

# **Diplomarbeit**

Energieoptimierung  
einer  
MPEG-Applikation

Thomas Hüls

Diplomarbeit  
am Lehrstuhl 12  
des Fachbereichs Informatik  
der Universität Dortmund

14. März 2002

**Betreuer:**

Dipl.-Inform. Stefan Steinke  
Prof. Dr. Peter Marwedel



1	Einleitung.....	5
1.1	Motivation.....	5
1.2	Ziele der Arbeit.....	6
1.3	Kapitelübersicht.....	6
2	MPEG-Grundlagen.....	9
2.1	MPEG-Standards.....	9
2.2	Arbeitsweise des MPEG-2 Dekoders.....	10
2.3	Hotspots im MPEG-2 Dekoder.....	19
3	Aufbau von energieoptimierenden Compilern.....	27
3.1	Compiler-Grundlagen.....	27
3.2	Optimierende Compiler.....	29
3.3	Energieoptimierende Compiler.....	32
3.4	Standard-Optimierungen.....	33
4	Vorgehensweise.....	37
4.1	Methodik.....	37
4.2	Betrachtete HW-Komponenten.....	38
4.3	Verwendete Softwaretools.....	44
5	Optimierung auf Spezifikationsebene.....	51
5.1	double -> float Transformation.....	51
5.2	float Bitbreitenverringerng.....	54
5.3	float -> integer Transformation.....	55
5.4	Fazit.....	59
6	Optimierung auf Quellcode- bzw. HIR-Ebene.....	61
6.1	Schleifenoptimierungen.....	61
6.2	Arrayoptimierungen.....	66
7	Optimierung auf MIR-Ebene.....	73
7.1	Überblick.....	73
7.2	Constant Propagation.....	75
7.3	Constant Folding.....	76
7.4	Copy Propagation.....	77
7.5	Common Subexpression Elimination.....	77
7.6	Dead Code Elimination.....	78
7.7	Jump Optimization.....	78
7.8	Fazit.....	78
8	Optimierung auf LIR- und Assembler-Ebene.....	81
8.1	gcc Backend Optimierungen.....	82
8.2	Weitere Optimierungen.....	84
9	Speicherhierarchie.....	89
9.1	Scratch-Pad.....	89
9.2	Cache.....	92
9.3	Vergleich Scratch-Pad und Cache.....	97
10	Ergebnisse und Diskussion.....	99
11	Zusammenfassung und Ausblick.....	105
	Anhang A: Thumb-Kernbefehlssatz.....	107
	Anhang B: Ergebnisse.....	108
	Anhang C: Iropt-Skript.....	113
	Literaturverzeichnis.....	115



# 1 Einleitung

## 1.1 Motivation

Die Zahl der eingebetteten informationsverarbeitenden Systeme (EIS) in unserem Alltag wächst immer weiter. Diese Systeme müssen immer leistungsfähiger sein. Mehr Leistung bedeutet aber meistens auch mehr Energieverbrauch und eine höhere Wärmeabgabe des EIS. Man findet EIS z.B. in einer Heizungssteuerung oder Mikrowelle. Bei dieser Art von EIS ist es nicht so schlimm, wenn der Energieverbrauch weiter ansteigt, weil das Gerät dafür in seiner Funktionalität verbessert wurde. Das Gerät ist nämlich an eine feste Stromversorgung angeschlossen. Aber zu den EIS gehören auch viele mobile Geräte, wie z.B. Handies, tragbare MP3-Player oder tragbare DVD-Player. Bei den mobilen Geräten ist der Energieverbrauch sehr wichtig. Sie benötigen wie stationäre Geräte eine Stromversorgung. Diese besteht aus Batterien oder Akkus, die aber nur eine begrenzte Kapazität besitzen. Die Höhe des durchschnittlichen Energieverbrauchs und die Kapazität des Energiespeichers bestimmen die Laufzeit des EIS, bevor der Energiespeicher (Akku) wieder aufgefüllt werden muss. Bei mobilen Geräten wünscht man sich aber neben mehr Leistungsfähigkeit auch eine längere Betriebsdauer.

Um diesen Forderungen nach höherer Leistungsfähigkeit und längerer Betriebsdauer gerecht zu werden, muss dem EIS mehr Energie zur Verfügung gestellt oder der Verbrauch gesenkt werden. Die Entwicklung größerer, leistungsfähigerer Energiespeicher würde dazu führen, dass ein EIS länger als bisher genutzt werden kann. Andererseits würde sich durch die Vergrößerung des Energiespeichers das gesamte EIS vergrößern bzw. an Gewicht zunehmen. Das widerspricht aber der Forderung nach immer kleineren EIS, die eine immer längere Laufzeit haben sollen. Diese Forderung kann nur erfüllt werden, wenn der Energieverbrauch des EIS reduziert wird. Das hätte zur Folge, dass bei gleichem Akku eine längere Betriebsdauer verfügbar wäre. Ausserdem würde sich die Wärmeabgabe des EIS verringern, seine Lebensdauer erhöhen, der Verbrauch an Primärenergieressourcen verringert und die Umwelt geschont.

Zu den mobilen EIS gehören auch mobile DVD-Player, die das Abspielen von Video-DVDs an jedem Ort ermöglichen. Die Dekodierung des komprimierten Datenstroms auf einer Video-DVD erfordert einen MPEG-2 Dekoder. Eine MPEG-Applikation ist eine datenintensive Anwendung (data-dominated application). Das bedeutet, dass ein hoher Datendurchsatz zwischen Hauptspeicher und CPU existiert, der hohe Energiekosten verursacht. Es ist wichtig, herauszufinden, wieviel Optimierungspotenzial bezüglich der Energieoptimierung einer MPEG-Applikation

möglich ist. Durch diese Analysen kann z.B. festgestellt werden, welche Weiterentwicklungen an den heute verfügbaren Techniken und Werkzeugen zukünftig vorgenommen werden sollten.

### 1.2 Ziele der Arbeit

In dieser Arbeit soll für eine Multimedia-Applikation (MPEG) das Energiesparpotenzial durch die Optimierung von Software evaluiert werden. Dafür werden Optimierungen auf unterschiedlichen Ebenen betrachtet. Dabei wird sowohl die Softwareebene als auch die Hardwareebene berücksichtigt, sofern diese mit der Softwareebene in Verbindung steht. Bei der Applikation, die in dieser Arbeit betrachtet wird, handelt es sich um einen MPEG-2 Dekoder.

Die Betrachtung der Software wird in mehrere Ebenen unterteilt: Spezifikationsebene, Quellcode- und *HIR*-Ebene, *MIR*-Ebene, *LIR*- und Assembler-Ebene. Als Beispiel für Optimierungen auf der Quellcode-Ebene sei hier [CO01] genannt, wo durch manuelle Optimierungen des C-Quellcodes bis zu 60% des Speicherplatzbedarfs eingespart werden konnten.

Auf der Hardwareebene kann man z.B. durch Verringerung der Versorgungsspannung und der Taktrate Energieeinsparungen von 58% erreichen [OIY99]. In dieser Arbeit wird auf der Hardwareebene der Einsatz unterschiedlicher Speicherhierarchien in Kombination mit unterstützender Software betrachtet.

Die Ergebnisse der in dieser Arbeit untersuchten Optimierungen werden für die einzelnen Ebenen festgehalten. Das Ziel ist, eine Kombination der Optimierungen von unterschiedlichen Ebenen zu finden, die für diese Arbeit die maximale Energieeinsparung bedeutet. Dabei werden zum einen Optimierungen benutzt, wie sie in gängigen Compilern integriert sind. Zusätzlich werden Optimierungen untersucht, die noch nicht als fester Bestandteil von Compilern existieren. Dadurch soll gezeigt werden, welches Potenzial für die Energieoptimierung eines MPEG-2 Dekoders vorhanden ist.

### 1.3 Kapitelübersicht

In Kapitel 2 werden die Grundlagen des MPEG-Standards näher dargestellt. Desweiteren wird die Arbeitsweise des betrachteten MPEG-2 Dekoders erklärt und auf wichtige Funktionen des MPEG-2 Dekoders eingegangen.

Die Grundlagen von Compilern werden in Kapitel 3 behandelt. Neben den Grundlagen gibt es auch eine Einführung zu optimierenden Compilern und speziell zu energieoptimierenden Compilern. Am Ende des Kapitels werden einige Standard-Optimierungen vorgestellt.

Kapitel 4 stellt die für die Simulation betrachtete Systemarchitektur vor. Es wird auf die Hardware, bestehend aus Prozessor und Speicher, auf den Simulationsablauf und die dafür nötige Software näher eingegangen.

Das erste Kapitel, das Optimierungen behandelt, ist Kapitel 5. Hier werden Optimierungen auf der Spezifikationsebene vorgestellt. In Kapitel 6 wird der Einsatz von Optimierungen auf der Quellcode- bzw. HIR-Ebene erklärt. Optimierungen der MIR-Ebene können in Kapitel 7 nachgelesen werden. Welche Auswirkungen Optimierungen auf der Assemblercode- und LIR-Ebene haben, zeigt Kapitel 8.

Die Nutzung verschiedener Speicherhierarchien zur Energieoptimierung wird in Kapitel 9 untersucht.

Kapitel 10 stellt eine Kombination der vorgestellten Optimierungen vor, die eine hohe Energieeinsparung ermöglicht.

Kapitel 11 fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick.



## 2 MPEG-Grundlagen

Die Abkürzung *MPEG* bezeichnet einen Standard für das Komprimieren von Videodaten und ist benannt nach der Gruppe, die diesen Standard festlegte, der Moving Picture Experts Group [MPEG01]. Der MPEG-Standard nutzt aus, dass sich aufeinanderfolgende Einzelbilder (Frames) eines Videostreams oft nur geringfügig unterscheiden. Speichert man nur die Unterschiede zwischen den einzelnen Frames, kann man eine Reduktion der Datenmenge erreichen.

In Kapitel 2.1 werden kurz einige MPEG-Standards vorgestellt und erklärt, welcher MPEG-Standard für diese Diplomarbeit eingesetzt wurde.

Die Arbeitsweise eines MPEG-Dekoders wird in Kapitel 2.2 näher betrachtet. In dem Kapitel werden auch die Grundlagen des gewählten Standards erklärt.

Kapitel 2.3 stellt dar, welche Funktionen im gewählten MPEG-Dekoder für die Betrachtung in dieser Diplomarbeit wichtig sind.

### 2.1 MPEG-Standards

Der MPEG-1 Standard wurde 1991 festgelegt. Er ist auch bekannt als ISO-Standard 11172: *Coding of moving pictures and associated audio – for digital storage media at up to about 1.5 Mbit/s* [BK95]. Er unterstützt eine Auflösung von 352 Pixel horizontal und 288 Pixel vertikal (352\*240 Pixel bei NTSC) und ist auf eine Datenübertragungsrate von 1,5 Mbit/s optimiert. MPEG-1 besteht aus den drei Komponenten: MPEG Video, MPEG Audio und MPEG System. Letzteres ist zuständig für das Zusammenführen von MPEG Video und MPEG Audio und muss gewährleisten, dass Video und Audio synchron sind. Die Datenrate von 1,5 Mbit/s teilt sich wie folgt auf: 1,1 Mbit/s für MPEG Video, 128 kbit/s für MPEG Audio und die übrige Datenrate für MPEG System [ES98].

1994 folgte ein neuer leistungsfähigerer MPEG-Standard, der MPEG-2 Standard oder auch ISO Standard 13818: *Generic coding of moving pictures and associated audio* [BK95]. Dieser Standard ermöglicht unterschiedliche Auflösungen und variable Datenübertragungsraten bis zu 100 Mbit/s, obwohl der MPEG-2 Standard bei Standard TV-Auflösung (720 \* 576 Pixel) ursprünglich nur für eine Datenübertragungsrate von 4 bis 9 Mbit/s vorgesehen war. Der MPEG-2 Standard findet zum Beispiel Verwendung bei der Video-DVD und digitalem Fernsehen. Außerdem bietet der MPEG-2 Standard mehrere Audio- und Videospuren. Der MPEG-2 Standard ist

rückwärtskompatibel zum MPEG-1 Standard, da er dessen Auflösung und Datenübertragungsrate weiterhin unterstützt.

Der MPEG-4 Standard wurde 1998 festgelegt [MPEG02]. Er ermöglicht die Datenübertragung von Videos bei gleicher Bildqualität und niedrigeren Bitraten als der MPEG-1 und MPEG-2 Standard. In dieser Diplomarbeit wird aber nicht weiter auf den MPEG-4 Standard eingegangen.

Der in dieser Diplomarbeit benutzte MPEG Video-Dekoder entspricht dem MPEG-2 Standard, da dieser Standard ein breites Anwendungsspektrum hat und als kompletter C-Quellcode zur Verfügung steht [MC02]. Er wird z.B. in DVD-Abspielgeräten eingesetzt und ist die Grundlage für den Standard beim digitalen Fernsehen [DVB02].

## 2.2 Arbeitsweise des MPEG-2 Dekoders

Im MPEG-2 Standard wird jedes einzelne Bild in viele Makroblocks aufgeteilt, von links nach rechts und von oben nach unten. Ein Makroblock wiederum ist 16\*16 Pixel groß und in 8\*8 Pixel große Blöcke unterteilt. Deshalb müssen die horizontale und vertikale Auflösung jedes Bildes ein Vielfaches von 16 Pixeln sein. Wie im MPEG-1 Standard werden im MPEG-2 Standard die Bilder im YCbCr Farbraum kodiert. Dieser besteht aus drei Komponenten:

- Y – Luminanz (Helligkeitsinformationen)
- Cb – Chrominanz Blau (Farbinformationen Blau)
- Cr – Chrominanz Rot (Farbinformationen Rot)

Das Verhältnis dieser drei Komponenten dient der Notation für den YCbCr-Farbraum: A:B:C. Im MPEG-1 und MPEG-2 Standard wird das Format 4:2:0 benutzt. Das bedeutet, dass die Helligkeits- und Farbinformationen im Verhältnis „vier zu zwei zu null“ verteilt sind. Zu den Luminanz-Blöcken (4) wird jeweils ein Chrominanz-Block je Farbe (Blau und Rot) in horizontaler Richtung (2) gespeichert und kein Block in vertikaler Richtung (0).

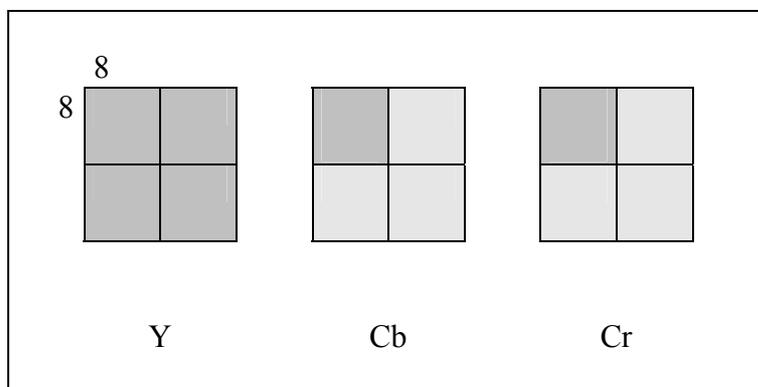
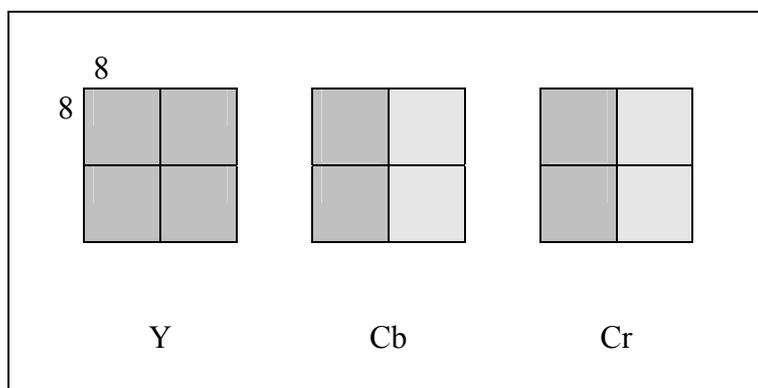


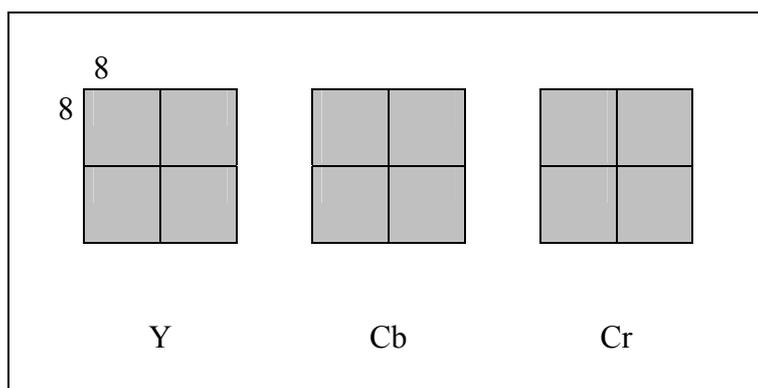
Abbildung 2.1: Makroblock im 4:2:0 Format

Für dieses Format sieht ein Makroblock wie in *Abbildung 2.1* aus. Dieser 4:2:0 Makroblock besteht daher aus vier 8\*8 Pixel großen Blöcken Luminanz, einem 8\*8 Pixel großen Block Chrominanz Blau und einem 8\*8 Pixel großen Block Chrominanz Rot. Die Reduzierung der Farbinformation kann durchgeführt werden, da das menschliche Auge empfindlicher auf eine Änderung der Helligkeit (Luminanz) reagiert als auf Farbänderungen (Chrominanz).



*Abbildung 2.2: Makroblock im 4:2:2 Format*

Der MPEG-2 Standard unterstützt im Gegensatz zum MPEG-1 Standard zusätzlich Makroblocks im Format 4:2:2 (*Abbildung 2.2*) und 4:4:4 (*Abbildung 2.3*) [BK95]. Bei Nutzung des 4:2:2 Formats hat man eine bessere Farbqualität als im 4:2:0 Format, weil zu den Luminanz-Blöcken (4) jeweils ein Chrominanz-Block je Farbe (Blau und Rot) in horizontaler Richtung (2) und jeweils ein Chrominanz-Block in vertikaler Richtung (2) gespeichert wird.



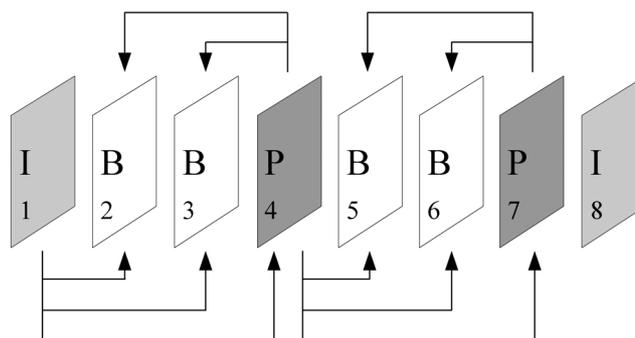
*Abbildung 2.3: Makroblock im 4:4:4 Format*

Die beste Qualität erreicht man mit dem 4:4:4 Format [ES98], bei dem zu den vier Luminanz-Blöcken je vier Chrominanz-Blöcke für Blau und für Rot gespeichert werden.

Um die Arbeitsweise eines MPEG-2 Dekoders verstehen zu können, muss man erst einmal wissen, welche unterschiedlichen Bildtypen existieren. Im MPEG-2 Standard sind wie im MPEG-1 Standard vier verschiedene Bildtypen festgelegt. Diese vier Bildtypen haben die Bezeichnungen: I-Pictures, P-Pictures, B-Pictures und D-Pictures [ES98].

- I-Pictures (intra-pictures) sind vollständige Bilder, die nicht auf Grundlage anderer Bilder berechnet werden.
- P-Pictures (predicted pictures) sind Bilder, die durch Bewegungskompensation (motion compensation) aus vorhergegangenen I-Pictures oder P-Pictures berechnet werden.
- B-Pictures (bidirectionally predicted pictures) sind Bilder, die durch Bewegungskompensation aus vorhergegangenen und/oder nachfolgenden I-Pictures oder P-Pictures berechnet werden. B-Pictures sind keine Grundlage zur Berechnung anderer Bildtypen und können deshalb stärker komprimiert werden.
- D-Pictures (DC-coded Pictures) sind, wie I-Pictures, vollständige Bilder. Sie dienen zur Realisierung des sichtbaren schnellen Vorlaufs oder Rücklaufs eines MPEG-2 Videostreams und können nicht mit den anderen Bildtypen zusammen verwendet werden. Deshalb wird hier nicht näher auf die D-Pictures eingegangen.

Die Abhängigkeiten zwischen den drei Bildtypen I-Pictures, P-Pictures und B-Pictures sind in *Abbildung 2.4* dargestellt.

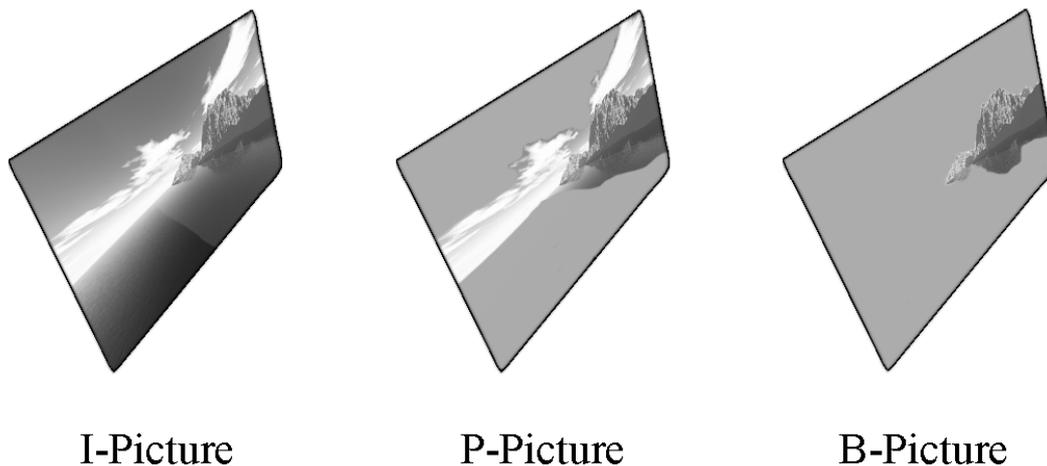


*Abbildung 2.4: Abhängigkeiten zwischen den Bildtypen einer Video-Sequenz[BK95]*

Man kann gut erkennen, dass I-Pictures nur als Grundlage für die Berechnung anderer Bildtypen dienen und dass sie selbst aus keinen anderen Bildtypen errechnet

werden. P-Pictures werden nur aus den vorhergehenden I-Pictures oder P-Pictures berechnet. Deutlich zu erkennen ist, dass die B-Pictures aus vorhergehenden und nachfolgenden I-Pictures oder P-Pictures berechnet werden und dass sie selbst keine Grundlage für eine weitere Berechnung von B-Pictures oder P-Pictures sind.

*Abbildung 2.5* verdeutlicht den unterschiedlichen Informationsgehalt der verschiedenen Bildtypen.



*Abbildung 2.5: Informationsgehalt der verschiedenen Bildtypen*

In [BK95] wird in einem Beispiel die durchschnittliche Größe der I-Pictures, P-Pictures und B-Pictures einer Videosequenz angegeben. I-Pictures haben demnach eine Größe von 156 kbits, P-Pictures belegen 62 kbits und B-Pictures nur 15 kbits für das in [BK95] genannte Beispiel. Die Datenmenge eines P-Pictures kann also nur ca. 40% der Datenmenge eines I-Pictures entsprechen. Ein B-Picture kann nur ca. 10% der Datenmenge eines I-Pictures belegen.

In *Abbildung 2.6* ist ein Blockschaltbild eines MPEG-Dekoders zu sehen. Die Daten, die dem MPEG-2 Dekoder zu Beginn vorliegen, sind nach Huffman kodiert. Eine Huffman-Kodierung basiert auf der Häufigkeitsverteilung der zu kodierenden Werte [MA99].

Der erste Schritt des MPEG-2 Dekoders besteht darin, die vorliegenden Daten nach Huffman zu dekodieren (VLD<sup>1</sup>). Die dekodierten Daten sind jetzt quantisierte Koeffizienten der diskreten Kosinus Transformation (DCT<sup>2</sup>). Die DCT-Koeffizienten entstanden vorher beim Kodiervorgang mit Hilfe der diskreten Kosinus Transformation, die die Zahlenwerte für Helligkeit und Farbe jedes einzelnen Pixels vom Zeit- in den Frequenzbereich transformiert hat. Der oben links stehende DCT-Koeffizient der 8\*8 großen DCT-Koeffizientenmatrix wird DC-Komponente oder Gleichanteil genannt. Je weiter man sich in der DCT-Koeffizientenmatrix nach rechts und nach unten bewegt, desto höher werden die zugehörigen Frequenzen. Da das

<sup>1</sup> VLD – variable length decoding

<sup>2</sup> DCT – discrete cosine transformation

menschliche Auge niedrige Frequenzen besser wahrnehmen kann, können die höheren Frequenzen stärker komprimiert werden. Die Komprimierung wird durch die Quantisierung erfüllt [TEC02]. Quantisiert wurden die Werte beim Kodiervorgang mit Hilfe einer Quantisierungsmatrix. Eine Quantisierungsmatrix besteht aus 8\*8 Feldern. Diese Größe entspricht der Größe der in den Makroblocks enthaltenen Blöcke und der DCT-Koeffizientenmatrix. Jedes Feld enthält einen ganzzahligen Wert im Bereich von 1 bis 255. Beim Kodiervorgang werden die jeweils zuvor ermittelten 64 DCT-Koeffizienten – für einen 8\*8 Pixel großen Block - durch den Wert im entsprechenden Feld der Quantisierungsmatrix dividiert und auf die nächste ganze Zahl gerundet.

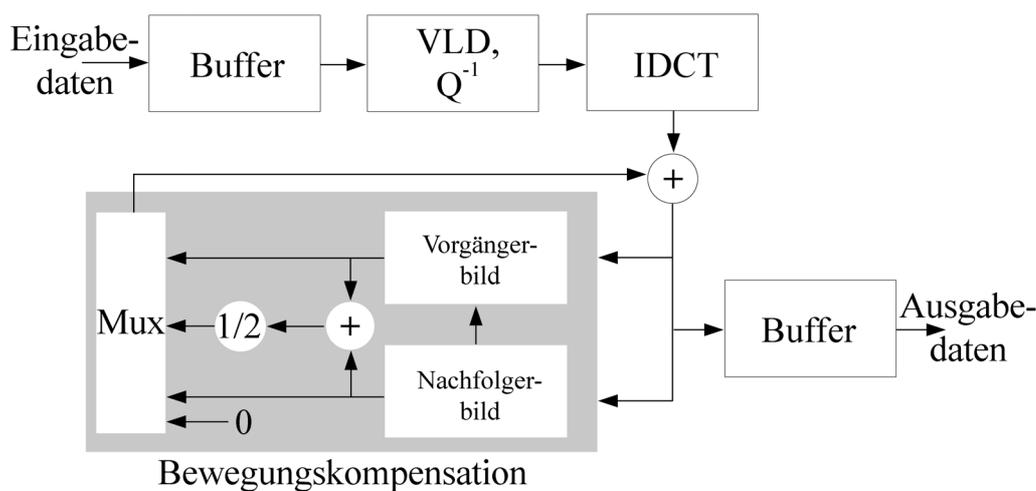


Abbildung 2.6: Blockschaltbild eines MPEG-Dekoders

Sind z.B. alle Werte in der Quantisierungsmatrix gleich 1, würden die DCT-Koeffizienten nicht verändert werden, man würde also durch die Quantisierung die Kompression nicht beeinflussen.

Eine typische Quantisierungsmatrix enthält aber nicht nur den Wert 1 (siehe *Abbildung 2.6*). Wie man sieht, steigen die Werte von links oben nach rechts unten hin an. Das hängt damit zusammen, dass man die DCT-Koeffizienten, die weiter unten rechts stehen, also die mit der höheren Frequenz, nicht so stark gewichtet.

Deren neuer quantisierter Wert wird also durch die Division kleiner als der Wert, der weiter links oben steht. Da die Quantisierungsmatrix beliebige Werte enthalten kann, wird diese in den Daten mit abgespeichert, sodass dieselbe Quantisierungsmatrix dem Dekoder zur Verfügung steht. Der MPEG-2 Dekoder macht jetzt nichts anderes, als die jeweiligen Werte mit den Werten aus der Quantisierungsmatrix zu multiplizieren. Die Werte, die man jetzt erhält, sind die DCT-Koeffizienten, aus denen man nun die ursprünglichen Bilddaten über die inverse diskrete Kosinus Transformation (IDCT<sup>3</sup>). und anschließender Bewegungskompensation berechnen kann. Die Genauigkeit ist natürlich abhängig

<sup>3</sup> IDCT – inverse discrete cosine transformation

von der gewählten Quantisierung, da hier durch hohe Werte in der Quantisierungsmatrix auch eine hohe Ungenauigkeit entstehen kann. Es werden zwei verschiedene Quantisierungsmatrizen eingesetzt. Für I-Pictures wird eine andere Quantisierungsmatrix verwendet als für P- und B-Pictures. Die Quantisierungsmatrix in *Abbildung 2.7* wird für I-Pictures benutzt.

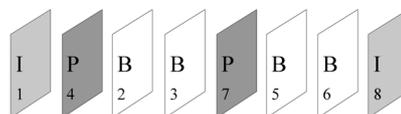
8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

*Abbildung 2.7: Beispiel für eine Quantisierungsmatrix [ES98]*

Die Bewegungskompensation erfasst beim Kodieren eines Videostreams, ob ein Makroblock des aktuellen Bildes auch im Nachfolgebild, gegebenenfalls an anderer Position enthalten ist. Dann muss nur der Bewegungsvektor für diesen Makroblock gespeichert werden und man erreicht dadurch eine weitere Kompression.

### 2.2.1 Beispiel einer Videostream-Dekodierung

Zur Veranschaulichung soll folgender Videostream dekodiert werden:



*Abbildung 2.8: Kodierter Videostream*

Anhand des Bildtyps des aktuellen Bildes wird entschieden, welcher Verarbeitungsschritt für dieses Bild folgt. Handelt es sich um ein I-Picture (I1 in *Abbildung 2.8a*), besteht der nächste Schritt in der IDCT.

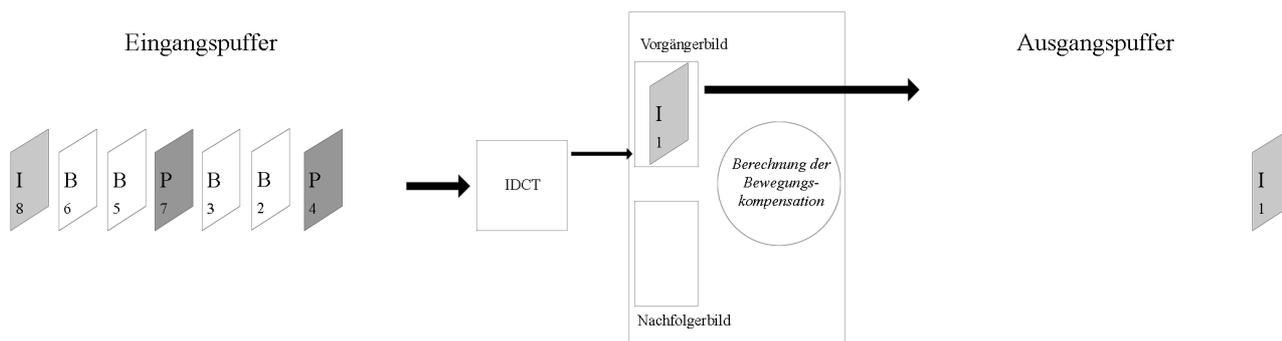


Abbildung 2.8a: Dekodierung Schritt 1

Durch das Anwenden der IDCT auf die DCT-Koeffizienten erhält man die originalen, aber gerundeten, Bildwerte. Jetzt hat der MPEG-2 Dekoder ein Einzelbild mit fast originalen Bilddaten vorliegen. Eine Bewegungskompensation ist für dieses Bild nicht notwendig. Das Bild kann jetzt in den Ausgangspuffer geschrieben werden und wird als Vorgängerbild für die Bewegungskompensation gespeichert.

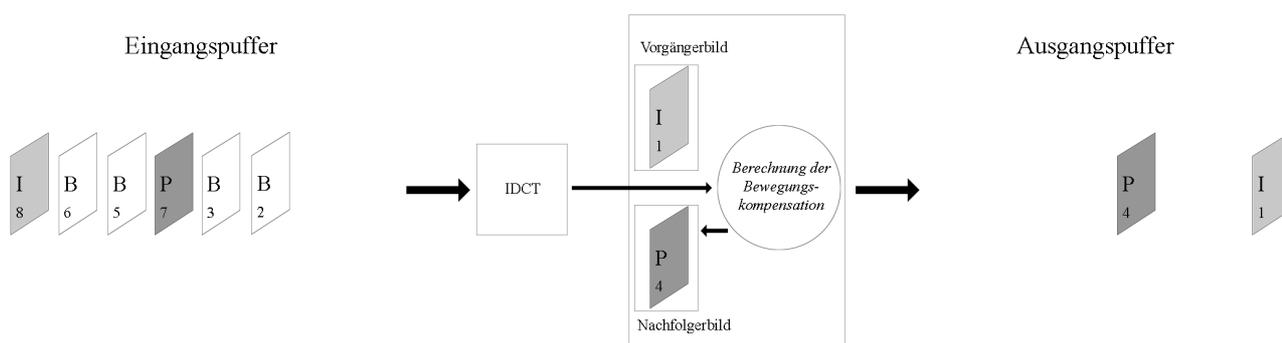


Abbildung 2.8b: Dekodierung Schritt 2

Bei einem P-Picture (P4 in *Abbildung 2.8b*) ist zusätzlich zur IDCT eine Berechnung der Bewegungskompensation notwendig. Das daraus berechnete Bild wird in den Ausgangspuffer geschrieben und als Nachfolgebild für die Bewegungskompensation festgehalten.

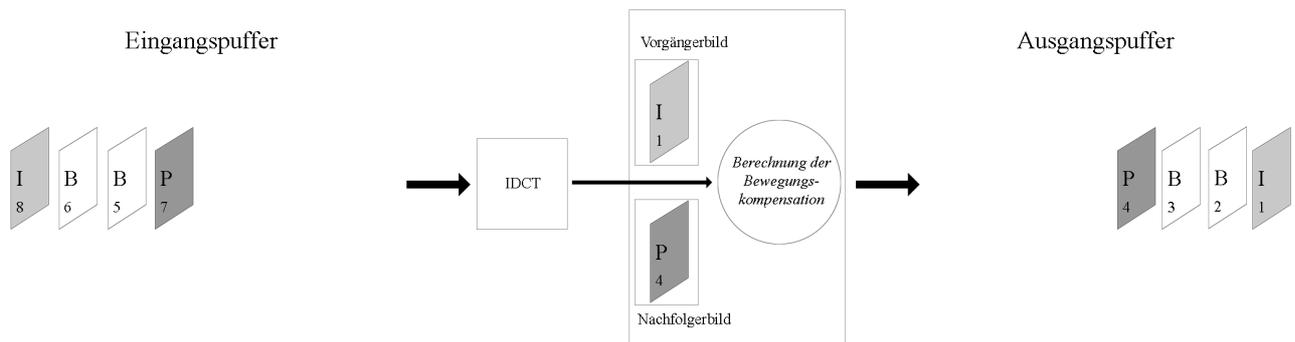


Abbildung 2.8c: Dekodierung Schritt 3

Erst jetzt werden die B-Pictures (B<sub>2</sub> und B<sub>3</sub> in *Abbildung 2.8c*) dekodiert. Das liegt daran, dass diese Bilder mittels der Bewegungskompensation aus dem vorher erwähnten I-Picture und P-Picture berechnet werden müssen. Die B-Pictures werden nur noch in den Ausgangspuffer geschrieben. Für eine weitere Verwendung in der Bewegungskompensation sind sie bedeutungslos, da sie ja keine Grundlage für die Berechnung anderer Bilder sein können.

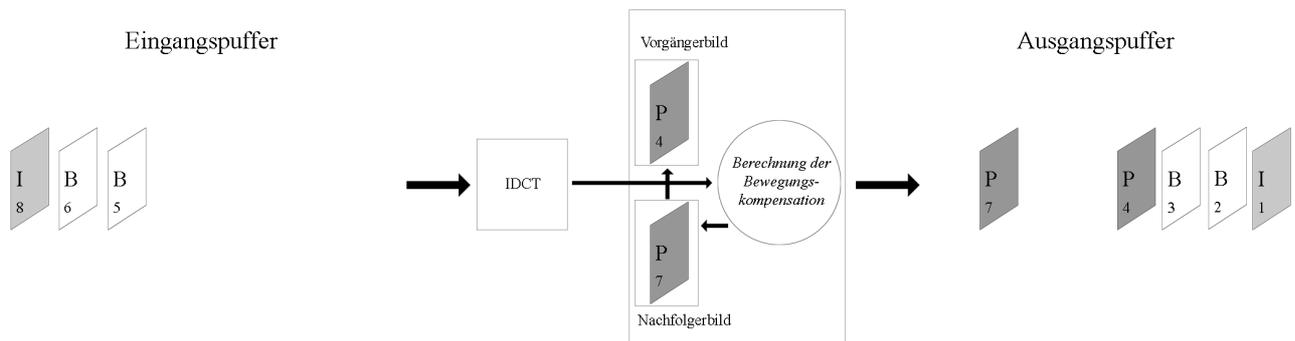


Abbildung 2.8d: Dekodierung Schritt 4

Folgt jetzt ein weiteres P-Picture (P<sub>7</sub> in *Abbildung 2.8d*), muss das P-Picture, das als Nachfolgebild für die Bewegungskompensation festgehalten wird, in den Speicher für das Vorgängerbild geschoben werden (P<sub>4</sub> in *Abbildung 2.8d*).

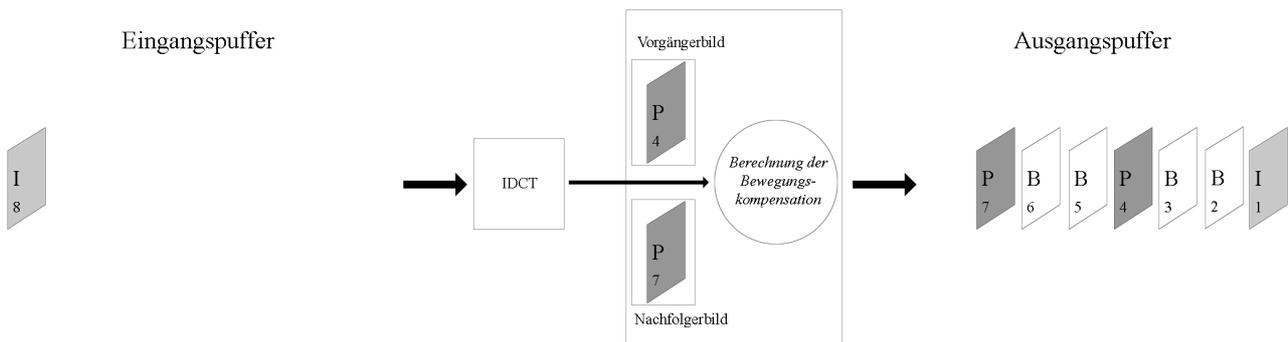


Abbildung 2.8e: Dekodierung Schritt 5

Das dort enthaltene I-Picture (I1) wird nämlich jetzt nicht mehr benötigt und das neue P-Picture muss nach erfolgreicher Dekodierung als neues Nachfolgebild gespeichert werden, um die nächsten B-Pictures (B6 und B7 in *Abbildung 2.8e*) dekodieren zu können.

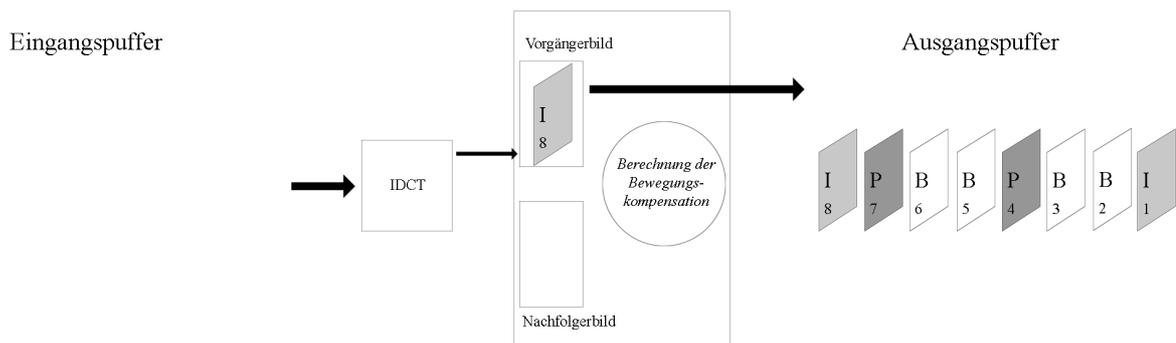


Abbildung 2.8f: Dekodierung Schritt 6

Das letzte Einzelbild in diesem Videostreambeispiel ist ein weiteres I-Picture (I8 in *Abbildung 2.8f*). Dieses I-Picture durchläuft wie jedes I-Picture die IDCT. Die für die Berechnung der Bewegungskompensation existierenden Vorgänger- und Nachfolgebilder werden nicht mehr benötigt, da für I-Pictures keine Bewegungskompensation durchgeführt werden muss. Das Nachfolgebild wird gelöscht und das I-Picture wird als neues Vorgängerbild festgehalten. Außerdem wird das I-Picture in den Ausgangspuffer geschrieben. Der Ausgangspuffer enthält jetzt die Sequenz aus dem Eingangspuffer in dekodierter Form.

Durch die unterschiedlichen Bildtypen muss nicht für jedes Bild eine Berechnung der Bewegungskompensation durchgeführt werden. Die IDCT muss aber für jedes einzelne Bild berechnet werden.

## 2.3 Hotspots im MPEG-2 Dekoder

Ein Hotspot ist der Bereich eines Programms, der den größten Anteil am Energieverbrauch hat. Da die gesamte MPEG-2 Dekodersoftware nicht erfolgreich mit der für diese Diplomarbeit benutzten Testumgebung übersetzt und ausgeführt werden konnte, konnte nicht festgestellt werden, welche Funktionen im MPEG-2 Dekoder wieviel Energie verbrauchen.

Eine Funktion, die einen großen Anteil an der Ausführungszeit eines Programms hat, verbraucht in der Regel auch mehr Energie in einem Programmdurchlauf, als eine Funktion, deren Anteil an der Ausführungszeit viel geringer ist. Um die Funktion mit dem größten Laufzeitanteil herauszufinden, wurde die MPEG-2 Software mit dem *GCC*<sup>4</sup> für Linux mit der Option „-pg“ übersetzt, um zusätzliche Profiling-Informationen in die Binärdatei mit einzukompilieren. Da in der Binärdatei zusätzliche Profiling-Informationen vorhanden sind, wird bei der Programmausführung eine zusätzliche Datei erzeugt. Diese Datei enthält alle Informationen über die letzte Programmausführung. Der Inhalt der Datei kann mit dem Tool *GProf*<sup>5</sup> angezeigt werden. Ein Auszug der Ausgabe mittels *GProf* ist in *Abbildung 2.9* zu sehen.

Flat profile:

Each sample counts as 0.01 seconds.

% time	self	seconds	name
81.72	229.78		Reference_IDCT
3.73	10.48		form_component_prediction
3.33	9.37		Add_Block
2.69	7.56		Saturate
2.60	7.30		Putbyte
2.07	5.83		store_yuv1
1.31	3.68		Clear_Block
0.62	1.75		Flush_Buffer
0.39	1.09		Decode_MPEG2_Non_Intra_Block
0.31	0.86		Show_Bits

*Abbildung 2.9: GProf Ausgabe*

Wie man in *Abbildung 2.9* und *Abbildung 2.10* sieht, hat die Funktion *Reference\_IDCT* in der MPEG-2 Software einen sehr hohen Anteil von 81% an der Ausführungszeit. Nachdem der MPEG-2 Dekoder mit verschiedenen großen Eingabedaten gestartet wurde, zeigte sich, dass die Funktion *Reference\_IDCT* je nach Eingabedaten zwischen 65% und 85% der Ausführungszeit benötigt.

<sup>4</sup> GCC – Gnu C Compiler

<sup>5</sup> GProf – Gnu Profiler

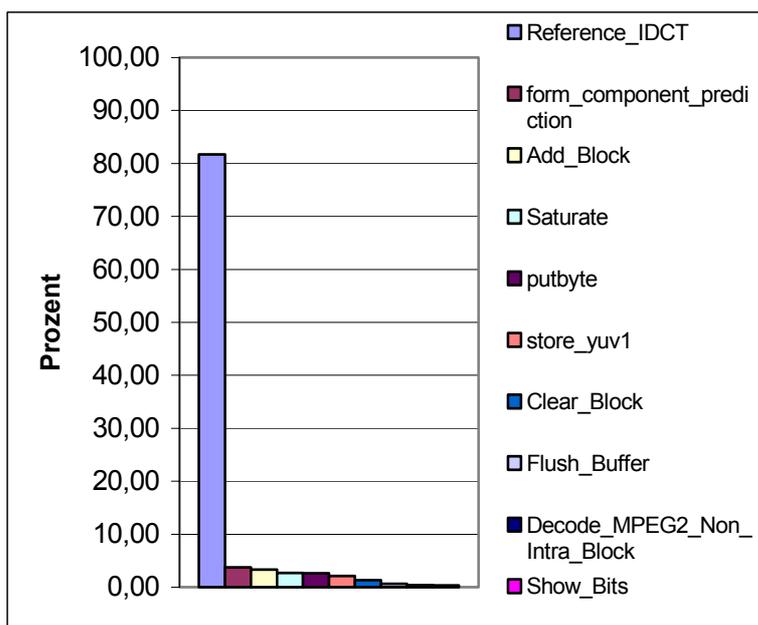


Abbildung 2.10: Anteile an Programmausführungszeit

Die Schlussfolgerung war, dass eine Funktion, die im Durchschnitt ca. 75% der Ausführungszeit für sich alleine beansprucht, auch den größten Anteil beim Energieverbrauch des gesamten Programms hat. Daher scheint es sinnvoll, die Funktion *Reference\_IDCT* in der MPEG-2 Software für diese Diplomarbeit näher zu betrachten.

Aufgrund der Tatsache, dass die gesamte MPEG-2 Dekoder-Software zum Zeitpunkt der Erstellung der Diplomarbeit nicht mit dem lehrstuhleigenen energieoptimierenden Compiler *encc* kompiliert werden konnte, wurde die Funktion *Reference\_IDCT* als eigenes Programm ausgelagert. Jetzt bot sich die Möglichkeit, die Funktion *Reference\_IDCT* mit dem *encc* zu kompilieren und den lehrstuhleigenen Profiler *enProfiler* zur Analyse zu benutzen. Außerdem wäre eine manuelle Optimierung der gesamten MPEG-2 Dekoder-Software zu aufwendig und daher nicht sinnvoll.

### 2.3.1 Reference\_IDCT

Die Funktion *Reference\_IDCT* der MPEG-2 Software sieht folgendermaßen aus:

```
void Reference_IDCT(block)
short *block;
{
    int i, j, k, v;
    double partial_product;
    double tmp[64];
    for (i=0; i<8; i++)
```

```

for (j=0; j<8; j++)
{
    partial_product = 0.0;

    for (k=0; k<8; k++)
        partial_product+= c[k][j]*block[8*i+k];
    tmp[8*i+j] = partial_product;
}
for (j=0; j<8; j++)
for (i=0; i<8; i++)
{
    partial_product = 0.0;

    for (k=0; k<8; k++)
        partial_product+= c[k][i]*tmp[8*k+j];
    v = (int) floor(partial_product+0.5);
    block[8*i+j]=(v<-256) ? -256 : ((v>255) ? 255 : v);
}
}

```

Die Funktion benutzt zur Berechnung Werte vom Datentyp „double float“ aus einem zweidimensionalen Feld, der globalen Matrix `c[8][8]`, die vor Benutzung der *Reference\_IDCT* initialisiert werden muss. Dies geschieht durch die Funktion *Initialize\_Reference\_IDCT*:

```

void Initialize_Reference_IDCT()
{
    int freq, time;
    double scale;

    for (freq=0; freq < 8; freq++)
    {
        scale = (freq == 0) ? sqrt(0.125) : 0.5;
        for (time=0; time<8; time++)
            c[freq][time]=scale*cos((PI/8.0)*freq*(time+0.5));
    }
}

```

In der Funktion *Reference\_IDCT* werden zweimal drei ineinander verschachtelte for-Schleifen nacheinander ausgeführt. In den ersten drei ineinander verschachtelten for-Schleifen wird ein Feld mit 64 temporären Werten gefüllt.

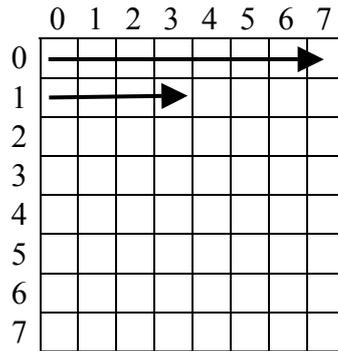


Abbildung 2.11: Feld mit Eingabedaten

Diese temporären Werte werden durch die Multiplikation der Eingabedaten mit der vorher erwähnten Matrix  $c$  und einer Addition berechnet. Die Eingabedaten sind ein  $8 \times 8$  Pixel großer Block (Abbildung 2.11).

Die Eingabedaten werden, beginnend in Zeile 0, von links nach rechts durchlaufen. Dabei wird jedes einzelne Feld mit dem entsprechenden Feld in der Zeile 0 der Matrix  $c$  multipliziert. Die Produkte werden aufsummiert und nach einem Durchlauf der Zeile als temporärer Wert festgehalten. Danach wird die Zeile 0 der Eingabedaten erneut durchlaufen, nur dass dieses Mal die einzelnen Felder mit den entsprechenden Feldern in der Zeile 1 der Matrix  $c$  multipliziert werden.

So wird jede Zeile acht Mal durchlaufen, da die Matrix  $c$  acht Zeilen hat. Danach wird in Zeile 1 der Eingabedaten gewechselt, usw.. So entsteht eine neue  $8 \times 8$  Matrix mit 64 temporären Werten.

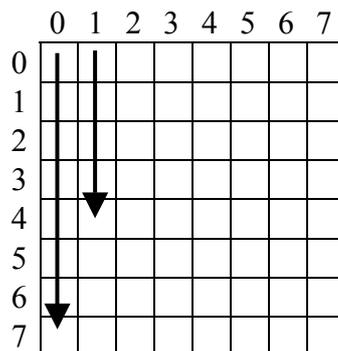


Abbildung 2.12: Feld mit temporären Daten

Die temporären Werte werden in den folgenden drei verschachtelten for-Schleifen in anderer Reihenfolge (Abbildung 2.12) mit der Matrix  $c$  multipliziert und addiert. Zur Berechnung des endgültigen Wertes wird der aktuelle Wert gerundet, auf einen Bereich von  $-256$  bis  $255$  beschränkt (Sättigungsarithmetik) und als endgültiger Wert im ursprünglichen Eingabedatenfeld abgespeichert.

### 2.3.2 Fast\_IDCT

Die Funktion *Fast\_IDCT* stellt eine Alternative zur Funktion *Reference\_IDCT* dar. In der MPEG-2 Dekoder-Software sind beide Funktionen enthalten. Man hat die Möglichkeit beim Aufruf der MPEG-2 Dekoder-Software per Kommandozeilenparameter zu entscheiden, welche der beiden Funktionen für die Berechnung der IDCT benutzt werden soll. Standardmäßig wird die Funktion *Fast\_IDCT* benutzt. Die Funktion besteht aus 2 Schleifen, die jeweils die Funktionen *idctrow* (horizontale IDCT) bzw. *idctcol* (vertikale IDCT) aufrufen. Der größte Unterschied zur Funktion *Reference\_IDCT* ist, dass die Funktion *Fast\_IDCT* mit Variablen vom Datentyp „integer“ rechnet. Dies führt aber auch dazu, dass die Genauigkeit der *Fast\_IDCT* eingeschränkt ist. Bei den berechneten Werten ergeben sich leichte Unterschiede zu den Berechnungen mittels der Funktion *Reference\_IDCT*.

```
void Fast_IDCT(block)
short *block;
{
    int i;

    for (i=0; i<8; i++)
        idctrow(block+8*i);

    for (i=0; i<8; i++)
        idctcol(block+i);
}
```

Die Funktion *idctrow* berechnet die IDCT zeilenweise. Sie bekommt als Funktionsparameter einen Zeiger auf eine Zeile der Eingabedaten übergeben.

```
static void idctrow(blk)
short *blk;
{
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;
    ...
}
```

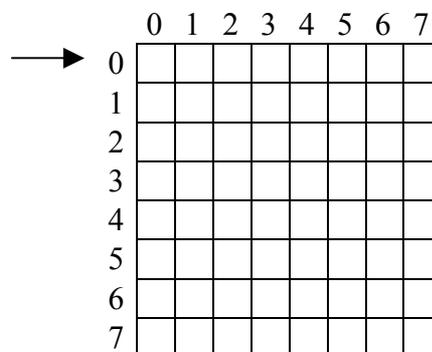


Abbildung 2.13: Zeilenweise Berechnung per *idctrow*

Ausgehend von diesem Zeiger werden alle 8 Elemente in der Zeile berechnet.

Bei der Funktion *idctcol* wird die IDCT spaltenweise berechnet. Die Funktion bekommt einen Zeiger auf eine Spalte der Eingabedaten übergeben. Die Daten sind durch die vorher ausgeführte Funktion *idctrow* schon verändert worden.

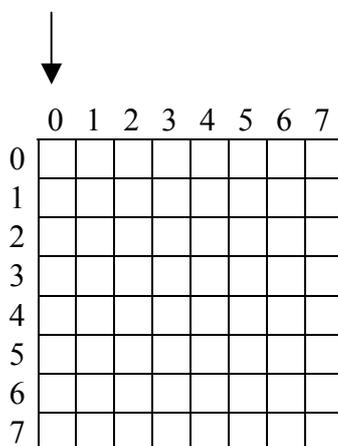


Abbildung 2.14: Spaltenweise Berechnung per *idctcol*

```
static void idctcol(blk)
short *blk;
{
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;
    ...
}
```

Vor der Nutzung der Funktion *Fast\_IDCT* muss zur Initialisierung die Funktion *Initialize\_Fast\_IDCT* aufgerufen werden:

```
void Initialize_Fast_IDCT()
{
    int i;

    iclp = iclip+512;
    for (i= -512; i<512; i++)
        iclp[i] = (i<-256) ? -256 : ((i>255) ? 255 : i);
}
```

Diese Funktion füllt ein Feld vom Datentyp „short integer“ mit 1024 Werten im Bereich von -256 bis +255. Diese Werte werden zur Berechnung der vertikalen IDCT mit der Funktion *idctcol* benötigt und realisieren eine Sättigungsarithmetik.

Die Funktionen *Reference\_IDCT* und *Fast\_IDCT* sind die wichtigsten Funktionen der MPEG-2 Dekoder-Software für diese Diplomarbeit.

Im nächsten Kapitel wird der Aufbau von energieoptimierenden Compilern behandelt. Dazu werden zuerst die Grundlagen anhand eines einfachen Compilers erklärt. Anschließend wird auf 2 Modelle für optimierende Compiler und die Klasse der energieoptimierenden Compiler eingegangen. Im letzten Teil des Kapitels werden einige Standardoptimierungen vorgestellt.



## 3 Aufbau von energieoptimierenden Compilern

Compiler sind Programme, die ein anderes Programm von einer Sprache in eine andere Sprache übersetzen. Der hier benutzte *energy aware c compiler (encc)* übersetzt Programme von der Sprache C in die Maschinsprache für den ARM-Prozessor. Compiler können unterschiedliche Optimierungsziele haben. In dieser Diplomarbeit wird ein energieoptimierender Compiler benutzt.

Kapitel 3.1 stellt kurz die Grundlagen von Compilern dar, indem die Struktur eines einfachen Compilers erklärt wird.

Optimierende Compiler werden in Kapitel 3.2 behandelt. Es werden 2 verschiedene Modelle für optimierende Compiler vorgestellt. Außerdem werden die jeweiligen Vor- und Nachteile der beiden Modelle behandelt.

In Kapitel 3.3 wird näher auf die Klasse der energieoptimierenden Compiler eingegangen und es werden Erklärungen zu Energiemodellen gegeben.

Zum Abschluss werden in Kapitel 3.4 einige Standardoptimierungen vorgestellt. Die Optimierungen werden anhand von Beispielen erklärt.

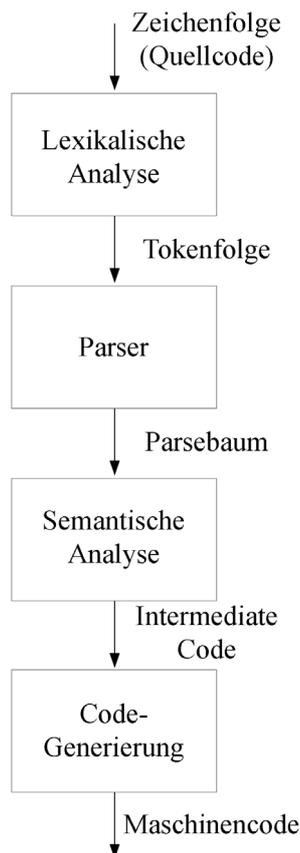
### 3.1 Compiler-Grundlagen

Ein Compiler muss für die Übersetzung eines Programms von einer Hochsprache in die Maschinsprache nicht selber auf der Zielarchitektur lauffähig sein. So ist es z.B. möglich, mit einem Compiler, der auf einer Sun Sparcstation unter Solaris läuft, C-Quellcode für die Ausführung auf einem Intel 80x86-kompatiblen Rechner und einem Linux-Betriebssystem zu übersetzen. Diese Compiler nennt man *Crosscompiler*. Der Programmierer entwickelt sein Programm in einer von ihm bevorzugten Programmiersprache. Dabei ist es egal, auf welcher Zielarchitektur das Programm später ausgeführt werden soll. Wichtig ist nur, dass ein Compiler existiert, der das Programm von der gewählten Hochsprache in die Maschinsprache für die Zielhardware übersetzen kann.

Ein Compiler besteht aus mehreren Phasen (*Abbildung 3.1*), die beim Kompilieren eines Programms durchlaufen werden müssen [MU97]:

- **Lexikalische Analyse:**

Hier wird der Quelltext des zu kompilierenden Programms analysiert. Die einzelnen Zeichen des Programms werden zu Tokens zusammengefasst, die dem Vokabular der Hochsprache des Quellcodes entstammen. Gibt es für eine Folge von Zeichen kein gültiges Token im Vokabular der Hochsprache, werden Fehlermeldungen erzeugt.



*Abbildung 3.1: Struktur eines einfachen Compilers [MU97]*

- **Parser:**

Die Folge von Tokens aus der vorherigen Phase wird analysiert. Mittels der Tokens wird eine intermediate representation, eine Zwischendarstellung, in Form einer Baumstruktur erstellt, den sogenannten Parsebaum. Außerdem wird eine Symboltabelle erzeugt, die alle Variablen des Programms und deren Eigenschaften enthält. Auch in dieser Phase werden bei auftretenden Fehlern entsprechende Fehlermeldungen generiert.

- **Semantische Analyse:**

Anhand des Parsebaums und der Symboltabelle der vorherigen Phase wird überprüft, ob das Programm die semantischen Voraussetzungen der Programmiersprache erfüllt, z.B. ob alle benutzten Variablen auch deklariert und vom korrekten Datentyp sind. Nach dieser Phase hat man eine semantisch korrekte Zwischendarstellung (*intermediate representation – IR*) und Symboltabelle. Bei Fehlern wird auch hier eine Fehlermeldung generiert.

- **Code-Generierung:**

In dieser Phase wird anhand des Codes der Zwischendarstellung der für die Zielarchitektur korrekte Objekt-Code, z.B. Assembler-Code generiert.

Der Assembler-Code muss jetzt von einem weiteren Tool verarbeitet werden, dem Assembler. Der Assembler übersetzt den Assembler-Code in Objekt-Code. Um eine auf der Zielhardware ausführbare Datei zu erhalten, muss der Objekt-Code des Programms, der auch aus mehreren Dateien bestehen kann, mit eventuell benötigten Bibliotheksfunktionen zusammengeführt werden. Diesen Vorgang nennt man Linken und er wird von einem Linker durchgeführt.

Der hier betrachtete Compiler benötigt zusätzlich einen Assembler und einen Linker, da diese Phasen nicht im Compiler enthalten sind.

## 3.2 Optimierende Compiler

Mögliche Optimierungsziele von optimierenden Compilern sind:

- **Programmgröße**

Ein Compiler mit dem Optimierungsziel Programmgröße versucht die Größe der Binärdatei zu minimieren, damit sie bei der Ausführung auf der Zielhardware möglichst wenig Speicherplatz benötigt.

- **Geschwindigkeit**

Soll das Programm auf der Zielhardware in möglichst kurzer Zeit ausgeführt werden, wählt man einen Compiler, der das Optimierungsziel Geschwindigkeit verfolgt.

- **Energie**

Ist das Optimierungsziel des Compilers Energie, will man erreichen, dass das Programm während der Ausführung auf der Zielhardware möglichst wenig Energie verbraucht.

Die Wahl des Optimierungsziels des Compilers ist heutzutage im allgemeinen über eine auszuwählende Option möglich. Für die unterschiedlichen Optimierungsziele muss also nicht jeweils ein eigener Compiler existieren.

Es existieren hauptsächlich 2 Modelle [MU97] für optimierende Compiler (*Abbildung 3.2*). Im Modell (a) in *Abbildung 3.2* wurde die Phase Code-Generierung wieder in 3 Phasen aufgeteilt. Die erste der 3 neuen Phasen ist der Translator, der einen *low-level intermediate code* erzeugt. Der *low-level intermediate code* oder die *low-level intermediate representation* (LIR) ist eine Zwischendarstellung in einer Pseudo-Maschinensprache, die aber schon abhängig von der Zielarchitektur ist [MU97]. Z.B. können Variablen schon als Register oder Speicheradressen dargestellt werden.

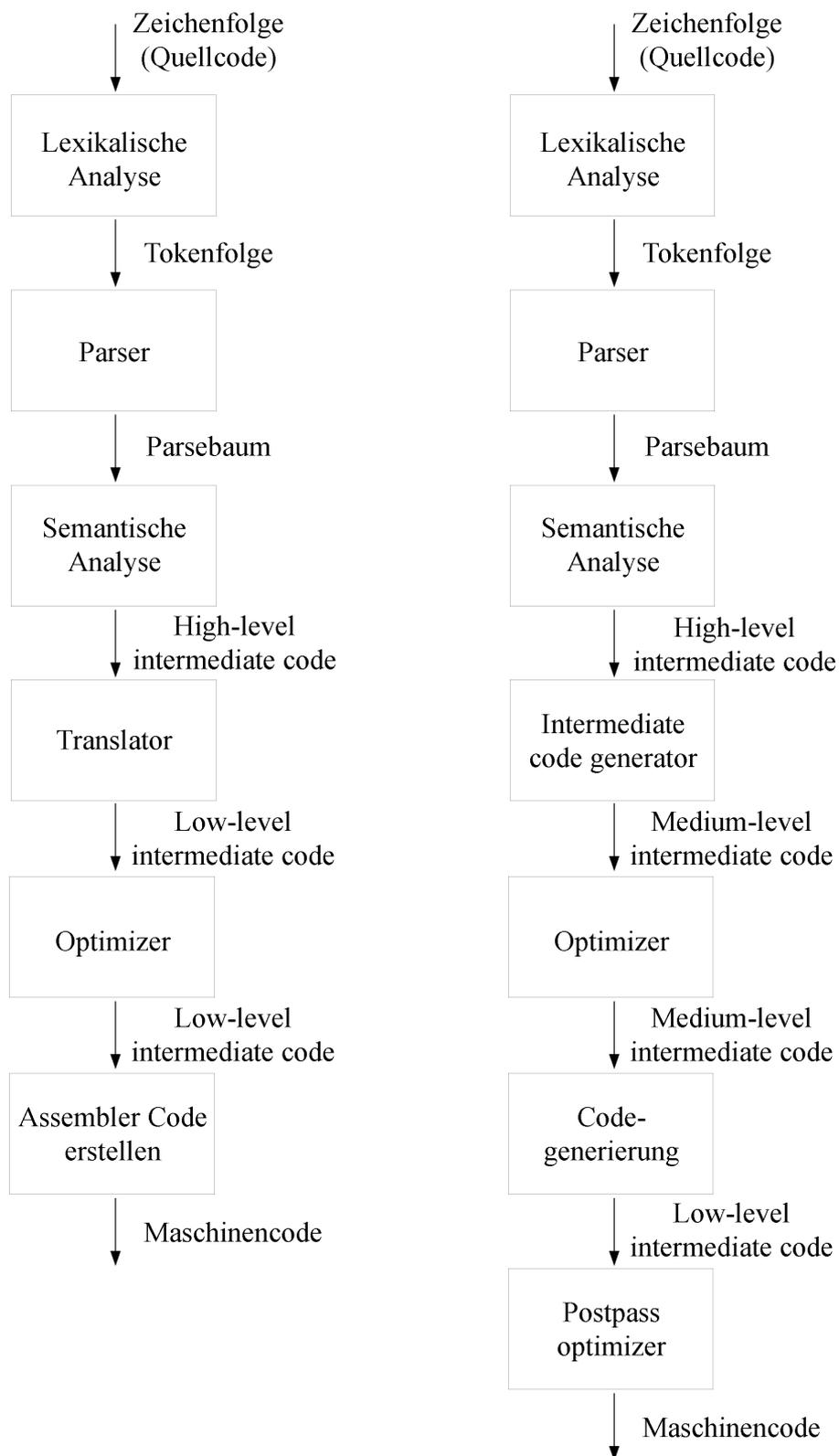
Die nächste Phase bildet der Optimizer. Er führt Optimierungen an der vorliegenden LIR durch. Der endgültige Maschinencode entsteht in der letzten Phase, dem final assembly. Hier wird aus der optimierten LIR der Maschinencode generiert. Weil bei diesem Modell alle durchgeführten Optimierungen nur auf der LIR durchgeführt werden, hat dieses Modell auch den Namen *low-level model*.

Auch beim Modell (b) wird die letzte Phase des einfachen Compilermodells, die Code-Generierung, durch neue Phasen detailliert. In diesem Fall werden hier 4 Phasen eingeführt. Direkt nach der semantischen Analyse existiert jetzt ein *Intermediate Code Generator*, eine Stufe, die eine Zwischendarstellung (IR) erzeugt.

Bei der hier erzeugten IR handelt es sich um *medium-level intermediate code* bzw. eine *medium-level intermediate representation* (MIR), die unabhängig von der Zielarchitektur ist.

Die Optimierungen des Optimizer werden auf der MIR durchgeführt. Anschließend wird aus der optimierten MIR eine LIR generiert. Diese LIR wird in der letzten Phase noch zielarchitekturabhängig optimiert und man erhält den endgültigen Maschinencode. Dieses Modell trägt den Namen *mixed model*, weil die Optimierungen sowohl auf der MIR als auch auf der LIR durchgeführt werden.

Die Frage, die sich jetzt stellt, ist: Welches der beiden Modelle ist das bessere Modell?



(a) *low-level model*

(b) *mixed model*

Abbildung 3.2: Strukturen optimierender Compiler [MU97]

Man muss sich daran orientieren, was man mit dem Compiler erreichen will. Soll z.B. der optimierende Compiler nur für eine Zielarchitektur eingesetzt werden, bietet sich das *low-level model* an. Natürlich kann man hier auch auf das *mixed model* zurückgreifen. Das *mixed model* ist aufgrund seiner Unterscheidung von LIR und MIR Ebene aber komplexer als das *low-level model*. Genau aus diesem Grund, der Unterscheidung von LIR und MIR ist das *mixed model* besser geeignet, wenn man den Compiler nicht nur für eine Zielarchitektur erstellen will, sondern später eventuell den Compiler für eine neue Zielarchitektur portieren möchte. Dann muss man nämlich die Optimierungen auf der MIR Ebene nicht an die neue Zielarchitektur anpassen, da diese Ebene ja noch architekturunabhängig ist. Geändert werden müssen dann die beiden letzten Phasen, in denen die LIR generiert und optimiert wird. Genau diese Phasen müsste man auch bei Nutzung des *low-level model* ändern.

Ein weiterer Vorteil des *mixed model* ist, dass manche Optimierungen nur auf bestimmten Ebenen sinnvoll sein können. Optimierungen, die man nur auf MIR Ebene anwenden sollte, würden mit dem *low-level model* kaum oder nur sehr aufwendig zu realisieren sein. Es gibt auch Optimierungen, die man auf MIR Ebene oder auf LIR Ebene einsetzen kann, je nachdem, ob man das *mixed model* oder das *low-level model* gewählt hat. Genau diese Optimierungen müsste man bei einem Wechsel auf eine andere Zielarchitektur im *low-level model* erneut anpassen, bei Nutzung des *mixed model* könnte man diese Optimierungen ohne Änderungen weiterbenutzen, da man sie vorausschauend auf MIR Ebene implementiert hat.

### 3.3 Energieoptimierende Compiler

Hier soll eine bestimmte Klasse der optimierenden Compiler näher betrachtet werden, die energieoptimierenden Compiler. Energieoptimierende Compiler versuchen durch ihre Optimierungen den Energieverbrauch eines Programms bei der Ausführung auf der Zielhardware zu minimieren. Ein Compiler mit dem Optimierungsziel Geschwindigkeit kann unter Umständen durch die Optimierung der Geschwindigkeit auch einen geringeren Energieverbrauch erreichen. Dies ist ein positiver Nebeneffekt. Aber für eine effizientere Energieoptimierung benötigt man einen energieoptimierenden Compiler.

Ein energieoptimierender Compiler muss wie andere Compiler die Eigenschaften der Zielhardware berücksichtigen. Dazu gehört auch der verwendete Befehlssatz. Im Gegensatz zu anderen Compilern ist für den energieoptimierenden Compiler interessant, wieviel Energie ein bestimmter Befehl während seiner Ausführung verbraucht. Wichtig ist natürlich auch, wieviele CPU-Zyklen ein Befehl für seine Ausführung benötigt. All diese Informationen erhält der Compiler aus einem Energiemodell für die Zielarchitektur.

In dieser Diplomarbeit wird ein Energiemodell für den ARM7TDMI-Prozessor [ARM95] benutzt. Das benutzte Energiemodell wurde von *Steinke et al.* [SKWM01] entwickelt. Es setzt auf dem Energiemodell von *Tiwari* auf, das in [TI96] und [TMW94] beschrieben wird.

*Tiwari* berücksichtigt *Base-Costs* und *Inter-Instruction-Costs*. *Base-Costs* sind die Basiskosten, die für jede einzelne Instruktion entstehen. *Inter-Instruction-Costs* sind Kosten, die beim Wechsel von einer Instruktion zu einer anderen Instruktion entstehen. *Tiwari* berücksichtigt in seinem Energiemodell keine Kosten, die durch den Speicher verursacht werden.

In [SKWM01] wurde das Energiemodell dahingehend erweitert, dass auch Speicherkosten und die Kosten der Datenkodierungen auf den Bussen berücksichtigt werden. Ein entscheidender Vorteil des verbesserten Energiemodells ist die Berücksichtigung der *functional unit change costs (FU-Change-Costs)*. Damit sind Kosten gemeint, die entstehen, wenn z.B. nach einer ALU-Instruktion eine Multiplikation ausgeführt wird, wenn also die Funktionseinheit des Prozessors gewechselt wird.

Dieses Energiemodell kann im Compiler mit Hilfe einer im Compiler integrierten Kostenfunktion genutzt werden. Der Compiler kann somit „kostengünstigen“ Maschinencode erzeugen.

In [SKWM01] finden sich weitergehende Erläuterungen zu Energiemodellen.

### 3.4 Standard-Optimierungen

Wie zuvor schon erwähnt, kann es sinnvoll sein, Optimierungen nur auf bestimmten Ebenen anzuwenden. In [MU97] wird eine Unterteilung vorgestellt, auf welcher Ebene welche Optimierung sinnvoll ist. Die Ebenen werden wie folgt unterteilt:

1. Optimierungen werden auf Quellcode-Ebene oder HIR (*high-level intermediate representation*)-Ebene angewendet. Auf diesen Ebenen existieren noch Schleifenkonstrukte und Arrayzugriffe in bekannter Quellcode-Form. Beispiele für diese Optimierungen sind: *Scalar replacement of array references*, *Data-cache optimizations* (z.B. *Loop fusion*).
2. Benutzt man das *mixed model* oder das *low-level model* werden die Optimierungen auf MIR- bzw. beim *low-level model* auf LIR-Ebene angewendet. Beispiele für diese Optimierungen sind: *copy propagation*, *constant propagation*, *Dead-code elimination*, *Loop-invariant code motion*, *common subexpression elimination*, *Induction-variable strength reduction*, *constant folding*.
3. Die Optimierungen werden auf einer niedrigen Ebene, der LIR-Ebene benutzt, um hardwareabhängige Optimierungen zu implementieren. Beispiele für diese Optimierungen sind: *Branch optimization and conditional moves*, *Dead-code elimination*, *Software pipelining*, *Basic-block and branch scheduling*, *Branch prediction*.

4. Zum Zeitpunkt des Linkens können Optimierungen durchgeführt werden. Beispiele für diese Optimierungen sind: *Interprocedural Register Allocation*, *Aggregation of global references*

Detaillierte Erläuterungen der einzelnen Optimierungen lassen sich in *Muchnick* [MU97] nachlesen.

Im folgenden werden einige Beispiele von Optimierungen aus [MU97] dargestellt, die auch im weiteren Verlauf dieser Arbeit benutzt werden.

***Scalar replacement of array references:***

In dem Beispiel (a) wird der Zugriff auf das Array `C[i][j]` durch eine Variable `ct` ersetzt. Die Variable `ct` kann einem Register zugewiesen werden. Dadurch reduziert sich die Anzahl der Zugriffe auf den Speicher in der innersten Schleife.

```
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
    for (k=0;k<N;k++) {
      C[i][j]=C[i][j]
        +A[i][k]*B[k][j];
    }
  }
}
```

(a1) Original-Programm

```
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
    ct=C[i][j];
    for (k=0;k<N;k++) {
      ct=ct+A[i][k]*B[k][j];
    }
    C[i][j]=ct;
  }
}
```

(a2) nach scalar replacement

***Loop invariant code motion:***

Bei der Loop invariant code motion (LICM) werden bestimmte Teile aus einer Schleife ausgelagert. In Beispiel (b) ist eine LICM zu sehen. Während das Originalprogramm (b1) in der innersten Schleife 2 Additionen und 2 Multiplikationen ausführt, konnte durch LICM erreicht werden, dass in der innersten Schleife nur noch eine Addition ausgeführt werden muss (b2).

<pre> for (i=1;i&lt;100;i++) {   l=i*(n+2);   for (j=i;j&lt;100;j++) {     a[i][j]=100*n       +10*l+j;   } } </pre>	<pre> t1=10*(n+2); t2=100*n; for (i=1;i&lt;100;i++) {   t3=t2+i*t1;   for (j=i;j&lt;100;j++) {     a[i][j]=t3+j;   } } </pre>
(b1) Original-Programm	(b2) nach LICM

Dazu werden außerhalb der beiden Schleifen zusätzlich eine Addition und zwei Multiplikationen eingeführt, die zusätzliche Variablen benötigen. In der innersten Schleife reduziert sich dadurch die Anzahl der benutzten Variablen.

Diese beiden Optimierungen können auf Quellcode-Ebene oder HIR-Ebene angewendet werden.

Weitere sehr wichtige Optimierungen sind unter anderem *Constant Propagation*, *Constant Folding* und *Dead Code Elimination*.

<pre> b=3; c=4*b; if (c&gt;b) {   e=a+b; } else {   d=b+2; } </pre>	<pre> b=3; c=4*3; if (c&gt;3) {   e=a+3; } else {   d=3+2; } </pre>
(c1) Original-Programm	(c2) nach Constant Propagation

<pre> b=3; c=12; if (c&gt;3) {   e=a+3; } else {   d=5; } </pre>	<pre> c=12; if (c&gt;3) {   e=a+3; } else {   d=5; } </pre>
(c3) nach Constant Folding	(c4) nach Dead Code Elimination

Wie man am Beispiel (c) sehen kann, kann eine Kombination von verschiedenen Optimierungen und deren wiederholte Anwendung mehrere Zeilen Programmcode auf einen Ausdruck reduzieren.

```
c=12;
if (12>3) {
    e=a+3;
} else {
    d=5;
}
```

(c5) nach Constant Propagation

```
e=a+3;
```

(c6) endgültiger optimierter Code

Hat man z.B. zuerst eine *Constant Propagation* durchgeführt, anschließend ein *Constant Folding* angewendet und schließlich eine *Dead Code Elimination* benutzt, kann man die Optimierungen erneut in der gleichen Reihenfolge anwenden.

Man sollte nach einem Optimierungsschritt, in dem eine weitere Optimierung möglich war, evtl. vorher durchgeführte Optimierungen wiederholen. Denn jeder Optimierungsschritt kann, je nach Kombination der Optimierungen, zu neuem Optimierungspotenzial für andere Optimierungsschritte führen.

Dieses Kapitel machte mit den Grundlagen von Compilern und im Besonderen mit den Grundlagen der optimierenden Compiler vertraut. Im nächsten Kapitel wird die für diese Diplomarbeit verwendete Arbeitsumgebung vorgestellt.

## 4 Vorgehensweise

Für die Erzeugung von Assemblercode aus C-Quellcode wurde der am Informatik-Lehrstuhl 12 der Universität Dortmund entwickelte *encc* (*energy aware c compiler*) [EN02] benutzt. Der *encc* ist nur ein Bestandteil der verwendeten Testumgebung. Zu der Testumgebung gehört unter anderem auch ein Simulator, der den erzeugten Binärcode für verschiedene Hardwaremodelle ausführen kann. Um den Energieverbrauch bei verschiedenen Speicherhierarchien feststellen zu können, mussten zwei verschiedene Hardwaremodelle benutzt werden.

Kapitel 4.1 stellt die Methodik und einige Grundlagen vor.

In Kapitel 4.2 wird die für die Simulation betrachtete Hardware vorgestellt. Dazu gehören der *Atmel AT91M40400* Mikrocontroller [ATM99a], der die Nutzung von *Scratch-Pad Speicher* unterstützt, sowie der *ARM710T* [ARM7T98], der Cache als OnChip-Speicher enthält.

Die Testumgebung wird in Kapitel 4.3 näher betrachtet. Dort wird der vollständige Simulationsablauf erklärt. Durch einen Simulationslauf kann man den Energieverbrauch einer Applikation bestimmen. Alle anderen Komponenten, die zusätzlich zum *encc* zum Einsatz kommen, werden dort vorgestellt.

### 4.1 Methodik

In dieser Diplomarbeit werden unterschiedliche Ebenen betrachtet. Die oberste Ebene ist die Spezifikationsebene. Als Spezifikationsebene wird hier die Ebene bezeichnet, auf der festgelegt wird, welche Eigenschaften die zu entwickelnde Applikation haben soll. Für die Applikation wird auf dieser Ebene definiert bzw. spezifiziert, welche Anforderungen von ihr erfüllt werden müssen, welche Funktion sie erfüllen muss. Man kann eine Applikation mit Hilfe einer Spezifikationsprache wie z.B. *UML* [UML02] spezifizieren, visualisieren und dokumentieren. *UML* kann aber auch zur Entwicklung von Nicht-Softwaresystemen genutzt werden.

Die Wahl der nächsten Ebenen orientiert sich an den nach *Muchnick* [MU97] existierenden Zwischendarstellungen, die im Zusammenhang mit den optimierenden Compilern in Kapitel 3 schon vorgestellt wurden.

Daher wird als nächste Ebene die Quellcode-Ebene bzw. die *HIR*-Ebene gewählt. Auf diesen beiden Ebenen existieren noch Schleifenkonstrukte und Arrayzugriffe in bekannter Quellcodeform, z.B. in der Programmiersprache C.

Die *MIR*-Ebene stellt die nächste Betrachtungsstufe dar und besteht aus einem maschinenunabhängigen 3-Adresscode. Sie befindet sich zwischen der *HIR*- und der *LIR*-Ebene. Die *LIR*-Ebene wird zusammen mit der Assemblercode-Ebene betrachtet. Diese Ebene ist in einer Pseudo-Maschinensprache geschrieben, die schon von der Zielarchitektur abhängig ist, während es sich bei der Assemblercode-Ebene um die Maschinensprache für die Zielarchitektur handelt.

## 4.2 Betrachtete HW-Komponenten

Für die Simulation wurden 2 unterschiedliche Hardwaremodelle betrachtet: der Mikrocontroller *AT91M40400* der Firma „Atmel“, der den ARM7TDMI-Prozessor der Firma „ARM“ (*Advanced RISC Machines Ltd.*) enthält, und der ARM710T-Prozessor. Die beiden Prozessoren ARM7TDMI und ARM710T enthalten denselben ARM7T Prozessor-Core.

### 4.2.1 Atmel Mikrocontroller AT91M40400

Der Mikrocontroller AT91M40400 besitzt 4 Kbyte Onchip-Speicher, der als Scratch-Pad dient. Das Blockschaltbild des ARM7TDMI-Prozessors ist in *Abbildung 4.1* zu sehen. Der ARM7TDMI besitzt folgende Eigenschaften [ARM95]:

- 32 Bit RISC Prozessor
- ARM-Befehlssatz und Thumb-Befehlssatz
- 3stufige Pipeline: Instruction Fetch, Instruction Decode und Execute
- 31 32 Bit-Register und 6 Statusregister
- 8-Bit, 16-Bit und 32-Bit Datentypen
- 32 Bit Adress- und Datenbus
- niedriger Energieverbrauch
- 3,3 Volt Versorgungsspannung

Der Prozessor wird von ARM nicht als Chip verkauft, sondern kann nur als Core für eigene Mikrocontroller lizenziert werden.

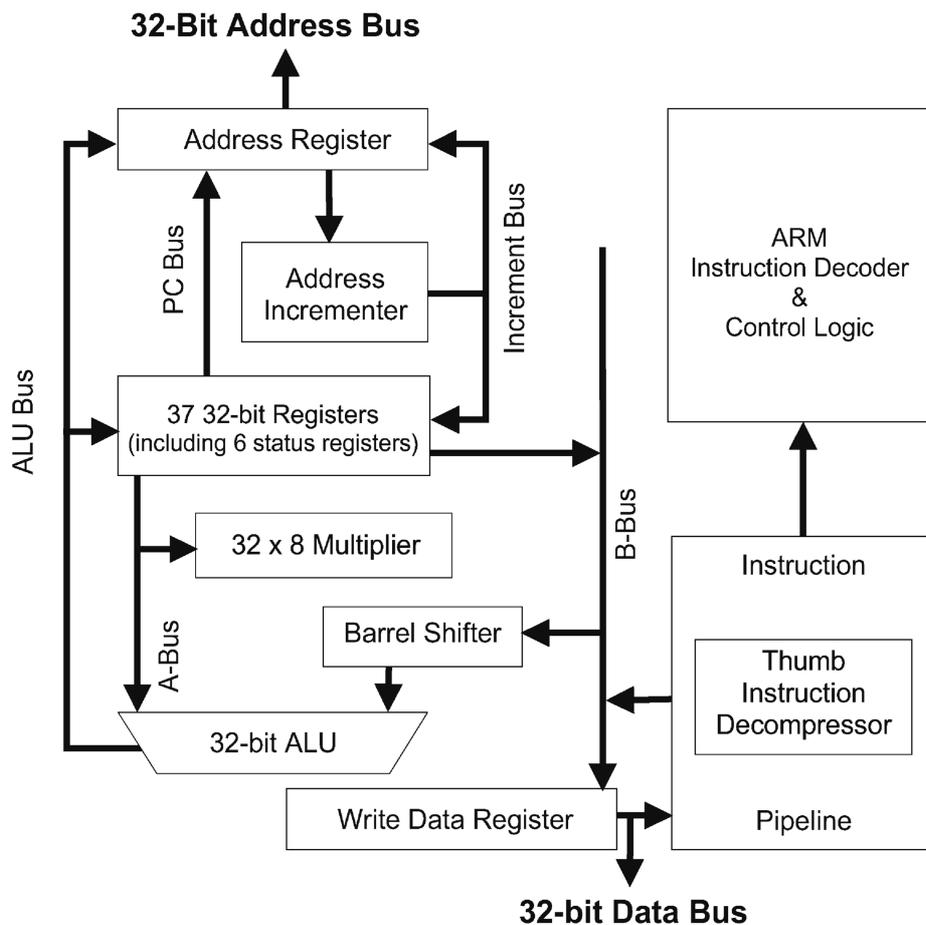


Abbildung 4.1: Blockschaltbild des ARM7TDMI-Prozessors [ATM99b]

Der ARM7TDMI Prozessor unterstützt neben dem 32-Bit ARM-Befehlssatz auch den 16-Bit Thumb-Befehlssatz. Durch die Nutzung des Thumb-Befehlssatzes ist es möglich, die Codegröße zu reduzieren und damit den Bedarf an Speicher zu reduzieren, was zu geringeren Speicherkosten führt [CT02]. Während der Programmausführung kann jederzeit mit einem speziellen Branch-Befehl zwischen ARM-Befehlssatz und Thumb-Befehlssatz hin- und hergeschaltet werden.

Die 37 Register des ARM-Prozessors sind aber nicht alle gleichzeitig nutzbar. Bei Benutzung des ARM-Befehlssatzes kann man über die Register r0 bis r15 sowie 2 zusätzliche Statusregister verfügen.

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

Im Thumb-Modus  
voller Zugriff

---

Im Thumb-Modus  
nur eingeschränkt  
nutzbar

Abbildung 4.2: Registeraufbau des ARM7TDMI-Prozessors

Im Thumb-Modus stehen nur noch die Register r0 bis r7 für vollen Zugriff zur Verfügung. Die Register r8 bis r12 können nur von speziellen Befehlen im Thumb-Modus genutzt werden: MOV, ADD und CMP. Die Register r13 bis r15 stehen für den Stackpointer, das Link-Register und den Program Counter [MI98]. Die Register r0 bis r7 haben auch die Bezeichnung *Low Registers* und die Register r8 bis r15 werden mit *High Registers* bezeichnet.

Da der Prozessor ARM7TDMI selbst keinen Onchip-Speicher besitzt, musste ein Mikrocontroller ausgewählt werden, der diesen Onchip-Speicher integriert hat. Der Atmel AT91M40400 Mikrocontroller besitzt 4 kByte Onchip-Speicher, wie im Blockschaltbild des Mikrocontrollers in *Abbildung 4.3* zu erkennen ist.

Die Eigenschaften des AT91M40400-Mikrocontrollers sind folgende:

- eingebettete ICE<sup>6</sup>-Schnittstelle (Embedded ICE)
- 4KByte On-Chip-Speicher (Scratch-Pad)
- erweiterter Interrupt-Controller mit 8-stufigen Prioritäten (AIC)
- 32 programmierbare I/O-Leitungen
- Vollprogrammierbares *External Bus Interface* (EBI)

---

<sup>6</sup> ICE - Integrated Circuit Emulation

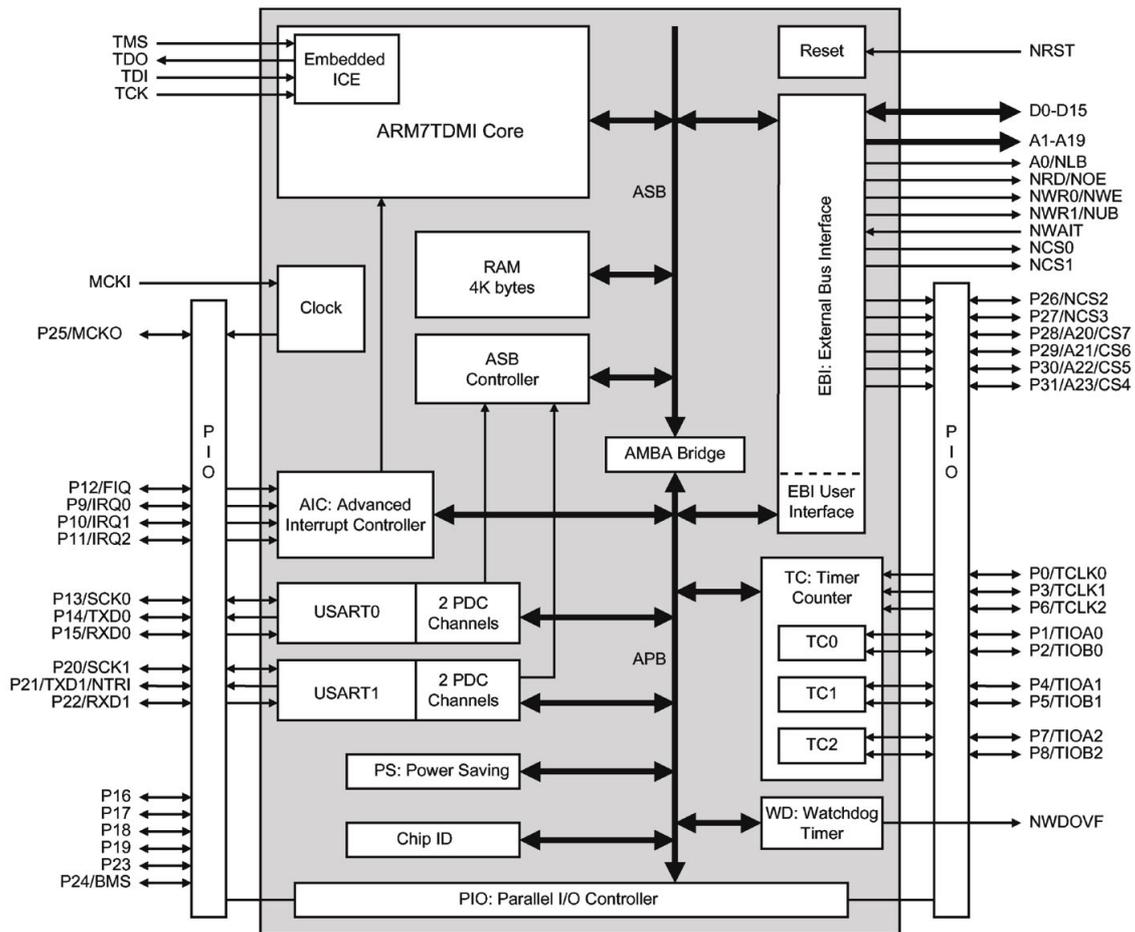


Abbildung 4.3: Blockschaltbild des AT91M40400 Mikrocontroller [ATM99a]

Dieser Mikrocontroller ist z.B. auf dem Evaluationboard AT91EB01 der Firma Atmel [ATM98] enthalten. Dieses Evaluationboard steht am Informatik-Lehrstuhl 12 der Universität Dortmund als Hardware zur Verfügung. Benutzt wurde es von *Theokaridis* [TH00] im Rahmen seiner Diplomarbeit, um den Energieverbrauch des Prozessors zu messen. Die Ergebnisse werden für die Berechnung des Energiebedarfs einer Simulation benutzt. Darauf wird später in Kapitel 4.2.4 näher eingegangen.

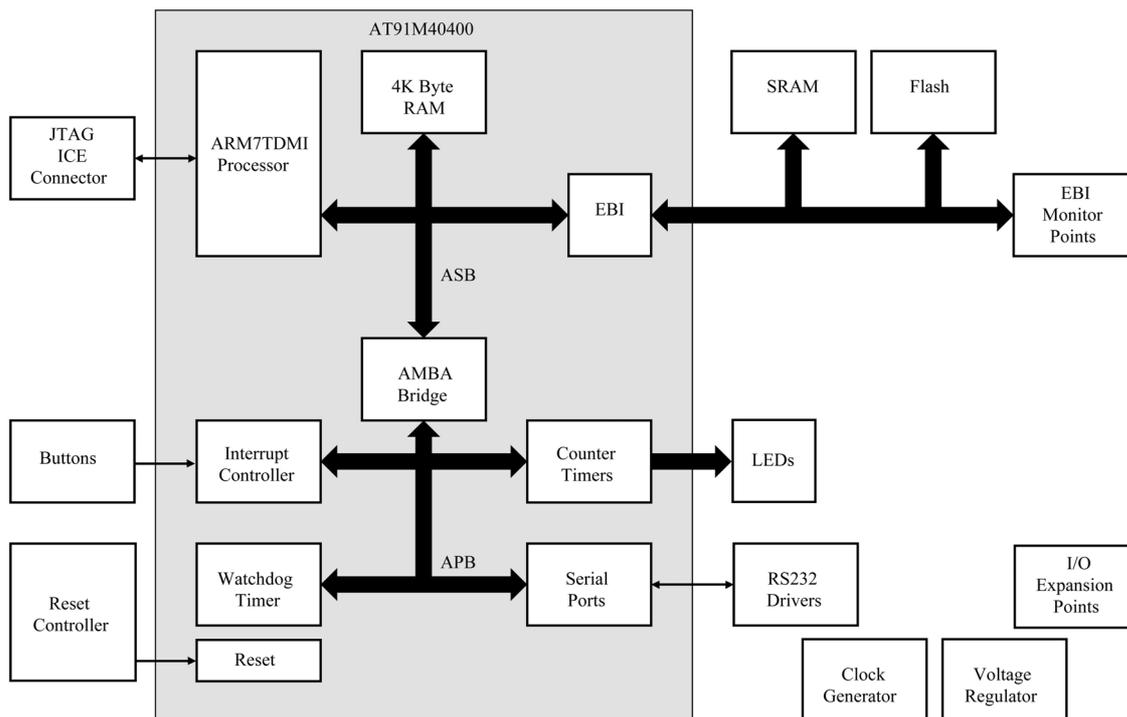


Abbildung 4.4: Blockschaltbild des AT91EB01 Evaluationboard [ATM98]

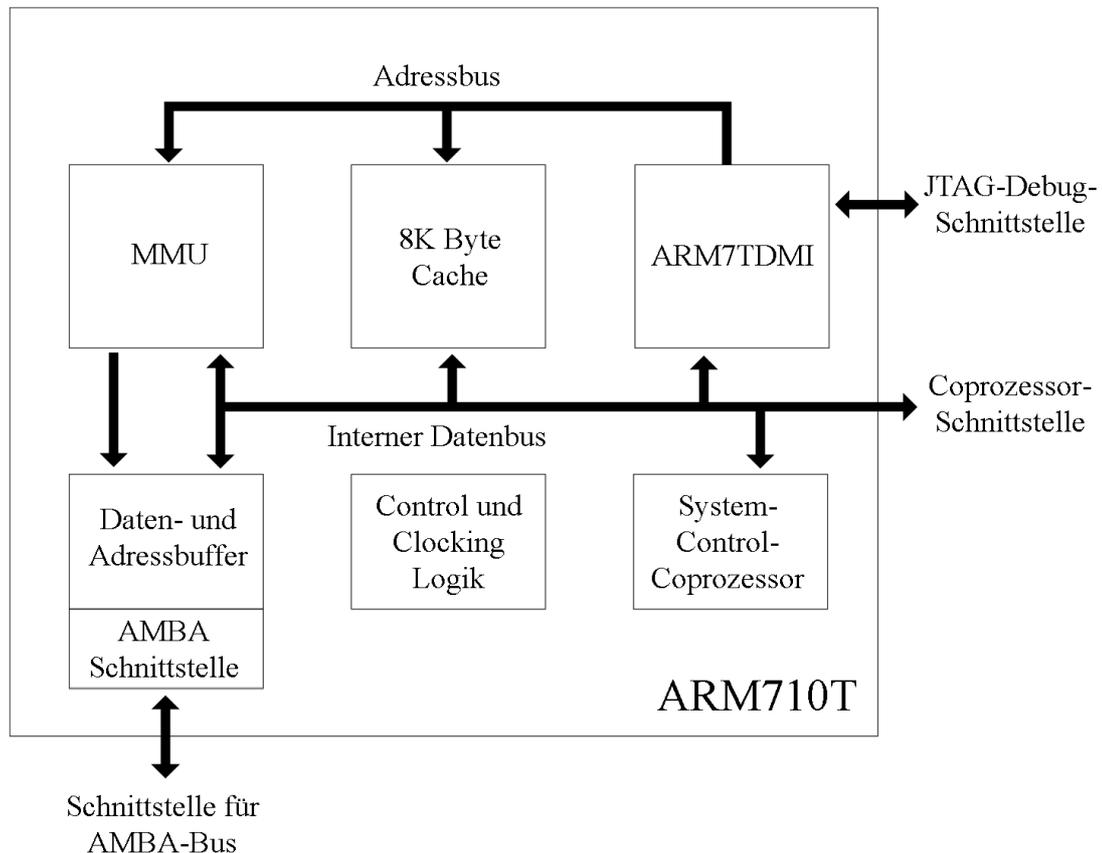
In *Abbildung 4.4* ist das Blockschaltbild des Evaluationboards AT91EB01 zu sehen. Es enthält den Mikrocontroller AT91M40400 mit dem ARM7TDMI-Prozessor und 4 Kbyte Onchip-Speicher, sowie SRAM, Flash-Speicher und Debugger-Hardware.

Das Evaluationboard besitzt folgende Eigenschaften [ATM98]:

- Atmel AT91M40400 Mikrocontroller mit ARM7TDMI-Core
- 512K Byte 16 Bit-SRAM (erweiterbar auf 2048K Byte)
- 128K Byte 16 Bit-Flash, davon sind 64K Byte frei verfügbar
- 16-Bit Datenbus und 24-Bit Adressbus
- zwei serielle Schnittstellen
- 3.3 V Versorgungsspannung
- 33 MHz Systemtaktfrequenz (kann bei Bedarf herabgesetzt werden)

## 4.2.2 ARM710T

Damit man den Energieverbrauch eines Caches simulieren kann, benötigt man ein Hardware-Modell, das einen Cache-Speicher besitzt. Als Beispiel sei hier das Prozessormodul „CMA 222 ARM710T“ der Firma „Cogent Computer Systems“ [CC98] genannt. Es besitzt den ARM710T Prozessor, der einen Onchip Cache Speicher von 8K Byte enthält. *Abbildung 4.5* zeigt das Blockschaltbild des ARM710T.



*Abbildung 4.5: Blockschaltbild des ARM710T[LE01]*

Zu den Eigenschaften des auf dem „CMA 222 ARM710T“ enthaltenen ARM710T gehören laut [LE01] und [CC98]:

- ARM7TDMI-Core mit ARM- und Thumb-Befehlssatz
- 8 Kbyte unified Cache mit 4- fach Set-Assoziativität
- „Write Through“ und „No Write Allocation“ Strategie

- 3,3 V Versorgungsspannung

Dieses Modell wurde unter anderem von *Lee* in [LE01] im Rahmen seiner Diplomarbeit benutzt, um den Energieverbrauch bei Nutzung eines Cache-Speichers zu messen.

### 4.3 Verwendete Softwaretools

Im Rahmen dieser Diplomarbeit kommen verschiedene Software-Tools zum Einsatz, die es ermöglichen, C-Quellcode zu kompilieren und später mittels Simulator auszuführen. Anschließend kann festgestellt werden, wieviel Energie das Programm während der Programmausführung benötigt

Während eines kompletten Simulationsdurchlaufes werden mehrere Softwaretools eingesetzt.

Die Softwaretools sind im einzelnen:

- Frontend: LANCE [LA01]
- Backend: *ir2asm\_ARM* [EN02]
- Assembler und Linker [ARM98]
- Simulator: ARMulator (inclusive Cache-Simulator) [ARM99]
- Analyzer: EnProfiler [SC00]

In *Abbildung 4.6* ist die Reihenfolge der Softwaretools während der 3 möglichen Simulationsdurchläufe „Typ A“, „Typ B“ und „Typ C“ zu sehen.

Das Frontend LANCE V2.0 übersetzt ANSI-C Sourcecode in eine zielhardware-unabhängige Zwischendarstellung (IR) [LA01]. Bei dieser IR handelt es sich nach [MU97] um eine *medium-level-intermediate-representation* (MIR).

Diese MIR besteht aus einem 3-Adresscode. Z.B. wird aus der C-Quellecodezeile:

```
a = b * 4 + 6;
```

folgender 3-Adresscode:

```
t1 = b_54 * 4;  
t2 = t1 + 6;
```

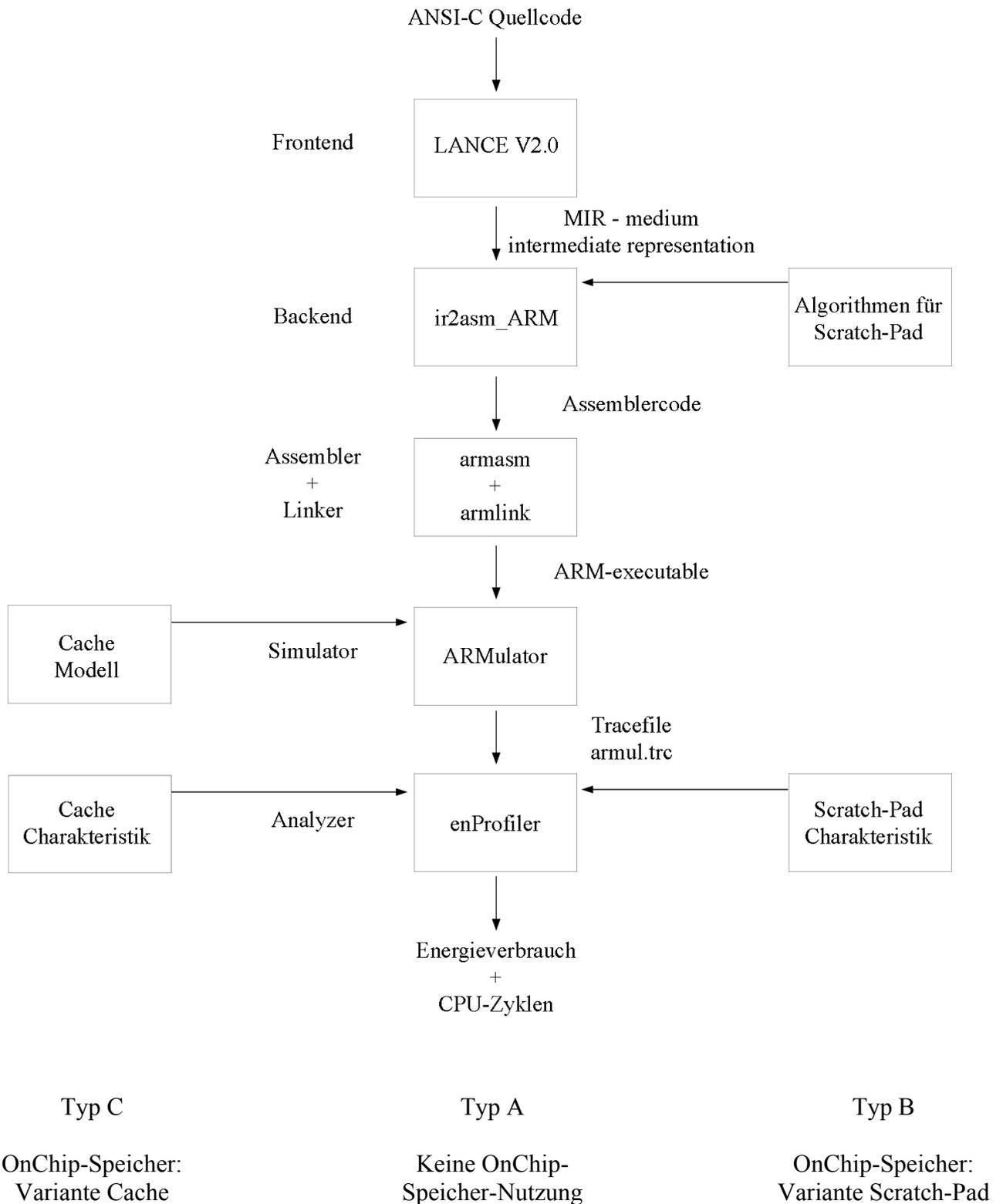


Abbildung 4.6: Simulationsabläufe

Nach Erzeugung der MIR führt das Frontend über das Skript *iropt* noch einige Standardoptimierungen wie *jump optimization*, *common subexpression elimination*, *dead code elimination*, *constant folding*, *constant propagation* und *copy propagation* durch [LA01]. Ist eine Optimierung beendet, werden vorherige Optimierungen eventuell erneut aufgerufen. Dies geschieht solange, bis keine Optimierung mehr erreicht werden kann. Die genaue Funktionsweise von *iropt* wird in Kapitel 7.1 vorgestellt. Das Frontend erzeugt durch Anwendung des *iropt*-Skriptes eine optimierte MIR.

*Hinweis:* Die folgende Beschreibung ist gültig für den Simulationsdurchlauf „Typ A“, bei dem kein OnChip-Speicher benutzt wird (*Abbildung 4.6*). Der Durchlauf „Typ A“ wurde in allen Kapiteln bis auf Kapitel 9 eingesetzt. In Kapitel 9 wurden die Durchläufe „Typ B“ und „Typ C“ benutzt.

Die optimierte MIR wird vom Backend als Eingabedatei übernommen. Das Backend wurde am Informatik-Lehrstuhl 12 der Universität Dortmund entwickelt und ist ein energieoptimierender Compiler. Der Compiler erzeugt aus der MIR 16Bit Thumb-Assemblercode für die Prozessoren ARM7TDMI und ARM710T. Er benutzt dazu das in Kapitel 3.3.1 erwähnte Energiemodell. Für das Erzeugen von Assemblercode bedient sich der Compiler einer dynamischen Kostenfunktion: hat der Compiler bei der Codeselektion die Auswahl zwischen 2 verschiedenen Maschinenbefehlsfolgen, entscheiden die im Energiemodell enthaltenen Energiekosten für diese Befehlsfolgen, welche der beiden Folgen erzeugt wird. Das Backend führt nach der Codeselektion eine Registerallokation durch und anschließend Optimierungen wie *instruction scheduling* und *register pipelining*. Das Backend gibt nach der Ausführung optimierten Assemblercode im ARM7-Thumb-Befehlssatz aus.

Der Assemblercode wird vom Assembler *armasm* und vom Linker *armlink* in eine für den ARM-Prozessor ausführbare Datei transformiert. Die Tools *armasm* und *armlink* stammen aus dem Original ARM Software Development Tool 2.50 Kit (ARM SDT 2.50 Kit) [ARM98].

Die ausführbare Datei wird mit dem Tool *ARMulator* („*ARM-Instruction-Level Simulator*“) ausgeführt, der auch dem ARM SDT 2.50 Kit entnommen ist. Der *ARMulator* bietet die Möglichkeit, während der Programmausführung eine Trace-Datei auszugeben [ARM99]. In dieser Trace-Datei sind sowohl alle ausgeführten Maschinenbefehle und Taktzyklen als auch alle Speicherzugriffe enthalten.

Im letzten Schritt des Simulationsdurchlaufes wird die vom *ARMulator* erzeugte Trace-Datei analysiert. Diese Aufgabe erfüllt der auch am Informatik-Lehrstuhl 12 der Universität Dortmund entwickelte *enProfiler*. Der früher *Traceanalyzer* genannte *enProfiler* wurde ursprünglich von Schwarz im Rahmen seiner Diplomarbeit [SC00] implementiert. Lee [LE01] erweiterte den *enProfiler*, sodass dieser in der Lage war, die Nutzung von Cache-Speicher korrekt zu analysieren.

Für den Simulationsdurchlauf „Typ B“ sind folgende Ergänzungen zu beachten (*Abbildung 4.6*). Im Backend werden Algorithmen für die Scratch-Pad Nutzung

aufgerufen. Die Algorithmen unterstützen unterschiedliche Größen für den Scratch-Pad-Speicher (64, 128, 256, 512, 1024, 2048, 4096 oder 8192 Byte). Das Backend erzeugt zwei Dateien mit Assemblercode. Eine Datei enthält den Codeanteil, der später im Hauptspeicher liegen soll, die andere Datei enthält Code, der im Scratch-Pad abgelegt wird. Der nächste Unterschied zu „Typ A“ tritt bei der Nutzung des *enProfilers* auf. Dieser erhält zusätzliche Angaben über die Scratch-Pad Charakteristik, um eine korrekte Berechnung durchführen zu können.

Möchte man die Nutzung von Cache-Speicher simulieren, muss man den Simulationsdurchlauf „Typ C“ wählen (*Abbildung 4.6*). Dieser nutzt, genau wie „Typ A“, das Backend ohne jegliche Änderungen. Erst bei der Simulation durch den *ARMulator* muss der Ablauf angepasst werden. Der *ARMulator* benötigt Kenntnisse über das zu verwendende Cache-Modell, da die Nutzung des Cache-Speichers zur Programmlaufzeit und damit während der Simulation gesteuert wird. Der Cache-Speicher kann die gleichen Byte-Größen annehmen wie der Scratch-Pad-Speicher. Der zweite Unterschied zu „Typ A“ tritt erneut beim *enProfiler* auf, der die Cache-Charakteristik kennen muss, um korrekte Ausgaben liefern zu können.

Der *enProfiler* erzeugt schließlich einen Report, der Werte wie den Energieverbrauch und die CPU-Zyklen eines ausgeführten Programms enthält. Die *Abbildungen 4.7 - 4.12* zeigen einen Auszug aus diesem Report für einen Simulationsdurchlauf nach „Typ B“ des Beispielprogramms *ref\_kernel.c*, das in Kapitel 5 vorgestellt wird.

```
1. # Memory areas :
   | Area | Start - End | Size (Byte) | Accesses | Energy (µJ) |
   |-----|-----|-----|-----|-----|
   | Stack | 57fd98 - 57fff7 | 608 | 1473863 | 66982.996 |
```

*Abbildung 4.7: enProfiler-Ausgabe Teil 1*

Der Teil der *enProfiler*-Ausgabe in *Abbildung 4.7* zeigt die Größe, Anzahl der Zugriffe und den Energieverbrauch für den verwendeten Stackbereich an.

```
2. # Memory size :
   | Memoryunit | Inst | Data |
   |-----|-----|-----|
   | offchip (Byte) | 612 | 3176 |
   | onchip (Byte) | 64 | 0 |
```

*Abbildung 4.8: enProfiler-Ausgabe Teil 2*

Auskunft über den benutzten Speicher mit einer Aufteilung in OffChip- (Hauptspeicher) und OnChip-Speicher (Scratch-Pad-Speicher) ist in *Abbildung 4.8* zu sehen. Man kann genau erkennen, wieviel Speicher von Instruktionen oder von Daten belegt wird. In diesem Beispiel sind 64 Byte des OnChip-Speichers mit Instruktionen belegt. Der OnChip-Speicher hat in diesem Beispiel nur eine Größe von 64 Byte und ist damit voll belegt.

```
3. # function/basic block accesses and energy :
```

Type	Name	Addr	Acc	Energy (µJ) Processor	Memory
F	Reference_IDCT	5000f4	31	17660.1649	35434.1553
BB	Reference_IDCT	5000f4	31	9.1667	18.6725
BB	_M_35	500106	0	0.0000	0.0000
BB	LL17_7	500108	248	8.4866	17.8560
BB	_M_36	50010e	0	0.0000	0.0000
BB	LL16_7	500110	1984	176.3776	433.7818
BB	_M_45	50011a	15872	420.9254	761.8560
BB	...	...	...	...	...

Abbildung 4.9: enProfiler-Ausgabe Teil 3

Die Zugriffe auf Funktionen und ihre Basisblöcke werden im dritten Bereich (Abbildung 4.9) aufgelistet. Man kann die Anzahl der Zugriffe auf einzelne Funktionen und Basisblöcke ablesen, sowie die Speicheradressen, wo die Funktion oder der Basisblock beginnt. In dieser Tabelle ist der Energieverbrauch aufgeteilt in Prozessor und Speicher. So kann man z.B erkennen, welche Funktion bzw. welcher Basisblock hohen Energieverbrauch durch Nutzung des Speichers verursacht.

```
4. # basic block access scheme :
```

Source - BB	Destination - BB	Count
main	Initialize_Reference_IDCT	1
Initialize_Reference_IDCT	LL8_0	1
LL8_0	LL3_0	1
LL3_0	sqrt	1
sqrt	_M_9	1
_M_9	LL4_0	1
LL4_0	LL7_0	8
LL7_0	__16_ddiv	64
__16_ddiv	_M_12	64
_M_12	__16_dflt	64
__16_dflt	__16_dmul	15936
__16_dmul	_M_13	64
...	...	...

Abbildung 4.10: enProfiler-Ausgabe Teil 4

In Abbildung 4.10 ist dargestellt, welcher Basisblock *Destination-BB* wie häufig nach einem anderen Basisblock *Source-BB* aufgerufen wird.

```

5. # Memory accesses :

```

Memoryunit	Acc	Width	Inst	Data	Energy (nJ)
offchip	read	1 Byte	0	64	15.5
offchip	read	2 Byte	791817	15872	24.0
offchip	read	4 Byte	6855334	875551	49.3
onchip	read	2 Byte	428544	0	0.5
onchip	read	4 Byte	0	15872	0.5
offchip	write	2 Byte	0	1984	29.9
offchip	write	4 Byte	0	694523	41.1
SUM	access		8075695	1603866	

Abbildung 4.11: enProfiler-Ausgabe Teil 5

Die Zugriffe auf den Speicher werden detailliert in *Abbildung 4.11* aufgelistet. Ablesen kann man die Anzahl der Speicherzugriffe für die unterschiedlichen Speicherarten aufgeteilt nach Instruktionen und Daten. Außerdem sind in der Tabelle die Energiekosten für einen einzelnen Speicherzugriff für jede Speicherart enthalten. In der letzten Zeile der Tabelle kann man noch die Summe für die Speicherzugriffe für Instruktionen und Daten erfahren.

```

6. # CPU-Cycles : 38661948

```

```

7. # Energy values :

```

	SUM	Processor	Memory
Energy (μJ)	597820.751	168326.032	429494.719
Power (mW)	510.3	143.7	366.6
Current (mA)	154.6	43.5	111.1

Abbildung 4.12: enProfiler-Ausgabe Teil 6

*Abbildung 4.12* zeigt den Energieverbrauch und die Anzahl der benötigten CPU-Zyklen. Die Anzahl der CPU-Zyklen (6.) beträgt 38.661.948 und der Energieverbrauch (7.) insgesamt 597.820,751 μJ. Die Ausgabe des *enProfilers* gibt außerdem Auskunft über die Aufteilung des Energieverbrauchs in Prozessor- und Speicherkosten. In diesem Beispiel fallen für den Prozessor 168.326,032 μJ an Energiekosten an und 429.494,719 μJ für den Speicher. Die gleiche Aufteilung existiert für die Leistung und den Strom.

In diesem Kapitel wurde die für diese Diplomarbeit verwendete Arbeitsumgebung vorgestellt. Das nächste Kapitel zeigt die ersten eingesetzten Optimierungen.



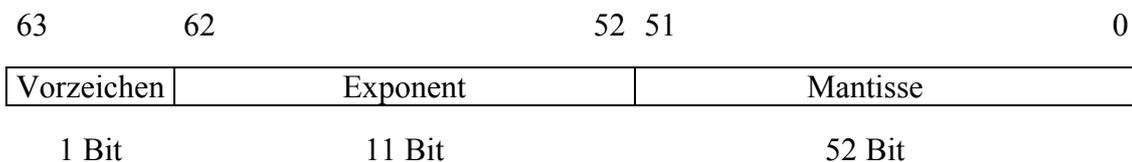
## 5 Optimierung auf Spezifikationsebene

Bei der Optimierung auf Spezifikationsebene hat man in dieser Diplomarbeit einen Trade-off zwischen Genauigkeit der zu optimierenden MPEG-Applikation und der Performance bzw. des Energieverbrauchs der MPEG-Applikation, je nachdem, was das Optimierungsziel ist.

In diesem Kapitel soll gezeigt werden, welche Möglichkeiten der Optimierung auf der Spezifikationsebene anhand von Transformationen der benutzten Datentypen vorhanden sind.

### 5.1 double -> float Transformation

Das Programm *ref\_kernel.c* benutzt den Datentyp „double“ zur Berechnung der inversen diskreten Kosinus Transformation. Der IEEE 754 Standard definiert *double-precision-floating-point* als 64 Bit breiten Wert, dessen Bit 0-51 der Mantisse, Bit 52-62 dem Exponenten und Bit 63 dem Vorzeichen entsprechen [IEEE95] (*Abbildung 5.1*).



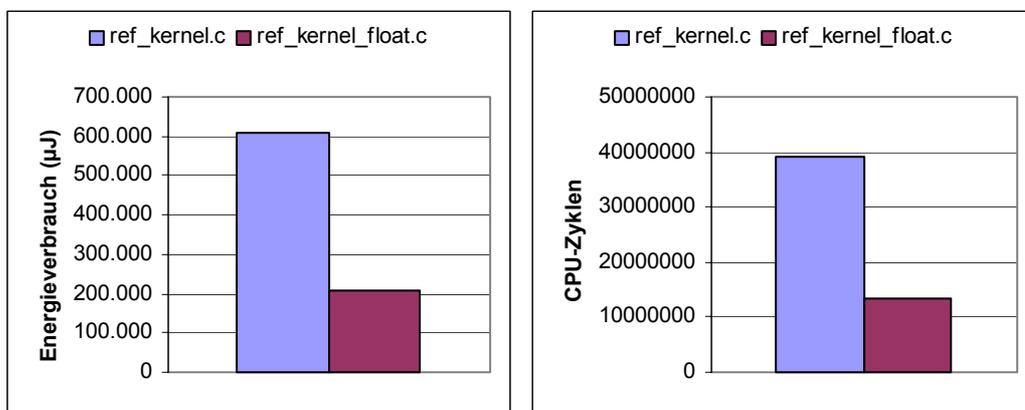
*Abbildung 5.1: double-precision-floating-point nach IEEE 754*

Im Programm *ref\_kernel.c* wurden alle Variablen vom Datentyp „double“ auf Variablen vom Datentyp „float“ geändert. Die neue Programmversion (*ref\_kernel\_float.c*) enthielt keine Variablen vom Datentyp „double“ mehr. Die Bitbreite der Mantisse konnte so von 52 Bit auf 23 Bit reduziert werden.

Inwieweit diese Änderung Auswirkungen auf die Genauigkeit des Programms hat, musste im nächsten Schritt überprüft werden. Würde die Genauigkeit des Programms sich verschlechtern, müsste überprüft werden, ob die Abweichungen noch tolerierbar sind. Es war zu erwarten, dass der Energieverbrauch durch die Datentypänderung sinkt, da die Bitbreite des benutzten Datentyps von 64 Bit auf 32 Bit verringert wurde.

Durch den Vergleich der Ausgaben der Programme *ref\_kernel.c* und *ref\_kernel\_float.c* sollte überprüft werden, ob der Wechsel vom Datentyp „double“ auf den Datentyp „float“ Auswirkungen auf die Ausgabe des Programms *ref\_kernel\_float.c* hat. Es stellte sich heraus, dass die Ausgabe beider Programme identisch war. Daraus konnte man schließen, dass für die hier benutzte Software und die verwendeten Testdaten ein Wechsel der Datentypen zulässig ist und deshalb der Datentypwechsel weiter untersucht werden sollte.

Da die Änderungen des Programms keinen Einfluss auf die Genauigkeit für die Testdaten hatten, wurde dieser Ansatz weiter untersucht. In *Abbildung 5.2* sieht man den Energieverbrauch und die ausgeführten CPU-Zyklen der Programme *ref\_kernel.c* und *ref\_kernel\_float.c*.



*Abbildung 5.2: ref\_kernel.c und ref\_kernel\_float.c im Vergleich*

Wie man deutlich sieht, kann man durch die Änderung der Datentypen eine beachtliche Energieeinsparung erreichen. Das Programm *ref\_kernel.c* verbraucht 609.504 µJ, während das Programm *ref\_kernel\_float.c* sich mit 207.035 µJ begnügt. Die „float“-Variante verbraucht demnach nur ca. 34 % der Energie der „double“-Variante.

Im Bereich der Performance, also der benötigten Anzahl an CPU-Zyklen zur Ausführung des Programms, lässt sich fast das gleiche Ergebnis beobachten. Während *ref\_kernel.c* 38.995.260 CPU-Zyklen benötigt, sind es bei *ref\_kernel\_float.c* nur 13.464.956 Zyklen, also ca. 35 %.

Derart starke Einsparungen liegen nicht nur in dem Wechsel von einem 64-Bit Datentyp auf einen 32-Bit Datentyp begründet, sondern vielmehr auch in der Nutzung anderer Softwarebibliotheksfunktionen. Welche Bibliotheksfunktionen zum Einsatz kamen, konnte nur mit dem im ARM SDT 2.50 Kit enthaltenen *armprof*<sup>7</sup> [ARM98] festgestellt werden. Das Tool *armprof* bietet die Möglichkeit, die Anteile einzelner Funktionen an der Programmausführungszeit prozentual

<sup>7</sup> *armprof* – ARM Profiler

auszugeben. Dabei werden auch eventuell benutzte Bibliotheksfunktionen berücksichtigt. Die *armprof*-Ausgabe gibt Aufschluss über die benutzten Bibliotheksfunktionen.

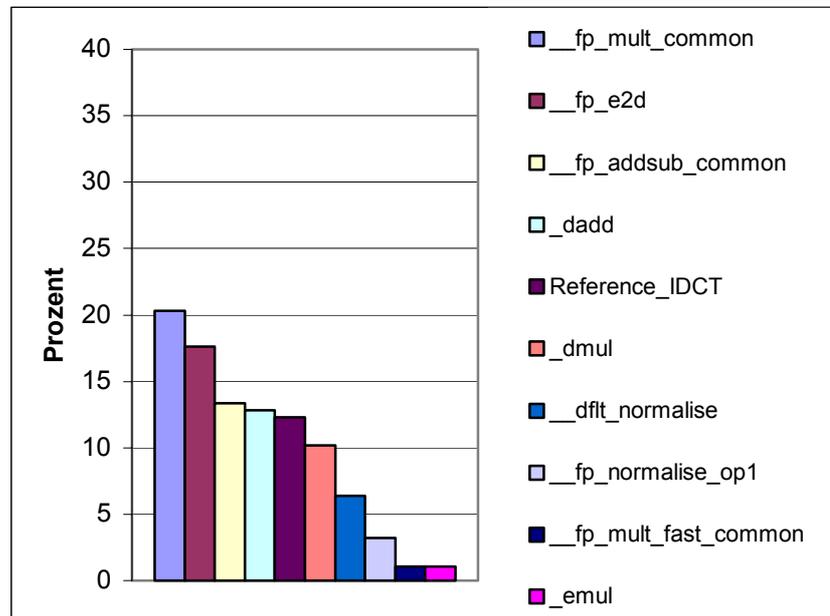


Abbildung 5.3: Anteile an Programmausführungszeit - *ref\_kernel.c*

In *Abbildung 5.3* kann man erkennen, dass sich unter den 10 Funktionen mit dem größten Anteil an der Programmausführungszeit 9 Bibliotheksfunktionen befinden. *Hinweis: Bibliotheksfunktionen erkennt man daran, dass ihr Funktionsname mit einem Unterstrich beginnt.* Vier dieser Funktionen haben einen größeren Anteil an der Ausführungszeit als die Funktion *Reference\_IDCT*. Diese vier Funktionen kommen zusammen auf einen Anteil von 64,15 % gegenüber 12,29 % Anteil der Funktion *Reference\_IDCT*.

Die Verteilung sieht bei dem energiesparsameren *ref\_kernel\_float.c* anders aus (*Abbildung 5.4*). Hier sieht man, dass die Funktion *Reference\_IDCT* die Funktion mit dem zweitgrößten Anteil (25,97 %) an der Programmausführungszeit ist. Sie wird nur noch übertroffen von der Bibliotheksfunktion *fmul*, die 35,06 % der Zeit benötigt.

Berücksichtigt man hierbei, dass die „float“-Variante nur ca. 30 % der Ausführungszeit benötigt und der Anteil der Funktion *Referenz\_IDCT* an der Ausführungszeit sich verdoppelt, kann man sagen, dass die Bibliotheksfunktionen, die beim Datentyp „double“ benutzt werden, wesentlich „teurer“ sind, als die Funktionen, die beim Datentyp „float“ benutzt werden.

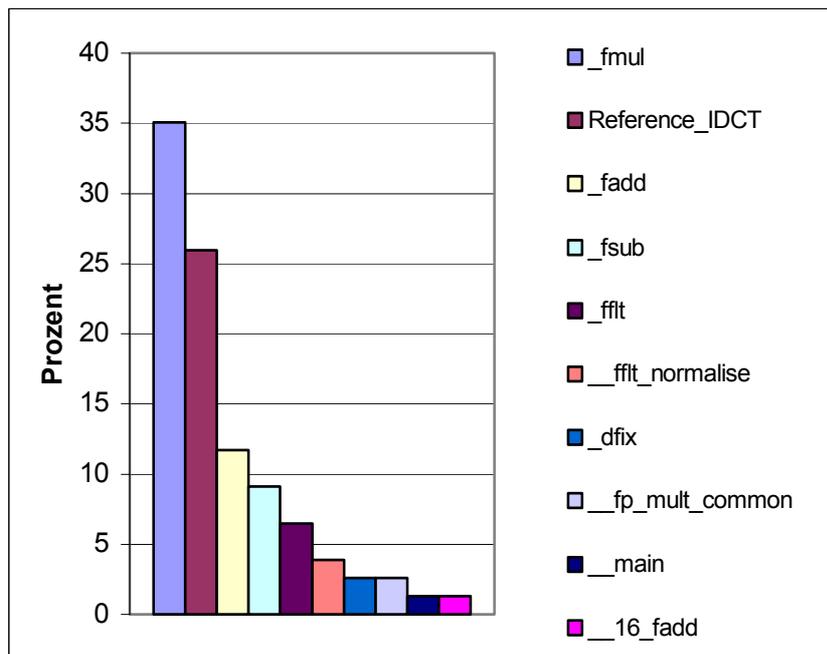


Abbildung 5.4: Anteile an Programmausführungszeit - ref\_kernel\_float.c

## 5.2 float Bitbreitenverringern

Tong et al. beschreiben in [TNR00] eine Reduzierung der Bitbreite der Mantisse von einfachen Fließkommazahlen zur Energieoptimierung. Der Datentyp *single-precision-floating-point* wird als 32 Bit breiter Wert definiert [IEEE95], dessen Bit 0-22 der Mantisse, Bit 23-30 dem Exponenten und Bit 31 dem Vorzeichen entsprechen (Abbildung 5.5).

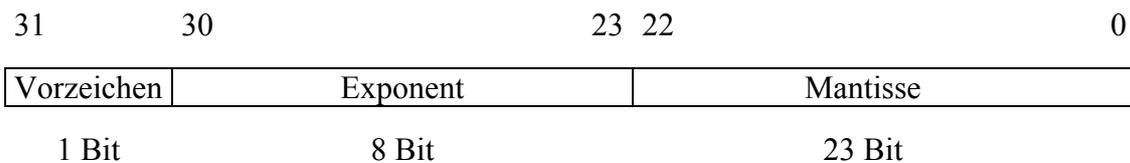


Abbildung 5.5: single-precision-floating-point nach IEEE 754

Laut [TNR00] ist eine Reduzierung der Bitbreite der Mantisse bei *single-precision-floating-point* von 23 Bit auf 11 Bit möglich, ohne dass dabei die Genauigkeit der in [TNR00] getesteten Programme beeinflusst wird. Die getesteten Programme sind:

- *Sphinx III*, ein Spracherkennungsprogramm [SP02]
- *ALVINN*, Programm aus SPECfp92, das Sensordaten auswertet [PJ02]
- *PCASYS*, Verarbeitung und Kategorisierung von Bildinformationen über Fingerabdrücke [NI02]
- *Bench22*, ein Bildverarbeitungsbenchmark
- *Fast DCT*, ein Programm, das sowohl eine DCT als auch eine IDCT enthält

In [TNR00] wird beschrieben, dass zur Reduzierung der Bitbreite der Mantisse, die Multipliziereinheit verändert werden muss. Anstelle einer Standard *single-precision-floating-point* Multipliziereinheit, soll eine vereinfachte Multipliziereinheit benutzt werden, die in [TNR00] näher beschrieben wird. Diese kann nur 8-Bit in einem Durchlauf verarbeiten, während die Standard-Multipliziereinheit 24 Bit berechnen kann. Sollen mit der vereinfachten Einheit 24 Bit berechnet werden, sind 3 Zyklen nötig, für 16 Bit sind 2 Zyklen nötig.

Im Falle der Programme *Sphinx III* und *ALVINN* reicht eine 8-Bit Multiplikation aus und dadurch kann eine Energieeinsparung von 78% für die Multipliziereinheit erreicht werden. Bei den Programmen *PCASYS*, *Bench22* und *Fast DCT* werden 9-16 Bit benötigt, was 2 Zyklen für die vereinfachte Multipliziereinheit bedeutet und immerhin noch 32% Energieeinsparung gegenüber der Standard 24-Bit Multipliziereinheit bringt.

Eine Änderung der Fließkomma-Arithmetikeinheit war nicht möglich, da die betrachtete Hardware keine Fließkomma-Arithmetikeinheit enthielt. Die andere Möglichkeit, die Bitbreite der Mantisse durch Änderungen der Softwarebibliotheken zu reduzieren, war im Rahmen dieser Diplomarbeit zu aufwändig und wurde daher nicht weiter untersucht. Aber dieses Beispiel zeigt, welches Potenzial im Bereich der Fließkomma-Arithmetik vorhanden ist.

### 5.3 float -> integer Transformation

Eine andere Möglichkeit der Optimierung ist, den verwendeten Algorithmus eines Programms durch einen anderen Algorithmus zu ersetzen. Der neue Algorithmus erfüllt die Aufgabe schneller als der alte Algorithmus. Als Beispiel ist hier das Sortierverfahren SelectionSort genannt, das eine Laufzeit  $O(n^2)$  hat. Der Sortieralgorithmus HeapSort hat eine Laufzeit von  $O(n \log n)$  [GU92]. Beide Verfahren erreichen das gleiche Ergebnis, aber das HeapSort-Verfahren ist schneller.

Es kann durchaus sein, dass der schnellere Algorithmus nicht dieselbe Genauigkeit erreicht, wie der ursprünglich verwendete Algorithmus. Dadurch wird die Spezifikation des gesamten Programms geändert. Die Genauigkeit des optimierten Programms entspricht nicht mehr der in der Spezifikation geforderten Genauigkeit des originalen Programms.

Zur ersten Orientierung bezüglich des Anteils einzelner Funktionen an der Programmausführungszeit wurde die vollständige MPEG-2 Dekoder-Software auf einem Intel Pentium 200MMX mit 64 Mbyte Speicher unter Linux mit dem *GCC* kompiliert und anschließend ausgeführt. Für die Laufzeitanalyse wurde das in Kapitel 2.3 vorgestellte Tool *Gprof* eingesetzt. Als Eingabedaten wurde ein MPEG-2 Videostream mit der Auflösung 720 Pixel \* 576 Pixel verwendet. Dieser Videostream enthält 95 Einzelbilder und hat eine Dateigröße von 2,3 MByte.

In den Kapiteln 2.3.1 und 2.3.2 wurden die Funktionen *Reference\_IDCT* und *Fast\_IDCT* vorgestellt. Diese Funktionen unterscheiden sich in ihrer Geschwindigkeit und ihrer Genauigkeit. Während der MPEG-2 Dekoder bei Nutzung der Funktion *Reference\_IDCT* 281,18 Sekunden benötigt, hat der MPEG-2 Dekoder das Dekodieren des Videostreams mit der Funktion *Fast\_IDCT* schon nach 76,40 Sekunden abgeschlossen.

Abbildung 5.6 zeigt die Anteile einzelner Funktionen der MPEG-2 Dekoder-Software an der Programmausführungszeit bei Nutzung der Funktion *Reference\_IDCT*. Die Funktion *Reference\_IDCT* hat mit ca. 80% den größten Anteil an der Ausführungszeit.

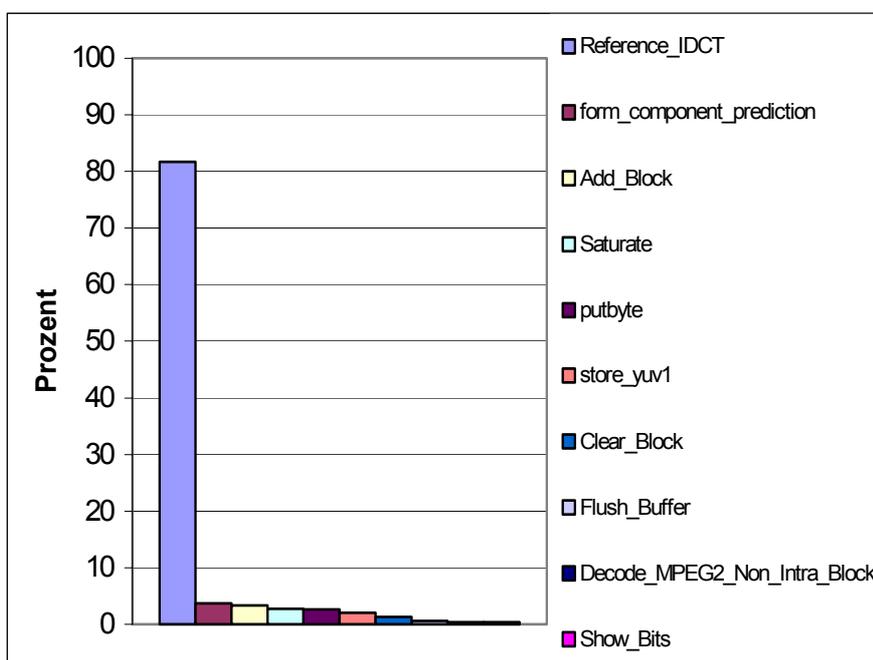


Abbildung 5.6: Anteile an Programmausführungszeit

Im Vergleich dazu sieht man in *Abbildung 5.7* die prozentualen Anteile, wenn die Funktion *Fast\_IDCT* benutzt wird. Die Funktion *Fast\_IDCT* selbst hat nur einen Anteil von etwas über 2 Prozent. Das liegt aber daran, dass die schnelle inverse diskrete Kosinus Transformation eigentlich aus 3 Funktionen besteht: *Fast\_IDCT*, *idctrow* und *idctcol*. Die Funktionen *idctrow* und *idctcol* werden nacheinander aus der Funktion *Fast\_IDCT* heraus aufgerufen.

Addiert man die prozentualen Anteile der 3 einzelnen Funktionen, erhält man einen gesamten prozentualen Anteil an der Programmausführungszeit der gesamten MPEG-2 Dekoder Software von fast 30% für die schnelle inverse diskrete Kosinus Transformation (*Abbildung 5.8*).

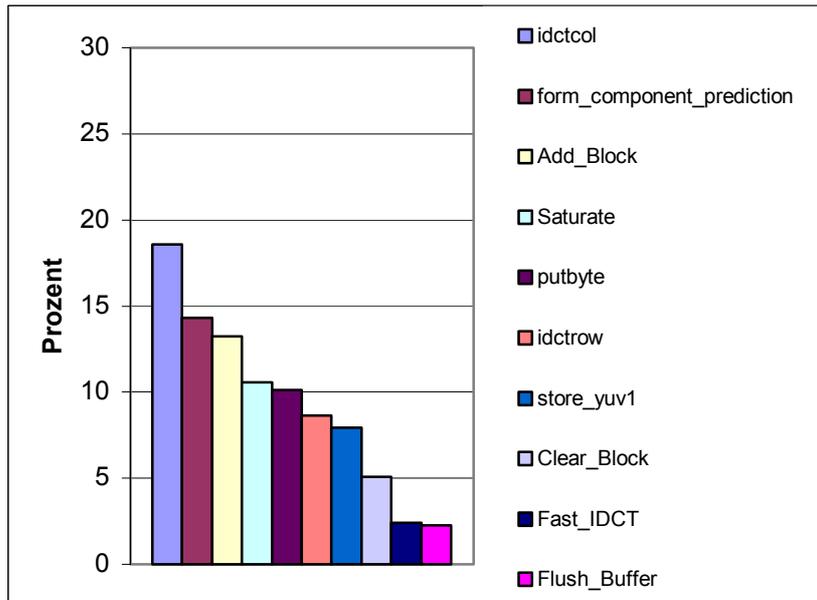


Abbildung 5.7: Anteile an Programmausführungszeit

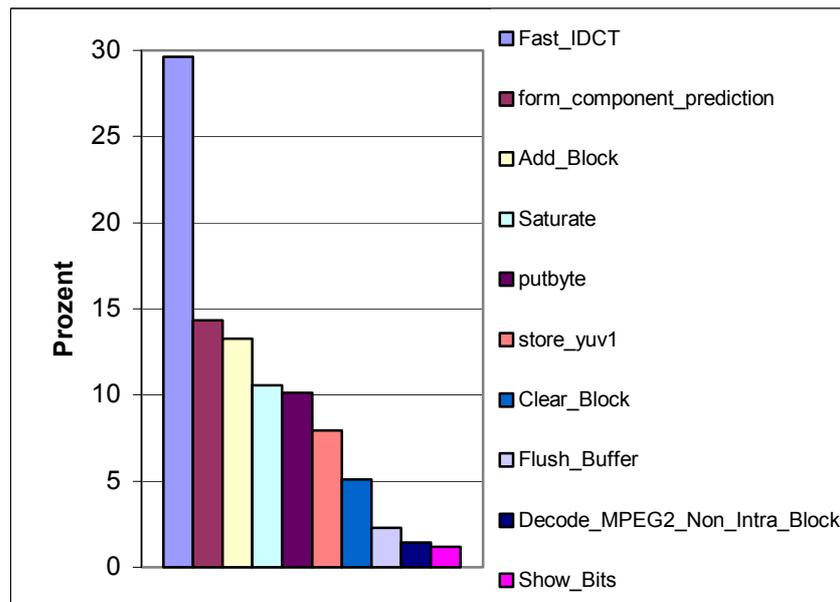


Abbildung 5.8: Anteile an Programmausführungszeit

Die inverse diskrete Kosinus Transformation hat immer einen hohen Anteil an der Ausführungszeit. Dabei ist egal, ob man die Funktion *Reference\_IDCT* oder die schnellere *Fast\_IDCT* benutzt. Diese Aussage trifft aber nur auf das hier benutzte Testsystem mit Intel Pentium-Prozessor zu. Es handelt sich dabei um einen anderen Prozessor als den ARM-Prozessor, den wir ihn in Kapitel 4 kennengelernt haben.

Im nächsten Schritt ist es natürlich interessant, zu erfahren, wieviel Energie die MPEG-2 Dekoder-Software je nach gewählter Funktion auf einem eingebetteten System verbraucht. Dazu wird die gesamte MPEG-2 Dekoder-Software auf einem Rechner mit Solaris-Betriebssystem kompiliert. Das ist aber mit dem *encc* zu diesem Zeitpunkt noch nicht möglich. Die einzige Möglichkeit besteht darin, als Compiler den *tcc* [ARM98] der Firma ARM einzusetzen. Dieser Compiler benutzt Standardoptimierungen wie *Common Subexpression Elimination (CSE)*, *Loop Invariant Code Motion (LICM)* und *Constant Folding* und erzeugt Maschinencode im Thumb-Befehlssatz.

Dieser Maschinencode wird dann mit dem *ARMulator* ausgeführt. Der *ARMulator* schreibt eine Trace-Datei, in der Informationen über die Programmausführung enthalten sind. Der *enProfiler* wertet diese Datei anschließend aus und schreibt eine Ergebnisdatei.

Die MPEG-2 Dekoder Software wurde beim ersten Durchlauf mit der Funktion *Reference\_IDCT* kompiliert. Als Eingabedaten wurde nicht die zu Beginn dieses Kapitels erwähnte Testdatei benutzt, sondern ein Videostream mit 5 Einzelbildern. Dieser Videostream hat eine Auflösung von 32 Pixel \* 16 Pixel und eine Dateigröße von 2,2 Kbyte, entspricht also ca. einem Tausendstel der ersten Testdatei. Dies war nötig, da die Simulation durch den *ARMulator* sehr lange für die Ausführung benötigt. Ein kompletter Simulationsdurchlauf der *Reference\_IDCT* Version benötigt mit der „kleinen“ Testdatei zwischen 15 und 20 Minuten, je nach Auslastung des verwendeten Rechners.

Im ersten Durchlauf verbrauchte der MPEG-2 Dekoder ungefähr 1.171.281  $\mu\text{J}$  und benötigte 75.306.702 CPU-Zyklen zur Programmausführung.

Wesentlich geringer fiel der Energieverbrauch bei Nutzung der Funktion *Fast\_IDCT* für die inverse diskrete Kosinus Transformation aus. Hierbei betrug der Energieverbrauch 56.929  $\mu\text{J}$  bei 3.840.997 CPU-Zyklen. Das entspricht beim Energieverbrauch 4,86 Prozent gegenüber der Nutzung der *Reference\_IDCT*, also einer Einsparung von 95,14 Prozent. Die Anzahl der CPU-Zyklen verringerte sich um 94,90 Prozent auf 5,10 Prozent. Dabei muss aber berücksichtigt werden, dass die Genauigkeit der *Fast\_IDCT* nicht der Genauigkeit der *Reference\_IDCT* entspricht.

In *Abbildung 5.9* sind der Energieverbrauch und die benötigten CPU-Zyklen für die Nutzung beider Funktionen dargestellt.

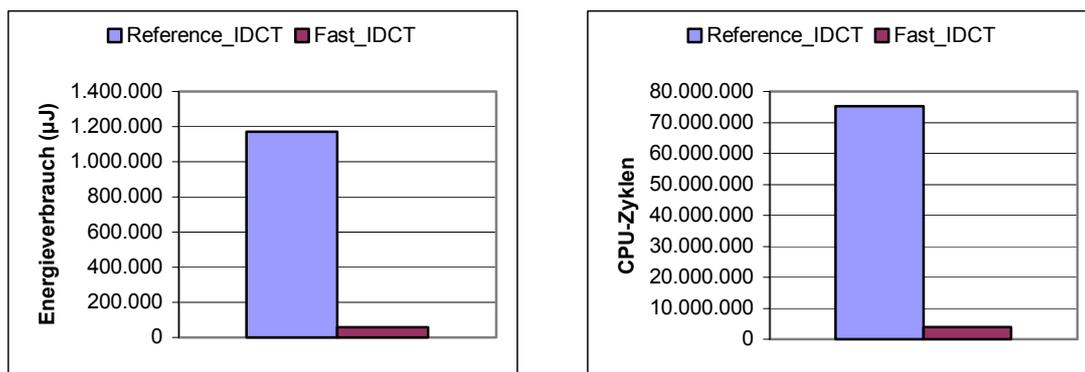


Abbildung 5.9: *Reference\_IDCT* und *Fast\_IDCT* im Vergleich

Ein Grund für diesen gravierenden Unterschied im Energieverbrauch liegt im jeweils verwendeten Datentyp. Während die Funktion *Fast\_IDCT* Variablen vom 32 Bit breiten Datentyp „integer“ benutzt, greift die *Reference\_IDCT* auf den 64 Bit breiten Datentyp „double float“ zu. Dies führt dazu, dass die benutzten Variablen doppelt so viel Speicher benötigen. Die Berechnungen mit Variablen vom Datentyp „double float“ werden auf der betrachteten Hardware von Softwarebibliotheksfunktionen ausgeführt. Die Nutzung von Funktionen aus Softwarebibliotheken ist im Gegensatz zur Nutzung der gleichen Funktionalität mittels vorhandener Hardware sehr „teuer“. Eine höhere Anzahl an CPU-Zyklen wird benötigt und dadurch steigt auch der Energieverbrauch. Eine entsprechende Hardware für die Berechnung mit dem Datentyp „double float“ ist aber nicht vorhanden. Deshalb kann auf die Nutzung der Softwarebibliotheken nicht verzichtet werden.

## 5.4 Fazit

Das Ändern des Datentypen von „double“ in „float“ hat keine Änderung der Ergebnisse bei den gewählten Testdaten ergeben und sollte daher auf jeden Fall angewendet werden. Eine Nutzung der Funktion *Fast\_IDCT*, die den Datentyp „integer“ benutzt, als Ersatz für die Funktion *Reference\_IDCT* bringt eine noch deutlichere Energieersparnis. Es muss aber berücksichtigt werden, dass für die *Fast\_IDCT* nicht die gleiche Genauigkeit wie bei der *Reference\_IDCT* erreicht wird.

Im nächsten Kapitel werden Optimierungen auf der Quellcode- und HIR-Ebene untersucht.



## 6 Optimierung auf Quellcode- bzw. HIR-Ebene

Die Optimierungen auf der Quellcode- bzw. HIR-Ebene erlauben keine Änderung der Spezifikation einer Anwendung. Ein Beispiel für die Quellcode-Ebene ist C-Programmcode. Durch die Optimierung darf sich die Ausgabe einer Anwendung nicht verändern.

In Kapitel 6.1 werden Schleifenoptimierungen und deren Auswirkungen auf den Energieverbrauch dargestellt. Schleifen sind ein wichtiger Bestandteil der betrachteten Funktion *Reference\_IDCT* und ein Großteil der Ausführungszeit wird in Schleifen verbracht. Deshalb kann hier ein großes Optimierungspotenzial vorliegen.

In Kapitel 6.2 werden Optimierungen betrachtet, die sich mit Arraytransformationen beschäftigen. Arraytransformationen können den Datentransfer zwischen CPU und Speicher optimieren und dadurch den Energieverbrauch erheblich reduzieren. Die Funktion *Reference\_IDCT* benutzt ein Array mit 64 Werten vom Datentyp „double“ und ein 2-dimensionales Array mit ebenfalls 64 Werten vom Datentyp „double“. Eine Optimierung der Zugriffe auf diese beiden Arrays kann also eine hohe Energieeinsparung bedeuten.

Allen Optimierungen liegt das Programm *ref\_kernel.c* zugrunde, das in Kapitel 5 erwähnt wurde.

### 6.1 Schleifenoptimierungen

Die Funktion *Reference\_IDCT* enthält mehrere Schleifen, wie in Kapitel 2.3.1 beschrieben wurde. Die Schleifen müssen daraufhin untersucht werden, ob sie Optimierungspotenzial für gängige Schleifenoptimierungen bieten.

Eine Optimierung ist die *loop invariant code motion* [MU97][BA94]. In *Abbildung 6.1* sieht man den Original Programmcode des ersten Schleifenblocks der Funktion *Reference\_IDCT*. Auffällig ist in Zeile 5 und 6 der Ausdruck `,8*i'`. Dieser Ausdruck steht somit einmal in der innersten Schleife (Zeile 5) und einmal in der mittleren Schleife (Zeile 6). Der Ausdruck hat aber keinerlei Abhängigkeiten in den beiden inneren Schleifen, sondern hängt nur von der äußeren Schleife (Zeile 1) ab. D.h. der Ausdruck wird in den inneren Schleifen nur gelesen und nicht verändert. Eine Änderung tritt erst ein, wenn die äußere Schleife erneut durchlaufen wird.

```

1  for (i=0; i<8; i++)
2    for (j=0; j<8; j++)
3      {
4        partial_product = 0.0;
5        for (k=0; k<8; k++)
6          partial_product+= c[k][j]*block[8*i+k];
7        tmp[8*i+j] = partial_product;
8      }

```

Abbildung 6.1: Original-Programmcode

Man kann eine neue Variable einführen, die den Ausdruck enthält. In *Abbildung 6.2* sieht man die modifizierte Version des Programmcodes nach der durchgeführten *loop invariant code motion (licm)*. Zeile 3 enthält jetzt die neue Variable `v_8`, der der Ausdruck `,8*i'` zugewiesen wird. Durch die Variable `v_8` ersetzt man in den inneren Schleifen den Ausdruck `,8*i'` (Zeile 7 und Zeile 8).

```

1  for (i=0; i<8; i++)
2  {
3    v_8 = 8*i;
4    for (j=0; j<8; j++)
5      {
6        partial_product = 0.0;
7        for (k=0; k<8; k++)
8          partial_product+= c[k][j]*block[v_8+k];
9        tmp[v_8+j] = partial_product;
10     }
11  }

```

Abbildung 6.2: Programmcode nach loop invariant code motion

Diese Transformation führt dazu, dass die Multiplikation `,8*i'` genau acht Mal im Gegensatz zu 576 Mal vor der Transformation ausgeführt wird. Berücksichtigt werden muss hier aber, dass durch das Hinzufügen einer neuen Variable ein weiteres Register belegt wird. Das kann bei Systemen mit geringer Registeranzahl zu zusätzlichen Kosten durch *Spilling* führen.

*Spilling* bedeutet, dass der Wert eines Registers zwischendurch im Speicher abgelegt werden muss, damit das Register anderweitig benutzt werden kann. Wird der Wert wieder benötigt, muss er vom Speicher in das Register zurückkopiert werden. Diese zusätzlichen Kopierbefehle verursachen zusätzliche Kosten bzw. erhöhen den Energieverbrauch einer Anwendung. Es kann daher besser sein, auf eine *licm* zu verzichten, da sonst durch die zusätzlichen Kopierbefehle keine Optimierung mehr eintritt.

In der Funktion *Reference\_IDCT* musste 1 Register mehr gespilt werden als bei der unveränderten Funktion. Das kann der Grund dafür sein, warum keine nennenswerte Optimierung eingetreten ist. Die Energieeinsparung des gesamten Programms beträgt 0,10%, während der Energieverbrauch der einzelnen Funktion *Reference\_IDCT* um 0,91% gesunken ist (*Abbildung 6.3*).

<i>Reference_IDCT</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)
Original	64.360,42430	100,00
nach <i>LICM</i>	63.774,56400	99,09

*Abbildung 6.3: Energieverbrauch der Funktion Reference\_IDCT*

Als nächste Optimierung wurde die *Loop Reversal*-Methode [BA94] untersucht. Bei der *Loop Reversal*-Optimierung geht es darum, dass man sich die Befehle, die den Inhalt eines Registers mit der Zahl Null vergleichen, sparen kann. In *Abbildung 6.4* sieht man den Programmcode einer Schleife, wie sie mehrmals in der *Reference\_IDCT* Funktion vorkommt. Bei der Optimierung muss darauf geachtet werden, dass vorhandene Abhängigkeiten zwischen den Zählvariablen verschachtelter Schleifen weiterhin erfüllt bleiben. Gegebenenfalls muss man den Programmcode innerhalb der Schleifen anpassen. Das war in diesem Beispiel aber nicht notwendig.

```
for (i=0; i<8; i++)
{
    ...
}
```

*Abbildung 6.4: Original-Programmcode*

*Loop Reversal* dreht die Richtung einer Schleife um. D.h., die Zählvariable der Schleife wird nicht von 0 bis 7 hochgezählt, sondern von 7 bis 0 runtergezählt. Den optimierten Programmcode kann man in *Abbildung 6.5* sehen.

```
for (i=7; i>=0; i--)
{
    ...
}
```

*Abbildung 6.5: Programmcode nach Loop Reversal*

Das Ergebnis dieser Optimierung wird sehr gut durch den vom Compiler erzeugten Assemblercode deutlich. Während vorher für die Überprüfung der Schleifenbedingung am Ende eines Schleifendurchlaufes drei Befehle benötigt werden, sind es nach Anwendung der *Loop Reversal* Methode nur noch 2 Befehle (*Abbildung 6.6*).

<pre> ... ADD r4, #1 CMP r4, #8 BLT LL3_0 ...                 </pre>	<pre> ... SUB r4, #1 BGE LL3_0 ...                 </pre>
<i>Original-Assemblercode</i>	<i>Assemblercode nach Loop Reversal</i>

*Abbildung 6.6*

Mit dieser Optimierung konnte eine geringe Energieeinsparung erzielt werden. Die Energieverbrauchseinsparung des Programms lag bei 0,08%. Die einzelne Funktion *Reference\_IDCT* konnte 1,74% Verbesserung erzielen (*Abbildung 6.7*).

<i>Reference_IDCT</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)
Original	64.360,42430	100,00
nach <i>Loop Reversal</i>	63.243,58110	98,26

*Abbildung 6.7: Energieverbrauch der Funktion Reference\_IDCT*

*Loop unrolling* bezeichnet das Abrollen von Schleifen [BA94]. In diesem Beispiel wurde die innere Schleife aus *Abbildung 6.8* komplett abgerollt und dadurch aufgelöst. Der Vorteil ist, dass man die Variable *k* nicht mehr benötigt und das Abfragen der Schleifenbedingung auch entfällt. Desweiteren können andere Standard-Optimierungen den Code anschließend weiter reduzieren.

```

1  for (i=0; i<8; i++)
2    for (j=0; j<8; j++)
3    {
4      partial_product = 0.0;
5      for (k=0; k<8; k++)
6        partial_product+= c[k][j]*block[8*i+k];
7      tmp[8*i+j] = partial_product;
8    }
                
```

*Abbildung 6.8: Original-Programmcode*

Der Programmcode wird durch die Optimierung größer (*Abbildung 6.9*), da der Schleifeninhalt für jeden Schleifendurchlauf einmal im Programmcode aufgeführt sein muss. Dennoch kann in diesem Beispiel bedeutend mehr Energie als bei den vorherigen Schleifenoptimierungen eingespart werden.

```

1  for (i=0; i<8; i++)
2    for (j=0; j<8; j++)
3    {
4      partial_product= c[0][j]*block[8*i];
5      partial_product+= c[1][j]*block[8*i+1];
6      partial_product+= c[2][j]*block[8*i+2];
7      partial_product+= c[3][j]*block[8*i+3];
8      partial_product+= c[4][j]*block[8*i+4];
9      partial_product+= c[5][j]*block[8*i+5];
10     partial_product+= c[6][j]*block[8*i+6];
11     partial_product+= c[7][j]*block[8*i+7];
12     tmp[8*i+j] = partial_product;
13   }

```

*Abbildung 6.9: Programmcode nach loop unrolling*

*Abbildung 6.10* zeigt den Energieverbrauch und die benötigten CPU-Zyklen für das Programm *ref\_kernel.c* in seiner Originalform und nach durchgeführtem *Loop unrolling*. In beiden Bereichen, also sowohl Energie als auch Performance, können 6,38% eingespart werden. Durch die vollkommen identische prozentuale Verbesserung wird deutlich, dass ein starker Zusammenhang zwischen Performance und Energieverbrauch existiert.

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
Original	609.504,289	100,00	38.995.260	100,00
nach <i>Loop unrolling</i>	570.626,187	93,62	36.507.191	93,62

*Abbildung 6.10: Energieverbrauch und CPU-Zyklen im Vergleich*

Durch die *Loop unrolling*-Optimierung kann für die *Reference\_IDCT* Funktion eine Verbesserung von 17,00% des Energieverbrauchs erreicht werden (*Abbildung 6.11*).

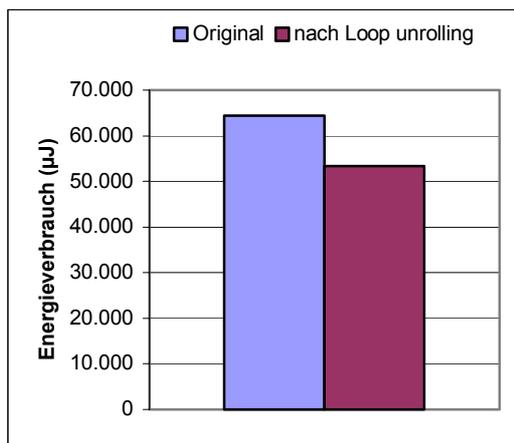


Abbildung 6.11: Energieverbrauch der Funktion *Reference\_IDCT*

Das *Loop unrolling* der innersten Schleife sollte auf jeden Fall durchgeführt werden. Ein zusätzliches teilweises Abrollen der mittleren Schleife mit Schrittweite 2 führte dazu, dass 34 Register „gespillt“ werden mussten (26 mehr „gespillte“ Register, als wenn nur die innerste Schleife komplett abgerollt wird). Daher kam das zusätzliche Abrollen der anderen beiden Schleifen nicht in Frage.

Ein alleiniges Abrollen der äußersten Schleife brachte keine Einsparung mit sich und wird hier deshalb nicht weiter betrachtet. Der Einsatz der anderen hier vorgestellten Schleifenoptimierungstechniken bringt für diese Diplomarbeit bei einzelner Betrachtung keine nennenswerten Vorteile.

## 6.2 Arrayoptimierungen

In der Funktion *Reference\_IDCT* wird mehrmals auf drei verschiedene Arrays zugegriffen. Das erste Array ist das globale zweidimensionale Array *c*, das 8\*8 Werte vom Datentyp „double“ enthält. Dieses Array wird zu Beginn der Programmausführung einmal mit Werten gefüllt. Im weiteren Programmablauf werden die Werte dieses Arrays nur noch ausgelesen. Bei dem zweiten Array handelt es sich um das eindimensionale Array *tmp*, das 64 Werte vom Datentyp „double“ enthält. Das Array *tmp* ist ein lokales Array der Funktion *Reference\_IDCT* und wird zur Speicherung von Zwischenergebnissen bei der Berechnung der inversen diskreten Kosinus Transformation benötigt. Bei dem dritten Array *block* handelt es sich um ein eindimensionales Feld mit 64 Werten vom Datentyp „short integer“. Diese 64 Werte stellen die zu Eingabewerte dar.

Array	Datentyp	Lesezugriffe	Schreibzugriffe	Gesamte Zugriffe
<i>c</i>	double	1024	0	1024
<i>tmp</i>	double	512	64	576
<i>block</i>	short int	512	64	576
Gesamt		2048	128	2176

Abbildung 6.12: Arrayzugriffe in der Funktion *Reference\_IDCT*

Wie man in *Abbildung 6.12* sieht, werden  $1024+576=1600$  Zugriffe auf Arrays vom Datentyp „double“ und 576 Zugriffe auf ein Array vom Datentyp „short integer“ durchgeführt. Insgesamt sind das 2176 Speicher-Zugriffe. Laut Cathoor et al. [CAT94] und [MIR98] haben die Energiekosten, die durch den verwendeten Speicher verursacht werden, den größten Anteil am Gesamtenergieverbrauch einer daten-intensiven Anwendung (data-dominated application). Für unser Beispielprogramm *ref\_kernel.c* trifft das ebenfalls zu (*Abbildung 6.13*).

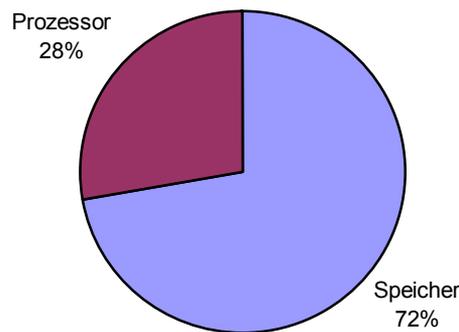


Abbildung 6.13: Energieverbrauchsverteilung

Die Speicherkosten haben mit  $439.582 \mu\text{J}$  einen Anteil von 72% am Gesamtenergieverbrauch von  $609.504 \mu\text{J}$ . Die restlichen 28% werden durch den Prozessor verursacht.

Zu Beginn des Programms *ref\_kernel.c* muss das globale Array *c* initialisiert werden. Dies geschieht durch die Funktion *Initialize\_Reference\_IDCT*. Wie man in *Abbildung 6.14* sehen kann, wird das Array *c* im Bereich  $c[0][0]$  bis  $c[0][7]$  immer mit dem gleichen Wert initialisiert, der Quadratwurzel aus  $0,125$ . Die erste Dimension des Arrays *c* bezeichnen wir hier mit Zeilen und die zweite Dimension mit Spalten. Das würde dann bedeuten, dass in der ersten Zeile des Arrays *c*, alle Spalten den gleichen Wert haben, nämlich die Quadratwurzel aus  $0,125$ .

```

1   for (freq=0; freq < 8; freq++)
2   {
3       scale = (freq == 0) ? sqrt(0.125) : 0.5;
4
5       for (time=0; time<8; time++) {
6           c[freq][time] = scale*
7               cos((PI/8.0)*freq*(time + 0.5));
8       }
9   }

```

*Abbildung 6.14: Programmcode Initialize\_Reference\_IDCT*

Da das Array im weiteren Programmablauf nur noch gelesen wird, hat man hier die Möglichkeit, die Größe des Arrays zu verringern. Man kann die erste Zeile komplett einsparen. Dadurch reduziert sich das Array von 8\*8 Feldern auf 7\*8 Felder. Durch die Reduzierung der Array-Größe verringert sich der Platzbedarf des Arrays im Speicher um 12,5%. Der Wert der ersten Zeile wird in einer neuen Variable *c\_1* vom Datentyp „double“ abgespeichert (*Abbildung 6.15*). Dadurch wird effektiv 10% weniger Speicherplatz für die Werte des Arrays *c* und der Variablen *c\_1* benötigt. Alle bisherigen Zugriffe auf die erste Zeile des Arrays *c* müssen noch angepasst werden in Zugriffe auf die neue Variable *c\_1*, damit das Programm weiterhin korrekt arbeitet.

```

1   c_1 = sqrt(0.125);
2
3   for (freq=1; freq < 8; freq++)
4   {
5       for (time=0; time<8; time++) {
6           c[freq-1][time] = 0.5*
7               cos((PI/8.0)*freq*(time + 0.5));
8       }
9   }

```

*Abbildung 6.15: Initialize\_Reference\_IDCT nach manueller Optimierung  
(Zeilenverringern des Arrays c)*

Die Zugriffe auf die übrigen Zeilen des Arrays *c* müssen insofern noch angepasst werden, als dass sich die Anzahl der Zeilen des Arrays um eins verringert hat.

Die Energieeinsparung, die durch diese Optimierung erreicht werden kann, beträgt 5,29% und die Anzahl der CPU-Zyklen verringert sich um 5,27% und hängt damit zusammen, dass der Datendurchsatz zwischen CPU und Speicher reduziert werden konnte.

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
Original	609.504,289	100,00	38.995.260	100,00
nach Zeilen- verringierung	577.233,734	94,71	36.938.383	94,73

Abbildung 6.16: Energieverbrauch und CPU-Zyklen im Vergleich

Es handelt sich hier zwar um eine manuell durchgeführte Optimierung, die nicht unbedingt bei anderen Anwendungen zum Einsatz kommen kann, aber die Energieeinsparung, die dadurch erzielt wird, spricht für eine Anwendung dieser Optimierung für das Programm *ref\_kernel.c*. Eine automatisierte Anwendung ist durch bestehende Verfahren noch nicht möglich, da es sich um sehr komplexe Änderungen im Programmcode handelt, bei denen Datenabhängigkeiten besonders berücksichtigt werden müssen.

Eine andere Technik wandelt die Zugriffe auf ein Array so um, dass über einen Zeiger auf das Array zugegriffen wird und nicht über eine Index-Variable. Um dieses Verfahren untersuchen zu können, wurde das zweidimensionale Array *c* (Abbildung 6.17) in ein eindimensionales Array (Abbildung 6.18) umgewandelt. Dieser Schritt alleine brachte keine Verbesserung im Energieverbrauch. Im Gegenteil, der Energieverbrauch erhöhte sich sogar um 0,40%.

```

1      for (k=0; k<8; k++)
2          partial_product+=
3          c[k][j]*block[8*i+k];

```

Abbildung 6.17: Original-Programmcode

```

1      for (k=0; k<8; k++)
2          partial_product+=
3          c[n++] *block[8*i+k];

```

Abbildung 6.18: modifizierter Programmcode

<i>Reference_IDCT</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)
Original	64.360,42430	100,00
eindimensionales Array	65.700,37829	102,08

Abbildung 6.19: Energieverbrauch der Funktion *Reference\_IDCT*

Der Energieverbrauch der einzelnen Funktion *Reference\_IDCT* stieg sogar um 2,08% an (Abbildung 6.19). Das liegt daran, dass für die Adressierung des eindimensionalen Arrays *c* die neue Variable *n* eingeführt werden muss und 3 Register mehr als sonst gespilt werden.

Die Umwandlung des zweidimensionalen Arrays in ein eindimensionales Array war als Vorbereitung nötig für den Zugriff - ausschliesslich über Zeiger - auf das Array. Dazu wird ein Zeiger vor Eintritt der entsprechenden Schleife auf den Index 0 des Arrays *c* gesetzt (Abbildung 6.20 – Zeile 1). In der innersten Schleife muss der Zeiger jedes Mal inkrementiert werden (Abbildung 6.20 – Zeile 6).

Durch die Umwandlung des Zugriffs auf die Werte des Arrays *c* über einen Zeiger wird nur eine minimale Energieeinsparung von insgesamt 0,06% erreicht. Im Gegensatz zum Original Programmcode wird keine der Schleifenvariablen, über die auf das Array *c* zugegriffen wird, eingespart.

```

1     dptr = &c[0];
2     for (j=0; j<8; j++)
3     {
4         for (k=0; k<8; k++)
5             partial_product+=
6                 *dptr++ * block[8*i+k];
7
8         tmp[8*i+j] = partial_product;
9     }
```

Abbildung 6.20: modifizierter Programmcode

Es wird sogar die zusätzliche Variable *dptr* eingeführt, über die als Zeiger auf die Arrayinhalte zugegriffen wird. Deshalb steigt der Energieverbrauch der Funktion *Reference\_IDCT* um 0,43%. Durch den Arrayzugriff über den Zeiger hat sich der Energieverbrauch für den Speicher um 0,11% verringert, während er für den Prozessor um 0,08% anstieg.

<i>ref_kernel.c</i>	Energieverbrauch (µJ)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
Original	609.504,289	100,00	38.995.260	100,00
nach <i>licm</i>	608.918,428	99,90	38.959.300	99,91
nach <i>Loop Reversal</i>	609.024,367	99,92	38.957.324	99,90
nach <i>Loop unrolling</i>	570.626,187	93,62	36.507.191	93,62
nach <i>Zeilenverringierung</i>	577.233,734	94,71	36.938.383	94,73
Eindimensionales Array <i>c</i>	611.919,332	100,40	39.153.542	100,41
Zugriff über Pointer	609.161,915	99,94	38.965.806	99,92

Abbildung 6.21: Energieverbrauch *ref\_kernel.c*

Die Anwendung der vorgestellten manuellen Arrayoptimierung (Zeilenverringern des Arrays  $c$ ) sollte auf jeden Fall durchgeführt werden. Die geringen Einsparungen durch die Reduzierung des zweidimensionalen Arrays auf ein eindimensionales und die anschließende Einführung des Zeigers sprechen eher gegen einen Einsatz. Interessant ist aber - wie bei den Schleifenoptimierungen - die spätere Kombination mit anderen Optimierungstechniken. Eine Übersicht findet sich in *Abbildung 6.21*.

Im nächsten Kapitel soll die nächste Ebene betrachtet werden, die *Medium-level-Intermediate-Representation (MIR) – Ebene*.



## 7 Optimierung auf MIR-Ebene

Für die Optimierungen auf der MIR-Ebene wird das Softwaretool *LANCE V2* eingesetzt, das am Informatik-Lehrstuhl 12 der Universität Dortmund entwickelt wurde [LA01].

*LANCE* erzeugt aus dem ihm übergebenen C-Source-Code eine *medium-level intermediate representation* (MIR) [MU97].

Nach der Erzeugung der MIR werden auf die MIR verschiedene Optimierungen angewendet. Das Optimierungspotenzial von Optimierungen auf der MIR-Ebene wird in diesem Kapitel untersucht.

Alle hier vorgestellten Optimierungen wurden auf das Programm *ref\_kernel.c* angewendet, das in Kapitel 5 schon vorgestellt wurde.

### 7.1 Überblick

Die hier betrachteten MIR-Optimierungen werden über das zu *LANCE V2* gehörende Skript *iropt* ausgeführt, das verschiedene Optimierungen in einer Schleife ausführt. Die Wahl der Optimierungen ist zielplattformabhängig. Der Ablauf der Optimierungen für den ARM-Prozessor ist in *Abbildung 7.1* dargestellt. Es existieren noch 2 weitere Optimierungen *LICM* und *Induction Variable Elimination (IVE)*, die für den ARM-Prozessor aber keine Verbesserungen bringen.

Zu Beginn des Skriptes wird für jede Optimierung eine Variable mit „TRUE“ initialisiert, die eine Ausführung der Optimierung bewirkt. Diese Variablen können im Laufe des Skriptes auf „FALSE“ oder auch „TRUE“ gesetzt werden, um den Optimierungsablauf zu beeinflussen. Hat die Variable für eine Optimierung den Wert „FALSE“, darf diese Optimierung nicht angewendet werden.

Das Skript führt eine Schleife aus, die solange durchlaufen wird, bis von keiner der ausgeführten Optimierungen bei einem Schleifendurchlauf eine Änderung der MIR erreicht werden konnte. Beim ersten Schleifendurchlauf wird jede Optimierung ausgeführt. Die Ausführung in folgenden Schleifendurchläufen ist abhängig von den gesetzten Variablen für jede Optimierung.

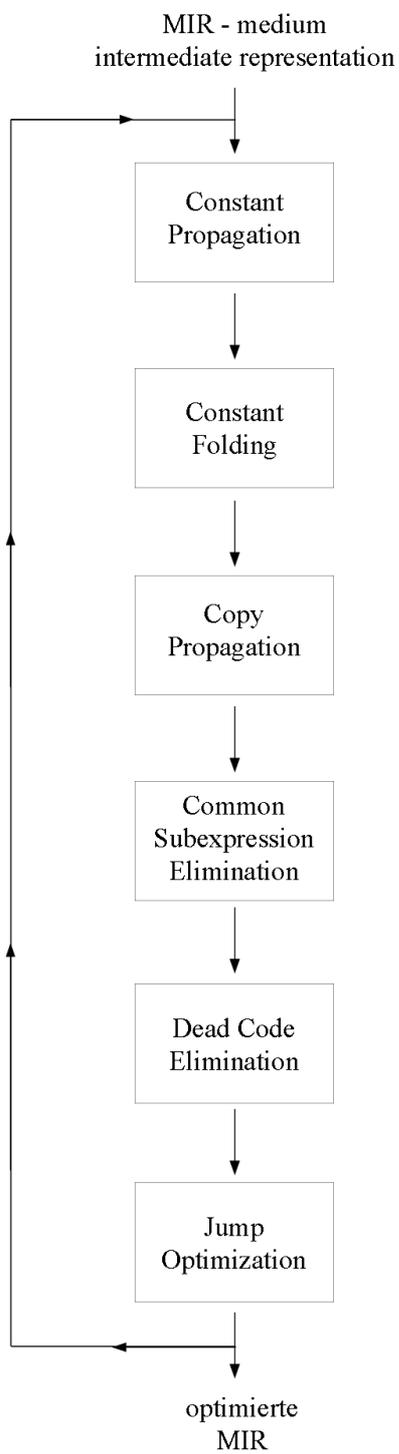


Abbildung 7.1: Ablauf der MIR Optimierungen für den ARM-Prozessor

Der folgende Graph stellt noch einmal die mögliche Reihenfolge beim *iropt*-Skript dar.

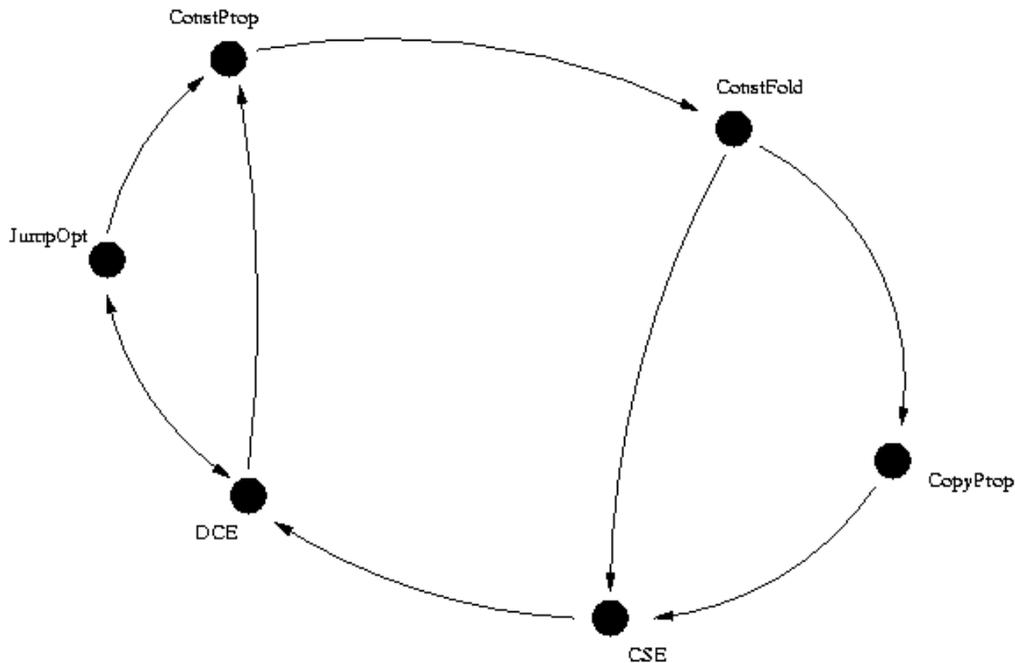


Abbildung 7.2: Ablaufreihenfolge beim *iropt*-Skript

## 7.2 Constant Propagation

Die erste Optimierung, die vom *iropt*-Skript aufgerufen wird, ist die *Constant Propagation*. Bei der *Constant Propagation* werden in der MIR Variablen gesucht, deren Werte konstant und bekannt sind. Ist eine solche Variable gefunden, kann jeder Ausdruck, in dem die Variable vorkommt, verändert werden.

```
b=3;
c=4*b;
e=a+c;
```

(a1) Beispiel-Programm

```
b=3;
c=4*3;
e=a+c;
```

(a2) nach *Constant Propagation*

An Beispiel (a) sieht man die Veränderungen, die durch die *Constant Propagation* erzeugt werden. Die Variable *b* wird durch ihren Wert 3 ersetzt. Diese Optimierung alleine bringt für das Programm *ref\_kernel.c* keine Einsparungen. Sie ist auch vielmehr als Vorbereitung der MIR für den nächsten Schritt, das *Constant Folding*,

gedacht. Nutzt man nur die *Constant Propagation* kommt es zu einer Verschlechterung beim Energieverbrauch von 0,02 %. Die Anzahl der CPU-Zyklen steigt ebenfalls um 0,02%. Das liegt daran, dass sich die Anzahl der IR Ausdrücke von 208 auf 210 vergrößert hat, weil die Optimierung davon ausgeht, dass nach ihr eine *Dead Code Elimination* angewendet wird.

Kann eine Änderung durch *Constant Propagation* erreicht werden, werden die zu Beginn erwähnten Variablen für ausgewählte Optimierungen auf „TRUE“ gesetzt, sodass diese Optimierungen ausgeführt werden dürfen. Zu den ausgewählten Optimierungen gehören bei der *Constant Propagation* das *Constant Folding*, die *Dead Code Elimination* und die *Common Subexpression Elimination*. Außerdem wird die Variable für die *Constant Propagation* auf „FALSE“ gesetzt. Das Setzen der eigenen Variable auf „FALSE“ ist unabhängig von einer erreichten Veränderung der MIR und wird von jeder Optimierungstechnik durchgeführt.

### 7.3 Constant Folding

Bei der nächsten Optimierung, dem *Constant Folding*, werden mehrere Konstanten eines Ausdrucks zu einer Konstanten zusammengefasst.

```
b=3 ;  
c=4 * 3 ;  
e=a+c ;
```

(b1) Beispiel-Programm

```
b=3 ;  
c=12 ;  
e=a+c ;
```

(b2) nach *Constant Folding*

In Beispiel (b) geschieht dies mit dem Ausdruck „ $c=4 * 3$ “;“. Für diesen Ausdruck kann der Wert des Ausdrucks bestimmt werden und durch das Ergebnis ersetzt werden. Die der MIR-Generierung und -Optimierung folgende Codegenerierung spart so eine Multiplikationsanweisung ein.

Führt man nur diese Optimierung für die MIR des Programms *ref\_kernel.c* aus, ergibt sich keine Änderung für Energieverbrauch und Performance.

Kombiniert man eine *Constant Propagation* mit einem anschließenden *Constant Folding*, erreicht man für das hier untersuchte Programm eine Verschlechterung von 0,07 % beim Energieverbrauch sowie 0,06 % Erhöhung der benötigten Anzahl an CPU-Zyklen. Auch hier vergrößert sich die Anzahl der IR Ausdrücke aus dem gleichen Grund wie bei alleiniger Anwendung der *Constant Propagation*.

Kann beim *Constant Folding* eine Änderung der MIR erreicht werden, wird der *Constant Propagation* und der *Common Subexpression Elimination (CSE)* die Ausführung erlaubt.

## 7.4 Copy Propagation

Es folgt nun die Betrachtung der *Copy Propagation*. In Beispiel (c) sieht man, dass die Variable  $b$  den Wert der Variablen  $d$  zugewiesen bekommt und dass die Variable  $b$  im weiteren Programm benutzt wird. Ähnlich wie bei der *Constant Propagation* wird jedes weitere Auftreten der Variablen  $b$  durch die Variable  $d$  ersetzt, unter der Voraussetzung, dass die Variable  $b$  im weiteren Programmablauf nicht verändert wird.

```
b=d;
c=4*b;
e=a+c;
```

(c1) Beispiel-Programm

```
b=d;
c=4*d;
e=a+c;
```

(c2) nach *Copy Propagation*

Das führt dazu, dass der Ausdruck in der ersten Zeile des Beispiel (c) nicht mehr benötigter Code ist und deshalb jetzt eine *Dead Code Elimination* angewendet werden könnte.

Kann durch *Copy Propagation* eine Änderung der MIR erreicht werden, werden die Variablen für *CSE* und *DCE* auf „TRUE“ gesetzt.

Auch durch die Anwendung der *Copy Propagation* konnte keine Verbesserung des Energieverbrauchs erreicht werden.

## 7.5 Common Subexpression Elimination

Durch den Einsatz der *Common Subexpression Elimination (CSE)* werden mehrmals benutzte Ausdrücke gesucht und durch eine einzelne Variable ersetzt.

```
b=d+2;
c=4*b;
e=d+2;
```

(d1) Beispiel-Programm

```
t1=d+2
b=t1;
c=4*b;
e=t1;
```

(d2) nach *CSE*

An Beispiel (d) wird die Vorgehensweise deutlich. Der Ausdruck „ $d+2$ “ kommt in diesem Beispiel zweimal vor und kann zu Beginn des Programms in einer neuen Variable  $t1$  gespeichert werden. Jedes weitere Auftreten von „ $d+2$ “ wird nun durch  $t1$  ersetzt. Die Addition müsste in diesem Beispiel nur einmal ausgeführt werden und nicht zweimal.

Die *CSE* setzt bei durchgeführter Veränderung der MIR die Variablen für die *Constant Propagation*, *Copy Propagation* und *DCE* erneut auf „TRUE“. Eine Energieeinsparung tritt auch mit dieser Optimierung nicht ein.

## 7.6 Dead Code Elimination

Die *Dead Code Elimination (DCE)* sucht im nächsten Schritt nicht mehr benötigte Ausdrücke in der MIR und eliminiert diese.

```
b=d;  
c=4*d;  
e=a+c;
```

(e1) Beispiel-Programm

```
c=4*d;  
e=a+c;
```

(e2) nach *DCE*

In Beispiel (e) ist es der Ausdruck „b=d“, der entfernt werden kann, wenn die Variable *b* auch im weiteren Programmablauf nicht mehr benötigt wird.

Die *DCE* setzt die Variablen für *Constant Propagation*, *Copy Propagation* und *CSE* auf „TRUE“.

## 7.7 Jump Optimization

Als letzter Schritt in jedem Schleifendurchlauf wird die *Jump Optimization* angewendet. Die *Jump Optimization* wird in jedem Fall am Ende eines Schleifendurchlaufs aufgerufen.

Die *Jump Optimization* dient der Optimierung von Sprüngen in der MIR. Sie wertet Sprungbedingungen aus und kann diese verändern. Durch geschickte Änderungen können bestimmte Sprungziele überflüssig werden. Deshalb ist es auch sinnvoll, dass die *Jump Optimization* nach Änderung der MIR die Variable für die *DCE* auf „TRUE“ setzt.

## 7.8 Fazit

Das Programm *ref\_kernel.c* benötigte 0,15 % mehr Energie und 0,14 % mehr CPU-Zyklen für die Ausführung, wenn das *iropt*-Skript benutzt wurde (*Abbildung 7.4*). Das liegt daran, dass nach Anwendung des *iropt*-Skriptes kein optimaler Assemblercode generiert wird (*Abbildung 7.3*).

<pre> ADD r0,r7,#1 MOV r7,r0 CMP r0,#8 BGE _M_21 _M_23   B LL8_0 _M_21 </pre> <p>mit <i>iropt</i></p>	<pre> ADD r7,r7,#1 CMP r7,#8 BGE LL10_7 _M_37   B LL16_7 LL10_7 </pre> <p>ohne <i>iropt</i></p>
---	---

Abbildung 7.3: Unterschiede im Assemblercode

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
ohne <i>iropt</i>	609.504,289	100,00	38.995.260	100,00
mit <i>iropt</i>	610.445,773	100,15	39.050.309	100,14

Abbildung 7.4: Energieverbrauch *ref\_kernel.c*

Es scheint sinnvoll, auf eine Optimierung der MIR bei Betrachtung der Programms *ref\_kernel.c* zu verzichten, da keine Verbesserung des Energieverbrauchs erreicht werden kann. Es ist aber möglich, dass die Optimierung der gesamten MPEG2-Dekoder Software Einsparungen bringt. Das konnte aber nicht untersucht werden, da die MIR der gesamten MPEG2-Dekoder Software zu diesem Zeitpunkt noch nicht mit dem *encc* kompiliert werden konnte.

Im nächsten Kapitel sollen die Möglichkeiten zur Optimierung auf der *low-level intermediate representation (LIR)* Ebene bzw. Assemblercode-Ebene betrachtet werden.



## 8 Optimierung auf LIR- und Assembler-Ebene

Die LIR-Ebene ist nach [MU97] die unterste Ebene der Zwischendarstellungen. Die *low-level intermediate representation (LIR)* wurde in dieser Diplomarbeit mit dem am Informatik-Lehrstuhl 12 der Universität Dortmund entwickelten Tool *encc (energy aware c compiler)* [EN02] generiert und optimiert. Bei dem Tool *encc* handelt es sich um das Backend der Codeerzeugung, das als Eingabedatei die Ausgabe des Tools *LANCE*, die optimierte *MIR*, benutzt.

Der *encc* führt mehrere Phasen aus. Die erste Phase besteht in der Auswahl der benötigten Maschinenbefehle für die in der *MIR* enthaltenen Ausdrücke. Diese Phase nennt man *Codeselection*. Die Maschinenbefehle werden mittels einer Kostenfunktion ausgewählt, der das in Kapitel 3.3.1 erwähnte Energiemodell zugrunde liegt. Somit wird aus allen möglichen Maschinenbefehlen für einen *MIR*-Ausdruck der jeweils energiekostengünstigste Befehl oder eine Befehlsfolge ausgewählt. In dieser Phase werden noch virtuelle Register benutzt. Die Anzahl der virtuellen Register ist nicht beschränkt.

In der zweiten Phase, dem *Instructionscheduling*, wird untersucht, ob die Änderung der Reihenfolge der Befehlsausführung zu Optimierungen führen kann. Man kann durch solche Befehlsverschiebungen z.B. die Nutzung physikalischer Register einsparen, wie in [ST02] an einem Beispiel gezeigt wird. Dadurch kann ein ansonsten notwendiges *Spilling* eines Registers entfallen. *Spilling* bedeutet, dass der Inhalt eines Registers über einen STORE-Befehl im Speicher abgelegt wird und später wieder mit einem LOAD-Befehl aus dem Speicher in ein Register eingelesen wird. Das *Spilling* verursacht durch diese Befehle zusätzliche Kosten. Diese Phase kann beim Programmaufruf des *encc* ein- bzw. ausgeschaltet werden.

Die dritte Phase ist die *Registerallokation*, in der eine Zuordnung der virtuellen Register zu den physikalischen Registern der verwendeten Hardware gesucht wird. Dabei kann es vorkommen, dass bei nicht ausreichender Anzahl an physikalischen Registern „gespillt“ werden muss.

In der letzten Phase können zusätzliche Optimierungen ausgeführt werden, die in diesem Kapitel am Beispiel von Assembler-Code vorgestellt werden. Außerdem wird gezeigt, welche Reduzierung des Energieverbrauchs durch den Einsatz der vorhandenen Optimierungen erzielt werden kann. Anschließend werden weitere Möglichkeiten der Optimierung der *LIR* bzw. des Assembler-Codes für das Programm *ref\_kernel.c* untersucht.

## 8.1 encc Backend Optimierungen

Im Backend sind die 3 Standardoptimierungen *Constant Folding*, *Constant Propagation* und *Copy Propagation* fest integriert. Im Vergleich dazu sind beim ARM Compiler *tcc* z.B. folgende Optimierungen enthalten [ARM02]:

- *Common Subexpression Elimination (CSE)*
- *Loop Invariant Code Motion (LICM)*
- *Constant Folding*
- *Table Driven Peepholing*

Während der *Codeselection* wird beim *encc* die in Kapitel 3 vorgestellte Optimierung *Constant Folding* durchgeführt. Dies geschieht auf der *LIR* Ebene vor der Erzeugung des Assembler-Codes. Zusätzlich erfolgt eine *Constant Propagation*. In der Phase *Registerallokation* wird eine *Copy Propagation* angewendet. Zusätzlich zu diesen fest integrierten Optimierungen kann man über Optionen beim Programmaufruf des *encc* Optimierungen wie *Instructionscheduling*, *Registerpipelining* oder *LICM* ein- bzw. ausschalten.

*Hornbach* hat in seiner Diplomarbeit [HO01] alle Optimierungen<sup>8</sup> des *encc* deaktiviert, um herauszufinden, welchen Einfluss sie auf den Energieverbrauch von Programmen haben. Dazu führte er verschiedene Optimierungsläufe durch. Im ersten Durchlauf wurde auf alle Optimierungen verzichtet. Um die *encc\_opt* deaktivieren zu können, mussten die Quelldateien des *encc* manipuliert und der *encc* neu kompiliert werden. In einem weiteren Durchlauf wurden die *encc\_opt* benutzt. Die Werte der benötigten CPU-Zyklen und des Energieverbrauchs dieser beiden Durchläufe wurden anschliessend miteinander verglichen. *Abbildung 8.1* stellt die von *Hornbach* erzielten durchschnittlichen Ergebnisse dar.

	Energieverbrauch	CPU-Zyklen
Ohne Optimierungen	100,00%	100,00%
Mit <i>encc_opt</i>	87,64%	86,59%

*Abbildung 8.1: Verbesserung durch encc\_opt [HO01]*

*Hornbach* hat Messungen mit verschiedenen Beispielprogrammen durchgeführt [HO01]. Das Programm mit dem höchsten Energieverbrauch bei *encc\_opt* ist der

---

<sup>8</sup> Diese Optimierungen werden im folgenden Text mit *encc\_opt* bezeichnet.

Bubble Sort-Algorithmus, der 6.597,033  $\mu\text{J}$  verbraucht, während *ref\_kernel.c* bei *encc\_opt* 609.504,289  $\mu\text{J}$  verbraucht. Somit verbraucht der Bubble Sort-Algorithmus nur knapp 1,1% der Energie, die das Programm *ref\_kernel.c* verbraucht. Daher kann man sagen, dass diese Programme im Vergleich zu dem hier betrachteten *ref\_kernel.c* sehr kleine Programme sind. Würde man auch bei *ref\_kernel.c* die gleichen Einsparungen erzielen, läge der Energieverbrauch ohne *encc\_opt* bei geschätzten 695.463,5885  $\mu\text{J}$ .

Bei dem Programm *ref\_kernel.c* handelt es sich aber um ein Programm mit intensivem Datentransfer zwischen Prozessor und Speicher, wodurch es sich von den in [HO01] benutzten Programmen unterscheidet. Die *encc\_opt* wurden - wie in [HO01] beschrieben - deaktiviert und der Energieverbrauch und die benötigten CPU-Zyklen ermittelt, um die Annahmen bezüglich *ref\_kernel.c* zu überprüfen.

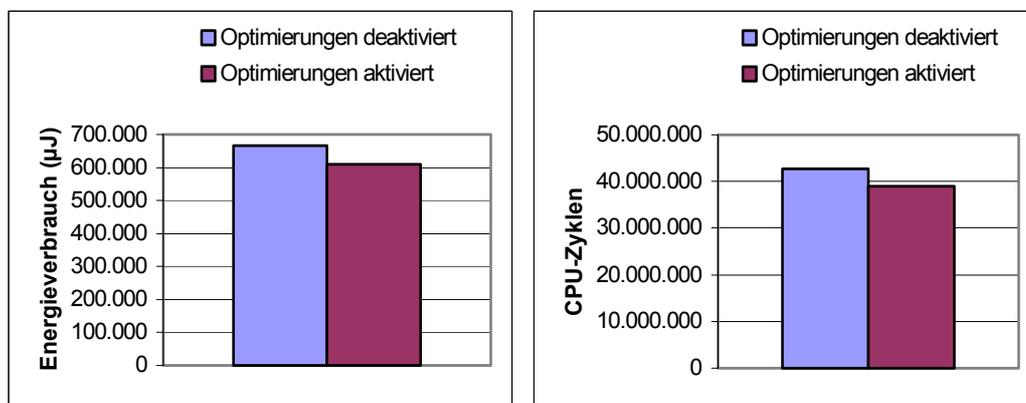


Abbildung 8.2: *ref\_kernel.c* und *encc*-Optimierungen

Wie man in *Abbildung 8.2* sehen kann, hat man ohne *encc\_opt* einen höheren Energieverbrauch, als wenn die *encc\_opt* genutzt werden. Das gleiche gilt für die Anzahl der benötigten CPU-Zyklen.

<i>ref_kernel.c</i>	Energieverbrauch	CPU-Zyklen
Ohne Optimierungen	100,00%	100,00%
Mit <i>encc_opt</i>	91,50%	91,45%

Abbildung 8.3: *ref\_kernel.c* und *encc*-Optimierungen

*Abbildung 8.3* zeigt die prozentuale Veränderung des Energieverbrauchs und der CPU-Zyklen, wenn auf die *encc\_opt* verzichtet wird. In beiden Fällen, also der Benutzung der *encc\_opt* und dem Verzicht auf *encc\_opt*, wurde keine vorherigen Optimierungen auf einer höheren Ebene durchgeführt.

Die *encc\_opt* erreichen also auch für das hier betrachtete Programm *ref\_kernel.c* eine starke Verbesserung des Energieverbrauchs und der CPU-Zyklen.

## 8.2 Weitere Optimierungen

Das eigentliche Ziel der Arbeit von *Hornbach* war die Implementierung von Standardoptimierungen auf *LIR* Ebene. Bei der in [HO01] betrachteten *LIR* handelt es sich um die objektorientierte *GeLIR*-Klassenbibliothek nach [GE01]. Zu den implementierten *GeLIR* Optimierungen zählen *Constant Folding*, *Constant Propagation*, *Copy Propagation*, *Dead Code Elimination*, *Redundant Load Elimination* und *Redundant Store Elimination*. *Hornbach* konnte ähnliche Ergebnisse erreichen, wie durch die Verwendung der *encc\_opt* (Abbildung 8.4).

	Energieverbrauch	CPU-Zyklen
Ohne Optimierungen	100,00%	100,00%
Mit <i>encc_opt</i>	87,64%	86,59%
<i>GeLIR</i> Optimierungen	88,11%	87,55%

Abbildung 8.4: Verbesserung durch *GeLIR*-Optimierungen

Da die *GeLIR*-Optimierungen in Einzelfällen sogar bessere Resultate erzielten als die *encc*-Optimierungen, wie es beim Algorithmus Heap-Sort vorkam, wäre es möglich, dass eine Anwendung der *GeLIR*-Optimierungen auf das Programm *ref\_kernel.c* weitere Einsparungen bringen könnte. Andererseits waren die *GeLIR*-Optimierungen in Einzelfällen auch schlechter als die *encc*-Optimierungen, sodass es auch zutreffen könnte, dass sich der Energieverbrauch des *ref\_kernel.c* verschlechtern würde. Dies wurde hier aber nicht weiter untersucht, sondern soll nur zeigen, dass mit den *GeLIR*-Optimierungen eine weitere Möglichkeit zur Optimierung existiert.

Der *encc* gibt als Resultat Assemblercode für den ARM7-Prozessor aus. Im nächsten Schritt wurde der generierte Assemblercode für das Programm *ref\_kernel.c* untersucht, um weitere Möglichkeiten zur Optimierung zu finden.

Bei der Analyse des Assemblercodes fiel der Ausdruck in Beispiel (a1) auf. Der erste Befehl kopiert den Inhalt von Register *r1* nach Register *r2*. Der zweite Befehl kopiert anschließend den Inhalt von Register *r2* nach Register *r1*. Hier wird deutlich, dass die Inhalte der beiden Register schon nach dem ersten Kopierbefehl die richtigen Inhalte aufweisen.

<pre>MOV r2, r1 MOV r1, r2 BL cos</pre>	<pre>MOV r2, r1 BL cos</pre>
(a1) Original-Assemblercode	(a2) optimierter Assemblercode

Der zweite Kopierbefehl kann also eingespart werden. Beispiel (a2) zeigt den optimierten Assemblercode nach Einsparung des zweiten Kopierbefehls.

Ein weiterer Ausdruck mit der Möglichkeit zur Optimierung ist in Beispiel (b1) zu sehen.

<pre>LDR r0, [SP, #4] MOV r1, r0 LDR r0, [SP, #0]</pre>	<pre>LDR r1, [SP, #4] LDR r0, [SP, #0]</pre>
(b1) Original-Assemblercode	(b2) optimierter Assemblercode

In das Register *r0* wird mit dem ersten Befehl der Inhalt der Speicheradresse, auf die der Stackpointer zeigt, mit einem Offset von 4 Byte geladen. Dieser Wert wird im zweiten Befehl von Register *r0* nach Register *r1* kopiert. Der dritte Befehl lädt den Inhalt der Adresse, auf die der Stackpointer zeigt, ohne Offset in das Register *r0*. In diesem Beispiel kann der Kopierbefehl eingespart werden, wenn man den ersten Befehl entsprechend anpasst. Dazu lädt man mit dem ersten Befehl den Inhalt der Stackpointer-Speicheradresse mit Offset 4 direkt in das Register *r1*. Der Inhalt wird nämlich nie in Register *r0* benötigt, da dieses Register im nächsten Schritt mit einem anderen Wert beschrieben wird.

Beispiel (c1) zeigt den nächsten Ausdruck.

<pre>ADD r0, r6, #1 MOV r6, r0 CMP r0, #8 ...</pre>	<pre>ADD r6, #1 CMP r6, #8 ...</pre>
(c1) Original-Assemblercode	(c2) optimierter Assemblercode

Der Wert aus Register *r6* wird um die Konstante 1 erhöht und dann in Register *r0* abgelegt. Der nächste Befehl kopiert den Wert aus Register *r0* in Register *r6*. Anschließend wird der Inhalt von Register *r0* mit der Konstanten 8 verglichen. Im weiteren Datenfluss wird das Register *r0* in jedem Fall mit neuem Inhalt beschrieben, sodass es unerheblich ist, ob *r0* den Inhalt von *r6* enthält. Daher ist die angewendete

Optimierung gültig. Man addiert die Konstante 1 direkt auf den Inhalt des Registers *r6* auf und führt den Vergleich mit der Konstanten 8 auch mit dem Inhalt des Registers *r6* durch. Dadurch kann man erneut einen Kopierbefehl einsparen.

Eine Optimierung von bedingten Sprüngen stellt Beispiel (d2) dar.

<pre>                 ADD r7, #1                 CMP r7, #8                 BLT LL8_0 _M_18                 B LL1_0 LL8_0                 ...             </pre>	<pre>                 ADD r7, #1                 CMP r7, #8                 BGE LL1_0 _M_18                 <del>B LL8_0</del> LL8_0                 ...             </pre>
(d1) Original-Assemblercode	(d2) optimierter Assemblercode

In (d1) wird achtmal überprüft, ob der Inhalt von Register *r7* kleiner als 8 ist. Ist *r7* kleiner als 8, wird das Label *LL8\_0* als nächstes angesprungen (*BLT* – *branch less than*). Zwischen dem bedingten Sprung und dem Sprungziel existiert nur ein einziger Befehl, nämlich der unbedingte Sprung zum Label *LL1\_0*. Das Label *LL1\_0* wird nur angesprungen, wenn *r7* nicht mehr kleiner als 8 ist, also größer oder gleich als 8 ist. Im ARM-Thumb Befehlssatz existiert auch der Befehl, eine Verzweigung durchzuführen, wenn ein „größer oder gleich“-Vergleich zutrifft. Dieser Befehl hat das Mnemonic *BGE* und bedeutet *branch greater equal*. Man ersetzt den Befehl *BLT* durch den invertierten Befehl *BGE* und vertauscht die Sprungziele *LL8\_0* und *LL1\_0*. Durch diese Änderung entsteht neuer *Deadcode*, nämlich der unbedingte Sprung zum Label *LL8\_0*, das sich direkt hinter diesem unbedingten Sprung befindet. Daher kann man den unbedingten Sprung *B LL8\_0* eliminieren.

Eine *Redundant Load Elimination* konnte im Beispiel (e2) angewendet werden.

<pre>                 ADD r0, #1                 STR r0, [SP, #516]                 LDR r0, [SP, #516]                 CMP r0, #8             </pre>	<pre>                 ADD r0, #1                 STR r0, [SP, #516]                 CMP r0, #8             </pre>
(e1) Original-Assemblercode	(e2) optimierter Assemblercode

Man kann direkt erkennen, dass der *LDR*-Befehl überflüssig ist, da der Inhalt von Register *r0* einen Befehl vorher auf dem Stack abgelegt und nicht verändert wird. Deshalb ist es erlaubt, den *LDR*-Befehl an dieser Stelle zu eliminieren.

Die erwähnten Optimierungen des Assembler-Codes aus den Beispielen (a)-(d) führten nur zu einer geringen Optimierung.

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
ohne ASM-Optimierungen	609.504,289	100,00	38.995.260	100,00
mit ASM-Optimierungen	608.932,767	99,91	38.958.277	99,91

Abbildung 8.5: Energieverbrauch *ref\_kernel.c*

Das liegt daran, dass die Optimierungen sich nicht sehr oft in dem Assembler-Code des Programms *ref\_kernel.c* anwenden lassen. Nur die in (d) vorgestellte Optimierung ist zur Zeit im *encc* implementiert.

*Stouraitis* untersucht im SOFLOPO-Projekt [SO01] die Optimierung von Assembler-Code für ARM-Prozessoren. Dabei wurde neben dem ARM-Befehlssatz auch der hier betrachtete Thumb-Befehlssatz untersucht. Während des SOFLOPO-Projektes entstand eine Software, mit der es möglich ist, Assembler-Code automatisiert zu optimieren und anschließend den Binärcode mittels des Prozessors auf einem ARM Development Board auszuführen, um die geschätzten Werte mit realen Werten vergleichen zu können. Dabei muss berücksichtigt werden, dass es sich bei dem Development Board nicht um das Atmel AT91EB01 Development Board [ATM98] handelt, sondern um ein spezielles PCMCIA Development Board eines Atmel AT76C502 WLAN MAC Controller [SO01]. Um den Energieverbrauch für den ARM7-Core messen zu können, musste außerdem das benutzte PCMCIA Development Board modifiziert werden.

Bei dem Assembler-Code, der optimiert wurde, handelt es sich um eine kommerzielle Implementierung des *IEEE 802.11 Wireless Multimedia Protocol* [SO01]. Der Assembler-Code besteht aus 16 einzelnen Assembler-Dateien. Jede Datei wurde einzeln betrachtet. Durch das entwickelte Software-Tool konnte eine Optimierung von 9,17% für den Energieverbrauch der *IEEE 802.11* Software erreicht werden. In [SO01] wird aber nicht beschrieben, mit welchen Tools der Assembler-Code erzeugt worden ist. Auch fehlen jegliche Aussagen zu eventuell eingesetzten bzw. nicht eingesetzten Optimierungen vor dem Zeitpunkt der Generierung des Assembler-Codes. Unklar ist auch, ob sich die Einsparung von 9,17% auf die gesamte *IEEE 802.11* Software bezieht.

Wie man gesehen hat, kann man mit Optimierungen auf der LIR- und der Assembler-Ebene gute Optimierungen erreichen (*Abbildung 8.6*). Deshalb sollte man alle zur Verfügung stehenden Optimierungen ausnutzen, um energieoptimierten Code zu erhalten.

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
ohne <i>encc_opt</i>	666.093,582	100,00	42.640.855	100,00
mit <i>encc_opt</i>	609.504,289	91,50	38.995.260	91,45
ohne ASM- Optimierungen	609.504,289	100,00	38.995.260	100,00
mit ASM- Optimierungen	608.932,767	99,91	38.958.277	99,91

*Abbildung 8.6: Energieverbrauch ref\_kernel.c*

Im nächsten Kapitel wird gezeigt, welche Möglichkeiten der Energieeinsparung durch die Wahl der Speicherhierarchie existieren.

## 9 Speicherhierarchie

Als nächste Optimierungsmöglichkeit wird die Hardwareebene untersucht. Auf dieser Ebene soll die Nutzung von Scratch-Pad- und Cache-Speicher betrachtet werden. Bei diesen beiden Speichern handelt es sich um OnChip-Speicher. Zugriffe auf diesen Speichertyp sind beim Energieverbrauch günstiger als Zugriffe auf den Hauptspeicher [LE01]. Die Nutzung von OnChip-Speichern kann daher zu Energieeinsparungen führen. Untersucht hat dies *Lee* in [LE01] für unterschiedliche Scratch-Pad- und Cache-Speichergrößen und unterschiedliche Cache-Organisationen.

Scratch-Pad- und Cache-Speicher sind OnChip-Speicher, die sich in ihrer Anwendung und in ihrem Aufbau unterscheiden. Während der Scratch-Pad-Speicher softwaremäßig angesteuert wird, übernimmt zusätzliche Hardware die Steuerung des Cache-Speichers. Welche Energieeinsparungen erreicht werden können hat *Lee* an 8 Beispielprogrammen gezeigt [LE01].

In Kapitel 9.1 soll für das Programm *ref\_kernel.c* untersucht werden, welche Scratch-Pad-Speichergröße die beste Einsparung bringt.

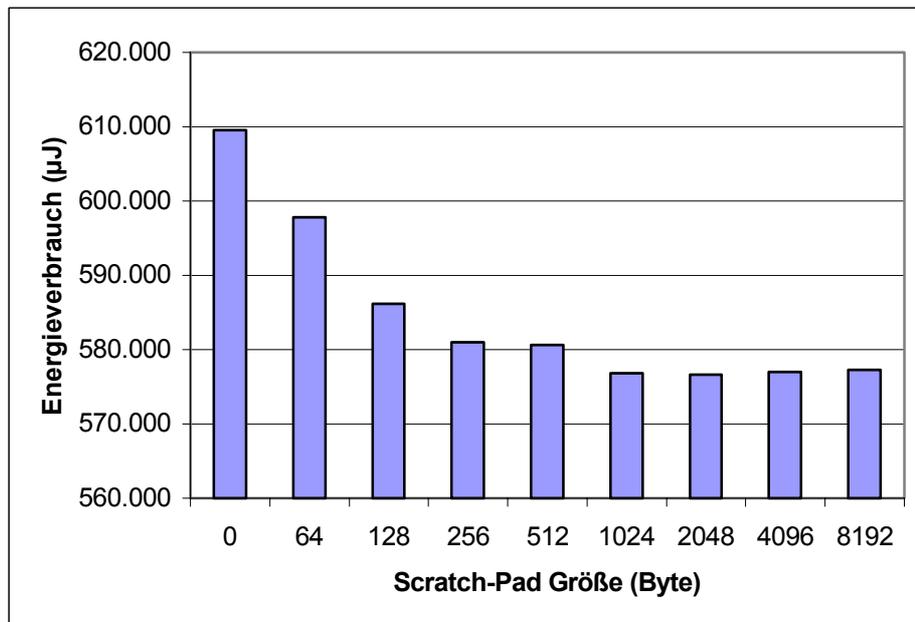
Der Einfluss der Größe des Cache-Speichers soll in Kapitel 9.2 gezeigt werden. Dabei wird nur eine Organisation von Cache-Speichern betrachtet und nicht wie in [LE01] fünf verschiedene Cache-Assoziativitäten.

### 9.1 Scratch-Pad

Die Nutzung des OnChip-Speichers als Scratch-Pad-Speicher erfordert für den hier betrachteten Ansatz einen speziellen Algorithmus, der entscheidet, welche Programmteile vom OffChip-Speicher in den OnChip-Speicher verschoben werden sollen. Diese Entscheidung findet zum Zeitpunkt der Codegenerierung statt und legt die Verteilung der einzelnen Programmteile auf die beiden Speicherarten vor der Programmausführung fest. Es handelt sich hierbei um einen statischen Algorithmus.

Dieser statische Scratch-Pad Algorithmus wird in der Diplomarbeit von *Zobiegala* [ZO01] und in [DATE02] näher betrachtet. Ein dynamischer Scratch-Pad Algorithmus wird zur Zeit von *Grunwald* im Rahmen seiner Diplomarbeit [GR02] entwickelt und bald verfügbar sein.

Unter Verwendung des statischen Scratch-Pad-Algorithmus wurde der Energieverbrauch des Programms *ref\_kernel.c* für die Scratch-Pad-Größen 0, 64, 128, 256, 512, 1024, 2048, 4096 und 8192 Byte untersucht. Den Energieverbrauch für *ref\_kernel.c* bei unterschiedlichen OnChip-Größen zeigt *Abbildung 9.1* (Hinweis: Wegen der geringen absoluten Wertänderungen wurde die Skala für den Energieverbrauch angepasst und beginnt deshalb mit 560.000  $\mu\text{J}$  statt mit 0  $\mu\text{J}$ ).



*Abbildung 9.1: Energieverbrauch für ref\_kernel.c bei Nutzung von Scratch-Pad-Speicher*

Man erreicht schon mit kleiner Scratch-Pad-Größe eine Energieeinsparung. Ab der Scratch-Pad-Größe von 1024 Byte hat sich der Energieverbrauch für das Programm *ref\_kernel.c* zwischen 575.173  $\mu\text{J}$  und 576.135  $\mu\text{J}$  eingependelt. Das liegt daran, dass der Scratch-Pad Algorithmus ab einer Scratch-Pad-Größe von 2048 Byte keine weiteren Programmteile gefunden hat, die er in den Scratch-Pad verlagern konnte. Nur bei einer Scratch-Pad-Größe von 8192 Byte kann noch eine zusätzliche Variable in den Scratch-Pad verlegt werden. Trotzdem hat man bei 8192 Byte keine Verringerung beim Energieverbrauch mehr, da die Energiekosten pro Zugriff bei dieser OnChip-Speichergröße am höchsten sind.

Auch die Anzahl der CPU-Zyklen erreicht ab einer Scratch-Pad-Größe von 1024 Byte keine großen Veränderungen mehr. In [LE01] wird gesagt, dass der Energieverbrauch pro individuellem Zugriff auf den Scratch-Pad-Speicher sich bei steigender Scratch-Pad Größe erhöht (*Abbildung 9.2*). Unter dieser Berücksichtigung sollte man die Größe des Scratch-Pad-Speichers so wählen, dass die Ausnutzung möglichst effizient ist. Für das Programm *ref\_kernel.c* zeigte sich, dass bei einer Größe von 1024 Byte noch die volle Ausnutzung des OnChip-Speichers erreicht werden konnte. Ab 2048 Byte wurde der OnChip-Speicher aber nicht mehr voll ausgenutzt.

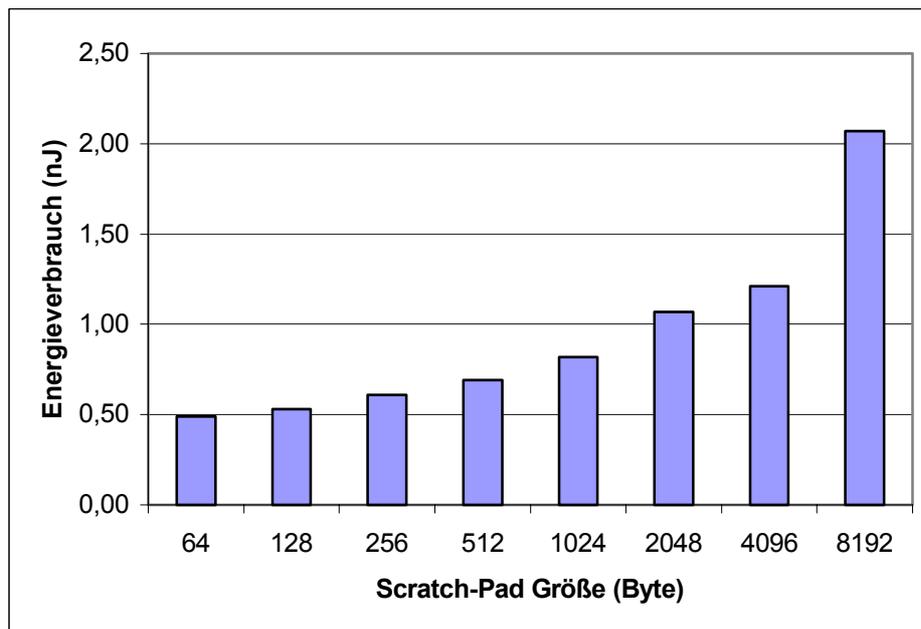


Abbildung 9.2: Energieverbrauch pro Scratch-Pad Zugriff [LE01]

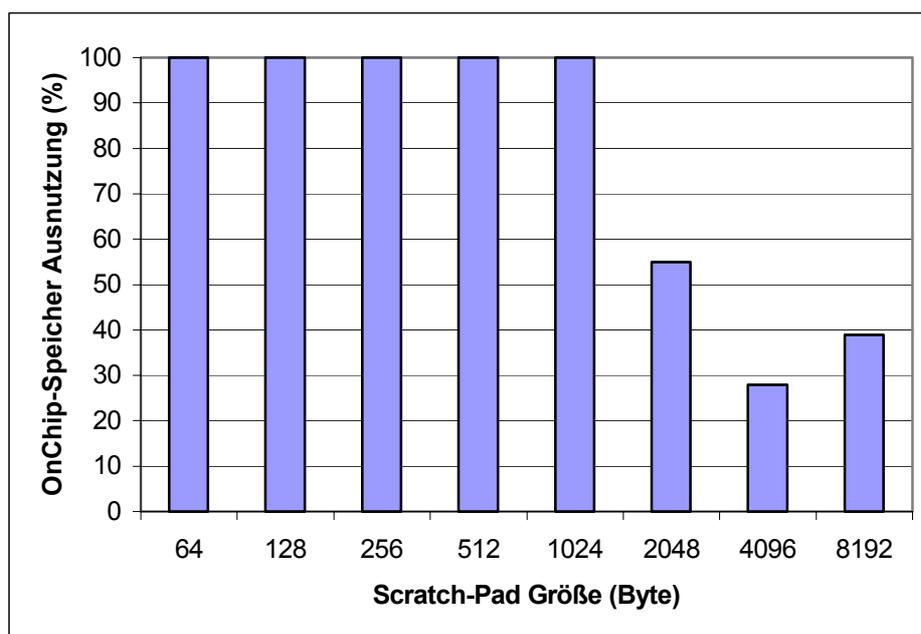


Abbildung 9.3: OnChip-Speicher Ausnutzungsrate für `ref_kernel.c`

Der Einsatz eines Scratch-Pad-Speichers als OnChip-Speicher bringt eine maximale Energieeinsparung von bis zu 5,47%. *Abbildung 9.3* zeigt, dass der OnChip-Speicher ab einer Größe von 2048 Byte noch Speicherplatz zur Verfügung hat. Z.B. könnten die in `ref_kernel.c` verwendeten *floating-point* Bibliotheksfunktionen in den Scratch-Pad verlagert werden. Das Problem ist nur, dass der Scratch-Pad-Algorithmus in der derzeitigen Implementierung keine Kenntnis über die Größe der Bibliotheksfunktionen besitzt und daher nicht entscheiden kann, ob er eine

Bibliotheksfunktion in den OnChip-Speicher verlagern kann oder nicht. Würden diese in den Scratch-Pad verlagert werden, hätte man eine wesentlich höhere Einsparung als 5,47%. Die Energieeinsparung für die einzelnen Bibliotheksfunktionen wird bei Scratch-Pad-Nutzung auf ungefähr 70% abgeschätzt. Die Bibliotheksfunktionen haben einen Anteil von ca. 75% am Gesamtenergieverbrauch. Man könnte daher schätzungsweise den Energieverbrauch um maximal 77,50% reduzieren.

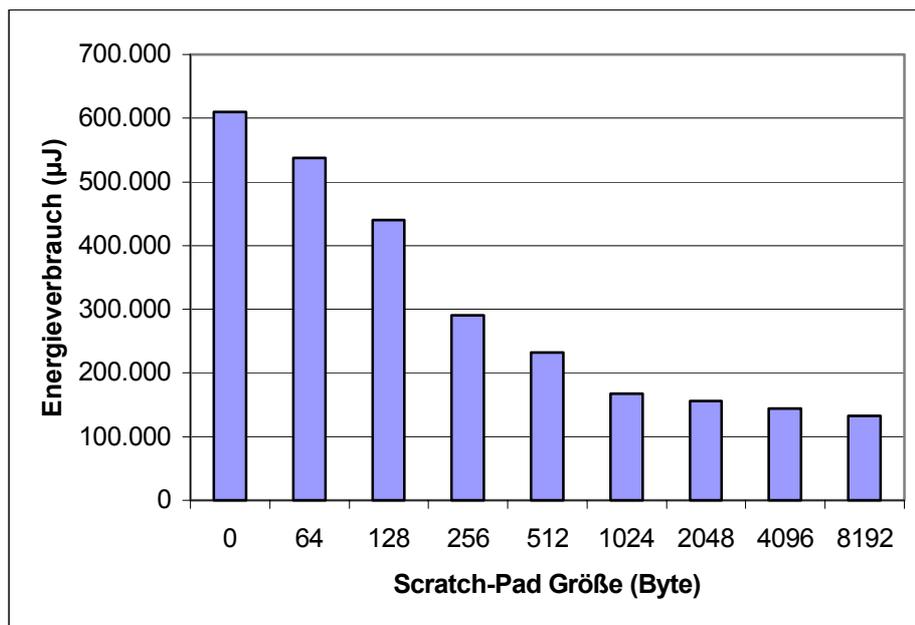


Abbildung 9.4: geschätzter Energieverbrauch bei Verlagerung der Bibliotheksfunktionen auf den Scratch-Pad-Speicher

Schon bei kleineren Scratch-Pad-Größen werden „teure“ Bibliotheksfunktionen auf den Scratch-Pad-Speicher gelegt. Bei einer Scratch-Pad-Größe von 2048 Byte sind fast alle Bibliotheksfunktionen auf den Scratch-Pad verlagert worden. Das weitere Verlegen der restlichen Programmteile bringt nur noch wenig zusätzliche Energieeinsparung.

## 9.2 Cache

Bei der Optimierung des Energieverbrauchs von Programmen durch den Einsatz von Cache als OnChip-Speicher untersuchte Lee in [LE01] folgende fünf Cache-Organisationsformen: Direct-Mapped, 2-fach assoziativ, 4-fach assoziativ, 8-fach assoziativ und voll assoziativer Cache. Nähere Erläuterungen zu den einzelnen Organisationsformen können in [LE01] nachgelesen werden. Lee kam zu dem Ergebnis, dass ein 2-fach assoziativer Cache die schnellste und energiesparsamste der untersuchten Cacheorganisationen für die dort betrachteten Benchmarks ist. Aus diesem Grund wird im Rahmen dieser Diplomarbeit für das Programm *ref\_kernel.c* nur der 2-fach assoziative unified Cache betrachtet.

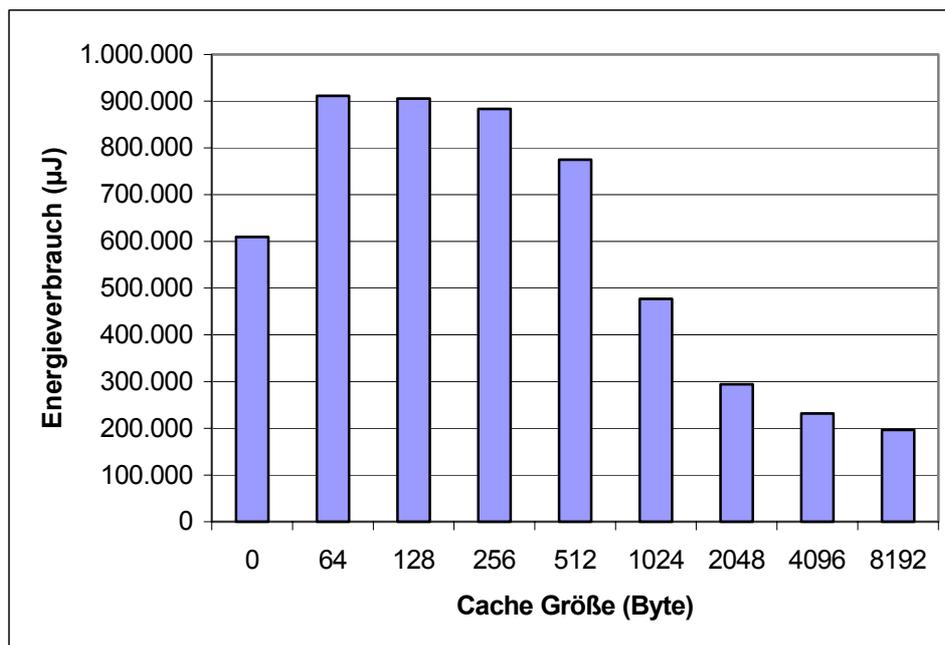


Abbildung 9.5: Energieverbrauch für `ref_kernel.c` bei Nutzung von Cache-Speicher

Untersucht wurden für `ref_kernel.c` die Cache-Größen 0, 64, 128, 256, 512, 1024, 2048, 4096 und 8192 Byte. Die Ergebnisse für den Energieverbrauch des Programms sieht man in *Abbildung 9.5*. Der Wert in der ersten Spalte stellt den Energieverbrauch dar, wenn kein Cache benutzt wird.

Wie man erkennen kann, liegt der Energieverbrauch bei Nutzung der Cache-Größen 64, 128, 256 und 512 Byte sogar über dem Energieverbrauch, wenn kein OnChip-Speicher (0 Byte) benutzt wird. Erst ab einer Cache-Größe von 1024 Byte sinkt der Energieverbrauch für das Programm `ref_kernel.c` unter den Energieverbrauch bei 0 Byte Cache-Größe.

Der gleiche Verlauf ergibt sich auch für die Anzahl der CPU-Zyklen (*Abbildung 9.6*). Die Frage, die sich dann stellt, ist, warum der Energieverbrauch bei Einsatz von kleinen Caches ansteigt? Der Grund dafür ist die Anzahl der Cache Misses. Befindet sich das gewünschte Datum nicht im Cache, tritt ein Cache Miss auf und das Datum muss aus dem Hauptspeicher gelesen werden. Benutzt man keinen Cache, hätte man nur die Kosten für das Holen des Datums aus dem Hauptspeicher.

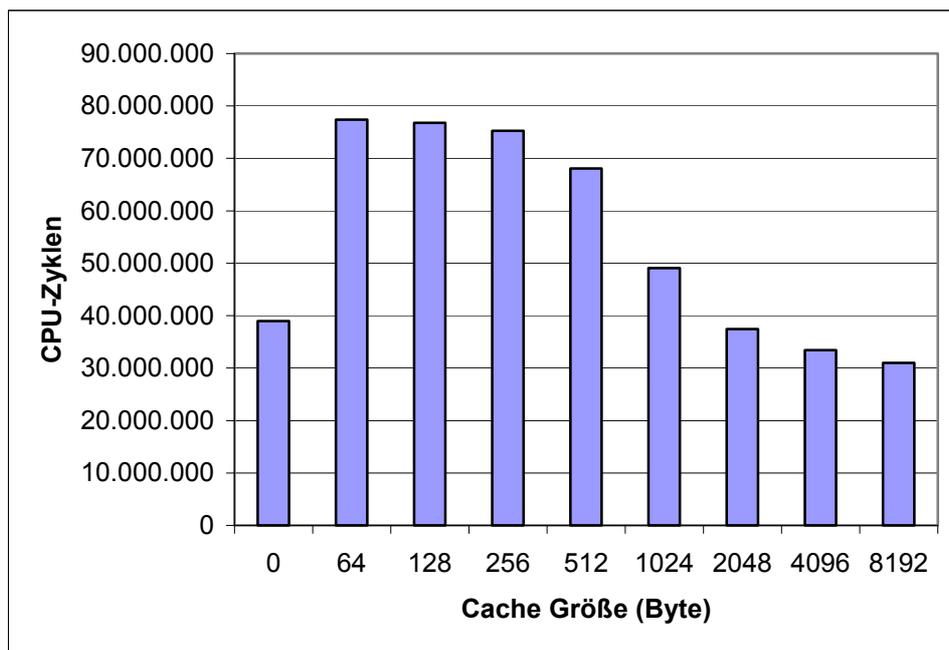


Abbildung 9.6: CPU-Zyklen für `ref_kernel.c` bei Nutzung von Cache-Speicher

Den Verlauf der Cache Hit-Rate in Abhängigkeit der Cache-Größe zeigt *Abbildung 9.7*. Die Anzahl der Cache Misses nimmt mit steigender Cache-Größe kontinuierlich ab. Je größer der Cache, desto mehr Daten können im Cache gehalten werden. Dadurch verringert sich die Wahrscheinlichkeit, dass sich ein gewünschtes Datum nicht im Cache-Speicher befindet und dadurch ein Cache Miss entsteht.

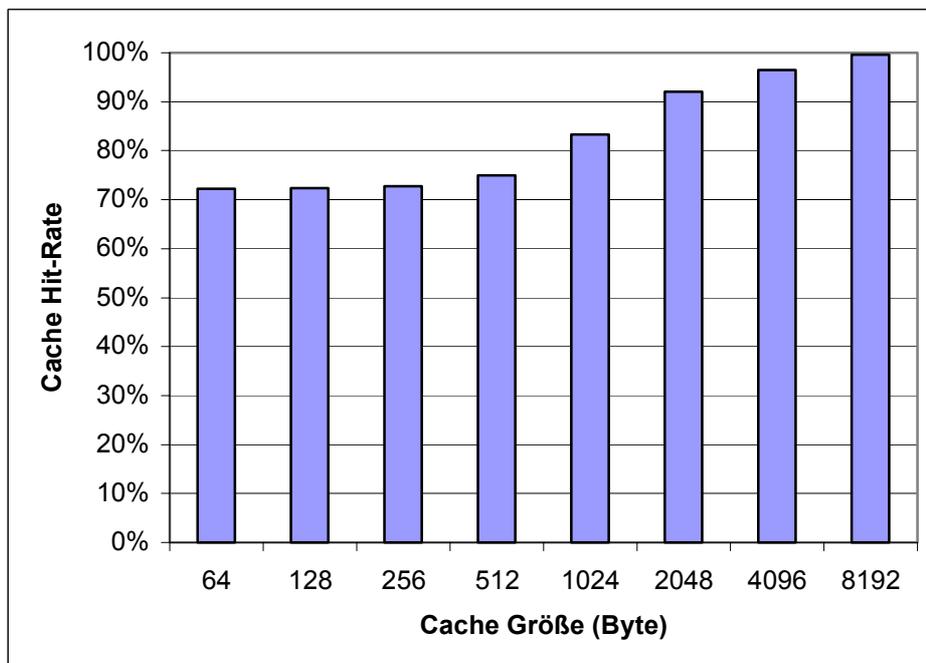


Abbildung 9.7: Cache Hit-Rate für `ref_kernel.c`

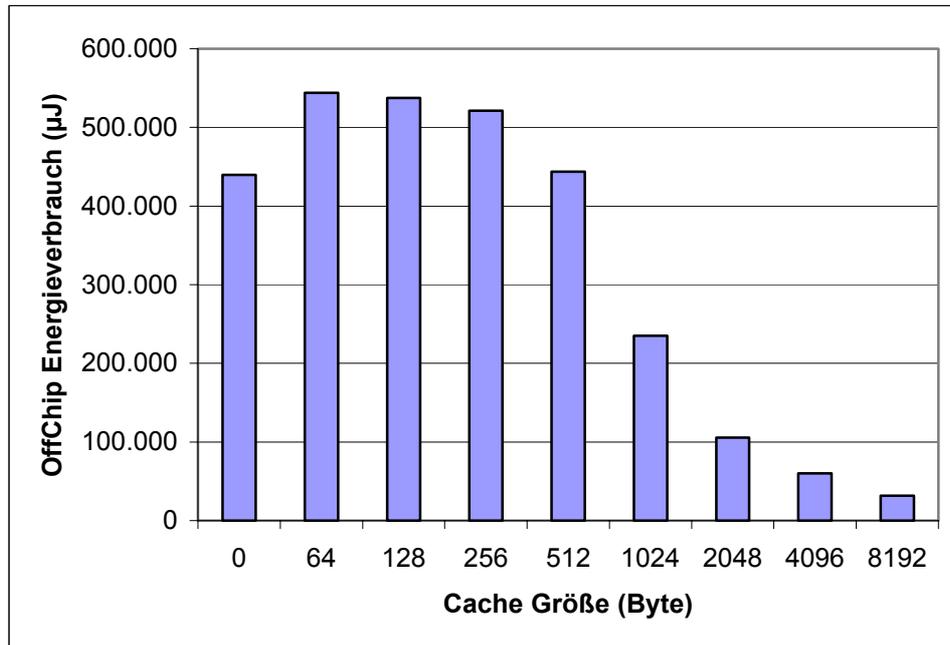


Abbildung 9.8: OffChip Energieverbrauch für `ref_kernel.c` bei Nutzung von Cache-Speicher

Bei sinkender Cache Miss-Anzahl sinkt auch der Energieverbrauch für den OffChip-Speicher (Abbildung 9.8). Tritt während eines Zugriffs kein Cache Miss auf, muss nämlich nicht mehr auf den OffChip-Speicher zugegriffen werden, um das gewünschte Datum zu erhalten.

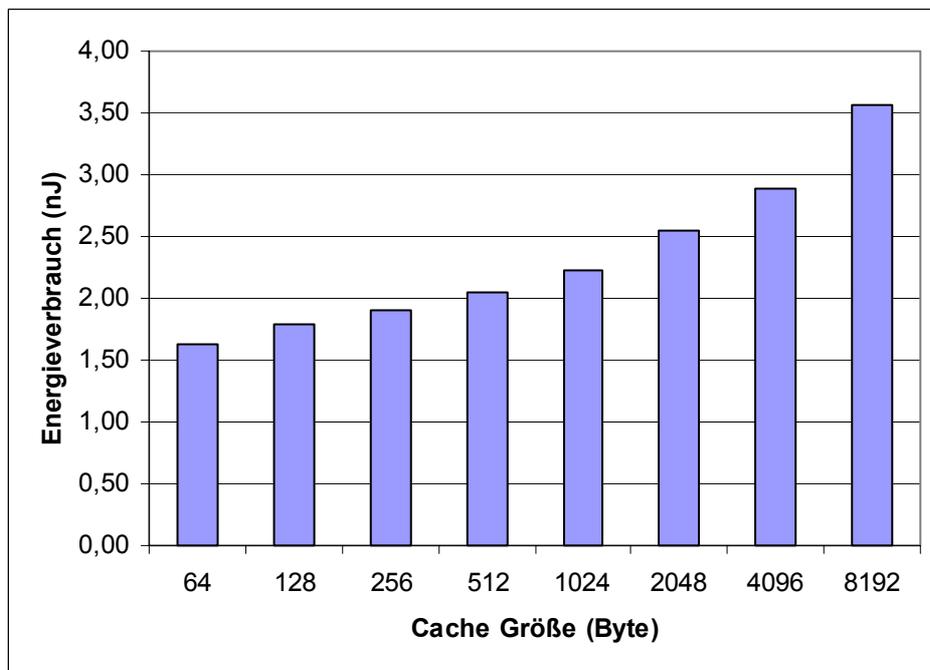


Abbildung 9.8: Energieverbrauch pro Cache Zugriff [LE01]

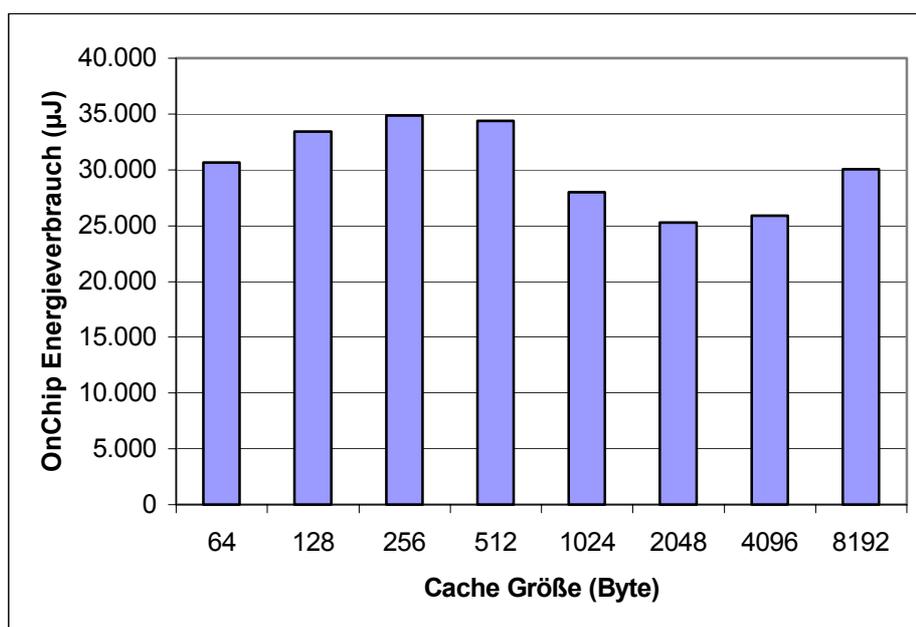


Abbildung 9.10: OnChip Energieverbrauch für `ref_kernel.c` bei Nutzung von Cache-Speicher

Abbildung 9.8 zeigt den von Lee ermittelten Energieverbrauch pro Zugriff für einen 2-fach assoziativen Cache. Der Energieverbrauch für einzelne Zugriffe auf den Cache steigt mit wachsender Cache-Größe kontinuierlich an.

Den Energieverbrauch für den OnChip-Speicher zeigt *Abbildung 9.10*. Mit steigender Cache-Größe steigt auch der Energieverbrauch für den Cache an. Dies gilt aber nur bis zu einer Cache-Größe von 256 Byte. Bei einer Größe von 512 Byte sinkt zum ersten Mal der Energieverbrauch des OnChip-Speichers. Bei dieser Größe sinkt auch die Anzahl der Cache Misses in stärkerem Ausmaße als bei den vorherigen Cache-Größen.

Der Energieverbrauch für den OnChip-Speicher sinkt für die Cache-Größen 512, 1024 und 2048 Byte. Bei den Größen 4096 und 8192 Byte steigt der Energieverbrauch für den OnChip-Speicher wieder an. In diesen Bereichen ist zwar die Anzahl der Cache Misses minimal, aber der OnChip-Speicher benötigt mit steigender Größe mehr Energie pro Cache-Zugriff [LE01].

### 9.3 Vergleich Scratch-Pad und Cache

Wie man gesehen hat, erreicht man bei allen hier betrachteten OnChip-Speichergrößen durch Scratch-Pad-Nutzung bessere Resultate als mit einem Cache-Speicher. Der Cache-Speicher verursacht für kleine Speichergrößen sogar eine starke Verschlechterung des Energieverbrauchs für das hier betrachtete Programm *ref\_kernel.c*. Der Grund dafür liegt in der hohen Anzahl der Cache Misses, die es bei der Scratch-Pad-Nutzung nicht gibt. Dem Programm ist zur Ausführungszeit bekannt, ob ein gewünschtes Datum im OffChip- oder OnChip-Speicher liegt. Die Entscheidung, welche Programmteile in welchem Speicher liegen, wird nämlich zum Zeitpunkt der Codegenerierung getroffen.

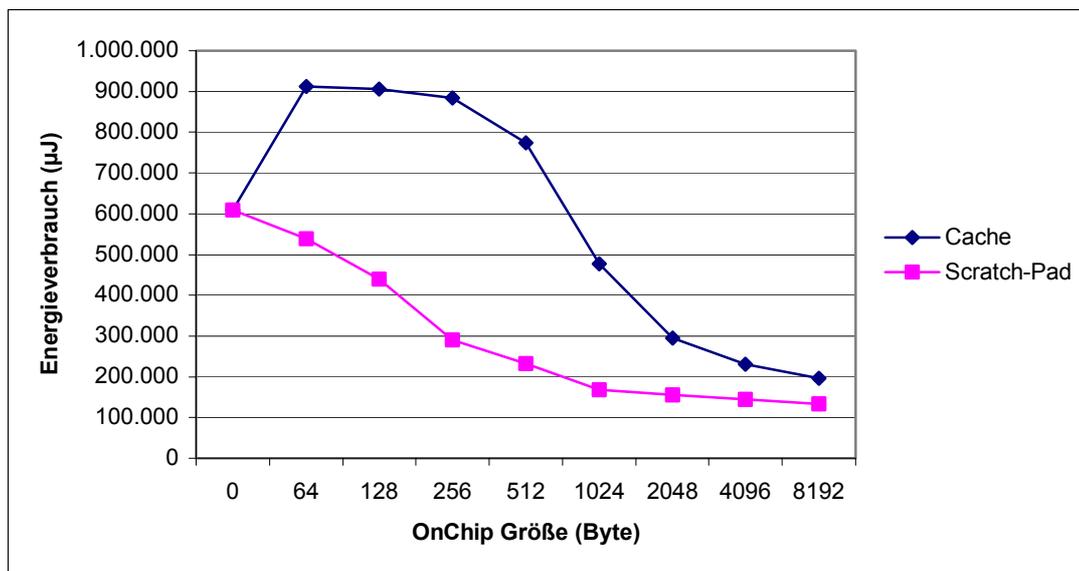


Abbildung 9.11: Energieverbrauch Cache und Scratch-Pad im Vergleich

Aufgrund des niedrigeren Energieverbrauchs bei Scratch-Pad-Nutzung (Abbildung 9.11), sollte auf den Einsatz von Cache-Speicher verzichtet werden. Es sollte auf jeden Fall die Scratch-Pad-Variante als OnChip-Speicher eingesetzt werden. Die hier betrachteten Ergebnisse für die Scratch-Pad-Nutzung beruhen auf einer Abschätzung für den Fall, dass Bibliotheksfunktionen auf den Scratch-Pad verlagert werden können, was zur Zeit noch nicht im Scratch-Pad-Algorithmus implementiert ist. *Abbildung 9.12* zeigt, dass die Kosten für einen einzelnen Zugriff auf den OnChip-Speicher beim Scratch-Pad-Speicher niedriger sind als beim Cache-Speicher.

Für die Nutzung von Scratch-Pad-Speicher ist außerdem kein zusätzlicher Cache-Controller nötig, wie man ihn bei der Nutzung von Cache-Speicher benötigt. Das heisst, dass Scratch-Pad-Speicher bei gleicher Byte-Größe einen weiteren Vorteil gegenüber dem Cache-Speicher besitzt, da der Tag-Speicher und die Komparatoren zusätzliche Fläche auf dem eingebetteten System einnimmt.

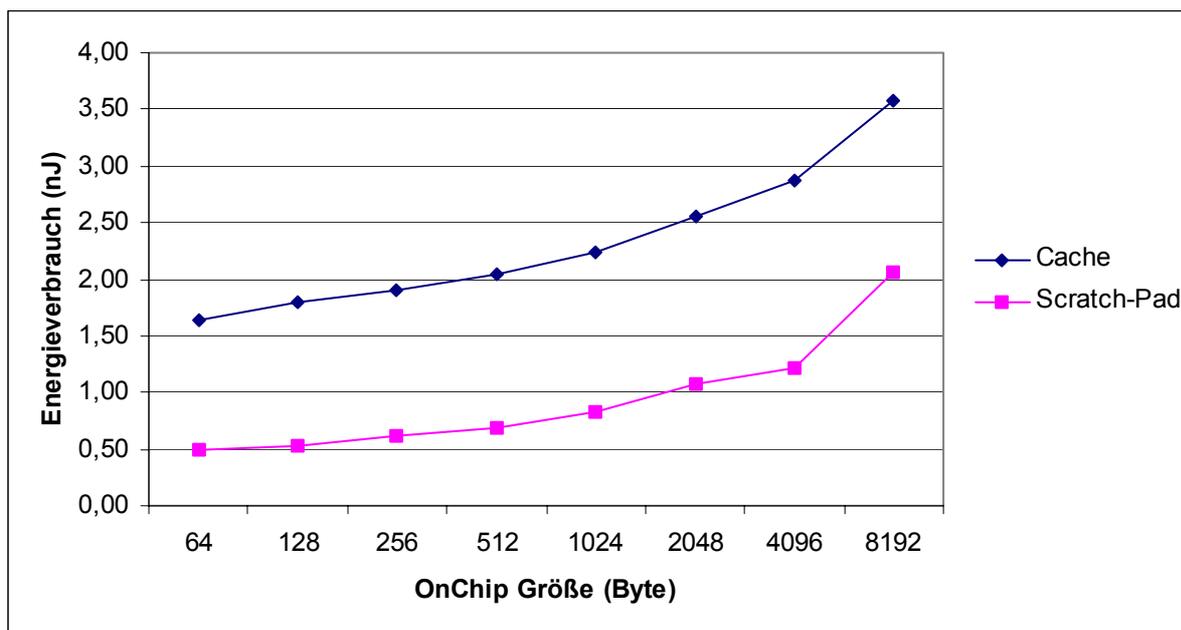


Abbildung 9.12: Energieverbrauch für einzelne OnChip-Speicher-Zugriffe für Cache und Scratch-Pad im Vergleich

Wie gut die Ergebnisse für einen dynamischen Scratch-Pad-Algorithmus sind, kann im Moment noch nicht gesagt werden. Es ist davon auszugehen, dass der Energieverbrauch bei Einsatz eines dynamischen Scratch-Pad Algorithmus weiter reduziert werden kann, da dann die Programmteile im OnChip-Speicher während der Programmausführung ausgetauscht werden können.

Im nächsten Kapitel werden die bisher erzielten Ergebnisse zusammengestellt und durch Kombination der vorgestellten Optimierungen gezeigt, wie stark der Energieverbrauch für das Programm *ref\_kernel.c* insgesamt reduziert werden kann.

## 10 Ergebnisse und Diskussion

In diesem Kapitel wird eine Kombination der vorher untersuchten Optimierungen gesucht, die eine maximale Energieeinsparung erzielt. Dazu werden die Ergebnisse aus den einzelnen Kapiteln betrachtet und die benutzten Optimierungen angewendet, wenn dadurch Energieeinsparungen erzielt werden konnten.

In Kapitel 5 wurde eine Optimierung vorgestellt, die, ohne Genauigkeitsverluste zu verursachen, die Bitbreite des verwendeten Datentyps von 64 Bit (*double*) auf 32 Bit (*float*) reduzierte. Die erzielten Energie- und Performannewerte können Abbildung 10.1 entnommen werden.

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
<i>Original</i>	609.504,289	100,00	38.995.260	100,00
<i>Spezifikations- ebene</i>	207.035,400	33,97	13.464.956	34,53

Abbildung 10.1: Verwendung von 64-Bit und 32-Bit Datentypen im Vergleich

Dass die Einsparungen so hoch sind, liegt daran, dass der Datentyp von „double“ auf „float“ reduziert wurde. Möchte man z.B. zwei „double“-Variablen addieren, benutzt man die Bibliotheksfunktion `_dadd` [ARM98]. Nach der Reduzierung des Datentyps von „double“ auf „float“ nutzt man die äquivalente Bibliotheksfunktion `_fadd` [ARM98]. Die Funktion `_fadd` verbraucht nur ungefähr 50% der Energie, die die Funktion `_dadd` verbraucht. Außerdem werden nicht mehr 4 Register für eine Addition benötigt, sondern nur noch 2 Register. Das senkt den Registerdruck und die Kosten für den Datentransfer zwischen Speicher und Prozessor werden gesenkt.

Diese Optimierung bildet den Startpunkt für die Kombination mit den anderen Optimierungen.

Die nächsten Optimierungen waren die Schleifen- und Arrayoptimierungen aus Kapitel 6. Dabei kam zuerst das *Loop Unrolling* zum Einsatz. Anschließend wurden die *LICM* und die *Loop Reversal*-Transformation aus Kapitel 6.1 angewendet. Als einzige Arrayoptimierung wurde die manuelle Optimierung aus Kapitel 6.2 durchgeführt. Durch diese Optimierungen konnte der Energieverbrauch weiter gesenkt werden (*Abbildung 10.2*).

*Hinweis:* Die prozentualen Angaben für den Energieverbrauch und die CPU-Zyklen sind in den folgenden Abbildungen unterteilt in Änderungen zur Originalversion ohne Optimierung (linke Spalte) und Änderungen zur vorherigen Optimierung (rechte Spalte).

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)		CPU-Zyklen	CPU-Zyklen (%)	
<i>Spezifikations- ebene</i>	207.035,400	33,97	100,00	13.464.956	34,53	100,00
<i>Spez.- und Quellcodeebene</i>	190.929,833	31,33	92,22	12.390.280	31,77	92,02

Abbildung 10.2: Einsatz von Schleifen- und Arrayoptimierungen

Die Optimierungen auf der *MIR*-Ebene aus Kapitel 7 wurden nicht angewendet, da sich für das hier betrachtete Programm *ref\_kernel.c* keine Verbesserungen ergaben. Das liegt daran, dass die im *encc* fest integrierten Optimierungen für *ref\_kernel.c* bessere Resultate erreichen können, wenn die *MIR* nicht optimiert worden ist. Diese Optimierungen waren, da fest integriert, bei fast allen Simulationen aktiviert und sind deshalb auch in den Optimierungsphasen 1 und 2 schon enthalten.

Die Ausnahme bildet die Simulation in Kapitel 8.1, wo die integrierten Optimierungen deaktiviert wurden, um festzustellen, welchen Einfluss sie auf den Energieverbrauch der *ref\_kernel.c* haben. Die dabei erreichte Einsparung bei Energieverbrauch und CPU-Zyklen spricht aber eindeutig für die weitere Anwendung der integrierten Optimierungen.

Die Optimierungen aus Kapitel 8.2 wurden als nächstes durchgeführt. Dabei handelt es sich um die Optimierungen auf der Assemblercode-Ebene.

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)		CPU-Zyklen	CPU-Zyklen (%)	
<i>Spez.- und Quellcodeebene</i>	190.929,833	31,33	100,00	12.390.280	31,77	100,00
<i>Spez.-, Quellcode- und LIR-Ebene</i>	132.151,072	21,68	69,21	8.667.871	22,23	69,96

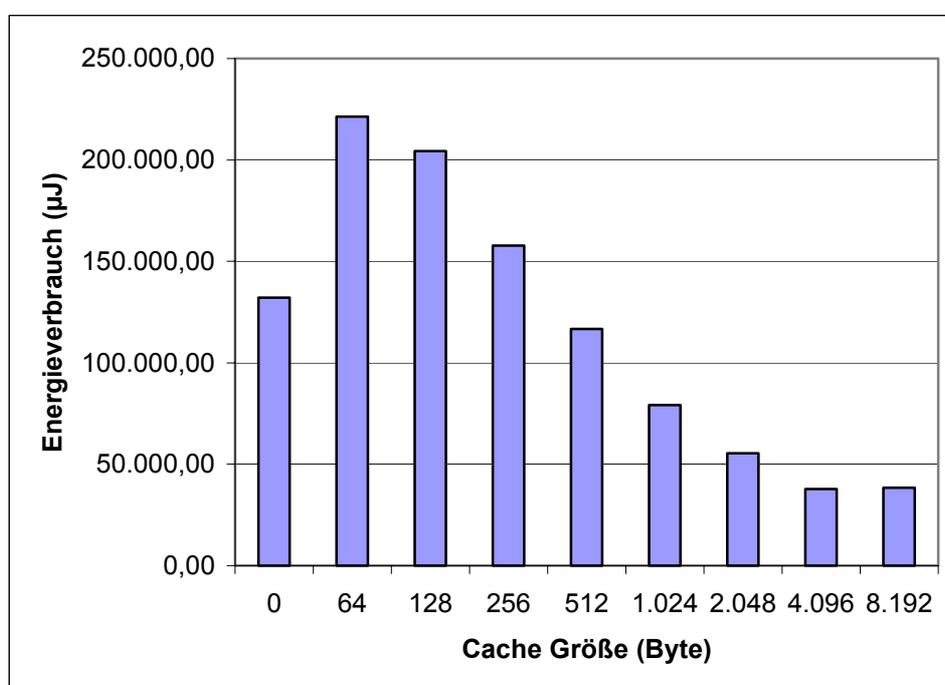
Abbildung 10.3: Optimierungen auf Assemblercode-Ebene

Hier kann man zum ersten Mal erkennen (*Abbildung 10.3*), dass sich die zuletzt angewendeten Optimierungen in der Kombination mit anderen Optimierungen stärker bemerkbar machen, als wenn sie einzeln durchgeführt werden, wie in Kapitel 8.2 gezeigt. Der Energieverbrauch konnte im Vergleich zu den Optimierungen aus Kapitel 5 und 6 (*Abbildung 10.3*) auf 69,21% reduziert werden.

Zusammen mit den zusätzlich angewendeten Schleifen- und Arrayoptimierungen entstand ein Assemblercode, der mehr Optimierungspotenzial bot, als der in Kapitel 8.2 betrachtete Assemblercode.

Bisher konnte der Energieverbrauch des Programms *ref\_kernel.c* auf 21,68% und die Anzahl der benötigten CPU-Zyklen auf 22,23% reduziert werden. Im nächsten Schritt soll die Optimierung durch den Einsatz von Scratch-Pad-Speicher (Kapitel 9) hinzukommen, um noch bessere Resultate zu erzielen. Da die Ergebnisse zur Zeit auf Schätzungen beruhen, wurde für die Betrachtung in diesem Kapitel die Optimierung für den Cache-Speicher gewählt, da hierfür reale Werte existieren.

Dabei zeigte sich, dass ein Cache-Speicher für eine Anwendung auch zu groß sein kann, da ein großer Cache pro Zugriff „teurer“ ist als ein kleinerer Cache (*Abbildung 10.4*).



*Abbildung 10.4: Optimierung durch Cache-Speicher*

Der geringste Energieverbrauch entstand bei einer Cache-Größe von 4.096 Byte. Bei 8.192 Byte verringerte sich zwar die Anzahl der Cache-Misses von 7.849 auf 790 und damit auch die Anzahl der Kosten für den Off-Chip-Speicher, aber der Energieverbrauch für den On-Chip-Speicher stieg aufgrund der Cache-Größe gleichzeitig in so starkem Masse an (*Abbildung 10.5*), dass der Gesamtenergieverbrauch mit 38.348 µJ über dem Energieverbrauch von 37.820 µJ bei einer Cache-Größe von 4.096 Byte lag.

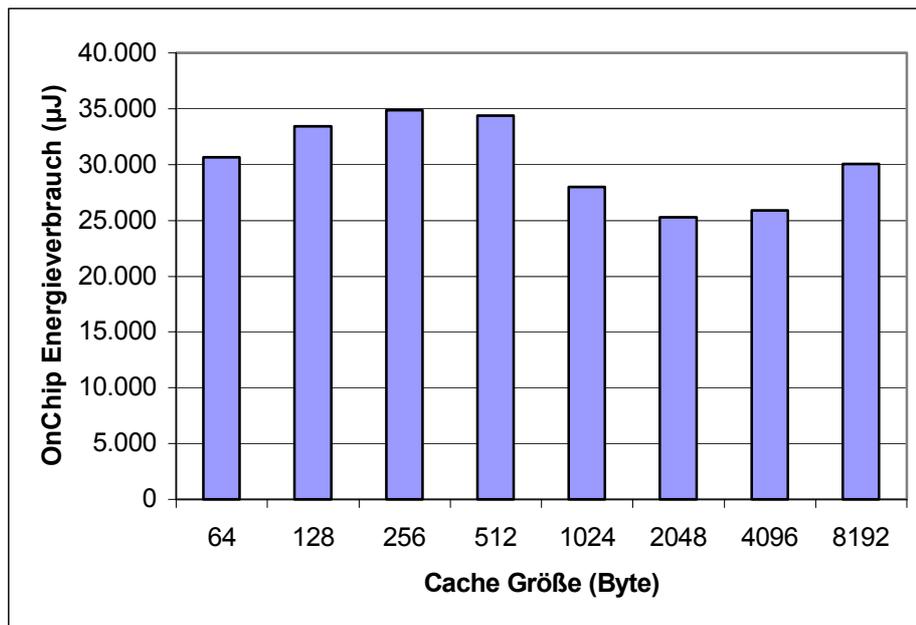


Abbildung 10.5: Cache-Speicher Energieverbrauch

Nach dieser Optimierung (Cache-Speicher mit 4.096 Byte) sieht man, dass der Energieverbrauch im Gegensatz zu den vorherigen Optimierungen viel stärker reduziert werden konnte, als die benötigte Anzahl an CPU-Zyklen (Abbildung 10.6). Der Energieverbrauch wurde auf 28,62 % und die CPU-Zyklen auf 78,05 % gegenüber den vorherigen Optimierungen reduziert. Dieser Unterschied liegt darin begründet, dass durch den geschickten Einsatz eines Cache-Speichers der Datentransfer zwischen Off-Chip-Speicher und CPU deutlich reduziert werden kann, die Anzahl der auszuführenden CPU-Befehle aber gleich bleibt.

<i>ref_kernel.c</i>	Energieverbrauch (µJ)	Energieverbrauch (%)		CPU-Zyklen	CPU-Zyklen (%)	
<i>Spez.-, Quellcode- und LIR-Ebene</i>	132.151,072	21,68	100,00	8.667.871	22,23	100,00
<i>Spez.-, Quellcode- und LIR-Ebene + Cache</i>	37.819,725	6,21	28,62	6.765.023	17,35	78,05

Abbildung 10.6: Optimierungen durch Cache-Speicher

Insgesamt konnte der Energieverbrauch auf 6,21% und die Anzahl der CPU-Zyklen auf 17,35% des originalen Programms *ref\_kernel.c* verringert werden. *Abbildung 10.7* und *Abbildung 10.8* zeigen den Verlauf der CPU-Zyklen und des Energieverbrauchs über die vier Optimierungskombinationen.

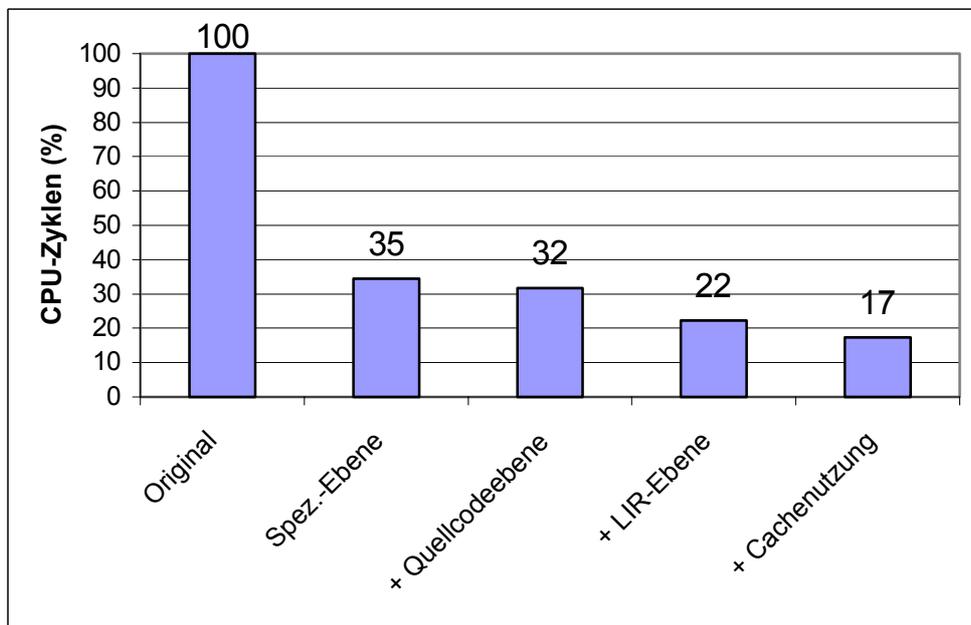


Abbildung 10.7: CPU-Zyklen im Vergleich

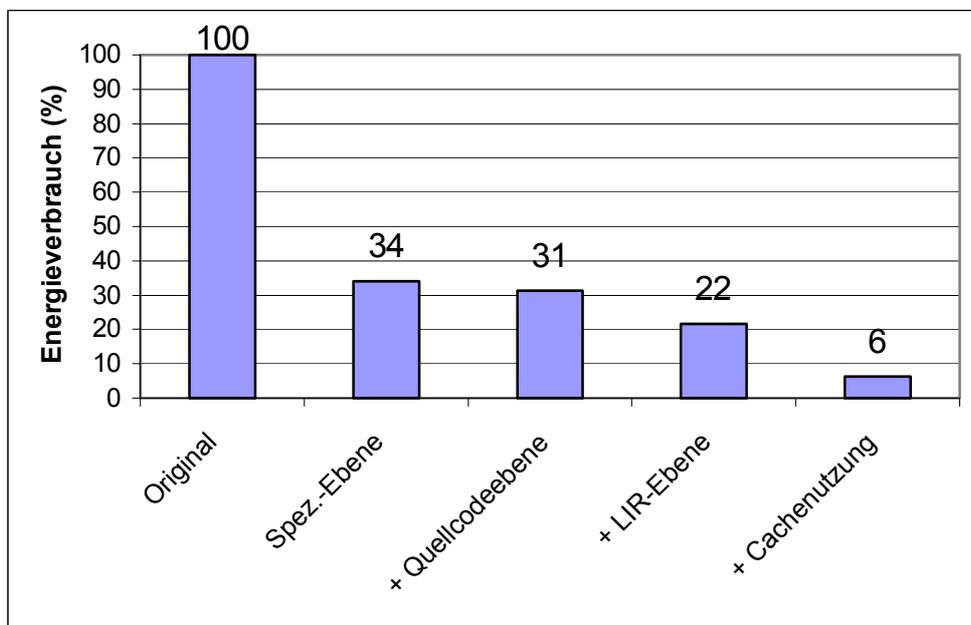


Abbildung 10.8: Energieverbrauch im Vergleich

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
<i>Original</i>	609.504,289	100,00	38.995.260	100,00
<i>Spezifikations- ebene</i>	207.035,400	33,97	13.464.956	34,53
<i>Spez.- und Quellcodeebene</i>	190.929,833	31,33	12.390.280	31,77
<i>Spez., Quellcode- und LIR-Ebene</i>	132.151,072	21,68	8.667.871	22,23
<i>Spez., Quellcode- und LIR-Ebene + Cache</i>	37.819,725	6,21	6.765.023	17,35
<i>Spez., Quellcode- und LIR-Ebene + Scratch-Pad (geschätzt)</i>	29.733,99	4,87	-	-

*Abbildung 10.9: Optimierungskombinationen im Überblick*

Die Ergebnisse der einzelnen Optimierungskombinationen sind in *Abbildung 10.9* tabellarisch aufgeführt. Es wird deutlich, dass durch die Kombination mehrerer aufeinanderfolgender Optimierungen sehr gute Resultate erzielt werden können. Die letzte Zeile enthält den geschätzten Wert für die Nutzung von Scratch-Pad-Speicher anstelle von Cache-Speicher.

Das nächste Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick.

# 11 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurden die unterschiedlichen Ebenen betrachtet, die bei der Entstehung von Software existieren. Angefangen mit der Entwurfs- oder Spezifikationsebene, geht die Betrachtung über die *HIR*- sowie die Quellcode-Ebene zur *MIR*-Ebene. Anschließend folgt die *LIR*- und Assemblercode-Ebene. Zusätzlich zur Betrachtung der Ebenen wurde auch der Einsatz von OnChip-Speicher für die Programmausführung untersucht. Alle Ebenen wurden unter dem Gesichtspunkt der Minimierung des Energieverbrauchs betrachtet.

Man hat gesehen, dass schon beim Entwurf der Software der spätere Energieverbrauch stark beeinflusst werden kann. Das konnte z.B. sehr deutlich an der überlegten Wahl der verwendeten Datentypen für das Programm *ref\_kernel.c* gezeigt werden.

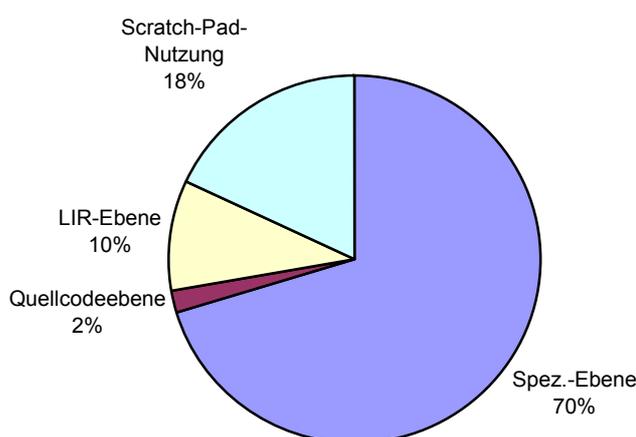
Auf den folgenden *HIR*- und Quellcode-Ebenen und der *MIR*-Ebene hat man die Möglichkeit, automatisiert Optimierungen durchführen zu lassen, die zielarchitekturunabhängig sind. Diese Optimierungen bieten den Vorteil, dass sie auch für andere Prozessoren als den hier betrachteten ARM7-Prozessor unverändert eingesetzt werden können.

Die *LIR*- und Assemblercode-Ebene ermöglicht durch ihre Zielarchitekturnähe ein besonderes Potenzial für Optimierungen. Auf dieser Ebene hat man die Möglichkeit, das Wissen über die spätere Zielarchitektur optimal auszunutzen. Der Aufbau und die Arbeitsweise der Zielhardware sind auf dieser Ebene im Detail bekannt und lassen dadurch hardwarenahe Optimierungen zu.

Im letzten Schritt wurde der Einsatz von OnChip-Speicher untersucht. Dabei wurde Scratch-Pad-Speicher mit Cache-Speicher verglichen. Das Resultat war, dass für das hier betrachtete Programm *ref\_kernel.c* die Wahl des Scratch-Pad-Speichers die höheren Einsparungen ermöglichte. Durch Abschätzung konnte herausgefunden werden, dass der statische Scratch-Pad-Algorithmus bis zu 77,50% Energie einsparen könnte, wenn die Bibliotheksfunktionen auf den Scratch-Pad gelegt werden können.

Im Falle kleiner OnChip-Speichergrößen dürfte aufgrund der hier erzielten Ergebnisse für das Programm *ref\_kernel.c* die Nutzung von Cache-Speicher nie in Frage kommen, da der Energieverbrauch sich bei kleinen OnChip-Speichergrößen sogar erhöht. Ausserdem benötigt ein Scratch-Pad-Speicher weniger Chipfläche als ein Cache-Speicher gleicher Größe. Der Cache-Speicher benötigt zusätzlich zum OnChip-Speicher eine Steuerung, die den Cache während der Programmausführung verwaltet.

Die Verlagerung von Bibliotheksfunktionen auf den Scratch-Pad-Speicher würde zu stärkeren Energieeinsparungen führen als bei der Nutzung von Cache-Speicher. Das hat zumindest die Schätzung gezeigt. Welche Ergebnisse in der Realität erreicht werden können, muss dann untersucht werden, wenn die Möglichkeit der Verlagerung von Bibliotheksfunktionen vorhanden ist.



*Abbildung 11.1: Aufteilung des Einsparungspotenzials*

In dieser Diplomarbeit konnte gezeigt werden, dass Energieoptimierungen für das Programm `ref_kernel.c` auf verschiedenen Ebenen möglich und sinnvoll sind. Anhand von *Abbildung 11.1* wird deutlich, dass das größte Potenzial mit 70% auf der Spezifikationsebene vorhanden ist. Auf die Quellcode-Ebene entfallen 2% und auf die LIR-Ebene 10%. Einen Anteil von 18% am Einsparungspotenzial erreicht die Nutzung von Scratch-Pad-Speicher.

Obwohl die Hardwarehersteller weiterhin anstreben, Hardware zu entwickeln, die weniger Energie bei gleicher oder besserer Leistung verbraucht, bieten die Optimierungen auf der Softwareebene dennoch ein großes Einsparungspotenzial. Daher dürfen Optimierungen auf der Softwareebene nicht vernachlässigt werden. Im Vergleich zur Hardwareentwicklung bringt die Softwareentwicklung in der Regel niedrigere Entwicklungskosten und kürzere Entwicklungszeiten mit sich.

## Anhang A: Thumb-Kernbefehlssatz

Instruktion	Instruktionsart	Zyklen	Aktion
ADC (Op1, Op2)	Add with Carry	1	Op:=Op1 + Op2 + C-bit
ADD (Op1, Op2)	Add	1	Op:=Op1 + Op2
AND (Op1, Op2)	AND	2	Op:=Op1 AND Op2
ASR (Op1, Op2)	Arithmetic Shift Right	1	Op:=Op1 ASR Op2
BIC (Op1, Op2)	Bit Clear	1	Op:=Op1 AND NOT Op2
CMN (Op1, Op2)	Compare Negative	1	CPSR Flags:= Op1 + Op2
CMP (Op1, Op2)	Compare	1	CPSR Flags:= Op1 - Op2
EOR (Op1, Op2)	EOR	1	Op:=Op1 EOR Op2
LSL (Op1, Op2)	Logical Shift Left	2	Op:=Op1 « Op2
LSR (Op1, Op2)	Logical Shift Right	2	Op:=Op1 » Op2
MUL (Op1, Op2)	Multiply	2-5	Op:=Op1 * Op2
NEG (Op1, Op2)	Negate	1	Op:= - Op2
ORR (Op1, Op2)	OR	1	Op:=Op1 OR Op2
ROR (Op1, Op2)	Rotate Right	1	Op:=Op1 ROR Op2
SBC (Op1, Op2)	Subtract with Carry	1	Op:=Op1 - Op2 - Not C-Bit
SUB (Op1, Op2)	Subtract	1	Op:=Op1 - Op2
B (Op1)	Unconditional Branch	3	PC:= Op1
BCC (Op1)	Conditional Branch	3	PC:= Op1
BL (Op1)	Branch and Link	3	LR:= PC - 2; PC:= Op1
BX (Op1)	Branch and Exchange	3	PC:= Op1
LDMIA (Op1, OpN)	Load Multiple	3-10	OpN:= [Op1]; Op1:= Op1 + 4 * N
LDR (Op1, Op2)	Load Word	3	Op1:= [Op2]
LDRB (Op1, Op2)	Load Byte	3	Op1:= [Op2]
LDRH (Op1, Op2)	Load Halfword	3	Op1:= [Op2]
LDRSB (Op1, Op2)	Load signed Byte	3	Op1:= [Op2]
LDRSH (Op1, Op2)	Load signed Halfword	3	Op1:= [Op2]
MOV (Op1, Op2)	Move Register	1	Op1:= Op2
STMIA (Op1, OpN)	Store Multiple	2-9	[OpN]:= Op1; Op1:= Op1 + 4 * N
STR (Op1, Op2)	Store Word	2	[Op2]:= Op1
STRB (Op1, Op2)	Store Byte	2	[Op2]:= Op1
STRH (Op1, Op2)	Store Halfword	2	[Op2]:= Op1
POP (OpN)	Pop Multiple	3-10	OpN:= [SP]; SP:= SP - 4 * N
PUSH (OpN)	Push Multiple	2-9	[SP]:= OpN; SP:= SP - 4 * N
SWI (Op1)	Software Interrupt	2	LR:= PC - 2; PC:= Op1

## Anhang B: Ergebnisse

Kapitel 2:

Funktionsname	Laufzeitanteil %
Reference_IDCT	81,72
form_component_prediction	3,73
Add_Block	3,33
Saturate	2,69
putbyte	2,60
store_yuv1	2,07
Clear_Block	1,31
Flush_Buffer	0,62
Decode_MPEG2_Non_Intra_Block	0,39
Show_Bits	0,31

*Tabelle zu Abbildung 2.10*

Kapitel 5:

	Energieverbrauch ( $\mu$ J)	CPU-Zyklen
ref_kernel.c	609.504	38995260
ref_kernel_float.c	207.035	13464956

*Tabelle zu Abbildung 5.2*

Funktionsname	Laufzeitanteil %
fp_mult_common	20,32
fp_e2d	17,64
fp_addsub_common	13,36
dadd	12,83
Reference_IDCT	12,29
dmul	10,16
dflt_normalise	6,41
fp_normalise_op1	3,20
fp_mult_fast_common	1,06
emul	1,06

*Tabelle zu Abbildung 5.3*

Funktionsname	Laufzeitanteil %
fmul	35,06
Reference_IDCT	25,97
fadd	11,68
fsub	9,09
fflt	6,49
fflt_normalise	3,89
dfix	2,59
fp_mult_common	2,59
main	1,29
16_fadd	1,29

Tabelle zu Abbildung 5.4

Funktionsname	Laufzeitanteil %
Reference_IDCT	81,72
form_component_prediction	3,73
Add_Block	3,33
Saturate	2,69
putbyte	2,60
store_yuv1	2,07
Clear_Block	1,31
Flush_Buffer	0,62
Decode_MPEG2_Non_Intra_Block	0,39
Show_Bits	0,31

Tabelle zu Abbildung 5.6

Funktionsname	Laufzeitanteil %
idctcol	18,57
form_component_prediction	14,32
Add_Block	13,25
Saturate	10,56
putbyte	10,14
idctrow	8,65
store_yuv1	7,93
Clear_Block	5,09
Fast_IDCT	2,42
Flush_Buffer	2,28

Tabelle zu Abbildung 5.7

Funktionsname	Laufzeitanteil %
Fast_IDCT	29,64
form_component_prediction	14,32

Add_Block	13,25
Saturate	10,56
putbyte	10,14
store_yuv1	7,93
Clear_Block	5,09
Flush_Buffer	2,28
Decode_MPEG2_Non_Intra_Block	1,45
Show_Bits	1,19

Tabelle zu Abbildung 5.8

	Energieverbrauch ( $\mu\text{J}$ )	CPU-Zyklen
Reference_IDCT	1.171.281	75306702
Fast_IDCT	56.930	3840997

Tabelle zu Abbildung 5.9

## Kapitel 6:

<i>Reference_IDCT</i>	Energieverbrauch ( $\mu\text{J}$ )	Energieverbrauch (%)
Original	64.360,42430	100,00
nach <i>LICM</i>	63.774,56400	99,09

<i>Reference_IDCT</i>	Energieverbrauch ( $\mu\text{J}$ )	Energieverbrauch (%)
Original	64.360,42430	100,00
nach <i>Loop Reversal</i>	63.243,58110	98,26

<i>Reference_IDCT</i>	Energieverbrauch ( $\mu\text{J}$ )	Energieverbrauch (%)
Original	64.360,42430	100,00
nach <i>Loop Unrolling</i>	53.418,29250	83,00

Tabelle zu Abbildung 6.11

	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)
Gesamt	609.504	100,00
Speicher	439.583	72,12
Prozessor	169.921	27,88

Tabelle zu Abbildung 6.13

Kapitel 7:

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
ohne <i>iropt</i>	609.504,289	100,00	38.995.260	100,00
mit <i>iropt</i>	610.445,773	100,15	39.050.309	100,14

Kapitel 8:

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
ohne ASM-Optimierungen	609.504,289	100,00	38.995.260	100,00
mit ASM-Optimierungen	608.932,767	99,91	38.958.277	99,91

<i>ref_kernel.c</i>	Energieverbrauch ( $\mu$ J)	Energieverbrauch (%)	CPU-Zyklen	CPU-Zyklen (%)
ohne <i>encc_opt</i>	666.093,582	100,00	42.640.855	100,00
mit <i>encc_opt</i>	609.504,289	91,50	38.995.260	91,45

Tabelle zu Abbildung 8.2

Kapitel 9:

Scratchpad-Größe (Byte)	Energieverbrauch ( $\mu$ J)	CPU-Zyklen	Energieverbrauch (nJ) pro Zugriff
64	597.821	38.661.948	0,49
128	586.182	38.312.764	0,53
256	581.003	38.158.012	0,61
512	580.665	38.153.725	0,69
1.024	576.776	37.962.877	0,82
2.048	576.630	37.917.113	1,07
4.096	576.982	37.957.692	1,21
8.192	577.306	37.899.257	2,07

Tabelle zu den Abbildungen 9.1, 9.2 und 9.3

---

Cache-Größe (Byte)	Energieverbrauch ( $\mu$ J)	CPU-Zyklen	Energieverbrauch (nJ) pro Zugriff	Cache Hit- Rate (%)	Cache Energie- verbrauch ( $\mu$ J)
64	911.630	77.398.769	1,63	72,24	30.642
128	905.447	76.794.512	1,79	72,39	33.420
256	883.794	75.247.518	1,90	72,79	34.843
512	774.549	68.053.982	2,05	74,92	34.369
1024	477.154	49.083.728	2,23	83,28	27.964
2048	294.365	37.470.231	2,55	92,11	25.282
4096	231.350	33.440.883	2,88	96,47	25.876
8192	196.358	30.965.772	3,57	99,63	30.030

*Tabelle zu den Abbildungen 9.5, 9.6, 9.7, 9.8, 9.9 und 9.10*

## Anhang C: Iropt-Skript

```
#!/bin/sh

# Bourne shell skript to apply all IR
# optimization tools to a file,
# (multiple times if necessary)

# exit codes:
# 0: OK
# 2: error

# -----
# startup message and usage
# information

PRGNAME=$0

echo
echo "
*****
*****"
echo " *** LANCE V2.0 - IR
optimization script ***"
echo "
*****
*****"

if [ $# != 1 ]; then
    echo " use:" $PRGNAME "<IR-C file>"
    exit 0
fi

RUN_CONSTPROP=true
RUN_CONSTFOLD=true
RUN_COPYPROP=true
RUN_CSE=true
RUN_DCE=true
RUN_JMPOPT=true;
RUN_IVE=true;
RUN_LICM=true;

CHANGE=true

IRNAME=`echo $1 | sed -e 's/.*\///'`

if [ ! $1 = $IRNAME ]; then
    echo " * Copying IR file '$1' ->
'$IRNAME' "
    cp $1 $IRNAME
fi

echo "iropt log file" > iropt.log
while [ $CHANGE = true ]; do
    CHANGE=false

    # constant propagation
    if [ $RUN_CONSTPROP = true ]; then
        RUN_CONSTPROP=false
        echo " * Running constant
propagation"
        constprop $IRNAME >> iropt.log
        case $? in
            1 ) CHANGE=true
                RUN_CONSTFOLD=true
                RUN_DCE=true
                RUN_CSE=true ;;
            0 ) ;;
            * ) exit 2 ;;
        esac
    fi

    # constant folding
    if [ $RUN_CONSTFOLD = true ]; then
        RUN_CONSTFOLD=false
        echo " * Running constant
folding"
        cfold $IRNAME >> iropt.log
        case $? in
            1 ) CHANGE=true
                RUN_CONSTPROP=true
                RUN_CSE=true ;;
            0 ) ;;
            * ) exit 2 ;;
        esac
    fi

    # copy propagation
    if [ $RUN_COPYPROP = true ]; then
        RUN_COPYPROP=false
        echo " * Running copy
propagation"
        copyprop $IRNAME >> iropt.log
        case $? in
            1 ) CHANGE=true
                RUN_DCE=true
                RUN_CSE=true ;;
            0 ) ;;
            * ) exit 2 ;;
        esac
    fi
fi
```

```

# common subexpression elimination
if [ $RUN_CSE = true ]; then
  RUN_CSE=false
  echo " * Running common
subexpression elimination"
  cse $IRNAME >> iropt.log
  case $? in
    1 ) CHANGE=true
      RUN_CONSTPROP=true
      RUN_DCE=true
      RUN_COPYPROP=true;;
    0 ) ;;
    * ) exit 2 ;;
  esac
fi
# dead code elimination
if [ $RUN_DCE = true ]; then
  RUN_DCE=false
  echo " * Running dead code
elimination"
  dce $IRNAME >> iropt.log
  case $? in
    1 ) CHANGE=true
      RUN_CONSTPROP=true
      RUN_CSE=true
      RUN_COPYPROP=true;;
    0 ) ;;
    * ) exit 2 ;;
  esac
fi
# jump optimization
echo " * Running jump
optimization"
jmpopt $IRNAME >> iropt.log
case $? in
  1 ) CHANGE=true
    RUN_DCE=true;;
  0 ) ;;
  * ) exit 2 ;;
esac

# # loop invariant code motion
# if [ $RUN_LICM = true ]; then
#   RUN_LICM=false
#   echo " * Running loop invariant
code motion"
#   licm $IRNAME >> iropt.log
#   case $? in
#     1 ) CHANGE=true;;
#     0 ) ;;
#     * ) exit 2 ;;
#   esac
# fi
# # induction variable elimination
# if [ $RUN_IVE = true ]; then
#   RUN_IVE=false
#   echo " * Running induction
variable elimination"
#   ive $IRNAME >> iropt.log
#   case $? in
#     1 ) CHANGE=true
#       RUN_CONSTPROP=true
#       RUN_COPYPROP=true
#       RUN_DCE=true;;
#     0 ) ;;
#     * ) exit 2 ;;
#   esac
# fi
done
# -----
# print message and return exit code

if [ ! $1 = $IRNAME ]; then
  echo " * Moving IR file
'$IRNAME' -> '$1' "
  mv $IRNAME $1
fi

echo " *** Optimization script done
***"

exit 0

```

*Optimierungsskript iropt*

## Literaturverzeichnis

- [ARM7T98]      *ARM710T Datasheet*, Advanced RISC Machines Ltd., 1998.
- [ARM95]      *ARM7TDMI Data Sheet*, Advanced RISC Machines Ltd., 1995.
- [ARM98]      *ARM Software Development Toolkit Version 2.50 Reference Guide*, Advanced RISC Machines Ltd., 1998.
- [ARM99]      *Application Note 32 ARMulator*, Advanced RISC Machines Ltd., URL: <http://www.arm.com>, 1999.
- [ARM02]      *armcc / tcc: Compiler optimizations*, Advanced RISC Machines Ltd., URL: [http://www.arm.com/support.nsf/html/sdt\\_build!OpenDocument#\\_Section23](http://www.arm.com/support.nsf/html/sdt_build!OpenDocument#_Section23), 2002.
- [ATM98]      *AT91EB01 Evaluation Board*, User Manual, Atmel Corporation, 1998.
- [ATM99a]      *AT91 ARM Thumb Microcontroller*, AT91M40400, Atmel Corporation, 1999.
- [ATM99b]      *Embedded RISC Microcontroller Core*, ARM7TDMI, Atmel Corporation, 1999.
- [BA94]      D. F. Bacon, S. L. Graham, O. J. Sharp: *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, pp.345-420, Vol.26, No.4, Dez. 1994
- [BK95]      V. Bhaskaran, K. Konstantinides: *Image and Video Compression Standards, Algorithms and Architectures*, Kluwer Academic Publishers, 1995.
- [CAT94]      F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, H. De Man: *Global communication and memory optimizing transformations for low power systems*, IEEE Intl. Workshop on Low Power Design, Napa CA, pp.203-208, April 1994

- [CC98] Cogent Computer Systems, *CMA222 ARM710T User's Manual*, URL: [http://www.cogcomp.com/pdf\\_files/user222.pdf](http://www.cogcomp.com/pdf_files/user222.pdf), 1998.
- [CO01] J. Cockx: *Whole Program Compilation for Embedded Software: the ADSL Experiment*, SCOPES 2001, Schloss Rheinfels, St. Goar, Germany, March 2001.
- [CT02] c't Magazin für Computertechnik, Heft 2/2002 Seite 198-202, - Verlag Heinz Heise.
- [DATE02] S. Steinke, L. Wehmeyer, B.S. Lee, P. Marwedel: *Assigning Program and Data Objects to Scratchpad for Energy Reduction*, DATE 2002, Paris/France, March 2002
- [DVB02] *DVB - Digital Video Broadcasting Project*, URL: <http://www.dvb.org/>, 2002.
- [EN02] *encc energy aware c compiler*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2001, URL: <http://ls12-www.cs.uni-dortmund.de/research/encc/Welcome.html>, 2002.
- [ES98] W. Effelsberg, R. Steinmetz: *Video Compression Techniques*, dpunkt Verlag, 1998.
- [GE01] M. Fiesel, M. Lorenz, L. Wehmeyer: *GeLIR Generic Low-Level Intermediate Representation – Introduction and Interface*, URL: <http://ls12-www.cs.uni-dortmund.de/research/gelir/>, 2002.
- [GR02] N. Grunwald: *Energieminimierung eingebetteter Programme durch die dynamische Nutzung eines Scratchpad-Speichers*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund (Entwurf), 2002.
- [GU92] R. H. Güting: *Datenstrukturen und Algorithmen*, Teubner Verlag Stuttgart, 1992.
- [HO01] L. Hornbach: *Generische Low-Level Optimierungen für RISC-Architekturen*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2001.
- [IEEE95] *ANSI/IEEE Standard 754*, Standard for Binary Floating Point Arithmetic, 1985.

- [LA01] LANCE V2.0, *LANCE - Retargetable C compiler*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2001, URL: <http://ls12-www.cs.uni-dortmund.de/research/LANCE>, 2001.
- [LE01] B.S. Lee: *Vergleich des Energieverbrauchs von Cache- und Scratch-Pad-Speichern für den ARM7-Prozessor*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2001.
- [MA99] P. Marwedel: *Prozessrechnertechnik*, Vorlesungsskript am Informatik-Lehrstuhl 12 der Universität Dortmund, 1999.
- [MC02] MPEG-2 Video Codec (with Source Code). URL: [ftp://ftp.mpegiv.com/pub/mpeg/mssg/mpeg2vidcodec\\_v12.tar.gz](ftp://ftp.mpegiv.com/pub/mpeg/mssg/mpeg2vidcodec_v12.tar.gz), 2002.
- [MI98] Mitel Corporation: *THUMB Introduction to Thumb*, URL: <http://www.mitelsemi.com>, März 1998.
- [MIR98] M. Miranda, F. Catthoor, M. Janssen, H. De Man: *High-Level Address Optimization and Synthesis Techniques for Data-Transfer Intensive Applications*, IEEE Trans. On VLSI Systems, no. 4, vol. 6, Dec. 1998.
- [MPEG01] The Official MPEG Committee Website, URL: <http://mpeg.telecomitalialab.com/>, 2001.
- [MPEG02] Overview of the MPEG-4 Standard, URL: <http://mpeg.telecomitalialab.com/standards/mpeg-4/mpeg-4.htm>, 2002.
- [MU97] S. S. Muchnick: *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.
- [NI02] *NIST FINGERPRINT IMAGE SOFTWARE (NFIS)*, URL: [http://www.itl.nist.gov/iad/894.03/databases/defs/nist\\_nfis.html](http://www.itl.nist.gov/iad/894.03/databases/defs/nist_nfis.html), 2002.
- [OIY99] T. Okuma, T. Ishihara, H. Yasuura, *Real-Time Task Scheduling for a Variable Voltage Processor*, ISSS, 1999.
- [PJ02] D. Pomerleau, T. Jochem: *ALVINN - Autonomous Land Vehicle In a Neural Network*, URL: <http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/alv/member/www/projects/ALVINN.html>, 2002.

- [SC00] R. Schwarz: *Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2000.
- [SKWM01] S. Steinke, M. Knauer, L. Wehmeyer, Prof. Dr. P. Marwedel: *An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations*, Int. Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS), 2001.
- [SO01] G. Sinevriotis, Th. Stouraitis: *SOFLOPO: Low Power Software Development for Embedded Applications - ESD Project 25403*, Public Final Report, University of Patras, Greece, URL: <http://www.vlsi.ee.upatras.gr/soflopo/>, January 2001.
- [SP02] SPHINX Group, URL: <http://www.speech.cs.cmu.edu/speech/sphinx.html>, 2002.
- [ST02] S. Steinke: *Energieoptimierung von Compilern*, Dissertation am Informatik-Lehrstuhl 12 der Universität Dortmund (Entwurf), 2002.
- [TEC02] tecChannel: *Videokompression mit MPEG*, URL: <http://www.tecchannel.de/multimedia/635/index.html>, 2002
- [TH00] M. Theokharidis: *Energiemessung von ARM7TDMI Prozessor-Instruktionen*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2000.
- [TI96] V. Tiwari. Dissertation: *Logic and System for Low Power Consumption*, Princeton University, November 1996.
- [TMW94] V. Tiwari, S. Malik, A. Wolfe: *Power Analysis of Embedded Software: A First Step towards Software Power Minimization*, IEEE Transactions on VLSI Systems, Dezember 1994.
- [TNR00] J. Y. F. Tong, D. Nagle, R. A. Rutenbar: *Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic*, IEEE Trans. on VLSI-Systems, 2000.
- [UML02] *UML - Unified Modelling Language*, URL: <http://www.uml.org>, 2002.
- [ZO01] C. Zobiegala: *Energieeinsparung durch compilergestützte Nutzung des On-Chip-Speichers*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2001.