# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

Die Anzahl der weltweit produzierten elektrischen Geräte wächst kontinuierlich an. Um komplexe Aufgaben leicht und kostengünstig zu erfüllen, enthalten manche dieser Geräte Prozessoren, die Programme ausführen. Beispielsweise können in einem Mobiltelefon Programme für Datenbanken und Spiele enthalten sein. Ein grundlegendes Problem bei Mobiltelefonen ist der begrenzte Energievorrat, der die Einsatzzeit bestimmt.

Die Entwicklung effizienter Programme gehört zu den wichtigsten Aufgaben innerhalb der Softwareentwicklung. Die heutigen Programme werden oft in einer Hochsprache, wie beispielsweise C oder Java, geschrieben. Hochsprachen gewährleisten schnelle Entwicklungszeiten und einen portablen Programmcode. Ein Compiler übersetzt diesen Programmcode in einen Maschinencode. Abschließend führt ein Prozessor innerhalb eines Systems diesen Code aus.

Ein Compiler beeinflusst bei der Übersetzung entscheidend die Qualität des Programmcodes, indem er eine bestimmte Übersetzung wählt. Unterschiedliche Optimierungsmethoden innerhalb des Compilers erzeugen nach verschiedenen Kriterien ein günstiges Programm. Das Ziel von Optimierungsmethoden kann die Laufzeit, der benötigte Speicherplatz, der Energieverbrauch oder auch die Leistungsaufnahme sein. Es gibt sehr viele verschiedene Compileroptimierungen (z.B. in [Sch99] vorgestellt), die sich auch unterschiedlicher Ansätze bedienen. Einige Klassen von Optimierungstechniken und einige Optimierungsverfahren sind in der Abbildung 1.1 dargestellt.

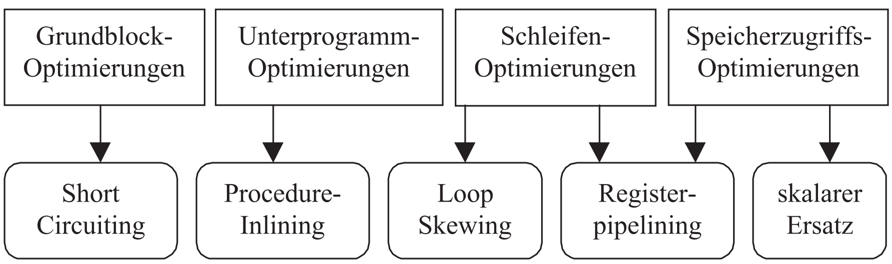


Abbildung 1.1: Einige Compileroptimierungen (basierend auf [Sch99])

Für diese Arbeit wird die Optierungstechnik Registerpipelining implementiert und untersucht. Sie optimiert Speicherzugriffe innerhalb von Schleifen. Diese Technik wurde für diese Diplomarbeit gewählt, weil Speicherzugriffe sehr viel Energie benötigen und da dieses Verfahren bisher noch nicht unter dem Aspekt des Energiesparens untersucht wurde.

Das Hauptoptimierungsziel innerhalb dieser Diplomarbeit ist die Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining. Der betrachtete ARM-Prozessor ist der ARM7TDMI-Prozessor, gehört zu der Familie der Mikroprozessoren und ist ein Mitglied der Advanced RISC Machines (ARM) Familie.

## 1.1 Ziel der Diplomarbeit

Das Ziel der Diplomarbeit besteht darin, den Energiebedarf von Programmen für den ARM-Prozessor durch Registerpipelining zu reduzieren. Zu diesem Zweck wurde zuerst eine geeignete Arbeitsumgebung errichtet, die die einfache Analyse eines Programms ermöglicht. Hierzu wurde ein Trace-Analyzer entwickelt, der die Eigenschaften eines Programms während der Ausführung untersucht. Ein Simulator bildet die Programmausführung des zu untersuchenden Systems nach und der Trace-Analyzer bewertet anschließend diesen Vorgang. Wegen der Simulation wurde darauf geachtet, dass die ermittelten Werte möglichst genau mit den tatsächlichen Werten übereinstimmen. Zum Abschluss der Diplomarbeit wurden Messungen durchgeführt, um die Qualität der eingesetzten Compiler zu beurteilen. Der Trace-Analyzer ermöglicht zusätzlich die Untersuchung einzelner Compileroptimierungen, indem diese innerhalb eines Compilers aktiviert oder deaktiviert werden. Daher liefert der Trace-Analyzer einen grundlegenden Beitrag für die Bewertung von Optimierungsverfahren nach den verschiedenen Optimierungskriterien.

Um schließlich das Ziel des geringeren Energieverbrauchs während der Programmausführung zu erreichen, müssen geeignete Optimierungstechniken in die Compiler eingebaut werden. Eine vielversprechende Technik, die den Energieverbrauch reduzieren kann, besteht in dem Optimierungsverfahren Registerpipelining. Um dessen Auswirkungen zu untersuchen, wurde diese Optimierungstechnik in den Compiler 'arm12cc' integriert, der an der Universität Dortmund Fakultät Informatik am Lehrstuhl 12 entwickelt wird. Dieser Compiler erzeugt einen Assemblercode für den ARM7TDMI-Prozessor und kann insbesondere den Energieverbrauch optimieren.

Diese Diplomarbeit leistet ebenfalls einen Beitrag zur effizienten Umsetzung der Optimierungstechnik Registerpipelining. Einige unterschiedliche Variationen der Optimierungstechnik werden entwickelt und bewertet. Im Vordergrund steht das Ziel das Verfahren effizient unter dem Gesichtspunkt der Energieoptimierung umzusetzen. Die besonderen Möglichkeiten des ARM7TDMI-Prozessors werden untersucht und berücksichtigt.

Bestimmte Analysen helfen bei der Durchführung der Optimierungstechnik Registerpipelining. Sie untersuchen die Programme, die optimiert werden sollen, und liefern Informationen, die die einzelnen Optimierungen unterstützen. Eine hierfür geeignete Array-Datenflussanalyse wurde aus der Literatur ausgewählt und an das Verfahren Registerpipelining und an den ARM7TDMI-Prozessor angepasst. Diese angepasste Array-Datenflussanalyse wird untersucht und um einige Möglichkeiten ergänzt. Die durchgeführten Erweiterungen sind teilweise auch auf allgemeine Verfahren übertragbar. Weiterhin werden noch andere Methoden für eine Programmanalyse vorgestellt, die die Einsatzmöglichkeiten der Optimierungstechnik Registerpipelining erweitern.

Das Verfahren Registerpipelining gehört zu den Speicherzugriffsoptimierungen und verwendet die speziellen Eigenschaften einer Speicherhierarchie. Deswegen untersucht diese Diplomarbeit auch die Verwendung der Speicherhierarchie unter dem Gesichtspunkt einer Optimierungstechnik. Der Schwerpunkt liegt aber bei dem Verfahren Registerpipelining.

## 1.2 Übersicht

Kapitel 2 dieser Diplomarbeit dient als Einführung in das Themengebiet Low Power. Es erfolgt eine Motivation des Themas und die Möglichkeiten der Compileroptimierungen werden erläutert. Daran anschließend stellt Kapitel 3 den ARM7TDMI-Prozessor vor. Insbesondere werden die Eigenschaften des Prozessors und die zur Verfügung stehende Speicherhierarchie betrachtet. Das Kapitel 4 erklärt die Optimierungstechnik Registerpipelining. Die Funktionsweise des Verfahrens und verschiedene Umsetzungsmöglichkeiten werden erläutert. Kapitel 5 beschreibt die Arbeitsumgebung und die darin vorgenommenen nötigen eigenen Entwicklungen. Es wird gezeigt, wie ein Programm analysiert und bewertet werden kann. Das Kapitel 6 stellt beispielhaft die Integration der Optimierungstechnik Registerpipelining in einem Compiler vor. Hierzu werden die einzelnen Implementierungsphasen und ihre Besonderheiten beschrieben. Kapitel 7 erläutert die $\delta$-Array-Datenflussanalyse, die das Verfahren Registerpipelining unterstützen kann. Der Aufbau und die Eigenschaften der Analyse werden beschrieben. Mit Hilfe des entwickelten Trace-Analyzers wurden Leistungsmessungen durchgeführt, die in Kapitel 8 aufgeführt sind. Insbesondere werden dort die Auswirkungen der Optimierungstechnik Registerpipelining auf ein Programm analysiert. Abschließend beschreibt Kapitel 9 zusammenfassend die Ergebnisse dieser Diplomarbeit und einen Ausblick auf zukünftige Entwicklungen.

Im Anhang sind die im Rahmen dieser Diplomarbeit implementierten Programme dokumentiert und der Thumb-Befehlssatz des ARM7TDMI-Prozessors wird dargestellt.

# Kapitel 2

# Low Power

Ein niedriger Energieverbrauch gehört neben einer hohen Geschwindigkeit oder einem geringen Speicherplatzbedarf zu den wichtigsten Optimierungszielen innerhalb der Programmentwicklung für eingebettete Systeme. Der Energieverbrauch sei definiert als die benötigte elektrische Energie während der Programmausführung.

Dieses Kapitel stellt eine Einführung in das Themengebiet Low Power dar. Zuerst wird das Thema motiviert und dann werden die elektrotechnischen Grundlagen und die Ursachen des Energieverbrauchs erläutert. Abschließend werden Möglichkeiten aufgezeigt, wie ein Compiler den Energieverbrauch reduzieren kann.

## 2.1 Motivation

Ein niedriger Energieverbrauch ist in Systemen wichtig, auf denen eine der folgenden Eigenschaften zutrifft (nach [SS99]):

- begrenzte Energiezufuhr
  Beispielsweise bei mobilen Systemen, die nur einen begrenzten Energiespeicher besitzen, kann eine Reduzierung des Energieverbrauchs die Dauer der Einsatzfähigkeit des Systems verlängern.

- hohe Energiekosten
  Eine Reduzierung des Energieverbrauchs kann Energiekosten und Produktionskosten einsparen. Beispielsweise kann in Systemen, in denen Solarzellen eingesetzt werden, eine Reduzierung des Energieverbrauchs auch zu einer Reduzierung der Anzahl der benötigten Solarzellen führen.

- kritische Wärmeentwicklung
  Ein niedriger Energieverbrauch führt zu einer geringeren Wärmeentwicklung. Hierdurch können besondere Kosten für die Kühlung und für bestimmte Gehäuse entfallen. Zusätzlich kann auch der Platzbedarf des Systems verringert werden.

- hohe Zuverlässigkeit
  Eine geringe Wärmeentwicklung führt zu einer höheren Zuverlässigkeit, da das Material weniger beansprucht wird. Dies resultiert beispielsweise aus einer Reduzierung der Elektromigration.

Insgesamt gibt es also sehr viele Situationen, in denen eine Reduzierung des Energieverbrauchs wichtig sein kann.

## 2.2 Grundlagen

Dieses Unterkapitel erläutert einige wichtige Grundlagen basierend auf [Her89]. Die Leistung $P$ ist definiert als das Produkt aus der Spannung $U$ und dem Strom $I$.

$$P = U \cdot I \tag{2.1}$$

Die Einheit von $P$ ist $1W$(Watt)$= 1V$(Volt)$\cdot 1A$(Ampere). Das Produkt aus der Leistung $P$ und der Zeit $t$ sei definiert als die Energie $E$ (in der Literatur oft auch als Energie $W$ dargestellt).

$$E = P \cdot t \tag{2.2}$$

Die Energie besitzt die Einheit $J$(Joule)$= Ws = VAs$. Insgesamt kann die Energie aus dem Produkt von Spannung, Strom und Zeit gebildet werden, wie es in der Gleichung 2.3 dargestellt ist.

$$E = U \cdot I \cdot t \tag{2.3}$$

## 2.3 Ursachen für den Energieverbrauch

Prinzipiell kann der Energieverbrauch nicht beliebig reduziert werden, denn im allgemeinen benötigen aktive Schaltungen auch Energie. Die Energiekosten innerhalb von CMOS-Schaltungen werden nach [SS99] unter anderem durch folgende Gegebenheiten verursacht:

- Switching Power
  Das Laden und Entladen von Kapazitäten während eines Schaltvorgangs kostet Energie.

- Short Circuit Power
  Beim Schalten von Transistoren treten kurzzeitig Kurzschlussströme auf, die Energie verbrauchen.

- Leakage Power
  Selbst wenn keine Schaltaktivitäten vorhanden sind wird Energie verbraucht. Dies ist aber ein geringer Anteil im Vergleich zu dem Energieverbrauch bei hohen Schaltaktivitäten.

Die elektrische Energie kann innerhalb von Schaltungen beispielsweise in Wärmeenergie oder in elektrische und magnetische Felder umgesetzt werden.

## 2.4 Reduktionsmöglichkeiten

Ein günstiger Energieverbrauch innerhalb eines Systems kann beim Entwurf von Schaltungen oder auch bei der Softwareentwicklung bedacht werden. Diese Diplomarbeit beschäftigt sich hauptsächlich mit einer geeigneten Softwareentwicklung.

Die heutigen Programme werden oft in einer Hochsprache geschrieben. Ein Compiler übersetzt den Programmcode in einem Maschinencode. Er besitzt deshalb viele Einflussmöglichkeiten, um den Energieverbrauch eines Systems während der Programmausführung zu senken.

Aus Kapitel 2.2 wird deutlich, dass das Produkt aus Spannung, Strom und Zeit die Energie ergibt. Deshalb kann eine Reduzierung dieser Werte den Energieverbrauch positiv beeinflussen:

- Spannung
  Ein Compiler besitzt keinen Einfluss auf diesen Wert. Die Versorgungsspannung ist für viele Bauteile auch fest vorgeschrieben.

- Strom
  Ein Compiler sollte für eine gute Übersetzung diejenigen Instruktionen wählen, die einen geringen Strom verursachen. Hierzu müssen detaillierte Informationen über die einzelnen Instruktionen vorliegen.

- Zeit
  Ein wichtiger Bestandteil der Energie ist die benötigte Zeit für die Programmausführung. Ein Compiler sollte deswegen ein Programm generieren, das möglichst wenig Zeit beansprucht. Optimierungsverfahren, die die benötigte Ausführungszeit senken, reduzieren oft auch den Energieverbrauch.

Ein Compiler kann von der Energie die Komponenten Zeit und Strom optimieren und sollte eine Übersetzung wählen, die eine gute Kombination aus beiden Werten bildet. In der folgenden Aufzählung sind nach [SS99] und [EM98] einige allgemeine Möglichkeiten aufgeführt, die den Energieverbrauch positiv beeinflussen können:

- Reduzierung der Schalthäufigkeit
  Die Reduzierung der Schalthäufigkeit gehört zu den wichtigsten Optimierungsmöglichkeiten für einen niedrigen Energieverbrauch. Ein Compiler sollte demnach ein Programm generieren, das möglichst wenig Schaltaktivitäten verursacht.

- Wahl günstiger Speicher
  Unterschiedliche Speicher benötigen auch unterschiedlich viel Energie. Beispielsweise verursachen Zugriffe auf einen langsamen Speicher oft einen hohen Energieverbrauch.

- Abschalten von Schaltungsteilen
  Um Schaltungsteile abschalten zu können, muss der Prozessor bestimmte Eigenschaften besitzen, damit der Compiler einen geeigneten Programmcode erzeugen kann.

- Speicherinhalte optimieren
  Wenn beispielsweise das Speichern einer '1' kostengünstiger ist als das Speichern einer '0', dann kann dies für eine Optimierung verwendet werden.

Manche dieser allgemeinen Optimierungstechniken lassen sich auf den Compiler übertragen. Die Reduzierung der Schalthäufigkeit steht hierbei im Vordergrund. Einige Compileroptimierungen werden in der nachfolgenden Aufzählung aufgeführt:

- Instruction Scheduling
  Instruction Scheduling kann die Energiekosten für das Spilling (Ein- und Auslagern von Werten in Registern) reduzieren, da sich die Bereiche verändern, in denen die Registerinhalte lebendig sind. Zudem können auch die Kosten für interne Zustandswechsel (siehe Kapitel 5.2) minimiert werden.

- Strength Reduction
  Die Optimierungstechnik Strength Reduction kann den Energieverbrauch senken, indem sie teure Befehle durch kostengünstigere ersetzt. Beispielsweise benötigt eine Multiplikation mit der Konstanten 4 oft mehr Energie als eine entsprechende Shift-Operation.

- Registerpipelining
  Das Verfahren Registerpipelining reduziert die Anzahl der Speicherzugriffe. Da Speicherzugriffe normalerweise hohe Schaltaktivitäten verursachen, kann hierdurch viel Energie gespart werden. Somit lohnt sich Wahrscheinlich die Anwendung der Optimierungstechnik Registerpipelining innerhalb eines Compilers.

Insgesamt existieren sehr viele Möglichkeiten, um den Energieverbrauch zu senken.

# Kapitel 3

# ARM7TDMI-Prozessor

Im Rahmen dieser Diplomarbeit wurde in einem Compiler das Verfahren Registerpipelining implementiert. Ein Ziel dieser Diplomarbeit war es, die Auswirkungen der Optimierungstechnik auf Programme zu untersuchen, die auf dem ARM7TDMI-Prozessor ausgeführt werden. Da einige Architekturmerkmale für die Optimierungstechnik relevant sind, stellt dieses Kapitel den Prozessor und seine Eigenschaften vor.

## 3.1 Aufbau

Dieses Unterkapitel beschreibt den ARM7TDMI-Prozessor nach [ARML95]. Er gehört zu der Familie der Mikroprozessoren und ist ein Mitglied der Advanced RISC Machines (ARM) Familie. Die Abbildung 3.1 zeigt die Architektur des Prozessors.

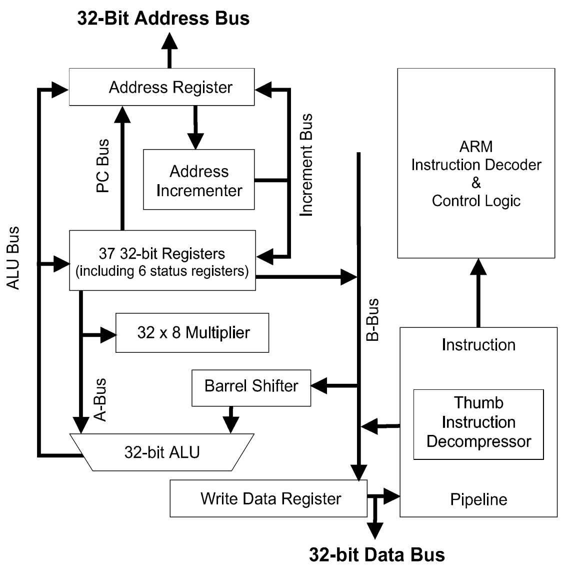


Abbildung 3.1: ARM7TDMI-Prozessor [Atm99a]

Die Architektur basiert auf dem Reduced Instruction Set Computer (RISC) Prinzip. Der Prozessor besitzt eine 3-stufige Pipeline, die aus Instruction-Fetch, Instruction-Decode und Execute besteht. Weiterhin stellt er zwei Instruktionssätze zur Verfügung:

- 32-Bit Arm-Instruktionssatz
- 16-Bit Thumb-Instruktionssatz

Ein Programmierer kann frei zwischen diesen beiden Instruktionssätzen wählen. Der Arm-Instruktionssatz bietet meistens Vorteile bei der Geschwindigkeit, während der Thumb-Instruktionssatz oft Vorteile bei der Programmgröße und beim Energieverbrauch besitzt. Dieser Unterschied entsteht beispielsweise dadurch, dass die Thumb-Instruktionen nur 16-Bit lang sind, weshalb das Bussystem nur 16-Bit übertragen muss, wodurch der Energieverbrauch reduziert wird.

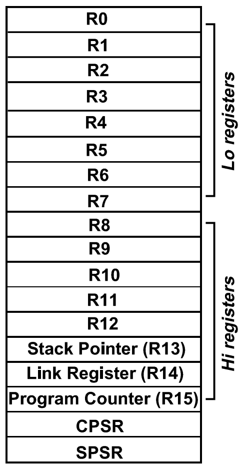


Abbildung 3.2: Thumb-Registersatz [ARML95]

Im Rahmen dieser Diplomarbeit wurde für die Implementierung der Optimierungstechnik Registerpipelining im Hinblick auf den niedrigen Energieverbrauch der Thumb-Instruktionssatz (siehe auch Anhang B) gewählt. Er besitzt einige Einschränkungen gegenüber dem Arm-Instruktionssatz. Der Codeumfang ist kleiner und anstatt 14 allgemein einsetzbarer Register (R0-R13) beinhaltet der Thumb-Instruktionssatz nur noch 8 dieser Register (R0-R7). Diese 8 Register gehören zu dem unteren Registersatz, wie es in der Abbildung 3.2 dargestellt ist. Die hohen Register R8 bis R12 können nur über wenige Operationen angesprochen werden. Das hohe Register R13 beinhaltet den Stack Pointer, R14 das Link Register und R15 den Programcounter.

Die beiden Registersätze werden aufeinander abgebildet, weshalb ihre Inhalte auch bei einem Wechsel zwischen Arm- und Thumb-Instruktionen innerhalb eines Programms erhalten bleiben. Beide Instruktionssätze unterstützen die Datentypen byte (8-Bit), halfword (16-Bit) und word (32-Bit).

## 3.2 EB01 Evaluation Kit

Für diese Diplomarbeit stand aus dem EB01 Evaluation Kit ein ATMEL-Evaluationboard zur Verfügung. Die Abbildung 3.3 zeigt den ATMEL AT91M40400, der sich auf dem Evaluationboard befindet. Auf diesem Chip wurde der ARM-Kern, Peripherie und ein 4-KB-RAM Speicher integriert.

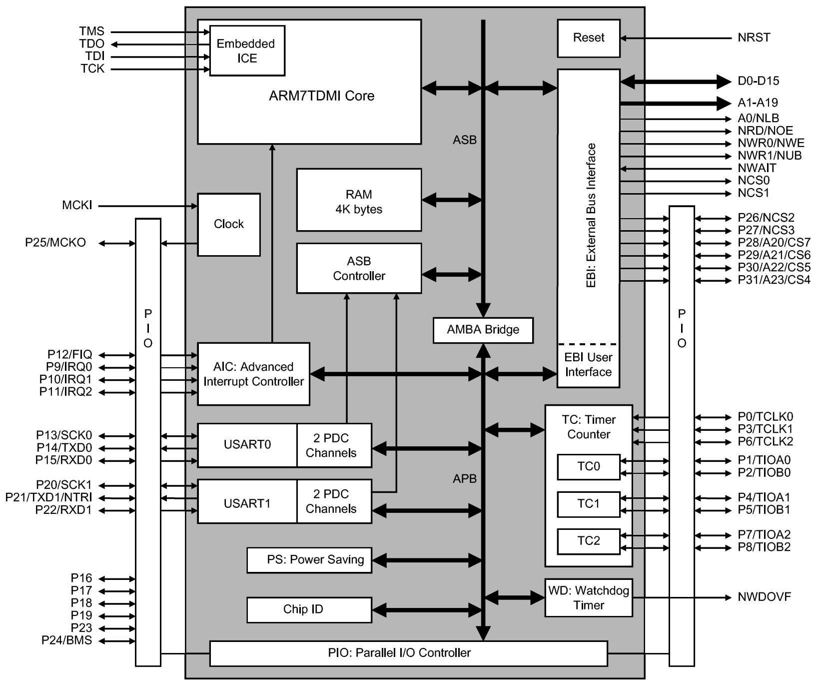


Abbildung 3.3: ATMEL AT91M40400 [Atm99c]

Weiterhin steht extern auf dem Evaluationboard ein 512-KB-RAM-Speicher und ein 128-KB-ROM-Speicher zur Verfügung. Der ARM7TDMI-Prozessor kann sowohl internen Speicher als auch externen Speicher über ein Bus-System adressieren. Der interne Speicher ist wesentlich kleiner und wird schneller und energiesparender angesprochen.

## 3.3 Speicherhierarchie

Dem ARM7TDMI-Prozessor steht bei Verwendung des Thumb-Instruktionssatzes die in der folgenden Aufzählung dargestellte Speicherhierarchie aus dem Blickwinkel der Optimierungstechnik Registerpipelining zur Verfügung. Die Register sind bei dieser Aufzählung ebenfalls ein Teil der Speicherhierarchie.

1. unterer Registersatz (R0-R7)

2. verwendbarer oberer Registersatz (R8-R12)

3. 4-KB On-Chip-Speicher (32-Bit,SRAM)

4. 512-KB externer Speicher (16-Bit,SRAM)

5. 128-KB externer Speicher (16-Bit,Flash-ROM)

Die Ordnung innerhalb dieser Speicherhierarchie ergibt sich aus den Fähigkeiten der einzelnen Speichereinheiten und aus der Eigenschaft, wie stark sie den zusätzlichen Speicher, den die Optimierungstechnik Registerpipelining benötigt (siehe Kapitel 4), unterstützen. Der Speicher, der weiter oben in der Speicherhierarchie angeordnet ist, zeichnet sich durch einen niedrigeren Energieverbrauch und durch eine höhere Geschwindigkeit aus. Der Unterschied zwischen den Registersätzen besteht nur im verwendbaren Befehlssatz. Der ROM-Speicher wird aufgrund seiner read-only Eigenschaft unten angeordnet.

Für einen Compiler ist es schwierig, die oberen Register geeignet einzusetzen. Der kommerzielle Standardcompiler 'tcc' für den ARM7TDMI-Prozessor verwendet die hohen Register R8 bis R12 nicht. Das Problem ist, dass für sie nur ein begrenzter Befehlssatz zur Verfügung steht. Die Inhalte der hohen Register können beliebig mit den 'mov'-Operationen zwischen den Registern R0 und R12 kopiert werden. Deshalb können sie effizient für das Spilling und auch für die Optimierungstechnik Registerpipelining eingesetzt werden.

# Kapitel 4

# Speicherzugriffsoptimierungen

Eine Optimierung der Speicherzugriffe kann den Energieverbrauch mindern. Diese Eigenschaft wird beispielsweise in [EM98] beschrieben.

Die entscheidenden Instruktionen für die folgenden Optimierungen sind zusammenfassend in der Tabelle 4.1 dargestellt. Die Werte für den Strom basieren auf Angaben aus [SS00]. Die einzelnen Werte in den Tabellen 4.1 bis 4.3 beziehen sich auf das ATMEL-Evaluationboard aus Kapitel 3.2.

| Instruktion | Speicherort | | Strom (mA) | | Speichergröße (Bytes) | |
|---|---|---|---|---|---|---|
| | Instruktion | Daten | AT91-M40400 | externer Speicher | Instruktion | Daten |
| mov | On-Chip | keine | 49,1 | 1,2 | 2 | 0 |
| | extern | | 42,4 | 119,2 | 2 | 0 |
| ldr | On-Chip | On-Chip | 50,2 | 1,2 | 2 | 4 |
| | | extern | 42,1 | 83,6 | 2 | 4 |
| | extern | On-Chip | 47,8 | 75,0 | 2 | 4 |
| | | extern | 44,2 | 115,8 | 2 | 4 |

Tabelle 4.1: Strom und Speicherplatz der wichtigen Instruktionen

Die Tabelle 4.2 zeigt die gesamten Energiekosten der Instruktionen aus Tabelle 4.1 unter der Voraussetzung, dass der Prozessor mit 33MHz getaktet wird und eine Spannung von 3,3V anliegt. Die externen Speicherzugriffe können einen Anteil von 86% am gesamten Energieverbrauch besitzen. Durch die Verwendung des On-Chip-Speichers anstelle des externen Speichers kann der Energieverbrauch teilweise um Faktor den 7,2 (112/15,42) reduziert werden. Der On-Chip-Speicher sollte deswegen dem externen Speicher vorgezogen werden. Weiterhin benötigt eine Kopieroperation ('mov'-Operation) weniger Energie und Taktzyklen als eine Speicherleseoperation ('ldr'-Operation), die in dem gleichen Speicher liegt. Die Tabelle 4.3 verdeutlicht diesen Zusammenhang. Beispielsweise entspricht der Energieverbrauch einer Speicherleseoperation im On-Chip-Speicher, die Daten aus dem externen Speicher liest, etwa dem Energieverbrauch von 15 'mov'-Operationen. Durch die gegebenen Leistungswerte in Tabelle 4.2 kann nur bei dieser Speicherkombination der Energieverbrauch wesentlich besser mit Hilfe von 'mov'-Operationen optimiert werden als die Anzahl der Taktzyklen.

| Instruktion | Speicherort Instruktion | Speicherort Daten | Energie (J) ($\cdot 10^{-9}$) | Taktzyklen | Leistung (mW) | Speicheranteil (extern) Energie | Speicheranteil (extern) Taktzyklen | Speicheranteil (extern) Leistung |
|---|---|---|---|---|---|---|---|---|
| mov | On-Chip-Speicher | - | 5,03 | 1 | 166,0 | 2% | 0% | 2% |
| ldr | | On-Chip | 15,42 | 3 | 119,6 | 2% | 0% | 2% |
| | | extern | 75,42 | 6 | 414,8 | 79% | 50% | 67% |
| mov | externer Speicher | - | 32,32 | 2 | 533,3 | 86% | 50% | 74% |
| ldr | | On-Chip | 49,12 | 4 | 405,2 | 68% | 25% | 61% |
| | | extern | 112,00 | 7 | 528,0 | 86% | 57% | 71% |

Tabelle 4.2: Zusammensetzung der Instruktionen (Taktzyklen nach [SS00])

| Speicherort der Instruktion | Speicherort der Daten | Anzahl äquivalenter 'mov'-Operationen bzgl. Energie | Anzahl äquivalenter 'mov'-Operationen bzgl. Taktzyklen |
|---|---|---|---|
| On-Chip-Speicher | On-Chip-Speicher | 3,1 | 3,0 |
| | externer Speicher | 15,0 | 6,0 |
| externer Speicher | On-Chip-Speicher | 1,5 | 2,0 |
| | externer Speicher | 3,5 | 3,5 |

Tabelle 4.3: Gewichtung einer 'ldr'-Operation mit 'mov'-Operationen

Der Grundgedanke bei vielen Speicherzugriffsoptimierungen ist es, Speicherzugriffe, die weit unten in der Speicherhierarchie liegen, möglichst weit oben anzuordnen. Ein Speicherinhalt kann in einem Register abgelegt werden, damit der Inhalt schnell und energiesparend erreichbar ist. Aktuelle Compiler verwenden verschiedene Zuordnungsstrategien, um Variablen in Registern zu halten. Viele Compiler ignorieren aber die Möglichkeit, auch Inhalte eines Arrays in Registern zu speichern, da es z.B. sehr schwierig ist, Datenabhängigkeiten zwischen Array-Elementen präzise festzustellen. Der Einsatz einer entsprechenden Optimierungstechnik, wie z.B. Registerpipelining, kann aber die Anzahl der Speicherzugriffe verringern und die Programmqualität verbessern.

Dieses Kapitel stellt die beiden Speicherzugriffsoptimierungen Redundant Load Elimination und Registerpipelining vor, die die Anzahl der Speicherzugriffe reduzieren. Die Grundidee der beiden Techniken ist eine Optimierung der Speicherlesezugriffe auf Arrays. Die Datenstruktur 'Array' beinhaltet eine Adressierung eines zusammenhängenden Speicherbereichs durch einen Index. Speicherzugriffe auf Instruktionen werden auch optimiert, indem manche Instruktionsspeicherzugriffe in einem Programm entfallen.

## 4.1 Redundant Load Elimination

Die Redundant Load Elimination (RLE) beseitigt redundante Speicherlesezugriffe. Dieses Unterkapitel beschreibt das Verfahren basierend auf [RB96] und [Fra99].

**Definition 4.1.1** Eine Speicherleseoperation ist an einer Position $k$ in einem Programm *partiell/total redundant*, wenn entlang einiger/aller Pfade zu $k$ der Speicherinhalt schon bekannt war und zwischenzeitlich nicht verändert wurde.

**Beispiel 4.1.2** Zwei redundante Speicherlesezugriffe

```
for( i=1; i <1000; i++)
{
  R1=a[i];        // (1)
  ...             // beliebiger Zwischencode
  a[i+1]=R4;      // (2)
  ...             // Zwischencode ohne Speicherzugriffe
  R3=a[i+1];      // (3)
}
```

Das Beispiel 4.1.2 enthält zwei redundante Speicherlesezugriffe. Die Variablen, die mit dem Buchstaben 'R' beginnen, kennzeichnen Register. Der Zugriff (3) ist total redundant, da der Wert von $a[i+1]$ bereits in R4 stand. Der Zugriff (1) ist dagegen partiell redundant, weil bis auf den Fall $i$=1 der Wert von $a[i]$ in R3 des vorangehenden Schleifendurchlaufs steht.

Das Beispiel 4.1.3 zeigt den speicherzugriffsoptimierten Programmcode des Beispiels 4.1.2. Der Inhalt von $a[i+1]$ wird zuerst in Register R8 zwischengespeichert, damit das Verfahren Redundant Load Elimination auch dann funktioniert, wenn das Register R4 zwischenzeitlich verändert wird. Um das total redundante Load zu eliminieren, wird in (3) der Speicherzugriff durch einen entsprechenden Registerzugriff ersetzt. Der partiell redundante Speicherzugriff in (1) wird hingegen durch einen Registerzugriff und durch einen Schleifenprolog ersetzt. Der Schleifenprolog ist für den Fall $i$=1 wichtig, um sicherzustellen, dass in R8 der richtige Wert steht.

**Beispiel 4.1.3** Elimination der redundanten Speicherlesezugriffe

```
R8=a[1];          // Schleifenprolog
for( i=1; i <1000; i++)
{
  R1=R8;          // (1)
  ...             // beliebiger Zwischencode
  a[i+1]=R4;      // (2)
  R8=R4;
  ...             // Zwischencode ohne Speicherzugriffe
  R3=R8;          // (3)
}
```

Folgende Werte kann die Optimierungstechnik Redundant Load Elimination verändern:

- Taktzyklen
  Die Anzahl der benötigten Taktzyklen kann reduziert werden, da eine Speicherleseoperation oft mehr Taktzyklen benötigt als eine 'mov'-Operation und da eventuell Teile der Berechnung der Indexfunktion innerhalb der Schleife entfallen.

- Leistung
  Die Umsetzung der Optimierung innerhalb einer Schleife erfolgt ausschließlich mit 'mov'-Operationen, weshalb die Leistungsaufnahme teilweise gesenkt wird.

- Energieverbrauch
  Da die Anzahl der Taktzyklen und manchmal auch die Leistung optimiert wird, kann der Energieverbrauch reduziert werden.

- Programmgröße
  Die Programmgröße vergrößert sich meistens aufgrund des Prologs und der zusätzlich eingefügten 'mov'-Operationen.

**Definition 4.1.4** *Induktionsvariablen* sind Variablen, deren Wert sich aus der Anzahl der Iterationen von umschließenden Schleifen ableiten lässt.

**Definition 4.1.5** Die *Indexfunktion* sei die Funktion, die angibt, wo in Abhängigkeit zu der Induktionsvariablen der Zugriff auf ein Array stattfinden soll.

**Definition 4.1.6** Die *Array-Referenz* $a[f(i)]$ besteht aus der Indexfunktion $f(i)$ und dem Array $a$.

Im Beispiel 4.1.3 ist $i$ eine Induktionsvariable, '$i$+1' eine Indexfunktion an der Stelle (2) und $a[i+1]$ eine Array-Referenz.

Zusätzliche Kosten ergeben sich oft aus dem Gebrauch von weiteren Registern. Bei dem Beispiel 4.1.3 könnte eventuell das Register R8 in weiteren Optimierungsschritten noch eingespart werden, wobei dann aber eventuell die Lebensdauer von R4 verlängert wird. Der Registerdruck innerhalb der Schleife erhöht sich.

Da die Programmoptimerungen mit einfachen Kopieroperationen durchgeführt werden, kann die Redundant Load Elimination hohe Register sinnvoll verwenden.

### 4.1.1 Verwendung des On-Chip-Speichers

Die Idee der Optimierungstechnik Redundant Load Elimination kann auch auf andere Speicher einer Speicherhierarchie angewendet werden. Bei dem ATMEL-Evaluationboard bietet sich eine Optimierung von Speicherzugriffen auf den externen Speicher durch Speicherzugriffe auf den On-Chip-Speicher an. Der optimierte Programmcode des Beispiels 4.1.2 ist im Beispiel 4.1.7 dargestellt. Das Array $a$ liegt im externen Speicher und $m$ sei ein Speicherplatz im On-Chip-Speicher.

Bei dieser Art der Umsetzung der Optimierungstechnik erhöht sich der Energieverbrauch und die Taktzyklenanzahl gegenüber der Implementierungsvariante aus dem Beispiel 4.1.3. Diese Umsetzung benötigt aber keine zusätzlichen Register, weshalb manchmal eine Durchführung lohnenswert seien kann. Gegenüber dem ursprünglichen Programm in Beispiel 4.1.2 kann der Energieverbrauch, die Leistungsaufnahme und die Taktzyklenanzahl gesenkt werden.

**Beispiel 4.1.7** Zwei redundante Speicherlesezugriffe

```
m=a[1];
for( i=1; i <1000; i++)
{
  R1=m;          // (1)
  ...            // beliebiger Zwischencode
  a[i+1]=R4;     // (2)
  m=R4;
  ...            // Zwischencode ohne Speicherzugriffe
  R3=m;          // (3)
}
```

Eine genaue Betrachtung der Beispiele 4.1.3 und 4.1.7 zeigt, dass sie bis auf die Ausdrücke 'R8' und '$m$' identisch sind. Eine gute Registerallokation könnte bei hohem Registerdruck eine Transformation von Beispiel 4.1.3 zu Beispiel 4.1.7 durchführen, weshalb die Verwendung des On-Chip-Speichers insbesondere beim Spilling berücksichtigt werden sollte.

## 4.2 Registerpipelining

Registerpipelining (RP) verallgemeinert die Redundant Load Elimination bzgl. größerer Iterationsdistanzen.

**Definition 4.2.1** Die *Iterationsdistanz* bezeichnet den Abstand zweier Indexfunktionen in Schleifeniterationen.

Wenn die Optimierungstechnik Redundant Load Elimination angewendet wird, liegt eine Iterationsdistanz von 0 oder 1 vor, da spätestens nach zwei Iterationen der ursprüngliche Wert überschrieben wird. In dem Beispiel 4.1.3 ist der Wert von $a[i+1]$ an der Position (2) mit der Iterationsdistanz 1 an der Stelle (1) verfügbar. Um bei der Optimierungstechnik Registerpipelining größere Iterationsdistanzen zu ermöglichen, werden zusätzliche Register benötigt, die Werte zwischenspeichern, die sonst während der Iterationen verloren gingen.

### 4.2.1 Einfaches Registerpipelining

Dieses Unterkapitel stellt eine mögliche Umsetzung des Verfahrens Registerpipelining nach [ED93] und [Fra99] vor. Beispiel 4.2.2 zeigt eine Speicherzugriffsoptimierung innerhalb einer Schleife, die mit Hilfe einer Registerpipeline realisiert wird.

Um eine Registerpipeline aufzubauen, werden zahlreiche Kopierbefehle am Ende des Schleifenkörpers positioniert. Die Tiefe der Registerpipeline ist in dem

Beispiel 4.2.2 gleich 2. Die Pipeline stellt sicher, dass ein bestimmter Wert (hier $a[i+2]$) auch nach zwei Schleifeniterationen vorhanden ist (hier in R7). Diese Eigenschaft wird bei (1) verwendet und führt dort zur Entfernung eines redundanten Speicherlesezugriffs. Zur Initialisierung der Registerpipeline wird vor dem Schleifenkopf ein Prolog eingefügt, der die Korrektheit der Pipeline bei der ersten Schleifeniteration sicherstellt. Die Tabelle 4.4 zeigt die Funktionsweise der Pipeline. Bestimmte Speicherzugriffe füllen die Pipeline bei R8 und während der einzelnen Iterationen werden die Werte an das Ende der Pipeline durchgereicht.

**Beispiel 4.2.2** Speicherzugriffsoptimierung mit einer Registerpipeline

```
// ursprüngliche Schleife

for( i=0; i<100; i++)
{
  R3=a[i];        // (1)
  R2=a[i+2];      // (2)

  R6+=R3+R2;// (3)
}
```

```
// Schleife mit Registerpipelining
R7=a[0];          // Schleifen-
R8=a[1];          // prolog
for( i=0; i<100; i++)
{
  R3=R7;          // (1)
  R2=a[i+2];      // (2)
  R5=R2;
  R6+=R3+R2;// (3)

  R7=R8;          // Register-
  R8=R5;          // pipeline
}
```

| Iteration | R7 | R8 |
|---|---|---|
| $i$=0 | $a[0]$ | $a[1]$ |
| $i$=1 | $a[1]$ | $a[2]$ |
| $i$=2 | $a[2]$ | $a[3]$ |
| ... | ... | ... |

Tabelle 4.4: Funktionsweise einer Registerpipeline

Beispiel 4.2.3 zeigt das Programm aus Beispiel 4.2.2 in einer Assemblerdarstellung. Die einzelnen Befehle werden in dem Anhang B beschrieben. Mehrere Operationen übersetzen eine Array-Referenz. Zuerst wird die Indexfunktion berechnet und danach erfolgt der Speicherzugriff. Das Verfahren Registerpipelining halbiert in dem Beispiel 4.2.3 die Anzahl der 'ldr'-Operationen innerhalb der Schleife.

**Beispiel 4.2.3** Assemblerdarstellung des Beispiels 4.2.2

```
// ursprüngliche Schleife                 // Schleife mit Registerpipelining
    mov  R4,#0        // i=0;                 mov  R4,#0        // i=0;
                                              mov  R2,#0        // R7=a[0];
                                              add  R2,SP
                                              lsl  R0,R4,#2
                                              ldr  R7,[R2,R0]
                                              add  R2,#4        // R8=a[1];
                                              ldr  R2,[R2,R0]
                                              mov  R8,R2
LL3                                       LL3
    cmp  R4,#100      // i<100?               cmp  R4,#100      // i<100?
    bge  LL1                                  bge  LL1
    mov  R2,#0        // R3=a[i];             mov  R3,R7        // R3=R7;
    add  R2,SP
    lsl  R1,R4,#2
    ldr  R3,[R2,R1]
    mov  R2,#0        // R2=a[i+2];           mov  R2,#0        // R2=a[i+2];
    add  R2,SP                                add  R2,SP
    add  R1,R4,#2                             add  R1,R4,#2
    lsl  R1,R1,#2                             lsl  R1,R1,#2
    ldr  R2,[R2,R1]                           ldr  R2,[R2,R1]
                                              mov  R5,R2        // R5=R2;
    add  R7,R3,R2     // R6+=R3+R2            add  R7,R3,R2     // R6+=R3+R2
    add  R6,R7                                add  R6,R7

                                              mov  R7,R8
                                              mov  R8,R5

    add  R4,R4,#1     // i++;                 add  R4,R4,#1     // i++;
    b    LL3                                  b    LL3
LL1                                       LL1
```

### 4.2.2 Optimierungseigenschaften

Äußere Kriterien beeinflussen die Optimierungstechnik Registerpipelining. In der nachfolgenden Aufzählung sind positiv wirkende Bedingungen aufgeführt:

- geringer Registerdruck
  Das Verfahren Registerpipelining benötigt oft zusätzliche Register. Bei einem hohen Registerdruck könnte zusätzliches Spilling die Codequalität wesentlich verschlechtern.

- geeignete Position der Speicherzugriffe
  Der Kontrollfluss innerhalb eines Programms muss für eine Optimierung berücksichtigt werden. Beispiel 4.2.4 zeigt diesen Sachverhalt. Wenn $i$ immer gleich $k$ ist, dann kann Registerpipelining den Zugriff auf $a[i]$ durch

einen früheren Zugriff auf $a[i+2]$ ersetzen. Das Problem ist aber, dass der Wert bei (1) nicht unbedingt erzeugt wird, weshalb dieser nicht immer wiederverwendet werden kann. Zwar könnte ein zusätzlich eingefügter 'else'-Zweig sicherstellen, dass dieser Wert immer produziert wird, aber die Effizienz des Verfahrens würde in Abhängigkeit von den Ausführungswahrscheinlichkeiten stark leiden. Ähnlich verhält es sich auch mit $a[i]$ und $a[i\text{-}2]$, wo eine Wiederverwendung bei $a[i\text{-}2]$ nicht unbedingt immer stattfindet. Auch hier hängt die Effizienz von Registerpipelining von den Ausführungswahrscheinlichkeiten ab.

**Beispiel 4.2.4** Kontrollstrukturen und Registerpipelining

```
for ( i=10; i<20; i++)
{
    ...
   if (i==k) n=a[i+2]; // (1)
   m=a[i];
   if (i==j) l=a[i-2];    // (2)
    ...
}
```

Allgemein ist es oft günstig, wenn beide Speicherzugriffe auf der untersten Ebene der Kontrollstrukturen innerhalb einer Schleife liegen. Sonst hängt die Effizienz der Optimierungstechnik Registerpipelining von den Ausführungswahrscheinlichkeiten ab.

- kleine Distanzen zwischen den Optimierungen
  Die Kopieroperationen müssen bei kleinen Iterationsdistanzen die Werte nicht so weit transportieren. Deswegen reichen wenige Operationen aus, um eine Optimierung durchzuführen.

- aufwendige Indexfunktionen für einen Speicherzugriff
  Bei dem Ersetzen von einem Speicherlesezugriff durch einen preiswerteren Registerzugriff entfällt manchmal zusätzlich die Berechnung der Indexfunktion innerhalb der Schleife, wenn der entsprechende Teil nicht mehr benötigt wird.

- relativ teure Datenspeicherzugriffe
  Die Speicherzugriffe, die eingespart werden können, sollten möglichst teuer sein im Verhältnis zu einem Registerzugriff. Die Kosten für einen Speicherzugriff hängen von dem verwendeten Speicher ab. Beispielsweise ist beim ATMEL-Evaluationboard ein Speicherzugriff auf den externen 512-KB Speicher teurer als ein Zugriff auf den On-Chip-Speicher.

- günstige Instruktionskosten
  Geeignete Instruktionskosten beeinflussen die Optimierung positiv. Ein günstiger Wert für die Instruktionslesekosten beim Instruktion-Fetch des

Prozessors ist beispielsweise abhängig von der Distanz zwischen den Speicherzugriffen und von der Indexfunktion. In der Instruction-Decode- und Execute-Phase sollte eine 'ldr'-Operation wesentlich teurer sein als eine 'mov'-Operation.

- viele Schleifendurchläufe
  Bei sehr vielen Schleifendurchläufen können die Kosten für einen Schleifenprolog vernachlässigt werden.

- bestimmte nachfolgende Optimierungsverfahren
  Nach der Durchführung der Optimierungstechnik Registerpipelining können einige andere Optimierungsverfahren die Programmqualität positiv beeinflussen. Es handelt sich hierbei um Optimierungsmöglichkeiten, die erst durch die Anwendung von Registerpipelining entstehen. Folgende Verfahren gehören beispielsweise zu dieser Gruppe:

  - Copy Propagation ([Sch99])
    Sie verwendet Kopieroperationen für eine Optimierung.
  - Useless Code Elimination ([Sch99])
    Die Optimierungstechnik Useless Code Elimination entfernt sinnlosen Code.

  Die Tabelle 4.5 zeigt ein Beispiel für die Anwendung der beiden Optimierungstechniken. Registerpipelining optimiert die Speicherlesezugriffe mit Hilfe von Kopieroperationen. Im Anschluss können die Optimierungstechniken Copy Propagation und Useless Code Elimination in diesem Beispiel die Kopieroperationen entfernen. Auf die Darstellung einer geeigneten Schleife wurde in der Tabelle verzichtet.

| C ohne RP | C mit RP | | |
|---|---|---|---|
| | nach RP | nach Copy Propagation | und nach Useless Code Elimination |
| $a[i]=i;$<br>$b=a[i];$<br>$c=a[i];$<br>$d=b+c;$ | $a[i]=i;$<br>$b=i;$<br>$c=i;$<br>$d=b+c;$ | $a[i]=i;$<br>$b=i;$<br>$c=i;$<br>$d=i+i;$ | $a[i]=i;$<br>$d=i+i;$ |

Tabelle 4.5: Anwendung von Copy Propagation und Useless Code Elimination

- das Fehlen bestimmter Optimierungstechniken
  Das Fehlen der folgenden Optimierungsverfahren kann das Verhalten der Optimierungstechnik Registerpipelining positiv beeinflussen:

  - Redundant Load Elimination
    Diese Technik vernichtet teilweise die Optimierungsmöglichkeiten, die auch das Verfahren Registerpipelining optimieren kann. Es existiert eine Überschneidung der Optimierungsmöglichkeiten.

- Common Subexpression Elimination ([Sch99])
  Die Common Subexpression Elimination optimiert zum Teil die Indexfunktionen, weshalb ihre Berechnung teilweise nicht mehr entfallen kann.
- Feldkontraktion und skalarer Ersatz ([Sch99])
  Einfache Variablen können Speicherzugriffe auf Arrays ersetzen und die Indexfunktionen können eventuell ebenfalls vereinfacht werden.
- Induction Variable Elimination
  Die Induction Variable Elimination spaltet Induktionsvariablen auf, wodurch Abhängigkeiten zwischen den einzelnen Speicherzugriffen verloren gehen können.

Insgesamt gibt es sehr viele Sachverhalte, die die Optimierungstechnik Registerpipelining beeinflussen können. In dieser Aufzählung sind neun positiv wirkende Kriterien aufgeführt. Diese Vielfalt erschwert letztlich eine genaue Bewertung der Optimierungstechnik.

Das Verfahren Registerpipelining kann folgende Werte eines Systems oder eines Programms verändern:

- Taktzyklen
  Die Anzahl der benötigten Taktzyklen kann reduziert werden, da eine Speicherleseoperation mehr Taktzyklen benötigt als eine 'mov'-Operation. Außerdem können Teile der Berechnung der Indexfunktion entfallen.

- Leistung
  Die Umsetzung der Optimierung innerhalb einer Schleife erfolgt ausschließlich mit 'mov'-Operationen und diese vermindern manchmal die Leistungsaufnahme.

- Energieverbrauch
  Eine Verringerung des Energieverbrauchs ist möglich, da sowohl die Taktzyklenanzahl als auch die Leistungsaufnahme reduziert werden kann.

- Programmgröße
  Die Programmgröße erhöht sich meistens aufgrund des Prologs und der zusätzlich eingefügten 'mov'-Operationen.

### 4.2.3 Prolog- und Epiloggenerierung

Ein Prolog initialisiert eine Registerpipeline. Hierzu werden bestimmte Speicherinhalte in die entsprechenden Register der Pipeline kopiert. Beispiel 4.2.5 veranschaulicht einige Möglichkeiten für eine Prologgenerierung für die Optimierungstechnik Registerpipelining.

Die Anzahl der durchgeführten Speicherlesezugriffe ist in beiden Prologen aus dem Beispiel 4.2.5 gleich groß, wenn $j \geq 10$ ist. Sonst führt der zweite Prolog weniger Speicherzugriffe durch. Durch das Abrollen der Schleife am Schleifenanfang können auch noch weiter Vorteile entstehen, indem beispielsweise manche Instruktionen entfallen können. Nachteilig wirkt sich die erhöhte Programmgröße aus.

**Beispiel 4.2.5** Prologgenerierung

| // * nicht optimiert * | // * erster Prolog * | // * zweiter Prolog * |
|---|---|---|
| | R0=a[0]; | if(5 >=j) goto _L1 |
| | R1=a[1]; | R10=a[0]; |
| | R2=a[2]; | R0=R10; |
| | R3=a[3]; | a[5]=R10+1; |
| | R4=a[4]; | if(6 >=j) goto _L1 |
| | | R10=a[1]; |
| | | R1=R10; |
| | | a[6]=R10+1; |
| | | ... |
| for (i=5; i <j; i++) | for (i=5; i <j; i++) | for (i=10; i <j; i++) |
| { | { | { |
| R10=a[i-5]; | R10=R0; | R10=R0; |
| a[i]=R10+1; | ... | ... |
| } | } | } |
| | | _L1: |

Ein geeigneter Epilog am Ende einer Schleife kann manchmal ebenfalls die Programmqualität verbessern. Viele der Kopieroperationen für die Optimierungstechnik Registerpipelining könnten in einem Epilog entfallen. Voraussetzung für die Anwendbarkeit ist ein genaues Wissen über die Anzahl der durchzuführenden Iterationen der Schleife. Der Energieverbrauch und die Anzahl der Taktzyklen könnten hierdurch zusätzlich reduziert werden.

Problematisch wirken sich dagegen Schleifen aus, die mehrere Ein- und Ausgänge besitzen. Ein Schleifeneingang kennzeichnet sich dadurch, dass direkt nach ihm die Schleife durchlaufen wird. Ein Schleifenausgang kann beispielsweise durch einen 'goto'-Befehl im inneren einer Schleife realisiert werden, der zu einer außerhalb der Schleife liegenden Instruktion springt. Ein Prolog muss für jeden Schleifeneingang existieren, wodurch der benötigte Programmspeicher vergrößert wird. Für jeden Schleifenausgang kann ebenfalls ein passender Epilog generiert werden. Weiterhin bildet jeder Schleifeneingang und -ausgang einen Punkt, an dem das Verfahren Registerpipelining ein korrektes Programm erzeugen muss. Um dies zu erreichen, wird der Prolog oder der Epilog geeignet angepasst.

### 4.2.4 Verbessertes Registerpipelining

Eine Array-Datenflussanalyse vergrößert die Anzahl der Einsatzmöglichkeiten der Optimierungstechnik Registerpipelining (siehe Kapitel 6.4). In [RB96] und in [Fra99] wird ein aufwendiges Verfahren beschrieben, dass die Speicherzugriffe und alle Kopieroperationen innerhalb der Pipeline optimal platzieren kann. Dieses Verfahren benötigt unter anderem vier Array-Datenflussanalysen, die Ausführungswahrscheinlichkeiten aller Pfade im Kontrollflussgraphen und eine Möglichkeit, um ein Flussproblem zu lösen. Das Hauptziel dieser Diplomarbeit ist die Untersuchung der Auswirkungen des Verfahrens Registerpipelining auf den Energieverbrauch. Hierzu muss aber nicht eine Umsetzung der Optimierungstechnik analysiert und implementiert werden, die zum Teil einen optimalen Code generiert.

Eine einfache, effiziente und im Rahmen dieser Diplomarbeit implementierte Umsetzung des Verfahrens Registerpipelining wird in diesem Kapitel beschrieben. Sie erzielt bessere Ergebnisse als die im Kapitel 4.2.1 aufgeführte Implementierungsvariante und benötigt keine exakten Ausführungswahrscheinlichkeiten der Instruktionen. Dementsprechend werden die Speicherzugriffe und die Kopierbefehle nicht optimal platziert. Das Verfahren kann sowohl die unteren als auch die hohen Register des ARM7TDMI-Prozessors verwenden.

In dem Kapitel 4.2.2 sind zahlreiche Kriterien aufgeführt, die das Verfahren Registerpipelining positiv beeinflussen. Viele dieser Kriterien lassen sich aber kaum verändern. Folgende Überlegungen können trotzdem zu einer Verbesserung der Optimierungstechnik führen:

- Eine bestimmte Anzahl an Kopieroperationen wird immer benötigt, um einen Wert über eine bestimmte Iterationsdistanz zu übergeben. Deshalb können nur andere Kopieroperationen eingespart werden oder die Iterationsdistanz muss verkleinert werden.
- Die Kopieroperationen für die Registerpipeline müssen nicht unbedingt am Ende der Schleife stehen.
- Die Speicherzugriffe könnten ebenfalls umplatziert werden.

Das Ziel ist es, die Distanz auch bzgl. Iterationen zwischen der Erzeugung eines Wertes und seiner Verwendung möglichst energiesparend für eine spätere Ausführung zu überbrücken. Im Beispiel 4.2.2 wird in der mit Registerpipelining optimierten Schleife an der Stelle (2) ein Wert erzeugt, der bei (1) zwei Iterationen später verwendet wird. Im weiteren Verlauf dieses Kapitels spielt dieser Wert, der erzeugt und dann für das Verfahren Registerpipelining anstelle eines Speicherzugriffes verwendet wird, eine entscheidene Rolle. Drei Verbesserungsmöglichkeiten, die die Distanz zwischen den Werten verkürzen können, sind in der nachfolgenden Aufzählung aufgeführt:

- Die Verwendung sollte früh in einer Schleife durchgeführt werden.
- Die Lebensdauer des erzeugten Werts sollte verlängert werden.
- Die Erzeugung sollte spät innerhalb einer Schleife stattfinden.

Die Kopieroperationen werden bei dieser Umsetzung des Verfahrens immer direkt bei den Speicherzugriffen platziert, die bei jeder Iteration der Schleife durchlaufen werden. Die sinnvollen Varianten bei einer Umsetzung der Optimierung entstehen aus den möglichen Positionen der Pipeline und aus dem Versuch, die Pipeline möglichst kurz zu halten. Insgesamt ergeben sich die vier Implementierungsvarianten 'VNormal', 'VVerwendung', 'VErzeugung' und 'VBeides', die nachfolgend beschrieben werden. Der Buchstabe 'V' kennzeichnet explizit die Varianten dieser Implementierung von Registerpipelining.

### Einzelne Optimierungen

Die von Registerpipelining grundlegenden Optimierungsvarianten für einzelne Speicherzugriffsoptimierungen sind in den Tabellen 4.6 und 4.7 dargestellt. Alle Beispiele in den Tabellen 4.6 bis 4.13 besitzen normalerweise einen passenden Prolog und eine umschließende Schleife, in der die Induktionsvariable $i$ um 1 erhöht wird. Die Markierungen '//1','//2' und '//3' bezeichnen innerhalb einer Tabelle die gleiche Programmposition.

Die Implementierungsvariante 'Einfach' stellt die Umsetzung der Optimierungstechnik aus Kapitel 4.2.1 dar. Bei den folgenden optimierten Beispielen wird bei der Variablen $d$ der für Registerpipelining entscheidende Wert erzeugt, der später für die Erzeugung der Variablen $n$ verwendet wird. Die Tabelle 4.6 enthält die Varianten, bei denen eine Verwendung vor einer Erzeugung stattfindet (das $n$ liegt vor dem $d$) und die Tabelle 4.7 Beispiele, wo die Erzeugung vor der Verwendung auftritt (das $d$ liegt vor dem $n$).

| C ohne RP | C mit RP (Registerpipelining) | | | | |
|---|---|---|---|---|---|
| | Einfach | VNormal | VVerwendung | VErzeugung | VBeides |
| ... | ... | ... | ... | ... | ... |
| $n$=$a$[$i$-3];//1 | $n$=R0;//1 | $n$=R0;//1 | //1 | $n$=R0;//1<br>R0=R1;<br>R1=$d$; | //1 |
| ... | ... | ... | ... | ... | ...<br>$n$=R1;<br>R1=$d$; |
| $d$=$a$[$i$];//2 | $d$=$a$[$i$];//2<br>R3=$d$; | $d$=$a$[$i$];//2<br>R0=R1;<br>R1=R2;<br>R2=$d$; | $d$=$a$[$i$];//2<br>$n$=R1;<br>R1=R2;<br>R2=$d$; | $d$=$a$[$i$];//2 | $d$=$a$[$i$];//2 |
| ... | ...<br>R0=R1;<br>R1=R2;<br>R2=R3; | ... | ... | ... | ... |

Tabelle 4.6: Beispiele für eine Verwendung vor der Erzeugung

Die Variante 'VNormal' positioniert die Registerpipeline direkt nach der Erzeugung des Wertes. Zur Umsetzung dieser Variante können auch hohe Register innerhalb der Pipeline bis auf das erste und letzte Register verwendet werden. Im Vergleich zu der Variante 'Einfach' kann manchmal bereits ein Register und eine Kopieroperation einspart werden, wie bei dem Beispiel in der Tabelle 4.6. Interessant ist, dass diese Einsparung nur dann funktioniert, wenn die Verwendung vor der Erzeugung durchgeführt wird.

Die Durchführung der Varianten 'VVerwendung', 'VErzeugung' und 'VBeides' ist abhängig von den Lebensbereichen der Variablen $n$ und $d$. Die Variante 'VVerwendung', die auf der Variante 'VNormal' basiert, positioniert zusätzlich die Verwendung des Wertes in die Registerpipeline. Die Grundidee ist eine frühe Erzeugung des Wertes, um die Registerpipeline am Anfang zu optimieren. Hierfür darf der verwendete Wert aber nur in einem bestimmten Bereich leben,

| C ohne RP | C mit RP (Registerpipelining) | | | | |
|---|---|---|---|---|---|
| | Einfach | VNormal | VVerwendung | VErzeugung | VBeides |
| … | … | … | … | … | …<br>$n$=R1;<br>R1=$d$; |
| $d$=$a$[$i$];//1 | $d$=$a$[$i$];//1<br>R2=$d$; | $d$=$a$[$i$];//1<br>R0=R1;<br>R1=R2;<br>R2=$d$; | $d$=$a$[$i$];//1<br>$n$=R1;<br>R1=R2;<br>R2=$d$; | $d$=$a$[$i$];//1 | $d$=$a$[$i$];//1 |
| … | … | … | … | … | … |
| $n$=$a$[$i$-2];//2 | $n$=R0;//2 | $n$=R0;//2 | //2 | $n$=R0;//2<br>R0=R1;<br>R1=$d$; | //2 |
| … | …<br>R0=R1;<br>R1=R2; | … | … | … | … |

Tabelle 4.7: Beispiele für eine Erzeugung vor der Verwendung

der in Abhängigkeit der Reihenfolge der Speicherzugriffe bestimmt wird. Zwei bzw. drei dieser Bereiche ergeben sich aus den Grenzen der ursprünglichen Position und der neuen Position der Verwendung. Für den Fall, den die Tabelle 4.6 darstellt, darf die Variable $n$ in dem nicht optimierten Programm nur zwischen der Markierung '//1' und '//2' leben. In der Tabelle 4.7 darf das $n$ nur in dem Bereich zwischen den Markierungen '//2' und '//1' leben. Dies stellt den Bereich von der Markierung '//2' bis zum Schleifenende und vom Schleifenanfang bis zu '//1' dar. Eine Liveness-Analyse kann eine Bestimmung der Lebendigkeiten durchführen. Insgesamt kann hierdurch eine zusätzliche Kopieroperation und ein Register gegenüber der Variante 'VNormal' eingespart werden.

Die Implementierungsvariante 'VErzeugung' erhöht dagegen eventuell die Lebendigkeit des erzeugten Wertes. Die Registerpipeline wird nach der Verwendung platziert. Die Idee ist, die Lebensdauer des erzeugten Wertes zu verlängern und die Pipeline an ihrem Ende zu verkürzen. Es muss überprüft werden, ob der erzeugte Wert solange leben darf, um die Optimierung durchzuführen. Das Programm darf den Wert zwischen seiner Erzeugung und der geplanten Wiederverwendung nicht verändern. Wenn die Verwendung vor der Erzeugung stattfindet, darf der Wert zudem am Schleifeneingang nicht leben. Auch hier wird ein Register und ein Kopierbefehl gegenüber der Variante 'VNormal' eingespart.

Weiterhin ist eine Kombination der beiden vorherigen Fälle möglich. Die Variante 'VBeides' platziert die Registerpipeline direkt vor der Erzeugung und zusätzlich wird die Verwendung des Wertes vorgezogen. Durchführbar ist diese Implementierungsmöglichkeit, wenn der erzeugte Wert nur einmal innerhalb der Schleife beschrieben wird, am Schleifeneingang nicht lebt und wenn die Variante 'VVerwendung' durchführbar ist. Die Variante 'VBeides' kann zwei Register und zwei Kopieroperation gegenüber der Variante 'VNormal' einsparen.

Die Implementierungsvarianten 'VVerwendung', 'VErzeugung' und 'VBeides' verlängern die Lebensdauer von Registern, weshalb sie auch zusätzliche untere Register bei ihrer Anwendung benötigen können.

Insgesamt kann die Anzahl der Kopierbefehle pro Optimierung gegenüber der ursprünglichen Variante 'Einfach' um bis zu drei Befehle reduziert werden. Dies ist in der Tabelle 4.6 dargestellt, wo die Variante 'Einfach' gegenüber der Variante 'VBeides' drei Kopieroperationen mehr benötigt. Bei den Beispielen aus Tabelle 4.7 können hingegen maximal nur zwei Kopieroperationen eingespart werden. Die Ursache hierfür liegt in der Anordnung der Speicherzugriffe. Daher sollte das Verfahren zuerst immer überprüfen, ob eine Verwendung vor der Erzeugung durchgeführt werden kann. Die zugrunde liegende Idee ist eine späte Erzeugung des Wertes und eine frühe Verwendung.

Dieser Abschnitt stellt noch zwei Spezialfälle vor. Der Fall 'VVerwendung' enthält noch den zusätzlichen Fall, dass keine Kopieroperationen benötigt werden, wie es in Tabelle 4.8 dargestellt ist. Dies passiert bei einer Iterationsdistanz von 0 oder 1 und wenn das Register, in dem der Wert erzeugt wird, auch das Register der Verwendung darstellt. Eine Umbenennung der Register kann hierzu auch sinnvoll sein. Weiterhin kann es auch vorkommen, dass kein Prolog benötigt wird, wenn die Iterationsdistanz gleich 0 ist. Die Tabelle 4.9 zeigt diesen Fall. Die Optimierungstechnik Redundant Load Elimination könnte diese Fälle ebenfalls generieren. Der Fall 'VNormal' wurde bereits in Kapitel 4.1 vorgestellt.

| C ohne RP | C mit RP | |
|---|---|---|
| | VNormal | VVerwendung |
| ...<br>R0=$a$[$i$-1]; //1<br>...<br>R0=$a$[$i$]; //2<br>... | ...<br>R0=R1; //1<br>...<br>R0=$a$[$i$]; //2<br>R1=R0;<br>... | ...<br>//1<br>...<br>R0=$a$[$i$]; //2<br>... |

Tabelle 4.8: Spezialfall 'VVerwendung': keine zusätzlichen Kopieroperationen

| C ohne RP | C mit RP | | |
|---|---|---|---|
| | VNormal | VVerwendung | VErzeugung |
| ...<br>$a$[$i$]=$d$;//1<br><br>...<br>$n$=$a$[$i$];//2<br>... | ...<br>$a$[$i$]=$d$;//1<br>R1=$d$;<br>...<br>$n$=R1;//2<br>... | ...<br>$a$[$i$]=$d$;//1<br>$n$=$d$;<br>...<br>//2<br>... | ...<br>$a$[$i$]=$d$;//1<br><br>...<br>$n$=$d$;//2<br>... |

Tabelle 4.9: Spezialfall: kein Prolog

### Mehrere Optimierungen

Die Tabellen 4.10 und 4.11 zeigen Paare von Optimierungen. Bei den Variablen $d$ und $n$ werden Werte erzeugt, die bei den optimierten Programmen bei $n$ bzw. $m$ verwendet werden. In den Tabellen sind zwei Optimierungsvarianten angegeben. Der obere Eintrag bezieht sich auf die Optimierung des Wertes bei $n$ und der untere Eintrag bei $m$.

| C ohne RP | C mit RP (Registerpipelining) | | | |
|---|---|---|---|---|
| | VNormal<br>VNormal | VVerwendung<br>VVerwendung | VErzeugung<br>VErzeugung | VBeides<br>VVerwendung |
| … | … | … | … | … |
| *m*=*a*[*i*-4];//1 | *m*=V1;//1 | //1 | *m*=V1;//1<br>V1=*n*; | //1 |
| … | … | … | … | … |
| *n*=*a*[*i*-2];//2 | *n*=R1;//2<br>V1=V2;<br>V2=*n*; | //2<br>*m*=V1;<br>V1=*n*; | *n*=R1;//2<br>R1=*d*; | //2<br>*m*=V1;<br>V1=*n*; |
| … | … | … | … | …<br>*n*=*d*; |
| *d*=*a*[*i*];//3 | *d*=*a*[*i*];//3<br>R1=R2;<br>R2=*d*; | *d*=*a*[*i*];//3<br>*n*=R1;<br>R1=*d*; | *d*=*a*[*i*];//3 | *d*=*a*[*i*];//3 |

Tabelle 4.10: Zwei Optimierungen mit einer Verwendung vor der Erzeugung

| C ohne RP | C mit RP (Registerpipelining) | | | |
|---|---|---|---|---|
| | VNormal<br>VNormal | VVerwendung<br>VVerwendung | VErzeugung<br>VErzeugung | VBeides<br>VVerwendung |
| … | … | … | … | …<br>*n*=*d*; |
| *d*=*a*[*i*];//1 | *d*=*a*[*i*];//1<br>R1=R2;<br>R2=*d*; | *d*=*a*[*i*];//1<br>*n*=R1;<br>R1=*d*; | *d*=*a*[*i*];//1 | *d*=*a*[*i*];//1 |
| … | … | … | … | … |
| *n*=*a*[*i*-1];//2 | *n*=R1;//2<br>V1=V2;<br>V2=*n*; | //2<br>*m*=V1;<br>V1=*n*; | *n*=R1;//2<br>R1=*d*; | //2<br>*m*=V1;<br>V1=*n*; |
| … | … | … | … | … |
| *m*=*a*[*i*-2];//3 | *m*=V1;//3 | //3 | *m*=V1;//3<br>V1=*n*; | //3 |

Tabelle 4.11: Zwei Optimierungen mit einer Erzeugung vor der Verwendung

Die einzelnen Optimierungen beeinflussen sich bei ihrer Durchführung auch untereinander. Interessant ist, dass die Variante 'VBeides' nicht zweimal hintereinander angewendet werden kann. Auch eine Verbindung mit einer der Varianten 'VErzeugung' oder 'VVerwendung' kann teilweise nur in einer bestimmten Reihenfolge geschehen. Diese Einschränkungen bei der Kombination von Optimierungen entstehen dadurch, dass die möglichen Lebensbereiche der Variablen auf die Länge einer Schleifeniteration beschränkt sind. Die Eigenschaft der gegenseitigen Beeinflussung erschwert letztlich die Auswahl der durchzuführenden Programmoptimierungen.

### Pipelineverschmelzung

In der Tabelle 4.11 können noch weitere Verbesserungen durchgeführt werden, da manche Speicherinhalte in mehreren Registern stehen. Beispielsweise steht der Wert von $a[i\text{-}1]$ bei der Variante zweimal 'VVerwendung' sowohl in $n$ als auch in V1. Hier existiert zusätzliches Optimierungspotenzial, wenn mehrere Optimierungen durchgeführt werden. Um dieses Potenzial zu verwenden, werden die einzelnen Pipelines miteinander verschmolzen. Die Tabelle 4.12 zeigt die Anwendung einer Pipelineverschmelzung bei den Beispielen aus der Tabelle 4.11. Damit bei der Variante zweimal 'VNormal' am Ende einer Iteration nicht in R1 und in V2 der Inhalt von $a[i\text{-}1]$ steht, wird der Befehl 'V2=$n$' entfernt und die Instruktion 'V1=V2' an den Anfang der anderen Pipeline gestellt. Dann müssen die beiden Pipelines noch miteinander verbunden werden, indem V2 in R1 umbenannt wird.

| C ohne RP | C mit RP (Registerpipelining) | | | |
|---|---|---|---|---|
| | VNormal<br>VNormal | VVerwendung<br>VVerwendung | VErzeugung<br>VErzeugung | VBeides<br>VVerwendung |
| ... | ... | ... | ... | ...<br>$m$=$n$;<br>$n$=$d$; |
| $d$=$a$[$i$];//1 | $d$=$a$[$i$];//1<br>V1=R1;<br>R1=R2;<br>R2=$d$; | $d$=$a$[$i$];//1<br>$m$=$n$;<br>$n$=R1;<br>R1=$d$; | $d$=$a$[$i$];//1 | $d$=$a$[$i$];//1 |
| ... | ... | ... | ...<br>$m$=$n$; | ... |
| $n$=$a$[$i$-1];//2 | $n$=R1;//2 | //2 | $n$=R1;//2<br>R1=$d$; | //2 |
| ... | ... | ... | ... | ... |
| $m$=$a$[$i$-2];//3 | $m$=V1;//3 | //3 | //3 | //3 |

Tabelle 4.12: Optimierungen mit einer Pipelineverschmelzung

Eine Voraussetzung für eine Pipelineverschmelzung ist, dass eine Erzeugung vor einer Verwendung durchgeführt wird und dass der erzeugte Wert innerhalb einer anderen Pipeline als eine Verwendung auftritt. Vor der Anwendung des Verfahrens muss genau überprüft werden, ob eine Pipeline an einer anderen Stelle positioniert werden kann. Hierzu überprüft das Verfahren, ob die verwendeten Variablen in der Registerpipeline an der anderen Stelle belegt werden dürfen. Eine Liveness-Analyse kann hierzu Informationen liefern. Insgesamt kann eine Pipelineverschmelzung eine Kopieroperation und ein Register einsparen.

### Prologoptimierung

Eine weitere mögliche Verbesserung, die in Zusammenhang mit mehreren Optimierungen verwendet werden kann, ist eine Optimierung des Prologs auf Assemblerebene. Beim Beispiel 4.2.6 kann der Prolog auf Assemblerebene optimiert werden. Beispiel 4.2.7 stellt diese Optimierung dar. Die Berechnung der Indexfunktion kann im Prolog verkürzt werden, wenn die Indexfunktionen eine bestimmte Ähnlichkeit besitzen. In diesem Beispiel trifft dies auf die Indexfunktionen von $a[i\text{-}1]$ und $a[i+49]$ zu. Diese Optimierung kann insbesondere den benötigten Programmspeicherplatz reduzieren. Der Energieverbrauch und die Taktzyklenanzahl können nur unerheblich optimiert werden, da der Prolog außerhalb der Schleife liegt.

**Beispiel 4.2.6** Prologoptimierungsbeispiel

| // * ohne Registerpipelining * | // * mit Registerpipelining * |
|---|---|
| int $i$,$a$[100]; | int $i$,$a$[100]; |
| | R0=$a$[$i$-1]; |
| | R1=$a$[$i$+49]; |
| for ($i$=0; $i$<10; $i$++) | for ($i$=0; $i$<10; $i$++) |
| { | { |
| $a$[$i$]=$a$[$i$-1]+1; | R0=R0+1; |
| | $a$[$i$]=R0; |
| $a$[$i$+50]=$a$[$i$+49]+1; | R1=R1+1; |
| | $a$[$i$+50]=R1; |
| } | } |

**Beispiel 4.2.7** Prologoptimierung

| // * ohne Prologoptimierung * | // * mit Prologoptimierung * |
|---|---|
| mov R2,#0 | mov R2,#0 |
| sub R0,R2,#1 | sub R0,R2,#1 |
| lsl R0,R0,#2 | lsl R1,R0,#2 |
| add R1,SP,#0 | add R3,SP,#0 |
| ldr R0,[R1,R0] // R0=$a$[$i$-1]; | ldr R0,[R3,R1] // R0=$a$[$i$-1]; |
| mov R1,#49 | |
| add R1,R2,R1 | |
| lsl R1,R1,#2 | |
| add R3,SP,#0 | add R3,#200 |
| ldr R1,[R3,R1] // R1=$a$[$i$+49]; | ldr R1,[R3,R1] // R1=$a$[$i$+49]; |

### Zusammenfassung

Die in diesem Kapitel vorgestellte und auch für die Versuche in Kapitel 8 implementierte Optimierungstechnik Registerpipelining, besteht aus den folgenden Komponenten:

1. Positionierung der Speicherzugriffe
   Eine Verwendung sollte vor einer Erzeugung stattfinden innerhalb der Programmreihenfolge in einer Schleife, wie es in der Tabelle 4.6 dargestellt ist. Eine Umordnung der Instruktionen kann aber zusätzliches Spilling hervorrufen, weshalb auf diese Möglichkeit bei der durchgeführten Implementierung verzichtet worden ist.

2. Generierung der Instruktionen für eine Umsetzung
   Das Verfahren Registerpipelining optimiert das Programm mit Hilfe der Implementierungsvarianten 'VNormal', 'VVerwendung', 'VErzeugung' und 'VBeides'.

3. Pipelineverschmelzung
   Die Verschmelzung einzelner Pipelines kann jeweils eine Kopieroperation einsparen.

4. Prologoptimierung und Generierung
   Ein Prolog, der möglichst wenig Befehle enthält, wird berechnet und eingefügt.

5. Aufruf anderer Optimierungsverfahren
   Abschließend werden noch die Optimierungsverfahren Copy Propagation und Useless Code Elimination aufgerufen, um den Programmcode weiter zu verbessern.

Das Optimierungsverfahren Registerpipelining, dessen Aufbau in der vorherigen Aufzählung beschrieben ist, benötigt eine $\delta$-available-values Analyse (siehe Kapitel 7) und eine Liveness-Analyse der Registerinhalte. Insgesamt verwendet das Verfahren bis zu drei Register und Kopieroperationen pro Optimierung weniger als das Verfahren aus dem Kapitel 4.2.1. Hierdurch kann der Energieverbrauch, die Anzahl an Taktzyklen, die Leistungsaufnahme und der Speicherplatzbedarf reduziert werden. Die in diesem Unterkapitel beschriebene Vorgehensweise besitzt folgende Einschränkungen:

- Optimierungen innerhalb komplexer Kontrollstrukturen sind nicht möglich. Beispielsweise kann in Beispiel 4.2.4 das $a[i]$ nicht durch einen früheren Zugriff auf $a[i+2]$ ersetzt werden. Das Verfahren kann in dem Beispiel nur $a[i\text{-}2]$ optimieren.

- Die Registerpipeline wird nur bei den für eine Optimierung relevanten Speicherzugriffen platziert. Eine freie Platzierung könnte die Qualität der Optimierung verbessern.

Die Verwendung mehrerer Registerpipelines erzielt bei dem Beispiel in Tabelle 4.13 Vorteile gegenüber einer auf Instruktionsebene zusammenhängenden Registerpipeline. Eine zusammenhängende Registerpipeline fängt mit einer der vier möglichen Varianten an und kann danach nur noch die Fälle 'VNormal' und 'VVerwendung' abdecken. Für dieses Beispiel kann aber dreimal die Variante 'VErzeugung' eingesetzt werden.

| C ohne RP | C mit Registerpipelining | |
|---|---|---|
| | mehrere Pipelines | eine Pipeline |
| $m$=3;<br>$m$=$a$[$i$-3];<br>$n$=2;<br>$n$=$a$[$i$-2];<br>$o$=1;<br>$o$=$a$[$i$-1];<br>$p$=$a$[$i$]; | $m$=3;<br>$m$=$n$;<br>$n$=2;<br>$n$=$o$;<br>$o$=1;<br>$o$=$p$;<br>$p$=$a$[$i$]; | $m$=3;<br>$m$=R2;<br>$n$=2;<br>$n$=R1;<br>$o$=1;<br>$o$=$p$;<br>R2=R1;<br>R1=$o$;<br>$p$=$a$[$i$]; |

Tabelle 4.13: Nachteil einer einzelnen Registerpipeline

### 4.2.5 Berücksichtigung weiterer Prozessoreigenschaften

Spezielle Eigenschaften des ARM7TDMI-Prozessors können für die Optimierungstechnik Registerpipelining eingesetzt werden.

#### Verwendung von speziellen Befehlen

Der Thumb-Instruktionssatz beinhaltet eine Multiple-Load-Instruktion ('ldmia', Load Multiple Increment After) mit dem Format, das in der Abbildung 4.1 dargestellt ist. Bis zu 8 Speicherinhalte können mit einer Instruktion in den unteren Registern abgelegt werden.

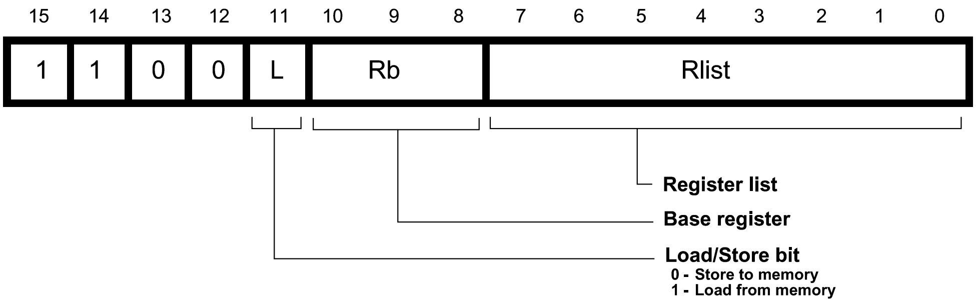


Abbildung 4.1: Format einer multiple Load/Store-Instruktion [ARML95]

Die 'ldmia'-Instruktion kann den Prolog bezüglich der Werte Programmgröße, Energieverbrauch, Leistungsaufnahme und Taktzyklen optimieren. Das Beispiel 4.2.8 zeigt eine Optimierung mit einem Multiple-Load-Befehl.

**Beispiel 4.2.8** Prologoptimierung

```
// * normal optimierter Prolog *
sub  R0,R2,#1
lsl  R0,R0,#2
add  R1,SP,R0
ldr  R3,[R0,#0]
ldr  R4,[R0,#4]
ldr  R5,[R0,#8]
ldr  R6,[R0,#12]
ldr  R7,[R0,#16]
```

```
// * mit ldmia-Instruktion *
sub   R0,R2,#1
lsl   R1,R0,#2
add   R3,SP,R1
ldmia R0!,{R3-R7}
```

#### Verwendung von speziellen Registern

Die Register R13 (Stackpointer) und R14 (Linkregister) könnten für das Verfahren Registerpipelining ebenfalls eingesetzt werden. Beispielsweise könnte eine Optimierungstechnik sie am Anfang einer Schleife sichern und dann verwenden. Es muss aber genau überprüft werden, ob dies auch in dem spezellen Fall möglich ist.

### 4.2.6 Pipeline im On-Chip-Speicher

Eine Speicherzugriffsoptimierung muss nicht unbedingt nur Speicherzugriffe durch Registerzugriffe ersetzen. Teure Speicherzugriffe auf den externen Speicher können auch durch kostengünstige Zugriffe auf den On-Chip-Speicher ersetzt werden.

Zahlreiche Speicherzugriffsoperationen können die benötigte Pipeline für eine Optimierungstechnik bilden. Dieser Ansatz ist manchmal lohnenswert und könnte aber auch von einer guten Registerallokation nach der Durchführung der Optimierungstechnik Registerpipelining erledigt werden. Eine andere Möglichkeit zur Umsetzung einer Pipeline wird in Beispiel 4.2.9 dargestellt. Das Array $m$ als Ringpuffer mit der Größe 32 liegt bei dem Beispiel im On-Chip-Speicher und das Array $a$ liegt im externen Speicher. Die Pipeline wird mit einer modulo-Operation und mit Speicherzugriffen im On-Chip-Speicher realisiert.

**Beispiel 4.2.9** Verwendung von On-Chip-Speicher

```
// * 1.Variante *
// ursprüngliche Schleife

for( i=32; i<999; i++)
{
  R0=a[i-32]+1;
  a[i]=R0;
}
```

```
// * 2.Variante *
// optimierte Schleife
for( i=0; i<32; i++)
{
  m[i]=a[i];
}

R2=0;
for( i=32; i<999; i++)
{
  R0=m[R2]+1;
  a[i]=R0;
  m[R2]=R0;
  R2=R2+1;
  R2=R2&31;
}
```

```
// * 3.Variante *
// mit Epilog
for( i=32; i<64; i++)
{
  R0=a[i-32]+1;
  a[i]=R0;
  R2=i&31;
  m[R2]=R0;
}

for( i=64; i<967;i++)
{
  R2=i&31;
  R0=m[R2]+1;
  a[i]=R0;
  m[R2]=R0;
}

for(i=967;i<999;i++)
{
  R2=i&31;
  R0=m[R2]+1;
  a[i]=R0;
}
```

Ein Vorteil dieser Methode ist, dass die Anzahl der Instruktionen, die für eine Optimierung benötigt werden, fast unabhängig ist von der Iterationsdistanz. Nur die Anzahl der benötigten Instruktionen für die 'modulo'-Operation kann sich verändern, denn diese kann nicht immer durch eine einfache 'und'-Operation übersetzt werden. Oft wird auch ein zusätzliches Register als Indexfunktion einer Array-Referenz benötigt.

Ein externer Speicherzugriff wird durch die Verwendung von mindestens einem On-Chip-Speicherzugriff und durch einige weitere Operationen eingespart. Die Tabelle 4.14 zeigt, wie viele weitere 'mov'-Operationen verwendet werden dürfen, um ein effizientes Verhalten sicherzustellen. Wenn die Instruktionen im externen Speicher liegen, müssen noch Teile der Berechnung der Indexfunktion entfallen, um ein effizientes Verhalten zu erreichen.

| Speicherort der Instruktion | Anzahl der 'mov'-Operationen | |
|---|---|---|
| | bzgl. Energie | bzgl. Taktzyklen |
| On-Chip-Speicher | 11.9 | 3 |
| externer Speicher | 2.0 | 1.5 |

Tabelle 4.14: Anzahl der noch verwendbaren 'mov'-Operationen

Die 'ldmia'- und 'stmia'-Operationen können insbesondere bei der zweiten Implementierungsvariante im Beispiel 4.2.9 effizient eingesetzt werden. Ein zusätzlich eingefügter Epilog kann hier im Gegensatz zu der Konstruktion einer Pipeline mit Registern einige Speicherzugriffsoperationen einsparen.

Durch diese Optimierung kann der Energieverbrauch, die Leistungsaufnahme und die Taktzyklenanzahl reduziert werden. Der benötigte Programmspeicherplatz vergrößert sich dagegen oft.

## 4.3 Weitere Optimierungstechniken

Folgende Optimierungstechniken führen ebenfalls Speicherzugriffsoptimierungen durch.

- Daten-Regeneration ([SS99])
  Die Optimierungstechnik Daten-Regeneration versucht Daten kostengünstig erneut zu berechnen, anstatt sie teuer aus einem Speicher zu laden.

- Redundant Store Elimination ([Fra99])
  Redundante Schreibzugriffe können entfernt werden.

- Felderweiterung ([Sch99])
  Diese Optimierungstechnik versucht, aufeinanderfolgende Speicherzugriffe auf unterschiedliche Speicherbänke abzubilden. Hierzu wird die Größe eines Feldes verändert.

- Skalarer Ersatz ([Sch99])
  Die Optimierungstechnik ersetzt einen Speicherzugriff durch ein Skalar.

Ein Compiler sollte auch die Speicherzugriffe auf die Instruktionen optimieren. Instruktionen werden aus dem Speicher geladen, bevor sie der Prozessor ausführen kann. Bei dem ATMEL-Evaluationboard kann es aus Gründen der Effizienz sinnvoll sein, die Programme im On-Chip-Speicher abzulegen. Da Programme aber oft größer sind als der zur Verfügung stehende 4-KB Speicher müssen geeignete Ein- und Auslagerungsstrategien verwendet werden. Die hierfür benötigten Speicherkopieroperationen sollten, wenn es möglich ist, im On-Chip-Speicher liegen. Programmteile, die in den On-Chip-Speicher verlegt werden, sollten im allgemeinen Schleifen enthalten, die öfters durchlaufen werden, damit diese Vorgehensweise lohnenswert ist.

## 4.4 Zusammenfassung

Speicherzugriffe besitzen einen Anteil von teilweise 86% an dem gesamten Energieverbrauch einer Instruktion. Deswegen können Optimierungstechniken, die die Speicherzugriffe optimieren, große Verbesserungen erzielen.

Die beiden Speicherzugriffsoptimierungen Redundant Load Elimination und Registerpipelining können den Energieverbrauch optimieren. Sie belegen Register mit den Speicherinhalten von Arrays, um effizient auf einen Wert erneut zugreifen zu können. Die Redundant Load Elimination kann Optimierungen bis zu einer Iterationsdistanz von 1 durchführen. Registerpipelining verallgemeinert die Redundant Load Elimination bzgl. größerer Iterationsdistanzen.

Spezielle Architekturmerkmale des ATMEL AT91M40400 und des ARM7TDMI-Prozessors unterstützen die Optimierungstechniken. Im Thumb-Instruktionssatz kann der hohe Registersatz des ARM7TDMI-Prozessors verwendet werden. Die Realisierung einer Pipeline als Ringpuffer im On-Chip-Speicher ist bei einem hohen Registerdruck und bei großen Iterationsdistanzen sinnvoll. Eine Multiple-Load-Instruktion kann einen Schleifenprolog optimieren.

Die vorgestellte und verwendete Implementierung der Optimierungstechnik Registerpipelining aus Kapitel 4.2.4 dieser Diplomarbeit kann bis zu drei Kopieroperationen und Register gegenüber der Variante, die in Kapitel 4.2.1 dargestellt ist, einsparen.

# Kapitel 5

# Arbeitsumgebung

Dieses Kapitel beschreibt die Umgebung, in der die Optimierungstechnik Registerpipelining enwickelt wurde. Es wird ein Weg von einem C-Programm bis zur Analyse der Ausführung skizziert und die dabei durchgeführten Aufgaben im Rahmen der Diplomarbeit werden erläutert.

## 5.1 Übersicht

Abbildung 5.1 skizziert Teile des Datenflusses innerhalb der Arbeitsumgebung, in der die Optimierungstechnik Registerpipelining liegt.

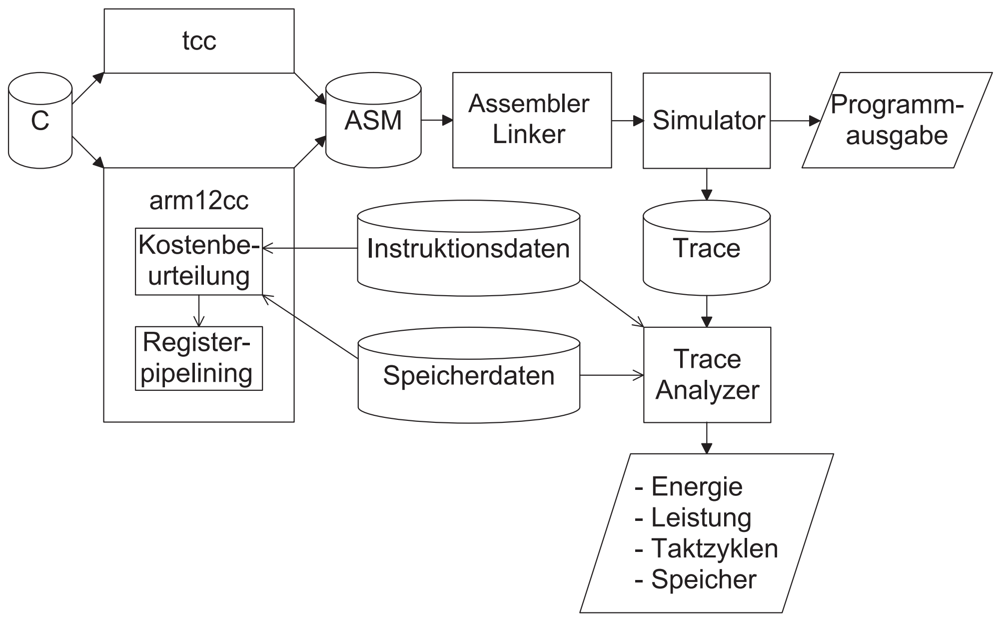


Abbildung 5.1: Datenfluss innerhalb der Arbeitsumgebung

Zuerst wird ein C-Programm an einen von zwei Compilern weitergeleitet. Der Compiler 'tcc' gehört zu den Standardcompilern für die ARM-Prozessoren. Er stammt aus dem ARM Software Development Toolkit v2.50 von ARM Limited. Die Implementierung des Compilers 'arm12cc' erfolgt an der Universität Dortmund am Lehrstuhl 12 des Fachbereichs Informatik. LANCE2 bildet das Frontend des Compilers und das Backend kann eine Energieoptimierung durchführen. Innerhalb dieses Compilers wurde die Optimierungstechnik Registerpipelining implementiert. Beide Compiler können ein C-Programm in einen Assemblercode für den ARM7TDMI-Prozessor übersetzen. Zwei Compiler werden verwendet, um die Qualität der erzeugten Programme miteinander vergleichen zu können und die Korrektheit zu validieren. Der von ihnen generierte Assemblercode wird an einem Assembler und Linker weitergeleitet. Diese erzeugen einen ausführbaren Code. Der nachfolgende Simulator erstellt daran anschließend eine Datei mit den Ausgaben des Programms und eine Datei mit einem Trace über den Programmverlauf. Der Trace enthält die Instruktionssequenz, die der Prozessor ausgeführt hat. Der Assembler, der Linker und der Simulator sind ebenfalls im Software Development Toolkit von ARM Limited enthalten.

Die Programmausgabe wird verwendet, um die Ergebnisse der Programme zu überprüfen. Der generierte Trace kann zu einem Trace-Analyzer weitergeleitet werden. Dieser analysiert den Trace und gibt abschließend folgende Eigenschaften über die Programmausführung bekannt:

- Energieverbrauch
- Leistungsaufnahme
- Anzahl der benötigten Taktzyklen
- Speicherplatzbedarf

Der Trace-Analyzer verwendet für die Berechnung der Daten die Inhalte der Dateien mit den Instruktionskosten und den Speicherkosten. Der Compiler 'arm12cc' verwendet diese Dateien ebenfalls, um die Übersetzung zu optimieren.

Im Rahmen dieser Diplomarbeit wurden viele der einzelnen Schnittstellen automatisiert. Der Anhang A stellt Programme vor, die hierfür implementiert wurden.

## 5.2 Trace-Analyzer

Der Trace-Analyzer, der im Rahmen dieser Diplomarbeit implementiert und entwickelt wurde, ermittelt die Eigenschaften eines simulierten Programms.

Die Analyse des Energiebedarfs kann z.B. durch eine der Techniken aus [VT94] oder [EM98] geschehen. An dieser Stelle wird ein Verfahren vorgestellt, das in beiden Literaturangaben vorhanden ist und den Energieverbrauch anhand eines von einem Simulator erstellten Traces ermitteln kann. Die Gleichung 5.1 berechnet die Energiekosten $E_P$ eines Programms $P$.

$$E_P = \sum_i (B_i \cdot N_i) + \sum_{i,j} (O_{ij} \cdot N_{ij}) + \sum_k E_k \tag{5.1}$$

Die Anwendung der Gleichung 5.1 liefert auch in der Praxis sehr genaue Werte nach [VT94]. Sie besteht insgesamt aus 3 Summen, die im einzelnen diese Bedeutung haben:

1. Summe
   Die Grundkosten, die während der Ausführung für die einzelnen Befehle entstehen, werden an dieser Stelle aufsummiert. $B_i$ entspricht den Energiekosten für eine Instruktion $i$ und $N_i$ gibt an, wie oft die Instruktion $i$ auftritt. Der Energieverbrauch in Abhängigkeit von unterschiedlichen Operanden kann an dieser Stelle ebenfalls berücksichtigt werden.

2. Summe
   Die Schaltungsaktivitäten, die durch Zustandswechsel (Inter-Instruction Effects) entstehen, werden hiermit aufaddiert. In [GS99] wurde beschrieben, dass es ausreicht, Paare von Instruktionen zu analysieren. Das $O_{ij}$ sei der zusätzliche Energieverbrauch bei einem Instruktionspaar $i$ und $j$ und $N_{ij}$ entspricht der Anzahl, wie oft das Instruktionspaar auftritt.

3. Summe
   Der Energieverbrauch von Besonderheiten (Ereignissen) wird aufsummiert. Hierzu gehören z.B. die Kosten von pipeline stalls.

Da das Verfahren Registerpipelining die Eigenschaften der Speicherhierarchie für eine Optimierung verwendet, sollten auch die unterschiedlichen Speicher und ihre Eigenschaften berücksichtigt werden. Die Spannung beträgt bei dem zur Verfügung stehendem Prozessor 3,3 Volt und er besitzt eine Taktfrequenz von 33MHz. Die Gleichung 5.2 stellt den Energieverbrauch einer Instruktion bei dem ATMEL-Evaluationboard dar. Die Anzahl der benötigten Taktzyklen für eine Instruktion sei $n$.

$$E = \frac{3.3V}{33MHz} I \cdot n \tag{5.2}$$

Bei der implementierten Version des Trace-Analyzers wird jeder Befehl einzeln betrachtet, um ihn individuell zu beurteilen. Die Grundkosten $P_i$ für eine Instruktion $i$ werden aus dem Produkt von durchschnittlichem Energieverbrauch des Prozessors $E_D$ und der Taktzyklenzahl $n$ gebildet.

$$P_i = E_D \cdot n \tag{5.3}$$

Das $S_i$ bezeichnet die Kosten für die durchgeführten Speicherzugriffe. Hierzu gehören die Speicherzugriffe für die Daten und für die Instruktionen. Die Gleichung 5.4 berechnet die Energiekosten $G_i$ für eine Instruktion des Traces.

$$G_i = P_i + S_i \tag{5.4}$$

Die Schaltungsaktivitäten, die durch Zustandswechsel entstehen, könnten zusätzlich aufaddiert werden. Geeignete Werte standen aber zur Zeit der Implementierung des Trace-Analyzers nicht zur Verfügung.

Die Gleichung 5.5 berechnet die Energiekosten $E_P$ eines Programms $P$ aus der Summe der Instruktionskosten.

$$E_P = \sum_i G_i \tag{5.5}$$

Der implementierte Trace-Analyzer berechnet den Energieverbrauch, wie in diesem Kapitel beschrieben. Weiterhin gibt der Trace-Analyzer die folgenden Werte bekannt:

- Taktzyklen
  Die Anzahl der Taktzyklen eines Programms kann aus dem Trace ermittelt werden.
- Leistung
  Der gesamte Energieverbrauch wird zur Berechnung dieses Wertes durch die Zeit geteilt.
- Programmspeicherplatz
  Der Linker ermittelt den benötigten Speicherplatz eines Programms, und der Trace-Analyzer gibt diesen Wert aus.
- Datenspeicherplatz
  Hierzu wird untersucht, wie viel Bytes ein Programm auf den Stack und den Heap ablegt.
- Anzahl der ausgeführten Instruktionen
- Anzahl der Speicherzugriffe
  Die genaue Verteilung der Speicherzugriffe auf die einzelnen Speicherbereiche wird untersucht und ausgegeben.

Der implementierte Trace-Analyzer wird in Anhang A vorgestellt und in Kapitel 8.2 wird die Genauigkeit der berechneten Energiewerte überprüft. Die Abbildung 5.2 zeigt exemplarisch die Ausgaben des Trace-Analyzers für ein untersuchtes Progamm.

## 5.3 Programm- und Datenverteilung

Um die Auswirkungen verschiedener Speicher beurteilen zu können, müssen die Programme und die Daten geeignet verteilt werden.

Zuerst wurde eine geeignete ANSI-C Library erzeugt, die eine Aufteilung eines Programms auf die unterschiedlichen Speicherbereiche erlaubt. Hierfür wurde eine bestehende Library abgeändert, indem einige Programmteile für die Initialisierung der Speicherbereiche aus der Library entfernt wurden. Stattdessen kann ein Programm einen Initialisierungscode ausführen, der beim Linken zu dem gesamten Programm hinzugefügt wird. Er kopiert das Programm und die Daten in die verschiedenen Speicherbereiche.

Die Verteilung der einzelnen Programmteile erfolgt mit Hilfe einer Datei, die exemplarisch in Beispiel 5.3.1 dargestellt ist. Die Vorgehensweise wird hier

Memory
Program : 88
Data : 384

executed Instructions : 1740
access to Datamemory : 1288 Byte

| Memoryunit | Instruction | Data |
|---|---|---|
| OFFCHIP read 4 Byte | 0 | 216 |
| ONCHIP read 2 Byte | 1740 | 0 |
| OFFCHIP write 4 Byte | 0 | 106 |

CPU-Cycles : 3840
Energy : $33.864/10^6$ Ws = $18.384/10^6$ Ws (Instruction) $+15.480/10^6$ Ws (Memory)
Power : 291.0 mW = 158.0 mW (Instruction) +133.0 mW (Memory)

Abbildung 5.2: Ausgaben des Trace-Analyzers für ein Programm

nach [ARML98] beschrieben. Die Datei gibt die Bereiche von drei Speichern an. Der Flash-ROM-Speicher beginnt an der Adresse 0x00400000, der On-Chip-Speicher (32-Bit-RAM) an der Adresse 0x00300000 und der externe Speicher (16-Bit-RAM) liegt bei 0x00500000.

**Beispiel 5.3.1** Speicheraufteilungsdatei

```
FLASH 0x00400000 0x00400000 {
    FLASH 0x00400000
    {
        * (+RO)
    }
    32bitRAM 0x00300000
    {
        test.o (+RO)
        test.o (Daten1,+RW)
    }
    16bitRAM 0x00500000
    {
        * (+RW,+ZI)
    }
}
```

Innerhalb der Speicheraufteilungsdatei wird nach den folgenden drei Datentypen unterschieden, die im Quellcode den einzelnen Programmteilen zugeordnet sind:

- read-only data (RO)
- read-write data (RW)
- zero-initialized data (ZI)

Zu Beginn der Ausführung kopiert ein Initialisierungsprogramm die einzelnen Programmteile in die Speicherbereiche, die über die Linkeroption '-scatter' angegeben wurden. Der read-only Speicher des Programms 'test.c' wird bei dem Beispiel im On-Chip-Speicher abgelegt. Die restlichen read-only Speicherbereiche, wie z.B. die aus der Library, kopiert das Programm in den ROM-Speicher. Einzelne Bereiche eines Programms können auch mit einem Namen gekennzeichnet werden. Der Datenbereich, der die Kennzeichnung 'Daten1' trägt, wird im On-Chip-Speicher platziert und die restlichen Daten befinden sich im externen Speicher.
Die Abbildung 5.3 zeigt die einzelnen Dateien, die der Linker benötigt. Der Compiler 'arm12cc' erzeugt ein Programm 'test.asm'. Dieses Programm kann dann den Assembler, Linker und Simulator durchlaufen.

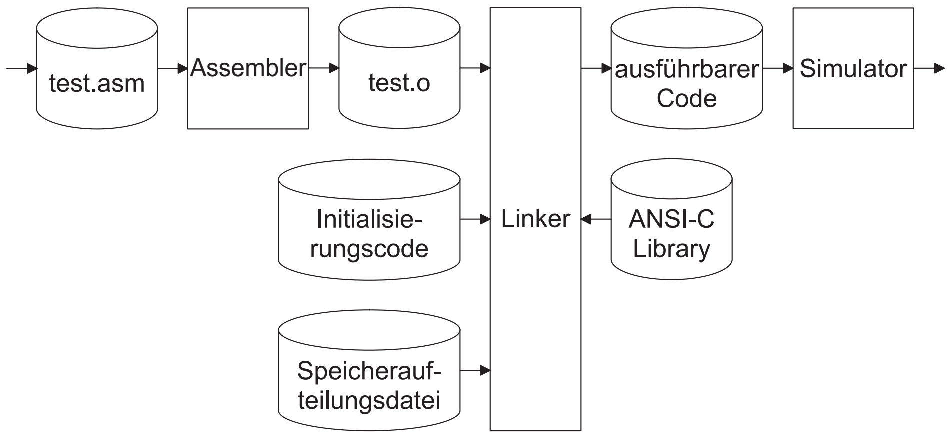


Abbildung 5.3: Dateien innerhalb der Codegenerieung

Insgesamt können hierdurch die einzelnen Programmteile frei platziert werden und eine Optimierung kann die Vorteile, wie z.B. Größe und Geschwindigkeit, der einzelnen Speicher verwenden. Die Initialisierungs- und Speicheraufteilungsdatei muss an die zugrunde liegende Hardware angepasst werden.

# Kapitel 6

# Integration von Registerpipelining

Dieses Kapitel beschreibt die Probleme, Lösungen, Entscheidungen und Entwicklungsschritte im Rahmen der Integration der Optimierungstechnik Registerpipelining in den Compiler 'arm12cc'. Die Ergebnisse dieser Diplomarbeit können hierdurch besser beurteilt, eingeordnet und reproduziert werden. Insbesondere können andere Implementierungen zu anderen Ergebnissen bei der Bewertung in Kapitel 8 führen.

## 6.1 Positionierung der Optimierungstechnik

Die Optimierungstechnik Registerpipelining kann an verschiedenen Positionen unterschiedlich implementiert werden. Prinzipiell gibt es die Möglichkeit, das Verfahren im Frontend, im Middleend oder im Backend innerhalb des Compilers 'arm12cc' zu platzieren, wie es in Abbildung 6.1 dargestellt ist.

Das Frontend und das Middleend besitzen den Vorteil, dass sie von der Hardware unabhängig sind. Beispielsweise hat sich die Diplomarbeit von Björn Franke [Fra99] mit diesem Thema beschäftigt. Das Middleend verwendet im Gegensatz zu dem Frontend des Compilers 'arm12cc' nur eine Untermenge der Programmiersprache C.

Die andere Möglichkeit ist die Platzierung der Optimierungstechnik Registerpipelining im Backend. Das hardwareabhängige Backend bietet den Vorteil einer höheren Qualität der Optimierung. Zum einen liegen präzise Informationen über den Registerdruck vor, wodurch teures Spilling berücksichtigt werden kann. Das Verfahren Registerpipelining kann die unteren und hohen Register des ARM7TDMI-Prozessors effizient verwenden. Zudem kann die Optimierungstechnik die einzelnen Programmoptimierungen einfach bewerten, da genaue Information über die Instruktionskosten vorliegen. Aus diesen Gründen wurde die Optimierungstechnik Registerpipelining im Rahmen dieser Diplomarbeit innerhalb des Backends implementiert.

Die Registerallokation des Compilers 'arm12cc' trägt dazu bei, den Registerdruck zu ermitteln. Während der Durchführung einzelner Optimierungen mit dem Verfahren Registerpipelining kann sich der Registerdruck ändern. Deshalb

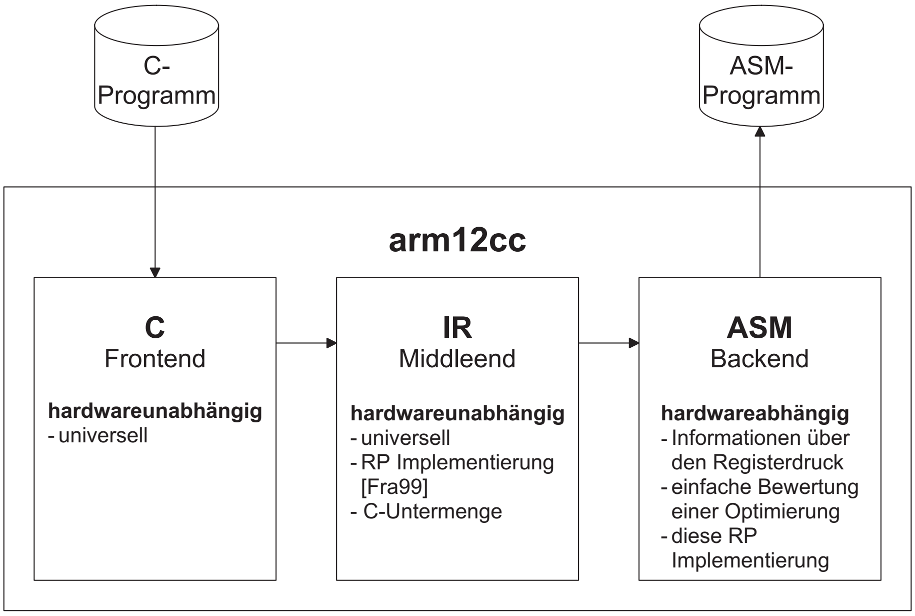


Abbildung 6.1: Positionierungsmöglichkeiten von Registerpipelining

wird die Registerallokation mehrmals aufgerufen, wodurch eine Rückkopplung zwischen den beiden Verfahren entsteht. Die Optimierungstechnik Registerpipelining wurde deshalb direkt bei der Registerallokation in dem Compiler 'arm12cc' platziert.

## 6.2 Eigenschaften der Optimierungstechnik

Eine der wichtigsten Aufgaben ist es, die Eigenschaften der Optimierungstechnik Registerpipelining festzulegen. Drei Möglichkeiten werden in folgender Aufzählung aufgeführt:

1. Die Optimierungstechnik darf in einem bestimmten Ausmaß fehlerhaften Programmcode erzeugen.

2. Der Programmierer gestattet explizit die Anwendung einer Optimierungstechnik durch eine Compilerdirektive, wie z.B. in High Performance Fortran die '!HPF$ INDEPENDENT' Direktive (nach [Sch99]).

3. Die Optimierungstechnik erstellt einen korrekten Programmcode.

Am einfachsten wäre die erste Möglichkeit. Bei einer Registerpipeline sollten die Register die Speicherinhalte zwischenspeichern und immer mit ihnen übereinstimmen. Beispiel 6.2.1 skizziert einen Fall, bei dem die Entscheidung sehr schwierig ist, ob zwei Speicherzugriffe im nachfolgenden Programmcode den gleichen Speicherplatz referenzieren. Um Fehlentscheidungen auszuschließen, müssen beispielsweise Alias-Analysen die jeweiligen Instruktionen genau untersuchen. Die Erzeugung unkorrekter Programme kann beispielsweise nicht ganz ausgeschlossen werden, falls unterschiedliche Arrays auch überlappend adressiert werden können.

Die zweite Möglichkeit bietet sich für diese Diplomarbeit nicht an, da eine Abstimmung des Compilers auf diese Option ziemlich umfangreich ist. Die Direktiven könnten dann aber sowohl für einzelne Speicherzugriffe, als auch für einzelne Instruktionen gelten.

**Beispiel 6.2.1** Problem: Bestimmung der adressierten Speicheradressen

```
...
int a[100],*b;
b=a+4+c;
...
  a[i]= ...        // überschneiden sich die adressierten
  b[i]= ...        // Speicherbereiche ?
...
```

Für diese Diplomarbeit wurde die dritte Möglichkeit ausgewählt. Abhängigkeiten, wie sie in Beispiel 6.2.1 auftreten, werden entweder eindeutig erkannt oder es wird eine konservative Lösung gewählt.

## 6.3 Wahl benötigter Daten

Die Indexfunktionen der Speicherzugriffe müssen bekannt sein, um Abhängigkeiten zu erkennen. In der nachfolgenden Aufzählung sind einige Möglichkeiten für die Ermittlung der Indexfunktionen mit den dazugehörigen Problemen und Nachteilen aufgeführt:

- C-Ebene
  Die Indexfunktionen werden aus dem C-Programm entnommen. Folgende Probleme ergeben sich:
  - Die Indexfunktionen müssen den Speicherzugriffen im Backend zugeordnet werden.
  - Zugriffe auf Arrays, die über Pointer ausgedrückt werden, können nur schwer erkannt werden.
- IR-Ebene mit Indexfunktionen
  Das Frontend kann die Indexfunktionen eines C-Programms an die IR (Intermediate Representation, Zwischencode) reichen und das Backend könnte dann die Indexfunktionen analysieren. Folgende Probleme und Nachteile treten auf:
  - Beliebig komplexe Indexfunktionen müssen im Backend vor den anderen Optimierungen und Analysen übersetzt werden, um eine hohe Effizienz sicherzustellen.
  - Eine eindeutige Zuordnung der Indexfunktionen zu den Speicherzugriffen auf Assemblerebene sollte existieren.

  - Zugriffe auf Arrays, die über Pointer ausgedrückt werden, können nur schwer erkannt werden.

- IR-Ebene ohne Indexfunktion
  Die Indexfunktionen müssen im Zwischencode wiedererkannt werden. Folgende Probleme ergeben sich:
  - Die Indexfunktionen müssen aus dem Zwischencode rekonstruiert werden.
  - Die Indexfunktionen müssen den Speicherzugriffen auf Assemblerebene zugeordnet sein.

- Assemblercode-Ebene
  Die Indexfunktionen müssen aus dem Assemblercode rekonstruiert werden. Folgende Probleme treten auf:
  - Eindeutige Wiedererkennung von Indexfunktionen innerhalb des Assemblercodes.

Die Indexfunktionen wurden für die Implementierung der Optimierungstechnik Registerpipelining innerhalb dieser Diplomarbeit aus dem Assemblercode rekonstruiert.

## 6.4 Aufbau des Verfahrens

Die in den Compiler 'arm12cc' integrierte Optimierungstechnik Registerpipelining lässt sich in folgende Komponenten zerlegen:

1. Erkennung von geeigneten Schleifen
   Ein Depth-First-Search-Algorithmus (DFS-Algorithmus) erkennt die vorhandenen Schleifen, indem er die Kanten des Kontrollflussgraphen in Tree-, Forward-, Cross- und Backkanten einteilt (Kanteneinteilung nach [Hof99]). Jede Backkante stellt einen Teil einer Schleife dar. Die Schleifen werden nach geschätzten Ausführungshäufigkeiten sortiert. Der Algorithmus arbeitet die Schleifen der Reihe nach ab, beginnend mit der Schleife, die am häufigsten durchlaufen wird.

   Das Verfahren analysiert die einzelnen Schleifen hinsichtlich der folgenden Eigenschaften:

   - Die Schleifen müssen die single-entry/single-exit Eigenschaft (genau einen Schleifeneintritt und einen Schleifenaustritt) besitzen. Diese Einschränkung vereinfacht die Implementierung der Optimierungstechnik Registerpipelining.
   - Die Schleife darf keine Funktionsaufrufe enthalten, da das gesamte Verfahren keine funktionsübergreifenden Analysen durchführt. Oft müssten auch zusätzliche 'push'- und 'pop'-Instruktionen eingefügt werden, wodurch das Verfahren an Effizienz verliert.

- Es dürfen keine 'push'- und 'pop'-Operationen in einer Schleife vorhanden sein, da diese Speicherzugriffe nicht analysiert werden. Dieser Fall tritt bei dem betrachteten Compiler aber nicht auf.

Falls eine der Eigenschaften nicht erfüllt ist, wird auf eine weitere Analyse der Schleife verzichtet.

2. Erkennung von Induktionsvariablen
Das Verfahren sucht als nächstes die Induktionsvariablen der Schleife. In dem Beispiel 6.4.1 wird die Induktionsvariable $i$ erkannt. Die Variablen $k$,$j$ und $t$ basieren auf $i$ und sind deshalb an dieser Stelle der Analyse bedeutungslos. Induktionsvariablen können mit Hilfe der meisten Operationen gebildet und eindeutig erkannt werden. Hierzu gehören die Befehle 'mov', 'add', 'sub', 'mul', 'lsl', 'lsr' und 'neg'. Beispielsweise ermöglicht das implementierte Verfahren in Beispiel 6.4.1 auch die Einsparung des Speicherzugriffes auf $a[j]$ durch einen früheren Zugriff auf $a[t]$.

**Beispiel 6.4.1** Erkennung von Induktionsvariablen

$i$=10;
while ($i$<20) {
  $k$=$i$+4;
  $t$=$k$·4;
  $j$=$t$-4;
  $a[t]$=$a[j]$+1;
  $i$=$k$-3;
}

An dieser Stelle wird auch ermittelt, um welchen Wert sich die Induktionsvariable pro Iteration verändert. In dem Beispiel 6.4.1 wird das $i$ pro Iteration um 1 erhöht. Innerhalb einer Schleife muss mindestens eine Induktionsvariable auftreten, sonst wird die Analyse der Schleife beendet.

3. Erkennung von Iterationsgrenzen
Bekannte Iterationsgrenzen der Induktionsvariablen können die Ergebnisse der Array-Datenflussanalyse (siehe Kapitel 7) verbessern.

   Am Schleifenanfang wird ein komplexes Verfahren verwendet, um den Startwert der Induktionsvariable zu ermitteln. In dem Beispiel 6.4.2 wird die Induktionsvariable $b$ erkannt und der Startwert könnte beispielsweise '360+SP' sein. In diesem Fall liegt das Array auf dem Stack und $a[90]$ befindet sich an der Position 360 bezüglich des Stackpointers. Das implementierte Verfahren erlaubt bei diesem Beispiel die Optimierung der Speicherzugriffe '*($b$-1)' und '*($b$-2)'.

   Um das Iterationsende der Schleife zu bestimmen, analysiert das Verfahren die Instruktionen vor dem Schleifenaussprung. Gesucht wird eine 'cmp'-Anweisung. Falls eine geeignete Instruktion gefunden wird, kann das Iterationsende der Schleife ermittelt werden.

**Beispiel 6.4.2** Startwertanalyse

```
b=&a[90];
while (i <20) {
  *b=*(b-1)+*(b-2);
  i+=2;
  b++;
}
```

Falls das Verfahren einen Wert nicht bestimmen konnte, dann wird dieser als unbekannt gekennzeichnet.

4. Rekonstruktion der Indexfunktionen
   Das Verfahren sucht zuerst die Speicherzugriffe innerhalb der Schleife. Bei jedem dieser Speicherzugriffe wird anschließend die zugehörige Indexfunktion rekonstruiert. Tabelle 6.1 zeigt das allgemeine Vorgehen.

| Assemblercode | | Ersetzungsregeln | Ergebnis |
|---|---|---|---|
| add | R0,SP,#0 | 'R0' → '(SP+0)' | 5. '[((SP+0)+(R2-1)·4+0]' |
| sub | R3,R2,#1 | 'R3' → '(R2-1)' | 4. '[(R0+(R2-1)·4)+0]' |
| lsl | R3,R3,#2 | 'R3' → 'R3·4' | 3. '[(R0+R3·4)+0]' |
| add | R0,R0,R3 | 'R0' → '(R0+R3)' | 2. '[(R0+R3)+0]' |
| ldr | R0,[R0,#0] | 'R0' → '[R0+0]' | 1. '[R0+0]' |

Tabelle 6.1: Rekonstruktion einer Indexfunktion

Jeder Instruktion wird eine Ersetzungsregel zugeordnet. Eine Ersetzungsregel spiegelt das Verhalten der Instruktion wieder. In dem Beispiel aus der Tabelle 6.1 gibt es einen Speicherzugriff 'ldr R0,[R0,#0]', der die Indexfunktion mit 'R0+0' initialisiert. Die einzelnen Ersetzungsregeln werden dann entgegengesetzt zur Reihenfolge der Programmausführung auf diese Indexfunktion angewendet. Hierbei müssen aber Kontrollstrukturen beachtet werden. Insgesamt wird bei dem Beispiel die Indexfunktion '((SP+0)+(R2-1)·4)+0' berechnet. Die folgenden Schritte vereinfachen die Funktion zu dem Ausdruck '4·R2+SP-4':

- Multiplizieren
  (Beispiel: '((SP+0)+(R2-1)·4)+0' → '((SP+0)+(R2·4-1·4))+0')
- Minuszeichen vor den Klammern entfernen
  (Beispiel: '-(a-b)' → '+(-a+b)')
- Klammern aufheben
  (Beispiel: '((SP+0)+(R2·4-1·4))+0' → 'SP+0+R2·4-1·4+0')
- Konstanten berechnen
  (Beispiel: 'SP+0+R2·4-1·4+0' → 'SP+R2·4-4')
- Distributivgesetz anwenden
  (Beispiel: 'Rx·4-2·Rx' → '2·Rx')

**Definition 6.4.3** Eine *affine Funktion* sei $f(x) = ax + b$ mit zwei konstanten Zahlen $a$ und $b$.

Die Indexfunktion '4·R2+SP-4' könnte beispielsweise mit dem C-Programmteil '...=$a$[$i$-1];' generiert werden. Das R2 stellt bei diesem Beispiel die Induktionsvariable dar. Abschließend wird noch überprüft, ob die Indexfunktion konstant, affin oder nicht affin ist. Die angepasste Array-Datenflussanalyse (siehe Kapitel 7) benötigt diese Informationen.

Ein Vorteil dieser Vorgehensweise ist, dass sich die Induktionsvariable an beliebigen Stellen im Programm verändern darf, da jede Indexfunktion bis zum Anfang der Schleife zurückverfolgt wird. Bei dem Beispiel 6.4.1 würde die Indexfunktion von $a[j]$ '4·$i$+12' lauten.

Die Analyse, die hier für die Indexfunktionen eingesetzt wird, kommt auch bei der Erkennung der Induktionsvariablen und bei ihrer Grenzwertanalyse zum Einsatz. Die Induktionsvariablen werden analysiert, indem die Ersetzungsregeln entgegen der Ausführungsreihenfolge auf jede mögliche Induktionsvariable ab dem Schleifenende bis zum Schleifenanfang angewendet werden. Die Menge der möglichen Induktionsvariablen ergibt sich aus den verwendeten Variablen im Schleifenkörper. Das Verfahren markiert auch Variablen, die innerhalb der Schleife konstant sind, als Induktionsvariablen. Im Beispiel 6.4.1 ergibt sich für $i$ die Funktion '(($i$+4)-3)', die dann zu '$i$+1' vereinfacht werden kann. In dieser Funktion muss die ursprüngliche Variable (hier $i$) noch vorhanden sein, damit es sich um eine Induktionsvariable handelt. Der Wert, um den sich die Induktionsvariable pro Iteration erhöht, ist 1 im Beispiel 6.4.1.

Für die Erkennung des Startwertes einer Induktionsvariablen werden die Ersetzungsregeln ab dem Schleifenanfang entgegen der Ausführungsreihenfolge auf die Induktionsvariable angewendet. Ersetzungsregeln von Instruktionen außerhalb der Schleife ermitteln deswegen den Startwert. Das Verfahren muss immer die Kontrollstrukturen beachten. Für die Ermittlung des Abbruchkriteriums werden die Ersetzungsregeln auf die Vergleichsoperation 'cmp' geeignet angewendet.

Vor der Vereinfachung einer Indexfunktion erfolgt eine Reduzierung der Anzahl der Induktionsvariablen innerhalb der Funktion. Voraussetzung für die Durchführung dieser Reduzierung ist, dass die nachfolgenden Werte bekannt sind:

- Schrittweite der Induktionsvariablen
- Startwert der Induktionsvariablen

Das Verfahren Registerpipelining generiert die Induktionsvariable $Vx$, wobei das $x$ für eine bestimmt Zahl steht. Diese Variable erhält die Ersetzungsregel '$Vx$', die Iterationsschrittweite 1 und den Startwert 0. Alle anderen Induktionsvariablen erhalten folgende Ersetzungsregel:

'(Iterationsschrittweite·$Vx$+Startwert)'

**Beispiel 6.4.4** Unterschiedliche Induktionsvariablen

```
int i,r,s,z,a[1500];
...
z=0;
r=2;
s=4;
for(i=1; i <200; i++)
{
   r+=1;
   a[r]=i;
   r+=3;
   z+=a[s-1];
   s+=4;
}
...
```

**Beispiel 6.4.5** Assemblerdarstellung des Beispiels 6.4.4

```
...
     mov  V0,#0            // z=0;
     mov  V2,#2            // r=2;
     mov  V4,#4            // s=4;
     mov  V6,#1
     add  V8,SP,#0
     mov  V10,#12
     mov  V12,#2
LL4
     add  V14,V2,#1        // r+=1;
     lsl  V16,V14,#2
     str  V6,[V8,V16]      // a[r]=i;
     add  V2,V14,#3
     ldr  V18,[V8,V10]
     add  V0,V0,V18        // z+=a[s-1];

     add  V4,V4,#4
     mov  V21,#16
     add  V10,V10,V21
     mov  V23,V12          // for(i=1; i <200; i++)
     add  V6,V6,#1
     add  V12,V12,#1
     cmp  V23,#200
     blt  LL4
...
```

Die Tabelle 6.2 enthält exemplarisch die Ersetzungsregeln für das Beispiel 6.4.4 und 6.4.5, das mit der Optimierungstechnik Induction Variable Elimination optimiert wurde. Die Tabelle 6.3 zeigt die einzelnen Speicherzugriffe und ihre Indexfunktionen. Das $Vx$ entspricht in dem Beispiel der Variablen $V27$. Diese Vorgehensweise ermöglicht auch die Optimierung von $a[s\text{-}1]$ in dem Beispiel 6.4.4 mit Hilfe der Optimierungstechnik Registerpipelining.

| Induktions-variable | Schritt-weite | Start-wert | Ersetzungsregel |
|---|---|---|---|
| V2 | 4 | 2 | (#4·V27+#2) |
| V4 | 4 | 4 | (#4·V27+#4) |
| V6 | 1 | 1 | (#1·V27+#1) |
| V8 | 0 | #0+#1·SP | (#0·V27+#0+#1·SP) |
| V10 | 16 | 12 | (#16·V27+#12) |
| V12 | 1 | 2 | (#1·V27+#2) |
| V27 | 1 | 0 | V27 |

Tabelle 6.2: Ersetzungsregeln für die Induktionsvariablen

| Speicherzugriff | Indexfunktion | |
|---|---|---|
| | ursprünglich | vereinfacht |
| str V6, [V8, V16] | (V8+(V2+#1)·#4) | #16·V27+#12+#1·SP |
| ldr V18, [V8, V10] | (V8+V10) | #16·V27+#12+#1·SP |

Tabelle 6.3: Speicherzugriffe mit Indexfunktionen

5. Array-Datenflussanalyse
   Die Array-Datenflussanalyse analysiert den Datenfluss zwischen einzelnen Array-Referenzen innerhalb von Schleifen. Die Qualität der Analyse beeinflusst, wie häufig die Optimierungstechnik Registerpipelining angewendet werden kann. Im Beispiel 6.4.6 kann Registerpipelining den Speicherlesezugriff auf $a[i\text{-}2]$ durch einen früheren Zugriff auf $a[i]$ ersetzen. Eine gute Array-Datenflussanalyse stellt sicher, dass der Speicherzugriff auf $a[2{\cdot}i\text{-}5]$ keine Speicherinhalte verändert, die in einem Register der Pipeline zwischengespeichert sind.

**Beispiel 6.4.6** Aufgabe der Array-Datenflussanalyse

```
i=10;
while (i<50) {
  a[i]=a[i-2]+1;
  a[2·i-5]=i;
  i++;
}
```

Die Erkenntnisse einer Array-Datenflussanalyse können für unterschiedliche Optimierungen eingesetzt werden. Skalare Datenflussanalysen sind nicht auf Array-Referenzen anwendbar, da eine Array-Referenz in Abhängigkeit der Indexfunktion verschiedene Speicheradressen adressieren kann.

Nach [Fra99] ist das allgemeine Problem bei der Bestimmung von Datenabhängigkeiten zwischen Array-Referenzen nicht entscheidbar. Deswegen wird die Suche nach einer exakten Lösung häufig eingeschränkt:

- Das Problem wird nur innerhalb von speziellen Problemklassen untersucht (z.B. affine Funktionen).
- Eine Approximation wird verwendet.

In [Fra99] wurden einige Array-Datenflussanalysen untersucht und miteinander verglichen. Hieraus ergeben sich insgesamt drei geeignete Array-Datenflussanalysen, die für die Optimierungstechnik Registerpipelining eingesetzt werden können:

- DSA-Verfahren
  Das DSA-Verfahren erfordert einen sehr hohen Implementierungsaufwand.
- Stretched-Loop-Verfahren
  Der Aufwand beim Stretched-Loop-Verfahren ist hoch und es ermöglicht eine Analyse von affinen Indexfunktionen mit einer hohen Präzision. Die Schleifen, die untersucht werden, dürfen die multiple-entry/multiple-exit Eigenschaften besitzen. Das Verfahren benötigt aber auch die Ergebnisse eines angepassten $\delta$-Verfahrens, das die maximalen Iterationsdistanzen berechnet. Das Wort 'angepasst' wird an dieser Stelle verwendet, da die normale $\delta$-Array-Datenflussanalyse nur single-entry/single-exit Schleifen untersuchen kann.
- $\delta$-Verfahren
  Die $\delta$-Array-Datenflussanalyse repräsentiert das einfachste Verfahren innerhalb dieser Aufzählung und es kann affine Indexfunktionen mit einer mittleren Präzision analysieren.

Die Hauptaufgabe dieser Diplomarbeit ist die Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining. Deswegen sollte eine Array-Datenflussanalyse gewählt werden, die möglichst gut das Verfahren Registerpipelining unterstützt. Die $\delta$-Array-Datenflussanalyse kann relativ leicht an die speziellen Anforderungen des Verfahrens Registerpipelining und des ARM7TDMI-Prozessors angepasst werden. Zudem lässt sich das $\delta$-Verfahrens noch an vielen Punkten verbessern. Insgesamt bewirken die Veränderungen, die in Kapitel 7 beschrieben sind, dass die Analyse sehr präzise und schnell wird.

Die $\delta$-Array-Datenflussanalyse stellt Informationen für die Erkennung von Optimierungsmöglichkeiten zur Verfügung.

6. Optimierungsmöglichkeiten erkennen, generieren und bewerten
   Es gibt die Optimierungsvarianten 'VNormal', 'VVerwendung', 'VErzeugung' und 'VBeides', die zuerst erkannt, eventuell generiert und dann

bewertet werden. Die einzelnen Varianten hängen von dem jeweiligen Registerdruck und von den unterschiedlichen Umsetzungsmöglichkeiten des Verfahrens Registerpipelining (nach Kapitel 4.2.4) ab. Weiterhin werden auch die genauen Positionen der Speicherzugriffe bestimmt und berücksichtigt. Die Bewertung der einzelnen Optimierung erfolgt mit der Kostenbewertungsfunktion des Compilers.

7. Registerdruck ermitteln
   Das Verfahren ermittelt die verfügbaren Register, um zu entscheiden welche Optimierungen durchgeführt werden. Vor dem Aufruf des Verfahrens Registerpipelining wird die Registerallokation des Compilers 'arm12cc' aufgerufen und die von ihr gelieferten Werte werden hier analysiert. Bei der implementierten Version von Registerpipelining kann das Registerspilling optional unterdrückt werden, da dadurch die Optimierung an Effizienz verlieren würde. Trotzdem könnte eine Programmoptimierung erfolgreich sein,wenn beispielsweise das Spilling im On-Chip-Speicher erfolgt und wenn der eingesparte Speicherzugriff im externen Speicher lag.

8. Optimierungen umsetzen
   Nachdem der Registerdruck analysiert wurde, können die Optimierungen durchgeführt werden. Die Umsetzung einzelner Optimierungen kann mit unteren und hohen Registern erfolgen. Die Technik Copy Propagation kann aber nachfolgend die unteren Register besser optimieren. Deswegen sollten die unteren Register am Anfang und am Ende einer Registerpipeline stehen.

   Die Suche nach einer optimalen Lösung ist schwierig, da sich die einzelnen Programmoptimierungen auch untereinander beeinflussen und auch wegen der nachfolgenden Rückkopplung. Das Verfahren, dass die Optimierungen bei dieser Implementierung auswählt, verwendet einen einfachen Greedy-Algorithmus.

   Wenn das Verfahren keine Speicherzugriffe ersetzen konnte, wird die Analyse der Schleife beendet.

9. Registerallokation und Rückkopplung
   Eine Rückkopplung wurde an dieser Stelle eingefügt, weil die Bestimmung des Registerdrucks nach einzelnen Veränderungen nicht nachvollzogen wird. Die Optimierungstechniken Copy Propagation und Useless Code Elimination optimieren zuerst den generierten Programmcode. Die Registerallokation des Compilers 'arm12cc' wird aufgerufen und der Registerdruck wird erneut ermittelt (Punkt 7 der Aufzählung). Versuche in der Praxis haben aber ergeben, dass die Rückkopplung selten zusätzliches Optimierungspotential freilegt.

## 6.5 Zusammenfassung

Die Optimierungstechnik Registerpipelining wurde in dieser Diplomarbeit im Backend des Compilers 'arm12cc' platziert, um die Codegenerierung zu optimieren. Das Verfahren besitzt die folgenden Eigenschaften:

- Integration der Optimierungstechnik
  - Schnittstelle
    Die Schnittstellen zu anderen Komponenten innerhalb des Compilers sind einfach gehalten um zukünftige Entwicklungen nicht zu stören. Beispielsweise kann die Registerallokation relativ unabhängig von dem Verfahren Registerpipelining entwickelt werden.
  - Anforderungen an den Kompilierungsprozess
    Die Optimierungstechnik benötigt keine Ausführungswahrscheinlichkeiten, die durch ein dynamisches Profiling ermittelt werden müssen. Diese Eigenschaft vereinfacht den Ablauf der Codegenerierung.
  - Kostenmodell
    Das Kostenmodell des Compilers 'arm12cc' wird für die Bewertung der einzelnen Instruktionen eingesetzt. Diese Eigenschaft stellt sicher, dass bei einer Änderung des Kostenmodells die Funktionsweise der Optimierungstechnik erhalten bleibt.
- Optimierungseigenschaften
  - Redundant Load Elimination
    Die Optimierungstechnik Registerpipelining führt auch eine Redundant Load Elimination durch, die in Kapitel 4.1 beschrieben ist.
  - Speicherzugriffsarten
    Array- und auch Pointerzugriffe werden optimiert.
  - Optimierungspotential
    Wenn eine Optimierungsmöglichkeit erkannt wurde, dann wird das zur Verfügung stehende Optimierungspotential weitestgehend ausgeschöpft. Bessere und umfangreichere Analysen könnten die Einsatzmöglichkeiten der Optimierungstechnik aber noch vergrößern.
- Schleifenanalyse
  - Schleifenart
    Registerpipelining kann beliebig konstruierte Schleifen (z.B.'while' oder 'goto') analysieren. Um eine Optimierung durchzuführen, müssen die Schleifen aber die single-entry/single-exit Eigenschaft besitzen.
  - Präzision
    Die angepasste $\delta$-Array-Datenflussanalyse, die in Kapitel 7 vorgestellt wird, erreicht im Vergleich zu den anderen in [Fra99] vorgestellen Verfahren eine sehr hohe Präzision.
- Induktionsvariablen
  - Position
    Durch die Art der Ermittlung der Indexfunktionen darf sich die Induktionsvariable auch mehrmals im Schleifenkörper verändern.

- Aufbau
  Für eine genaue Analyse der Induktionsvariablen werden die meisten Befehle berücksichtigt.
- Analysemöglichkeiten
  Auch Speicherzugriffe mit unterschiedlichen Induktionsvariablen können analysiert werden (Beispiel 6.4.1 und 6.4.4).

Die Implementierung der Optimierungstechnik Registerpipelining zeichnet sich durch verschiedene positive Eigenschaften aus. Die verwendete $\delta$-Array-Datenflussanalyse wird in dem folgendem Kapitel vorgestellt.

# Kapitel 7

# $\delta$-Array-Datenflussanalyse

Dieses Kapitel stellt eine Array-Datenflussanalyse basierend auf dem $\delta$-Verfahren vor, dass in der Literatur [Fra99] und [ED93] beschrieben wird. Untersucht wird die Array-Datenflussanalyse bzgl. einer nachfolgenden Anwendung der Optimierungstechnik Registerpipelining für das ATMEL-Evaluationboard.

In den folgenden Abschnitten werden zuerst die grundlegenden Eigenschaften und Datenstrukturen mit den dazu gehörigen Operationen vorgestellt. Daran anschließend werden die ursprünglichen Transferfunktionen und die durchgeführten Änderungen beschrieben. Die Transferfunktionen und das iterative Lösungsverfahren berechnen danach die Iterationsdistanzen. Abschließend wird die angepasste $\delta$-Array-Datenflussanalyse untersucht und anhand eines Beispiels erklärt.

## 7.1 Grundlegende und ursprüngliche Eigenschaften

Die $\delta$-Array-Datenflussanalyse unterstützt einige Optimierungsverfahren, zu denen auch Registerpipelining gehört. Es beschränkt sich auf die Analyse affiner Indexfunktionen von Array-Referenzen, die relativ häufig vorkommen. Weiterhin müssen die Schleifen die single-entry/single-exit Eigenschaft aufweisen und die in ihnen enthaltene Induktionsvariable muss von 1 bis zu einer festen oberen Grenze laufen. Die Induktionsvariable besitzt die Schrittweite 1 und sie darf nur am Anfang oder am Ende der Schleife verändert werden.

Die Datenflussanalyse unterstützt unterschiedliche Analysen bzgl. must- und may-Informationen. Hierdurch wird unterschieden, ob bestimmte Eigenschaften bei einer Instruktion gelten können (may) oder müssen (must). Die Optimierungstechnik Registerpipelining benötigt die Analyse $\delta$-available-values. Es handelt sich um ein must-Problem und es wird eine Vorwärtsanalyse verwendet, die die Datenflussrichtung vorwärts analysiert. Für das Verfahren Registerpipelining analysiert die $\delta$-Array-Datenflussanalyse, welche Speicherinhalte über welche Iterationsdistanzen erhalten bleiben (Richtung: vorwärts), und nicht verändert werden. Dies umschließt direkte Definitionen von Speicherinhalten (Beispiel: $a[i]= \ldots$ ), als auch den Gebrauch von Speicherinhalten (Beispiel: $\ldots =a[i]$).

**Definition 7.1.1** Eine *Referenz* sei ein lesender oder ein schreibender Speicherzugriff. Ein *Gebrauch* sei ein lesender Zugriff. Eine *Definition* sei ein schreibender Zugriff.

Die Idee des $\delta$-Verfahrens ist die Berechnung maximaler Iterationsdistanzen. Sie bezeichnen die maximale Anzahl an Iterationen, mit der ein Speicherinhalt erhalten bleibt. In dem Beispiel 7.1.2 bleibt der Speicherinhalt von $a[i]$ über 4 Iterationen erhalten. Das $a[i\text{-}5]$ beschreibt fünf Iterationen später den Wert von $a[i]$.

**Beispiel 7.1.2** Berechnung maximaler Iterationsdistanzen

```
...
for( i=10; i<100; i++) {
   a[i-5]=5;
   b+=a[i];          // Wert verändert sich nach 5 Schleifeniterationen
}
...
```

## 7.2 Grundlegende Datenstrukturen und Operationen

Diese Diplomarbeit verzichtet im Gegensatz zu der Originalliteratur ([ED93] und [Fra99]) auf die explizite Darstellung von Verbänden. Stattdessen wird eine implementierungsnähere Beschreibung gewählt. Zur Kennzeichnung, dass eine bestimmte Eigenschaft über keine Iterationsdistanz gültig ist, wird das Zeichen $\perp$ (bottom) verwendet. Das Zeichen $\top$ (top) kennzeichnet hingegen die Gültigkeit über jede Iterationsdistanz. Ein Programm kann diese zwei Symbole mit Hilfe eines Flags zusätzlich zu dem Iterationsdistanzwert $id$ darstellen. Hierzu sei eine Iterationsdistanz $\delta$ folgendermaßen definiert:

$$\delta = \begin{cases} \perp & \text{, falls } flag_\delta = 0 \\ id_\delta & \text{, falls } flag_\delta = 1 \\ \top & \text{, falls } flag_\delta = 2 \end{cases} \tag{7.1}$$

Somit ergibt sich eine Iterationsdistanz $\delta$ aus den beiden Werten $id_\delta$ und $flag_\delta$. Die später benötigten Operationen auf Iterationsdistanzen seien wie folgt definiert:

$$\delta++ = \begin{cases} \top & \text{,falls } \delta = \top \\ \perp & \text{,falls } \delta = \perp \\ id_\delta + 1 & \text{,sonst} \end{cases} \tag{7.2}$$

$$\min(\delta, \beta) = \begin{cases} \perp & \text{,falls } \delta = \perp \text{ oder } \beta = \perp \\ \top & \text{,falls } \delta = \top \text{ und } \beta = \top \\ id_\beta & \text{,falls } \delta = \top \\ id_\delta & \text{,falls } \beta = \top \\ \min(id_\delta, id_\beta) & \text{,sonst} \end{cases} \tag{7.3}$$

$$\max(\delta, \beta) = \begin{cases} \top & \text{,falls } \delta = \top \text{ oder } \beta = \top \\ \perp & \text{,falls } \delta = \perp \text{ und } \beta = \perp \\ id_\beta & \text{,falls } \delta = \perp \\ id_\delta & \text{,falls } \beta = \perp \\ \max(id_\delta, id_\beta) & \text{,sonst} \end{cases} \tag{7.4}$$

Die $\delta$-Array-Datenflussanalyse analysiert die einzelnen Schleifen eines Programms. Sie beginnt mit der Schleife, die am häufigsten ausgeführt wird. Innerhalb der einzelnen Schleifen wird eine Datenstruktur benötigt, um die Transferfunktionen zu speichern. Eine Transferfunktion ist einer Instruktion zugeordnet und spiegelt eine bestimmte Eigenschaft der Instruktion wieder. Für die Optimierungstechnik Registerpipelining wird untersucht, über welche Iterationsdistanz ein Wert erhalten bleibt.

Im Gegensatz zu der Originalliteratur wird zur Speicherung der Transferfunktionen eine wesentlich kleinere Datenstruktur verwendet, nur Einträge für Instruktionen besitzt, die Speicherzugriffe enthalten. Das $m$ ist die Anzahl der Referenzen innerhalb der zu untersuchenden Schleife. Eine $(m+1, m)$-Matrix dient zur Speicherung der Transferfunktionen. Hierdurch kann eine Transferfunktion zu jedem möglichen Paar von Referenzen abgelegt werden. Durch die '+1' werden die Iterationsübergänge berücksichtigt. Die Referenzen und die Iterationsübergänge sind in Knoten abgelegt. Die Bezeichnung für einen Knoten entspricht einer Nummer $n$ mit $1 \leq n \leq m+1$. $f_n^d$ sei ein Element der Matrix und eine Funktion, die die Transferfunktion bei einem Knoten $n$ darstellt, die die Referenz $d$ beeinflusst. Der Knoten $m+1$ enthält keine Referenz und jeder andere Knoten enthält genau eine Referenz.

Zusätzlich wird gegenüber der Originalliteratur auf die Konstruktion eines Schleifenkontrollflussgraphen (Loop Control Flow Graph) verzichtet. Das Verfahren ist durch diese Änderung schneller und die Schleifen müssen nicht mehr die single-entry/single-exit Eigenschaft aufweisen. Das Hauptziel sollte die Analyse der innersten Schleife sein, auch wenn diese nicht die single-entry/single-exit Eigenschaft besitzt.

Der Graph $G = (V, E)$ wird zunächst konstruiert. Die Datenstruktur eines Graphen besteht in dieser Diplomarbeit aus Knoten mit den zugehörigen Adjazenzlisten. Abbildung 7.1 zeigt unterschiedliche Graphen für ein Beispielprogramm. In dem Beispiel ist $m$ gleich 3. In dem Graphen $G$ stellen die Instruktionen aus einem Programm einzelne Knoten dar und gerichtete Kanten beschreiben die Reihenfolge, in der die Instruktionen ausgeführt werden können. Bedingte Sprungbefehle besitzen deshalb oft zwei Knoten als mögliche Nachfolger.

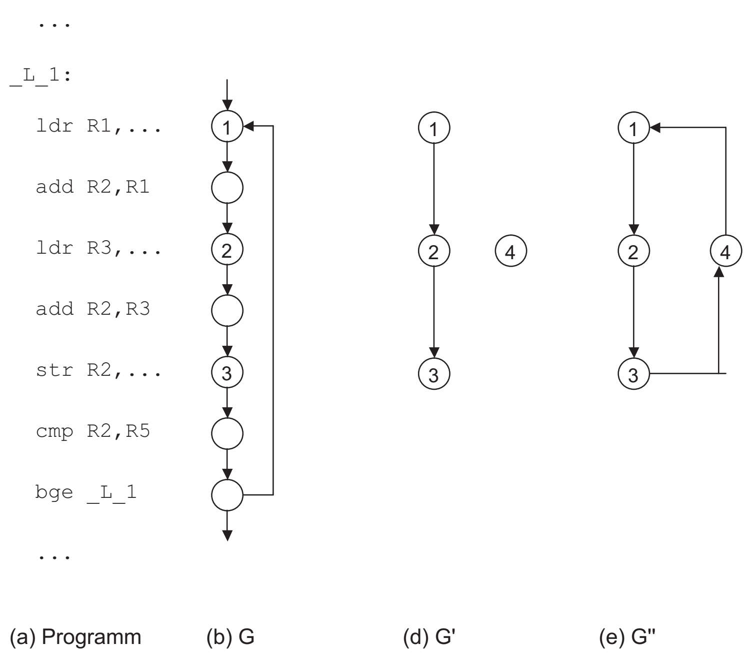


Abbildung 7.1: Überblick über die verwendeten Graphen

**Definition 7.2.1** Ein Knoten $v$ ist ein *Vorgänger/Nachfolger* von Knoten $u$, falls eine Kante von $v/u$ nach $u/v$ führt.

**Definition 7.2.2** Ein *Weg* von $v_1$ zu $v_r$ existiert genau dann, wenn es eine gerichtete Kantenfolge ($v_1$,$v_2$,...$v_r$) gibt. Innerhalb einer *gerichteten Kantenfolge* gibt es die gerichteten Kanten ($v_i$,$v_{i+1}$) für alle $i$ von 1 bis $r-1$.

**Definition 7.2.3** Ein *direkter Weg* von $v_1$ zu $v_r$, sei ein Weg auf dem keine weiteren Referenzen innerhalb der anderen Knoten auf dem Weg auftreten und der nur aus Tree-, Forward- und Crosskanten besteht (Kanteneinteilung durch einen DFS-Algorithmus nach [Hof99]).

Sei $G' = (V', E')$ ein Graph mit der Knotenmenge $V' = \{1, .., (m+1)\}$. Zwischen den einzelnen Knoten aus $V'$ existieren Kanten. Eine gerichtete Kante ($v_s$,$v_r$)$\in E'$ mit $v_s$,$v_r \in V'$ verbindet zwei Knoten $v_s$ und $v_r$ miteinander, wenn ein direkter Weg in dem Graphen $G$ für die entsprechenden Knoten von $v_s$ zu $v_r$ innerhalb der zu untersuchenden Schleife existiert.

Schleifen innerhalb des Graphen G werden für die Array-Datenflussanalyse nicht mehr berücksichtigt. Hierdurch kann eine schleifenübergreifende und schnelle Analyse (siehe Kapitel 7.6 und 7.7) durchgeführt werden.

Der Graph $G'' = (V', E'')$ wird konstruiert, indem folgende Kanten zusätzlich zu der Kantenmenge $E'$ in $E''$ eingefügt werden. Falls einer der Knoten keinen Vorgänger hat, dann erhält dieser Knoten den Knoten $m+1$ als Vorgänger.

Falls ein Knoten keinen Nachfolger hat, dann erhält dieser Knoten den Knoten $m+1$ als Nachfolger. Diese zusätzlichen Beziehungen zwischen den Knoten werden mit entsprechenden Kanten in $E''$ produziert.

## 7.3 Ursprüngliche Transferfunktionen

In den folgenden Kapiteln wird hauptsächlich ein must-Problem und eine Vorwärtsanalyse behandelt, da diese Konstellation für die Optimierungstechnik Registerpipelining benötigt wird. Nur das Kapitel 7.5.5 geht kurz auf may- und Rückwärts-Analysen ein.

Insgesamt gibt es drei unterschiedliche Arten von Transferfunktionen, die in diesem Kapitel nach [ED93] beschrieben werden. Die Erzeugungsfunktion und die Erhaltungsfunktion werden bei Knoten angewendet, die eine Referenz enthalten. Die Iterationsübergangsfunktion ist ausschließlich für den Knoten $m+1$ bestimmt. Das $i$ sei bei den Beispielen zu den Transferfunktionen eine Induktionsvariable, die pro Iteration um eins erhöht wird.

### 7.3.1 Erzeugungsfunktionen

Wenn der Knoten $n$ identisch ist mit dem Knoten, in dem die Referenz $d$ liegt, dann wird eine Erzeugungsfunktion[1] verwendet. Die gültige Iterationsdistanz beträgt mindestens 0. Mit $x$, der bisher gültigen Iterationsdistanz, ergibt sich:

$$f_n^d(x) = \max(x, 0) \tag{7.5}$$

### 7.3.2 Erhaltungsfunktionen

Wenn die Referenz $d'$, die durch den Knoten $n$ repräsentiert wird, die Werte einer Referenz $d$ nicht oder teilweise verändert, dann wird eine Erhaltungsfunktion verwendet. Die gültige Iterationsdistanz wird höchstens kleiner.

$$f_n^d(x) = \min(x, p_n^d) \tag{7.6}$$

Das $p_n^d$ sei eine Konstante, die mit Hilfe der folgenden Fallunterscheidung bestimmt wird.

1. Wenn $d$ und $d'$ voneinander unabhängig sind, indem unterschiedliche Arrays (z.B. $d = a[i]$ und $d' = b[i]$) angesprochen werden, oder wenn $d'$ nur ein Gebrauch ist, dann bleibt die Gültigkeit der Iterationsdistanz erhalten.

$$p_n^d = \top \tag{7.7}$$

[1] Die Bedingung für die Anwendbarkeit der Erzeugungsfunktionen wurde gegenüber der Originalliteratur leicht verändert, damit die Ergebnisse der Analyse leichter interpretiert werden können.

2. Sonst ist $d'$ eine Definition und es liegt eventuell eine Abhängigkeit vor.

$$pr(n,d) = \begin{cases} 0, & \text{falls ein Weg in G}' \text{ existiert von dem} \\ & \text{Knoten, der } d \text{ enthält, zu Knoten } n \\ 1, & \text{sonst} \end{cases} \tag{7.8}$$

Das $pr(n,d)$ spiegelt die grundlegende Iterationsdistanz zwischen zwei Anweisungen wieder. In dem Beispiel 7.3.1 währe $pr(n = 1, d) = 1$.

**Beispiel 7.3.1** Berechnung des Wertes $pr(n,d)$

for( $i$=1; $i$ <10; $i$++) {
  ... =$a$[$i$];  // $d' = a[i], n = 1$
  ... =$a$[$i$-2];  // $d = a[i\text{-}2]$
}

Sei $d = a[idx_1(i - \delta)]$ und $d' = a[idx_2(i)]$, wobei $a$ ein Array ist und $idx(i)$ eine Indexfunktion in Abhängigkeit von der Induktionsvariablen $i$ darstellt. Dann lässt sich $p_n^d$ wie folgt berechnen:

$$\begin{aligned} p_n^d = \quad & \max\{\delta \mid \delta \in [pr(n,d)..UB], \forall i \in [1..UB], \\ & \forall \delta' \in [pr(d,n)..\delta] : idx_2(i) \neq idx_1(i - \delta')\} \end{aligned} \tag{7.9}$$

$UB$ sei die feste obere Iterationsgrenze einer Induktionsvariablen. In dem Beispiel 7.3.1 ist $UB$ gleich 9. Mit der Gleichung 7.9 wird das maximale $\delta$ gesucht, so dass für alle möglichen Werte von $i$ und für alle relevanten Iterationsdistanzen kleiner gleich $\delta$ gilt, dass sich die entsprechenden Speicherbereiche nicht überschneiden. Für eine einfachere Implementierung kann die Gleichung noch leicht umgestellt werden mit

$$idx_1(i) = a_1 \cdot i + b_1 \wedge idx_2(i) = a_2 \cdot i + b_2 \tag{7.10}$$

$$\begin{aligned} & idx_1(i - k(i)) - idx_2(i) = 0 \\ \Longleftrightarrow \quad & k(i) = \tfrac{a_1 - a_2}{a_1} \cdot i + \tfrac{b_1 - b_2}{a_1} \end{aligned} \tag{7.11}$$

folgt:

$$\begin{aligned} p_n^d = \quad & \max\{\delta \mid \delta \in [pr(n,d)..UB], \forall i \in [1..UB], \\ & \forall \delta' \in [pr(d,n)..\delta] : k(i) \neq \delta'\} \end{aligned} \tag{7.12}$$

Das $k(i)$ gibt die Iterationsdistanz für ein $i$ an, bei der zwei Speicherzugriffe die gleiche Speicheradresse adressieren. Mit Hilfe der folgenden Fallunterscheidung erfolgt letztlich die Berechnung der $p_n^d$ Werte in Anlehnung an die Gleichung 7.12.

(a) $\exists i : k(i) = pr(d,n)$ :[2] Dann ändert $d'$ mindestens einen Wert der Referenz $d$ sofort. Dieser Fall tritt beispielsweise auf, wenn $d = d'$ und $pr(d,n) = 0$ ist. Es gilt:

$$p_n^d = \perp \tag{7.13}$$

(b) $\forall i : k(i) < pr(d,n)$ : Dann ändert $d'$ nie einen Wert der Referenz $d$. Ein Beispiel hierfür wäre $d = a[i]$ und $d' = a[i{+}5]$. Es gilt:

$$p_n^d = \top \tag{7.14}$$

(c) Sonst ändert $d'$ einen Wert der Referenz $d$. Es gilt:

$$p_n^d = \lceil \min\{k(i) | i \in [1..UB], k(i) > pr(d,n)\} \rceil - 1 \tag{7.15}$$

Gewählt wird von allen Speicherzugriffen, die sich überschneiden, der mit der niedrigsten dazwischen liegenden Iterationsdistanz. Hiervon wird die Zahl eins subtrahiert und es ergibt sich ein geeigneter Wert für $p_n^d$. Wenn beispielsweise $d' = a[i]$ und $d = a[i{+}5]$ ist, dann ist $p_n^d = 4$.

### 7.3.3 Iterationsübergangsfunktionen

Die Iterationsübergangsfunktion beschreibt einen Übergang zu einer nachfolgenden Iteration. Sie erhöht die Iterationsdistanz $x$ um eins und wird nur am Knoten $m+1$ verwendet.

$$f_n^d(x) = x++ \tag{7.16}$$

## 7.4 Optimierte Transferfunktionen

Die Voraussetzungen für die $\delta$-Array-Datenflussanalyse sind relativ schwerwiegend. Zum Beispiel muss eine Induktionsvariable von 1 bis zu einer festen oberen Grenze laufen. Eine Lösung hierfür wären Schleifentransformationen, die Schleifen umformen können, damit sie nachher dieses Kriterium erfüllen. Diese Transformationen sind dann aber an das Verfahren Registerpipelining gekoppelt und könnten anderen Optimierungsverfahren eventuell nicht mehr zur Verfügung stehen.

Ein weiteres Problem ist, dass bestimmte Abhängigkeiten nur schlecht erkannt werden. Beispiel 7.4.1 zeigt dieses Problem. Obwohl die Speicherzugriffe in diesem Beispiel auch unabhängig sein können, wird trotzdem eine Abhängigkeit bei einer Iterationsdistanz von 2 erkannt ($p_n^d = 1$ bei der Gleichung 7.17). Die Transferfunktionen aus den Gleichungen 7.6 und 7.15 sind hierfür verantwortlich.

[2] Dieser Bedingung wurde leicht abgeändert gegenüber der Originalliteratur ($\forall i : k(i) = pr(d,n)$), da ein einzelnes $i$, für das die Bedingung gilt, ausreicht, um die entsprechende Transferfunktion zu verwenden.

**Beispiel 7.4.1** Schlechte Erkennung der Abhängigkeiten
for( $i$=1; $i < UB$; $i$++)
{
  $a[2{\cdot}i]= \ldots;$
  $a[2{\cdot}i+3]= \ldots;$
}

$$pr(d,n) = 0, k(i) = \frac{2-2}{2} \cdot i + \frac{3-0}{2} = 1.5 \Longrightarrow p_n^d = 1 \tag{7.17}$$

Insgesamt scheint es sinnvoll zu sein, das Verfahren zu überarbeiten. Hierzu werden insbesondere die Transferfunktionen geeignet verändert.

Zuerst sollte auch der Fall von nicht affinen Indexfunktionen bei den Transferfunktionen berücksichtigt werden, da diese innerhalb von vielen Programmen auftreten können. Hierzu wird der entsprechende Iterationsdistanzwert in solchen Situationen auf $\perp$ gesetzt.

$f_1(i) = idx_1(i) = a_1 \cdot i + b_1$ und $f_2(i) = idx_2(i) = a_2 \cdot i + b_2$ sind affine monotone Funktionen, die in einem kartesischen Koordinatensystem zwei Geraden bilden, wenn $i$ über $f(i)$ aufgetragen ist. Die Gleichung 7.18 nach [HS81] ist genau dann erfüllt, wenn zwei Geraden parallel zueinander liegen. Dann gilt die Gleichung 7.19, da der Abstand zwischen den Geraden immer gleich groß ist.

$$a_1 - a_2 = 0 \tag{7.18}$$

$$\forall i : f_1(i) - f_2(i) = f_1(0) - f_2(0). \tag{7.19}$$

Damit beliebige ganzzahlige Zahlen $c \in \mathbb{Z}$ zu einer Induktionsvariablen hinzuaddiert werden dürfen, kann die Berechnung von $k(i)$, wie in der Gleichung 7.20 dargestellt, an diese neue Iterationsmöglichkeit angepasst werden. Im Prinzip erfolgt an dieser Stelle eine Verallgemeinerung von 1 zu $c$.

$$k(i) = \frac{a_1 - a_2}{c \cdot a_1} \cdot i + \frac{b_1 - b_2}{c \cdot a_1} \tag{7.20}$$

Weiterhin kann die Eigenschaft verwendet werden, dass die Tiefe der Registerpipeline physikalisch durch die Anzahl der Register beschränkt ist. Der ARM7TDMI-Prozessor stellt 16 Register zur Verfügung, weshalb die Tiefe der Registerpipeline durchaus ohne Einschränkungen bei der Optimierung auf 20 beschränkt werden kann. Werte größer als 20 würden starkes Spilling verursachen und somit die Effizienz der Optimierungstechnik Registerpipelining entscheidend verschlechtern. Die später angewendeten Verfahren brauchen dann nur noch die relevanten Daten (Iterationsdistanz $\leq$ 20) zu berechnen und können deswegen oft schneller arbeiten. Eine andere Möglichkeit wäre es, die Pipeline auf die Größe des On-Chip-Speichers (4096) zu beschränken, falls dieser verwendet wird.

Das $k(i)$ gibt die Iterationsdistanz für ein $i$ an, bei der zwei Speicherzugriffe die gleiche Speicheradresse adressieren. Als interessanter erweist sich aber die Frage, für welches $i(k)$ die Iterationsdistanz gleich $k$ ist. Hierzu wird die Gleichung 7.20 nach $i$ umgestellt und es ergibt sich die Gleichung 7.21.

$$i(k) = \frac{c \cdot a_1}{a_1 - a_2} \cdot k + \frac{b_2 - b_1}{a_1 - a_2} \tag{7.21}$$

Unter anderem bewirken diese Veränderungen auch, dass das Iterationsintervall der Induktionsvariablen nicht mehr bekannt sein muss, weil kein bestimmtes $i$ mehr in eine Gleichung eingesetzt wird. Es können später Schleifen der Form wie in dem Beispiel 7.4.2 dargestellt, durch die Umsetzung der beschriebenen Ideen und durch andere Transferfunktionen analysiert werden.

**Beispiel 7.4.2** Gestalt einer analysierbaren Schleife
// $j$ und $k$ sind ganze Zahlen oder Variablen
// $c$ muss eine bekannte ganze Zahl sein
for( $i = j$; $i \leqslant k$; $i$+=$c$) {
...
}

Die Präzision der $\delta$-Array-Datenflussanalyse kann verbessert werden, indem die Adressierungsart zusätzlich berücksichtigt wird. Beispielsweise kann der Speicherzugriff $a[2 \cdot i+1]$ auf Assemblerebene zu der Indexfunktion $f(i)=a \cdot i+b=8 \cdot i+4$ führen. Dies passiert dann, wenn die Elemente im Array $a$ eine Größe von 4-Bytes besitzen (z.B. generiert durch 'int $a$[10];'). Der ARM7TDMI-Prozessor besitzt nur die Möglichkeit, einen 8-Bit, 16-Bit oder einen 32-Bit Speicherplatz zu beschreiben oder zu lesen. Diese Adressierungsarten können direkt aus den jeweiligen Instruktionen abgeleitet werden. Die zu untersuchenden Indexfunktionen werden durch die Anzahl der adressierten Bytes (1,2 oder 4 Bytes) geteilt. Wenn die Werte innerhalb der Indexfunktionen weiterhin ganzzahlig sind und wenn die Anzahl der adressierten Bytes übereinstimmen, dann kann die Abhängigkeit im Beispiel 7.4.1 exakt erkannt werden. Die Grundidee ist eine Verschmelzung größerer Speicherbereiche (beispielsweise 4-Bytes) zu einer einzelnen untrennbaren Einheit. Bei unterschiedlichen Adressierungsarten besteht dagegen die Möglichkeit, beide Indexfunktionen durch die größere Anzahl an adressierten Bytes zu teilen.

Folgende Transferfunktionen unterstützen die in diesem Abschnitt beschriebenen Erweiterungen. Es wird aber nur der Fall genau betrachtet, bei dem die Werte der Indexfunktionen ganzzahlig bleiben. Die benötigten Daten für die Analyse werden, wie in dem Kapitel 6.4 beschrieben, berechnet.

### 7.4.1 Erhaltungsfunktionen

Nur die Berechnung des Wertes $p_n^d$ aus Kapitel 7.3.2 Unterpunkt 2 wird in diesem Abschnitt verändert. Die Erhaltungsfunktionen bleiben ansonsten gegenüber den ursprünglichen Transferfunktionen unverändert. Die Beispiele zu den Transferfunktionen werden hier und in dem Kapitel 7.5.2 aus Gründen der Verständlichkeit nicht auf der Assemblerebene betrachtet. Um den Wert $p_n^d$ zu berechnen, wird folgende Fallunterscheidung durchgeführt.

1. Die Gleichung 7.22 wird verwendet, wenn eine der folgenden Bedingungen gegeben ist:

   - Die Referenz $d$ oder $d'$ repräsentiert eine nicht affine Indexfunktion.
   - Die Inhalte von $a$ oder $b$ von $d$ oder $d'$ sind nicht ganzzahlig.
   - Die Induktionsvariablen der beiden Speicherzugriffe sind nicht gleich.

$$p_n^d = \bot \tag{7.22}$$

2. Die Geraden, die durch die Indexfunktionen gebildet werden, liegen parallel zueinander, weshalb der Abstand zwischen ihnen immer gleich ist. Die Iterationsdistanz bei der eine Überschneidung stattfindet sei $g$.

$$g = \{k(0)|k(0) \in \mathbb{N}_0\} \tag{7.23}$$

   (a) Wenn kein $g$ existiert, dann ändert $d'$ nie einen Wert der Referenz $d$. Dieser Fall tritt beispielsweise auf für $d' = a[2 \cdot i]$ und $d = a[2 \cdot i{+}1]$. Es gilt:

$$p_n^d = \top \tag{7.24}$$

   (b) Wenn $g = pr(d, n)$ ist, dann ändert $d'$ jeden Wert von $d$ innerhalb einer Iteration. Ein Beispiel hierfür wäre $pr(d, n) = 1$, $d = a[i]$ und $d' = a[i\text{-}1]$. Es gilt:

$$p_n^d = \bot \tag{7.25}$$

   (c) Sonst sei

$$p_n^d = g - 1 \tag{7.26}$$

   Diese Transferfunktion wird beispielsweise für $d = a[i{+}5]$ und $d' = a[i]$ verwendet, wo $p_n^d = 4$ ist.

3. Die Geraden besitzen einen Schnittpunkt. Das $g$ sei die kleinste Iterationsdistanz, bei der eine Überschneidung eintritt. Mit Hilfe der Gleichung 7.21 und 7.27 wird $g$ berechnet.

$$g = \min\{k|k \in [pr(d, n)..21], i(k) \in \mathbb{N}_0\} \tag{7.27}$$

   (a) Wenn kein kleinstes $g$ existiert, dann ändert $d'$ keinen Wert der Referenz $d$ innerhalb von 20 Iterationen. Es gilt:

$$p_n^d = 20 \tag{7.28}$$

(b) $g = pr(d, n)$: Dann verändert $d'$ einen Wert der Referenz $d$ sofort. Ein Beispiel hierfür wäre $pr(d, n) = 0$, $d = a[i]$ und $d' = a[2 \cdot i]$, wo $i(pr(d, n))$=0 und $g$=0 ist. Es gilt:

$$p_n^d = \perp \tag{7.29}$$

(c) Sonst sei

$$p_n^d = g - 1 \tag{7.30}$$

Beispielsweise für $pr(d, n) = 0$, $d = a[10 \cdot i]$ und $d' = a[i\text{-}5]$ ist $p_n^d = 4$, $g = 5$, $i = \frac{10}{9} \cdot 5 + \frac{-5}{9} = 5$ und $10 \cdot (i\text{-}5) = i\text{-}5$.

## 7.5 Weitere mögliche Erweiterungen

Es sind noch viele Ergänzungen bei der $\delta$-Array-Datenflussanalyse möglich. Folgende Erweiterungen wurden implementiert:

- Wenn die Iterationsgrenzen der Induktionsvariablen bekannt sind, kann die Qualität des Verfahrens verbessert werden.
- Register dürfen zu einer Indexfunktion addiert werden.
- Die Induktionsvariable darf auch um einen unbekannten Wert erhöht werden.
- Die Korrektheit des Verfahrens wird gewährleistet.

Weitere Möglichkeiten sind in der folgenden Aufzählung aufgeführt. Diese wurden aber bei der durchgeführten Implementierung nicht berücksichtigt:

- Andere Analysen neben einer must-Vorwärtsanalyse könnten durchgeführt werden.
- Gleitkommazahlen können unterstützt werden.
- Ein innerhalb der Schleife konstantes Register darf bei der Indexfunktion $f(i) = a \cdot i + b$ mit $a$ multipliziert werden.

In den nachfolgenden Unterkapiteln werden einige dieser möglichen Erweiterungen näher vorgestellt.

### 7.5.1 Bekannte Iterationsgrenzen

Bei bekannten Iterationsgrenzen einer Induktionsvariablen können die bisherigen Erhaltungsfunktionen leicht verändert werden. Die Gleichung 7.27 erhält hierzu die zusätzliche Einschränkung, dass das $i(k)$ in dem bekannten Iterationsintervall $I$ liegen muss, wie es in der Gleichung 7.31 dargestellt ist. Falls nur eine Iterationsgrenze bekannt ist, ist das Iterationsintervall in eine Richtung offen. Hierdurch kann die Qualität des Verfahrens verbessert werden.

$$g = \min\{k | k \in [pr(d, n)..21], i(k) \in \mathbb{N}_0 \wedge i(k) \in I\} \tag{7.31}$$

Für die Optimierungstechnik Registerpipelining ist es oft sinnvoll, nicht über die Grenzen der Induktionsvariablen zu laufen, sondern über die Iterationsdistanzen. Für viele andere Optimierungstechniken kann diese Vorgehensweise ebenfalls sinnvoll sein. Trotzdem kann es in manchen Fällen, wie in dem Beispiel 7.5.1, effizient sein, wenn nur das Iterationsintervall der Induktionsvariablen betrachtet wird. Dann könnte ein Aufbau der $\delta$-Array-Datenflussanalyse mit Hilfe der Gleichung 7.20 geschehen, oder beide Strategien könnten kombiniert eingesetzt werden. Das Verfahren ermittelt zuerst für die beiden Iterationsgrenzen die Iterationsdistanzen $q$ und $w$ mit der Gleichung 7.20. Wenn $q$ kleiner ist als $pr(d,n)$, dann wird $q$ gleich $pr(d,n)$ gesetzt und wenn $w$ größer ist als 21, dann erhält $w$ den Wert 21. Dann kann die Gleichung 7.32 anstelle der Gleichung 7.27 verwendet werden. Sinnvoll ist diese Vorgehensweise insbesondere dann, wenn die $\delta$-Array-Datenflussanalyse auch größere Iterationsdistanzen berechnet. Zusätzlich könnte auch noch die Eigenschaft der Monotonie der Werte $i(k)$ berücksichtigt werden.

$$g = \min\{k \mid k \in [q..w], i(k) \in \mathbb{N}_0\} \tag{7.32}$$

**Beispiel 7.5.1** Schleife mit einem kleinen Iterationsintervall
for( $i$=10; $i$ <15; $i$++) {
...
}

### 7.5.2 Register als Induktionsvariableninkrement

Ziel dieses Unterkapitels ist es, die Auswirkungen zu analysieren, wenn Register zu der Induktionsvariablen addiert bzw. subtrahiert werden. Der Inhalt des Registers muss ohne weitere Analysen als beliebig angenommen werden. Deshalb können potentielle Optimierungen nur eingeschränkt erkannt werden.

Nur die Berechnung des Wertes $p_n^d$ aus Kapitel 7.3.2 Unterpunkt 2 wird in diesem Abschnitt verändert. Die Transferfunktionen bleiben sonst unverändert.

1. Die Gleichung 7.33 wird verwendet, wenn eine der folgenden Bedingungen erfüllt ist:
   - Die Referenz $d$ oder $d'$ repräsentiert eine nicht affine Indexfunktion.
   - Die Inhalte von $a$ oder $b$ von $d$ oder $d'$ sind nicht ganzzahlig.
   - Die Induktionsvariablen der beiden Speicherzugriffe sind nicht gleich.

$$p_n^d = \perp \tag{7.33}$$

2. Wenn $pr(d,n) = 0$ ist, dann wird folgende Fallunterscheidung durchgeführt.
   (a) Wenn $a_1 = a_2$ und $b_1 \; mod \; a_1 \neq b_2 \; mod \; a_1$ ist, dann verändert $d'$ nie einen Wert der Referenz $d$. Beispielsweise tritt dies für $d = a[2{\cdot}i]$ und $d' = a[2{\cdot}i{+}1]$ auf. Es gilt:

$$p_n^d = \top \tag{7.34}$$

(b) Wenn $a_1 \neq a_2$, $b_1 = b_2$ und die Induktionsvariable immer ungleich 0 ist, dann ändert $d'$ nicht sofort einen Wert der Referenz $d$. Es gilt:

$$p_n^d = 0 \tag{7.35}$$

Dieser Fall tritt beispielsweise im Beispiel 7.5.2 auf, wenn die Auswirkungen von $a[i\text{-}1]$ auf $a[2{\cdot}i\text{-}1]$ analysiert werden. Eine Iterationsdistanz von 0 bedeutet, dass das $a[i\text{-}1]$ den Inhalt von $a[2{\cdot}i\text{-}1]$ nicht sofort überschreiben kann. Das für diese Diplomarbeit implementierte Verfahren Registerpipelining kann bei diesem Beispiel den zweiten Zugriff auf $a[2{\cdot}i\text{-}1]$ einsparen.

**Beispiel 7.5.2** Register als Induktionsvariableninkrement

```
...
for( i=10; i <20; i+=r)
{
  a[i-1]=a[2·i-1]+1;
  j=a[2·i-1];
   ...
}
...
```

(c) Sonst sei

$$p_n^d = \perp \tag{7.36}$$

3. Sonst wird folgende Fallunterscheidung durchgeführt.

(a) Wenn $a_1 = a_2$, $b_1 = b_2$ und wenn die Induktionsvariable nicht um 0 innerhalb einer Iteration erhöht wird, dann ändert $d'$ nie einen Wert der Referenz $d$ sofort. Somit sei

$$p_n^d = 0 \tag{7.37}$$

Ein Problem bei diesem Punkt aus der Aufzählung ist sicherzustellen, dass sich die Induktionsvariable um einen Wert, der ungleich 0 ist, verändern muss.

(b) Sonst sei

$$p_n^d = \perp \tag{7.38}$$

### 7.5.3 Addition von Registern zu den Indexfunktionen

Bei $f(i) = a \cdot i + b$ soll das $b$ auch noch Register enthalten dürfen, die innerhalb der Schleife konstant sind. Dies wird insbesondere deswegen benötigt, um die genaue Position eines Speicherzugriffs innerhalb des Speichers festzustellen zu können.

Weiterhin können hierdurch auch mehrdimensionale Arrays einfach analysiert werden. Im Beispiel 7.5.3 könnte der Zugriff auf $a[i{+}1][j]$ eventuell durch einen früheren Zugriff auf $a[i][j]$ ersetzt werden, wenn das $j$ innerhalb der Schleife konstant ist. Das $j$ fließt dann als Konstante in dem $b$ der Indexfunktion ein.

**Beispiel 7.5.3** Mehrdimensionale Arrays

for ( $i$=0; $i$<10; $i$++)
{
    ...
    $a[i][j] = a[i{+}1][j]$;
    ...
}

### 7.5.4 Sicherstellung der Korrektheit des Verfahrens

Die Erhaltungsfunktionen in dem Kapitel 7.3.2 unter Punkt 1 der Aufzählung arbeiten im allgemeinen nicht korrekt. Exemplarisch kann ein Fehler in dem Beispiel 6.2.1 auftreten, wo unterschiedliche Arrays in dem selben Speicherbereich abgelegt werden könnten. Für diesen Fall muss die Bedingung für die Transferfunktion, wie folgt abgeändert werden:

1. Wenn $d$ und $d'$ voneinander unabhängig sind, indem die Arrays sich nicht überschneiden (z.B. durch unterschiedliche Speicherorte), oder wenn $d'$ nur ein Gebrauch ist, dann bleibt die Gültigkeit der Iterationsdistanz erhalten.

$$p_n^d = \top \tag{7.39}$$

### 7.5.5 May- und Rückwärts-Analysen

Dieses Unterkapitel beschreibt nach [Fra99], wie die $\delta$-Array-Datenflussanalyse für verschiedene Optimierungen eingesetzt werden kann. Hierzu gehören beispielsweise Techniken, wie Redundant Store Elimination oder auch kontrolliertes Loop Unrolling. Diese anderen Optimierungen benötigen aber auch teilweise andere Analysen als die $\delta$-available-values Analyse. Durch einige Änderungen kann das Verfahren must- und may-Probleme behandeln und auch Vorwärts- und Rückwärtsanalysen durchführen.

Fur eine Rückwärtsanalyse werden die Kanten der Graphen $G'$ und $G''$ umgedreht und anstelle von $k(i)$ und $i(k)$ werden bei den Transferfunktionen $-k(i)$ und $-i(k)$ verwendet.

Um eine may-Analyse durchzuführen, muss die $\delta$-Array-Datenflussanalyse andere Transferfunktionen verwenden. Der Wert einer Referenz bleibt solange gültig, bis er mit Sicherheit nicht mehr gültig ist. Deswegen können die Transferfunktionen fast nur Werte außer $\bot$ und $\top$ enthalten, wenn die durch die Indexfunktionen gebildeten Geraden parallel liegen. Neben weiteren Änderungen müssen auch noch die Datenflussgleichungen in dem Kapitel 7.6 angepasst werden, indem die Funktion 'min()' durch 'max()' ersetzt wird.

## 7.6 Iteratives Lösungsverfahren

Mit Hilfe eines Datenfluss-Gleichungssystems und den Transferfunktionen kann der Datenfluss innerhalb einer Schleife ermittelt werden. Die hier beschriebene Vorgehensweise stammt aus [ED93][3].

Es wird für jeden Knoten $n$ aus $m{+}1$ Knoten ein Vektor $IN[n] = (x_1, ..., x_m)$ und ein Vektor $OUT[n] = (x_1, ..., x_m)$ angelegt. Sie enthalten die Iterationsdistanzen für jede der $m$ Referenzen beim Erreichen bzw. beim Verlassen des Knotens $n$. In einem Durchlauf des iterativen Lösungsverfahrens wird jeder Knoten einmal besucht und jedes Vektorelement wird höchstens einmal verändert. Die Knoten werden in einer besonderen Reihenfolge durchlaufen, damit das Verfahren möglichst schnell arbeitet. Zuerst wird der Knoten $m{+}1$ bearbeitet. Die Reihenfolge der restlichen Knoten ergibt sich aus einer DFS-Postorder-Suche in dem Graphen G$''$ beginnend mit dem Knoten $m + 1$. Initialisiert werden die einzelnen Vektoren wie folgt:

$$IN[n,d] = \begin{cases} \perp & \text{, falls } n{=}m{+}1 \\ \min(OUT[m,d] | m \in pred(n)) & \text{, sonst} \end{cases} \tag{7.40}$$

$$OUT[n,d] = \begin{cases} \top & \text{, falls } d \text{ in } n \text{ liegt} \\ IN[n,d] & \text{, sonst} \end{cases} \tag{7.41}$$

Das $IN[n,d] = x_d$ bezeichnet die Iterationsdistanz, mit der die Referenz $d$ den Knoten $n$ erreicht. Die Menge der Vorgängerknoten von dem Knoten $n$ in dem Graphen G$''$ ist $pred(n)$. Nach der Initialisierung arbeitet ein Algorithmus wie folgt weiter, bis keine Veränderungen innerhalb der Vektoren mehr stattfinden:

$$IN[n,d] = \min(OUT[m,d] | m \in pred(n)) \tag{7.42}$$

$$OUT[n,d] = f_n^d(IN[n,d]) \tag{7.43}$$

Abschließend stehen die Iterationsdistanzen, in denen die Speicherinhalte nicht verändert werden, in den entsprechenden Vektoren.

## 7.7 Eigenschaften

Die angepasste $\delta$-Array-Datenflussanalyse ermöglicht die Analyse von Abhängigkeiten zwischen einzelnen Speicherzugriffen innerhalb von Schleifen.

Die $m^2{+}m$ Transferfunktionen können in Zeit $O(m^2)$ berechnet werden. Die ursprünglichen Transferfunktionen des Verfahrens benötigten hingegen bei einer

[3] Das Verfahren wurde an die bisher beschriebenen Veränderungen angepasst. Berücksichtigt wurden insbesondere die geänderten Datenstrukturen.

naiven Implementierung der Gleichung 7.15 eventuell die Zeit $O(m^2 \cdot UB + y^2)$. Das $y$ entspricht der Anzahl der Instruktionen innerhalb der Schleife. Auf eine genaue und wahrscheinlich auch komplexe Laufzeitanalyse des iterativen Lösungsverfahrens wird verzichtet. In der Praxis hat sich gezeigt, dass die Laufzeit oft bei $O(m^2)$ liegt, da meistens 4 Durchläufe des Lösungsverfahrens ausreichen. Das ursprüngliche Lösungsverfahren der $\delta$-Array-Datenflussanalyse benötigt hingegen $O(y^2)$ Operationen. Diese Vorteile bei der Laufzeit des Verfahrens entstehen durch die Verwendung der Graphen $G'$ und $G''$. Die Konstruktion der Graphen kann mit $O(z^2 \cdot m)$ Operationen geschehen. Das $z$ entspricht der Anzahl der Knoten in dem Graphen $G$. Somit dominiert die Zeit zur Generierung der Graphen die Gesamtlaufzeit der angepassten Array-Datenflussanalyse.

Die angepasste $\delta$-Array-Datenflussanalyse weist insgesamt folgende Möglichkeiten auf:

- Schleifenform
  - Multiple-entry/multiple-exit Schleifen können analysiert werden.
  - Schleifen der Form wie im Beispiel 7.7.1 können untersucht werden. Die Iterationsgrenzen und die Schrittweite der Induktionsvariablen müssen nicht bekannt sein. Wenn sie bekannt sind, kann das Verfahren eine bessere Analyse der Schleife durchführen.

    **Beispiel 7.7.1** Gestalt einer analysierbaren Schleife
    // $j$,$k$ und $c$ sind ganze Zahlen oder Variablen
    for( $i = j$; $i \lesseqgtr k$; $i$+=$c$) {
    ...
    }

  - Innere als auch nicht innere Schleifen können analysiert werden.
  - Eine Analyse über die Schleifengrenzen hinaus wird durchgeführt. Beispielsweise kann die angepasste $\delta$-Array-Datenflussanalyse hierdurch auch genug Informationen liefern, um im Beispiel 7.7.2 den inneren Zugriff auf $a[i]$ mit Hilfe von Registerpipelining zu optimieren. Dieser Speicherlesezugriff könnte aber auch optimiert werden, indem eine Verschiebung von schleifeninvariantem Code stattfindet.

    **Beispiel 7.7.2** Analysemöglichkeiten über Schleifengrenzen hinaus

    ```
    for( i=1; i<10; i++)
    {
      a[i] = t;
      for( j=1; j<10; j++)
      {
        r+=a[i];
      }
    }
    ```

- Indexfunktionen
  - Nicht affine Indexfunktionen können von der Analyse zumindest behandelt werden.
  - Mehrdimensionale Arrays mit affinen Indexfunktionen können analysiert werden.
- Induktionsvariablen
  - Unterschiedliche Induktionsvariablen innerhalb von zwei Indexfunktionen werden berücksichtigt.
- Präzision
  - Da die Adressierungsart mit berücksichtigt wird, konnte die Präzision des Verfahrens entscheidend verbessert werden. Ohne die Verwendung dieser Informationen kann eine Analyse bei mehreren Adressierungsmöglichkeiten kaum fehlerfrei und präzise arbeiten. Beispiel 7.7.3 veranschaulicht diesen Zusammenhang. Ohne Berücksichtigung der Adressierungsart kann der zweite Speicherzugriff auf $a[i]$ mit Hilfe des ersten Zugriffs ersetzt werden, obwohl das $a[i\text{-}2]$ den Inhalt von $a[i]$ überschreibt. Auch die wesentlich verbesserten Transferfunktionen erhöhen die Präzision der angepassten $\delta$-Array-Datenflussanalyse.

**Beispiel 7.7.3** Unterschiedliche Adressierungsarten

```
__int8 a[100];              // 8-Bit Array
for( i=1; i <10; i++)
{
  ... =a[i];
 ((__int32)a[i-2])= ... ; // 32-Bit Schreibzugriff
  ... =a[i];
}
```

Im wesentlichen existieren folgende Einschränkungen bei der angepassten $\delta$-Array-Datenflussanalyse:

- Schleifenform
  - Wenn innerhalb einer Schleife weitere Schleifen existieren, verliert die vorgenommene Analyse an Qualität. Diese Situation kann verbessert werden, indem Informationen zwischen den einzelnen Schleifenanalysen transportiert werden.
- Indexfunktionen
  - Nur affine Indexfunktionen können genau analysiert werden. Eine Ausweitung auf andere Funktionsklassen könnte relativ einfach

möglich sein. Hierzu gehören beispielsweise monotone, stetige oder polynomielle Funktionen.

  - Mehrdimensionale Arrays werden nur eingeschränkt unterstützt, da ihre Indexfunktionen oft nicht affin sind. Für das Verfahren Registerpipelining ist der nicht affine Fall aber auch eher uninteressant.

- Induktionsvariablen

  - Unterschiedliche Induktionsvariablen bei zwei Indexfunktionen können nur schlecht untereinander analysiert werden. Hierfür müsste die $\delta$-Array-Datenflussanalyse von einer Induktionsvariablen abstrahieren und stattdessen den Startwert, den Endwert und die Schrittweite der Induktionsvariablen verwenden. Durch die Konstruktion der Indexfunktionen wird dieses aber bei der durchgeführten Implementierung der Optimierungstechnik berücksichtigt.

## 7.8 Ein Beispiel

An dem Beispiel 7.8.1 wird exemplarisch die $\delta$-Array-Datenflussanalyse durchgeführt. In [ED93] und [Fra99] untersucht die Datenflussanalyse explizit ein C-Programm. Diese Diplomarbeit beschäftigt sich jedoch mit der Analyse eines Assemblerprogramms, weshalb dies auch bei dem Beispiel berücksichtigt wird. Das Beispielprogramm 7.8.1 ist aus Gründen der Verständlichkeit als ein C-Programm angegeben.

**Beispiel 7.8.1** C-Programm

```
int a[100], b[100];
for( i=10; i<e; i++)
{
    r = a[i];            (n=1)
    a[i+1]=r·2;          (n=2)
    if (r>x) {
        r=b[2·i-4];      (n=3)
        a[i] = r;        (n=4)
    }
    r = a[2·i+98];       (n=5)
    b[2·i] = r;          (n=6)
}
```

Die Tabelle 7.1 stellt die Indexfunktionen des Beispiels 7.8.1 mit den dazu gehörigen Array-Referenzen auf Assemblerebene dar. Das virtuelle Register V2 entspricht der Induktionsvariablen $i$. Die Felder wurden bei diesem Beispiel relativ zum Stackpointer SP angelegt. Interessant ist, dass die Arrays $a$ und $b$ direkt hintereinander im Speicher abgelegt wurden und dass auf Assemblerebene beide Arrays fast gleich adressiert werden.

| $n$ | C-Ebene | Assemblerebene |
|---|---|---|
| 1 | $a[i]$ | 4·V2+SP |
| 2 | $a[i+1]$ | 4·V2+4+SP |
| 3 | $b[2 \cdot i\text{-}4]$ | 8·V2+384+SP |
| 4 | $a[i]$ | 4·V2+SP |
| 5 | $a[2 \cdot i+98]$ | 8·V2+392+SP |
| 6 | $b[2 \cdot i]$ | 8·V2+400+SP |

Tabelle 7.1: Indexfunktionen mit Array-Referenzen auf verschiedenen Ebenen

Zuerst werden die Transferfunktionen, wie in Kapitel 7.4 beschrieben, bestimmt. Die Tabelle 7.2 zeigt die Transferfunktionen für das Beispiel 7.8.1. Zur Kennzeichnung der verwendeten Transferfunktion wird eine Abkürzung mit der zugehörigen Aufzählungsnummer verwendet (Erzeugungsfunktionen EZ, Erhaltungsfunktionen EH, Iterationsübergangsfunktionen IT). Das Verfahren setzt die Iterationsübergangsfunktionen nur für $n$=7 und die Erzeugungsfunktionen nur auf einer der Diagonalen der Matrix ein.

<table>
<tr><th>$n$</th><th>$a[i]$</th><th>$a[i+1]$</th><th>$b[2 \cdot i\text{-}4]$</th><th>$a[i]$</th><th>$a[2 \cdot i+98]$</th><th>$b[2 \cdot i]$</th></tr>
<tr><td>1</td><td>max($x$,0 )<br>EZ<br>Gl.7.5</td><td colspan="5">min($x$,$\top$ )<br>EH-1<br>Gleichung 7.6&7.39</td></tr>
<tr><td>2</td><td>min($x$,$\top$ )<br>EH 2-2a<br>G.7.6&7.24</td><td>max($x$,0 )<br>EZ<br>Gl.7.5</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td><td>min($x$,$\top$ )<br>EH 2-2a<br>G.7.6&7.24</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td></tr>
<tr><td>3</td><td colspan="2">min($x$,$\top$ )<br>EH-1<br>Gleichung 7.6&7.39</td><td>max($x$,0 )<br>EZ<br>Gl.7.5</td><td colspan="3">min($x$,$\top$ )<br>EH-1<br>Gleichung 7.6&7.39</td></tr>
<tr><td>4</td><td>min($x$,$\bot$ )<br>EH 2-2b<br>G.7.6&7.25</td><td>min($x$,0 )<br>EH 2-2c<br>G.7.6&7.26</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td><td>max($x$,0 )<br>EZ<br>Gl.7.5</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td></tr>
<tr><td>5</td><td colspan="4">min($x$,$\top$ )<br>EH-1<br>Gleichung 7.6&7.39</td><td>max($x$,0 )<br>EZ<br>Gl.7.5</td><td>min($x$,$\top$ )<br>EH-1<br>G.7.6&7.39</td></tr>
<tr><td>6</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td><td>min($x$,$\top$ )<br>EH 2-2a<br>G.7.6&7.24</td><td>min($x$,20 )<br>EH 2-3a<br>G.7.6&7.28</td><td>min($x$,$\top$ )<br>EH 2-2a<br>G.7.6&7.24</td><td>max($x$,0 )<br>EZ<br>Gl.7.5</td></tr>
<tr><td>7</td><td colspan="6">$x$++<br>IT<br>Gleichung 7.16</td></tr>
</table>

Tabelle 7.2: Transferfunktionen des Beispiels 7.8.1

Exemplarisch wird die Transferfunktion für $n$=2 und $d = b[2 \cdot i\text{-}4]$ berechnet. Die Gleichungen 7.6, 7.28 und 7.31 ermitteln bei diesem Beispiel die Transferfunktion. Die Tabelle 7.3 zeigt die Berechnungen, die mit der Gleichung 7.31 durchgeführt werden. Jedes $i(k)$ ist kleiner als die untere Iterationsgrenze 10. Deswegen existiert kein kleinstes $g$ und somit wird die Gleichung 7.28 verwendet.

Wenn das Verfahren Registerpipelining auch für größere Iterationswerte verwendet werden soll, dann müssten auch für höhere Werte von $k$ die Werte von $i(k)$ berechnet werden. Für $k = 53$ wird der Wert 11 berechnet, denn es gilt

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $i(k)$ | -93 | -91 | -89 | -87 | -85 | -83 | -81 | -79 | -77 | -75 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| -73 | -71 | -69 | -67 | -65 | -63 | -61 | -59 | -57 | -55 | -53 |

Tabelle 7.3: Beispielhafte Berechnung der Werte $i(k)$

$2 \cdot (11 - 53) + 96 = 1 \cdot 11 + 1$ (Gleichung 7.11). Für diesen Fall lautet die Transferfunktion 'min($x$,52)'.

Der Ablauf des iterativen Lösungsverfahrens aus Kapitel 7.6 ist in der Tabelle 7.4 dargestellt. Zuerst wird eine Initialisierungsphase durchgeführt, und anschließend wird die Lösung iterativ bestimmt. Drei Iterationen werden durchgeführt, wobei die einzelnen Vektoren aus der zweiten und dritten Iteration übereinstimmen. Somit enthält die Tabelle 7.4 eine Iteration zu wenig.

Drei Optierungsmöglichkeiten für das Verfahren Registerpipelining können gefunden werden. Diese Möglichkeiten sind in der folgenden Aufzählung aufgeführt.

- Der Inhalt von $a[i{+}1]$ kann eine Iteration später bei $a[i]$ verwendet werden. Denn das Element $IN[1, a[i{+}1]]$ ist gleich 1 und die Iterationsdistanz zwischen ihnen beträgt ebenfalls 1.

- Der Inhalt von $a[2{\cdot}i{+}98]$ kann eine Iteration später bei $b[2{\cdot}i]$ verwendet werden, weil das Element $IN[6, b[2{\cdot}i]]$=20 größer als die Iterationsdistanz von 1 ist. Diese Optimierung ist aber auch nur deswegen möglich, weil die Arrays hintereinander im Speicher liegen.

- Der Inhalt von $b[2{\cdot}i]$ kann eventuell zwei Iterationsdistanzen später bei $b[2{\cdot}i$-$4]$ verwendet werden, da das Element $IN[4, b[2{\cdot}i]]$=20 größer als die dazwischen liegende Iterationsdistanz von 2 ist. Die Möglichkeit der Wiederverwendung von Speicherinhalten ist aber von der Bedingung der umschließenden 'IF'-Anweisung abhängig.

## 7.9 Zusammenfassung

Die $\delta$-Array-Datenflussanalyse ermöglicht innerhalb von Schleifen die Analyse von Abhängigkeiten zwischen einzelnen Speicherzugriffen. Das Verfahren wurde in dieser Arbeit erweitert und an die besonderen Gegebenheiten angepasst.

Die angepasste $\delta$-Array-Datenflussanalyse unterstützt den ARM7TDMI-Prozessor mit seinen Adressierungsmöglichkeiten und die Implementierung des Verfahrens Registerpipelining im Backend des Compilers. Die Datenflussanalyse berechnet nur die wirklich notwendigen Informationen (Iterationsdistanz$\leq$20) und kann hierdurch effizient arbeiten.

Die in Kapitel 7.7 aufgeführten Eigenschaften und die Integration der Analyse in dem Compiler 'arm12cc', wie sie in Kapitel 6.4 beschrieben ist, ermöglicht eine effektive Analyse. Das Verfahren Registerpipelining kann eine Verbesserung des Quellcodes relativ oft und präzise durchführen.

| Durchlauf | | Vektor | $a[i]$ | $a[i{+}1]$ | $b[2{\cdot}i\text{-}4]$ | $a[i]$ | $a[2{\cdot}i{+}98]$ | $b[2{\cdot}i]$ |
|---|---|---|---|---|---|---|---|---|
| Initiali-sierung | $n{=}1$ | IN | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |
| | | OUT | $\top$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |
| | $n{=}2$ | IN | $\top$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |
| | | OUT | $\top$ | $\top$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |
| | $n{=}3$ | IN | $\top$ | $\top$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |
| | | OUT | $\top$ | $\top$ | $\top$ | $\perp$ | $\perp$ | $\perp$ |
| | $n{=}4$ | IN | $\top$ | $\top$ | $\top$ | $\perp$ | $\perp$ | $\perp$ |
| | | OUT | $\top$ | $\top$ | $\top$ | $\top$ | $\perp$ | $\perp$ |
| | $n{=}5$ | IN | $\top$ | $\top$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |
| | | OUT | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\perp$ |
| | $n{=}6$ | IN | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\perp$ |
| | | OUT | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\top$ |
| | $n{=}7$ | IN | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\top$ |
| | | OUT | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\top$ |
| 1.Iteration | $n{=}1$ | IN | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\top$ |
| | | OUT | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\top$ |
| | $n{=}2$ | IN | $\top$ | $\top$ | $\perp$ | $\perp$ | $\top$ | $\top$ |
| | | OUT | $\top$ | $\top$ | $\perp$ | $\perp$ | 20 | 20 |
| | $n{=}3$ | IN | $\top$ | $\top$ | $\perp$ | $\perp$ | 20 | 20 |
| | | OUT | $\top$ | $\top$ | 0 | $\perp$ | 20 | 20 |
| | $n{=}4$ | IN | $\top$ | $\top$ | 0 | $\perp$ | 20 | 20 |
| | | OUT | $\perp$ | 0 | 0 | 0 | 20 | 20 |
| | $n{=}5$ | IN | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | | OUT | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | $n{=}6$ | IN | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | | OUT | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | $n{=}7$ | IN | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | | OUT | $\perp$ | 1 | $\perp$ | $\perp$ | 21 | 21 |
| 2.Iteration | $n{=}1$ | IN | $\perp$ | **1** | $\perp$ | $\perp$ | 21 | 21 |
| | | OUT | 0 | 1 | $\perp$ | $\perp$ | 21 | 21 |
| | $n{=}2$ | IN | 0 | 1 | $\perp$ | $\perp$ | 21 | 21 |
| | | OUT | 0 | 1 | $\perp$ | $\perp$ | 20 | 20 |
| | $n{=}3$ | IN | 0 | 1 | $\perp$ | $\perp$ | 20 | 20 |
| | | OUT | 0 | 1 | 0 | $\perp$ | 20 | 20 |
| | $n{=}4$ | IN | 0 | 1 | 0 | $\perp$ | 20 | **20** |
| | | OUT | $\perp$ | 0 | 0 | 0 | 20 | 20 |
| | $n{=}5$ | IN | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | | OUT | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | $n{=}6$ | IN | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | **20** |
| | | OUT | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | $n{=}7$ | IN | $\perp$ | 0 | $\perp$ | $\perp$ | 20 | 20 |
| | | OUT | $\perp$ | 1 | $\perp$ | $\perp$ | 21 | 21 |

Tabelle 7.4: Beispielhafter Ablauf des iterativen Lösungsverfahrens

Spezielle Informationen des Backends über die Positionen der einzelnen Arrays werden verwendet, um die $\delta$-Array-Datenflussanalyse zu verbessern, wie in dem Beispiel 7.8.1. Der Begriff Array beinhaltet normalerweise auch, dass die Datentypen innerhalb des Arrays gleich sind. Diese Eigenschaft wird jedoch von der Datenflussanalyse nicht benötigt. Das Verfahren kann alle Speicherzugriffe untersuchen. Insbesondere können auch Pointerzugriffe optimiert werden.

# Kapitel 8

# Untersuchungen und empirische Resultate

Dieses Kapitel untersucht die Programme, die im Rahmen dieser Diplomarbeit entwickelt worden sind. Zuerst wird überprüft, ob das Energiemodell aus Kapitel 5.2 den Energieverbrauch auch in der Praxis gut approximieren kann. Dann wird untersucht, welche Verbesserungen die Optimierungstechnik Registerpipelining erreichen kann.

## 8.1 Testverfahren

Der Referenz-Compiler 'tcc' generiert für die Ergebnisse der Versuche die Programme mit den Optionen '-O2' und '-Otime'.

Der Compiler 'arm12cc' verwendet für die Generierung der Programme die Option '-s energy', um eine Energieoptimierung im Backend durchzuführen. Auf den Zwischencode wendet der Compiler die folgenden Optimierungstechniken an:

- Constant Propagation
- Constant Folding
- Copy Propagation
- Common Subexpression Elimination
- Dead Code Elimination
- Jump Optimization
- (Loop Invariant Code Motion)

Die Optimierungstechnik Loop Invariant Code Motion verwendet der Compiler nur dann, wenn der generierte Assemblercode ohne die Optimierungstechnik Registerpipelining auch wirklich besser wird. Falls der Compiler die Optimierungstechnik für manche Ergebnisse nicht eingesetzt hat, wird dieses an geeigneter Stelle ausdrücklich erwähnt. Dieses betrifft meistens Programme mit hohem Registerdruck. Auf die zur Verfügung stehende Optimierungstechnik Induction Variable Elimination wurde hingegen grundsätzlich verzichtet, aufgrund der schlechten Ergebnisse, die ohne die Optimierungstechnik Registerpipelining erreicht wurden.

Während der Durchführung der Optimierungstechnik Registerpipelining wird im Backend eine Copy Propagation und eine Useless Code Elimination vor dem Aufruf der Registerallokation durchgeführt. Diese beiden Optimierungstechniken wurden im Rahmen dieser Diplomarbeit implementiert, um das Verfahren Registerpipelining nachträglich zu optimieren.

Das Verfahren Registerpipelining erzeugt bei den durchgeführten Versuchen keinen Spillcode, da dadurch die Programmoptimierungen an Effizienz verlieren würden.

Wenn Veränderungen am Versuchsaufbau durchgeführt wurden, wird dies explizit an den entsprechenden Stellen vermerkt. Ansonsten bleibt die Konfiguration der Compilers und des Verfahren Registerpipelining unverändert.

## 8.2 Validierung des Bewertungsverfahrens

Für die Bewertung der Ergebnisse des Trace-Analyzers wurden Messungen am Evaluationboard von ATMEL (siehe Kapitel 3.2) durchgeführt. Eine geeignete Arbeitsumgebung stand hierfür zur Verfügung. Ein digitales Amperemeter misst den Strom, der durch den ATMEL-AT91M40400-Chip oder durch den externen RAM-Speicher fließt. Der Strom, der beim ATMEL-AT91M40400-Chip gemessen wird, beinhaltet den Strom des ARM7TDMI-Prozessors, der Peripheriesteuerung und des On-Chip-Speichers. Damit die erhobenen Daten eine möglichst hohe Genauigkeit besitzen, werden die zu messenden Programme innerhalb einer kurzen Schleife mehrmals durchlaufen. Diese Vorgehensweise sichert die Stabilität der Messwerte.

Für die Validierung des Verfahrens wurden zwei Programme untersucht, die die wichtigsten Instruktionen enthalten. Hierzu gehören verschiedene Move-, Shift-, Sprung-, Load- und Store-Operationen und unterschiedliche arithmetische und logische Operationen. Die Tabelle 8.1 zeigt die ermittelten Werte.

| Programmtyp | Speicherort | | Strom (mA) | | |
|---|---|---|---|---|---|
| | Instruktionen | Daten | gemessen | simuliert | Abweichung |
| ohne Datenspeicherzugriffe | On-Chip | On-Chip | 52.9 | 48.5 | -11.2% |
| | | extern | | | |
| | extern | On-Chip | 295.4 | 296.9 | 0.1% |
| | | extern | | | |
| mit Datenspeicherzugriffe | On-Chip | On-Chip | 55.0 | 52.6 | 6.9% |
| | | extern | 202.5 | 205.8 | -2.2% |
| | extern | On-Chip | 246.6 | 248.8 | 0% |
| | | extern | 281.6 | 291.7 | 3.3% |

Tabelle 8.1: Gegenüberstellung der Werte für den Stromverbrauch

Durchschnittlich weichen die Werte des Trace-Analyzers um 4% ab. Zur Berechnung exakter Werte fehlen noch weitere Informationen, die in der nachfolgenden Aufzählung aufgeführt sind:

- Instruktionskosten in Abhängigkeit der unterschiedlichen Speicher
- Speicherzugriffskosten in Abhängigkeit der Operationen
- Berücksichtigung von unterschiedlichen Operanden
- Zustandswechselkosten (siehe Kapitel 5.2)

Das Verfahren Registerpipelining ändert nur einen Teil eines Programms ab. Deswegen werden viele Ungenauigkeiten bei der Energieberechnung des Trace-Analyzers sowohl im optimierten als auch im nicht optimierten Programm auftreten.

Insgesamt reicht somit die Genauigkeit der Energiewerte aus, um die Optimierungstechnik Registerpipelining mit Hilfe des Trace-Analyzers zu beurteilen.

## 8.3 Registerpipelining

Diese Kapitel untersucht und bewertet die Verbesserungen, die mit der Optimierungstechnik Registerpipelining erreicht wurden. Hierzu wird das Verfahren auf einige Programme angewendet, um die unterschiedlichen Verhaltensweisen der Optimierungstechnik zu charakterisieren.

### 8.3.1 Beispielprogramme und Benchmarks

Interessant sind die Auswirkungen der Optimierungstechnik Registerpipelining auf einige Beispielprogramme und Benchmarks. Die Benchmarks zeigen das Verhalten der Optimierungstechnik innerhalb komplexerer Programme. Zudem können einige Eigenschaften der Optimierungstechnik herausgearbeitet werden. Folgende Programme werden betrachtet:

- Positives Beispiel
  Das Beispiel 8.3.1 enthält ein konstruiertes Programm, das besonders gut bzgl. Energie, Taktzyklen und Speicherplatz optimiert werden kann.

  **Beispiel 8.3.1** Positives Beispiel

  ```
  int main() {
      int i,b,c,d,a[1500];
      d=0;
      for(i=1; i <200; i++) {
          b=a[i];
          c=b+5;
          d+=a[(i-1)·(b+1)+(b-1)·(c+1)-(c·b)-(i·b)+2+c];
      }
      return(d);
  }
  ```

Dieses Programm soll zeigen, dass der Einsatz von Registerpipelining für einige Programme sehr lohnenswert ist. Das Beispielprogramm verdeutlicht auch die Qualität der verwendeten Analyse für die Bestimmung der Grenzwerte, der Induktionsvariablen und der Indexfunktionen. Die komplexe Indexfunktion '$(i\text{-}1)\cdot(b+1)+(b\text{-}1)\cdot(c+1)\text{-}(c\cdot b)\text{-}(i\cdot b)+2+c$' kann analysiert und in diesem Fall zu '$i$' vereinfacht werden. Mit der Optimierung des zweiten Speicherzugriffes in dem Beispiel entfällt die Indexfunktion, weshalb sehr gute Resultate erreicht werden können.

- Lattice-Filter
  Die Optimierungstechnik Registerpipelining verbessert ein Programm, das einen Lattice-Filter enthält. Das Programm Lattice-Filter wurde ohne die Optimierungstechnik Loop Invariant Code Motion übersetzt, da sie sich hier negativ auswirkt.

- biquad_N_sections
  Aus der DSP-Stone Benchmark Suite kann das Programm biquad_N_sections, das den Datentyp int verwendet, mit der Optimierungstechnik Registerpipelining optimiert werden. Das N entspricht der Anzahl der Schleifeniterationen und es erhielt für die Versuche mit dem Programm biquad_N_sections den Wert 10.

  Bemerkenswert bei diesem Benchmark ist, dass die Optimierungstechnik Registerpipelining hier Pointer- statt Arrayzugriffe optimiert.

- Kernel 1,5,7,11 und 12
  Die Livermore Loops Benchmark-Suite enthält einige Kernels die überprüft werden. Ein Problem bei diesem Benchmark ist, dass er Gleitkommazahlen enthält und diese vom Compiler 'arm12cc' noch nicht unterstützt werden. Deshalb erhielten die verwendeten Variablen für die Versuche in dieser Diplomarbeit den Datentyp integer. Folgende Kernels konnte die Optimierungstechnik Registerpipelining verändern:

  - Kernel 1 (hydro fragment)
  - Kernel 5 (tri-diagonal elimination, below diagonal)
  - Kernel 7 (equation of state fragment)
    Dieses Programm wurde ohne die Optimierungstechnik Loop Invariant Code Motion übersetzt, da sie sich hier negativ auswirkt.
  - Kernel 11 (first sum)
  - Kernel 12 (first difference)

Die Tabellen 8.2, 8.3 und 8.4 zeigen für die aufgeführten Programme den Prozentsatz, um dem sich der Energieverbrauch, die Taktzyklenanzahl und die Leistungsaufnahme bei einer Energieoptimierung ändert. Die Resultate der Benchmarks sind unten in den Tabellen dargestellt, ab dem Programm biquad_N_sections.

| Programm/Benchmark | Speicherort (*P*=Programm, *D*=Daten) | | | |
|---|---|---|---|---|
| | *P* und *D* extern | *P* On-Chip *D* extern | *P* extern *D* On-Chip | *P* und *D* On-Chip |
| Positives Beispiel | -68,4% | -66,5% | -71,1% | -76,3% |
| Lattice-Filter | -4,0% | -17,7% | nicht Messbar | |
| biquad_N_sections | -0,8% | -6,0% | unverändert | -0,8% |
| Livermore Kernel 1 | -17,5% | -19,4% | -16,0% | -15,2% |
| Livermore Kernel 5 | -20,6% | -20,5% | -19,8% | -17,1% |
| Livermore Kernel 7 | -6,8% | -19,5% | -1,5% | -6,8% |
| Livermore Kernel 11 | -25,2% | -26,5% | -24,0% | -24,0% |
| Livermore Kernel 12 | -10,3% | -21,6% | unverändert | -9,9% |

Tabelle 8.2: Änderung der Energie bei einer Energieoptimierung

| Programm/Benchmark | Speicherort (*P*=Programm, *D*=Daten) | | | |
|---|---|---|---|---|
| | *P* und *D* extern | *P* On-Chip *D* extern | *P* extern *D* On-Chip | *P* und *D* On-Chip |
| Positives Beispiel | -71,3% | -72,3% | -73,2% | -75,2% |
| Lattice-Filter | -4,4% | -8,8% | nicht Messbar | |
| biquad_N_sections | -1,0% | -2,8% | unverändert | -0,5% |
| Livermore Kernel 1 | -16,6% | -16,8% | -15,5% | -15,1% |
| Livermore Kernel 5 | -19,2% | -18,6% | -18,4% | -17,2% |
| Livermore Kernel 7 | -7,3% | -12,6% | -2,3% | -5,9% |
| Livermore Kernel 11 | -24,5% | -24,8% | -23,9% | -23,9% |
| Livermore Kernel 12 | -10,7% | -11,4% | unverändert | -8,9% |

Tabelle 8.3: Änderung der Taktzyklen bei einer Energieoptimierung

| Programm/Benchmark | Speicherort (*P*=Programm, *D*=Daten) | | | |
|---|---|---|---|---|
| | *P* und *D* extern | *P* On-Chip *D* extern | *P* extern *D* On-Chip | *P* und *D* On-Chip |
| Positives Beispiel | +10,3% | +20,7% | +7,6% | -4,4% |
| Lattice-Filter | +0,4% | -9,7% | nicht Messbar | |
| biquad_N_sections | -0,2% | -3,2% | unverändert | -0,4% |
| Livermore Kernel 1 | -1,1% | -3,1% | -0,5% | -0,1% |
| Livermore Kernel 5 | -1,8% | -2,3% | -1,7% | +0,2% |
| Livermore Kernel 7 | +0,5% | -8,6% | +0,8% | -1,0% |
| Livermore Kernel 11 | -0,8% | -2,3% | -0,1% | -0,2% |
| Livermore Kernel 12 | +0,4% | -11,6% | unverändert | -1,1% |

Tabelle 8.4: Änderung der Leistung bei einer Energieoptimierung

Durchschnittlich kann bei den Benchmarks der Energieverbrauch um 13,7%, die Taktzyklenanzahl um 12,4% und die Leistungsaufnahme um 1,6% gesenkt werden. Deswegen optimiert Registerpipelining für eine Energieoptimierung die Taktzyklenanzahl wesentlich stärker als die Leistungsaufnahme.

Die Tabelle 8.5 zeigt die durchschnittliche Änderung der Energie für die einzelnen Speicherkombination und deren Zusammensetzung.

| | Speicherort ($P$=Programm, $D$=Daten) | | | |
|---|---|---|---|---|
| | $P$ und $D$ extern | $P$ On-Chip $D$ extern | $P$ extern $D$ On-Chip | $P$ und $D$ On-Chip |
| Energie | -13,5% | -18,9% | -10,2% | -12,3% |
| Taktzyklen | -13,2% | -14,5% | -10,0% | -11,9% |
| Leistung | -0,5% | -5,2% | -0,3% | -0,4% |

Tabelle 8.5: Durchschnittliche Energieänderung und ihre Zusammensetzung

Registerpipelining optimiert die Energie am besten, wenn die Instruktionen im On-Chip-Speicher und die Daten im externen Speicher liegen. Nur für diese Speicherkombination kann die Leistungsaufnahme bei einer Energieoptimierung wesentlich reduziert werden. Die Datenspeicherzugriffe sind relativ kostenintensiv im Vergleich zu den Instruktionslesekosten. Dieses Kostenverhältnis kommt oft in der Realität vor, da normalerweise die Instruktionen in einem energiesparenden ROM-Speicher und die Daten in einem relativ teuren RAM-Speicher liegen. Der ROM-Speicher des Evaluationboards wurde in dieser Diplomarbeit nicht betrachtet, da noch keine verwertbaren Daten über ihn vorlagen.

Die Tabelle 8.6 zeigt weitere Informationen zu den Optimierungen der Programme. Die einzelnen Werte beziehen sich auf die Speicherkombination, wenn sich die Instruktionen im On-Chip-Speicher und die Daten im externen Speicher befinden. Bei den Benchmarks nimmt die Programmgröße meistens zu und die Anzahl der Datenspeicherzugriffe verringert sich.

| Programm/ Benchmark | Programm-größe | ausgeführte Instruktionen | Datenspeicher-zugriffe |
|---|---|---|---|
| Positives Beispiel | -30,0% | -69,3% | -49,5% |
| Lattice-Filter | +10,1% | +4,4% | -26,5% |
| biquad_N_sections | +17,9% | +3,5% | -8,0% |
| Livermore Kernel 1 | +22,2% | -16,1% | -21,5% |
| Livermore Kernel 5 | 0,0% | -20,8% | -21,6% |
| Livermore Kernel 7 | +38,2% | +2,4% | -25,9% |
| Livermore Kernel 11 | 0,0% | -24,0% | -26,5% |
| Livermore Kernel 12 | +30,8% | +12,6% | -28,6% |

Tabelle 8.6: Weitere Änderungen durch Registerpipelining

Auffällig sind die guten Ergebnisse von Registerpipelining bei dem positiven Beispiel. Der Referenz-Compiler 'tcc' erzielt für dieses Programm schlechtere Energiewerte als der Compiler 'arm12cc' ohne die Optimierungstechnik Registerpipelining, weshalb auch bei ihm eine geeignete Optimierung sinnvoll wäre. Durch Registerpipelining entfallen etwa 50% der Datenspeicherzugriffe und 69%

der Instruktionslesezugriffe des Prozessors. Die Instruktionen eines Programms können hier besser optimiert werden als die Datenspeicherzugriffe. Deswegen wirken relativ teure Datenspeicherzugriffe auf die Ergebnisse, die in der Tabelle 8.2 dargestellt sind, nicht besonders positiv. Aufwendige Indexfunktionen können die Ergebnisse der Optimierungstechnik Registerpipelining verbessern.

Für die Optimierung des Programms Lattice-Filter stehen nur drei obere Register zur Verfügung. Registerpipelining kann diese Register für drei Programmoptimierungen verwenden, die die Anzahl der Datenspeicherzugriffe um 26,5% reduzieren. Deswegen kann die Optimierungstechnik die hohen Register effizient einsetzen. Da das Programm sehr viele Daten verwendet, konnten diese nicht im On-Chip-Speicher abgelegt werden, weshalb die entsprechenden Einträge in den Tabellen fehlen.

Da Pointerzugriffe meistens keine Indexfunktionen besitzen, deren Berechnung entfällt, kann die Energie des Programms biquad_N_sections kaum optimiert werden. Zudem entfallen nur 8% der Datenspeicherzugriffe. Wenn das Verfahren Registerpipelining für diesen Benchmark nicht den Registerdruck beachtet, dann vergrößert sich gegenüber dem nicht optimierten Programm der Energieverbrauch um 21,8% für den Fall, dass die Instruktionen und die Daten im externen Speicher liegen. Zusätzlich eingefügter Spillcode ist hierfür verantwortlich. Deshalb ist es für diese Optimierungstechnik unerlässlich, den Registerdruck zu beachten. Wenn die Instruktionen im externen Speicher und die Daten im On-Chip-Speicher liegen, dann wird keine Optimierung durchgeführt, da sonst der Energieverbrauch um bis zu 2,4% zunimmt.

Der Programmspeicherplatz vergrößert sich bei dem Kernel 7 um 38.2%, da der Prolog ziemlich lang ist und da 4 zusätzliche hohe Register am Funktionsanfang und -ende gesichert bzw. wiederhergestellt werden müssen. 25,9% der Datenspeicherzugriffe entfallen, wohingegen der Prozessor 2,4% mehr an Instruktionen ausführt. Diese Daten erklären zum Teil den relativ hohen Energiegewinn in der Tabelle 8.2 für den Fall, dass sich die Instruktionen im On-Chip-Speicher und die Daten im externen Speicher befinden. Bei den Programmen Lattice-Filter, biquad_N_sections und Kernel 12 verhält es sich ähnlich.

Ein weiterer Aspekt, der die besonders gute Energieoptimierung für diese Programme erklärt, wird anhand des Programms Kernel 12 aufgezeigt. Registerpipelining kann die Leistungsaufnahme des Programms Kernel 12 um 11,6% reduzieren, wenn sich die Instruktionen im On-Chip-Speicher und die Daten im externen Speicher befinden. Die Leistungsaufnahme vermindert sich beim AT91M40400-Chip um 1,5% und beim externen Speicher um 20.5%. Diese große Verringerung entsteht dadurch, dass weniger externe Speicherzugriffe relativ über mehr Taktzyklen ausgeführt werden. Die Optimierungstechnik spart 28,6% der Speicherzugriffe auf den externen Speicher ein und mindert die Taktzyklenanzahl nur um 11,4%. Deswegen wird für diese Speicherkombination die Energie besonders gut optimiert.

Das Verfahren Registerpipelining optimiert eine Speicherleseoperation in dem Kernel 11. Dieses Programm erreicht den höchsten Energiegewinn der überprüften Benchmarks. Die Abbildung 8.1 zeigt den entscheidenden Programmausschnitt aus dem Kernel. Registerpipelining führt eine Optimierung (bei $x[k\text{-}1]$) in der innersten Schleife bei einem kleinen Schleifenkörper durch.

Zwischen $x[k]$ und $x[k\text{-}1]$ liegt eine Iterationsdistanz von 1 und die Berechnung der Indexfunktion, die zwei zusätzliche Instruktionen umfasst, kann entfallen. Deswegen können die sehr guten Werte erzielt werden.

```
...
for( k=1; k<n; k++ ) {
    x[k]=x[k-1]+y[k];
}
...
```

Abbildung 8.1: Programmausschnitt aus Kernel 11

### 8.3.2 Einzelne Speicherzugriffe

Die Optimierungstechnik Registerpipelining optimiert in diesem Unterkapitel einzelne Speicherzugriffe für eine genaue Untersuchung der erzielten Ergebnisse. Unter der Verwendung verschiedener Speicher werden die Auswirkungen der Iterationsdistanz, der Indexfunktion und der Implementierungsvarianten untersucht.

#### Iterationsdistanz

Die Auswirkung der Iterationsdistanz auf die Verbesserungsmöglichkeiten der Optimierung wird anhand des Programms in Abbildung 8.2 untersucht.

```
int a[200];

int test() {
    int i,b,*c;
    b=0;
    c=a;
    for( i=1; i <100; i++) {
        b+=*c;          // erster Speicherzugriff
        b+=*(c+x);      // zweiter Speicherzugriff
        c+=1;
    }
    return(b);
}
```

Abbildung 8.2: Programm zum Testen unterschiedlicher Iterationsdistanzen

Das 'x' in dem Programm repräsentiert die Iterationsdistanz zwischen den beiden Speicherzugriffen. Es steht beispielsweise für Werte von 0 bis 16. Eine einzelne Load-Operation ('ldr r0,[r1,#x]') repräsentiert die gesamte Indexfunktion des Speicherzugriffes in dem nicht mit Registerpipelining optimierten Programm. Deswegen können die Auswirkungen der Iterationsdistanz genau analysiert werden.

Die Abbildung 8.3 stellt die Anzahl der benötigten Instruktionen innerhalb der Schleife dar, wenn Registerpipelining unabhängig vom Registerdruck und vom verwendeten Kostenmodell durchgeführt wird. Die Anzahl der eingefügten Instruktionen ist bei dem mit Registerpipelining optimierten Programm abhängig von der Iterationsdistanz. Ein Ziel bei der Implementierung der Optimierungstechnik Registerpipelining ist es, den Anstieg am Anfang der Kurve niedrig zu halten. Dies wird bei diesem Beispiel mit Hilfe der Technik Copy Propagation und mit der Optimierungsvariante 'VBeides' (siehe Kapitel 4.2.4) erreicht. Für die Iterationsdistanzen 0 und 1 müssen keine zusätzlichen Befehle eingefügt werden und der Speicherlesezugriff entfällt. Wenn Kopieroperationen die Registerpipeline bilden, lässt sich der Anstieg der Anzahl der Instruktionen um 1 ab einen bestimmten Punkt (hier von 1 bis 9) nicht vermeiden. Ab der Iterationsdistanz 10 spillt der Compiler einige Register, weshalb die Anzahl der neu eingefügten Instruktionen stark ansteigt.

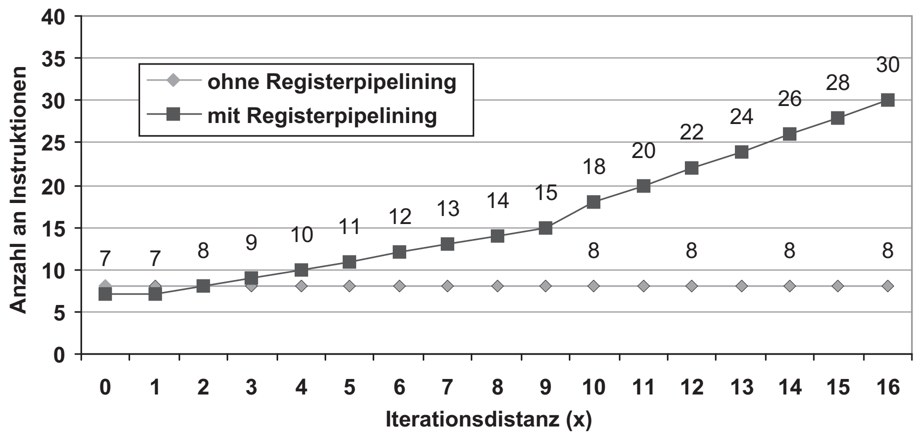


Abbildung 8.3: Instruktionsanzahl in Abhängigkeit der Iterationsdistanz

Die Abbildung 8.4 zeigt den Energieverbrauch in Abhängigkeit von der Iterationsdistanz und der unterschiedlichen Speicherorte. Der Energieverbrauch wächst mit der Iterationsdistanz an. Spätestens ab der Iterationsdistanz 10 verändert Registerpipelining das Programm nicht mehr, um Spilling zu vermeiden.

Die Abbildungen 8.5 und 8.6 zeigen die Zusammensetzung der Energiewerte aus Abbildung 8.4. Ein Vergleich zwischen den Energiewerten und den Taktzyklen in Abbildung 8.7 verdeutlicht, dass die Optimierungstechnik Registerpipelining die Energiewerte besser als die Anzahl der Taktzyklen optimieren kann, wenn die Instruktionen im On-Chip-Speicher und die Daten im exter-

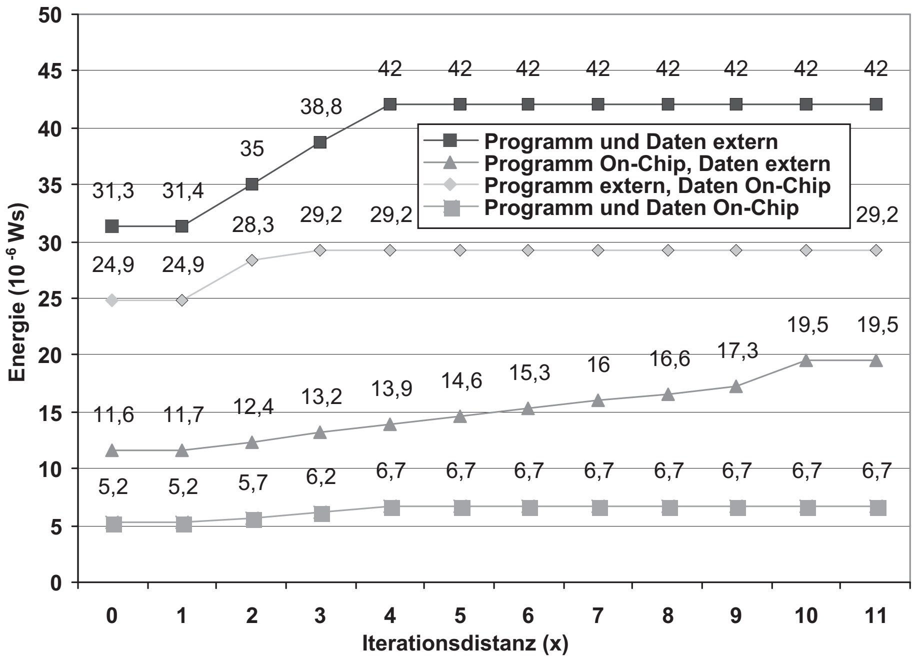


Abbildung 8.4: Energievergleich (nach Energie optimiert)

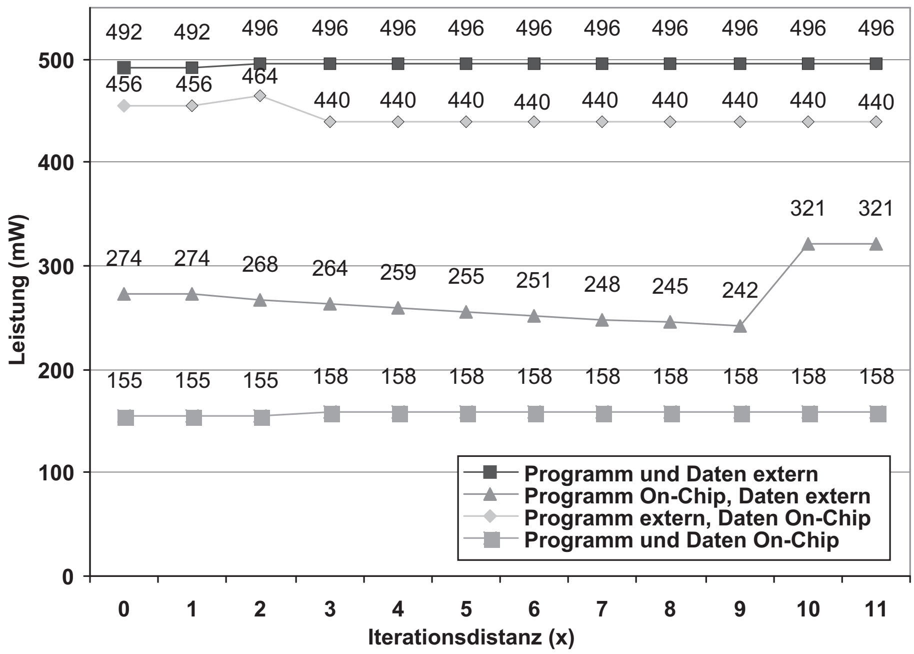


Abbildung 8.5: Leistungsvergleich (nach Energie optimiert)

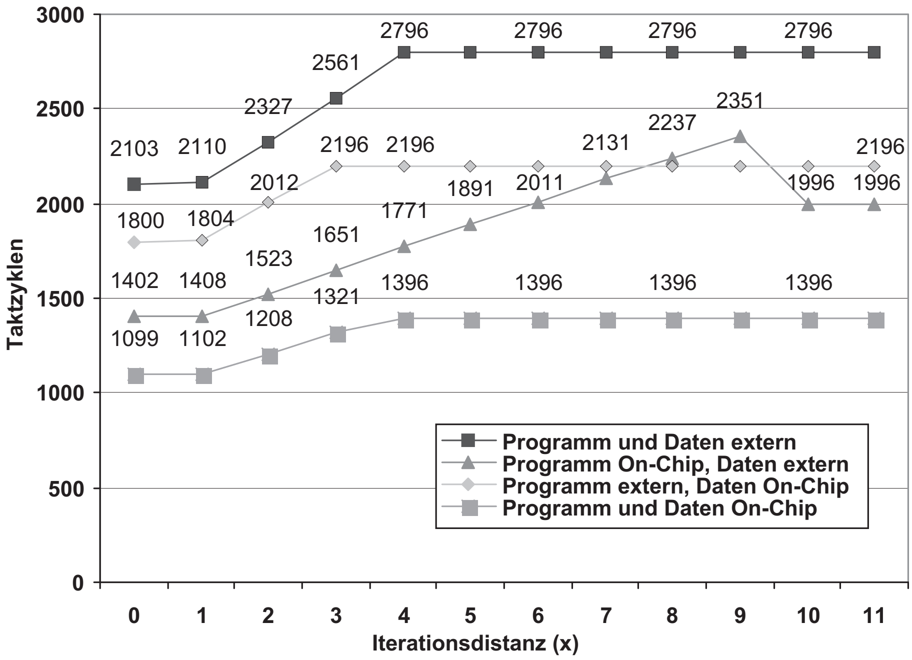


Abbildung 8.6: Taktzyklenvergleich (nach Energie optimiert)

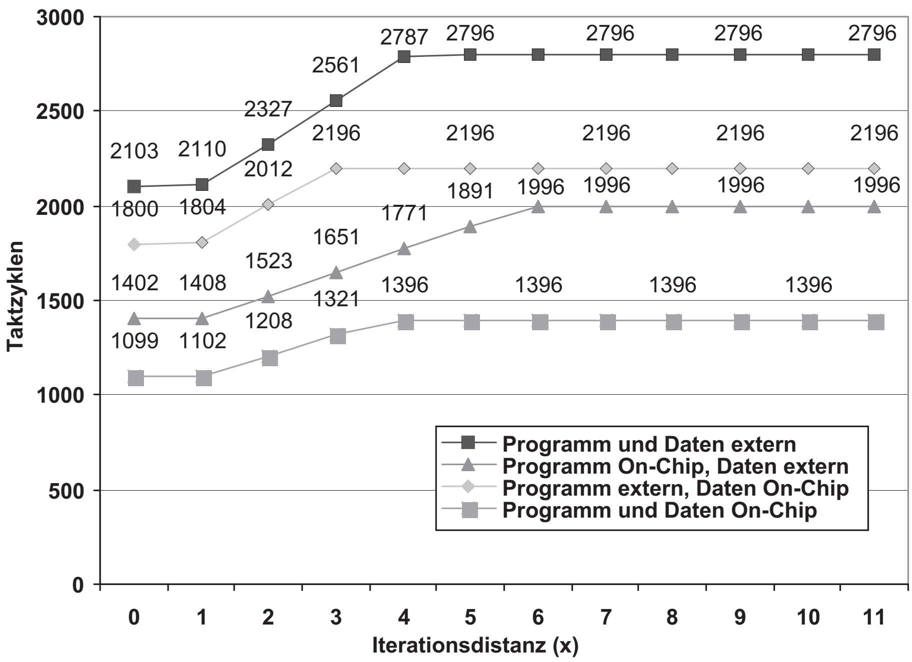


Abbildung 8.7: Taktzyklenvergleich (nach Taktzyklen optimiert)

nen Speicher liegen. Die Abbildung 8.5 zeigt, dass nur bei dieser Speicherkombination die Leistungsaufnahme des Beispielprogramms reduziert werden kann. Hauptsächlich senkt Registerpipelining die Leistungsaufnahme des externen Speichers, indem die Anzahl der Speicherzugriffe gleich bleibt und die Anzahl der Taktzyklen erhöht wird.

Aufgrund der gemessenen Werte können in der Tabelle 8.7 Iterationsdistanzen angegebenen werden, bis zu denen sich die Anwendung der Optimierungstechnik für dieses Beispiel lohnt.

| Speicherort | | Iterationsdistanz | |
|---|---|---|---|
| Programm | Daten | Energie | Taktzyklen |
| On-Chip | On-Chip | 3 | 3 |
| | extern | 9 | 5 |
| extern | On-Chip | 2 | 2 |
| | extern | 3 | 4 |

Tabelle 8.7: Grenzen der Iterationsdistanzen für eine sinnvolle Optimierung

Unterschiedliche Optimierungsziele des Compilers beeinflussen auch die Durchführung des Verfahrens Registerpipelining. Dies tritt beispielsweise in den Abbildungen 8.4, 8.6 und 8.7 bei einer Iterationsdistanz von 8 auf, wenn die Instruktionen im On-Chip-Speicher und die Daten im externen Speicher liegen. Unter dem Gesichtspunkt der Energie wird eine Programmoptimierung durchgeführt, wohingegen unter dem Aspekt der Zeit keine Optimierung stattfindet.

Genau umgekehrt verhält es sich bei der Iterationsdistanz 4, wenn die Instruktionen und die Daten im externen Speicher liegen. Registerpipelining führt eine Optimierung bzgl. der Zeit durch, aber nicht bzgl. der Energie ($42{,}4 \cdot 10^{-6}$Ws mit und $42{,}0 \cdot 10^{-6}$Ws ohne Optimierung).

### Indexfunktion

Wenn die Optimierungstechnik Registerpipelining einen Speicherzugriff durch einen Registerzugriff ersetzt, kann manchmal auch die Berechnung der Indexfunktion entfallen. Die Tabelle 8.8 zeigt die Anzahl der zusätzlich eingesparten Instruktionen bei der Berechnung verschiedener Indexfunktionen. Um die Werte zu ermitteln, wurden die angegebenen Speicherzugriffe mit den vorhandenen beiden in dem Programm aus Abbildung 8.2 ausgetauscht.

| erste Instruktion | zweite Instruktion | Anzahl der zusätzlich eingesparten Instruktionen |
|---|---|---|
| $b$+=*$c$; | $b$+=*$c$; | 0 |
| $b$+=$a$[$i$]; | $b$+=$a$[$i$]; | 0 |
| $b$+=$a$[$i$+1]; | $b$+=$a$[$i$]; | 2 |
| $b$+=$a$[2·$i$+1]; | $b$+=$a$[2·$i$-1]; | 2 |

Tabelle 8.8: Auswirkungen der Indexfunktion

Bis zu zwei zusätzliche Instruktionen können bei der Berechnung der Indexfunktionen entfallen. Zumeist handelt es sich hierbei um eine Addition bzw. Subtraktion und um eine Shift-Operation.

Beispielsweise kann die Optimierungstechnik Registerpipelining beim Kernel 5 in der Schleife eine 'lsl', 'sub' und 'ldr'-Operation durch eine 'mov'-Operation ersetzen. Auf diesem Teilstück des Programms werden die Anzahl der Taktzyklen um bis zu Faktor 5 optimiert. Da aber noch mehr Instruktionen in der Schleife liegen, fällt später der Gewinn relativ zum gesamten Programm kleiner aus.

**Implementierungsvarianten**

Die unterschiedlichen Implementierungsvarianten für Registerpipelining aus Kapitel 4.2.4 besitzen einen erheblichen Einfluss auf die Optimierung eines Programms.

Exemplarisch wird der Kernel 12 aus der Livermore Loops Benchmark-Suite analysiert, wenn Instruktionen und Daten im externen Speicher liegen. In dem Benchmark erfolgt eine Optimierung mit der Implementierungsvariante 'VBeides'. Für den folgenden Versuch verwendet Registerpipelining verschiedene Optimierungsvarianten aus Kapitel 4.2.4, um die Unterschiede zwischen ihnen zu verdeutlichen. Hierfür wurde das Optimierungsverfahren geeignet verändert, damit es nicht mehr die Variante 'VBeides' wählt. Die Ergebnisse sind in der Tabelle 8.9 dargestellt. Die Implementierungsvariante 'VBeides' ist um 12,4% besser als die Variante 'VNormal', die zwei 'mov'-Operationen mehr benötigt. Die Optimierungsvarianten 'VErzeugung' und 'VVerwendung' erreichen gleich gute Resultate, da sie auch gleich viele 'mov'-Operationen verwenden. Sie sind in diesem Fall um 7,2% schlechter als die Variante 'VBeides' und um 6,1% besser als die Variante 'VNormal'.

| Optimierungs-variante | Energie ($\cdot 10^{-6}$ Ws) |
|---|---:|
| VBeides | 51,7 |
| VErzeugung | 55,4 |
| VVerwendung | 55,4 |
| VNormal | 59,0 |

Tabelle 8.9: Unterschiedliche Optimierungsvarianten

## 8.4 Weitere Untersuchungen

In dieser Diplomarbeit wurden einige weitere Möglichkeiten vorgestellt, wie die Optimierungstechnik Registerpipelining verändert werden kann. Dieses Kapitel untersucht einige dieser Ansätze.

### 8.4.1 Pipeline im On-Chip-Speicher

Das Kapitel 4.2.6 beschreibt die Möglichkeit der Verwendung des On-Chip-Speichers anstelle von Registern für die Realisierung einer Pipeline. Die Abbildung 8.8 zeigt die Energiewerte für die erste und zweite Implementierungsvariante aus Beispiel 8.4.1. Das 'x' stellt in dem Programm die Iterationsdistanz zwischen den beiden Speicherzugriffen dar und repräsentiert bestimmte Werte zwischen 0 und 16. Das Array $a$ liegt im externen Speicher und das Array $m$ im On-Chip-Speicher. Der Compiler 'arm12cc' übersetzt die erste Variante mit und die zweite Variante ohne Registerpipelining.

**Beispiel 8.4.1** Optimierung mit Ringpuffer im On-Chip-Speicher

```
// * 1.Variante *
// ursprüngliche Schleife
int m[16], a[300];
#define x 0       // 0-16
...
int i,r,s;
r=0;
for( i=16; i<250; i++)
{
  r+=a[i-x];
  s=a[i];
}
```

```
// * 2.Variante *
// optimierte Schleife
int m[16], a[300];
#define x 2       // 2,8,16
...
int i,r,s,z;
r=0;
for( i=0; i<x; i++)
{
  m[i]=a[i+16-x];
}
z=0;
for( i=16; i<250; i++)
{
  r+=m[z];
  s=a[i];
  m[z]=s;
  z++;
  z=z&(x-1);
}
```

Nach Abbildung 8.8 sollten die Instruktionen für eine erfolgreiche Optimierung im On-Chip-Speicher liegen. Dann kann bei einer Iterationsdistanz von 16 die Energie um 15.4% reduziert werden. Beim Beispiel 4.2.6 kann eine Pipeline im On-Chip-Speicher ab einer Iterationsdistanz von 10 Vorteile gegenüber dem normalen Registerpipelining erzielen. Hier ist aber zu beachten, dass die Realisierung eines Ringpuffers manchmal mehr Operationen benötigt. Wenn der Registerdruck in dem Beispiel größer wäre, könnte eine Positionierung der Pipeline im On-Chip-Speicher auch für kleinere Iterationsdistanzen sinnvoll sein.

Die Taktzyklenanzahl und die Leistung fließen in die Energie ein. Die Leistungsaufnahme des externen Speichers kann bei den beiden sinnvollen Speicherkombinationen optimiert werden, während die Taktzyklenanzahl zunimmt.

Die Realisierung einer Pipeline im On-Chip-Speicher erzielt insbesondere bei einem hohen Registerdruck und bei großen Iterationsdistanzen Vorteile gegenüber einer normalen Registerpipeline, wenn die Instruktionen im On-Chip-Speicher und die Daten im externen Speicher liegen. Die Datenspeicherzugriffe sollten im Verhältnis zu den Instruktionslesekosten relativ teuer sein, damit eine Optimierung sinnvoll ist.

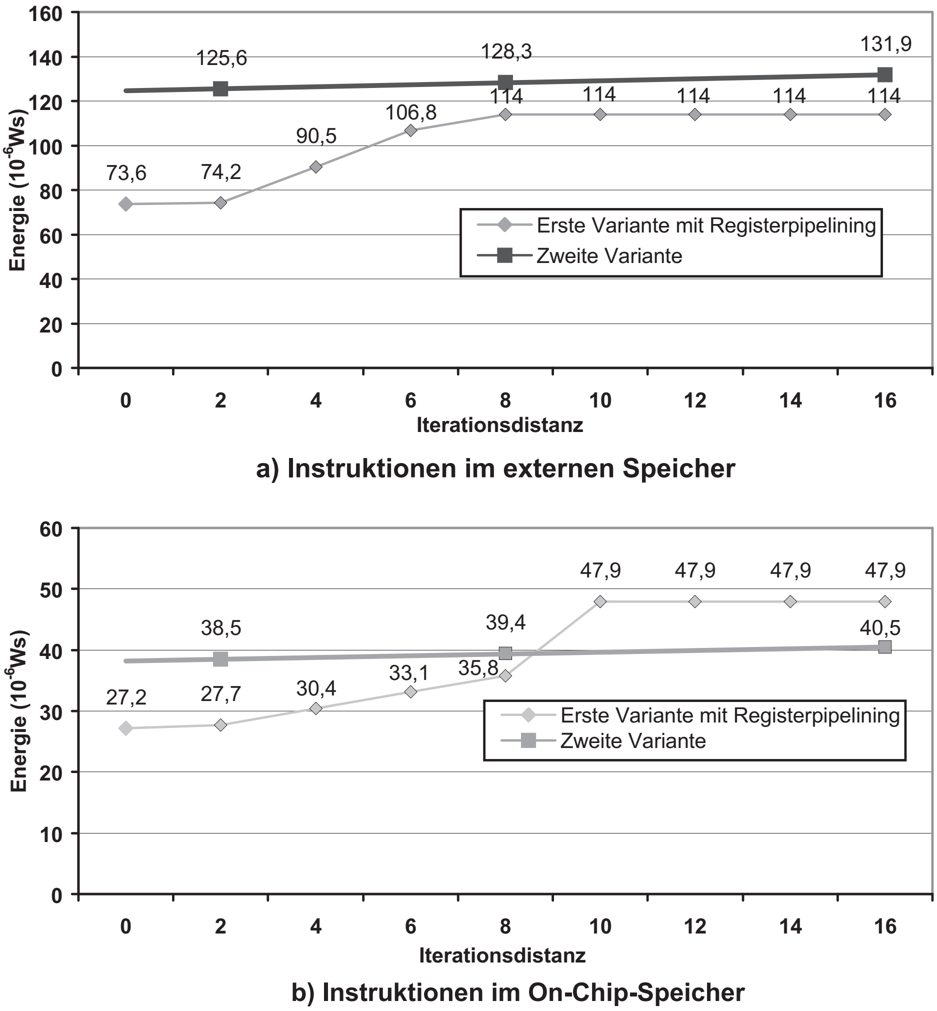


Abbildung 8.8: Energievergleich mit unterschiedlichen Pipelinepositionen

### 8.4.2 Multiple Load Instruktion

Die 'ldmia'-Instruktion kann, wie in Kapitel 4.2.5 beschrieben, zu einer Verbesserung der Prologgenerierung führen. Da der Prolog aber normalerweise nicht so oft ausgeführt wird wie das Innere einer Schleife, kann hierdurch das gesamte Programm nur unwesentlich optimiert werden.

Als Beispiel dient in diesem Abschnitt eine einzelne 'ldmia'-Operation die zahlreiche 'ldr'-Operationen ersetzt. Die Anzahl der benötigten Instruktionen, um aufeinanderfolgende Speicherinhalte einzulesen, kann um bis zu Faktor 8 reduziert werden, falls die Instruktion 8 Speicherinhalte liest. Für die verschiedenen Speicherzugriffsmöglichkeiten stehen in der Tabelle 8.10 die Anzahl der benötigten Taktzyklen. Der Wert '#Speicherzugriffe' steht für eine Zahl von 1 bis 8. Die Anzahl der Taktzyklen kann fast um bis zu Faktor 3 reduziert werden.

| Speicherort | | Taktzyklenanzahl | |
|---|---|---|---|
| Programm | Daten | 'ldmia'-Operation | 'ldr'-Operation |
| On-Chip | On-Chip | 2+#Speicherzugriffe | 3·#Speicherzugriffe |
| | extern | 2+4·#Speicherzugriffe | 6·#Speicherzugriffe |
| extern | On-Chip | 3+#Speicherzugriffe | 4·#Speicherzugriffe |
| | extern | 3+4·#Speicherzugriffe | 7·#Speicherzugriffe |

Tabelle 8.10: Taktzyklenvergleich (Werte der 'ldr'-Operation nach [SS00])

Die Abbildung 8.9 zeigt einen Energievergleich zwischen einer 'ldmia'-Operation, die sieben Speicherinhalte einliest, und sieben 'ldr'-Operationen. Die 'ldmia'-Operation reduziert den Energieverbrauch der Speicherzugriffe um bis zu 75,8%.

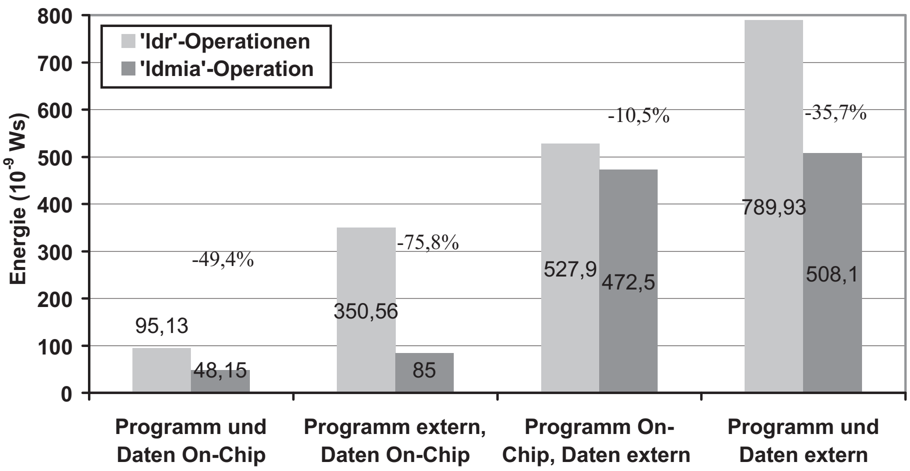


Abbildung 8.9: Energievergleich mit 7 Speicherzugriffen

Mit Hilfe der 'ldmia'-Operation kann ein beachtenswerter Gewinn erzielt werden. Die Instruktion kann einen geeigneten Schleifenprolog optimieren, wenn die Induktionsvariable pro Iteration um 1 erhöht wird.

## 8.5 Zusammenfassung

Die durchschnittliche Abweichung des Trace-Analyzers bei den durchgeführten Versuchen beträgt 4%. Die Genauigkeit der Energiewerte reicht aus um die Optimierungstechnik Registerpipelining mit Hilfe des Trace-Analyzers zu beurteilen.

Vorteile erzielt die Optimierungstechnik Registerpipelining bei Programmen, die bestimmte Datenspeicherzugriffe oft durchführen. Der Energieverbrauch, die Geschwindigkeit und die Leistungsaufnahme kann optimiert werden. Der Platzbedarf eines Programms erhöht sich meistens. Bei den überprüften Benchmarks konnte die Optimierungstechnik Registerpipelining bei einer Optimierung der Energie durchschnittlich den Energieverbrauch um 13,7%, die

Taktzyklenanzahl um 12,4% und die Leistungsaufnahme um 1,6% senken. Somit ist die Reduzierung der Taktzyklen hauptsächlich für die Optimierung der Energie verantwortlich. Die Leistungsaufnahme des externen Speichers kann wesentlich optimiert werden, wenn die Anzahl der Speicherzugriffe im Verhältnis zu der Anzahl der Taktzyklen abnimmt. Auf die Leistungsaufnahme des AT91M40400-Chips hat die Optimierungstechnik einen geringen Einfluss, da alle Befehle einen ähnlich starken Strom von etwa 41mA bis 53mA hervorrufen.

Für den interessanten Fall, dass die Datenspeicherzugriffe relativ kostenintensiv im Vergleich zu den Instruktionslesekosten sind, kann Registerpipelining bei einer Energieoptimierung der Benchmarks durchschnittlich den Energieverbrauch um 18,9%, die Taktzyklenanzahl um 14,5% und die Leistungsaufnahme um 5,2% senken.

Die verwendete Implementierung der Optimierungstechnik Registerpipelining aus Kapitel 4.2.4 dieser Diplomarbeit kann den Energieverbrauch eines Benchmarks um 12,4% mehr reduzieren als die Implementierung, die in Kapitel 4.2.1 dargestellt ist.

Zur Realisierung einer Pipeline kann auch ein Ringpuffer im On-Chip-Speicher verwendet werden. Diese Positionierung kann aber nur bei einem hohen Registerdruck oder bei großen Iterationsdistanzen Vorteile erzielen, wenn sich die Daten im externen Speicher und die Instruktionen im On-Chip-Speicher befinden. Der Energieverbrauch kann dann um 15,4% reduziert werden.

Die Multiple-Load-Instruktion kann zu einer Verbesserung der Prologgenerierung führen. Da der Prolog aber normalerweise nicht so oft ausgeführt wird wie das Innere einer Schleife, kann hierdurch das gesamte Programm nur unwesentlich optimiert werden. Die 'ldmia'-Operation kann den Energieverbrauch der Speicherzugriffe um 75,8% reduzieren.

# Kapitel 9

# Zusammenfassung und Ausblick

Optimierende Compiler sind für eine effiziente Codegenerierung notwendig. Die Optimierung des Energieverbrauchs gehört seit einigen Jahren zu den wichtigen Zielen innerhalb der Codegenerierung und wird zukünftig weiter an Bedeutung gewinnen.

Speicherzugriffe können einen Anteil von 86% am gesamten Energieverbrauch einer Instruktion beim ATMEL-Evaluationboard besitzen. Deswegen können Optimierungstechniken, die die Speicherzugriffe optimieren, große Verbesserungen erzielen.

Die Speicherzugriffsoptimierung Registerpipelining legt Speicherinhalte in Registern ab, um effizient und energiesparend erneut auf einen Wert zugreifen zu können. Die Optimierungstechnik kann hierfür den hohen Registersatz des ARM7TDMI-Prozessors im Thumb-Instruktionssatz erfolgreich verwenden. Zur Realisierung einer Pipeline kann auch ein Ringpuffer im On-Chip-Speicher verwendet werden.

Vorteile erzielt die Optimierungstechnik bei Programmen, die bestimmte Datenspeicherzugriffe öfters durchführen. Der Energieverbrauch, die Geschwindigkeit und die Leistungsaufnahme kann optimiert werden, wohingegen sich der Platzbedarf eines Programms meistens erhöht. Bei den überprüften Benchmarks konnte die Optimierungstechnik Registerpipelining bei einer Optimierung der Energie durchschnittlich den Energieverbrauch um 13,7%, die Taktzyklenanzahl um 12,4% und die Leistungsaufnahme um 1,6% senken. Somit ist die Reduzierung der Taktzyklen hauptsächlich für die Optimierung der Energie verantwortlich. Die Leistungsaufnahme des externen Speichers kann nur wesentlich reduziert werden, wenn die Anzahl der Speicherzugriffe im Verhältnis zu der Anzahl der Taktzyklen abnimmt. Auf die Leistungsaufnahme des AT91M40400-Chips hat die Optimierungstechnik einen geringen Einfluss, da alle Befehle etwa einen Strom von 41mA bis 53mA hervorrufen. Die besten Resultate kann Registerpipelining erzielen, wenn die Instruktionen im On-Chip-Speicher und die Daten im externen Speicher liegen. Hier konnte eine Energieoptimierung bei den überprüften Benchmarks durchschnittlich den Energieverbrauch um 18,9%, die Taktzyklenanzahl um 14,5% und die Leistungsaufnahme um 5,2% senken.

Wenn die heutigen Compiler die Optimierungstechnik Registerpipelining nicht verwenden, dann existieren auch Gründe dafür. Viele Prozessoren besitzen hochentwickelte Cache-Strukturen, die die Verbesserungen durch das Verfahren (auch bzgl. Taktzyklen) verringern. Zudem treten Speicherzugriffe, die auf die selbe Speicheradresse zugreifen und zwischen denen eine Iterationsdistanz von mehr als 1 liegt, nicht häufig genug auf, um einen großen Gewinn zu erzielen. Die Redundant Load Elimination deckt bereits einen großen Teil des Optimierungspotentials des Verfahrens Registerpipelining ab. Weiterhin werden umfangreiche und spezielle Analysen benötigt, um die Korrektheit der durchgeführten Optimierungen zu gewährleisten.

Die Verwendung von Registerpipelining in einem Compiler hängt von der Einfachheit, Korrektheit und Anwendbarkeit ab. Eine präzise Untersuchung von nicht affinen Indexfunktionen scheint unter diesem Aspekt nicht sinnvoll zu sein, da sie kaum Anwendungsgebiete besitzen.

Die vorgestellte und verwendete Implementierung der Optimierungstechnik Registerpipelining aus Kapitel 4.2.4 dieser Diplomarbeit kann den Energieverbrauch eines Benchmarks um 12,4% mehr reduzieren als die Implementierung, die in Kapitel 4.2.1 dargestellt ist. Durch eine bessere Implementierung, wie sie beispielsweise in [RB96] beschrieben ist, kann das Verfahren Registerpipelining noch bessere Ergebnisse liefern. Es bleibt zu prüfen, wie hoch dieser Gewinn ausfällt.

Die entwickelte Datenflussanalyse sichert ein möglichst großes Anwendungsgebiet der Optimierungstechnik und besitzt eine sehr hohe Präzision. Durch die Art und Weise, wie die Indexfunktionen rekonstruiert werden, kann das Verfahren alle Speicherzugriffe analysieren und berücksichtigen. Die Vor- und Nachteile der angepassten $\delta$-Array-Datenflussanalyse im Vergleich zu anderen Datenflussanalysen könnten noch genau untersucht werden. Bei dem implementierten Verfahren Registerpipelining hat sich dieses $\delta$-Verfahren bewährt.

Programme sollten in einem schnellen und energiesparenden ROM-Speicher abgelegt werden. Interessant wäre eine Betrachtung des ROM-Speichers auf dem Evaluationboard gewesen, da dieser relativ zu dem RAM-Speicher oft eine sehr niedrige Leistungsaufnahme besitzt. Gerade die Speicherkombination, bei der ein günstiger Instruktionsspeicher und ein teurer Datenspeicher vorhanden ist, ist für das Verfahren Registerpipelining wünschenswert. Zum Abgabetermin dieser Diplomarbeit lagen noch keine verwertbaren Daten über den ROM-Speicher des Evaluationboards vor.

Die Energieberechnung des Trace-Analyzers wich durchschnittlich um 4% von den tatsächlichen Werten ab. Genauere Ergebnisse sind zu erwarten, wenn die hierfür relevanten Versuche mit dem Evaluationboard abgeschlossen sind. Demnach wäre eine genaue Überprüfung der ermittelten Werte in dieser Diplomarbeit in Zukunft sinnvoll.

Eine genaue Betrachtung des Arm-Instruktionssatzes im Hinblick auf eine Energieoptimierung und des Verfahrens Registerpipelining könnte auch sinnvoll sein.

# Anhang A

# Dokumentation der Implementierung

Dieses Kapitel dokumentiert die Programme, die im Rahmen dieser Diplomarbeit implementiert wurden. Abbildung A.1 skizziert die Positionen dieser Programme während der Übersetzung und Analyse eines C-Programms.

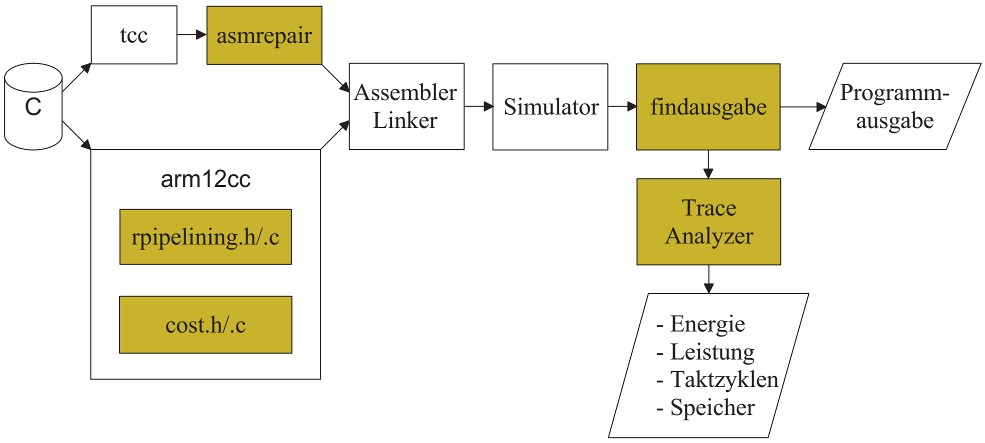


Abbildung A.1: Position der implementierte Programme

## A.1 Programme der Arbeitsumgebung

Das Kapitel 5 stellt die Arbeitsumgebung vor, innerhalb derer die nachstehenden Programme eingesetzt werden. Dies war insgesamt notwendig, um die erzielten Verbesserungen der Optimierungstechnik Registerpipelining effizient beurteilen zu können.

### A.1.1 asmrepair

Das Programm 'asmrepair' entfernt kleinere Fehler aus dem Assemblercode, der vom Compiler 'tcc' generierten wurde. Die Bedienung erfolgt über die

Kommandozeile, wobei folgende Parameter zu übergeben sind:

*asmrep <Quelldatei> <Zieldatei>*

Der Assemblercode der Quelldatei wird zeilenweise eingelesen, eventuell korrigiert und in die Zieldatei geschrieben. Das Beispiel A.1.1 verdeutlicht die Arbeitsweise von dem Programm. Bei dem Beispiel wurde zuerst der Fehler des doppelten 'DCB'-Befehls korrigiert und daran anschließend wurden noch die beiden letzten 'DCD'-Befehle eingefügt, damit die passenden C-Bibliotheken auch wirklich eingebunden werden. Der Linker würde sonst die Importierung als redundant einstufen.

**Beispiel A.1.1** Transformation der Quelldatei in die Zieldatei

```
// Quelldatei generiert vom 'tcc':     // korrigierte Zieldatei:
...                                    ...
main                                   main
  push      {lr}                         push      {lr}
  adr       R0,F1L1                      adr       R0,F1L1
  bl        _printf                      bl        _printf
  mov       R0,#0                        mov       R0,#0
  pop       {pc}                         pop       {pc}
F1L1                                   F1L1
  DCB       DCB 'Test'                   DCB       'Test'     // geändert
...                                    ...
  EXPORT    main                         EXPORT    main
  IMPORT    _main                        IMPORT    _main
  IMPORT    __main                       IMPORT    __main
  IMPORT    _printf                      IMPORT    _printf
...                                    ...
  END                                    DCD       __main     // geändert
                                         DCD       _main      // geändert
                                         END
```

### A.1.2 findausgabe

Das Programm 'findausgabe' untersucht und filtert die Ausgaben des Simulators, um die Programmausgaben des simulierten Programms zu finden. Hierzu werden die Ausgaben des Simulators in die erste Zieldatei und die Ausgaben des Programms in die zweite Zieldatei geschrieben. Die Bedienung erfolgt über die Kommandozeile mit folgenden Parametern:

*findausgabe <Quelldatei> <1.Zieldatei> <2.Zieldatei>*

Die Quelldatei wird zeilenweise analysiert und die Ausgaben werden in die erste Zieldatei oder in die zweite Zieldatei geschrieben.

### A.1.3 traceanalyzer

Das Programm 'traceanalyzer' analysiert einen Trace des Simulators 'armsd' und berechnet hieraus die Werte, die in Kapitel 5.2 beschrieben werden. Die Bedienung erfolgt über die Kommandozeile mit folgenden Parametern:

*traceanalyzer* <*Tracedatei*> <*Simulatorausgabe*> <*Linkerausgabe*>
<*Instruktionsdaten*> <*Speicherdaten*>

Der Traceanalyzer analysiert zeilenweise die angegebene Tracedatei mit Hilfe der Informationen aus den Dateien mit den Instruktions- und Speicherdaten. Das Programm filtert zuerst die für die Analyse relevanten Instruktionen aus dem angegeben Trace mit Hilfe der Ausgaben des Simulators heraus. Bestimmte Befehle wurden als Prolog und Epilog zu dem jeweiligen Assemblerprogramm hinzugefügt, damit der Simulator 'armsd' korrekt arbeiten kann und damit die Instruktionen an den richtigen Speicherpositionen stehen. Aus den Ausgaben des Linkers wird die Programmgröße entnommen. Zusätzlich können mit der Option '-d' Informationen über die Analyse der einzelnen Instruktionen ausgegeben werden.

## A.2 Programmteile des Compilers

### A.2.1 Kostenbewertung

Die Dateien 'cost.h' und 'cost.c' beinhalten eine Kostenbewertung für die Instruktionen. Der Compiler 'arm12cc' und der Traceanalyzer binden die Datei 'cost.h' mit dem '#include'-Befehl ein. Es werden Funktionen zur Verfügung gestellt, um die entsprechenden Dateien einzulesen und auf die jeweiligen Information zuzugreifen. Unterschieden wird in Funktionen für Instruktions- und Speicherdaten.

Für die Daten, die die Instruktionen betreffen, werden unter anderem die folgenden Operationen bereitgestellt:

- Einlesen einer Datei mit den entsprechenden Daten
- Ermittlung der Taktzyklen, des Speicherplatzes und des Prozessorstroms für eine Instruktion
- Ermittlung der Anzahl der Datenspeicherzugriffe

Für die Daten, die den Speicher betreffen, werden unter anderem die folgenden Operationen bereitgestellt:

- Einlesen einer Datei mit den entsprechenden Daten
- Anfragen beantworten bzgl. der Werte Startadresse, Größe, Bitbreite, Waitstates und Stromwerte eines Speicher

### A.2.2 Optimierungstechnik Registerpipelining

Die Dateien 'rpipelining.h' und 'rpipelining.c' beinhalten die Optimierungstechnik Registerpipelining. Der Compiler 'arm12cc' bindet die Datei 'rpipelining.h' mit dem '#include'-Befehl ein. Zu den folgenden Themen wurden Funktionen implementiert:

- RegisterPipelining
  Die Optimierungstechnik Registerpipelining, wie sie in Kapitel 6.4 beschrieben ist, wird aufgerufen. Wenn der Rückgabewert der entsprechenden Funktion eine 0 ist, wurden keine Optimierungen durchgeführt.

- UCE
  Für die Optimierungstechnik Useless Code Elimination steht eine Funktion zur Verfügung. Der Rückgabewert entspricht der Anzahl der entfernten Instruktionen.

- BBCP
  Eine Copy Propagation kann durchgeführt werden und der Rückgabewert der Funktion entspricht der Anzahl der durchgeführten Optimierungen.

- IterationCount
  Eine Funktion schätzt die Häufigkeiten mit der eine Instruktion durchlaufen wird. Diese Funktion wurde für die Registerallokation des Compilers 'arm12cc' geschrieben, um das Spilling zu optimieren. Berücksichtigung finden alle Arten von Kontrollstrukturen.

# Anhang B

# Thumb-Instruktionssatz

| Assemblerinstruktionen | Takt-zyk-len | Speicher (Bytes) Instruk-tion | Daten | Wirkung |
|---|---|---|---|---|
| ADC Rd,Rs | 1 | 2 | 0 | Rd=Rd + Rs + C-Bit |
| ADD Hd,Hs | 1 | 2 | 0 | Hd=Hd + Hs |
| ADD Hd,Rs | 1 | 2 | 0 | Hd=Hd + Rs |
| ADD Rd,#Offset8 | 1 | 2 | 0 | Rd=Rd + imm8 |
| ADD Rd,Hs | 1 | 2 | 0 | Rd=Rd + Hs |
| ADD Rd,PC,#Imm | 1 | 2 | 0 | add from address from PC |
| ADD Rd,Rs,#Offset3 | 1 | 2 | 0 | Rd=Rs + immed3 |
| ADD Rd,Rs,Rn | 1 | 2 | 0 | Rd=Rs + Rn |
| ADD Rd,SP,#Imm | 1 | 2 | 0 | Rd=SP + imme8 · 4 |
| ADD SP,#-Imm | 1 | 2 | 0 | SP=SP - imme7 · 4 |
| ADD SP,#Imm | 1 | 2 | 0 | SP=SP + imme7 · 4 |
| AND Rd,Rs | 1 | 2 | 0 | Rd=Rd AND Rs |
| ASR Rd, Rs, #Offset5 | 1 | 2 | 0 | arithmetic shift right |
| ASR Rd,Rs | 2 | 2 | 0 | arithmetic shift right |
| B | 3 | 2 | 0 | unconditional branch |
| BIC Rd,Rs | 1 | 2 | 0 | Rd=Rd AND NOT Rm |
| BL | 4 | 4 | 0 | long branch with link |
| BX Hs | 3 | 2 | 0 | R15= Hs AND 0xFFFFFFFE |
| BX Rs | 3 | 2 | 0 | R15= Rs AND 0xFFFFFFFE |
| Bxx | 3 | 2 | 0 | conditional branch |
| CMN Rd,Rs | 1 | 2 | 0 | compare negativ |
| CMP Hd,Hs | 1 | 2 | 0 | compare registers |
| CMP Hd,Rs | 1 | 2 | 0 | compare registers |
| CMP Rd,#Offset8 | 1 | 2 | 0 | compare immediate |
| CMP Rd,Hs | 1 | 2 | 0 | compare registers |
| CMP Rd,Rs | 1 | 2 | 0 | compare registers |
| EOR Rd,Rs | 1 | 2 | 0 | Rd=Rd EOR Rs |
| LDMIA Rb!,Rlist | 2-10 | 2 | 0-32 | loads list of registers |
| LDR Rd,[PC,#Imm] | 3 | 2 | 4 | load PC-relativ |
| LDR Rd,[Rb,#Imm] | 3 | 2 | 4 | Rd=[Rb+immed5 · 4] |
| LDR Rd,[Rb,Ro] | 3 | 2 | 4 | Rd=[Rb+Ro] (word) |
| LDR Rd,[SP,#Imm] | 3 | 2 | 4 | load SP-relativ |
| LDRB Rd,[Rb,#Imm] | 3 | 2 | 1 | Rd=[Rb+immed5] (byte) |
| LDRB Rd,[Rb,Ro] | 3 | 2 | 1 | Rd=[Rb+Ro] (byte) |
| LDRH Rd,[Rb,#Imm] | 3 | 2 | 2 | Rd=[Rb+immed5 · 2] |
| LDRH Rd,[Rb,Ro] | 3 | 2 | 2 | Rd=[Rb+Ro] (halfword) |

| Assemblerinstruktionen | Takt-zyklen | Speicher (Bytes) Instruktion | Speicher (Bytes) Daten | Wirkung |
|---|---|---|---|---|
| LDRSB Rd,[Rb,Ro] | 3 | 2 | 1 | load signed byte |
| LDRSH Rd,[Rb,Ro] | 3 | 2 | 2 | load signed halfword |
| LSL Rd, Rs, #Offset5 | 1 | 2 | 0 | Rd=Rs << immed5 |
| LSL Rd,Rs | 2 | 2 | 0 | Rd=Rd << Rs |
| LSR Rd, Rs, #Offset5 | 1 | 2 | 0 | Rd=Rs >> immed5 |
| LSR Rd,Rs | 2 | 2 | 0 | Rd=Rd >> Rs |
| MOV Hd,Hs | 1 | 2 | 0 | Hd=Hs |
| MOV Hd,Rs | 1 | 2 | 0 | Hd=Rs |
| MOV Rd,#Offset8 | 1 | 2 | 0 | Rd=immed8 |
| MOV Rd,Hs | 1 | 2 | 0 | Rd=Hs |
| MUL Rd,Rs | 2-5 | 2 | 0 | Rd=Rs · Rd |
| MVN Rd,Rs | 1 | 2 | 0 | Rd=NOT Rs |
| NEG Rd,Rs | 1 | 2 | 0 | Rd=-Rs |
| ORR Rd,Rs | 1 | 2 | 0 | Rd=Rd OR Rs |
| POP Rlist,PC | 5-13 | 2 | 4-36 | pop and return |
| POP Rlist | 2-10 | 2 | 0-32 | pop registers from stack |
| PUSH RegList,LR | 2-10 | 2 | 4-36 | push LR and registers |
| PUSH Reglist | 1-9 | 2 | 0-32 | push registers onto stack |
| ROR Rd,Rs | 2 | 2 | 0 | rotate right |
| SBC Rd,Rs | 1 | 2 | 0 | Rd=Rd - Rs + C-Bit |
| STMIA Rb!,Rlist | 1-9 | 2 | 0-32 | stores list of registers |
| STR Rd,[Rb,#Imm] | 2 | 2 | 4 | [Rb+immed5 · 4]=Rd |
| STR Rd,[Rb,Ro] | 2 | 2 | 4 | [Rb+Ro]=Rd (word) |
| STR Rd,[SP,#Imm] | 2 | 2 | 4 | store SP-relativ |
| STRB Rd,[Rb,#Imm] | 2 | 2 | 1 | [Rb+immed5]=Rd (byte) |
| STRB Rd,[Rb,Ro] | 2 | 2 | 1 | [Rb+Ro]=Rd (byte) |
| STRH Rd,[Rb,#Imm] | 2 | 2 | 2 | [Rb+immed5 · 2]=Rd |
| STRH Rd,[Rb,Ro] | 2 | 2 | 2 | [Rb+Ro]=Rd (halfword) |
| SUB Rd,#Offset8 | 1 | 2 | 0 | Rd=Rd - imm8 |
| SUB Rd,Rs,#Offset3 | 1 | 2 | 0 | Rd=Rs - immed3 |
| SUB Rd,Rs,Rn | 1 | 2 | 0 | Rd=Rs - Rn |
| SWI Value8 | 3 | 2 | 0 | software interrupt |
| TST Rd,Rs | 1 | 2 | 0 | test bits |

Tabelle B.1: Thumb-Instruktionssatz (nach [ARML99])

Die Tabelle B.1 zeigt den Thumb-Instruktionssatz. Das Register 'Rd' ist ein unteres Register und das 'd' bezeichnet, dass das Register den Zielort (destination) der Operation darstellt. Ein Register, das mit dem Buchstaben 'H' beginnt, kennzeichnet ein hohes Register. Die angegebenen Taktzyklen beziehen sich auf den ARM7TDMI-Prozessor ohne Berücksichtigung des Speichers.

Weitere Einzelheiten zu den jeweiligen Instruktionen sind in [ARML99], [Atm99b] und [ARML95] aufgeführt.

# Literaturverzeichnis


[ARML95] Advanced RISC Machines Ltd, ARM: *ARM7TDMI Data Sheet.* 1995.

[ARML98] Advanced RISC Machines Ltd, ARM: *Application Note 48, Scatter Loading.* 1998.

[ARML99] Advanced RISC Machines Ltd, ARM: *Thumb Instruction Set Quick Reference Card.* ARM QRC 0001D, 1999.

[Atm99a] Atmel: *ARM7TDMI Embedded RISC Microcontroller.* 1999.

[Atm99b] Atmel: *ARM7TDMI (Thumb) Datasheet.* 1999. Literature Number:0673B.

[Atm99c] Atmel: *AT91M40400 Summary AT91 ARM Thumb Microcontroller.* 1999.

[ED93] E. Duesterwald, R. Gupta, M. Soffa: *A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations.* Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation, 28(6):68–77, 1993.

[EM98] E. Macii, M. Pedram, F.Somenzi: *High-Level Power Modeling, Estimation, and Optimization.* TRANSACTION ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, 17(11):1061–1079, 1998.

[Fra99] Franke, B.: *Analysen und Methoden optimierender Compiler zur Steigerung der Effizienz von Speicherzugriffen in eingebetteten Systemen.* Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Diplomarbeit, 1999.

[GS99] G. Sinevriotis, T. Stouraits: *Power Analysis of the ARM 7 Embedded Microprocessor.* PATMOS'99, 9th int. Workshop, 1999.

[Her89] Hering, Martin, Stohrer: *Physik für Ingenieure.* VDI Verlag, 1989.

[Hof99] Hofmeister, T.: *Effiziente Algorithmen.* Universität Dortmund, Fakultät Informatik, Lehrstuhl II, Sommersemester 1999.

[HS81] H. Schneid, L. Warlich, H. Wallrabenstein: *Schülerduden 'Die Mathematik I'.* Bibliographisches Institut Mannheim, Wien, Zürich, 1981.

[RB96] R. Bodik, R. Gupta: *Array Data Flow Analysis for Load-Store Optimizations in Fine-Grain Architectures.* International Journal of Parallel Programming, 24(6):481–512, 1996.

[Sch99] Schwiegelshohn, U.: *Parallele Rechnersysteme II.* Universität Dortmund, Fakultät Elektrotechnik, Vorlesungsunterlagen, 1999.

[SS99] S. Steinke, L. Wehmeyer: *Low Power Code Generierung.* Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Vortragsfolien, 1999.

[SS00] S. Steinke, M. Theokharidis: *Stromwerte und Taktzyklen.* Interne Arbeitsunterlagen, 2000.

[VT94] V. Tiwari, S. Malik, A. Wolfe: *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization.* IEEE Transactions on VLSI Systems, 2(4):437–445, 1994.