Kind $c$ von $a$ mit dessen Entfernung $d(c)$ wieder aus $Q$ wie in Abbildung 4.8(a) gezeigt. Die Entfernung von $c$ wird im Vektor durch $D[c] = d(c) = 136$ gesetzt, sowie für dessen Kind $d$ die Entfernung zur Quelle $d(d) = D[c] + 1 * 18 = 154$ berechnet und in die Schlange geschrieben. Im vierten Schritt wird das zweite Kind $b$ von $a$ aus der Schlange geholt und dessen Entfernung $D[b] = d(b)$ gesetzt und in Abbildung 4.8(b) aktualisiert. Hier wird wieder für das einzige Kind $d$ dessen Entfernung zur Quelle $(d(d) = D[b] + 2 * 18 = 233)$ über $b$ berechnet und das Paar in $Q$ geschrieben, so dass der Knoten zwei mal mit unterschiedlichen Entfernungen in der Warteschlange steht. Im fünften Iterationsschritt wird das Paar $(d, d(d))$ aus $Q$ geholt und die Entfernung $D[d] = d(d) = 154$




(a) Schritt 3 (b) Schritt 4

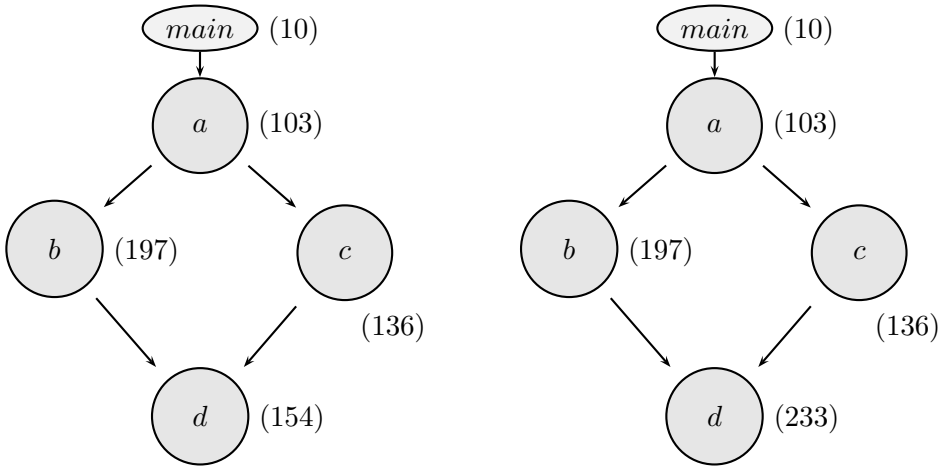
**Abbildung 4.8:** Dijkstra: Schritte 3 und 4

eingetragen (Abbildung 4.9(a)). Der Knoten $d$ aus Abbildung 4.9(b) wird im sechsten und letzten Schritt aus der Warteschlange geholt – diesmal als Weg über den Knoten $b$. Der dafür benutzte Weg $main \rightarrow a \rightarrow b \rightarrow d$ ist länger als der bisher bekannte $main \rightarrow a \rightarrow c \rightarrow d$, deshalb wird dessen Entfernung $D[d] = d(b) = 233$ als größte Wegkosten zu $d$ eingetragen.

| **Iterations-schritt** | **Akt. Knoten** | **Q** | **D[main]** | **D[a]** | **D[b]** | **D[c]** | **D[d]** |
|---|---|---|---|---|---|---|---|
| Init | - | $(main, 10)$ | 0 | 0 | 0 | 0 | 0 |
| 1 | $main$ | $(a, 103)$ | 10 | 0 | 0 | 0 | 0 |
| 2 | $a$ | $(c, 136)$, $(b, 197)$ | 10 | 103 | 0 | 0 | 0 |
| 3 | $c$ | $(b, 197)$, $(d, 154)$ | 10 | 103 | 0 | 136 | 0 |
| 4 | $b$ | $(d, 154)$, $(d, 233)$ | 10 | 103 | 197 | 136 | 0 |
| 5 | $d$ | $(d, 233)$ | 10 | 103 | 197 | 136 | 154 |
| 6 | $d$ | $\emptyset$ | 10 | 103 | 197 | 136 | 233 |

**Tabelle 4.3:** Iterationsschritte für den Dijkstra-Algorithmus

Man sieht bereits, dass die Kantengewichte im Spannbaum den Laufzeiten der einzelnen Ausführungspfade einschließlich der Zielknoten entsprechen. Dies wird dadurch erreicht, dass die Berechnung einer Kante $(v, x)$ bereits als Summe der Entfernung von $v$ zur Wurzel und dem Produkt der Ausführungshäufigkeit des Zielknotens $x$ mit seiner Laufzeit bei einmaliger Ausführung definiert worden ist. Für eine Analyse des längsten Ausführungs-




(a) Schritt 5 (b) Schritt 6

**Abbildung 4.9:** Dijkstra: Schritte 5 und 6

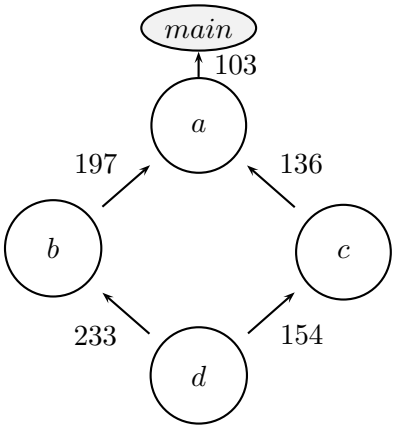



**Abbildung 4.10:** Dijkstra: Resultierender Spannbaum

pfades in Kapitel 4.5.4 muss dieser erst durch den folgenden Algorithmus anhand des gewonnenen Spannbaums bestimmt werden:

**ReconstructLongestPath**

Nachdem der Spannbaum mit allen längsten Wegen von der Quelle *main* zu sämtlichen Senken erzeugt worden ist, muss der insgesamt längste Weg gefunden werden. Aufgrund der geänderten Richtung der Kanten im Spannbaum ist *main* die einzige Senke, und die Senken aus dem Kontrollflussgraphen $G$ sind nun die Quellen. Da die Gesamtlaufzeit ausgehend von *main* an jede Kante annotiert worden ist, müssen nur die ausgehenden Kanten von den ehemaligen Senken betrachtet werden. Algorithmus 6 zeigt den Pseudocode, um den längsten Pfad als Liste von Knoten aus dem Spannbaum $\Pi$ zu gewinnen, der vom abgewandelten Dijkstra erzeugt worden ist.

Der längste Weg startet demnach an der Quelle mit dem größten Kantengewicht, dessen Kante in Zeile 2 bestimmt wird. Von dieser Quelle ausgehend, die in Zeile 3 der Knotenliste als erstes Element hinzugefügt wird, muss jetzt dem Weg mit dem jeweils größten Kantengewicht bis zur Senke *main* gefolgt werden. Dazu wird immer der zuletzt besuchte Knoten in *last* gespeichert (Zeile 4 für den ersten Knoten). Die Schleife ab Zeile 5 bestimmt die ausgehende Kante vom Knoten *last* mit dem größten Kantengewicht, fügt den direkten Nachfolger der Kante zur Liste der Knoten auf dem längsten Weg hinzu und speichert den neu gefundenen Knoten als zuletzt besuchten (Zeile 7). Die Schleife läuft so lange durch, bis keine Kante mehr vom zuletzt besuchten Knoten ausgeht. Das ist genau dann der Fall, wenn *main* der zuletzt besuchte Knoten ist, da *main* zugleich die einzige Senke ist. Alle auf dem Weg besuchten Knoten ergeben zusammen den Worst-Case-Pfad, der nun zur Optimierung herangezogen werden kann.

**Eingabe** : Spannbaum $\Pi$: Knotenmenge $V$, Kantenmenge $E$
**Ausgabe** : Knotenliste $kn$

```
   /* Start des Algorithmus                                    */
1  ReconstructLongestPath (SpB Π, Knotenliste kn)
2      for (v, w) ∈ (Π → E) | max g(v, w) do
3          kn = v;
4          last = v;
5      while ∃(last, x) ∈ (Π → E) | max g(last, x) do
6          kn = kn + {x};
7          last = x;
```

**Algorithmus 6** : Pseudocode des ReconstructLongestPath-Algorithmus

Für den Spannbaum aus Abbildung 4.10 ist die Quelle dem größten ausgehenden Kantengewicht zugleich die einzige Quelle $d$. Das Kantengewicht

von $(d, b)$ beträgt 233, und somit ist bereits die Länge des Worst-Case-Pfades bekannt. Dieser geht von $d$ zu $b$ und über die einzige ausgehende Kante von $b$ zu $a$ weiter. Dort führt die einzige Kante wieder zu $main$, und der längste Ausführungspfad $d \rightarrow b \rightarrow a \rightarrow main$ ist gefunden.

### 4.5.4 Optimierung

Der als Pseudocode notierte Algorithmus 7 ist für die Optimierungen durch das Lockdown von Funktionen in den Cache zuständig. Er benutzt alle in Kapitel 4.5 bisher eingeführten Algorithmen, deren Zusammenspiel im Folgenden erklärt wird. Die Zeilen 2-6 führen Variablen ein für den Callgraphen, den Kontrollflussgraphen, den Spannbaum, den Abstandsvektor und die Knotenliste für den längsten Weg. Der Callgraph wird bereits in Zeile 2 durch den in Kapitel 4.5.1 beschriebenen Algorithmus erzeugt. Nachdem der Callgraph erzeugt worden ist, wird in Zeile 8 durch die abgewandelte Tiefensuche aus Kapitel 4.5.2 der Kontrollflussgraph erzeugt. An dieser Stelle fallen zusammen mit den beiden zur Erzeugung des Callgraphen nötigen Aufrufen bereits alle nötigen WCET-Analysen durch aiT an. Zeile 8 benutzt die vorgestellten Methoden für den Graph-Algorithmus, um den Spannbaum mit Hilfe des abgewandelten Dijkstra aus dem Kontrollflussgraphen zu erzeugen und darin den längsten Weg zu bestimmen (Zeile 9). Wenn der Worst-Case-Pfad bestimmt ist, wird in Zeile 10 getestet, ob es überhaupt noch eine Funktion auf dem Pfad gibt, die nicht gelockt ist und in den freien Cache passt. Da eine Funktion wie in Abbildung 4.4(a) auf dem Pfad mehr als einmal auftreten kann, müssen zuerst alle Ausführungshäufigkeiten jeder ungelockten Funktion über den gesamten Pfad aufsummiert werden (Zeile 11-13). Da jede Kante auf dem längsten Pfad $kn$ mit einer Laufzeit versehen ist, wird zur Bestimmung der Ausführunghäufigkeiten die äquivalente Kante im Kontrollflussgraphen betrachtet, die die Anzahl der Aufrufe einer Funktion angibt. Zeile 12 summiert die Anzahl der Aufrufe für jede Funktion auf dem Pfad auf, falls die Funktionsnamen der Knoten übereinstimmen. Sind alle Ausführungshäufigkeiten bekannt, kann der Reward durch Gleichung 4.6 mit Hilfe der in den Knoten gespeicherten Informationen zur Worst-Case-Laufzeit berechnet werden, was in Zeile 13 geschieht. Tabelle 4.4 zeigt die Berechnung der Rewards für das Beispiel aus Abbildung 4.4(a).

Wenn die Rewards aller Funktionen auf dem längsten Ausführungspfad bekannt sind, muss nur noch das Element mit dem größten Reward aus der Liste ausgewählt werden (Zeile 14). Die gewählte Funktion wird in Zeile 15 gelockt und anschließend der freie Cache aktualisiert (Zeile 16). Die Berechnung des längsten Pfades des Kontrollflussgraphen muss in Zeile 17-18 erneut ausgeführt werden, bevor ein weiteres Element gelockt werden kann. Durch das Locking einer Funktion könnte sonst der optimierte Ausführungspfad so in der Laufzeit verkürzt sein, dass ein Wechsel des Worst-Case-Pfades stattfindet. Die drei Schritte

| Funktion | Ausführungsh. | Größe | WCET | Opt. WCET | Reward |
|---|---|---|---|---|---|
| main | 1 | 16 | 10 | 6 | 0.25 |
| a | 1 | 20 | 93 | 37 | 2.8 |
| b | 2 | 10 | 47 | 31 | 3.2 |
| c | 3 | 13 | 33 | 20 | 3 |
| d | 1 | 12 | 18 | 16 | 0.17 |

**Tabelle 4.4:** Rewardberechnung für den KFG aus Abbildung 4.4(a)

- Berechnung des längsten Pfades
- Berechnung der Rewards von ungelockten Funktionen auf dem Pfad
- Lockdown eines Elementes

können so oft wiederholt werden, bis entweder alle Funktionen gelockt sind oder der I-Cache keinen Platz mehr für weitere Funktionen bietet.

**Eingabe** : Menge der Funktionen $X$
**Ausgabe** : Menge der zu lockenden Funktionen $L$

```
   /* Start des Algorithmus                                          */
 1 Optimize (X, L)
 2     Callgraph: C =CreateCG();
 3     Kontrollflussgraph: G = Null;
 4     Spannbaum: Π = Null;
 5     Abstandsvektor: D[v];
 6     Knotenliste: kn;
 7     DFS(C, G, "main", 0, "main");
 8     Dijkstra (G, Π, D[v], "main");
 9     ReconstructLongestPath (Π, kn);
10     while (∃v ∈ kn | (v ∈ U) ∧ (s(v) ≤ F_c) do
11         forall x ∈ U do
12             a(x) = Σ(G → g(y, z)) | {y, z} ∈ kn ∧ x.Name = y.Name;
13             Berechne r_A(x);
14         for (x ∈ U | max r_A(x)) do
15             Locke Funktion x (U = U \ x ∧ L = L ∪ x);
16             F_c = F_c − s(x);
17         Dijkstra (G, Π, D[v], "main");
18         ReconstructLongestPath (Π, kn);
```

**Algorithmus 7** : Pseudocode des Graph-Algorithmus

Für die Erzeugung des Callgraphen mit den Worst-Case Execution Times für ein System ohne Cache und ein komplett gelocktes Programm sind genau

2 aiT-Aufrufe notwendig. Für die Erzeugung des Kontrollflussgraphen sind im Worst-Case für jede Funktion außer *main* maximal 2 weitere WCET-Analysen durch aiT nötig, so dass ingesamt maximal

$$\begin{aligned} A(n) &= 2 + 2(n-1) \\ &= 2n \end{aligned}$$

aiT-Aufrufe nötig sind. Die maximale Anzahl an Aufrufen wird nur in dem Fall nötig, dass das analysierte Programm $n$ verschiedene Ausführungspfade mit maximal zwei Knoten hat (*main* und ein Kind). In der Praxis ist so ein Kontrollfluss nur selten gegeben – selbst der komplexe MPEG-Benchmark aus Kapitel 5.2.5 benötigt nur 16 aiT-Aufrufe, was auf 8 konkurrierende Kontrollflüsse schließen lässt. Im Gegensatz zu den bisher vorgestellten Selektionsalgorithmen sind nach der Erzeugung des KFG keine weiteren aiT-Analysen notwendig, um Funktionen zu Lockdown auszuwählen.

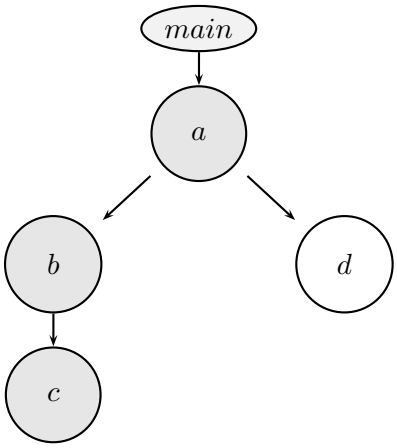
### 4.5.5 Mögliche Probleme

Es ist bereits erwähnt worden, dass aus dem Analysevorgang von aiT zur Berechnung der WCET eines Programms in der aktuellen Version lediglich die Ausführungszeit einer Funktion ausgelesen werden kann. Je nach Struktur des Programms bzw. des Kontrollflusses innerhalb einer Funktion kann ein Locking einen nicht vorhersehbaren Wechsel im Kontrollfluss verursachen.

Besitzt eine Funktion selbst nebenläufige Kontrollflüsse, so können diese momentan nicht erfasst werden, da eine Funktion immer als Blackbox betrachtet wird. Lockt man eine Funktion, so kann innerhalb derselben ein Wechsel im Kontrollfluss stattfinden. Dieser Wechsel wird i.d.R. bedingt durch einen von einem nebenläufigen Kontrollflusspfad ausgehenden Aufruf weiterer Funktionen.

Bei dem Beispiel in Abbildung 4.11 sieht es zunächst so aus, als sei der linke Ausführungspfad $main \rightarrow a \rightarrow b \rightarrow c$ aufgrund der Laufzeit der Funktionen $b$ und $c$ länger als der rechte, der mit $d$ endet. Es ist aber durchaus möglich, dass die Gesamtlaufzeit von $b$ und $c$ geringer ist als die von $d$, dann ist die Laufzeit der $for$-Schleife in Funktion $a$ entscheidend für den längsten Ausführungspfad. Wenn die Worst-Case-Laufzeit von $b + c + for$-Schleife nur wenig größer ist als die von $d$, dann würde ein Locking von $a$ einen Wechsel des Worst-Case-Pfades verursachen, und der längste Pfad wäre $main \rightarrow a \rightarrow d$ . Solch ein Wechsel würde von dem Graph-Algorithmen nicht erfasst werden, und evtl. werden dadurch Funktionen auf nicht kritischen Teilen des Kontrollflussgraphen gelockt.

Eine ganz ähnliche Situation durch konkurrierende Ausführungspfade innerhalb einer Funktion ergibt sich in Bild 4.12. Eine ähnliche Schleife wie im letzten Beispiel hat hier die Funktion $b$. So kann die Laufzeit des längsten



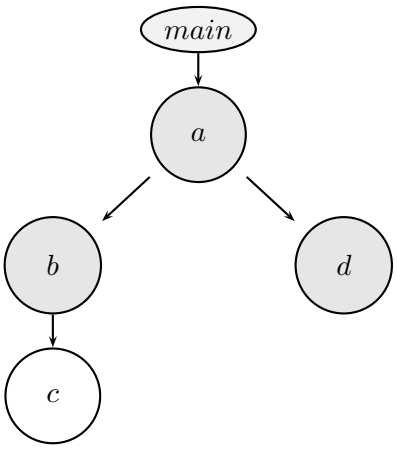

```
1  void a(int count)
2  {
3    if (count > 100)
4    {
5      for (int i=0;
6           i < 1000; i++)
7        {
8          ...
9        }
10     b();
11   } else
12   d();
13   ...
14 }
```

**Abbildung 4.11:** Kontrollflussgraph und Quellcode einer Funktion eines problematischen Programms

Kontrollflusspfades innerhalb der Funktion so groß sein, dass selbst durch ein Locking der Funktion der alternative Pfad mit Element $c$ nie der längste ist. Ein solcher Callgraph wird in den Kontrollflussgraphen in Bild 4.13 transformiert.

Durch den entstehenden Umweg im KFG von $b$ über $c$ nach $d$ wird die Funktion $c$ relevant bei der Suche nach den längsten Pfaden. Normalerweise wäre es gar nicht nötig, die Funktion zu locken, da sie in keinem Fall auf dem realen Worst-Case-Pfad liegt.

Die erwähnten Probleme entstehen durch die beschränkte Möglichkeit, die WCET-Analysevorgänge der verwendeten aiT-Version für die ARM-Familie auszuwerten. Verbesserungen können aber nur mit Bibliotheken implementiert werden, die Zugriff auf innere Zustände während der Analyse ermöglichen, wie sie für die TriCore-Version von aiT verfügbar sind. Oben genannte Konstellationen sind in der Praxis eher selten, so dass die Nachteile für die verwendeten Benchmarks an keiner Stelle zu weniger guten Optimierungen führen.




```
1  void b(int count)
2  {
3    if (count > 100)
4      for (int i=0;
5           i < 1000; i++)
6        {
7          ...
8        }
9    else
10   c();
11   ...
12 }
```

**Abbildung 4.12:** Problematischer Callgraph und Quellcode der betreffenden Funktion

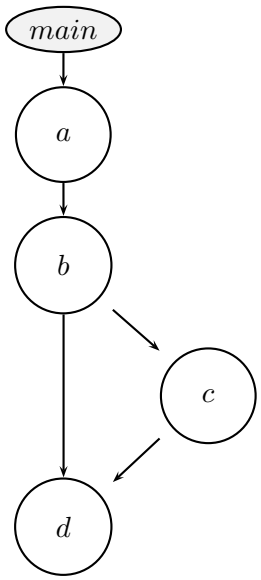



**Abbildung 4.13:** Kontrollflussgraph für Beispiel 4.12

# Kapitel 5

# Ergebnisse und Benchmarks

Um den Gewinn von Optimierungen zu testen, egal ob Energie, Speicher oder Laufzeit eingespart werden soll, benötigt man geeignete Benchmarks. Die möglichst praxisnahen Benchmarks sollen zusätzlich die korrekte Funktion der Optimierungs-Algorithmen verifizieren. In der Regel benutzt man also ein Programm, das dem späteren Einsatzzweck der zu optimierenden Programme ähnelt, und misst die benötigte Energie, die Programmgröße oder Ausführungszeit des Programms durch geeignete Werkzeuge. Für die Programmgröße und die damit erzielten Einsparungen an Speicher lässt sich dies noch relativ einfach bewerkstelligen. Die benötigte Energie bei einer Programmausführung ist stark vom Zielsystem abhängig und lässt sich ohne reale Hardware nur abschätzen. Hierfür existieren jedoch bereits gut gepflegte Datenbanken für diverse Systeme und viele verwendete Befehle, so dass man auf Erfahrungen und Messungen zurückgreifen kann.

Um die im vorigen Kapitel entwickelten Algorithmen auf deren Sparpotential zu testen, benötigt man geeignete Softwarewerkzeuge, die die Laufzeit des optimierten Programms messen bzw. berechnen. Obwohl die Algorithmen für die Optimierung der Worst-Case Execution Time entwickelt worden sind, wird die durchschnittliche Laufzeit für einen Satz an Eingabedaten (*Average-Case Execution Time - ACET*) für alle Benchmarks ebenfalls gemessen. Dies ermöglicht den Vergleich, ob Optimierungen, die speziell für die WCET entwickelt worden sind, die durchschnittliche Laufzeit von Programmen in selbem Maße verbessern.

In Kapitel 5.1 wird zunächst auf die verwendeten Werkzeuge zur Messung der ACET und WCET eingegangen und erläutert, welche Besonderheiten bei Systemen mit gelockten Caches beachtet werden müssen. Kapitel 5.2 stellt die verwendeten Benchmarks vor und gibt einen Überblick über die erzielten Ergebnisse der Optimierungen durch statisches Cache Locking bzgl. der Ausführungszeiten. Das darauf folgende Kapitel 5.3 vergleicht die hier vorgestellten Ansätze mit anderen Selektionsverfahren, um bestimmte Elemente für eine Optimierung auszuwählen. Abschließend vergleicht Kapitel 5.4 noch

die auf dem Host-System benötigte Zeit, um die in Kapitel 4 vorgestellten Selektionsalgorithmen auszuführen und ein optimiertes Binärprogramm für das Zielsystem zu generieren.

## 5.1 ACET/WCET Benchmark-Grundlagen

Die Werkzeuge für die Messung bzw. Analyse der Ausführungszeiten von Programmen auf dem verwendeten Zielsystem sind bereits in Kapitel 2.2.2 vorgestellt worden. An dieser Stelle wird noch auf die Besonderheiten bei der Verwendung von Systemen mit gelockten Caches eingegangen, da aiT die Analyse von solchen Systemen in der aktuellen Version nicht oder nur auf Umwegen unterstützt. Zunächst wird die Messung der ACET vorgestellt:

### ACET-Messungen

Der bereits vorgestellte ARM Sourcelevel Debugger ist ein Werkzeug, das sich für Simulationsläufe von Programmen genauso eignet wie für das Debuggen von Programmen. Dafür benötigt er kein reales Zielsystem, da alle verwendeten CPU-Komponenten in Software auf dem Host-System emuliert werden. Von der verwendeten Version 4.6.0 werden alle Prozessoren der ARM6, ARM7, der selten anzutreffenden ARM8 und der ARM9 Familie sowie die StrongARM CPUs unterstützt. Da die ARM920T Prozessoren mit eingeschlossen sind, wird sogar die Simulation von Systemen mit Caches unterstützt, in die Daten und Instruktionen geladen und anschließend gelockt werden können. Die Definition eigener Cache-Modelle sowie die fertigen Implementierungen einiger unterstützter Prozessoren beschreibt [Adv98e].

Wie bei jedem Debugger wird das zu analysierende Programm geladen, bevor man es zum Debuggen laufen lassen kann. Es ist möglich, beliebige Breakpoints zu setzen und an jedem dieser Breakpoints Statistiken durch den Befehl `print $statistics` ausgeben zu lassen, die folgende Informationen bereitstellen:

- **Instructions:**
  Anzahl der ausgeführten Instruktionen
- **Core_Cycles:**
  Anzahl der für die Ausführung der Instruktionen benötigten Takte
- **S_Cycles:**
  Anzahl der sequentiellen Zugriffe auf eine Adresse bzw. aufeinanderfolgende Worte im Speicher
- **N_Cycles:**
  Anzahl der nicht aufeinanderfolgende Zugriffe auf den Speicher

- **C_Cycles:**
  Anzahl der Takte des Coprozessors
- **Wait_States:**
  Anzahl der Takte, bei denen die CPU auf den Abschluss von Speicherzugriffen wartet

Die für eine Laufzeitmessung interessanten Informationen sind die Takte, die von der CPU benötigt werden, um die Instruktionen auszuführen und die Takte, die die CPU zusätzlich auf Speicherzugriffe warten muss. Durch den automatischen Instruction-Fetch tauchen Instruktionen, die in einem Prozessortakt geholt werden können, nicht in den Wartezyklen auf. Also verursacht eine Lese- oder Schreiboperation bei einem DRAM mit vier Waitstates (engl. für Wartezyklen; Abk. *WS*) genau drei Wartezyklen in der Statistik. Hiermit kann man die Auswirkungen des Lockdowns von Instruktionen auf die Laufzeit bzw. den Einfluss auf das Speicherinterface sehr gut beurteilen.

Ein typischer Messvorgang besitzt zwei Breakpoints – den ersten bei Beginn des Hauptprogramms, also an der Adresse der ersten Instruktion der *main* Routine. Der zweite Breakpoint wird an der Stelle gesetzt, wo der Assembler den Aufruf der Funktion *exit*() einfügt – also direkt nach Beendigung des Hauptprogramms. So ist es möglich, allein den Gewinn der Optimierungen durch statisches Cache-Locking an der Laufzeit des Hauptprogramms und allen von dort aufgerufenen Funktionen zu messen. Der Overhead, der durch den Startup-Code mit den darin enthaltenen Lockdown-Routinen und durch die Funktionen für die Aufräumarbeiten vor Beendigung entsteht, ist nur von der zu lockenden Cache-Größe abhängig und ist einmal separat bestimmt worden.

Der ARM Sourcelevel Debugger kann die Statistiken auch inkrementell ausgeben, d.h. die Änderungen seit der letzten Ausgabe anzeigen. So wird am ersten Breakpoint eine absolute Ausgabe angestoßen, die dann die Laufzeit des Startup-Codes ausgibt. Am zweiten Breakpoint, also nach Ende des Hauptprogramms, wird durch den Befehl `print $statistics_inc` die inkrementelle Laufzeit seit der letzten Ausgabe angezeigt, die dann der des Hauptprogramms entspricht. Wie man die hier benutzten Befehle und alle weiteren unterstützen Funktionen des ARM Sourcelevel Debugger zum Performance-Monitoring nutzt, zeigt [Adv98b].

### WCET-Messungen

Die aktuell eingesetzte Version 1.6r5 von aiT unterstützt die Analyse von Programmen in Systemen mit normalen Caches, bietet aber keine Unterstützung für gelockte Caches. Daher muss die Transparenz des Caches benutzt werden, um gelockte Speicherbereiche zu simulieren: Aus Sicht des Prozessors sind Speicherbereiche, die im Cache vorhanden sind und deshalb von dort geladen werden, einfach Bereiche, auf die mit nur einem Takt

Verzögerung zugegriffen werden kann. Da aiT die Definition von beliebigen Speicherbereichen mit unterschiedlichen Zugriffzeiten unterstützt (siehe Kapitel 2.2.2), wird für einen gelockten Speicherbereich einfach eine Zugriffsverzögerung von einem Takt angegeben. Alle ungelockten Bereiche haben weiterhin eine Zugriffsverzögerung von 4 Prozessortakten.

Eine Beschreibung des Speicherlayouts für den ADPCM De-/Encoder mit 512 Bytes gelocktem Cache-Speicher, die durch die WCETOPT-Toolchain (vgl. Kapitel 3) generiert wird, sieht folgendermaßen aus:

```
1  # Not locked Area at the beginning
2  MEMORY_AREA: 0x0 0xDF 1:1 4 READ&WRITE CODE&DATA
3  # Locked element: my_abs
4  MEMORY_AREA: 0xE0 0xEB 1:1 1 READ&WRITE CODE&DATA
5  # Locked element: my_fabs
6  MEMORY_AREA: 0xEC 0xF7 1:1 1 READ&WRITE CODE&DATA
7  # Not locked Area between
8  MEMORY_AREA: 0xF8 0x1A7 1:1 4 READ&WRITE CODE&DATA
9  # Locked element: my_cos
10 MEMORY_AREA: 0x1A8 0x1BB 1:1 1 READ&WRITE CODE&DATA
11 # Not locked Area between
12 MEMORY_AREA: 0x1BC 0xD73 1:1 4 READ&WRITE CODE&DATA
13 # Locked element: filtez
14 MEMORY_AREA: 0xD74 0xDA3 1:1 1 READ&WRITE CODE&DATA
15 # Locked element: filtep
16 MEMORY_AREA: 0xDA4 0xDBB 1:1 1 READ&WRITE CODE&DATA
17 # Not locked Area between
18 MEMORY_AREA: 0xDBC 0x1087 1:1 4 READ&WRITE CODE&DATA
19 # Locked element: __divsi3
20 MEMORY_AREA: 0x1088 0x1137 1:1 1 READ&WRITE CODE&DATA
21 # Rest of the memory (not locked)
22 MEMORY_AREA: 0x1138 0xFFFFFF 1:1 4 READ&WRITE CODE&DATA
```

**Listing 5.1:** Teil einer aiT AIS-Datei mit Speicherlayout

Vor jeder Zeile, die einen Speicherbereich beschreibt, steht eine Kommentarzeile von WCETOPT mit der Beschreibung des Bereiches bzw. der gelockten Funktion. Zu Anfang (Zeile 2) steht immer ein ungelockter Bereich mit 4 WS, der im Beispiel von Adresse `0x0` bis `0xDF` reicht. Direkt darauf folgt die Funktion `my_abs`, die von `0xE0` bis `0xEC` reicht und mit 1 WS als im Cache gelockt simuliert wird. In Zeile acht folgt ein ungelockter Bereich, der bis zur nächsten gelockten Funktion reicht – es werden also nur gelockte Funktionen in einzelne Speicherbereiche gefasst. Alle ungelockten Funktionen werden nicht explizit aufgelistet.

Diese Technik macht es möglich, Systeme mit gelockten Caches auch mit Programmen zu analysieren, die zwar unterschiedliche Speicherbereiche mit eigenen Zugriffzeiten unterstützen, aber keine gelockten Caches. Die gleiche Vorgehensweise der Simulation gelockter Speicherbereiche durch entsprechende Speicherlayouts lässt sich auch für den ARM Sourcelevel Debug-

ger anwenden. Dort hat bei Tests die Definition von Speicherbereichen mit 1 WS Verzögerung zu den gleichen Ergebnissen in der Laufzeit geführt wie beim Locking der entsprechenden Regionen.

Bezieht man den Startup-Code mit in die WCET-Analyse ein, so ist es durch Angabe des entsprechenden Speicherlayouts möglich, das komplette Programm auf einem System mit gelocktem Cache zu analysieren. Dadurch lassen sich garantierte Worst-Case Laufzeiten für Systeme mit harten Zeitschranken bereits ab dem Start eines Programmes angeben, inklusive der zusätzlich zum Lockdown notwendigen Instruktionen, die sich im Startup-Code befinden.

## 5.2 Benchmarks

Für die Verifizierung der Funktion und der Optimierung der Worst-Case-Laufzeit sind fünf Benchmarks ausgewählt worden, die über die Komplexität von einfachen Sortieralgorithmen hinausgehen, die z.T. zum Test von Basisblockoptimierungen eingesetzt werden. Das erste Programm aus Kapitel 5.2.1 stammt aus der *SNU-RT Benchmark Suite* [SNU]. Im darauf folgenden Kapitel 5.2.2 wird eine Codec-Implementierung von *Sun Microsystems* [G72] für die WCET-Analyse angepasst. Beide Benchmarks encodieren Audiodaten zuerst verlustbehaftet, bevor diese anschließend wieder decodiert werden. In Kapitel 5.2.3 wird ein C Programm der *MDH WCET BENCHMARK SUITE* [MRT] verwendet, das automatisch aus einem Statemate Modell generiert worden ist. Das nächste Programm in Kapitel 5.2.4 ist ein Kompressionswerkzeug, das ursprünglich aus der *SPEC95 Suite* [SPE95] stammt und einen Eingabestrom aus einem Puffer komprimiert. Anschließend wird in Kapitel 5.2.5 ein MPEG2 Encoder der *MPEG Software Simulation Group* [MSS96] als komplexer Media-Benchmark verwendet, wie er bereits in vielen Programmen zum Encodieren von Videoströmen eingesetzt wird. Der Vergleich in Kapitel 5.3 verwendet ein zusätzliches Programm, um die Vorteile der hier entwickelten Verfahren gegenüber anderen Selektionsverfahren darzustellen.

Die Benchmarks sind so ausgelegt bzw. modifiziert worden, dass sie ohne Kommandozeilenoptionen zu starten sind. Außerdem liegen alle Eingabedaten in Form von Arrays vor, damit jeder Zugriff und die resultierenden Verzögerungen für die WCET-Analysen berechenbar sind und nicht vorhersehbare I/O-Operationen vermieden werden. Die Ausgabedaten werden ebenfalls in genügend große Arrays geschrieben, so dass auch Schreibvorgänge in der Geschwindigkeit vorhersagbar sind.

Für alle Benchmarks wird ein System mit 33 MHz und 32-Bit DRAM angenommen, der mit 4 WS Verzögerung angesprochen werden kann. Der Cache des Prozessors läuft ebenfalls mit voller Taktfrequenz und lässt bei Zugriffen eine Verzögerung von nur 1 WS zu. Die Ausführungszeiten sind zur

besseren Übersicht in Millisekunden für die angegebene Taktgeschwindigkeit umgerechnet. Dargestellt werden für jeden Benchmark die von jedem in dieser Arbeit präsentierten Algorithmus erzielten Worst-Case-Laufzeiten für jede Cachegröße. Außerdem wird für den jeweiligen Benchmark die ACET ermittelt, die der Graph-Algorithmus durch seine Optimierungen erzielen konnte. Bei der Untersuchung der WCETs der einzelnen Benchmarks hat sich herausgestellt, dass alle drei in dieser Arbeit entwickelten Selektionsalgorithmen sehr ähnliche Lösungen berechnen, so dass die resultierenden WCETs der Selektionsverfahren nur in Einzelfällen voneinander abweichen. Aus diesem Grund ist darauf verzichtet worden, die ACETs für sämtliche Selektionsverfahren zu bestimmen.

Zusätzlich wird für jede Cachegröße der maximale Overhead und die ermittelten ACET- sowie WCET-Laufzeiten des analysierten Programms auf einem System mit ungelocktem Cache angegeben. Der Overhead sind die Takte, die beim Programmstart benötigt werden, um die ausgewählten Funktionen in den Cache zu locken. Dies ermöglicht einen direkten Vergleich von Kosten und Nutzen einer Optimierung, wobei die Kosten im Vergleich zum erzielten Speedup bei der WCET relativ gering sind. Die ACET- und WCET-Messungen bei ungelocktem Cache ist mit einer nicht optimierten Version der Benchmarks und den entsprechenden Einstellungen für die Cachegröße mit den Werkzeugen aus Kapitel 5.1 durchgeführt worden.

### 5.2.1 ADPCM En-/Decoder

Der ADPCM En-/Decoder Benchmark entstammt der *SNU-RT Benchmark Suite for Worst Case Timing Analysis* [SNU] und benutzt die Adaptive Differenzielle Pulse-Code Modulation (ADPCM), mit deren Hilfe PCM-Signale in ein platzsparenderes Format konvertiert werden. Im Gegensatz zum PCM werden nur die Änderungen im Vergleich zum vorigen Wert gespeichert, was eine Platzersparnis von 25% erlaubt (Differenzielle PCM). Außerdem sind die Quantisierungsschritte variabel, was für ein gegebenes Signal-Rausch-Verhältnis die benötigte Bandbreite weiter reduziert. Das Testprogramm generiert ein Sinussignal als Eingabestrom, das mit Hilfe des Codecs komprimiert wird. Das encodierte Signal wird anschließend wieder decodiert, bevor der Benchmark endet.

Folgende Eigenschaften des ausführbaren Programms sind für den Benchmark interessant:

- 109 KB Programmgröße
- 19 Funktionen
- davon 7 aus System-Bibliotheken
- 950 Zeilen C-Quellcode

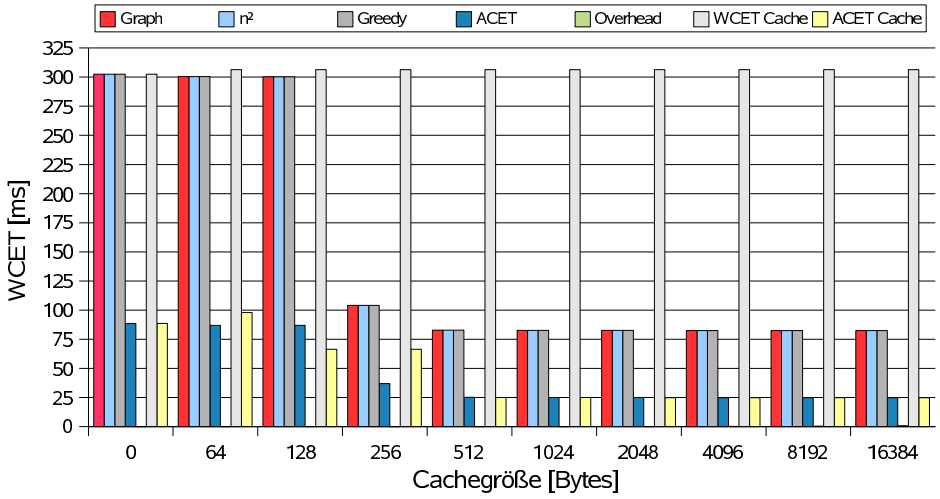



**Abbildung 5.1:** ADPCM En-/Decoder Benchmark

Die Anzahl der Funktionen umfasst sowohl alle im Quelltext enthaltenen als auch alle von dort aufgerufenen Funktionen aus Bibliotheken. Die Anzahl Funktionsaufrufe von Funktionen in externen Bibliotheken ist zusätzlich interessant um zu sehen, welche Teile des Programms durch gängige Optimierungsverfahren auf Quelltextebene nicht erfasst werden.

Die Abbildung 5.1 zeigt die Ergebnisse der Mess- bzw. Analysedurchläufe für den ADPCM-Benchmark. Man sieht bereits, dass alle eingesetzten Algorithmen die gleichen Optimierungen erzielen, da das Programm einen fast ausschließlich sequentiellen Ausführungspfad hat. Der einzige nebenläufige Pfad ist so kurz, dass er niemals die WCET bestimmt. Die starke Reduktion der WCET bei einer Cachegröße von 256 Bytes, die eine Beschleunigung der Programmausführung im Worst-Case um Faktor drei darstellt, wird durch das Locking der Funktion $\_divsi3$ erzielt. Diese Funktion bildet die von dem Befehlssatz [Adv03] der ARM-Familie nicht unterstützte Integer-Division nach und wird bei der Codierung für die Berechnung von Sinus und Cosinus häufig verwendet. Die Divisionsfunktion ist Bestandteil der Standard-C-Library für die ARM-Familie und würde bei konventionellen Optimierungen, die ausschließlich auf Quelltextebene arbeiten, nicht beschleunigt oder müsste aufwändig nachimplementiert werden. Insgesamt war eine Reduktion der WCET von 302 ms auf 83 ms durch die Algorithmen ab einer Cachegröße von 1024 Bytes möglich, was einer Einsparung von 72% entspricht. Weitere Vergrößerungen des Caches haben keinen signifikanten Einfluss auf die WCET.

Die Average-Case Execution Time profitiert von den Optimierungen in ähnlicher Weise wie die WCET, auch wenn die Kurve keine so drastischen Einsparungen von einer zur nächsten untersuchten Cachegröße zeigt. Dafür

werden häufiger kleine Gewinne erzielt, so dass die ACET um 72% von 89 ms auf 25 ms ab einer Cachegröße von 512 Bytes gesenkt werden kann. Der maximale Overhead bei einer Cachegröße von 16 KB macht nur etwa 3% der geringsten ACET aus und ist zu vernachlässigen.

Die WCET des Programms auf einem System mit ungelocktem Cache liegt für alle analysierten Cachegrößen etwa 2% über der Worst-Case-Laufzeit eines Systems ohne Cache. Bei der Analyse der Speicherzugriffe sagt aiT derart viele Cache-Misses vorher, dass der erzielte Gewinn kleiner ist als die zusätzlich benötigte Zeit, um den Cache zu füllen. Durch eine Zeilengröße von 32 Byte werden u.U. auch solche Instruktionen geladen, die gar nicht verwendet werden. Im direkten Vergleich zwischen einem System mit normal betriebenem Cache und einem mit gelocktem Cache erzielen die in dieser Arbeit vorgestellten Selektionsalgorithmen eine Verbesserung von bis zu 73%.

Die gemessene ACET auf einem System mit herkömmlichem Cache zeigt ähnliche Werte wie sie durch den Graph-Algorithmus erzielt worden ist. Allerdings sind bei kleinen Caches (z.B. 64 und 256 Bytes) um 10 bzw. 80% längere Laufzeiten auf einem System mit ungelocktem Cache zu beobachten. Durch die wenigen Cachezeilen wird der Inhalt des Caches durch andere Programmteile häufig verdrängt, so dass auch hier die Kosten für Cache-Misses den erzielten Gewinn übersteigen. Ab einer Cachegröße von 512 Bytes lässt sich zwischen der ACET bei ungelocktem Cache und der gelockten Version des Graph-Algorithmus kein signifikanter Unterschied mehr erkennen. Im direkten Vergleich lässt sich feststellen, dass zwar Vorteile bei einzelnen Cachegrößen für die einzelnen Betriebsmodi des Caches zu beobachten sind. Für die meisten untersuchten Cachegrößen liegen die Einsparungen in der ACET aber mit einer Abweichung von weniger als 1% fast gleich auf.

Wie stark die einzelnen Abweichungen zwischen den Ergebnissen des gelockten und des ungelockten Caches sind, hängt von dem Verhältnis zwischen Programmgröße und eingesetzter Cachegröße ab. Zudem hängt von der Programmgröße bzw. von der Größe der Funktionen mit den längsten Laufzeiten ab, ab welcher Cachegröße der ungelockte Cache schneller ist. Gibt es mehrere rechenintensive Funktionen, die der Cache nicht alle fassen kann, verdrängen sich die Funktionen bei wechselnder Ausführung gegenseitig aus dem Cache. In diesem Fall ist der gelockte Cache bei richtiger Auswahl der Funktionen zum Lockdown schneller.

Dieses Verhalten des Benchmarks auf einem System mit herkömmlichem Cache bzgl. der ACET und der WCET ist auch bei den folgenden Benchmarks zu beobachten. Deshalb werden an den entsprechenden Stellen zwar die gemessenen Unterschiede in der Laufzeit genannt, die Ursache wird aber nicht jedesmal erneut erläutert. Genauere Untersuchungen des Einflusses einer Speicherhierarchie auf obere Schranken und Vorhersagbarkeit der Ausführungszeiten von Programmen in Realzeit-Systemen stellt [WM04] an.

### 5.2.2 G723

Der G723 Codec ist eine Erweiterung des bereits vorgestellten ADPCM-Verfahrens zur bandbreitenreduzierten Übertragung von Sprache. Der Codec ist für noch geringere Bandbreiten entworfen als der bereits vorgestellte ADPCM-Codec und wird z.B. bei der Voice over IP Kommunikation eingesetzt. Der Quelltext ist ursprünglich von Sun Microsystems entwickelt [G72] und so modifiziert worden, dass 256 Bytes Eingabedaten aus einem Array gelesen werden und nach der Codierung in ein Ausgabearray geschrieben werden.

Die Binärdatei für den ARM-Prozessor besitzt folgende Daten:

- 107 KB Größe
- 15 Funktionen
- Keine verwendeten System-Bibliotheken
- 1620 Zeilen C-Quellcode

Der Codec besitzt eher komplexe Funktionen, auf die die benötigte Rechenleistung relativ gleich verteilt ist. Deshalb skaliert der Benchmark mit steigender Cache-Größe, und die Kurven in Abbildung 5.2 für WCET und ACET weisen keine so starken Knicke auf wie die des ADPCM-Benchmarks. Alle Selektionsverfahren können eine Senkung der Worst-Case Execution Time des Programms von 122 ms auf 49 ms erreichen, was einer Einsparung von 60% Rechenzeit entspricht. Die ACET profitiert etwas weniger von den Optimierungen mit einer Verkürzung der Laufzeit um 52% von 67 ms auf 32 ms.

Die WCET eines Systems mit ungelocktem Cache ist für alle untersuchten Cachegrößen mit 132 ms um 8% höher als bei einem System ohne Cache, was einer Verlängerung der Laufzeit um 10 ms entspricht. Hieraus resultieren maximale Einsparungen von bis zu 63% der Selektionsalgorithmen gegenüber einem System mit normal operierendem Cache.

Die ACET dieses Systems liegt bei den Messungen bis zu einer Größe von 8 KB zwischen 0,5% (512 Bytes Cache) und 26,5% (256 Bytes Cache) über der ACET des Programms mit gelocktem Cache. Dies entspricht einer Laufzeitverlängerung von bis zu 11,26 ms. Erst ab einer Cachegröße von 8 KB gleicht die durch ein System mit normal operierendem Cache erzielte ACET der durch die Selektionsalgorithmen erzielten Laufzeit. Insgesamt ist Cache-Locking für die ACET des G723-Benchmarks vorteilhaft, da es zu ACET-Reduktionen von bis zu 26,5% gegenüber normal betriebenen Caches führt.

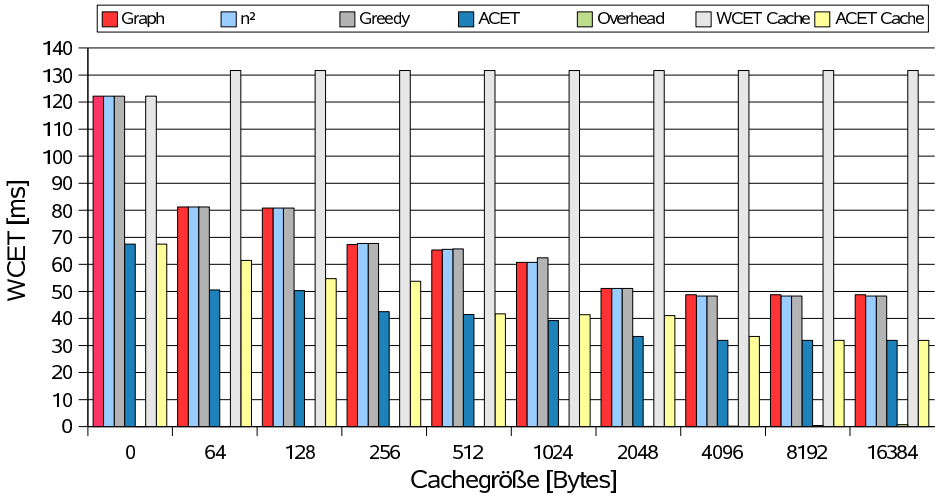



**Abbildung 5.2:** G723 Benchmark

### 5.2.3 Statemate

Der Statemate-Benchmark testet die entwickelten Optimierungen für statisches Cache-Locking mit Programmen, die durch Statecharts-Modelle erstellt worden sind. Das Modell, aus dem das Programm erzeugt wird, implementiert eine intelligente Fensterhebersteuerung aus dem Automobilbereich mit Einklemmschutz und halbautomatischem Öffnen bzw. Schließen der Fenster. Der C-Quellcode des Programms ist durch den *STAtechart Real-time-Code generator STARC* des C-Lab [ESS99] generiert und anschließend wie jedes andere Programm übersetzt worden (vgl. Seite 19).

Der Statemate-Benchmark benutzt sehr viele bedingte Verzweigungen in Form von `if` sowie `sitch - case` Befehlen und arbeitet überwiegend mit ganzzahligen Additionen. Die Steuerung hat in Form der Binärdatei folgende Daten:

- 145 KB Programmgröße
- 21 Funktionen
- Keine verwendeten System-Bibliotheken
- 1201 Zeilen C-Quellcode

Das Diagramm in Abbildung 5.3 zeigt die Ergebnisse des Benchmarks mit dem Statemate-Programm. Man sieht deutlich, dass die Worst-Case Execution Time von dem Lockdown der kritischen Funktionen bei allen Algorithmen mehr profitiert als die ACET. Im Worst-Case tragen fast alle Funktionen gleich zur Laufzeit bei, weshalb die WCET mit steigender Cachegröße stetig abnimmt. Da von der zu betrachtenden Funktionsmenge auch einige mit

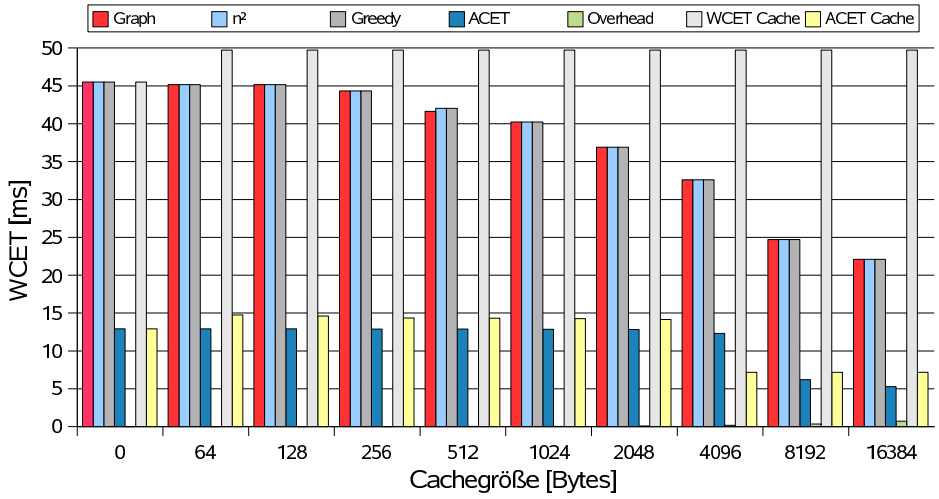



**Abbildung 5.3:** Statemate Benchmark

Größen kleiner als 1 KB vorhanden sind, senken auch kleine Cachegrößen die WCET bereits deutlich. Insgesamt kann bei 16 KB Cachegröße die WCET um 52% von anfangs 46 ms auf 22 ms gesenkt werden.

Die Ursache für die bereits sehr niedrige ACET ist, dass die Eingabedaten für den Simulationsdurchlauf dem Best-Case sehr ähnlich sind und daher nur wenig Funktionen überhaupt ausgeführt werden. Die Funktionen, die ausgeführt werden, sind jedoch so groß, dass die kleinste erst bei 4 KB in den Cache gelockt werden kann. Mit steigender Cachegröße können immer mehr Funktionen gelockt werden, so dass die ACET dann weiter sinkt. Ein deutlicher Speedup ist erst bei einer Cachegröße von 8 KB zu sehen. Trotzdem kann die ACET per Cache-Locking sogar von 13 ms auf 6 ms gesenkt werden, was einer Verkürzung der Laufzeit um 53% entspricht.

Auf Systemen mit ungelocktem I-Cache liegt die WCET mit 50 ms ungefähr 4,5 ms über der eines Systems ohne Cache, was einer Verlängerung im Worst-Case um 10% entspricht. Insgesamt lässt sich durch die in dieser Arbeit präsentierten Selektionsalgorithmen eine Verringerung der WCET um bis zu 56% gegenüber eines Systems mit normal betriebenem Cache erzielen.

Die ACET der vom Graph-Algorithmus optimierten Version liegt bei allen Cachegrößen durchschnittlich 12% unter der Laufzeit eines Systems mit herkömmlichem Cache, was im Mittel 1,5 ms weniger Rechenzeit entspricht. Nur bei 4 KB Cachegröße ist der herkömmliche Cache um 41% schneller, was 5 ms Laufzeit entspricht. Im direkten Vergleich der ACET zwischen einem System mit normal betriebenem Cache und einem mit gelocktem Cache erzielen die in dieser Arbeit vorgestellten Selektionsalgorithmen eine Verbesserung von bis zu 26%.

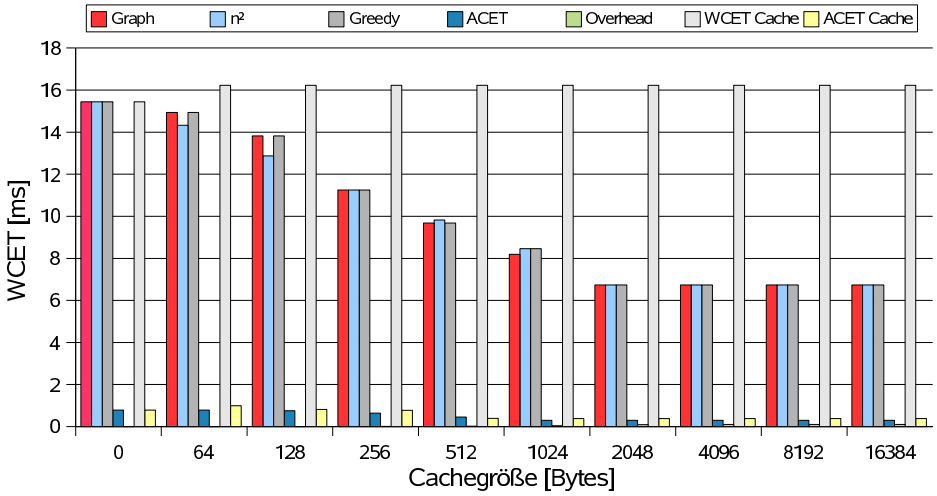



**Abbildung 5.4:** Compress-Benchmark

### 5.2.4 Compress

Der Compress-Benchmark stammt aus der SPEC95-Suite [SPE95] und ist vom *Mälardalen Research and Technology Centre* (MRTC) für die WCET-Berechnung angepasst worden [MRT]. So verarbeitet der Algorithmus die Eingabedaten aus einem Puffer mit Zufallsdaten, um den Besonderheiten bei der Analyse von Programmen in Realzeitsystemen Rechnung zu tragen. Die Komprimierung der Eingabedaten erfolgt mit einer adaptiven Lempel-Ziv-Codierung. Die Größe des Eingabepuffers ist 50 Bytes, und die Größe des Wörterbuchs ist auf die für LZ-Komprimierung übliche Größe von 4096 Einträgen beschränkt.

Die Binärdatei des Benchmarks besitzt folgende Eigenschaften:

- 105 KB Dateigröße
- 11 Funktionen
- 2 davon aus System-Bibliotheken
- 528 Zeilen C-Quellcode

In Abbildung 5.4 sind die Ergebnisse der ACET- und WCET-Analysen dargestellt. Die einzelnen WCET-Analysen weichen bei 64 und 128 Bytes Cache mit einem Vorteil von 5% für den $n^2$-Algorithmus voneinander ab. Der Graph-Ansatz kann aber bei 1024 Bytes Cache einen Geschwindigkeitsvorteil von 3% gegenüber den anderen beiden Algorithmen erzielen. Diese Abweichungen zwischen dem Graph- und dem $n^2$-Algorithmus sind sehr ungewöhnlich und liegen nicht an einem Fehlverhalten einer der Algorithmen. Zum

einen ist der Kontrollflussgraph rein sequentiell, d.h. er besitzt nur einen langen Pfad, so dass beide Algorithmen die selbe Lösung bestimmen sollten. Zum anderen liefern beide für identische Eingaben der Funktionslaufzeiten auch wirklich die gleichen Lösungen. Die Ursache für die beobachteten Abweichungen liegt bei dem WCET-Analysewerkzeug aiT, das für die selbe Funktion unterschiedliche Ausführungszeiten berechnet in Abhängigkeit davon, welche Funktionen noch mit in den Cache gelockt werden. So wird die Rewardberechnung für einzelne Funktionen verfälscht, obwohl nur die reinen Funktionen ohne Daten oder Operanden in den Cache gelockt werden. Da sich alle Algorithmen ab einer Cachegröße von 2 KB gleich verhalten, lässt sich eine Reduktion der WCET von 15 ms um 56% auf 7 ms erreichen.

Die Säulen der ACET zeigen, dass die ohnehin geringe durchschnittliche Ausführungszeit von 0,8 ms um 62% auf 0,3 ms gesenkt werden kann. Ab einer Cachegröße von 1024 Bytes wird aber kein nennenswerter Gewinn mehr durch Vergrößerung des Caches erzielt und der Gewinn durch den steigenden Overhead wieder aufgehoben.

Ein ungelocktes System besitzt im Vergleich zu einem System ohne Cache beim Compress-Benchmark eine um 8% oder 1,3 ms höhere WCET. Die in Kapitel 4 vorgestellten Selektionsalgorithmen können im direkten Vergleich zu einem System mit normalem Cache eine Verringerung der WCET um bis zu 58% erzielen, was einer Verkürzung der Worst-Case-Laufzeit um 10 ms entspricht.

Die ACET ist bei der Optimierung des Programms durch einen der vorgestellten Selektionsalgorithmen in den meisten Fällen 22% geringer als bei einem System mit herkömmlichem Cache. Diese Einsparung wird jedoch ab einer Cachegröße von 512 Bytes durch den Overhead für das zu Beginn nötige Füllen des Caches wieder aufgehoben. Lediglich bei 512 Bytes Cache liegt der herkömmliche Cache 13% vor den vom Graph-Algorithmus erzielten Ergebnissen. Im direkten Vergleich des gelockten Caches mit einem System, das einen herkömmlichen Cache verwendet, ist kein Unterschied in der ACET zu erkennen, wenn der Overhead in die Berechnung der Laufzeit mit einbezogen wird. Denn zum einen muss in beiden Fällen der Cache gefüllt werden und zum anderen finden ab einer Cachegröße von 2048 Bytes alle rechenintensiven Funktionen im Cache Platz.

### 5.2.5 MPEG2-Encoder

Dieser Benchmark entstammt dem MPEG2 Encoder/Decoder Pack, das von der *MPEG Software Simulation Group* [MSS96] implementiert worden ist. Der Quellcode ist wie die anderen Benchmarks zur WCET-Analyse so abgeändert worden, dass alle Eingabedaten aus Arrays gelesen werden und die ausgegebenen Daten wieder in Arrays geschrieben werden. Das Testprogramm verarbeitet einen Videostrom im YUV-Farbmodell und erzeugt dar-

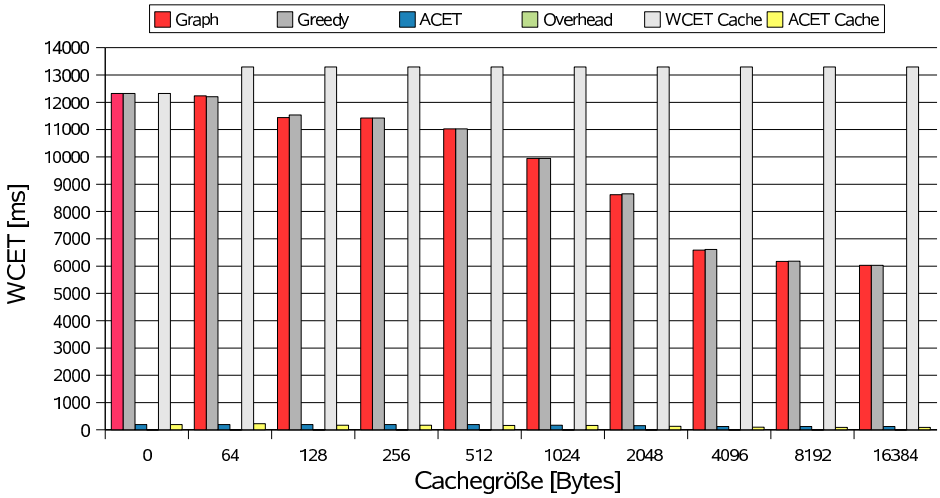



**Abbildung 5.5:** MPEG2-Benchmark

aus einen MPEG2-Datenstrom in derselben Auflösung. Das zu encodierende Testvideo besteht aus 3 Bildern mit einer Auflösung von 128*128 Pixeln.

Die Binärdatei im ARM-ELF Format hat folgende Eigenschaften:

- 595 KB Größe
- 210 Funktionen
- davon 118 aus System-Bibliotheken
- 7916 Zeilen C-Quellcode
- 72 KB Eingabedaten in Form von Char-Arrays

Bei der Menge von 210 Funktionen im reinen Programmcode, also ohne Startup- und Exit-Code, würden sich für den $n^2$-Algorithmus nach Gleichung 4.9 maximal 22.155 nötige aiT-Analysevorgänge ergeben. Selbst bei der kleinsten betrachteten Cachegröße von 64 Bytes bleiben noch 78 Funktionen, die aufrund ihrer Größe in den Cache gelockt werden könnten und deshalb mit maximal 3.081 Analysevorgängen durch aiT betrachtet werden müssten. Für ein Programm dieser Größe benötigt ein einzelner WCET-Analysevorgang durch aiT mit dem auf Seite 81 beschriebenen System bereits mehr als fünf Minuten, weshalb auf eine Analyse des Benchmarks mit dem $n^2$-Algorithmus verzichtet wird.

Obwohl der Kontrollfluss des Programms diverse Verzweigungen zu nebenläufigen Ausführungspfaden aufweist, erzielen die beiden angewendeten Greedy- und Graph-Algorithmen ganz ähnliche Ergebnisse. Der Worst-Case-Pfad ist meist so lang, dass auch durch Optimierungen kein Pfadwechsel

stattfindet. Findet doch einmal ein Pfadwechsel statt, sind die neu hinzugekommenen Pfadabschnitte im Vergleich zum gesamten Ausführungspfad so kurz, dass sich eine falsche bzw. ausbleibende Optimierung in der Gesamtlaufzeit kaum niederschlägt. Trotzdem lassen sich Nachteile der Greedy-Optimierung in Größenordnungen von 1-2% messen, die sich auf die Probleme auf Seite 35 in der Rewardberechnung bei Wechsel des längsten Ausführungspfades zurückführen lassen.

Durch die angewendeten Optimierungen lässt sich eine Reduktion der Worst-Case Execution Time um 51% von 12.327 ms auf 6.032 ms erreichen. Die im Vergleich zur WCET sehr niedrige ACET von 196 ms für ein System ohne Cache kann beim Einsatz eines gelockten Caches von 16 KB Größe um 38% auf 121 ms gesenkt werden.

Die WCET liegt bei einem System mit herkömmlichem Betrieb des Caches ca. 9% über der WCET des Benchmarks auf einem System ohne Cache und das entspricht 1.000 ms mehr Ausführungszeit. Im direkten Vergleich erzielen die beiden Greedy- und Graph-Algorithmen eine Reduktion der WCET um bis zu 55% gegenüber einem System mit normalem Cache.

Die Average-Case Execution Time liegt für die kleinste gemessene Größe des Caches von 64 Bytes ungefähr 18% über der Version des Benchmarks für das Locking in den Cache. Alle größeren ungelockten Caches können eine um durchschnittlich 15% geringere Laufzeit als die gelockten Versionen erzielen, was 24,5 ms weniger Zeit für die Ausführung entspricht. Vergleicht man ein System mit herkömmlichem Cache mit den erzielten Ergebnissen der ACET durch die angewendeten Selektionsalgorithmen, ist die Ausführung bei normalem Cache ab einer Cachegröße von 128 Bytes bis zu 35% schneller.

## 5.3 Vergleiche mit anderen Selektionsverfahren

Die bisher verwendeten fünf Benchmarks weisen sehr lange Worst-Case-Ausführungspfade auf, die entweder keine oder im Vergleich dazu sehr kurze nebenläufige Pfade haben. Die bisher angewendete Einzelpfadanalyse aus Kapitel 4.2, die in [CPIM05, PD02] Anwendung findet, optimiert nur Elemente entlang des einmal ermittelten Worst-Case-Ausführungspfads. Die Stärke der in Kapitel 4.3 bis 4.5 entwickelten Ansätze gegenüber dieser Einzelpfadanalyse, ist jedoch die Möglichkeit, auch Elemente auf nebenläufigen Pfaden zu optimieren.

Um einen Vergleich zu ermöglichen, ist der Multisort-Benchmark aus [WM05] in leicht veränderter Form verwendet worden, dessen Callgraph dem Kontrollflussgraphen gleicht und in Abbildung 5.6 gezeigt wird. Dieser Benchmark ist aufgrund seiner geringen Komplexität bewusst nicht bei den Benchmarks aus Kapitel 5.2 aufgezählt worden. Das Programm implementiert drei Sortierfunktionen, die alternativ das selbe Array mit 100

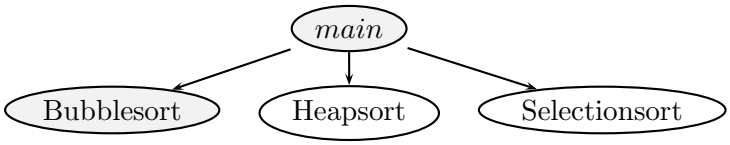



**Abbildung 5.6:** Call- und Kontrollflussgraph des Multisort-Benchmarks

vorzeichenlosen ganzen Zahlen von 32 Bit Breite sortieren. Dabei entscheidet das erste Element des Arrays, welche Sortierfunktion benutzt wird, indem in der Funktion *main* der Rest einer Division durch 3 über eine `switch - case` Anweisung verzweigt. Bleibt bei der Revision kein Rest, dann wird Bubblesort benutzt, bei Rest 1 wird Selectionsort aufgerufen, und bei Rest 2 findet Heapsort Anwendung. So ist sichergestellt, dass das Programm drei konkurrierende Ausführungspfade ähnlicher Länge besitzt, damit die Auswirkungen auf die zwei Klassen von Algorithmen getestet werden können.

Der Multisort-Benchmark ist den Algorithmen zur Graph- und Einzelpfad-Optimierung unterzogen worden, deren Resultate in Abbildung 5.7 dargestellt sind. Die beiden Knoten *main* und *Bubblesort*, die bei dem unoptimierten Programm den Worst-Case-Pfad bilden, sind in der Abbildung 5.6 grau hinterlegt. Wird jetzt bei einer Cachegröße von 128 Bytes die Funktion *Bubblesort* gelockt, verkürzt sich die Worst-Case-Ausführungszeit so sehr, dass ein Wechsel des längsten Ausführungspfades stattfindet. In diesem Fall ist der neue Worst-Case-Pfad $main \rightarrow Selectionsort$. Wird auch *Selectionsort* bei einer Cachegröße von 1024 Bytes durch den Graph-Algorithmus gelockt, verkürzt sich die Worst-Case-Laufzeit erneut deutlich. Es findet zwar ein erneuter Pfadwechsel statt, aber die Laufzeit des ungelockten Heapsort-Algorithmus ist in allen Fällen kürzer als die des in den Cache gelockten Bubblesort-Algorithmus, so dass keine weiteren Optimierungen mehr möglich sind. Die leichte Veränderung bei 512 Bytes wird durch das Locking der *main* Funktion herbeigeführt, die lediglich einen kleinen Gewinn von 2% ermöglicht.

Der Einzelpfad-Algorithmus aus Kapitel 4.2 kann nicht so große Einsparungen in der WCET erzielen, da ein eventueller Pfadwechsel nicht erkannt wird. Ein derartiger Wechsel im Kontrollfluss findet nach dem Lockdown der Funktion *Bubblesort* bei einer Cachegröße von 128 Bytes statt. Danach kann noch wie beim Graph-Algorithmus die Funktion *main* gelockt werden (512 Bytes Cachegröße), die ebenfalls auf dem zu Beginn ermittelten Worst-Case-Pfad liegt. Alle weiteren Vergrößerungen des zur Verfügung stehenden Caches bringen keine Verbesserungen der Worst-Case-Laufzeit mehr, da diese nun durch den Pfad $main \rightarrow Selectionsort$ bestimmt wird.

Durch diesen Nachteil benötigt der Multisort-Benchmark, der durch den Einzelpfad-Algorithmus optimierte worden ist, ab einer Cachegröße von 1 KB

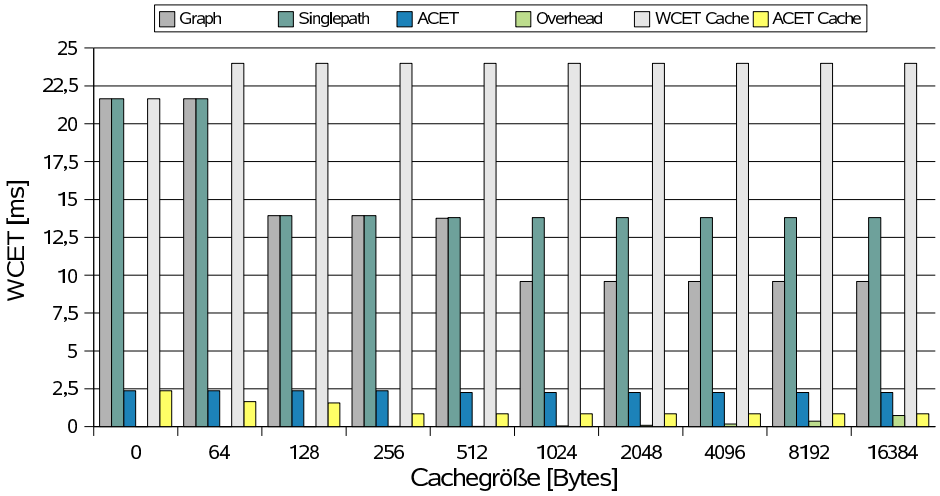



**Abbildung 5.7:** Vergleich der Einzelpfadanalyse mit dem Graph-Algorithmus

etwa 20% mehr Zeit für die Ausführung im Worst-Case als beim Graph-Algorithmus. Insgesamt kann der Graph-Algorithmus die WCET um 55% von 22 ms auf 10 ms senken. Der Einzelpfad-Algorithmus kann dagegen nur 36% Laufzeit im Worst-Case einsparen und erzielt damit eine maximale Zeit von 13,8 ms. Die Schere zwischen den beiden Algorithmen wird bei Programmen umso größer, je mehr nebenläufige Pfade ein Programm hat und je kleiner die Differenz zwischen der Länge dieser Pfade und dem Worst-Case-Ausführungspfad ist.

Die ACET kann dagegen kaum von den WCET-Optimierungen profitieren, da bei den getesteten Eingabedaten der von den Algorithmen nicht optimierte Heapsort-Algorithmus verwendet wird. Trotzdem lässt sich eine Reduktion der ACET um 5% von 2,37 ms auf 2,26 ms messen, die durch das Locking der *main* Funktion in den Cache erreicht wird.

Auch hier liegt die WCET für die Ausführung auf einem System mit ungelocktem Cache ungefähr 11% über der Laufzeit auf einem System ganz ohne Cache, was 2,4 ms mehr Rechenzeit entspricht. Im direkten Vergleich zu diesem System mit normalem Cache kann der Graph-Algorithmus eine Reduktion der WCET um 60% erreichen.

Da der normale ungelockte Cache nicht auf Funktionen des längsten Ausführungspfades beschränkt ist, kann er die ACET bereits ab 64 Bytes verwendetem Cache um 30% im Vergleich zum Graph-Algorithmus senken. Bei einem direkten Vergleich erzielt ein System mit normalem Cache gegenüber den Selektionsalgorithmen aus Kapitel 4 eine Verbesserung der ACET von bis zu 64%.

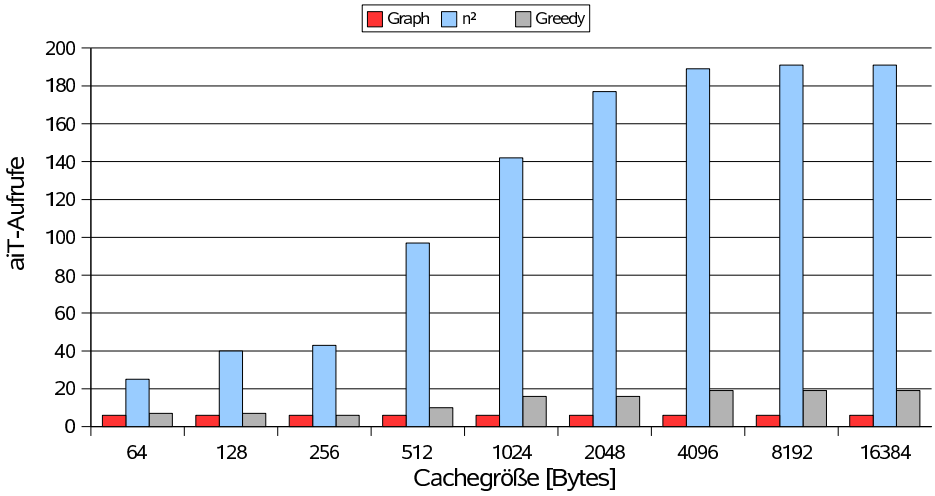



**Abbildung 5.8:** Benötigte aiT-Aufrufe für den ADPCM-Benchmark

## 5.4 Zeitaufwand für Selektionsalgorithmen

In die Kategorie Benchmarks fallen auch die Messergebnisse des Zeitaufwands in Form von CPU-Zeit, die ein Algorithmus benötigt, um die Funktionen für das Lockdown in den Cache auszuwählen. Zwar werden solche Optimierungen meist offline, also beim Entwurf eines Eingebetteten Systems eingesetzt, jedoch benötigen die vorgestellten Algorithmen sehr unterschiedliche Rechenleistungen, um die Funktionen mit dem größten Reward zu bestimmen. Die meiste Rechenleistung benötigen dabei nicht die entwickelten Algorithmen, sondern der eingesetzte WCET-Analyzer aiT. Da ein einzelner Analysevorgang selbst für kleine Binärdateien schon 30 Sekunden dauern kann, ist es sinnvoll, die Anzahl der benötigten aiT-Aufrufe niedrig zu halten. Als Beispiel für Analysen, die mehr Zeit benötigen, sei auf den MPEG2-Benchmark auf Seite 75 verwiesen, wo eine einzige WCET-Analyse bereits über fünf Minuten Zeit auf dem unten angegebenen Testsystem in Anspruch nimmt.

Um eine beispielhafte Bestimmung der benötigten aiT-Aufrufe durchzuführen, sind der ADPCM En-/Decoder aus Kapitel 5.2.1 und der MPEG2-Encoder aus Kapitel 5.2.5 verwendet worden. Für beide Benchmarks sind die nötige Anzahl an aiT-Analysevorgängen und die nötige Rechenzeit für Cachegrößen von 64 Bytes bis 16384 Bytes bestimmt worden.

Abbildung 5.8 trägt für die drei Algorithmen die benötigte Anzahl der aiT-Aufrufe über der jeweils verwendeten Cachegröße für den ADPCM-Benchmark ab. Die Anzahl der Aufrufe wird im Gegensatz zu den übrigen Benchmarks erst von 64 Bytes Cachegröße startend bis zu 16 KB Cache dargestellt. Die Laufzeit des Greedy-Algorithmus sollte linear mit der Anzahl

der zu lockenden Funktionen eines Programms ansteigen. Im Beispiel ist die Kurve nicht linear, da zum einen die auf der x-Achse aufgetragene Cachegröße exponentiell wächst. Zum anderen ist es möglich, dass bei steigender Cachegröße weniger Funktionen gelockt werden, da nun größere Funktionen mit höherem Reward in den Cache passen und diesen möglicherweise komplett ausfüllen. Außerdem kann es einen Punkt geben, an dem ein noch größerer Cache keine weiteren Funktionen mehr fassen kann, weil bereits alle Funktionen gelockt sind. Tendenziell steigt beim ADPCM-Benchmark jedoch die Laufzeit des Greedy-Algorithmus mit der Cachegröße, weil immer mehr Funktionen in den Cache passen und deshalb auch analysiert werden müssen.

Für den $n^2$ Algorithmus sieht man bereits, dass die benötigte Anzahl von Analysevorgängen durch aiT annähernd quadratisch wächst. Ab 4 KB Cachegröße ist keine signifikante Änderung der aiT-Aufrufe mehr zu erkennen, weil fast alle Funktionen in den Cache gelockt worden sind und so keine zusätzlichen Analysevorgänge durch mehr Cache nötig werden. Die Anzahl der WCET-Messungen steigt normalerweise quadratisch mit der Anzahl der Funktionen, die für das Beispielprogramm konstant ist. Der Anstieg der aiT-Aufrufe lässt sich hier durch die wachsende Cachegröße erklären, die es nach und nach erlaubt, immer mehr Funktionen zu locken. Dies macht die Optimierung von Programmen mit vielen Funktionen für Systeme mit großen Caches sehr zeitaufwändig und benötigt viel Rechenleistung.

Der Graph-Algorithmus zeigt bereits bei diesem relativ kleinen Testprogramm, dass die Anzahl der aiT-Aufrufe deutlich geringer ist als bei den beiden anderen Ansätzen. Zu Beginn muss nur einmal der Kontrollflussgraph mit allen nebenläufigen Pfaden aufgebaut werden, was für das Beispielprogramm lediglich sechs Analysen durch aiT benötigt. Ist der KFG bekannt, so können beliebig viele Selektionsdurchläufe durchgeführt werden, ohne aiT erneut starten zu müssen. Deshalb genügen für die Optimierungen aller Cachegrößen die zu Beginn ausgeführten sechs aiT-Aufrufe.

Um die Laufzeit der einzelnen Algorithmen vergleichen zu können, ist ein Testsystem mit folgenden Daten eingesetzt worden:

- AMD Athlon$^{TM}$64 4000+ Prozessor
- 2400 MHz Taktfrequenz
- L1-Cache: 64 + 64 KB (Daten + Instruktionen)
- L2-Cache: 1024 KB
- 3 GB RAM

Das Diagramm in Abbildung 5.9 zeigt für die drei entwickelten Selektionsalgorithmen die benötigte CPU-Zeit für den ADPCM-Benchmark, um für die auf der x-Achse aufgetragenen Cachegrößen die Funktionen zum Locking

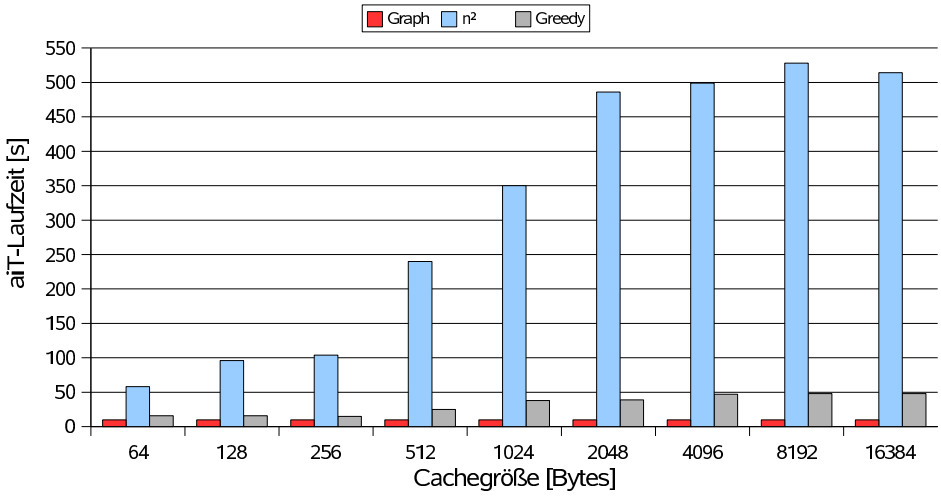



**Abbildung 5.9:** Benötigte CPU-Zeit für den ADPCM-Benchmark

auszuwählen. Es zeigt sich, dass die benötigte Laufzeit der Algorithmen sich äquivalent zur Anzahl der benötigten aiT-Aufrufe verhält. Bei größeren Binärdateien mit vielen Funktionen und konkurrierenden Ausführungspfaden ist der Graph-Algorithmus noch weiter im Vorteil, da die anderen beiden Ansätze immer das gesamte Programm bei jeder WCET-Analyse betrachten. Beim Aufbau des Kontrollflussgraphen durch den Graph-Algorithmus wird aiT nur zweimal zur Analyse des gesamten Programms herangezogen, alle anderen aiT-Aufrufe starten bei konkurrierenden Ausführungspfaden, die zwangsläufig kürzer sind als der Worst-Case-Ausführungspfad. Deshalb ist es wahrscheinlich, dass die benötigte Laufzeit eines aiT-Analysedurchgangs mit steigender Anzahl der Aufrufe für den Graph-Algorithmus abnimmt, da die zu untersuchenden Sub-Kontrollflussgraphen immer kleiner werden.

Der Greedy-Algorithmus benötigt zwischen 16 Sekunden für die Auswahl der Elemente zum Lockdown bei einer Cachegröße von 64 Bytes und 48 Sekunden bei 16 KB Cache. Dem gegenüber steht die Laufzeit des $n^2$-Ansatz von 58 Sekunden bei 64 Bytes Cache bis 514 Sekunden für eine Cachegröße von 16 KB. Der Graph-Algorithmus benötigt für alle Cachegrößen konstant 10 Sekunden und damit bis zu 79% weniger Laufzeit als der Greedy-Algorithmus bzw. sogar 98% weniger Rechenzeit als der $n^2$-Ansatz.

Abbildung 5.10 zeigt die benötigten aiT-Aufrufe für den MPEG-Benchmark aus Kapitel 5.2.5. Der $n^2$-Algorithmus ist wie beim Benchmark in Kapitel 5.2.5 von der Messung ausgeschlossen worden, da die Laufzeit für die Selektion der Elemente auf dem Testsystem mehrere Monate betragen würde. Es ist gut zu erkennen, dass der Graph-Algorithmus immer 52 aiT-Analysevorgänge benötigt, um den Kontrollflussgraphen aufzubauen und danach beliebig viele Funktionen in den Cache zu locken. Beim Greedy-

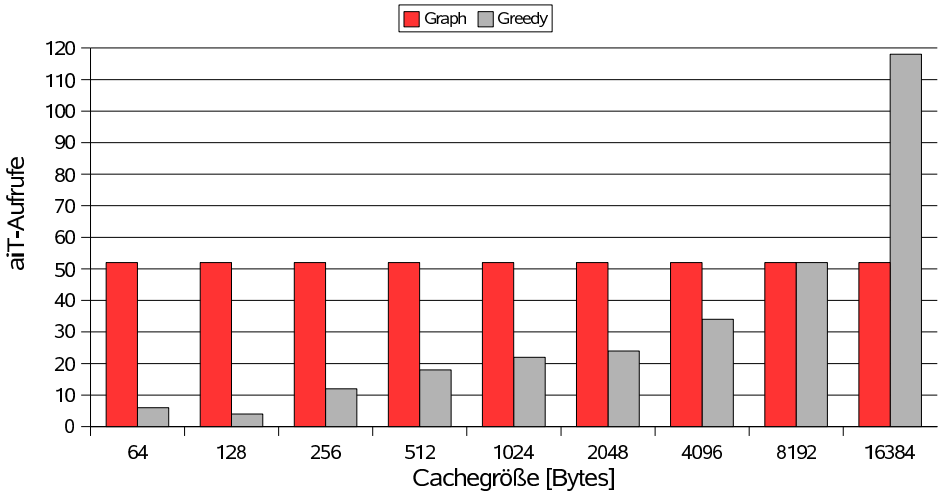



**Abbildung 5.10:** Benötigte aiT-Aufrufe für den MPEG-Benchmark

Algorithmus ist die Zahl der benötigten aiT-Aufrufe nur von der Anzahl der zu lockenden Funktionen abhängig, weshalb bis 8 KB Cachegröße weniger aiT-Aufrufe nötig sind als beim Graph Algorithmus. So werden zwischen 6 aiT-Analysen für eine Cachegröße von 64 Bytes und 118 Analysen für eine Cachegröße von 16 KB benötigt. Für 128 Bytes Cache werden deshalb nur 4 aiT-Aufrufe benötigt, da weniger Funktionen mit größerem Reward gelockt werden als bei 64 Bytes Cachegröße.

Die Rechenzeit, die die Algorithmen beim MPEG2-Benchmark benötigen, um die Menge der zu lockenden Funktionen auszuwählen, ist in Abbildung 5.11 abgebildet. Der Graph-Algorithmus benötigt für alle Cachegrößen konstant 2.179 Sekunden, da die meiste Zeit für die aiT-Analysen aufgewandt wird. Sobald der Kontrollflussgraph aufgebaut ist, geschieht die Selektion der zu lockenden Funktionen in weniger als zwei Sekunden. Obwohl der Greedy-Algorithmus z.T. bedeutend weniger aiT-Aufrufe benötigt (48 weniger bei 128 Bytes Cachegröße), liegt die beste Rechenzeit für einen Selektionsdurchlauf bei 2.994 Sekunden. Im schlechtesten Fall bei 16 KB Cachegröße benötigt der Greedy-Algorithmus sogar 75.572 Sekunden, um mit Hilfe von aiT die Funktionen zum Locking in den Cache auszuwählen.

Hier zeigt sich deutlich, dass die benötigte Zeit, um den Kontrollflussgraphen aufzubauen, nicht von der Anzahl der aiT-Aufrufe abhängt, sondern von der Komplexität des KFG. Nur bei den beiden ersten Analysevorgängen des Graph-Algorithmus beginnt aiT mit *main* als Einsprungspunkt für die Analyse, danach immer bei den Funktionen, an denen ein nebenläufiger Pfad im KFG beginnt. Deshalb wird die benötigte Laufzeit pro Analysevorgang mit steigender Anzahl von aiT-Aufrufen kleiner. Dadurch ist der Graph-

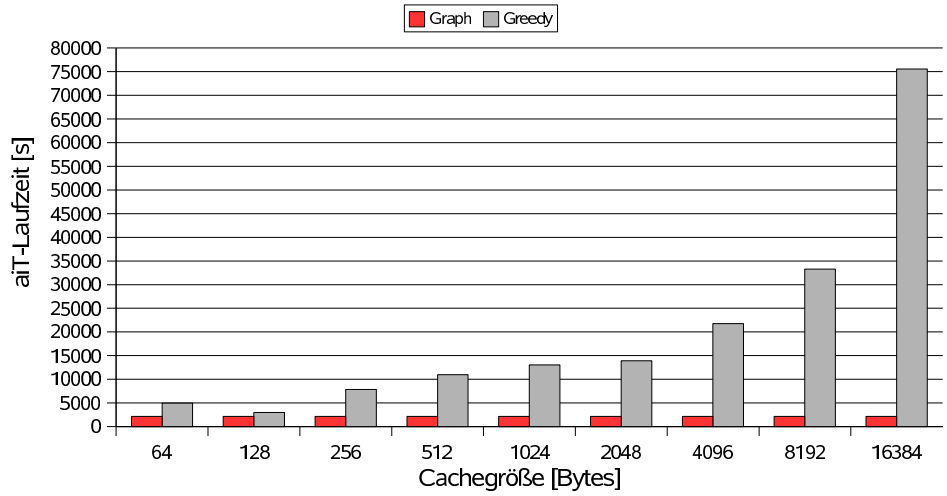



**Abbildung 5.11:** Benötigte CPU-Zeit für den MPEG-Benchmark

Algorithmus zwischen 17% (128 Bytes Cachegröße) und 97% schneller als der Greedy-Algorithmus bei 16384 Bytes Cache.

# Kapitel 6

# Zusammenfassung und Ausblick

Im letzten Kapitel sind die Messergebnisse der in Kapitel 4 vorgestellten Optimierungsverfahren für statisches Cache-Locking präsentiert und ein Vergleich mit bisherigen Techniken angestellt worden. Außerdem ist die nötige Rechenleistung für eine Optimierung zur Entwurfszeit ermittelt worden, um die Anwendbarkeit der Algorithmen zu testen. Diese Ergebnisse sollen im Folgenden kurz zusammengefasst und danach Anregungen für weitere Entwicklungen gegeben werden.

## 6.1 Zusammenfassung

In dieser Arbeit ist der Einfluss von statischem Cache-Locking auf die WCET von Programmen untersucht worden. Dazu sind Funktionen ausgewählt worden, die in den I-Cache einer ARM-Plattform gelockt werden. Alle implementierten Selektionsalgorithmen ermöglichen eine Auswahl von Funktionen zur Entwurfszeit eines Systems, um die Worst-Case Execution Time des zu optimierenden Programms durch Lockdown in den Cache zu senken. Die Qualität der Auswahl hängt bei dem Greedy-Ansatz von der Struktur des Kontrollflusses ab, wohingegen der $n^2$-Ansatz und der Graph-Algorithmus für beliebige Programme vergleichbar gute Lösungen finden. So lässt sich die WCET der analysierten kleineren Benchmarks um 50 - 72% senken. Selbst der vergleichsweise große MPEG2-Encoder profitiert schon von Cachegrößen ab 128 Bytes, so dass sich die Ausführungszeit im Worst-Case bei einer Cachegröße von 16 KB ebenfalls auf ungefähr die Hälfte senken lässt.

Der Vorteil gegenüber bisherigen Optimierungen, die nur entlang des einmal ermittelten Worst-Case-Pfades optimieren, ist die Möglichkeit auch Wechsel im Kontrollfluss zu berücksichtigen. Wieviel ein Programm davon im Vergleich zur Einzelpfadanalyse profitiert, ist stark vom Kontrollfluss des zu optimierenden Programms abhängig. Trotzdem sind Einsparungen

der hier entwickelten Selektionsalgorithmen gegenüber der Einzelpfadanalyse von 20% für die WCET des zu optimierenden Programms ermittelt worden.

Betrachtet man ein System mit gelocktem Cache, wird die Average-Case Execution Time für die meisten Benchmarks durch die auf WCET ausgelegten Verfahren in ähnlichem Maße positiv beeinflusst. Die Reduzierung der Ausführungszeit liegt für die getesteten Programme mit gelockten Funktionen zwischen 30% und 70%, abhängig von der Konstellation der Eingabedaten. Der Multisort-Benchmark zeigt, dass es auch Fälle gibt, wo die ACET so gut wie überhaupt nicht von den WCET-Optimierungen profitiert. Hier konnten lediglich 5% der benötigten Laufzeit im Average-Case eingespart werden. Bei Programmen mit wenigen Funktionen, die sich gegenseitig aus dem Cache verdrängen, ist die erzielte ACET sogar besser als bei einem System mit ungelocktem Cache gleicher Größe. So konnten nocheinmal Einsparungen von bis zu 30% gegenüber einem System mit normal operierendem Cache erzielt werden. Mit steigender Anzahl von Funktionen, die nicht alle in den Cache gelockt werden können, wird der Vorteil kleiner. So ist beim MPEG-Benchmark der ungelockte Cache durschnittlich 15% schneller als die durch Lockdown in den Cache optimierte Version.

Im direkten Vergleich zwischen einem System mit herkömmlichem Cache und einem mit gelocktem Cache erzielen die vorgestellten Selektionsalgorithmen immer eine Reduktion der WCET, die dann von 52% bis 73% reicht. Bei einem Vergleich der erzielten ACET eines Systems mit normal betriebenem Cache verhalten sich die erzielten Ergebnisse durch Cache-Locking indifferent. So erreichen die auf Optimierung der WCET ausgelegten Verfahren der Selektionsalgorithmen bei einigen Benchmarks (G723, Statemate) eine Verkürzung der Laufzeit von 2 - 27% gegenüber einem System mit normalem Cache. Dagegen benötigt ein System mit ungelocktem Cache beim MPEG-Benchmark zwischen 6% und 26% weniger Rechenzeit als beim Lockdown. Die übrigen Benchmarks (ADPCM, Compress) zeigen Vorteile in der ACET bei einzelnen Cachegrößen für ein System mit herkömmlichem Cache oder auch für den gelockten Cache. Mit steigender Cachegröße gleichen sich die Laufzeiten aber soweit an, dass die gleichen Ergebnisse erzielt werden.

Sind bei den Worst-Case und durchschnittlichen Laufzeiten der optimierten Programme keine signifikanten Abweichungen zwischen den einzelnen Selektionsalgorithmen zu erkennen, treten bei der benötigten Rechenleistung für die Auswahl von Elementen zum Locking in den Cache große Unterschiede auf. Die benötigte Anzahl von aiT-Aufrufen und damit die Rechenzeit steigt für den $n^2$-Ansatz quadratisch, so dass dieses Auswahlverfahren nur für Programme mit wenigen Funktionen benutzbar ist. Der Greedy-Algorithmus löst das Selektionsproblem in einer zur Anzahl der Funktionen linearen Zeit. Dafür ist die zusammengestellte Menge der in den Cache zu lockenden Funktionen z.T. weniger gut als bei den anderen beiden Verfahren.

Der Graph Algorithmus benötigt für die meisten hier getesteten Programme die wenigsten aiT-Aufrufe und hat in jedem Fall die kürzeste Ausführungszeit. Die Qualität der getroffenen Auswahl an Funktionen, die in den Cache gelockt werden, ist besser als beim Greedy-Ansatz und bis auf eine Ausnahme bei den Benchmarks so gut wie beim $n^2$-Algorithmus. Diese Ausnahme wird durch den verwendeten WCET-Analyzer aiT verursacht und nicht durch die Funktion der Algorithmen.

Für die Anwendung der Algorithmen ist zusätzlich noch der benötigte Lockdown-Code für die ARM™-Plattform erstellt und in den Startup-Code integriert worden. Dieser Startup-Code ist zusammen mit den drei entwickelten Selektionsalgorithmen in die eigens entwickelte WCETOPT-Toolchain eingefügt worden. Diese Toolchain generiert mit Hilfe der ARM-GCC und aiT in einem Übersetzungsvorgang die optimierte Version einer Binärdatei, die die WCET durch das statische Locking von Teilen des Programms in den I-Cache senkt.

## 6.2 Ausblick

Die hier vorgestellte Toolchain und die dafür implementierten Selektionsalgorithmen ist entwickelt worden, um die Worst-Case Execution Time eines Programms durch statisches Cache-Locking zu optimieren. Dabei ergeben sich weitere Fragestellungen und Erweiterungsmöglichkeiten, die an dieser Stelle angesprochen werden sollen:

### Lockdown von Daten

Alle hier vorgestellten und entwickelten Optimierungen befassen sich mit der Optimierung der Worst-Case Execution Time eines Programms durch das Locking von Befehlssequenzen in den I-Cache. Obwohl dabei gute Ergebnisse erzielt worden ist, begrenzt die Zugriffsgeschwindigkeit auf den langsamen Hauptspeicher eines Systems nach wie vor die WCET, wenn größere Datenmengen verarbeitet werden. Deshalb ist es sinnvoll, den Einfluss von in den Cache gelockten Datenobjekten auf die Laufzeit eines Programms zu untersuchen.

### Dynamisches Lockdown

Die hier entwickelten Selektionsalgorithmen werden alle zur Entwurfszeit eines Systems eingesetzt und locken den Inhalt des Caches statisch. Das bedeutet, dass der Inhalt einmal geladen wird und sich von diesem Zeitpunkt an nicht mehr ändert. Vorstellbar wäre es aber, den Inhalt des gelockten Caches auch während der Laufzeit eines Programms zu ändern, um nicht mehr benötigte Funktionen zu verdrängen und die als nächstes ausgeführte Funktion zu locken. So könnte der vorhandene Cache mehrfach belegt

werden und immer nur der gerade am häufigsten benötigte Teil vorgehalten werden.

### Betrachtung von Kontexten

Aufgrund der beschränkten Schnittstelle zum WCET-Analyzer aiT ist es nur möglich, die Ausführungszeit einzelner Funktionen über alle Kontexte aufsummiert zu ermitteln. Bei Funktionen, deren Worst-Case-Laufzeit stark daten- oder parameterabhängig ist, ist es in einigen Fällen möglich, durch Analyse des Binärprogramms feste Eingabedaten oder Parameter für einzelne Kontexte zu bestimmen. So könnte für diese Kontexte die obere Zeitschranke, die eine Funktion für die Ausführung benötigt, präziser bestimmt werden. Ein Werkzeug um Programmanalysatoren zu generieren, die solche Arbeiten ebenfalls übernehmen können, ist in [Mar99] entwickelt worden.

### WCET-Datenbank

Damit der eingesetzte WCET-Analyzer aiT sichere Angaben über die Worst-Case Execution Time eines Programms oder einer Funktion machen kann, muss der Benutzer für komplexere Schleifen und Rekursionen manuelle Angaben über die Häufigkeit machen. Bei neu implementierten Programmteilen bleibt keine andere Möglichkeit, als diese mit den Eingabedaten für den Worst-Case zu analysieren. Verwendet man jedoch Bibliotheken, macht es Sinn, eine Datenbank mit WCET-Informationen anzulegen, damit Nutzer der Bibliotheken vorausgegangene Analyse-Ergebnisse weiterverwenden können.

Alternativ wäre es möglich, bestehende Compiler um ein `#pragma` Kommando zu erweitern, das die obere Grenze gezielt für Schleifen und Rekursionen im Quelltext angibt. So könnten die Angaben über obere Schranken beim Übersetzungsvorgang direkt mit in die Bibliothek geschrieben werden. Dadurch wäre man nicht auf eine Datenbank angewiesen, weil alle nötigen Informationen direkt in die Bibliothek integriert sind, um weitgehend automatische WCET-Analysen durchzuführen.

### Betrachtung von verbleibendem Cache

Alle bisher durchgeführten Analysen und Benchmarks arbeiten mit Cachegrößen in 2er Potenzen, in die Instruktionen gelockt werden. Bei einer simulierten Größe des Caches von 16 KB kann durchaus ein Teil ungenutzt bleiben, da er von den Funktionen nicht komplett ausgefüllt wird. Für jede Funktion wird ein einzelner Speicherbereich für die Analysen angegeben, dessen Zugriffsverhalten einem Cache entspricht. Dabei ist bisher außer Acht gelassen worden, was der verbleibende Cache für Auswirkungen auf die ACET und WCET Laufzeit von Programmen hat, da für alle Ana-

lysen ein System ohne Cache, aber mit unterschiedlichen Speicherbereichen, angenommen worden ist.

### Verwertbarkeit für andere Optimierungen

Die vorgestellten Selektionsalgorithmen sind entwickelt worden, um Elemente für ein statisches Locking in den Cache auszuwählen und damit die WCET zu senken. Trotzdem lässt sich die Problemstellung der auszuwählenden Elemente auch auf andere Techniken wie Scratchpad-Optimierungen oder andere Granularitäten übertragen, die z.B. Basisblöcke als atomare Einheiten betrachten. Dieses Anwendungsgebiet bedarf jedoch einer gesonderten Betrachtung, genau wie eine Integration direkt in den Compiler, wo beispielweise die Informationen über den Linkvorgang direkt zur Verfügung stehen und nicht erst extrahiert werden müssen.

# Literaturverzeichnis


[Abs04] AbsInt Angewandte Informatik GmbH (Hrsg.): *Worst-Case Execution Time Analyzer aiT for ARM7*. 1.6r1. AbsInt Angewandte Informatik GmbH, 19. November 2004

[Adv98a] Advanced RISC Machines Ltd (ARM) (Hrsg.): *ARM™Software Development Toolkit - Reference Guide*. ARM DUI 0041C. Advanced RISC Machines Ltd (ARM), 1997, 1998. (Version 2.50) . http://www.arm.com

[Adv98b] Advanced RISC Machines Ltd (ARM™): *ARM™Software Development Toolkit - User Guide*. ARM DUI 0040D, 1997, 1998. (Version 2.50) . – 11.1 – 11.27 S. http://www.arm.com

[Adv98c] Advanced RISC Machines Ltd (ARM™): *ARM ELF File Format*. ARM DUI 00101-A, 1997,1998. http://www.arm.com

[Adv98d] Advanced RISC Machines Ltd (ARM™): *Configuring ARM Caches*. ARM DAI 0053B, February 1998. http://www.arm.com

[Adv98e] Advanced RISC Machines Ltd (ARM™): *ARMulator Cache Models*. ARM DAI 0051A, January 1998. http://www.arm.com

[Adv02] Advanced RISC Machines Ltd (ARM™): *ARM920T Technical Reference Manual*. ARM DDI 0151C, 2001, 2002. http://www.arm.com

[Adv03] Advanced RISC Machines Ltd (ARM™): *ARM™Instruction Set Quick Reference Card*. ARM QRC 0001H, 1995-2003. http://www.arm.com

[BBB+05] Benini, Luca ; Bertozzi, Davide ; Bogliolo, Alessandro ; Menichelli, Francesco ; Olivieri, Mauro: *The Journal of VLSI Signal Processing*. Volume 41, Number 2. Springer Netherlands, 2005, Kapitel *MPARM: Exploring the Multi-Processor SoC Design Space with SystemC* , S. 169–182

[BR06] BURGUIÈRE, Claire ; ROCHANGE, Christine: *History-based Schemes and Implicit Path Enumeration*, 2006 (6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis)

[CLRS01] CORMEN, Thomas H. ; LEISERSON, Charles ; RIVEST, Ronald L. ; STEIN, Clifford: *Introduction to Algorithms*. 2nd edition. MIT Press, 2001. – ISBN 0–262–53196–8

[CMW00] CAIN, Harold W. ; MILLER, Barton P. ; WYLIE, Brian J.: *Lecture Notes In Computer Science* . 2000. – ISBN 3–540–67956–1, Kapitel *A Callgraph-Based Search Strategy for Automated Performance Diagnosis* , S. 108 – 122

[Cor04] CORMEN, Thomas H.: *Algorithmen - eine Einführung*. Oldenbourg, 2004. – ISBN 3–486–27515–1

[CPIM05] CAMPOY, Antonio M. ; PUAUT, Isabelle ; IVARS, Angel P. ; MATAIX, Jose V. B.: *Cache contents selection for statically-locked instruction caches: an algorithm comparison*, 2005 (17th Euromicro Conference on Real-Time Systems (ECRTS'05))

[CRD00] CHIOU, Derek ; RUDOLPH, Larry ; DEVADAS, Srinivas: *Dynamic Cache Partitioning via Columnization*, 2000 (Proceedings of Design Automation Conference, Los Angeles)

[ESS99] ERPENBACH, Edwin ; STAPPERT, Friedhelm ; STROOP, Joachim: *Compilation and Timing Analysis of Statecharts Models for Embedded Systems*, 1999 (International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES))

[G72] *G723 Encoder*. Sun Microsystems Inc. `http://www.sun.com/`

[GCC] *GNU Compiler Collection*. http://gcc.gnu.org/,

[Gus] GUSTAFSSON, Andreas: *egypt - create call graph from gcc RTL dump*. `http://www.gson.org/egypt/egypt.html`

[HF04] HECKMANN, Reinhold ; FERDINAND, Christian: *Worst-Case Execution Time Prediction by Static Program Analysis*, 2004 (18th International Parallel and Distributed Processing Symposium (IPDPS'04))

[Mar99] MARTIN, Florian: *Generating Program Analyzers*, Technische Fakultät der Universität des Saarlandes, Diss., Juni 1999

[Mar06] MARWEDEL, Peter: *Embedded System Design*. 2nd edition. Springer Verlag, 2006. – ISBN 0–387–29237–3

[MPA] *MPARM*. Micrel Lab - D.E.I.S. / Universita' di Bologna. `http://www-micrel.deis.unibo.it/sitonew/research/mparm.html`

[MRT] *The Worst-Case Execution Time analysis project*. `http://www.mrtc.mdh.se/projects/wcet/wcet_bench/`. – Mälardalen Research and Technology Centre (MRTC)

[MSS96] *MPEG-2 Encoder / Decoder*. MPEG Software Simulation Group. `http://www.mpeg.org/MSSG/`. Version: 1.2, July 19. 1996

[OAR04] OAR CORPORATION: *RTEMS 4.6.6 On-Line Library*. `http://www.rtems.com`, 1988-2004

[PD02] PUAUT, Isabelle ; DECOTIGNY, David: *Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems*, 2002 (23rd IEEE Real-Time Systems Symposium (RTSS'02))

[SNU] *SNU real-time benchmarks suite*. Seoul National University. `http://archi.snu.ac.kr/realtime/benchmark`

[SPE95] *SPEC CPU95 Benchmarks*. Standard Performance Evaluation Corporation. `http://www.spec.org/cpu95/`. Version: 1995

[The02] THEILING, Henrik: *Control Flow Graphs for Real-Time System Analysis*, Universität des Saarlandes, Diss., 2002. – "Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis"

[TIS93] TIS COMMITTEE (Hrsg.): *Tool Interface Standard (TIS)*. Version 1.1. TIS Committee, October 1993. – Portable Formats Specification

[VLX03a] VERA, Xavier ; LISPER, Björn ; XUE, Jingling: *Data Cache Locking for Higher Program Predictability*, 2003 (International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)), S. 272 – 282

[VLX03b] VERA, Xavier ; LISPER, Björn ; XUE, Jingling: *Data Caches in Multitasking Hard Real-Time Systems*, 2003 (IEEE Real-Time Systems Symposium)

[VWM04] VERMA, Manish ; WEHMEYER, Lars ; MARWEDEL, Peter: *Cache-Aware Scratchpad Allocation Algorithm*, 2004 (Design, Automation and Test in Europe - DATE)

[WM04] WEHMEYER, Lars ; MARWEDEL, Peter: *Influence of Memory Hierarchies on Predictability for Time Constrained Embedded*

*Software* ECRTS, 2004 (4th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis)

[WM05] WEHMEYER, Lars ; MARWEDEL, Peter: *Influence of Onchip Scratchpad Memories on WCET prediction*, 2005 (Design, Automation and Test in Europe - DATE)

[Wol06] WOLF, Jürgen: *C von A bis Z*. Galileo Computing, 2006. – ISBN 3–89842–643–2

[Zie05] ZIEGLER, Peter-Michael: Ein Europäer in Paris. In: *c'T - Magazin für Computertechnik* (2005), Juni, Nr. 14/2006, S. 62 ff

# Abkürzungsverzeichnis

| | |
|---|---|
| ACET | Average-Case Execution Time |
| ADPCM | Adaptive Differenzielle Pulse-Code Modulation |
| AMD | Advanced Micro Devices, Inc. |
| ARM™ | Advanced RISC Machines Ltd. |
| CG | Callgraph (dt.: *Aufrufgraph*) |
| D-Cache | Data Cache (dt.: *Datencache*) |
| DFS | Depth-First Search |
| DRAM | Dynamic Random Access Memory |
| ELF | Executable and Linkable Format |
| GCC | GNU Compiler Collection |
| GDL | Graph Description Language |
| I-Cache | Instruction Cache (dt.: *Befehlscache*) |
| IP | Integer Programming (dt.: *Ganzzahlige Programmierung*) |
| IPET | Implicit Path Enumeration Technique |
| KFG | Kontrollflussgraph |
| LZ | Lempel-Ziv |
| MMU | Memory Management Unit |
| PCM | Pulse-Code Modulation |
| RAM | Random Access Memory |
| RISC | Reduced Instruction Set Computing |
| ROM | Read OnlyMemory |
| SPEC | Standard Performance Evaluation Corporation |
| SRAM | Static Random Access Memory |
| STARC | STAtechart Realtime-Code generator |
| WCET | Worst-Case Execution Time |
| WCETOPT | WCET OPTimizer (Toolchain für WCET-Optimierung) |
| WS | Waitstates (dt.: *Wartezyklen*) |

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Liste der Algorithmen