Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von
Stefan Steinke

Dortmund
2002

Tag der mündlichen Prüfung:

Dekan / Dekanin:

Gutachter:

Vorwort

Die vorliegende Arbeit ist während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik XII der Universität Dortmund unter der Betreuung von Prof. Dr. Peter Marwedel entstanden. Ich möchte mich an dieser Stelle bei allen Personen bedanken, die zur Entstehung und Vollendung dieser Arbeit beigetragen haben.

Ich danke Herrn Prof. Dr. Peter Marwedel für die Möglichkeit an seinem Lehrstuhl zu forschen und für seine Ratschläge und die Unterstützung während der vergangenen Jahre, in der die hier dokumentierten Untersuchungen durchgeführt wurden. Weiterhin möchte ich Herrn Prof. Dr. Peter Padawitz für die Bereitschaft zur Erstellung des Zweitgutachtens danken.

Eine große Freude waren stets die konstruktiven Gespräche und fachlichen Diskussionen mit den Arbeitskollegen Steven Bashford, Markus Lorenz und Lars Wehmeyer. Die Möglichkeit, eigene Ideen vorzustellen, fachliche Meinungen bei Problemen einzuholen und Einreichungen für Konferenzen Korrekturlesen zu lassen, waren sehr wichtige Hilfen für mein eigenes Forschungsvorhaben.

Neben den Kollegen am Lehrstuhl haben auch die Arbeiten der von mir betreuten Diplomanden Nils Grunwald, Thomas Hüls, Markus Knauer, Bo-Sik Lee, Rüdiger Schwarz, Michael Theokharidis und Christoph Zobiegala durch die Bearbeitung einzelner konkreter wissenschaftlicher Fragestellungen diese umfassende Arbeit mit ermöglicht. Die Zusammenarbeit mit den Diplomanden und die gemeinsame Arbeit an den Problemen der Forschung haben mir stets Freude bereitet.

Die für die Forschung investierte Zeit, die über die geregelte Arbeitszeit häufig weit hinausging, fehlte leider für das Zusammensein mit meiner Familie. Ich danke daher insbesondere meiner toleranten Frau und meinen Kindern, dass sie dieses Projekt mit ermöglichten.

Für meine Frau Elisabeth und für meine Kinder Eva, Ute, Judith und Georg

Inhaltsverzeichnis

Kurz-Zusammenfassung

In der Arbeitswelt und in der Freizeit hat die Nutzung von mobilen elektronischen Geräten wie Handys oder PDAs in den letzten Jahren stark zugenommen. Die Funktionen dieser Geräte nehmen sowohl in der Anzahl als auch in der Komplexität weiter zu, wodurch die Kapazitätsgrenze der Akkus häufiger erreicht wird. Dies schränkt die Anwender ein und führt zu der Motivation, den Energieverbrauch zu reduzieren. Außerdem sind andere neue mobile Applikationen zukünftig nur realisierbar, nachdem der Energieverbrauch vorab weiter reduziert wurde.

Neben der bekannten Optimierung der Hardware der Geräte auf Energieverbrauch liefert der steigende Anteil der Software ein neues Potenzial zur Energieeinsparung. Das Ziel dieser Arbeit ist die systematische Untersuchung dieses Energieeinsparungspotenzials bei der Ausführung der Applikationssoftware, welches durch modifizierte oder neue Compilertechniken erreicht werden kann.

Zu Beginn der Arbeit werden die Grundlagen des Energieverbrauchs untersucht und daraus Ansatzpunkte für die Energiereduzierung durch Software entwickelt. Innerhalb des betrachteten Entwurfsablaufs eingebetteter Systeme liefert die Phase der SW-Synthese die Möglichkeit, Einfluss auf den generierten Maschinencode zu nehmen. Im Compiler liegen ausreichende Informationen zur Abschätzung des späteren Energiebedarfs vor, wenn ein entsprechendes Energiemodell integriert wird. Das in dieser Arbeit neu vorgestellte Energiemodell berücksichtigt die Unterschiede im Energieverbrauch in Abhängigkeit von den ausgeführten Instruktionen, ihren verwendeten Funktionseinheiten, den Zugriffen auf verschiedene Speicher sowie den Bitmustern der über Busse transportierten Daten. Diese Eigenschaften sind eine notwendige Voraussetzung zur umfassenden Untersuchung des Potenzials bei der Codegenerierung.

Die verschiedenen Bestandteile und Phasen eines Compilers werden auf der Basis dieses Energiemodells systematisch betrachtet und auf ihr Einsparungspotenzial und die mögliche Integration des Optimierungsziels des Energieverbrauchs hin untersucht. Die Phasen im Front-End des Compilers bieten wenig Ansatzpunkte, da noch kein Bezug zu den Maschineninstruktionen und dem jeweiligen Energieverbrauch hergestellt werden kann. Den Schwerpunkt bilden somit die Phasen im Back-End mit der Instruktionsauswahl, der Instruktionsanordnung, der Registerallokation und den maschinenabhängigen Optimierungen.

Im Detail werden die Phasen und Optimierungen betrachtet, in denen der Energieverbrauch einen Einfluss auf die Verarbeitung hat und die energiesparenden Optimierungen ausführlich beschrieben, die den größten Effekt aufzeigen. Insbesondere die Zugriffe auf den Speicher weisen einen hohen Anteil am Gesamtenergieverbrauch auf, so dass sich hieraus ein großes Potenzial ergibt.

Daher bilden Optimierungen zur effizienteren Nutzung des Speichers den Schwerpunkt der Untersuchungen. Neben der Anwendung bekannter Optimierungen zur effizienteren Nutzung der Prozessorregister werden neue Optimierungen vorgestellt, die eine effiziente Nutzung kleiner, frei adressierbarer Onchip-Speicher unterstützen. Die bisher eingesetzten Caches beinhalten eine Hardwaresteuerung zum Einlagern von häufig verwendeten Programmteilen und Daten. Dieser Mechanismus kann die Programmausführung nennenswert beschleunigen, verbraucht aber in der zusätzlichen Hardware relativ viel Energie für häufige Adressvergleiche.

Die Einbeziehung der während des Compilerlaufs vorliegenden Informationen bei der Entscheidung für die Programmteile und Daten, die in den Onchip-Speicher verlagert werden, bietet ein hohes Energieeinsparungspotenzial. Das dafür notwendige Verfahren wird sowohl als statische Variante mit einer festen Zuordnung von Programmteilen und Daten zum Hauptspeicher und Onchip-Speicher beschrieben als auch in einer erweiterten Variante mit integriertem Umkopieren der Blöcke während des Programmablaufs.

Als Abschluss der Arbeit wird untersucht, wie alternative Codierungen auf Bussen zur Reduzierung des Energieverbrauchs genutzt werden können.

Insgesamt konnte mit dieser Arbeit das Energieeinsparungspotenzial durch einen Compiler in seinen jeweiligen Phasen aufgezeigt werden, sowie neue Techniken, die die Speicherzugriffe effizienter generieren,

vorgestellt werden. Der Energieverbrauch einer Applikation lässt sich dadurch in den betrachteten Fallbeispielen um ca. 50% gegenüber heute eingesetzten Systemen reduzieren.

Kapitel 1

Einleitung

1.1 Motivation

In den letzten Jahren hat die Anzahl elektronischer Systeme sowohl im privaten als auch im industriellen Bereich stark zugenommen. Durch die technische Weiterentwicklung wurden immer mehr Geräte entwickelt, die durch die höhere Integration und durch die automatisierte Produktion auch für einen Großteil der Bevölkerung bezahlbar wurden. Dadurch sind eine höhere Leistungsfähigkeit, gesteigerte Funktionalität und verbesserte Qualität bei geringeren Kosten möglich geworden. Stellvertretend für die Dynamik der Entwicklung können die Steuerungen in der Automobilindustrie betrachtet werden: Während die Elektrik erst langsam Einzug hielt, wurde die Anzahl der elektronischen Steuerungen in Fahrzeugen der Oberklasse in nur wenigen Jahren auf teilweise über 100 eingebettete Systeme gesteigert. Neben den bekannteren Anwendungen der Motorsteuerung und des Airbags fanden auch viele für Anwender im Wesentlichen unsichtbar vernetzte Geräte für Sitzverstellung, Fensterheber, Außenspiegelsteuerung oder Fahrdynamikregelung Einzug in die Fahrzeuge. In den nächsten Schritten werden mechanische oder hydraulische Verbindungen zwischen Lenkrad und Rädern oder zwischen Bremspedal und Bremsen durch so genannte X-by-wire-Systeme ersetzt. Dadurch können eine höhere Zuverlässigkeit, geringere Größe, geringeres Gewicht, mehr Freiheitsgrade im Design und verringerter Energieverbrauch erzielt werden.

Neben der Automobilindustrie hat insbesondere die Einführung von Handys der GSM-Generation mit einer Stückzahl von 50 Millionen Geräten in Deutschland im Jahr 2001 die Kommunikation umwälzend verändert. Die Anwender haben diese Geräte akzeptiert, mit denen sie überall erreichbar und durch SMS jederzeit mit kurzen Nachrichten kontaktierbar sind. Aber auch die Nachfrage nach elektronischen Notizbüchern, so genannten PDAs, oder tragbaren CD- oder MP3-Playern haben die Industrie enorme Stückzahlen produzieren lassen und zur Entwicklung immer neuer Produkte angetrieben.

Für diese mobilen Systeme sind in der Entwicklung besondere Anforderungen an Größe und Gewicht zu berücksichtigen. Sie müssen möglichst schnell auf den Markt gebracht werden (time-to-market), andererseits aber auch eine hohe Zuverlässigkeit besitzen, da spätere Reparaturen oder Updates der Betriebssoftware teuer und für den Kunden ärgerlich sind. Eine weitere wichtige Anforderung ergibt sich aus der Mobilität und der notwendigen Energieversorgung dieser eingebetteten mobilen Systeme. Ein geringer Energieverbrauch soll dafür sorgen, dass die Batterien möglichst selten aufgeladen werden müssen und dass Gewicht und Größe des Energiespeichers kleiner werden.

Generell hat der Energieverbrauch von Prozessoren und elektronischen Systemen auf folgende Faktoren einen entscheidenden Einfluss:

Gewicht und Größe des Energiespeichers

Eine höhere Energiespeicherkapazität erlaubt längere Nutzungs- und Standby-Zeiten vor einer erneuten Aufladung. Allerdings führt dies auch zu höherem Gewicht (z. B. 33% Anteil am Gesamtgewicht beim Nokia 61xx Handy [Seg01]) und größeren Abmessungen, was insbesondere bei portablen Systemen einen großen Nachteil darstellen kann. Weiterhin sind typischerweise mit einer größeren Kapazität auch höhere Beschaffungskosten verbunden. Die Weiterentwicklung von Akkumulatoren in den letzten 30 Jahren hat bei der weit verbreiteten Nickel-Cadmium Technologie nur eine Verdopplung der Kapazität erreicht [Tiw96]. Dies ist ungleich weniger als die Steigerung des Energieverbrauchs elektronischer Systeme. Selbst der Wechsel auf andere neuere Batterietechnologien kann dieses Wachstum an benötigter Leistung bei weitem nicht ausgleichen. Die hohe Bedeutung der Akku-Kapazität und des Gewichtes ergab auch eine Befragung der Handy-Nutzer, die als die drei wichtigsten Kaufkriterien "Einfache Bedienung" (66%), "Lange Akkubetriebsdauer" (63%) und "Geringes Gewicht" (54%) ermittelte [All01]. Die Senkung des Energieverbrauchs erlaubt daher die Nachteile der notwendigen Erhöhung der Kapazitäten teilweise oder auch vollständig zu kompensieren.

Abmessungen der Systeme

Da der größte Teil der in eingebetteten Systemen verbrauchten Leistung als Wärme abgeführt wird, müssen auch die physikalischen Grenzen betrachtet werden. Bei Vermittlungsstellen von Telekommunikationsnetzen führt beispielsweise die hohe Dichte der Schaltkreise die Systeme an die Grenzen der physikalisch möglichen Wärmeabfuhr. Eine Energieverbrauchsreduktion ermöglicht höhere Packungsdichten der Systeme oder - umgekehrt - reduzierte Maßnahmen für die Kühlung. Aber auch bei mobilen Systemen wie Notebooks limitiert die hohe Energieabgabe und die begrenzte Oberfläche die Baugrößen. Bei einer Umgebungstemperatur von 25^oC und einer vom Benutzer als noch angenehm empfundenen Temperatur der Tastatur von 40^oC verbleibt zur passiven Wärmeabgabe eine Differenz von 15^oC. Hieraus ergibt sich für die typischen Abmessungen von Notebooks eine maximale Leistung[1] von 15 bis 17W zzgl. 4 bis 8W bei Einsatz eines Lüfters, die noch abgeführt werden kann. Diese Grenzen sind bereits erreicht und begrenzen die Entwicklung leistungsfähigerer Notebooks. Für Systeme mit noch kleineren Abmessungen bedeutet dies zwangsläufig auch eine nochmals verringerte Leistungsabgabe [Int98].

Chip-Gehäuse und Kühlungsmaßnahmen

Die heutigen Gehäuseformen integrierter Schaltkreise erlauben sowohl die Verwendung preiswerter Plastikgehäuse als auch von teuren Keramikvarianten. Letztere erlauben höhere Betriebstemperaturen und damit eine erhöhte Wärmeabfuhr zu Lasten höherer Herstellungskosten. Wenn der Energieverbrauch daher für einen Schaltkreis verringert werden kann, können dadurch eventuell Plastik- statt der teuren Keramikgehäuse verwendet werden. Eine geringere Wärmeabgabe kann weiterhin die Aufwendungen für aktive und passive Kühlung der Bausteine reduzieren und damit auch den Geräuschpegel verringern.

Umweltschutz

Die US Environmental Protection Agency hat 1992 das "Energy Star"-Logo eingeführt, welches für PC-Systeme vergeben wird, die für CPU, Monitor und Drucker jeweils weniger als 30W Standby-Leistung benötigen [Tiw96]. Damit soll der Tatsache Rechnung getragen werden, dass PCs bereits einen Anteil von 3% am Gesamtenergieverbrauch des kommerziellen Sektors im Jahr 2000 ausmachten. Die jährliche

[1] Das Display wird für diese Kalkulation nicht eingerechnet, da es aufgrund seiner getrennten Anbringung am Gehäuse nicht zur Erwärmung der Tastatur beiträgt.

Steigerungsrate liegt mit 4,1% im Vergleich zu einer durchschnittlichen Gesamtsteigerung von 1% an der Spitze [EIA]. Hinzu kommen Anteile durch PCs in den Privathaushalten, weitere prozessorgesteuerte Bürogeräte sowie eine Vielzahl eingebetteter Systeme.

Zuverlässigkeit und Lebensdauer

Die Temperatur von integrierten Schaltkreisen hat erheblichen Einfluss auf die Lebensdauer. Die Langzeit-Zuverlässigkeit (MTBF = Mean Time Between Failure) verschlechtert sich bei einer Erhöhung der Chip-Temperatur um jeweils 10^{o}C um 50% [Fre97]. Auch wenn man meinen könnte, dass elektronische Geräte schon nach wenigen Jahren wegen der technischen Weiterentwicklung veraltet sind und durch die neueste Generation ersetzt werden, sind einige Systeme, wie die oben genannten Vermittlungsstellen, als Investition auf Jahrzehnte geplant. Ein früher Ausfall des Schaltkreises bedeutet dann eine verringerte Wirtschaftlichkeit. Bei vielen elektronischen Systemen werden an die Lebensdauer auch exakt definierte Anforderungen gestellt. Wenn die Lebensdauer durch erhöhte Temperatur einiger Bausteine gefährdet ist, muss dies durch Redundanz oder teurere Bauteilvarianten kompensiert werden.

Aus den angeführten Gründen ergibt sich die Motivation, den Energieverbrauch von eingebetteten Systemen zu verringern. Im Folgenden werden daher die Grundlagen für den Energieverbrauch vorgestellt, die Ansatzpunkte für die Optimierungen liefern. Diese Ansätze werden dann in einen Entwurfsablauf integriert, dessen prinzipieller Aufbau präsentiert wird. Das letzte Unterkapitel stellt die Ziele dieser Arbeit im Detail vor und gibt einen Überblick über ihren weiteren Aufbau sowie verwandte Arbeiten der entsprechenden Gebiete.

1.2 Grundlagen des Energieverbrauchs

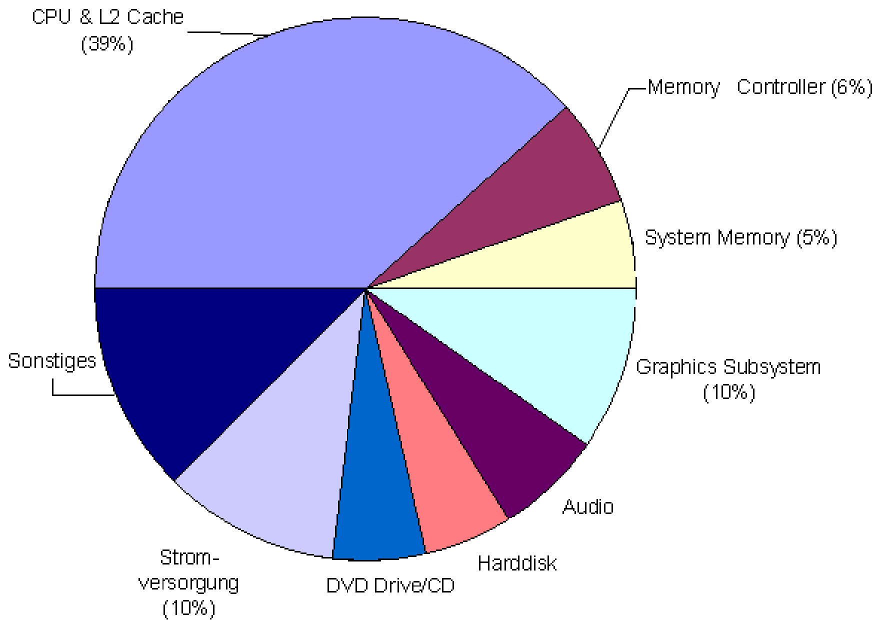


Abbildung 1.1: Energieanteil der Komponenten eines Notebooks bei der Ausführung der Applikation "WinBench"

Am Beispiel eines Notebooks werden in Abbildung 1.1 die Anteile der unterschiedlichen Teilsysteme am Gesamtleistungsverbrauch dargestellt. Basis ist die durchschnittliche Leistung bei der Ausführung eines

Benchmarks, die insgesamt 24,8W zzgl. 4,3W für das Display beträgt. Für die fast ausschließlich aus CMOS[2]-Schaltkreisen gefertigten Teilsysteme CPU, Speicher mit Cache, Memory Controller und Hauptspeicher sowie dem Grafiksubsystem werden 60% der Gesamtleistung verbraucht [Int98].

Tabelle 1.1: International Technology Roadmap for Semiconductors

	'02	'04	'07	'10	'13	'16	
Feature size	130	90	65	45	32	22	nm
Transistoren	348	553	1.106	2.212	4.424	8.848	Mio
F_{max} (μP)	2.317	3.990	6.739	11.511	19.348	28.751	MHz
A_{max}	112	178	357	714	1.427	2.854	mm^2
V_{dd}	1,0	1,0	0,7	0,6	0,5	0,4	V
$P_{v,max}/\mu P$	140	160	190	218	251	288	W
$P_{v,max}/Batt.$	2,6	3,2	3,5	3,0	3,0	3,0	W

In den nächsten Jahren und Jahrzehnten werden sich verschiedene Trends überlagern. Die Anzahl der Transistoren in den Prozessoren wird bis 2016 um den Faktor 25 weiter stark ansteigen (siehe Tab. 1.1), die Chipfläche einschließlich des Onchip-RAMs wird sich ebenfalls um den Faktor 25 vergrößern und die Taktrate um den Faktor 12 erhöhen. Andererseits wird sich die Versorgungsspannung um 60% und die Technologiegröße auf 17% verringern [ITR01].

Trotz aller Anstrengungen in der Forschung und Entwicklung geht man davon aus, dass der Leistungsverbrauch von Prozessoren mit hoher Rechenleistung und Kühlung von den 140W im Jahre 2002 auf 288W im Jahre 2016 weiter ansteigt. Ebenso reichen die Energiesparmaßnahmen bei batteriebetriebenen Prozessoren nicht aus, so dass auch deren Leistungsverbrauch noch langsam weiter ansteigt. Die Notwendigkeit, Energie zu sparen, wird daher zukünftig sogar noch steigen.

Nach dieser Motivation des Energiesparens werden zum besseren Verständnis nachfolgend die Ursachen des Energieverbrauchs betrachtet, um diese durch verschiedene Maßnahmen in weiteren Schritten zu reduzieren.

1.2.1 Leistung und Energie

Zu Beginn müssen die Leistung P und die Energie E definiert werden. Die Leistung P basiert auf der Spannung U und dem Strom I:

$$P(t) = U(t) * I(t)$$

Bei konstanter Spannung U, wie es beispielsweise bei Prozessoren häufig der Fall ist, ist die Leistung P somit proportional zum Strom I.

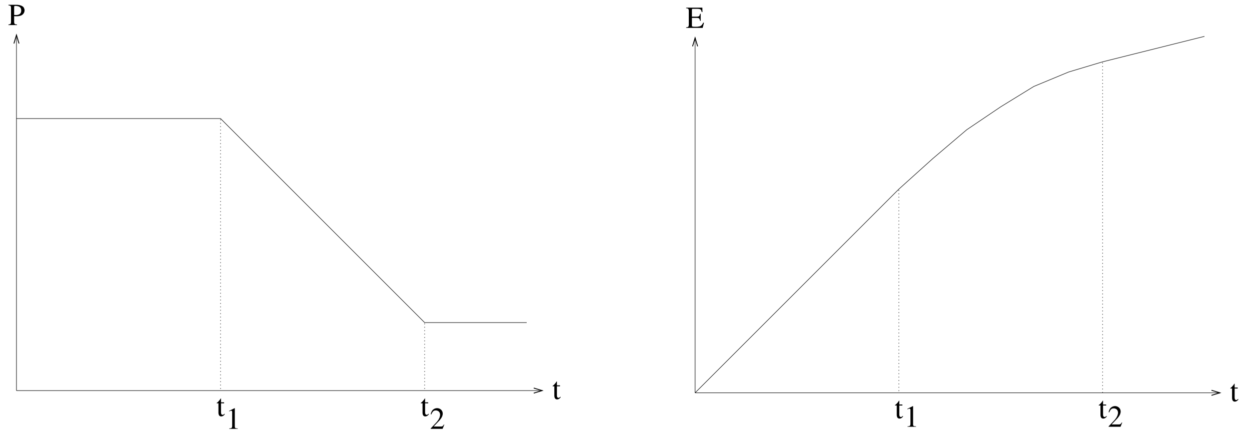
Weiterhin soll noch die maximale Leistung P_{max} bestimmt werden, die beispielsweise für die Dimensionierung von Komponenten wie Netzgeräten entscheidend ist:

$$P_{max} = \max(P(t)) = \max(U(t) * I(t))$$

Die Energie E, die durch ein System in der Zeit T verbraucht wird, berechnet sich auf Basis der Leistung P wie folgt [RJ98]:

$$E = \int_0^T P(t)dt$$

[2]CMOS = Complementary Metal-Oxide Semiconductor



(a) Leistung (b) Energie

Abbildung 1.2: Verhältnis zwischen Leistung und Energie

In Abbildung 1.2 wird das Verhältnis zwischen Leistung und Energie an einem Beispiel dargestellt. Für ein System mit der in (a) dargestellten Leistung entspricht der Verlauf der Energie der in (b) dargestellten Charakteristik. Bei konstanter Leistung ab dem Zeitpunkt *0* bis zum Zeitpunkt t_1 ist die Energie linear über die Zeit ansteigend. Zwischen den Zeitpunkten $t_1$und t_2 nimmt die Leistung linear ab und die Energie steigt langsamer an bis zum Zeitpunkt t_2, ab dem der Leistungsverbrauch wieder konstant und somit auch der Energieverlauf wieder linear wird.

Im Folgenden werden die Durchschnittswerte der Leistung P_{avg} betrachtet, die wie folgt definiert ist:

$$P_{avg} = \frac{1}{T} \int_0^T P(t)dt$$

Daraus ergibt sich als alternative Berechnung der Energie E auf Basis der durchschnittlichen Leistung P_{avg} für die Laufzeit T:

$$E = T * P_{avg}$$

Bei Betrachtung eines konstanten Zeitraums T ergibt sich aus dieser Gleichung, dass kein Unterschied zwischen der Optimierung nach P_{avg} und der Optimierung nach E besteht. Anders verhält es sich beim Vergleich unterschiedlicher Betrachtungszeiträume, z. B. weil die Abarbeitung zweier alternativer Programme P_1 und P_2 eine unterschiedliche Laufzeit T_1 bzw. T_2 benötigt. Bei konstanten Strömen I_1 bzw. I_2 und konstanter Spannung U, wie sie bei den z. Z. eingesetzten Systemen überwiegt, ist die Leistung P_{avg} unabhängig von der Laufzeit:

$$P_{avg,1} = I_1 * U \text{ bzw. } P_{avg,2} = I_2 * U$$

Da der Strom in Abhängigkeit der abgearbeiteten Befehle schwankt, ist bei der Optimierung nach P_{avg} daher das Programm P mit kleinerem Strom I vorteilhafter.

Die Energie ist aber das Produkt der Leistung P mit der Laufzeit T_1 bzw. T_2 und verändert sich proportional mit der Zeit:

$$E_1 = P_{avg,1} * T_1 = I_1 * U * T_1 \text{ bzw. } E_2 = I_2 * U * T_2$$

In diesem Fall ist das Programm mit kleinerem Produkt $I * T$ bezogen auf die Energie günstiger, was zu unterschiedlichen Ergebnissen im Vergleich zur Leistungsoptimierung führen kann. Weitere Unterschiede entstehen durch die Möglichkeit der Reduzierung der Taktfrequenz f und der Spannung U. Die Spannungsreduzierung hat Einfluss auf den Strom I und natürlich die Spannung U, so dass für die Energie E mit einer Konstanten c in erster Näherung gilt [OIY99]:

$$E = c * U^2$$

Es hängt nun von den Optimierungszielen ab, ob auf Leistung P oder Energie E optimiert werden muss. Bei einer Optimierung aufgrund der Batteriekapazität, die einen Energiespeicher darstellt, muss nach Energie E optimiert werden, da dieser Speicher durch die abgegebene Energie E limitiert wird. Kürzere Laufzeiten T, nach denen ein Prozessor in den energiesparenden Standby-Zustand geschaltet werden kann, reduzieren den Energieverbrauch und ermöglichen einen kleineren Energiespeicher. Anders ist es z. B. bei der Auslegung eines Netzteils. Bei der Dimensionierung des Netzteils hat die Laufzeit keine Auswirkungen. Jedoch muss das Netzteil für die maximale Leistung P_{max}, die zu einem beliebigen Zeitpunkt abgerufen wird, dimensioniert werden. Bei der Optimierung von P_{max} ist kein Ausgleich durch geringeren Leistungsbedarf zu anderen Zeiten möglich. Im Folgenden wird die häufigere und wichtigere Energieoptimierung den Schwerpunkt bilden und daher nach der Energie E optimiert.

Es werden nun nachfolgend die Ursachen des Energieverbrauchs in CMOS-Schaltungen, der vorherrschenden Technologie für Prozessoren und Speicher, betrachtet, um Ansatzpunkte für eine Reduzierung des Verbrauchs aufzuzeigen.

In der CMOS-Technologie werden die folgenden Leistungskomponenten unterschieden [WE94]:

- die dynamische Verlustleistung bei Schaltvorgängen durch die Schaltleistung P_{sw} bzw. Schaltenergie E_{sw} und die Kurzschlussleistung P_{sc} bzw. Kurzschlussenergie E_{sc},
- die statische Verlustleistung P_{lk} bzw. Verlustenergie E_{lk} durch Leckströme und andere kontinuierlich auftretende parasitäre Ströme.

Am Beispiel des CMOS-Inverters in Abbildung 1.3 werden diese Komponenten dargestellt und auf die Nutzbarkeit zum Energiesparen untersucht. In der Darstellung modelliert die Lastkapazität C_{out} die mit dem Ausgang verbundenen nachfolgenden Eingangs- und Leitungskapazitäten.

1.2.2 Schaltstrom

Die Schaltenergie E_{sw} ist die Energie, die beim Umschalten eines Gatter-Ausgangs benötigt wird. Wenn das Gatter einen stabilen Zustand hat, fließt praktisch kein Strom zwischen OUT und V_{dd} oder OUT und *Gnd*. Wenn das Gatter schaltet, wird elektrische Ladung von V_{dd} zur Lastkapazität C_{out} oder von C_{out} nach *Gnd* transportiert. Hierbei fällt die Leistung am PMOS- oder NMOS-Transistor ab. Für die Energie E_{sw}, die durch das zweimalige Umladen der Lastkapazität in der Zykluszeit $T = \frac{1}{f}$ verbraucht wird, ergibt sich [BM98]:

$$E_{sw} = \int_0^T I(t) * U(t)dt = V_{dd} \int_0^{V_{dd}} C_{out} dV_{out} = C_{out} * V_{dd}^2$$

Entsprechend ergibt sich für die Leistung P_{sw} bei jeweils zweimaligem Umschalten während eines Taktes:

$$P_{sw} = \frac{1}{2} C_{out} * V_{dd}^2 * f$$

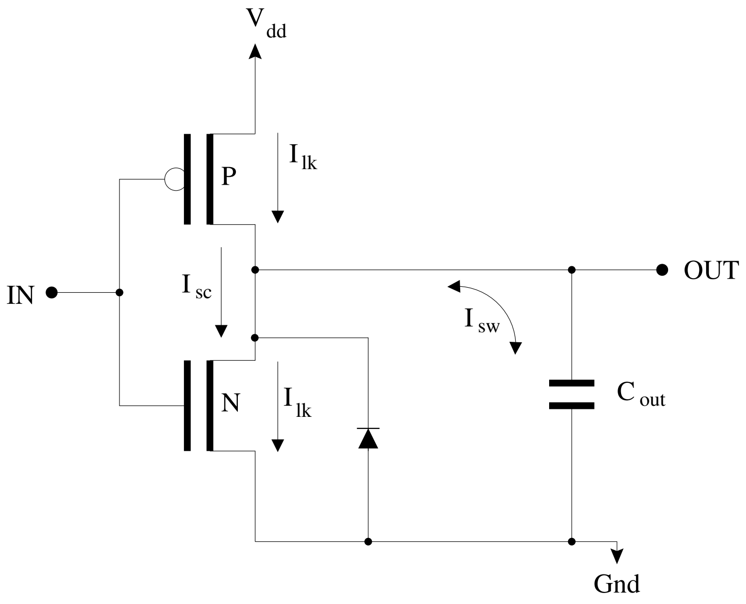


Abbildung 1.3: CMOS-Inverter

Der Energieverbrauch E_{sw} ist proportional zu der Anzahl der Schaltvorgänge eines Gatters. Eine Verringerung der Anzahl der Umschaltungen wäre daher eine Möglichkeit, den Energieverbrauch zu senken. Weitere Ansatzpunkte sind die Reduzierung der Spannung V_{dd}, die in erster Näherung quadratisch eingeht. Hier gibt es insbesondere Verbesserungen durch die Reduzierung der allgemeinen Versorgungsspannung des Cores oder der dynamischen Reduktion während der Laufzeit. Da der Anteil von E_{sw} am Gesamtenergieverbrauch bei aktiven Schaltungen 70 bis 90% beträgt [Syn96], ist das potenzielle Einsparvolumen bei der Schaltaktivität sehr hoch.

1.2.3 Kurzschlussstrom

Bisher sind wir davon ausgegangen, dass die Ladung vollständig durch die Ausgangskapazität aufgenommen wurde. Da die Eingänge jedoch eine begrenzte Flankensteilheit besitzen, sind die beiden PMOS- und NMOS-Transistoren für eine kurze Zeit beide durchgeschaltet. Während dieser Zeit gibt es einen Strom, der von V_{dd} durch die beiden Transistoren nach *Gnd* fließt. Dieser Strom wird Kurzschlussstrom genannt. Der Anteil der Leistung, die durch diesen Kurzschlussstrom verbraucht wird, kann wie folgt abgeschätzt werden [WE94]:

$$P_{sc} = \frac{\beta}{12}(V_{dd} - 2V_t)^3 \frac{t_{rf}}{T}$$

oder:

$$E_{sc} = \frac{\beta}{12}(V_{dd} - 2V_t)^3 t_{rf}$$

mit t_{rf}= Anstiegs-/Abfallzeit, β = Verstärkungsfaktor, T = Taktperiode, V_{dd} = Versorgungsspannung, V_t = Schwellspannung

Der Anteil des Kurzschlussstroms wird in mehreren Untersuchungen auf 10% des durchschnittlichen Gesamtstromverbrauchs geschätzt [BM98].

1.2.4 Statische Leckströme

Der Energieverlust durch Leckströme entsteht durch zwei unterschiedliche Effekte: die gesperrten PN-Übergänge der Transistoren (Dioden) und die "Subthreshold"-Leckströme im Transistor. Bei aktiven Schaltungen beträgt der Anteil der Leckströme weniger als 1% [Syn96]. Bei neueren Designs mit niedrigerer Schwellenspannung ist dieser Anteil stark angestiegen. Andererseits wird er durch technologische Fortentwicklungen auch teilweise wieder reduziert, wie beispielsweise durch die von Intel vorgestellte "Adaptive Body Bias"-Technik, die durch eine selektive Vorspannung am Substrat die Standby-Leistung um mehr als den Faktor 3 verringert [Int02]. Insgesamt betrachtet überwiegt der Energieanteil durch das Umschalten der Transistoren während des aktiven Betriebs.

Im Gegensatz zu den aktiven Schaltungen werden die Leckströme in den Standby-Zeiten, die je nach Applikation einen sehr hohen Anteil an der Gesamtzeit haben können, dominierend. Im Standby, in dem keine Transistoren schalten, sind die statischen Leckströme allein für den Energieverbrauch verantwortlich. Hier kann neben den technologischen Weiterentwicklungen das zeitweise Abschalten von Schaltungsteilen oder die Verringerung der Versorgungsspannung Reduzierungen bewirken.

1.3 Ansatzpunkte für Energieeinsparung

Man erhält somit als Modell für den Energieverbrauch E_{total} die folgende Gleichung für einen Zeitraum von n Taktzyklen:

$$E_{total} = n * (E_{sw} + E_{sc} + E_{lk}) =$$
$$n * C_{out} * V_{dd}^2 + n * \frac{\beta}{12}(V_{dd} - 2V_t)^3 t_{rf} + n * E_{lk}$$

Da der Schaltstrom I_{sw} den wesentlichen Anteil des Energieverbrauchs verursacht, wird nachfolgend der Schwerpunkt auf die Reduzierung des Schaltstroms mit der daraus resultierenden Energie E_{sw} gelegt:

$$E_{sw} = n * C_{out} * V_{dd}^2$$

Ansatzpunkt ist hier die Schalthäufigkeit n, die Ausgangskapazität C_{out} und die Versorgungsspannung V_{dd}. Diese Parameter werden einzeln betrachtet:

1. Reduktion der Schalthäufigkeit n
   Jede Verbesserung der Performance ohne Erhöhung der Taktrate führt im Allgemeinen auch zu einer verringerten Anzahl von auszuführenden Befehlen und damit auch durchschnittlich gesehen zu einer Verringerung der Schalthäufigkeit. Weiterhin kann durch eine optimierte Auswahl von Befehlen der Anteil an aktiven Schaltungsteilen verringert werden, was ebenso zu einer Verringerung der Schalthäufigkeit beiträgt. Neuere Prozessoren bieten ebenfalls die Möglichkeit, Teile des Chips, die vorübergehend nicht benötigt werden, abzuschalten. Dies kann automatisch durch die Hardware geschehen oder durch Einfügung entsprechender Befehle in die Software.

2. Reduktion der geschalteten Kapazitäten C_{out}
   Die Werte der Ausgangskapazitäten umfassen eine große Spannbreite. Im Gegensatz zu Gattern, die nur ein nachfolgendes Gatter treiben müssen, existieren Busse, an die mehrere Bausteine mit großer Eingangskapazität über lange Leitungen angeschlossen sind. Die Anzahl der Schaltvorgänge auf den Bussen spielt daher eine große Rolle und eine Optimierung dieser wenigen Leitungen bewirkt eine nennenswerte Verbesserung. Ähnliches gilt auch für die Wahl der Ressourcen, wie z. B. Speichern. In Systemen mit mehreren unterschiedlichen Speichern können durch eine geschickte Wahl Zugriffe auf langsame, große Speicher durch Zugriffe auf schnelle, kleinere Speicher ersetzt werden. Dies bringt mehrfachen Nutzen durch die Einsparung von Taktzyklen und einen verringerten Energieverbrauch im Speicherbaustein.

3. Reduktion der Versorgungsspannung V_{dd}
   Moderne Prozessoren besitzen häufig die Möglichkeit der softwaremäßigen Variation der Versorgungsspannung des Prozessors. Hierdurch können in Phasen, wo nicht die volle Rechenleistung benötigt wird, die Spannung reduziert und die Befehle langsamer ausgeführt werden. Da die Versorgungsspannung in erster Näherung quadratisch in den Energieverbrauch eingeht, liefert dies trotz der Verlangsamung noch Energieverbesserungen.

Diese hier aufgezeigten Ansätze können nun im Entwurfsprozess aufgegriffen werden, um bei der Generierung von Maschinenprogrammen automatisch Verbesserungen einzubauen. Im nachfolgenden Unterkapitel wird der prinzipielle Entwurfsablauf und Ansatzpunkte zur Optimierung durch Software aufgezeigt. Die ebenfalls mögliche Optimierung der Hardware, die auch den Energieverbrauch reduziert, ist nicht Gegenstand der Arbeit.

1.4 Entwurfsablauf elektronischer Systeme

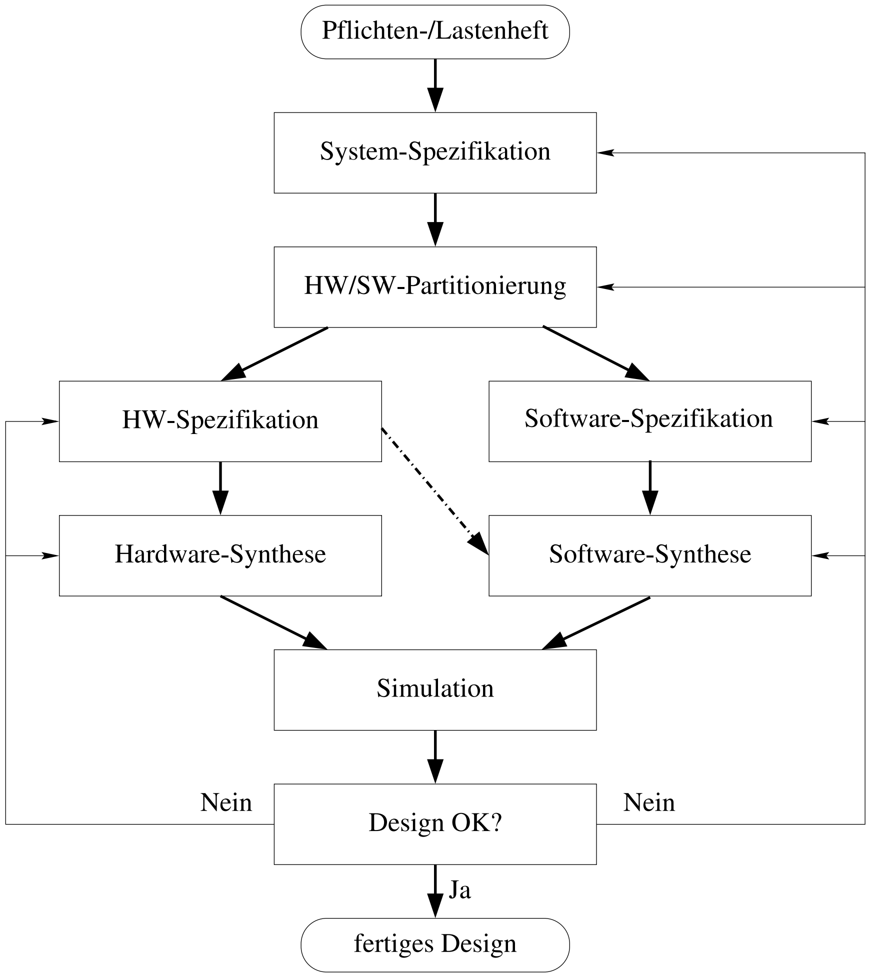


Abbildung 1.4: Entwurfsablauf für elektronische Systeme

Der Softwareanteil in elektronischen Systemen steigt aufgrund der höheren Flexibilität, der kürzeren Entwicklungszeit und der verringerten Notwendigkeit, spezielle Hardwarebausteine zu entwickeln. Gestützt wird dieser Trend durch die komplexer werdenden Systeme und die steigende Funktionalität. Da diese

Software aber auch schnell entwickelt werden muss und immer umfangreicher wird, ist es sehr aufwendig, direkt in Assembler zu programmieren, sodass der Maschinencode fast ausschließlich automatisch durch einen Compiler generiert wird. Dieser bietet sich dann auch als Ansatzpunkt für Optimierungen an. Zuerst soll jedoch der generelle Ablauf bei der Entwicklung eines Systems betrachtet werden.

Bei der Untersuchung des Designprozesses ist vorab festzustellen, dass für das Ziel der Energiereduzierung keine allgemeine und einfache Lösung existiert, und daher beim Design dieser Systeme auf allen Ebenen versucht werden muss, den Verbrauch zu verringern. Ein typischer Entwurfsablauf für Systeme, der sowohl den Entwurf von Hardware als auch von Software umfasst, wie es beispielsweise für die eingebetteten Systeme üblich ist, wird in Abbildung 1.4 dargestellt. Auf oberster Ebene wird mit der *System-Spezifikation* des Systems begonnen. Hier können die Anforderungen aus dem Lastenheft oder dem Pflichtenheft zu einer Spezifikation weiterentwickelt werden. Im nächsten Detaillierungsschritt *HW/SW-Partitionierung* wird über die Aufteilung der Funktionalität in Hardware und Software entschieden.

Während die Software den Vorteil hat, dass sie in der Produktion - abgesehen vom Speichermedium - kostenlos vervielfältigt werden kann, ergibt sich die Notwendigkeit, Funktionalitäten in Hardware zu realisieren, meistens durch die benötigte Rechenleistung. Während die Programme in Form einzelner Instruktionen hintereinander abgearbeitet werden, kann z. B. ein MP3-Encoder oder -Decoder in Hardware parallel zum Prozessor Konvertierungen vornehmen. Die HW/SW-Partitionierung ist die notwendige Vorarbeit, um die beiden Bestandteile in den nachfolgenden Schritten jeweils getrennt zu spezifizieren und anschließend automatisch als Hardware oder Software zu *synthetisieren*. Die HW-Synthese kann ggf. auf die Auswahl von vorhandenen Hardwarebausteinen beschränkt sein, wenn der Entwurf von Spezialbausteinen nicht notwendig ist. Die Software hingegen besteht heute zum großen Teil aus einem speziell für eine Anwendung geschriebenen Programm. Für die Synthese bzw. Transformation der Hochsprachenprogramme in Maschinencode werden nach Optimierungen auf der Hochsprachenebene [CWG$^+$98, CDK$^+$02, Fal02] Compiler eingesetzt, die seit mittlerweile 30 Jahren elementar für eine effiziente Softwareentwicklung sind. Als unterschiedliche Optimierungsziele waren früher nur Performance und Speicherbedarf relevant. In den letzten Jahren ist aus oben genannten Gründen das Optimierungsziel Energieverbrauch hinzugekommen. Hierbei sind Informationen aus der Hardware-Spezifikation notwendig, um z. B. die entworfene Speicherhierarchie bei der Programmgenerierung effizient auszunutzen. Dies ist ein wichtiger Aspekt dieser Arbeit, der in späteren Kapiteln noch vertieft wird.

Die Zusammenfügung von Hardware und Software ergibt danach das spezifizierte System, welches durch *Simulation* auf Einhaltung der Anforderung an Funktionalität, Performance und Energieverbrauch überprüft werden kann. Wenn die Anforderungen nicht erfüllt sind, ist ein Rücksprung in eine der vorherigen Phasen notwendig. Es muss nun versucht werden, möglichst automatisch und mit einer minimalen Anzahl von Rücksprüngen insbesondere zu frühen Designphasen, die verschiedenen Entwicklungsstufen zu verbinden und mit unterschiedlichen Optimierungen den Energieverbrauch zu senken.

1.5 Überblick und verwandte Arbeiten

Das Ziel dieser Arbeit ist es, die Möglichkeiten eines Compilers zu untersuchen, energiesparende Programme zu generieren. Der Schwerpunkt liegt auf Techniken, die sich von den üblichen Performance- und Programmgrößen-Optimierungen unterscheiden. Detailliert vorgestellt werden insbesondere die Techniken, die den größten Energiegewinn liefern.

Als Basis der Untersuchungen wird ein Compiler-Framework, welches in Ergänzung zu bisherigen Performance- und Codegrößen-Optimierungen auch auf Energieverbrauch optimieren kann, präsentiert. Die Grundlage für die Energieoptimierung bildet ein Energiemodell auf Instruktionsebene. Das Framework erlaubt damit die Betrachtung der verschiedenen Compilerphasen und die jeweiligen Möglichkeiten, auf den Energieverbrauch Einfluss zu nehmen. Die profitabelsten Techniken werden detailliert vorgestellt, wozu insbesondere die Optimierung der Speicherhierarchie gehört.

Die Arbeit beginnt nach dieser Einleitung in Kapitel 2 mit dem prinzipiellen Aufbau von Compilern. Es wird das Framework beschrieben, welches für die Entwicklung von Compilern für RISC-Prozessoren verwendet werden kann. Den Schwerpunkt bilden die Berücksichtigung des Energieverbrauchs im Framework und die Ergänzung durch Energieoptimierungen.

Es sind verschiedene Compiler-Frameworks verfügbar, die für Forschungszwecke eingesetzt werden können. Diese werden in Kapitel 2.3 vorgestellt. Eine explizite durchgängige Unterstützung der Energieoptimierung in den Compilerphasen findet sich jedoch in keiner dieser Plattformen.

Als verwandte Arbeiten in der Phase der Instruktionsauswahl sind hier Untersuchungen von Krishnaswamy et al. [KG02] zu benennen, die das optimale Umschalten zwischen den beiden Instruktionssätzen des betrachteten RISC-Prozessors ARM7T zum Gegenstand haben. Außerhalb des Einflusses des Compilers wird in weiteren Arbeiten [Bel00] die Umschaltung der Spannung und Taktfrequenz des Prozessors durch das Betriebssystem beschrieben.

Als Grundlage der Untersuchungen in Kapitel 3 wurden im Rahmen dieser Forschungsarbeiten Energiemessungen an einem realen Prozessor und Speicherbausteinen durchgeführt [The00] und in ein neu entwickeltes Energiemodell [Kna01, SKWM01] einbezogen. Die bisher existierenden Modelle auf Instruktionsebene von Tiwari et al. [TL98, TMW94b, TMW96], Lee et al. [LEMC01, CKL00], Russell et al. [RJ98] und Simunic et al. [SBM99] sowie die Notwendigkeit für die Entwicklung eines neuen Modells werden beschrieben. Erste vorliegende Untersuchungen für den ARM7T von Sinvevriotis et al. [SS99] werden ebenfalls betrachtet. Weiterhin werden in Kapitel 3 die Grundlagen für die Implementierung dieses Modells im Compiler und die Integration des Compilers in die Software-Entwicklungsumgebung vorgestellt.

Mit den geschaffenen Grundlagen des Energiemodells und einer entsprechenden Datenbasis wird in Kapitel 4 der Speicher und insbesondere die Speicherhierarchie betrachtet. Aufgrund des relativ hohen Anteils des Energieverbrauchs bei Speicherzugriffen im Vergleich zum Prozessorverbrauch besteht hier ein noch größeres Potenzial. Im Rahmen einer ersten Optimierung werden nicht verwendete Prozessorregister benutzt, um Hauptspeicherzugriffe zu reduzieren. Im nächsten Schritt werden neben dem Hauptspeicher zusätzlich Caches und Scratchpad[3]-Speicher zur Hierarchie hinzugefügt. Die auf dieses System angewendeten neuen Optimierungen können auch für weitere Speicherhierarchien eingesetzt werden.

Frühere Forschungsergebnisse liegen insbesondere für das Verhalten und das Optimieren der Caches vor. In den Arbeiten von Przybylski [Prz90] und Carr [Car92] werden der grundsätzliche Aufbau und Optimierung von Speicherhierarchien mit Caches betrachtet und in [BAM98] das Energie- und Performance-Verhalten der Caches untersucht. Andere Untersuchungen nutzen die besonderen Eigenschaften der Speicherbausteine aus [DKV+01]. Speziell mit der Optimierung der Registerallokation beschäftigen sich Callahan et al. [CCK90]. Vorarbeiten zur Effizienzsteigerung von Speicherzugriffen finden sich auch bei Franke [Fra99].

Eine notwendige Voraussetzung für die Energieuntersuchungen ist ein Energiemodell für die Caches. Hier ist insbesondere das Modell von Wilton et al. [WJ94, WJ96], auf dem das Modell für den Scratchpad-Speicher von Banakar et al. [BSL+01, BSL+02] aufbaut, zu nennen. Untersuchungen über den Zusammenhang und Einfluss verschiedener Optimierungen und der Cacheorganisation auf den Energieverbrauch liegen von Kandemir et al. [KVIY00] sowie Shiue et al. [SC99] vor.

Die Ausnutzung des Scratchpad-Speichers für Daten und insbesondere Programmteile ist der Schwerpunkt der Arbeit in Kapitel 5. Diese neue Optimierung wird zu Beginn als statische Variante präsentiert, in der eine feste Belegung des Scratchpad während des Übersetzungsvorgangs bestimmt wird. Als Erweiterung wird die Belegung dynamisch während der Programmlaufzeit geändert und Programmteile jeweils neu hineinkopiert.

[3] kleiner, frei adressierbarer Onchip-Speicher

Ein Ansatz durch Hardware von Ishihara et al. [IY00] identifiziert häufig ausgeführte Instruktionssequenzen und fasst diese zu einer Menge zusätzlicher neuer Instruktionen zusammen, die während der Laufzeit durch einen Decompressor wieder restauriert werden. Bei Benini et al. [BMMP00] wird applikationsspezifisch ein Scratchpad-Speicher mit Decoder generiert. Verwandte Arbeiten, die sich auf Softwaremodifikationen beschränken, finden sich nur wenige; die softwaremäßige Auswahl von Programmteilen und die Verschiebung in einen Scratchpad-Speicher sind nicht bekannt. Der Schwerpunkt der bisherigen Forschung lag auf der Betrachtung der Daten [PDN97, PDN99, SFL98, KRI$^+$01].

Neben der Optimierung der Speichernutzung besteht eine weitere Möglichkeit darin, die Anzahl der Signaländerungen auf den Busleitungen zu reduzieren. Hierzu werden in Kapitel 6 spezielle Codierungstechnikenbeschrieben, die im Compiler implementiert werden können und zur Energiereduzierung beitragen.

In diesem Bereich finden sich sehr viele Forschungsarbeiten wie die Optimierung durch Veränderung der Registerzuordnung bei Mehta et al. [MOI$^+$97]. In den Arbeiten von Murgai et al. [MF99] wird die Bitwechselrate reduziert, in dem eine Verfälschung der Daten innerhalb einer vorgegebenen Toleranz zugelassen wird.

Beschränkt auf die Datencodierung der Instruktionen wurden von Sinevriotis et al. [SS99, SS01] Betrachtungen für den ARM7-Prozessor veröffentlicht. Von Su et al. [STD94] wurde neben der Gray-Codierung, entsprechende Hardwaremodifikationen voraussetzt, auch die Technik des Cold Scheduling, dem Umsortieren der Instruktionen unter Einhaltung der Kontrollfluss- und Datenabhängigkeiten, präsentiert. Dieses Verfahren wurde bei Tiwari et al. [TL98] für einen konkreten 32-bit Embedded Microcontroller angewandt. Weitere Arbeiten zu dieser Codierung der Instruktionen finden sich in [MC95, BR95, CN00, LLHT00, TCR98].

Abschließend wird in Kapitel 7 eine Zusammenfassung über die Arbeit präsentiert und ein Ausblick auf fortsetzende Forschungsarbeiten gegeben.

Kapitel 2

Energieoptimierender Compiler für RISC-Prozessoren

Zur Energieoptimierung werden in dieser Arbeit nur Methoden vorgestellt, die in einen Compiler integriert werden können. Dadurch kann auf einfache Art und Weise in jeder Entwicklungsumgebung durch Erweiterung bzw. Anpassung des Compilers eine Energieoptimierung ohne zusätzliche Phasen und ohne Erweiterung oder zeitliche Verlängerung des Entwurfsablaufs erzielt werden.

Für die beispielhafte Implementierung dieser Methoden wurde ein Compiler für die Programmiersprache C gewählt. Diese imperative Sprache hat einen hohen Verbreitungsgrad bei der Entwicklung technischer Software und hat die Assemblerprogrammierung zum Großteil verdrängt. Die Programmiersprache C wurde Anfang der 70er Jahre von Dennis Ritchie bei den Bell Laboratories für die Implementierung des Betriebssystems UNIX entwickelt und gilt als weitestgehend standardisierte Sprache, die vielen Programmierern inbesondere durch das 1978 erschienene Buch von Kernighan und Ritchie "The C Programming Language" [KR78] bekannt ist. Sie ist eine kleine und übersichtliche Sprache, lässt dem Programmierer andererseits viele Freiheitsgrade und ermöglicht damit einfache Zugriffe auf spezielle Hardwareeigenschaften des Systems. Dies befähigt sie insbesondere zum Einsatz für eingebettete Systeme. Auch ein Wechsel zur objektorientierten Sprache C++ kann problemlos erfolgen, da vorhandene Software aufgrund der Aufwärtskompatibilität weiterverwendet werden kann. Vorhandene C-Compiler können entweder durch eine vorgeschaltete C++ -> C-Transformation oder entsprechende interne Erweiterungen angepasst werden.

Neben der Entscheidung für die Hochsprache musste auch ein Prozessor ausgewählt werden. Während der Bereich der Arbeitsplatzrechner heute durch die Intel Pentium-Prozessorfamilie und die Befehlssatzkompatiblen Prozessoren von AMD u. a. dominiert wird, wird jährlich eine noch höhere Anzahl von Prozessoren weitgehend unsichtbar in technischen Systemen vornehmlich zu Steuerungszwecken eingesetzt. Einer der Marktführer ist die Fa. ARM Ltd. [ARM], die mit der ARM-Familie eine erfolgreiche 32-Bit-RISC-Prozessorserie entwickelt hat. Diese Prozessoren zeichnen sich durch ein schlankes Design mit wenigen Registern und insbesondere wenigen Kernbefehlen aus. Der Flächen- und Energiebedarf dieser RISC-Prozessoren ist sehr gering und sie werden daher in Handys und vielen anderen mobilen Applikationen in großen Stückzahlen eingesetzt. Der kleinste und energiesparendste Prozessor ist der ARM7-Prozessor [ARM95a], dessen Befehlssatz als Zielsprache des Compilers gewählt wurde.

Im folgenden Unterkapitel wird der grundsätzliche Aufbau eines Compilers beschrieben. Während die meisten Ausführungen auch auf andere Programmiersprachen und Prozessoren übertragbar sind, liegt der Schwerpunkt auf der Programmiersprache C und den RISC-Prozessoren als Zielprozessor. Die auch häufig in eingebetteten Systemen verwendeten DSPs (= Digitale Signal-Prozessoren) oder auch andere eher heterogen aufgebaute Prozessoren erfordern weitere Eigenschaften in den Compilern, die hier keinen Schwerpunkt darstellen.

2.1 Aufbau von Compilern

Die Aufgabe von Compilern ist allgemein die Übersetzung eines Programms aus einer Eingangssprache in eine Zielsprache. Der häufigste Einsatz betrifft die Transformation einer Hochsprache in ein äquivalentes Programm in der Maschinensprache eines Prozessors. Dadurch wird die semantische Lücke zwischen den beiden Sprachen geschlossen. Es erlaubt dem Programmierer, seine Programme unabhängig von dem gerade verwendeten Prozessor zu entwickeln und dies mit mächtigeren Sprachkonstrukten als die Maschinensprache es erlaubt. Intern existieren meistens noch ein bis drei weitere so genannte Zwischensprachen oder *Intermediate Representations (IR)*, die eine einfachere Struktur als die Ausgangssprache besitzen und die Basis für Optimierungen darstellen. Die ersten Phasen eines Compilers (siehe Abb. 2.1), die das so genannte Front-End bilden, überführen das Hochsprachenprogramm in eine Zwischendarstellung, die unabhängig von der Hochsprache und der Maschinensprache ist. Andere Hochsprachen können daher allein durch das Austauschen des Front-Ends und ohne Modifikation der folgenden Phasen des Back-Ends implementiert werden. Die Verwendung eines anderen Zielprozessors hingegen erfordert den Austausch oder die Anpassung des Back-Ends. Eine Zwischendarstellung bildet daher eine ideale einfache und maschinenunabhängige Darstellungsform.

Das Front-End eines Compilers unterteilt sich mindestens in die folgenden drei Phasen [Muc97], die in Abbildung 2.1 dargestellt sind.

2.1.1 Lexikalische Analyse

In dieser ersten Phase werden die Zeichen des Programms einzeln eingelesen. Auf Basis der Programmiersprache werden die einzelnen Zeichen durch Pattern-Matching zu *Tokens* zusammengefasst und an die nächste Phase weitergereicht. Tokens sind beispielsweise Schlüsselwörter, Operatoren, Konstanten, Literale und Satzzeichen. Bei unerlaubten Folgen werden entsprechende Fehlermeldungen generiert [ASU88].

2.1.2 Parser oder syntaktische Analyse

Die Folge erkannter Tokens wird nun verarbeitet und hierarchisch analysiert. Die Tokens werden zu grammatikalischen Sätzen zusammengefasst und als Parse-Baum (auch abstrakter Syntaxbaum genannt) oder in einer anderen Zwischendarstellung dargestellt. Weiterhin wird eine Symboltabelle, die alle Variablen des Programms und ihre Eigenschaften enthält, erzeugt. Illegale Kombinationen oder Folgen werden dem Programmierer als Syntaxfehler zurückgemeldet.

2.1.3 Semantische Analyse

Anhand des Parse-Baums wird statisch geprüft, ob das Programm die semantischen Anforderungen der Programmiersprache einhält. Beispielsweise wird die konsistente Deklaration und Verwendung von Variablen und ihrer Typen überprüft. Als Ergebnis der semantischen Analyse wird eine semantisch korrekte Zwischendarstellung (Medium-Level Intermediate Representation MIR) einschließlich der verschiedenen Symbolinformationen erzeugt (allgemein: annotierter abstrakter Syntaxbaum).

2.1.4 Back-End

Während die bisher beschriebene Zwischendarstellung MIR noch maschinenunabhängig ist, hängt der nachfolgende Teil des Compilers, das so genannte Back-End, von der gewählten Zielarchitektur des Prozessors ab. Die wichtigste Phase des Back-End, der Code-Generator, kann wiederum in die Phasen Instruk-

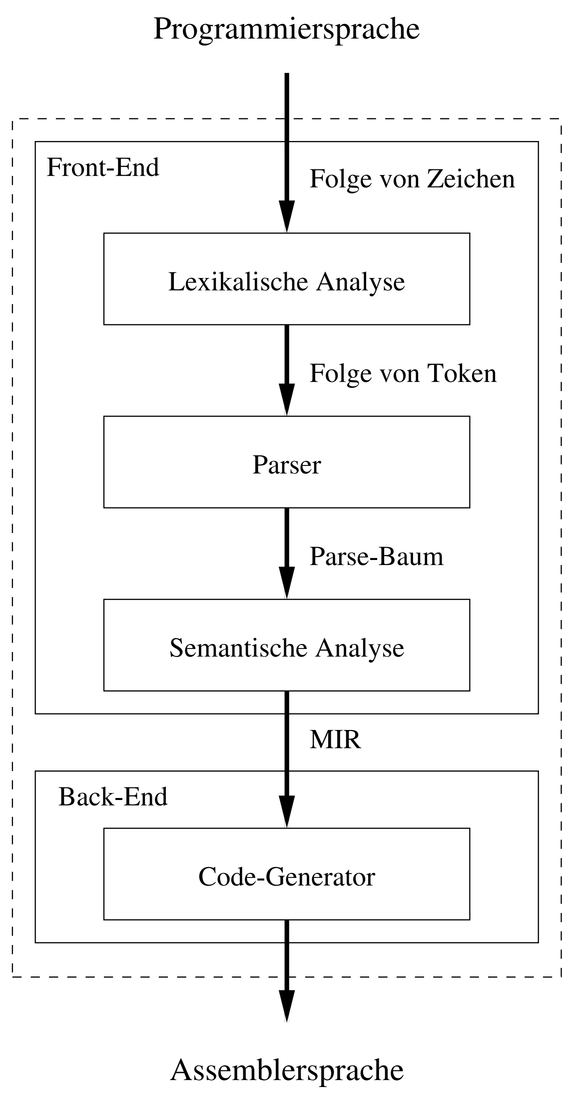


Abbildung 2.1: Compiler mit Phasenunterteilung

tionsauswahl, Registerallokation und Instruktionsanordnung unterteilt werden. Auf Basis der Maschinensprache des Prozessors werden während der Instruktionsauswahl die Konstrukte der Zwischendarstellung durch Instruktionen des Prozessors ersetzt. Entweder werden implizit in dieser Phase oder in einer zusätzlichen Phase, die Registerallokation genannt wird, die Register des Prozessors den Instruktionen zugeordnet. Die letzte Phase, die Instruktionsanordnung, sorgt für die Optimierung der Reihenfolge der Instruktionen, sodass die Anzahl der benötigten Prozessorregister verringert und teilweise ein Auslagern von Registerinhalten in den Hauptspeicher vermieden werden kann.

2.1.5 Optimierende Compiler

Die Anwender eines Compilers verlangen eine ausreichende Codequalität bzgl. Programmgröße, Performance oder Energieverbrauch. Dafür muss der generierte Code an mehreren Stellen während des Compilerlaufs durch so genannte optimierende Compiler verbessert werden. Da die Komplexität des Compilers selbst auch stets berücksichtigt werden muss, werden Optimierungen meistens lokal in einzelnen Phasen oder auf verschiedenen Zwischendarstellungen angewandt. Optimierungen, die über mehrere Phasen durchgeführt werden, sind eher zu vermeiden, da die Komplexität und Laufzeit des Compilers ansteigt und die Wartung dieser komplexen Software erschwert wird. Allerdings wird das vorhandene Optimierungspotenzial nicht unbedingt ausgenutzt. Während diese Begrenzung der Optimierung auf einzelne Compilerphasen bei RISC-Prozessoren möglich ist, muss bei anderen irregulären Zielprozessoren, z. B. DSPs, eine Phasenkopplung stattfinden, um die notwendige Codequalität erreichen zu können.

In optimierenden Compilern werden anstatt der bisher nur einfach vorhandenen Zwischendarstellung zwischen Front-End und Back-End auch innerhalb des Front-Ends und Back-Ends weitere Zwischendarstellungen als Basis für Optimierungen ergänzt.

Innerhalb des Front-Ends wird in einigen optimierenden Compilern eine High-Level Intermediate Representation (HIR) eingefügt, die noch die Schleifen- und Arraykonstrukte des Originalprogramms beinhaltet. Auf dieser Darstellung können insbesondere Optimierungen wie z. B. "skalare Ersetzung von Array-Referenzen" oder Schleifenoptimierungen ausgeführt werden.

Die schon beschriebene programmiersprachen- und maschinenunabhängige Darstellung zwischen Front-End und Back-End wird zur Unterscheidung als Medium-Level Intermediate Representation (MIR) bezeichnet. Die genaue Ausprägung der MIR unterscheidet sich jedoch von Compilersystem zu Compilersystem.

Als letzte Zwischendarstellung, die innerhalb des Back-Ends eingesetzt wird, wird eine so genannte Low-Level Intermediate Representation (LIR) eingeführt. Diese ist maschinenabhängig und muss daher für jeden Prozessor angepasst werden. Andererseits ermöglicht die Berücksichtigung der spezifischen Hardwareeigenschaften des Prozessors bessere Optimierungen des Maschinencodes. Grundsätzlich können auf der LIR alle Optimierungen mit höchster Güte implementiert werden. Allerdings wären diese Optimierungen für jeden Prozessor wieder neu zu programmieren, sodass es nahe liegend ist, architekturunabhängige Optimierungen auf den höheren Ebenen, MIR oder HIR, einzusetzen.

Das Compilerdesign mit einer MIR und einer LIR wird in heutigen Compilern z. B. von Sun für die SPARC-Architektur, von Intel für die x386-Architekturfamilie und von Silicon Graphics für MIPS eingesetzt [Muc97]. Andere Compiler, wie IBMs Compiler für den PowerPC, beschränken sich allein auf die LIR. Bei der Festlegung der Anzahl der Zwischendarstellungen bleibt stets abzuwägen, wie groß die Anzahl der zu unterstützenden Programmiersprachen, die Unterschiedlichkeit der betrachteten Zielarchitekturen sowie die Wiederverwendung der einzelnen Compilermodule sind. Daher kann eine allgemein gültige Form nicht festgelegt werden.

Als Grundlage für diese Arbeit können wir den in Abbildung 2.2 dargestellten Compileraufbau für einen optimierenden Compiler als Basis verwenden. Neben den schon beschriebenen Phasen wurden hier eine explizite Phase für die Erzeugung der HIR (= Intermediate-Code-Generator) sowie Zwischenphasen

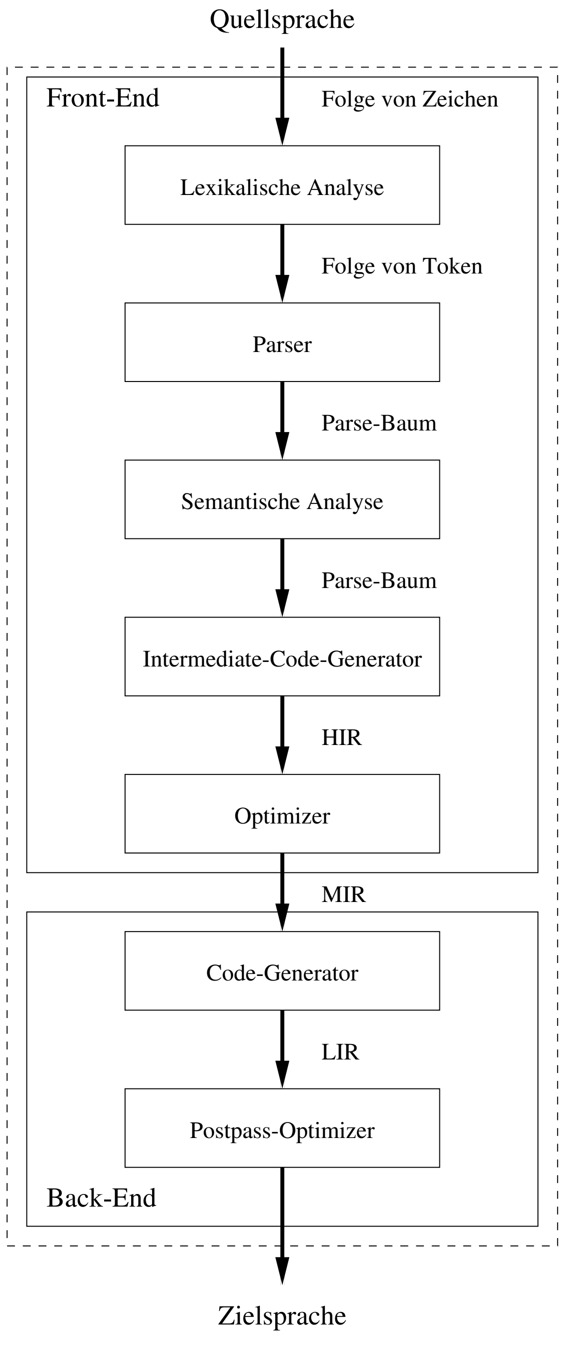


Abbildung 2.2: optimierender Compiler

zur Optimierung auf Basis der MIR (= Optimizer) und zur Optimierung auf Basis der LIR (= Postpass-Optimizer) eingefügt.

Im nächsten Unterkapitel wird die Zielarchitektur näher beschrieben, für die es galt, einen Compiler zu entwerfen.

2.2 Beschreibung der Zielarchitektur

2.2.1 RISC-Prozessoren

Für den Einsatz in eingebetteten Systemen können Prozessoren verschiedener Klassen verwendet werden. Abhängig von der Rechenleistung, der Möglichkeit der Parallelverarbeitung oder der Notwendigkeit zur Verarbeitung analoger Eingangs- und Ausgangssignale muss der geeignete Prozessor bestimmt werden. Insbesondere für energiesensitive Applikationen, die hier schwerpunktmäßig betrachtet werden, sind kleinere und einfachere Prozessoren zu bevorzugen. In diesem Bereich sind RISC-Prozessoren (= Reduced Instruction Set Computer) von Vorteil, da sie eine schlanke Architektur, wenige Register, eine geringe Zahl von Funktionseinheiten und einen kleinen Befehlssatz besitzen.

Zu diesen Prozessoren gehört auch die schon erwähnte ARM-Familie, die auf einem gemeinsamen Kern basiert und im Laufe der Jahre um weitere Prozessoren mit Zusatzbefehlen, Caches, o. ä. erweitert wurde. Als Besonderheit ist zu erwähnen, dass die Fa. ARM ausschließlich das Design vermarktet und der Prozessor von vielen Halbleiterherstellern in Lizenz gefertigt wird. Im Jahre 2000 wurden 400 Millionen ARM-Prozessoren in Lizenz hergestellt, was einem Marktanteil von 77% bei 32-Bit-Prozessoren entspricht [Sti02]. Eine der neuesten Varianten ist aus der Zusammenarbeit mit Intel erwachsen und wird unter dem Namen XScale [XSc] vermarktet. Der kleinste Prozessor mit dem geringsten Energieverbrauch ist der ARM7TDMI, der in vielen mobilen Applikationen wie Handys und MP3-Playern zum Einsatz kommt. Daher wurde dieser Prozessor als Grundlage für diese Arbeit gewählt, zumal die Optimierungen fast ohne Ausnahme auch auf die anderen Prozessoren der ARM-Familie übertragbar sind. Übereinstimmungen in der Architektur, wie etwa der Befehlssatz sowie Anzahl und Art der Funktionseinheiten, bestehen aber auch mit weiteren RISC-Prozessoren wie der MIPS32- [MIP99] und der SPARC-Architektur [SPA92].

Da der Aufbau des ARM7TDMI für die später präsentierten Optimierungen relevant ist, wird seine Architektur hier vorgestellt.

2.2.2 Architektur ARM7TDMI

Die Architektur des ARM7TDMI, dessen Blockschaltbild in Abbildung 2.3 dargestellt ist, besitzt folgende Merkmale:

- 32-Bit-Architektur,
- 32-Bit-Registerbank,
- gemeinsamer Daten- und Programmspeicher mit 32-Bit-Adressbus,
- 32-Bit-ARM-Instruktionssatz,
- zusätzlicher 16-Bit-Thumb-Instruktionssatz,
- drei-stufige Instruktions-Pipeline mit Fetch-, Decode- und Execute-Phasen,
- unterstützte Datenwortbreiten Byte (8-Bit), Halfword (16-Bit) und Word (32-Bit),

- 32-Bit-ALU, Barrelshifter, Multiplizierer,
- hohe MIPS[1] / Watt-Kennzahl.

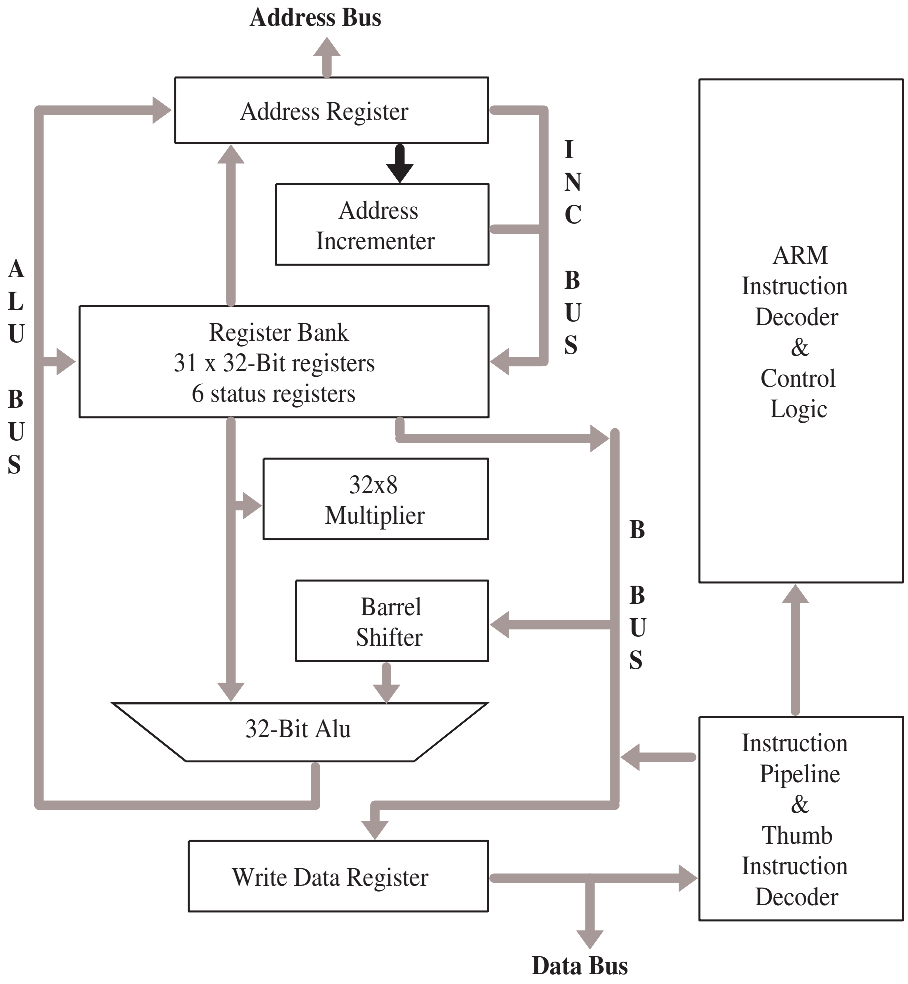


Abbildung 2.3: Blockschaltbild ARM7TDMI

Es ist anzumerken, dass RISC-Prozessoren im Gegensatz zu den CISC-Prozessoren (= Complex Instruction Set Computer) dafür ausgelegt sind, im Mittel in jedem Takt einen Befehl auszuführen. Der CPI-Wert (CPI = cycles per instruction) erreicht dann fast 1 [Mar00]. Dadurch wird fast in jedem Taktzyklus auf den Speicher zugegriffen, um den nächsten Befehl zu laden. Dieser höheren Anzahl von Speicherzugriffen zum Holen der Befehle steht im Vergleich zu den CISC-Prozessoren der Vorteil gegenüber, dass ein RISC-Prozessor weniger Transistoren und Fläche benötigt und dadurch sehr viel energiesparender bei der Ausführung eines Befehls arbeitet [SCG95]. Dieser Unterschied zwischen RISC- und CISC-Prozessoren wird noch zusätzlich durch komprimierte Instruktionssätze wie den Thumb-Instruktionssatz beim ARM7T verringert. Insgesamt kann man feststellen, dass RISC-Prozessoren für energiesensitive Applikationen besonders geeignet sind, da sich die kleinere Schaltung und geringere Fläche positiv für den Energieverbrauch auswirken und dadurch den Nachteil der geringeren Codedichte anders ausgleichen.

[1]Millionen Instruktionen pro Sekunde

2.2.3 ARM-Instruktionssätze

Der ARM7 beinhaltet den ARMv4-Instruktionssatz, der den für die Prozessorfamilie grundlegenden 32-Bit-RISC-Instruktionssatz darstellt und folgende Merkmale besitzt:

- 32-Bit-Instruktionswortbreite,
- Load/Store-Architektur (RISC-Eigenschaft),
- bedingte Ausführung von Befehlen,
- Multiply/Accumulate-Befehle (DSP-Eigenschaft).

Die geringe Anzahl von drei Pipeline-Stufen bedeutet, dass die maximale Taktfrequenz wegen der Aufteilung des Befehls in nur drei Phasen geringer ist als bei Prozessoren wie den neueren ARM9- und ARM10-Architekturen mit mehr Pipeline-Stufen. Andererseits ist weniger Hardware-Logik notwendig, um Datenabhängigkeiten zwischen den Befehlen zu behandeln. Da u. a. diese Logik einen erheblichen Anteil an der Komplexität eines Prozessors hat, kann der ARM7 die Befehle mit einem geringeren Energiebedarf abarbeiten.

Erweiterungen dieses ARMv4-Instruktionssatzes existieren für neuere Prozessoren der Familie z. B. als v5-Architektur mit zusätzlichen DSP-Befehlen, die eine Sättigungsarithmetik unterstützen.

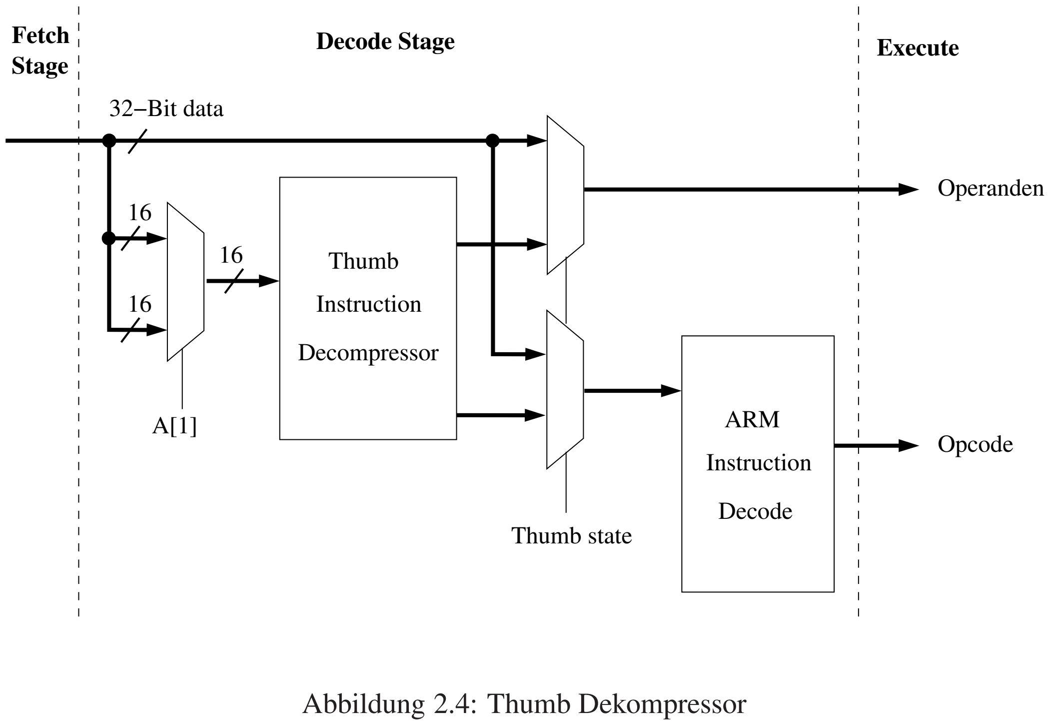


Abbildung 2.4: Thumb Dekompressor

Eine weitere Ergänzung, die durch ein zusätzliches “T” an der Architekturbezeichnung gekennzeichnet ist, existiert mit dem 16-Bit-Thumb Instruktionssatz. Dieser wurde mit dem Ziel einer Reduzierung der Codegröße entwickelt. Auf der Basis mehrerer Benchmarks wurden die am häufigsten verwendeten Befehle ermittelt. Diese Befehle sind in einem nur 16-Bit breiten Instruktionswort kodiert und werden intern nach dem Holen aus dem Speicher in der zweiten Pipeline-Stufe (siehe Abb. 2.4) ohne zusätzlich notwendige Taktzyklen auf den vorhandenen 32-Bit-Instruktionssatz expandiert [ARM95b]. Dies ist am Beispiel eines Add-Befehls in Abbildung 2.5 dargestellt. Programme, die in den Thumb-Instruktionssatz umkodiert werden, benötigen eine um durchschnittlich 30% geringere Programmgröße. Für die zusätzliche Thumb-Logik wird lediglich ein um 2% höherer Leistungsverbrauch bei der Ausführung eines Thumb-Befehls benötigt [Seg97].

Der Thumb-Befehlssatz wird aufgrund der verringerten Speicherzugriffe auch für energiesensible Anwendungen empfohlen. Besitzt das System zwischen Prozessor und Instruktionsspeicher einen 32-Bit-Datenbus können gleichzeitig zwei Thumb-Instruktionen in den Prozessor geholt werden. Die zweite Instruktion wird zwischengespeichert und im nächsten Schritt ausgeführt. Bei einem 16-Bit-Datenbus könnte dementsprechend pro Speicherzugriff nur ein Thumb-Instruktionswort gelesen werden, sodass im Vergleich zum 32-Bit-Datenbus die doppelte Anzahl an Speicherzugriffen mit halber Wortbreite ausgeführt wird.

Der Thumb-Instruktionssatz weist im Vergleich zum ARM-Instruktionssatz folgende Unterschiede auf:

- Adressierung der höheren Register R8 bis R15 nur eingeschränkt mit wenigen Befehlen möglich,
- keine Multiply/Accumulate-Befehle,
- keine bedingte Ausführung von Befehlen,
- geringere Zahl von Adressierungsarten,
- eingeschränkter Wertebereich von Immediates.

Durch diese Einschränkungen kann es notwendig werden, einen einzelnen ARM-Befehl durch mehrere Thumb-Befehle ersetzen zu müssen. Die Vor- und Nachteile des Thumb-Modus müssen daher für den Einzelfall in Abhängigkeit vom Optimierungsziel, der Datenbusbreite und der Speicher-Latenzzeit[2] abgewogen werden [KG02]. Es sind auch Kombinationen zwischen beiden Modi denkbar, bei denen die zeitkritischen Funktionen mit dem ARM-Instruktionssatz realisiert werden, um die Zeitanforderungen zu erfüllen, und zeitunkritische Funktionen mit dem Thumb-Modus. Dadurch wird der Programmspeicherbedarf bei nur geringem Performance-Nachteil und unter Einhaltung der harten Realzeitanforderungen verringert. Für die Optimierung des Energieverbrauchs hat bei hohem Energieanteil der Speicherzugriffe am Gesamtsystem der Thumb-Modus den Vorteil durch die verringerte Anzahl von Programmspeicherzugriffen, sodass insgesamt der Energieverbrauch verringert werden kann [The00].

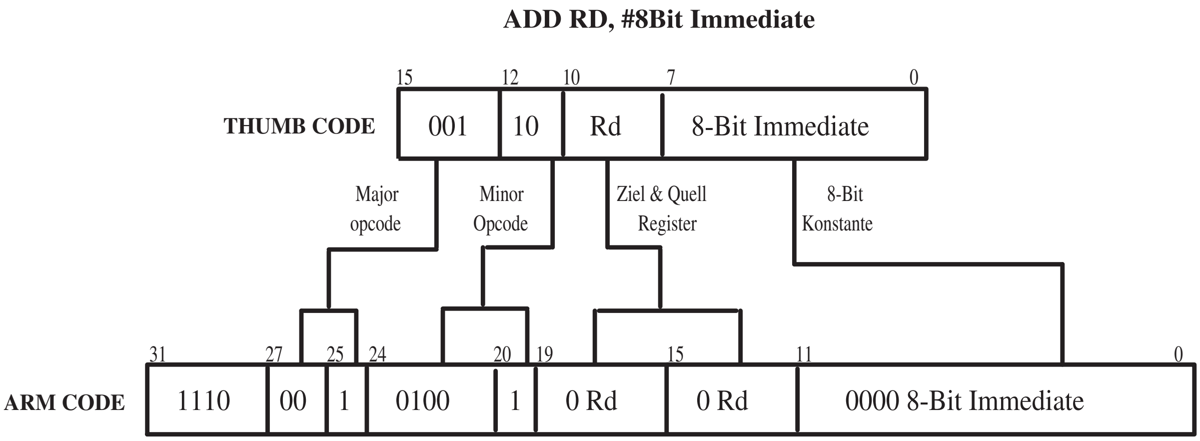


Abbildung 2.5: Expansion eines Add-Befehls vom Thumb in den ARM-Instruktionssatz

[2]Verzögerung beim Speicherzugriff

2.3 Existierende Compiler-Baukästen

Zur Durchführung von Forschungsarbeiten im Bereich der Compileroptimierungen muss als Ausgangsbasis eine entsprechende Umgebung gewählt werden. Die Untersuchungen in dieser Arbeit starten in dem Entwurfsablauf (siehe Abb. 1.4) nach der Aufteilung der Funktionalität in Hardware und Software, sodass die *HW/SW-Partitionierung* als gegeben angesehen wird. Die erstellte *HW-Spezifikation* wird ohne Berücksichtigung der Software-Spezifikation in der Phase *HW-Synthese* umgesetzt. Umgekehrt wird jedoch die erstellte *SW-Spezifikation* in der Phase *SW-Synthese* unter Berücksichtigung der HW-Spezifikation in ein ablauffähiges Programm überführt. Hier werden die speziellen Eigenschaften der gewählten Hardware in den Compilierungsprozess mit einbezogen. Neben den Hardwareeigenschaften wie Instruktionssatz oder Registeranzahl betrifft dies insbesondere zusätzliche Eigenschaften wie den Energieverbrauch bei der Abarbeitung einzelner Befehle oder beim Umschalten zwischen Funktionseinheiten sowie Speicherzugriffszeiten, den Energieverbrauch des Speichers und der Datenübertragung auf Bussen. Am Ende der Phase der *SW-Synthese* werden noch ein Assembler- und Linkerlauf durchgeführt bevor in der Phase *Simulation* die Korrektheit des generierten Designs validiert werden kann.

Die beschriebenen Anforderungen bzgl. der Berücksichtigung spezieller Hardwareeigenschaften wie Energieverbrauch werden von herkömmlichen Industrie-Compilern nicht erfüllt, sodass diese nicht als Basis der Arbeiten verwendet werden können. Die folgenden Compiler-Baukästen werden häufig zu Forschungszwecken für RISC-Prozessoren verwendet und nachfolgend mit ihren Eigenschaften beschrieben:

1. GCC
   Der GCC-Compiler [gcc] wurde 1984 von Richard Stallman im Rahmen der Free Software Foundation (FSF) entwickelt und zur Verfügung gestellt. Dieser retargierbare Compiler ist als Sourcecode verfügbar und kann frei verwendet, modifiziert und weiterverteilt werden. Im Laufe der Zeit wurden für den GCC Front-Ends für weitere Programmiersprachen wie Fortran, C++ und Java entwickelt. Ebenso wurden unterschiedliche Back-Ends für eine große Zahl von CISC- und RISC-Prozessoren wie x86, M68000, PowerPC, Sparc, MIPS und auch ARM ergänzt. Der GCC generiert den Code in ca. 20 Phasen [LM01], die auf der Zwischendarstellung ("Register Transfer Language" genannt) arbeiten. Der GCC zählt heute als weitverbreiteter und qualitativ hochwertiger Compiler, der für viele Betriebssysteme verfügbar ist. Für den gewählten ARM7-Prozessor mit Thumb-Instruktionssatz lag zum Zeitpunkt der Erhebung allerdings kein Back-End vor. Außerdem ist der GCC aufgrund seiner hohen Flexibilität auch sehr komplex in der Installation und Konfiguration. Desweiteren ist das Entwickeln eines neuen Back-Ends aufgrund der umfangreichen und komplexen Struktur nicht trivial. Die für diese Arbeit notwendige Eigenschaft der Bewertung des Energieverbrauchs neben der Programmgröße und Laufzeit ist nicht vorhanden.

2. LCC
   Neben dem GCC wird der LCC ("little C Compiler") sehr häufig als Basis für Forschungen im Bereich von Compileroptimierungen eingesetzt. Es handelt sich ebenfalls um einen retargierbaren C-Compiler, der allerdings nur aus 13.000 Zeilen Sourcecode [LM01] besteht und dessen Implementierung sehr ausführlich in Form eines Buches dokumentiert wurde [FH95]. Standardmäßig wird der LCC für die Prozessoren ALPHA, SPARC, MIPS und x86 zur Verfügung gestellt [lcc]. Als Zwischendarstellungen dienen Datenflussgraphen (DFGs) mit angehängten Typ- und Größeninformationen. Es sind nur sehr wenige Standard-Optimierungen und eine lokale Registerallokation implementiert, sodass die erwartete Codequalität geringer als z. B. beim GCC ist. Dieser Compiler wurde beispielsweise für die Arbeiten von Tiwari et al. [TMW94a] verwendet.

3. SUIF
   Unter dem Namen SUIF (Stanford University Intermediate Format) wird von der Stanford Compiler Group eine Infrastruktur zur Entwicklung von optimierenden und parallelisierenden Compilern

zur Verfügung gestellt [SUI]. Das Toolkit besteht aus Front-Ends für C (basierend auf dem LCC) und Fortran, einem optimierenden Back-End für den MIPS-Prozessor und verschiedenen Tools zur Entwicklung von Compilern. Eine Besonderheit ist der Loop-Level Parallelism and Locality Optimizer, der die Speicherhierarchie und Parallelität von Operationen ausnutzt. Im Gegensatz zu den bisher beschriebenen Compilern enthält SUIF zwei Zwischendarstellungen. Die high-SUIF IR enthält Konstrukte für Schleifen, Bedingungen und Arrayzugriffe, auf deren Basis komplexe Optimierungen ausgeführt werden können. Die alternative low-SUIF IR stellt maschinenabhängigen und assemblerähnlichen Code dar und ermöglicht ebenfalls die Ausführung verschiedener assemblernaher Optimierungen. Eine Unterstützung für die Optimierung des Energieverbrauchs ist nicht vorgesehen.

4. Zephyr
   Eine Erweiterung bzw. Ergänzung des SUIF-Ansatzes stellt das Projekt Zephyr [Zep, ADR98] dar, welches Tools für eine "nationale Compiler Infrastruktur" liefern soll. Das Ziel dieses Projektes ist es, Compiler aus einzelnen Teilen (Front-End, Back-End, Optimierer) zusammenbauen zu können und nur einzelne Teile, die für die jeweiligen Forscher wesentlich sind, individuell ersetzen zu müssen. Die Zwischendarstellungen des Compilers werden durch eine Abstract Syntax Description Language (ADSL) beschrieben, woraus automatisch Teile des Compilers generiert werden können. Im Einzelnen besteht Zephyr aus einem maschinenunabhängigen Optimierer (VPO) mit Instruktionsauswahl, Instruktionsanordnung und globalen Optimierungen, Back-End-Komponenten, die in dem maschinenabhängigen Teil verwendet werden können, sowie den notwendigen Verbindungen zwischen diesen Einzelteilen. Die maschinenabhängigen Compilerkomponenten werden aus kompakten Spezifikationen der Zielmaschine automatisch generiert. Eine besondere Unterstützung zur Optimierung des Energieverbrauchs ist nicht ersichtlich.

5. Trimaran
   Insbesondere für Instruction Level Parallel (ILP) Architekturen wurde die Compiler-Plattform Trimaran entwickelt [Tri]. Die ILP-Eigenschaft von Architekturen bedeutet, dass mehr als eine Operation pro Taktzyklus von einem einzelnen Prozessor ausgeführt wird. Die Tools von Trimaran sollen Forscher insbesondere bei Arbeiten an Back-Ends (bestehend aus Instruktionsauswahl, Registerallokation und maschinenabhängigen Optimierungen) unterstützen. Der besondere Schwerpunkt liegt auf Explicitly Parallel Instruction Computing (EPIC) Architekturen, wo durch den Compiler bestimmt wird, welche Abhängigkeiten zwischen den Operationen existieren und festgelegt wird, welche Operationen parallel ausgeführt werden können. Die Infrastruktur von Trimaran besteht aus einer Maschinenbeschreibungssprache (HMDES), einem Front-End für C einschließlich einer größeren Anzahl von maschinenunabhängigen High-Level-Optimierungen sowie einem Back-End (Instruktionsanordnung, Registerallokation, maschinenabhängigen Optimierungen), welches durch die Maschinenbeschreibung parametrisiert wird. Die einzelnen Stufen des Back-Ends können einfach modifiziert oder auch ersetzt werden.

6. LANCE
   Eine weitere Plattform für die Entwicklung von C-Compilern ist LANCE [LAN]. Das LANCE Front-End wurde an der Universität Dortmund entwickelt und liefert ein Front-End zur Konvertierung von C-Code in eine einfache maschinenunabhängige IR. Auf dieser IR, die einem C-Subset in Form eines 3-Adress-Code entspricht, werden Standard-Optimierungen ausgeführt. Mitgeliefert wird eine Bibliothek für den Zugriff auf die Datenstrukturen und verschiedene Analysen auf Basis dieser IR. Mit LANCE als Front-End wurden verschiedene Compiler für die folgenden Prozessoren entwickelt: TI 'C5x und C6x, AMS GEPARD DSP Core, Philips Trimedia und weitere applikationsspezifische Prozessoren (ASIPs). Für die Optimierung des Energieverbrauchs sind keine Vorkehrungen getroffen, da diese in Teilen des Back-Ends implementiert werden müssen, die maschinenspezifisch jeweils neu zu entwickeln sind.

Die Sichtung der verfügbaren Compiler ergab, dass diese lediglich auf die Optimierung nach Laufzeit und Programmgröße ausgerichtet sind. Das Optimierungsziel Energieverbrauch musste daher grundlegend neu eingebaut werden. Ein Back-End für den gewählten ARM7 Thumb-Instruktionssatz ist ebenfalls nicht verfügbar. Der weit verbreitete tcc, ein Thumb C-Compiler der Fa. ARM, bietet ebenfalls keine Energieoptimierung und ist auch nicht in Source-Form als Basis für Erweiterungen erhältlich.

Nach intensivem Abwägen der Vor- und Nachteile der verschiedenen Compiler-Baukästen wurde ein Compiler auf Basis von LANCE entwickelt. Den Ausschlag gab die einfache Struktur des Front-Ends mit den daraus resultierenden geringeren Aufwändungen für den Entwurf und die Implementierung des Back-Ends sowie der Support im eigenen Hause. Der Aufbau des entwickelten energieoptimierenden Compilers wird im nachfolgenden Unterkapitel beschrieben.

2.4 Compileraufbau für Energieoptimierung

Der für die Forschungen erstellte *encc*-Compiler (siehe Abb. 2.6) besteht aus dem Standard-LANCE Front-End und einem speziell entwickelten Back-End für den ARM7 Thumb-Instruktionssatz. Das Back-End

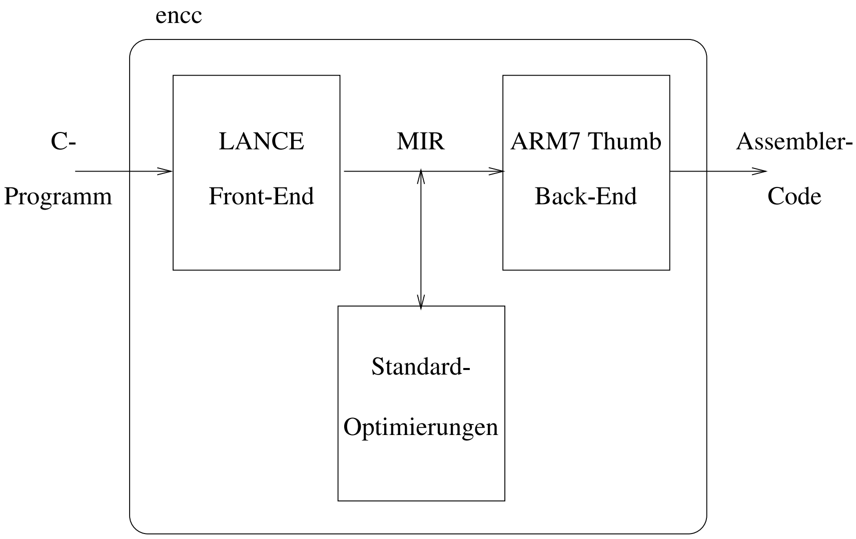


Abbildung 2.6: Aufbau des *encc*-Compilers

wurde prinzipiell nach den Vorschlägen von Appel et al. [AG98] implementiert und enthält keine zusätzlichen Phasen. Es besteht aus der Instruktionsauswahl, der Instruktionsanordnung, der Registerallokation, den Postpass-Optimierungen und den Ergänzungen zur Berücksichtigung des Energieverbrauchs. Nachfolgend wird das Design der einzelnen Compiler-Phasen unter besonderer Betrachtung der Eigenschaften eines energieoptimierenden Compilers beschrieben. Insbesondere bei der Instruktionsauswahl und den Optimierungen muss der Energieaspekt Berücksichtigung finden, da dort entsprechende Energieeinsparungen erzielt werden können.

2.4.1 Front-End

Das Front-End besteht aus den schon beschriebenen Standardphasen der lexikalischen Analyse, des Parsers und der semantischen Analyse. Diese Phasen im Front-End sind maschinenunabhängig und auch unabhängig vom Optimierungsziel (Performance, Programmgröße, Energieverbrauch). Modifikationen sind hier für die zusätzlich aufgeführten Energieoptimierungen nicht notwendig. Das LANCE-Front-End oder ein anderes verfügbares Front-End können daher unverändert eingesetzt werden.

2.4.2 IR-Transformationen

Als Ausgabe des Front-Ends wird bei LANCE maschinenunabhängiger 3-Adress-Code erzeugt. Dieser 3-Adress-Code verwendet als Sprachelemente ein Subset der C-Programmiersprache. Dies ist vorteilhaft, da diese IR (nach der Systematik von Muchnick [Muc97] handelt es sich um eine MIR) für eine Validierung direkt durch einen anderen C-Compiler übersetzt, das generierte Programm ausgeführt und mit den Ergebnissen anderer Compiler verglichen werden kann.

(a) vorher

```
int func1() {
  statement_1;
  ..
  statement_n;
}

int main(void) {
  ..
  func1();
  ..
  func1();  /* 2nd call */
}
```

(b) nachher

```
int main(void) {
  ..
  statement_1;  /* 1st inlining */
  ..
  statement_n;
 ....
  statement_1;  /* 2nd inlining */
  ..
  statement_n;
}
```

Abbildung 2.7: Optimierungstechnik *Function Inlining*

Auf Basis dieser IR werden unterschiedliche Standard-Optimierungen, wie *Dead Code Elimination*, *Copy Propagation, Constant Folding* oder *Function Inlining* ausgeführt. An dieser Stelle ist bereits das Optimierungsziel zu berücksichtigen. Während Optimierungen wie *Dead Code Elimination* stets für alle Ziele Geschwindigkeit, Programmgröße oder Energieverbrauch vorteilhaft sind, muss z. B. bei *Function Inlining* differenziert werden. Die Optimierung *Function Inlining* wird nachfolgend beispielhaft detaillierter betrachtet. Die Ergebnisse lassen sich entsprechend auf weitere Optimierungen übertragen.

Function Inlining (auch *Procedure Integration* oder *Automatic Inlining* genannt [Muc97]) bedeutet, dass eine Funktion an jeder aufrufenden Stelle direkt integriert wird (siehe Abb. 2.7). Dadurch können zusätzliche Befehle für die Übergabe von Parametern sowie Sprünge eingespart werden. Dies ist relativ betrachtet insbesondere für kleinere Funktionen vorteilhaft, da der Anteil von Instruktionen für Parameterübergaben und Sprünge dort stärker ins Gewicht fällt. Ein weiterer Vorteil des *Function Inlining* ist, dass nachfolgende Optimierungen wie das mehrfache Nutzen von Registern übergreifend arbeiten, da deren Analysen sonst an den Funktionsgrenzen enden. Nachteilig ist, dass bei mehr als einem Aufruf die Befehle der aufgerufenen Funktion mehrfach vorhanden sind und dadurch die Programmgröße ansteigt. Bei wenigen Aufrufen kann dies eventuell noch durch eingesparte Parameterübergaben ausgeglichen werden. Nicht anwendbar ist *Function Inlining* jedoch bei rekursiven Funktionsaufrufen, da dann größere Umstrukturierungen vorgenommen werden müssen.

Es ist offensichtlich, dass bei der Optimierung nach Geschwindigkeit die Anwendung des *Function Inlining* stets vorteilhaft ist, da der Aufruf der Funktion und die Parameterübergaben eingespart werden können. Für die Programmgröße ist bei steigender Anzahl von Funktionsaufrufen das Inlining jedoch von Nachteil, da die Funktion mehrfach im Programmspeicher vorliegt.

Bei der Optimierung nach Energieverbrauch kann man sich auf der IR-Ebene grundsätzlich nach der Performance-Optimierung richten. Da in dieser Phase noch keine Maschineninstruktionen den IR-Statements zugeordnet sind, können unterschiedliche Energieverbräuche der Maschineninstruktionen nicht berücksichtigt werden. Bei einem angenommenen gleichen Energieverbrauch $E_{IR-Stmt}$ für jedes der n IR-

Statements mit ebenfalls identischer gleicher Ausführungszeit $t_{IR-Stmt}$ ist der Gesamtenergieverbrauch eines Programms E_{gesamt} proportional zur Ausführungszeit t_{gesamt}:

$$t_{gesamt} = n * t_{IR-Stmt} \Rightarrow n = \frac{t_{gesamt}}{t_{IR-Stmt}}$$

$$E_{gesamt} = n * E_{IR-Stmt} = \frac{t_{gesamt}}{t_{IR-Stmt}} * E_{IR-Stmt} = \frac{E_{IR-Stmt}}{t_{IR-Stmt}} * t_{gesamt} = const * t_{gesamt}$$

Somit wäre die Energieoptimierung auf IR-Ebene identisch zu einer Geschwindigkeitsoptimierung. Die Programmgröße ist bei der Energieoptimierung hier nur indirekt relevant, da das Holen von Instruktionen dem Energieverbrauch $E_{IR-Stmt}$ zugeordnet werden kann und die Programmgröße nicht explizit in die Berechnung eingeht.

Als Fazit kann festgestellt werden, dass bei einer Menge von Optimierungen auf dieser Ebene nicht präzise genug entschieden werden kann, ob eine Optimierung vorteilig oder nachteilig ist. Optimierungen dieser Klasse müssen zu einem späteren Compilierungs-Zeitpunkt durchgeführt werden. Hierfür benötigte Informationen müssen daher durch die folgenden Phasen mitgeführt werden. Exemplarisch wird später in Kapitel 2.4.6 die *Loop Invariant Code Motion* betrachtet, da an diesem Beispiel gezeigt werden kann, wie die erreichte Codequalität von der Wahl der Zwischendarstellung abhängt.

2.4.3 Instruktionsauswahl

In der Phase der Instruktionsauswahl werden die Konstrukte der IR durch entsprechende Maschineninstruktionen überdeckt. Ausgangsbasis ist typischerweise eine graphbasierte oder baumbasierte Darstellung der IR. Unterstützung für die Erzeugung der Code-Generatoren, die die Instruktionsauswahl durchführen, bieten hier automatische Generatoren von Code-Generatoren auf Basis der entsprechenden Grammatiken [Muc97].

Im Fall des entwickelten encc-Compilers wird das Tool *Olive* zur Suche einer optimalen Lösung durch *Tree Pattern Matching* verwendet. *Olive* wurde von S. Tjiang entwickelt und basiert auf den beiden Code Generator Generatoren *Twig* von S. Tjiang, A.V. Aho und M. Ganapathi und *iburg* [FHP92] von D. Hanson und C. Fraser. Das Prinzip basiert auf der *dynamischen Programmierung*. Dynamisches Programmieren ist eine Algorithmenentwurfsstrategie, die auf

- einer Zerlegung und Bearbeitung des gegebenen Problems in Teilproblemen,
- der Eintragung der Teilergebnisse in Tabellen sowie
- dem Zusammensetzen der lokalen Entscheidungen zu der letztendlichen globalen Entscheidung

beruht.

Wir betrachten als Beispiel den folgenden Ausdruck in der Programmiersprache C, für den eine Überdeckung gefunden werden soll:

$$a = (c + d) * 2 + e$$

Das Verfahren wird auf die vom Front-End für diese Anweisung erzeugte IR, die als Baum dargestellt wird, (siehe Abb. 2.8) angewandt. Der von Olive generierte Code-Generator arbeitet nach dem Prinzip, dass jeweils an den Blättern des Baumes begonnen wird, eine Überdeckung für einen Teilbaum zu suchen. Von da aus wird fortgesetzt und versucht, diesen Teilbaum durch einen anderen Baum zu ersetzen, der geringere Kosten verursacht. Dieser Ansatz führt letztendlich für den Gesamtbaum zu einer Überdeckung

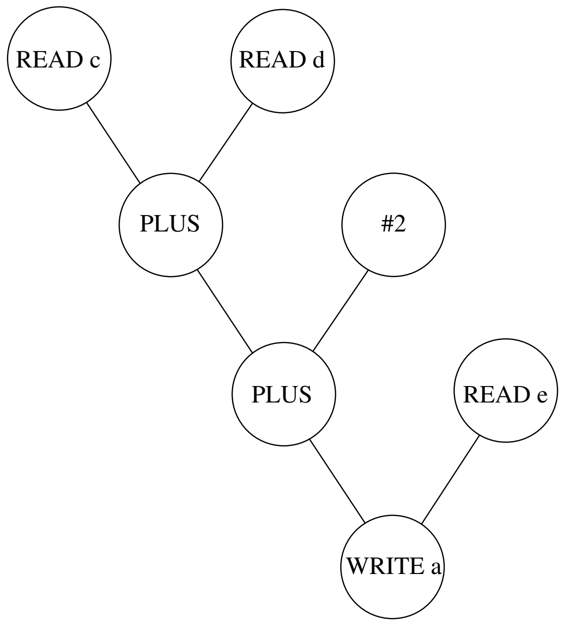


Abbildung 2.8: Baum zur Instruktionsauswahl

mit minimalen Kosten. Kosten sind in diesem Zusammenhang je nach dem gewählten Optimierungsziel entweder die Laufzeit, die Speichergröße oder der Energieverbrauch eines Befehls. Olive bewertet dabei alle möglichen Kombinationen, sofern nicht in einem Zwischenschritt absehbar ist, dass ein Zweig die optimale Lösung nicht mehr enthalten kann. Aufgrund der von Olive gefundenen optimalen Überdeckung werden dann die entsprechenden Maschineninstruktionen generiert. Im Gegensatz zu Tjiangs früherer Entwicklung *Twig,* bietet *Olive* eine umfangreichere Spezifikationssprache, ein schnelleres Pattern-Matching und eine allgemeinere Kostenfunktion [FHP92].

Für die Verwendung von *Olive* muss die Grammatik wie folgt spezifiziert werden:

$$label : pattern\,[\{cost\}]\,[= \{action\}]$$

Das Prinzip kann am Beispiel der Regel für einen ADD-Befehl betrachtet werden (Abb. 2.9).

Mit Hilfe dieser Regel kann *Olive* im Baum potenziell einen PLUS-Knoten überdecken, dessen Teilbäume und darüber liegender Weg durch ein Nonterminal *reg* dargestellt werden. In der ersten Phase wird die Überdeckung mit den geringsten Kosten gesucht und für die Berechnung jeweils der Kostenteil der Regel ausgeführt. In dem Beispiel sind die Kosten für diese Stufe im Baum, die Summe der Kosten für die beiden Teilbäume *cost[2]* und *cost[3]* zur Generierung der beiden *reg*-Parameter und zusätzlich Kosten von ″1″ für den Befehl ADD, der in dieser Regel generiert wird.

Wenn die Überdeckung mit den geringsten Gesamtkosten gefunden wurde, wird auf Basis der ausgewählten Regeln für die gefundene Überdeckung der *action*-Teil der gewählten Regeln ausgeführt. In dem *action*-Teil der Regel wird weiter in die darüber liegenden Zweige mit weiteren *action*-Aufrufen verzweigt und anschließend nach der Generierung der Befehle für die Berechnung der beiden *reg*-Parameter die Maschineninstruktion ADD generiert.

Wenn auch die gefundene Lösung optimal ist, so gilt dies nur für die Überdeckung eines einzelnen Baumes. Die Aufteilung der IR in Bäume und die Suche der optimalen Überdeckungen für diese Bäume führt nicht zwangsläufig zu einer optimalen Lösung über die Gesamtheit der Bäume. Daher kann für das gesamte Programm keine optimale Lösung garantiert werden.

```
/* ------------------------- pattern ------------------------- */
reg:  PLUS (reg, reg)
{
/* --------------------------- cost -------------------------- */
 /* the resulting cost is the sum of the costs for generating the */
 /* two parameter registers plus 1 for this add instruction */
  $cost[0] = $cost[2] + $cost[3] + 1;
} = {
/* ------------------------- action -------------------------- */
  int reg0, reg1, reg2;

 /* a new register for the result is created */
  reg0 = NewRegister();
  /* call for generating first parameter reg */
  reg1 = $action[2]();
  /* call for generating second parameter reg */
  reg2 = $action[3]();
  /* create the add instruction */
  GenInstr(ADD,reg0,reg1,reg2);
  /* return the resulting register to the next level */
  return(reg0);
}
```

Abbildung 2.9: Olive Spezifikation eines ADD-Befehls

Während dieser Mechanismus mit Code-Generator-Generatoren schon häufig für Performance- oder Programmgrößen-Optimierungen verwendet wurde, muss die Kostenfunktion für den Energieverbrauch neu entwickelt werden. Damit ergeben sich grundsätzlich Kostenfunktionen für vier unterschiedliche Optimierungsziele möglich:

- Ausführungszeit
  Für die Geschwindigkeit wird als Kostenfunktion die Anzahl der benötigten Taktzyklen gewählt. Die Kostenfunktion kann, wie in Abbildung 2.9 dargestellt, verwendet werden. Die Kosten von "1" entsprechen dann einem Taktzyklus für diesen Befehl. Für einen Baum werden die Kosten $cost_{performance}$ durch Summierung der Kosten der linken und rechten Teilbäume $cycles_{left}$ und $cycles_{right}$ und der Kosten des Knotens selbst $cycles_{this}$ berechnet:

  $$cost_{performance} = \sum cycles = cycles_{left} + cycles_{right} + cycles_{this}$$

  Der Vergleich zwischen zwei möglichen Lösungen (= Teilbäume $cycles_{left}$ und $cycles_{right}$) kann durch den Vergleich der Zyklen mithilfe der zentralen Vergleichsoperation $COSTLESS$ erfolgen:

  $$COSTLESS(solution_1, solution_2) = cycles_1 < cycles_2$$

- Programmgröße
  Für die Bestimmung der Programmgröße muss der benötigte Platz aller Instruktionen aufsummiert werden. Die Kosten "1" in der Kostenfunktion müssen dann durch die Anzahl der benötigten Bytes ersetzt werden:

  $$cost_{size} = \sum size = size_{left} + size_{right} + size_{this}$$

  Der Vergleich zweier möglicher Lösungen kann auf Basis der Größen erfolgen:

  $$COSTLESS(solution_1, solution_2) = size_1 < size_2$$

- Energie
  Für die Energie kann man vereinfacht davon ausgehen, dass jeder einzelne Befehl einen unterschiedlichen aber konstanten Energiebedarf hat. Für die einzelnen Befehle des Prozessors müssen diese jeweils unterschiedlichen Energiewerte als Datenbasis vorliegen. Für die Berechnung der Kosten einer Regel werden dann die Kosten der Teilbäume aufsummiert und die Energiekosten der in dieser Regel generierten Instruktionen aufaddiert:

  $$cost_{energy} = \sum energy = energy_{left} + energy_{right} + energy_{this}$$

  Der Vergleich zweier möglicher Lösungen kann auf Basis der Energie erfolgen:

  $$COSTLESS(solution_1, solution_2) = energy_1 < energy_2$$

- Leistung
  Die Leistung entspricht der Energie pro Zeit. Für dieses Optimierungsziel reicht die einfache Summenbildung über Energie oder Zyklen nicht aus, da man die Summe der bis zu diesem Baumknoten aufsummierten Energie $energy_{sum}$ und die Summe der Zyklen $cycles_{sum}$ berechnen und weiter propagieren muss, um auf Basis des Verhältnisses über die minimalen Kosten entscheiden zu können. Es muss daher zuerst für die Zusammenführung zweier Teilbäume T_{left} und T_{right} sowohl die Summe der Energie als auch die Summe der Zyklen berechnet werden:

  $$energy_{sum} = energy_{left} + energy_{right} + energy_{this}$$

  $$cycles_{sum} = cycles_{left} + cycles_{right} + cycles_{this}$$

  Der Vergleich zwischen zwei möglichen alternativen Lösungen für gültige Codesequenzen kann auf dieser Basis wie folgt ausgeführt werden:

  $$COSTLESS(solution_1, solution_2) = \frac{energy_1}{cycles_1} < \frac{energy_2}{cycles_2}$$

Um einen Compiler mit diesen vier Optimierungsmöglichkeiten zu realisieren, muss daher in allen Regeln die Summe der Taktzyklen, der Programmgröße und der Energie im *cost*-Teil mitgeführt werden. In der Vergleichsfunktion von *Olive* muss dann in Abhängigkeit der gewählten Optimierungsmöglichkeiten der entsprechende Vergleich ausgeführt werden.

(a)	(b)	(c)
`(1) v0 = [SP+#_c]`	`v0 = [SP+#_c]`	`v0 = SP + #_c`
`(2) v1 = [SP+#_d]`	`v1 = [SP+#_d]`	`v1-v3 = [v0++]`
`(3) v2 = [SP+#_e]`	`v2 = [SP+#_e]`	
`(4) v3 = v0 + v1`	`v3 = v0 + v1`	`v4 = v1 + v2`
`(5) v4 = v3 + v3`	`v3 = v3 * #2`	`v5 = v4 << #1`
`(6) v5 = v4 + v2`	`v4 = v3 + v2`	`v6 = v5 + v3`
`(7) [SP+#_a] = v5`	`[SP+#_a] = v4`	`[SP+#_a] = v6`

Abbildung 2.10: alternative Instruktionssequenzen

Für den ARM7-Befehlssatz können für das Beispiel aus Abbildung 2.8 die unterschiedlichen Instruktionssequenzen generiert werden, die in Abbildung 2.10 dargestellt sind. In Sequenz (a) wird die Multiplikation mit 2 durch eine Addition in Zeile 5 umgesetzt. Alternativ geschieht dies in Sequenz (b) durch einen Multiplikationsbefehl und in Sequenz (c) als weitere Alternative durch den Einsatz des Shifters. Außerdem wird

lattice Benchmark	Performance (Zyklen)	Energieverbrauch (μJ)
Performance-Optimierung	92.303	4.276,545
Energie-Optimierung	92.422	4.270,421

Tabelle 2.1: Performance vs. Energieoptimierung in der Instruktionsauswahl

in Sequenz (c) eine Alternative verwendet, in der die drei Ladebefehle für die drei Register durch einen Mehrfachladebefehl in Zeile 2 ersetzt werden. Es hängt nun von der Kostenbetrachtung ab, für welche der Sequenzen sich der Code-Generator entscheidet.

Nachdem die Maschineninstruktionen auf diese Weise erzeugt wurden, ist die Phase der *Instruktionsauswahl* abgeschlossen. Es liegt nun eine Instruktionssequenz auf der LIR-Ebene vor.

Untersuchungen des Einflusses der Instruktionsauswahl auf den Energieverbrauch zeigen keine großen Unterschiede im Vergleich zur Performance-Optimierung. Für den Benchmark *lattice* wurden mit dem später vorgestellten energieoptimierenden Compiler *encc* die in Tabelle 2.1 präsentierten Ergebnisse generiert, die eine Verbesserung des Energieverbrauchs von lediglich 0,14% zeigen. Der Grund liegt an der geringen Anzahl von Befehlen des RISC-Prozessors, die nur wenige Alternativen bei der Auswahl der Instruktionen ermöglicht. Da die Energie neben dem Strom auch von der Zeit abhängt, liefert eine Performance-Optimierung daher eine gute Näherung. Die Unterschiede beschränkten sich auf zwei Fälle:

1. Befehl für die Multiplikation mit 2
   Die Multiplikation mit 2 ist mit mehreren Befehlen möglich: dem Multiplikationsbefehl, dem Schieben nach Links um eine Stelle oder durch eine Addition. Die Addition ist energiemäßig um 2,5% günstiger als der ansonsten bei der Performance-Optimierung verwendete Shifter-Befehl

2. Laden einer Konstanten
   Alternativ zum Laden aus dem Speicher kann in Abhängigkeit von der Konstanten diese auch generiert werden. Im aufgetretenen Fall wurde dadurch gegenüber der Performance-Optimierung mithilfe der Datengenerierung der Energieverbrauch um 1% gesenkt.

Zu berücksichtigen ist, dass es sich hierbei um Einzelfälle handelt, sodass insgesamt nur eine sehr geringe Senkung des Energieverbrauchs während der Instruktionsauswahl möglich ist.

Nach dieser Betrachtung der Resultate der Phase der Instruktionsauswahl müssen den Befehlen noch reale Register des Prozessors zugeordnet worden. Da RISC-Prozessoren wie der ARM7 mehrere Register besitzen, die im Befehlssatz gleichwertig verwendet werden können, werden in der Instruktionsauswahl virtuelle Register vergeben. Hierbei geht man zunächst davon aus, dass beliebig viele Register zur Verfügung stehen. Eventuelle Beschränkungen durch die reale Anzahl physikalischer Register werden dann durch die spätere Phase der Registerallokation behandelt. Bei Prozessoren mit inhomogenen Registersätzen sollte die *Registerallokation* möglichst mit der *Instruktionsauswahl* verknüpft (Phasenkopplung) werden. Hier reicht es jedoch aufgrund der homogenen Architektur aus, diese Phasen getrennt voneinander zu behandeln.

2.4.4 Instruktionsanordnung

Das Ziel der Instruktionsanordnung bei RISC-Prozessoren ist die Steigerung der Geschwindigkeit und die Verringerung des Energieverbrauchs. Ansatzpunkt hierfür ist im Wesentlichen die Reduzierung der Anzahl gleichzeitig verwendeter Register, sodass temporäres Auslagern von Zwischenwerten vermieden werden kann.

Für die Änderung der Ausführungsreihenfolge der Maschineninstruktionen werden Architektureigenschaften des Prozessors und Eigenschaften der Kontrollstruktur des Programms ausgenutzt. Wir beschränken uns hier auf die Optimierungen, die für den energieoptimierenden Compiler relevant sind. Für den ARM-Prozessor ist hier besonders wichtig, dass virtuelle Register in der Instruktionsauswahl Verwendung finden, denen später in der Phase Registerallokation physikalische Register zugeordnet werden. Wenn nicht ge-

```
(1) r0 = #10       {r0,r4,r3}
(2) r2 = r4 - #5   {r0,r2,r3}
(3) r1 = r2 + r3   {r0,r1}
(4) r1 = r1 * r0   {r1}
```
(a) vorher

```
(1) r2 = r4 - #5   {r2,r3}
(2) r1 = r2 + r3   {r1}
(3) r0 = #10       {r0,r1}
(4) r1 = r1 * r0   {r1}
```
(b) nachher

Abbildung 2.11: Scheduling von Instruktionen zur Reduzierung der benötigten Register

nügend physikalische Register zur Verfügung stehen, müssen Zwischenwerte vorübergehend im Speicher abgelegt werden. Durch ein geeignetes Scheduling der Instruktionen kann diese Notwendigkeit verringert werden. In Abbildung 2.11 wird dies an einem Beispiel erläutert. In Abbildung 2.11a wird die ursprüngliche Reihenfolge dargestellt. Jeweils hinter der Instruktion sind die gerade lebendigen Register aufgeführt. Bei der "Instruktion 3" werden die Register *r0*, *r2* und *r3* aktuell benötigt oder enthalten die Registerwerte, die in nachfolgenden Instruktionen benötigt werden. Wenn durch weitere Instruktionen vor und nach diesem Codesegment weiterer Bedarf an physikalischen Registern entsteht, kann durch Verschieben der "Instruktion 1" der Register-Bedarf in Abbildung 2.11b reduziert werden. Der Bedarf an Registern reduziert sich dann an diesem Add-Befehl auf die Register *r2* und *r3*. Insgesamt reichen somit in dieser Instruktionssequenz 2 Register anstatt der vorher benötigten 3 Registern aus.

Das Auslagern eines Zwischenwertes in den Speicher (Spilling) mit energieintensiven Store- und Load-Befehlen hat einen hohen Anteil am Gesamtenergieverbrauch eines Programms, da die hohe Zahl der benötigten Register insbesondere in den häufig durchlaufenen innersten Schleifen eines Programms (hot spots) auftritt.

Eine weitere Eigenschaft energieoptimierender Compiler ist, dass aufeinander folgende Befehle durch ihre unterschiedlichen Bitmuster ein Umschalten der Werte auf den Busleitungen verursachen, die zwar keinen Unterschied in der Programmausführung, sehr wohl aber beim Energieverbrauch verursachen. Die bisherige vereinfachte Sicht eines Energieverbrauchs pro Befehl ohne Betrachtung des vorherigen oder nachfolgenden Befehls berücksichtigt dies nicht. Im nachfolgenden Kapitel 3 mit der Präsentation eines Energiemodells wird gezeigt, wie dies berücksichtigt werden kann und später im Kapitel 6 für eine Optimierung verwendet.

2.4.5 Registerallokation

Die Aufgabe der Registerallokation ist die effiziente Zuordnung von physikalischen Registern zu den virtuellen Registern, die nach der Instruktionsauswahl noch vorhanden sind. In einigen Compilern ist die Registerallokation in die Instruktionsauswahl integriert, wodurch aber Optimierungen wie die beschriebene Reduzierung der benötigten Register durch Instruktionsanordnung nicht mehr in verschiedenen Phasen umgesetzt werden können.

Als Ansatz im *encc*-Compiler wurde die Registerallokation von Appel et al. [AG98] implementiert. Es wird ein Interferenzgraph (siehe Abb. 2.13) erzeugt, dessen Knoten den virtuellen Registern (im Beispiel die Register V0 bis V13) entsprechen und dessen Kanten solche Register verbinden, die gleichzeitig verwendet werden. Register werden für einen Zeitraum (= lifetime) *lebendig* genannt, wenn sie Inhalte enthalten, die später noch verwendet werden. Durch eine Graphfärbung des Interferenzgraphen wird versucht, eine Zuordnung der virtuellen zu den vorhandenen physikalischen Registern vorzunehmen. Ist dies nicht möglich, müssen durch Spilling Zwischenwerte vorübergehend in den Speicher ausgelagert werden. Die

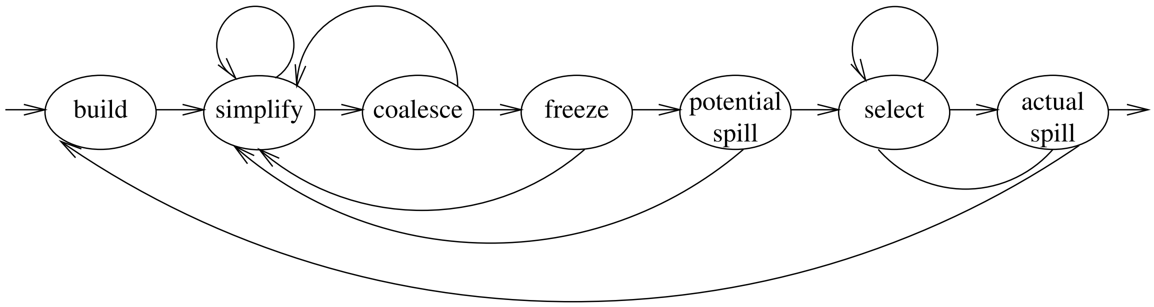


Abbildung 2.12: Graphfärbung mit Coalescing

optimale Registerallokation ist ebenso wie die Graphfärbung ein NP-vollständiges Problem. Der verwendete Algorithmus ist eine lineare Approximation, die gute Resultate liefert. Er besteht aus den folgenden Phasen (siehe Abb. 2.12):

- Build
  Zu Beginn wird ein Interferenzgraph generiert, dessen Knoten den virtuellen Registern entsprechen. Mit einer Datenflussanalyse wird festgestellt, welche Register an einem Punkt des Programms gleichzeitig verwendet werden und dafür eine Kante zwischen diesen virtuellen Registern im Graphen eingefügt. Kanten aufgrund von Move-Instruktionen werden gesondert markiert, da diese nur ein Umkopieren zwischen Registern bewirken und dieses evtl. durch Zusammenlegung (*Coalescing*) gelöst werden kann.

- Simplify
  Der Graph wird mit einer einfachen Heuristik eingefärbt. Alle Knoten, die einen kleineren Grad als die Zahl der Register haben, werden auf einem Stack gespeichert und aus dem Graphen entfernt, da für diese Knoten auf jeden Fall Register gefunden werden können. Dieser Vorgang wird solange wiederholt, wie in einem Durchlauf durch die noch vorhandenen Knoten mindestens ein weiterer Knoten diese Bedingung erfüllt.

- Coalesce
  Wenn eine Kante zwischen zwei Knoten (Registern) nur aufgrund einer Move-Instruktion existiert (in Abb. 2.13 als gestrichelte Verbindung dargestellt), können die beiden Knoten zusammengelegt werden und die Move-Instruktion gelöscht werden. Nach einer solchen Modifikation wird zur Phase *Simplify* zurückgesprungen.

- Freeze
  Um noch vorhandene Knoten weiter behandeln zu können, wird ein Knoten, der eine Move-Kante hat und möglichst wenig Kanten besitzt, ausgewählt und diese Kante zu einer Nicht-Move-Kante transformiert. Dies bedeutet, dass die zugehörige Move-Instruktion endgültig im Programm verbleibt und durch diese Nicht-Move-Kante neue Möglichkeiten durch Rücksprung zur Phase *Simplify* geprüft werden können.

- Potential Spill
  Wenn keine Knoten mit weniger Kanten als der Zahl der Register mehr vorhanden sind, muss eventuell ein Spilling eingebaut werden. Der Knoten, der die geringsten Kosten durch sein Spilling verursacht (s.u.), wird ausgewählt und auf einen Stack abgelegt.

- Select
  Der Reihe nach werden nun die Knoten wieder vom Stack geholt und Farben (physikalische Register) zugewiesen.

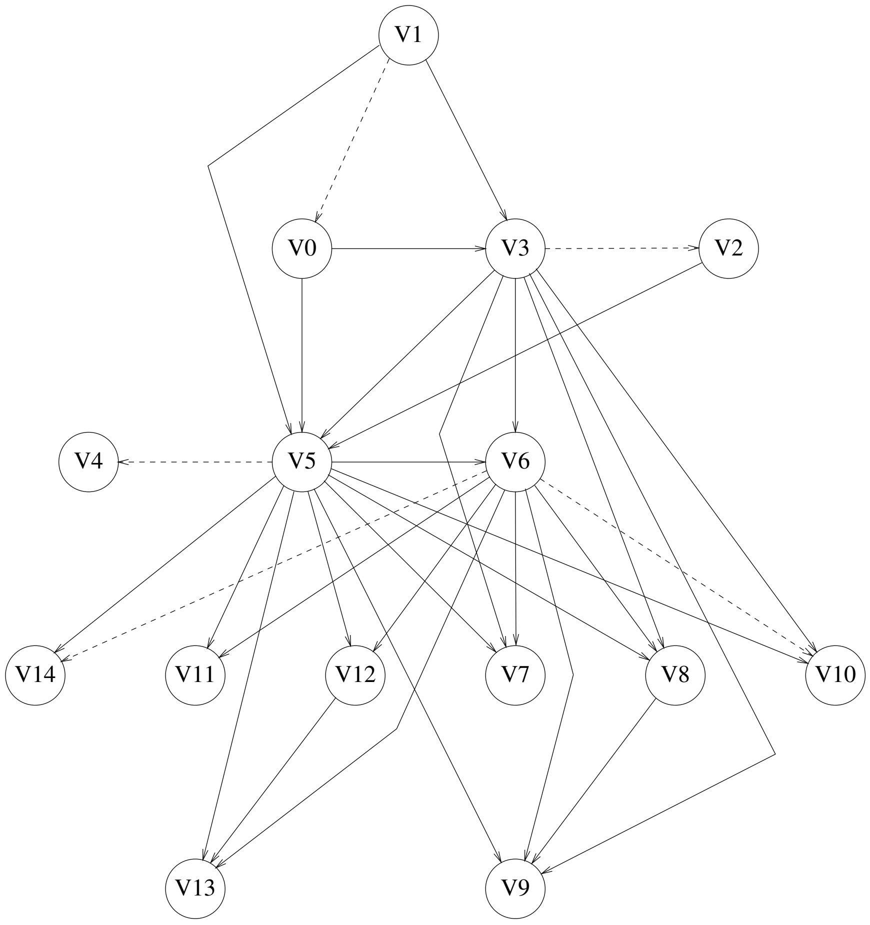


Abbildung 2.13: Interferenzgraph

- Actual Spill
  Wenn auch jetzt tatsächlich ein Spilling notwendig ist, werden entsprechende Instruktionen eingefügt. Das Spilling wird nur für den Teil des Programms eingefügt, wo dies aufgrund der zu geringen Zahl verfügbarer Register auch notwendig ist. Anschließend wird die Ausführung des Algorithmus in der Phase *Build* fortgeführt.

Dieser beschriebene Algorithmus ist fast unabhängig von der Optimierungsstrategie des Compilers. Relevant ist dies lediglich bei der Wahl des zu spillenden Registers. Hier gilt es, die genauen Energiekosten zu ermitteln, die durch die Einfügung von Load- und Store-Instruktionen in das generierte Programm entstehen. Da es aber nur um die Wahl des zu spillenden Registers geht und auf jeden Fall zusätzliche Energiekosten entstehen, unterscheiden sich die Geschwindigkeits- und Energieoptimierung nicht. Sehr wohl gibt es Unterschiede zur Programmgrößenoptimierung, da Load- und Store-Instruktionen in unterschiedlichen Basisblöcken unterschiedlich häufig aufgerufen werden. Bei einer reinen Code-Größenoptimierung werden Load- und Store-Instruktionen eher an wenigen Stellen, dafür aber möglicherweise in häufig ausgeführten Basisblöcken, eingefügt.

2.4.6 Optimierungen

Nach der Beschreibung des allgemeinen Aufbaus eines energieoptimierenden Compilers fehlen als Letztes noch die diversen Optimierungen, die entscheidend für die Codequalität sind. Ein Compiler sollte aufgrund seiner Größe möglichst modular aufgebaut werden, um die Wartung und Pflege zu erleichtern. Optimierungen arbeiten daher typischerweise auf einer der vorhandenen Intermediate Representations: HIR, MIR oder LIR (siehe Abb. 2.2). Für die richtige Wahl der Ebene sind einige Kriterien zu berücksichtigen:

- Eingangsinformationen für die jeweilige Optimierung
  Grundsätzlich entstehen beim Durchlauf durch die einzelnen Phasen des Compilers immer mehr Informationen. Wenn diese Informationen mitgeführt werden, sind diese folglich auf der LIR am Vollständigsten. Beispielsweise liegen erst dann Speicheradressen von Variablen vor. Eine Optimierung, die diese Speicheradressen benötigt, kann daher erst auf der LIR angewandt werden.

- Portabilität
  Wenn eine Optimierung auf einer höheren Ebene implementiert wird, ist sie prozessorunabhängig und kann für andere Prozessoren ohne Modifikation ebenfalls eingesetzt werden.

- Wiederholte Ausführung von Phasen
  Je später eine Optimierung ausgeführt wird, umso höher ist die Wahrscheinlichkeit, dass frühere Phasen wiederholt ausgeführt werden müssen. Beispielsweise kann eine Optimierung nach der Registerallokation es notwendig machen, dass diese nochmals ausgeführt werden muss.

```
ohne licm:

for (i = 0; i < 10; i++) {
  for (j = 0; j < 10; j++) {
    a[i][j] = i * 50;
  }
}
```

```
mit licm:

for (i = 0; i < 10; i++) {
  tmp = i * 50;
  for (j = 0; j < 10; j++) {
    a[i][j] = tmp;
  }
}
```

Abbildung 2.14: Loop Invariant Code Motion

Es ist also abzuwägen zwischen den notwendigen Eingangsinformationen, der gewünschten Portabilität, der Optimierungsgüte und der wiederholten Ausführung von Phasen, um zu entscheiden, auf welcher Ebene eine Optimierung ansetzt. Als Beispiel soll hier die *Loop Invariant Code Motion* [Muc97] betrachtet werden. Diese kann Befehle, die vom Schleifenkörper unabhängig sind, bereits vor dem Schleifeneintritt ausführen, sodass diese Befehle nur einmalig und nicht bei jedem Schleifendurchlauf ausgeführt werden müssen (siehe Abb. 2.14). Der Nachteil ist jedoch, dass das Ergebnis dieser vorgezogenen Berechnung während der vollständigen Schleifenausführung in einem Register gehalten werden muss. Wenn kein Register mehr frei ist, führt dies zu einem Spill. Ob noch ein Register verfügbar ist, kann aber frühestens nach der Instruktionsauswahl entschieden werden, wenn also zumindest anhand der aktuell benötigten virtuellen Register entschieden werden kann, ob evtl. noch ein Register frei bleibt. Um die Konsequenz für die Codequalität aufzuzeigen, ist in Tabelle 2.2 ein Vergleich der Optimierung auf MIR-Ebene gegenüber LIR-Ebene dargestellt. In der zweiten Spalte ist der Energieverbrauch ohne ausgeführte licm-Optimierung aufgezeigt, in den folgenden Spalten dann die Änderung bei Ausführung der Optimierung auf MIR- bzw. auf LIR-Ebene. Es ist deutlich zu sehen, dass auf der höheren Ebene teilweise starke Verschlechterungen auftreten. Die Ursache liegt darin, dass die Auswirkungen auf die später generierten Maschineninstruktionen nicht detailliert abgeschätzt werden können und dadurch teilweise ein Spilling eingefügt werden muss. Der Vollständigkeit halber ist in der letzten Spalte auch dargestellt, wie hoch die Verbesserung ist, wenn die *licm* auf MIR und auf LIR-Ebene durchgeführt wird. Diese zweimalige Ausführung auf unterschiedlichen Ebenen hat allerdings keine weiteren Vorteile.

Energy (in μJ)	ohne	MIR	LIR	MIR & LIR
biquad_N_sections	26,582	0%	-6,4%	-6,4%
bubble_sort	4.835,783	0%	0%	0%
heap_sort	1.406,755	+9,0%	0%	+9,0%
insertion_sort	2.113,717	-0,7%	0%	-0,7%
lattice	2.867,106	+89,8%	-2,8%	+87,2%
matrix-mult	82,282	-12,1%	0%	-12,1%
me_ivlin	14.324,786	+0,1%	0%	+0,1%
quick_sort	214,302	+1,1%	0%	+1,1%
ref_idct	649.739,375	+1,9%	0%	+1,9%
selection_sort	3.132,978	-0,1%	0%	-0,1%

Tabelle 2.2: licm-Optimierung auf unterschiedlichen IR-Ebenen

Andererseits ist es für andere Optimierungen, wie beispielsweise Schleifenoptimierungen, sinnvoll, diese auf einer höheren Ebene durchzuführen, da die Informationen über den Aufbau von Schleifen häufig im Compiler nicht weitergereicht werden und in der benötigten Form auch schon auf der HIR vorliegen.

Nach der Betrachtung des Compileraufbaus einschließlich der Optimierungen wird im folgenden Unterkapitel die Umgebung des Compilers beschrieben.

2.4.7 Einbindung in Entwicklungsumgebung

Um eine vollständige Umgebung zur Softwaregenerierung zu erhalten, wurden weitere Komponenten aus dem Standard-ARM Software Development Toolkit Version 2.5 ergänzt (siehe Abb. 2.15). Der in diesem Paket enthaltene Compiler konnte nicht verwendet werden, da er nicht in Source-Form erhältlich war. Der *encc*-Compiler wird nun in eine vollständige Toolkette mit folgenden zusätzlichen Komponenten integriert:

- Assembler

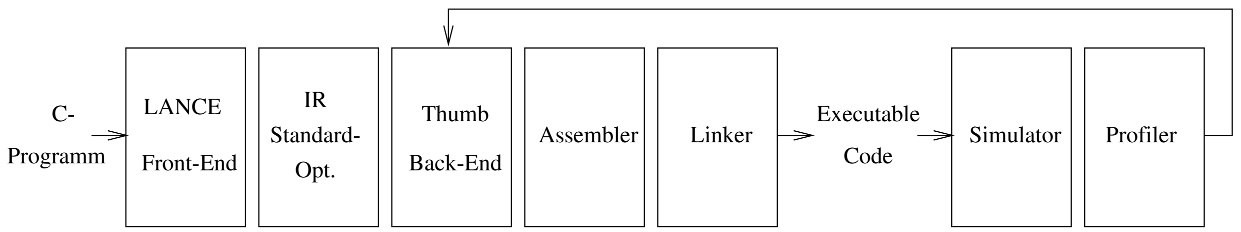


Abbildung 2.15: Toolkette mit Rücksprung für Ausführungsanalyse

Als Assembler wurde der Standard-Thumb-Assembler aus dem ARM SDT V2.50-Paket verwendet. Als Eingabe werden eine oder mehrere Assemblerdateien erwartet, in denen durch IMPORT und EXPORT-Befehle Adressbezüge hergestellt werden können. Reale Adressen werden dann im Linker zugeordnet. Als Ausgabe wird der Objectcode erzeugt. Für jeden Speicherbereich, also beispielsweise für Hauptspeicher und Onchip-Speicher, werden getrennte Dateien assembliert.

- Linker

  Der Linker stellt Bezüge zwischen den einzelnen Objectcodes her. Verwendet wurde der Linker aus dem ARM SDT V2.50 Paket. Neben den Objectcodes wird in einer Konfigurationsdatei spezifiziert, ab welcher Adresse die jeweiligen Objektmodule angeordnet werden sollen. Unterschieden werden hier die Adressbereiche für den Hauptspeicher und den Onchip-Speicher. Als Ausgabe wird vom Linker eine auf dem ARM7 ausführbare Datei erzeugt. Beim erstmaligen Durchlauf des Linkers in der Toolkette werden außerdem die Namen aller Basisblöcke mit den jeweils zugeordneten realen Adressen ausgegeben, damit der Profiler später durch Analyse der Adressen die genaue Ausführungshäufigkeit von Basisblöcken bestimmen kann.

- Simulator

  Zur Simulation des Prozessors wird der Standard-ARM-Simulator (ARMulator) verwendet. Er simuliert die vom Linker generierte Datei mit ausführbaren ARM7-Befehlen und verwendet zusätzlich die voreingestellte Konfiguration. Diese umfasst beispielsweise die Speicherwortbreiten und Wartezyklen sowie optional die Wahl eines Caches mit Größe, Assoziativität und Strategie. Vor der Ausführung der Simulation wird das Laden des Programms ausgeführt. Als Ausgabe wird ein Protokoll aller ausgeführten Befehle und ihrer Adressen generiert.

- enProfiler

  Da der verwendete Standard-ARM-Simulator nur die Anzahl der ausgeführten Befehle und der benötigten Prozessorzyklen ermittelt, reicht diese Information für Energieoptimierungen nicht aus. Um den Energieverbrauch zu berechnen, müssen die ausgeführten Instruktionen ausgewertet werden. Die Details sind abhängig von dem gewählten Energiemodell, welches im nächsten Kapitel vorgestellt wird.

  Als Ergebnis liefert der Profiler folgende Informationen:

  – Gesamtenergieverbrauch des Programms,
  – Ausführungshäufigkeit aller Basisblöcke,
  – Anzahl benötigter Prozessorzyklen,
  – Speicherbedarf im Hauptspeicher, im Onchip-Speicher und Stackgröße.

Da für einige Optimierungen die Anzahl der Durchläufe von Basisblöcken relevant ist, muss nach dem Durchlauf durch den Profiler, der die exakte Anzahl der Ausführungen eines Basisblockes bestimmt, in das Back-End zurückgesprungen werden. Insgesamt ergibt sich für die folgenden Untersuchungen die in Abbildung 2.15 dargestellte Abfolge von Tools.

Kapitel 3

Energiemodell

Um den Programmcode durch den Compiler generieren und optimieren zu können, müssen die Hardware-Eigenschaften des Systems, für das die Software generiert werden soll, bekannt sein. Zur Darstellung dieser Hardwareeigenschaften wird das System modelliert, d. h. die relevanten Eigenschaften mit notwendiger Genauigkeit z. B. durch mathematische Gleichungen nachgebildet. Für die Optimierungen bzgl. Programmgröße ist dies relativ einfach. Bei der Programmgröße sind die Instruktionswortbreite und die Anzahl der Instruktionen zu berücksichtigen. Bei fester Instruktionswortbreite, wie bei RISC-Prozessoren typisch, kann die Programmgröße aus dem Produkt der Instruktionswortbreite und der Anzahl der Befehle einfach bestimmt werden.

Bei einer Optimierung nach Geschwindigkeit (Performance) wird die Anzahl der Taktzyklen bei der Ausführung jedes einzelnen Befehls als Basis verwendet. Allerdings ist dies nur bedingt ausreichend. Wenn Speicherzugriffe zu Wartezyklen führen, muss die Wartezeit (=*Latenzzeit*) mit berücksichtigt werden. Bei dem Holen einer Instruktion (engl. Instructionfetch) ist dies bei konstanter Instruktionswortbreite ein konstanter Wert. Wenn im Compilierprozess die Entscheidung zwischen alternativen Instruktionssequenzen mit einer unterschiedlichen Anzahl von Befehlen oder unterschiedlichen Speicherzugriffen zu treffen ist, entsteht dadurch eine Differenz, die berücksichtigt werden muss. Die Speicher-Latenzzeit wird in den meisten Compilern nicht berücksichtigt, da Speicherzugriffszeiten nicht zu den Eingabeparametern gehören. Teilweise wird dieses Problem dem Benutzer durch eine geeignete Wahl von Parametern beim Aufruf des Compilers überlassen, z. B. bei der Generierung von Konstanten. An diesem Beispiel soll die Komplexität gezeigt werden, die durch den Benutzer kaum optimal berücksichtigt werden kann.

```
    r0 = [Lbl]; (3Z+1W+3W)
    ..
Lbl 00000412
                ==========
                  3Z+4W
```

(a) Laden eines Datenwortes

```
r0 = 50;        // 50    (1Z+1W)
r0 = r0 << 3;   // * 8   (1Z+1W)
r0 = r0 + 12;   // + 12  (1Z+1W)
                        ========
                         3Z+3W
```

(b) Konstantengenerierung

Abbildung 3.1: alternative Codesequenzen für große Konstanten

Der Thumb-Compiler des ARM SDT 2.50 entscheidet bei großen Konstanten, ob er diese im Programmspeicher ablegt und dann hieraus als 32-Bit-Datenwort lädt (Abb. 3.1a) oder alternativ durch mehrere arithmetische Befehle aus kleineren Konstanten (Abb. 3.1b), die im Instruktionswort codiert werden können, generiert. Der Benutzer des Compilers kann für einen Compilerlauf über einen Kommandozeilenparameter konfigurieren, wie viele Befehle als Alternative zum Laden der Konstanten aus dem Speicher akzeptabel sind.

In dem betrachteten Beispiel wird beim Laden der großen Konstante "412" ein 32-Bit-Datenzugriff (siehe Abb. 3.1a) mit einem Lade-Befehl in drei Zyklen (Z) ausgeführt. Alternativ kann das gleiche Datenwort mit drei Befehlen (Abb. 3.1b) generiert werden, die ebenfalls drei Zyklen benötigen. Unterschiede ergeben sich bei der Geschwindigkeit erst durch die Einbeziehung der Wartezyklen (W). Das betrachtete System benötigt für einen 16-Bit-Speicherzugriff einen Wartezyklus und für einen 32-Bit-Speicherzugriff drei Wartezyklen. Somit müssen für einen Ladebefehl für den Instructionfetch ein Wartzyklus zuzüglich drei Wartezyklen für das Laden des Datums aus dem Speicher berechnet werden. Bei der alternativen Generierung der Konstanten in drei Befehlen werden nur 3 * 1 Wartezyklus für die Instructionfetches benötigt. In dieser Systemkonfiguration und bei dieser Konstanten ist die Konstantengenerierung daher um einen Zyklus günstiger.

Während schon bei der Geschwindigkeit die Grenzen des Modells offensichtlich werden, gilt dies in noch stärkerem Maße für die Optimierung nach Energie. Da die Energie E einer Funktion der Zeit t, der Spannung U und dem Strom I entspricht,

$$E = \int_0^T P(t)dt = \int_0^T U(t) * I(t)dt$$

ist das zugrunde liegende Modell komplexer als jenes für die Geschwindigkeitsoptimierung. Die Optimierungsfunktion für Energie basiert auf der Zeit t wie bei der Geschwindigkeitsmodellierung, beinhaltet aber zusätzlich den Strom I und die Spannung U. Die Spannung U hängt von den Powermanagement-Eigenschaften des Prozessors ab. Frühere Prozessorentwürfe waren beschränkt auf eine konstante Eingangsspannung U. In einem weiteren Schritt gab es die Möglichkeit, den Prozessor in einen Standby-Modus zu versetzen (z. B. beim ARM7), wobei der vollständige Prozessor oder einzelne Funktionseinheiten abgeschaltet wurden. In den neuesten Designs wie z. B. dem ARM-basierten XScale-Prozessor [Int00] von Intel wird die Möglichkeit geboten, die Spannung und den Takt schrittweise zu reduzieren. Bei der verwendeten CMOS-Technologie wird der Strom I durch das Auf- bzw. Entladen der Kapazitäten C und die Spannung U bestimmt. Der Energieverbrauch hängt somit insgesamt in erster Näherung vom Quadrat der Spannung U ab [Yea98]. Wir betrachten nachfolgend beispielhaft ein Programm, welches zur Abarbeitung genau n Taktzyklen benötigt. Die Enerige E kann dann wie folgt berechnet werden:

$$E = n * C * U^2$$

Wenn nicht die volle Rechenleistung des Prozessors über einen gewissen Zeitraum benötigt wird, kann zuerst die Taktfrequenz und danach im gleichen Verhältnis die Spannung U reduziert werden. Die Laufzeit des Programms vergrößert sich dadurch in dem Verhältnis, in dem der Prozessortakt reduziert wurde. Da die Spannung U in die Berechnung des Energieverbrauchs quadratisch eingeht, kann durch die Verlangsamung der Programmausführung und die gleichzeitige Reduzierung der Versorgungsspannung eine Energieeinsparung erreicht werden. Die Grenze für die Reduzierung der Frequenz ist im Wesentlichen durch die benötigte Rechenleistung vorgegeben.

Da die Umschaltung der Spannung und des Taktes einige Prozessorzyklen benötigt, ist diese Methode nur für eine höhere Anzahl von Instruktionen vorteilhaft. In dem Beispiel in Abb. 3.2 wird zum Zeitpunkt *t1* begonnen, die Frequenz zu reduzieren. Dieser Vorgang ist zum Zeitpunkt *t2* abgeschlossen, sodass danach die Spannung reduziert werden kann, was zum Zeitpunkt *t3* abgeschlossen ist. Wenn wieder eine höhere Rechenleistung benötigt wird, muss umgekehrt zuerst zum Zeitpunkt *t4* die Spannung erhöht werden und danach ab Zeitpunkt *t5* die Frequenz. Insgesamt wird in diesem Beispiel die Rechenleistung auf 74% reduziert, der Energieverbrauch jedoch auf 50%.

Die Aufgabe einer automatischen Berechnung der zur Verfügung stehenden Zeit bis zur nächsten harten Deadline der Applikation, der Kalkulation der benötigten Rechenleistung und der daraus resultierenden optimalen Prozessorspannung U und Taktfrequenz f kann nur schwer in einen Compiler integriert

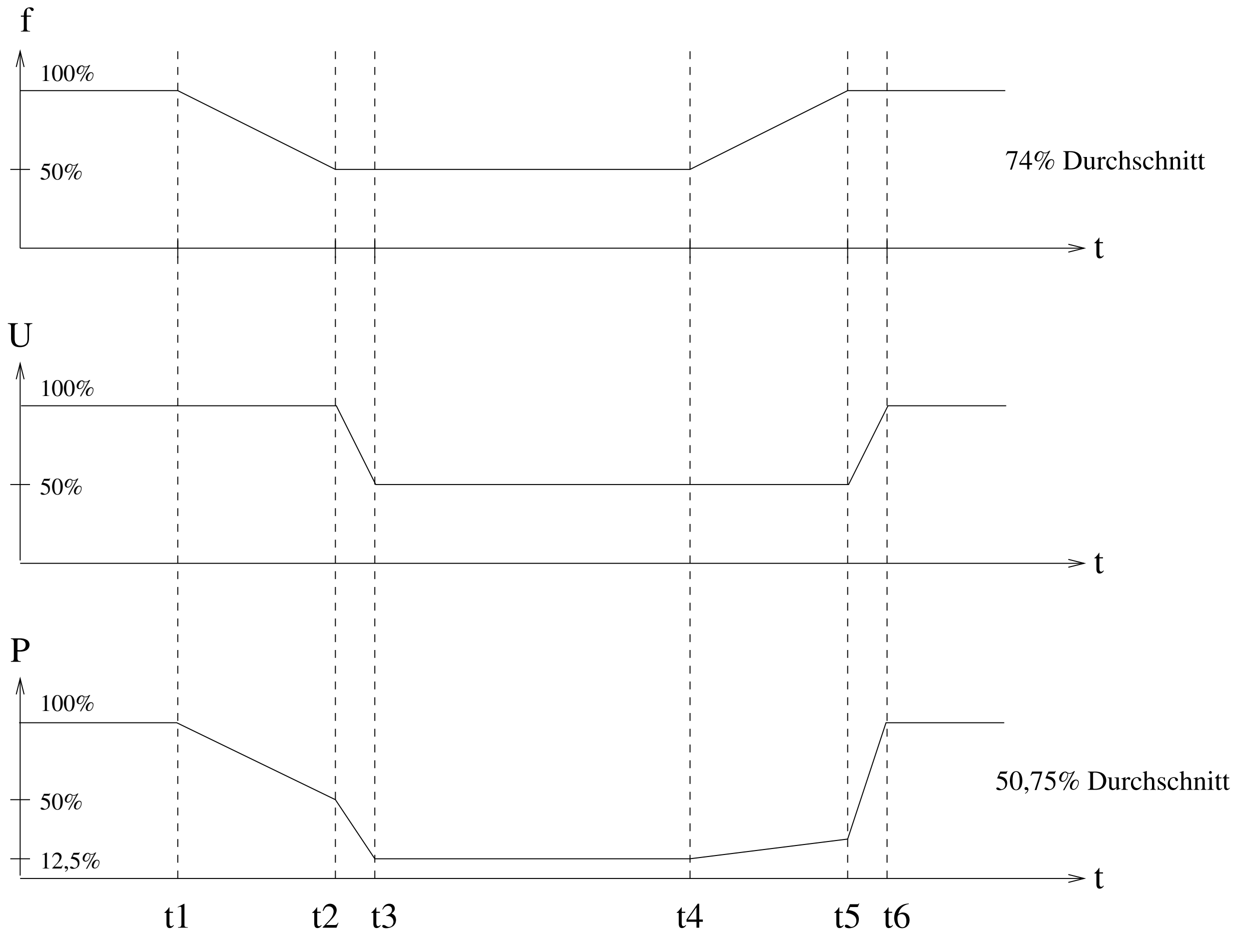


Abbildung 3.2: Reduktion der Frequenz f und der Spannung U

werden, da ein nennenswerter Rechenaufwand und zusätzlicher Zeitverlust für das Herunter- bzw. Hochschalten der Taktfrequenz und der Spannung entsteht. Daher kann diese Anpassung der Spannung und Frequenz nicht für einzelne Basisblöcke oder Funktionen erfolgen, sondern wird sinnvollerweise als Erweiterung des Betriebssystem realisiert [Bel00]. Ein Compiler kann dies nur durch die Berechnung der maximalen Ausführungszeit eines Programmteils oder einer Funktion unterstützen. Aus diesem Grund wird die Unterstützung dieser Eigenschaft neuerer Prozessoren hier nicht weiter betrachtet und im Folgenden von einer konstanten Spannung U ausgegangen. Die dargestellten Zusammenhänge zeigen aber, dass Geschwindigkeits- und Energieoptimierung nicht immer synchron verlaufen.

Der Strom I ist der letzte Parameter zur Berechnung der Energie E und gleichzeitig die komplexeste Komponente. Bisher wurde von einem konstanten Strom für alle Befehle ausgegangen. Bei einem aktiven Prozessor hängt der Strom aber u. a. von dem Ausgangszustand der internen Schaltung, den aktuell aktiven Funktionseinheiten, den verarbeiteten Daten, dem Stromverbrauch der Speicher bei Speicherzugriffen und weiteren Systemzuständen ab.

Es gilt nun, ein Modell für den Compiler zu entwerfen, das die Hardwareeigenschaften des Systems, die durch den Compiler beeinflusst werden können, berücksichtigt und andererseits den Energieverbrauch hinreichend genau ermittelt. Es muss das Prinzip gelten, dass die Wahl des Abstraktionslevels so hoch wie möglich und so niedrig wie nötig erfolgt, sodass der Energieverbrauch genau genug bestimmt und andererseits eine Bewertung möglichst schnell durchgeführt werden kann.

Im nachfolgenden Unterkapitel werden die Anforderungen an Energiemodelle, die für energieoptimierende Compiler relevant sind, beschrieben und diskutiert. Auf dieser Basis können dann existierende Energiemodelle untersucht und bewertet werden. Aufgrund der bestehenden Nachteile und Unzulänglichkeiten wird in den folgenden Unterkapiteln ein neues Energiemodell präsentiert. Die Grundlagen dieses Kapitels sind auch in Steinke et al. [SKWM01] beschrieben.

3.1 Anforderungen

Ein Energiemodell zur Verwendung in einem energieoptimierenden Compiler muss die folgenden zwei Bedingungen erfüllen:

1. Die Eigenschaften der Schaltung, die zu unterschiedlichem Energieverbrauch führen und die außerdem durch den Compiler beeinflussbar sind, müssen berücksichtigt werden. Dazu gehören beispielsweise Energiekosten von Bussen, da Buszugriffe aufgrund der vom Compiler ausgewählten Instruktionen ausgeführt werden. Andererseits ist es beispielsweise nicht notwendig, Spannungsschwankungen des Netzteils aufgrund von Laständerungen oder die Umgebungstemperatur zu modellieren. Der Energieverbrauch kann sich dadurch zwar signifikant ändern, allerdings können beide Informationen vom Compiler nicht beeinflusst werden.

2. Der Energieverbrauch ist genau genug zu berechnen. Das Kriterium hierfür ist, ob eine genauere Bestimmung zu anderen Entscheidungen und damit letztendlich zu einem anderen generierten Code führt.

Aus diesen Kriterien können nun folgende Eigenschaften ermittelt werden, die Berücksichtigung im Energiemodell finden sollen:

- Art von Maschineninstruktionen
  Der Compiler hat Einfluss auf die Auswahl der Instruktionen. Da jede Instruktion unterschiedlich viel Energie bei der Ausführung benötigt, wie später anhand der präsentierten Ergebnisse gezeigt

wird, sollte das Energiemodell die Instruktionen unterscheiden und ihren individuellen Beitrag zum Gesamtenergieverbrauch zuordnen. Es ist daher nahe liegend, ein Energiemodell auf Instruktionsebene zu verwenden. Höhere Betrachtungsebenen, die beispielsweise den unterschiedlichen Energieverbrauch von Befehlen nicht berücksichtigen, erfüllen die hier geforderte Eigenschaft nicht. Tiefere und detaillierte Ebenen sollten nur verwendet werden, wenn die oben geforderten Eigenschaften nicht auf der Instruktionsebene realisiert werden können.

- Scheduling von Instruktionen
  Bei unterschiedlicher Reihenfolge von Instruktionen werden auch die entsprechenden Funktionseinheiten eines Prozessors in unterschiedlicher Reihenfolge verwendet. Wenn eine Funktionseinheit nur in jedem zweiten Befehl angesprochen wird, ist durch das wechselweise Verwenden und Nichtverwenden der Funktionseinheit der Energieverbrauch höher, als wenn erst alle Befehle, die diese Funktionseinheit verwenden, und anschließend die Befehle ohne Verwendung dieser Funktionseinheit ausgeführt werden. Diese Eigenschaft kann zur Optimierung genutzt werden, wenn das dem Compiler zugrunde liegende Energiemodell diese Eigenschaft auch berücksichtigt. Falls dieses der Fall ist, kann der Compiler unter Berücksichtigung der Daten- und Kontrollflussabhängigkeiten die Ausführungsreihenfolge der Instruktionen optimieren.

- Speicherhierarchie
  In Systemen mit zwei oder mehr Speichern (z. B. Onchip- und Offchip-Speichern oder auch unterschiedlich großen Onchip-Speichern) kann der Compiler für Programmteile und Daten zwischen diesen Speichern auswählen. Voraussetzung ist, dass der Energieverbrauch jedes Speichers in das Energiemodell eingeht.

- Bitwechsel auf Bussen
  Für die Übertragung unterschiedlicher Adressen und Daten sind auf den Bussen Signalwechsel notwendig. Da Busse häufig lange Leitungen besitzen und große Kapazitäten treiben müssen, ist der Energieverbrauch beim Umschalten hoch. Durch unterschiedliche Maßnahmen kann versucht werden, die Anzahl der Bitwechsel zu reduzieren. Hierzu ist Voraussetzung, dass die Bitwechsel im Energiemodell Berücksichtigung finden.

Neben diesen Eigenschaften des betrachteten Systems selbst muss das Modell noch die folgenden allgemeinen Aspekte erfüllen:

- Bestimmung der Parameter des Energiemodells

  Die Parameter eines Energiemodells (= prozessorspezifische Konstanten des Modells) können grundsätzlich auf die folgenden beiden Arten bestimmt werden:

  1. Simulation
     Auf Basis einer Schaltungsbeschreibung können durch Simulation auf unterschiedlichen Hardware-Ebenen (siehe Abb. 3.3 aus [MPS98]) Parameter ermittelt werden. Für die Bestimmung des Energieverbrauchs reicht eine Simulation auf einer der höheren Ebenen nicht aus. Sie muss eher auf Gate-Level stattfinden und darüber hinaus auch Leitungslängen und -kapazitäten berücksichtigen. Das Problem dieses Ansatzes ist, dass für industriell hergestellte Prozessoren Schaltungsbeschreibungen vom Hersteller zum Schutz des eigenen Know-Hows nicht zur Verfügung gestellt werden. Beispielsweise sind für den ARM7TDMI nur wenige, nicht detaillierte Untersuchungsergebnisse [Seg97] frei verfügbar, die die Verteilung des Leistungsverbrauchs der wichtigsten Komponenten beschreiben (siehe Tab. 3.1).
     Ein interessanter Aspekt bei der Analyse der Daten ist festzustellen, ob durch einen Compiler überhaupt Einfluss auf den Energieverbrauch zu verzeichnen ist. Beispielsweise können

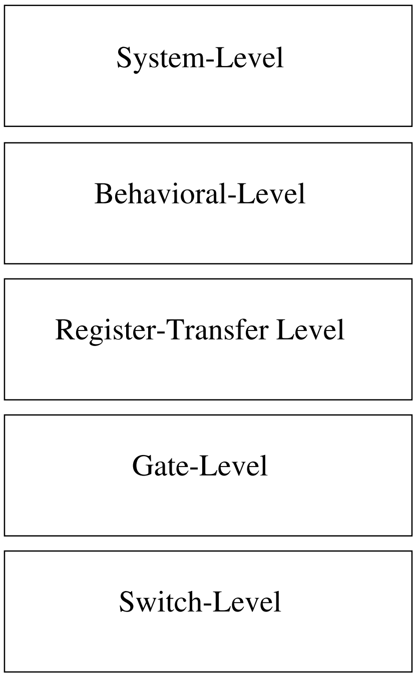


Abbildung 3.3: Simulationsebenen

Block	Leistungsverbrauch (%)
Data Path	
ALU	22,3
Register Bank	8,3
Shifter	4,6
Gesamt	58,7
Control	
Instruction Decoder	14,8
Clock Generator	6,7
Gesamt	41,3

Tabelle 3.1: Energieverbrauch der einzelnen Prozessorkomponenten des ARM7TDMI

die aufgeführten Anteile von ALU, Registern und Shifter durch die Wahl der Instruktionen beeinflusst werden. Weiterhin ist davon auszugehen, dass noch Anteile, die dem Control-Teil zugeordnet werden, ebenfalls abhängig von ausgeführten Instruktionen oder Speicheroperationen sind. Es kann bereits aus diesen Daten geschlossen werden, dass durch eine entsprechende Berücksichtigung des Energiemodells ein Anteil von mehr als 60% des Energieverbrauchs des Prozessors beeinflusst werden kann. Es ist allerdings auch ersichtlich, dass der Detaillierungsgrad dieser Daten nicht ausreicht, um Parameter eines Energiemodells in ausreichender Anzahl und Güte zu extrahieren. Dafür sind weitere und detailliertere Untersuchungen notwendig.
Ein weiterer Nachteil liegt in der Genauigkeit der Abschätzung durch Simulation. Mit verfügbaren Tools auf Gate-Level erreicht man im Vergleich zum Energieverhalten realer Schaltungen durchschnittliche Fehler in der Größenordnung von 10% und für den maximalen Fehler in Höhe von 20% [IY96]. Andere Untersuchungen [Seg97] zeigen Ungenauigkeiten von 6% auf Schaltungsebene.

2. Messung
Wenn keine Schaltungsbeschreibung vorliegt, kann mit einem real vorliegenden Prozessor durch Messung dessen Energieverhalten untersucht und die Parameter für das Energiemodell bestimmt werden. Zu berücksichtigen ist, dass auch hier nicht von detaillierten Kenntnissen der internen Schaltung auszugehen ist, sondern der Prozessor im Wesentlichen als Blackbox (= ohne Kenntnisse über das Innere des Bausteins) zu betrachten ist. Es muss daher zu einem Energiemodell ein Messverfahren entwickelt werden, welches durch geeignete Schritte diese Modellparameter bestimmen kann.

Bei der Entwicklung eines Energiemodells korrespondiert die Wahl der Parameter stark mit der Methode zur Bestimmung dieser Parameter. Nahe liegend ist es, gleichzeitig mit der Entwicklung eines Modells auch die geeignetere Methode festzulegen und diese bei der Wahl der Parameter zu berücksichtigen.

- Wiederverwendbarkeit des Modells

  Ein Modell soll für eine möglichst große Bandbreite von Systemen und Prozessoren gelten. Es soll daher so universell wie möglich sein, andererseits aber auch hinreichend präzise. Da der Schwerpunkt dieser Arbeit auf den RISC-Prozessoren liegt, sollen neben dem hier beispielhaft betrachteten ARM7-Prozessor auch Compiler für andere RISC-Architekturen mit diesem Energiemodell arbeiten können.

Alle Anforderungen, die in diesem Unterkapitel festgestellt wurden, können durch ein Energiemodell auf Instruktionsebene erfüllt werden. Es liegt weiterhin nahe, diese Ebene zu wählen, da sich auch der Compiler im Wesentlichen auf Instruktionen abstützt. Die Wahl einer tieferen Ebene als die der Instruktionen würde einen erhöhten Aufwand bei der Energieabschätzung zur Folge haben, ohne dass die Steigerung der Detaillierung Vorteile liefern würde. Es wird daher für diese Arbeiten die Instruktionsebene für das Energiemodell ausgewählt.

3.2 Energiemodelle auf Instruktionsebene

Eines der ersten Energiemodelle auf Instruktionsebene wurde von Tiwari et al. präsentiert [TMW94b, TMW96, TL98]. Es wurde ein Modell zusammen mit einer Messmethode vorgeschlagen und für unterschiedliche Prozessortypen RISC, CISC und DSP untersucht. Das Modell ist relativ einfach und basiert auf Basiskosten (*base cost)* und Interinstruktionskosten (*Interinstruktionskosten).* Die *Basiskosten* entsprechen den Energiekosten eines einzelnen Befehls. Die zusätzlichen Kosten durch den Wechsel von einem Befehl zu einem anderen Befehl werden durch die *Interinstruktionskosten* modelliert. Zusätzlich anfallende Effekte wie *pipeline stalls* und *cache misses* werden als Bestandteil der *Basiskosten* betrachtet.

Das Messverfahren ist durch die mehrfache Ausführung des zu messenden Befehls in einer Schleife realisiert. Dadurch wird erreicht, dass der zu messende Strom dauerhaft anliegt und leicht mithilfe eines Amperemeters gemessen werden kann. Die Anzahl desselben Befehls in dieser Schleife ist so hoch, dass der abweichende Energieverbrauch durch den Sprung auf den Messwert vernachlässigbar ist. Die Berücksichtigung anderer Systemkomponenten wie Speicher sowie Bitwechsel auf den Busleitungen werden in diesem Modell nicht berücksichtigt.

Auf der Basis des Modells von Tiwari et al. wurde von Sinevriotis et al. [SS99] eine Analyse des ARM7 durchgeführt. Als Ergebnis wurden zur Energiereduzierung verschiedene Ansätze (Geschwindigkeitsoptimierung, Scheduling, Strength Reduction) vorgeschlagen.

Ein weiteres Energiemodell auf Instruktionsebene wurde von Simunic et al. [SBM99] entwickelt. Es umfasst neben dem Prozessor auch andere Systemkomponenten, wie Speicher oder DC/DC-Wandler. Das Modell beschränkt sich auf Informationen aus den Datenblättern und daher werden beim Prozessor nur die beiden Zustände *active* und *idle* unterschieden. Genau bestimmt werden die Taktzyklen, die durch eine zyklengenaue Simulation ermittelt werden. Aus den einzelnen Energiewerten der Systemkomponenten pro Zyklus wird durch Summenbildung der Gesamtenergieverbrauch des Systems berechnet. Das Modell ist somit zur Vorhersage des Energieverbrauchs für komplette Programmabläufe geeignet. Durch diese Methode wird eine Genauigkeit von 5% für die Bewertung des Energieverbrauchs eines Programms erreicht.

Der große Vorteil dieses Ansatzes gegenüber dem von Tiwari et al. ist, dass die Notwendigkeit von Messungen entfällt und die benötigten Informationen allein aus Datenblättern erhältlich sind. Somit ist das Modell viel schneller und einfacher auf andere Prozessoren übertragbar. Weiterhin werden auch andere Systemkomponenten wie der Speicher berücksichtigt, die damit auch dem Compiler zur Optimierung zur Verfügung stehen. Die Grenzen des Energiemodells von Simunic et al. liegen allerdings bei der Bewertung unterschiedlicher Maschineninstruktionen, Bitwechseln auf Leitungen und dem Scheduling von Instruktionen. Änderungen dieser Faktoren führen nicht zu veränderten Energiewerten des Modells.

Ein weiteres Modell findet sich in Russell et al. [RJ98], wo der RISC Prozessor i960 in zwei Varianten untersucht wurde. Die Autoren haben mit der Methode von Tiwari et al. begonnen und mit einem digitalen Speicheroszilloskop den Stromverbrauch des Prozessors gemessen. Es wurde der Einfluss verschiedener Parameter, wie unterschiedliche Quell- und Zielregister, verschiedene Operanden, unterschiedliche Conditioncodes und die bedingte Ausführung untersucht. Um die Komplexität des Modells gering zu halten, wurden nur Parameter berücksichtigt, deren Anteil über 5% des Energieverbrauchs betrug. Lediglich der Wert der Operanden mit einem Einfluss von 5,4% lag für die untersuchte JF-Prozessor-Variante über dieser Schwelle. Alle übrigen Parameter wurden daher als irrelevant klassifiziert und in den folgenden Betrachtungen ausgeschlossen. In einem weiteren Schritt wurden die verschiedenen Instruktionen getrennt untersucht. Da die Messungen keine größeren Abweichungen zwischen den Instruktionen lieferten, wurde für das Modell ein konstanter Leistungsverbrauch über alle Instruktionen festgelegt. Dieses Modell kann für 99% aller Programme den Energieverbrauch mit einem Fehler kleiner als 8% errechnen. Da die Leistung als konstant angenommen wird, ist das Modell identisch mit einem Modell, welches nur die Anzahl der Zyklen berechnet und somit für die Optimierung auf Geschwindigkeit verwendet wird. Zu berücksichtigen ist ferner, dass der Speicher (sowohl Offchip- als auch Onchip-Speicher) nicht miteinbezogen wurde, sodass alle mögliche Optimierungen, die die Anzahl der Speicherzugriffe verändern, nicht mit diesem Modell betrachtet werden können.

Sehr viel detaillierter ist das Energiemodell von Lee et al. [LEMC01], welches gegenüber bisher vorgestellten Modellen die Pipeline-Stufen des Prozessors einzeln betrachtet. Für jeden Befehl in jeder Pipeline-Stufe existieren Parameter, die mit einer linearen Regression zu den Modellvariablen konvertiert werden.

In einer ersten Phase werden die Modellvariablen bestimmt, die einen wesentlichen Einfluss auf den Energieverbrauch des Prozessors haben, und in der zweiten Phase die verbliebenen Modellvariablen mithilfe der linearen Regression berechnet. Die Genauigkeit beträgt durchschnittlich 2,5%. Bisher ist das Modell jedoch auf Datenoperationen mit Befehlen beschränkt, die in einem Zyklus verarbeitet werden. Die Behandlung von Befehlen mit mehreren Zyklen und Pipeline-Stalls ist Bestandteil der zukünftigen Arbeit. Die Messung erfolgt mit einem speziellen Messaufbau und wurde von Chang et al. [CKL00] vorgestellt. Diese Messeinrichtung liefert Testdaten und speichert die gemessenen Stromwerte direkt in einem schnellen RAM. Dadurch ist die wiederholte Ausführung des zu messenden Befehls wie beim Ansatz von Tiwari et al. nicht notwendig. Als Ergebnis erhält man die Messwerte für die einzelnen Pipeline-Stufen Instruction Fetch, Decode und Execute des betrachteten ARM7-Prozessors. In den Untersuchungen wird gezeigt, dass der Einfluss von '0'- und '1'-Werten nicht unerheblich ist und festgestellt, dass die einfachen Energiemodelle und Simulationen nicht ausreichend sind. Diese letztere Aussage muss aber sicherlich eingeschränkt werden, da es vom Einsatzzweck des Modells abhängt, ob die Signalwerte relevant sind.

3.3 Energiemodell für RISC-Prozessoren

Da die bekannten Modelle nicht alle Anforderungen erfüllen, die zur Untersuchung der Energieoptimierung durch Compiler bestehen, wird nun ein neues Energiemodell vorgestellt. Nach einleitenden Definitionen wird das Energiemodell präsentiert. Es basiert auf der Betrachtung der Funktionseinheiten des Prozessors und berücksichtigt die Speicher als weitere Komponenten sowie die Busse zwischen diesen Komponenten. Weiterhin wird eine Messmethode in Kapitel 3.4 vorgestellt, mit deren Hilfe die Parameter für einen zu untersuchenden Prozessor bestimmt werden können. Zuletzt werden beispielhaft für den ARM7TDMI ermittelte Werte vorgestellt.

Definitionen

Zur Beschreibung des Modells sind folgende Funktionen und Begriffe zu definieren:

- Anzahl der '1' eines Wortes
  $w(x)$: Die Funktion $w(x)$ (= *Ones-Funktion*) bestimmt die Anzahl der '1' (nachfolgend *Ones* genannt) im Datenwort x
- Hamming-Distanz zwischen zwei Worten
  $h(x, y)$: Die Anzahl der unterschiedlichen Bits zwischen den beiden Datenworten x und y.
- Basiskosten eines Befehls
  $BaseCPU(x)$, $BaseMem(x)$: die Kosten, die innerhalb der CPU bzw. des Speichers bei der Ausführung einer einzelnen Instruktion x entstehen. Nicht enthalten sind die durch die Funktionen w und h modellierten Anteile.
- Aktivierungs- und Deaktivierungskosten von Funktionseinheiten
  $FUChange(x, y)$: Kosten für das Aktivieren oder Deaktivieren der Funktionseinheiten zwischen den Befehlen x und y. Durch den Wechsel von der Ausführung des Befehls x zur Ausführung des Befehls y werden unter Umständen Funktionseinheiten nicht mehr benötigt, die Deaktivierungskosten verursachen, während andere Funktionseinheiten neu verwendet werden und dadurch Aktivierungskosten verursachen. Die Summe dieser Kosten wird durch diese Funktion berechnet.

Für die Modellierung des Prozessorsystems wurde das System in die Komponenten Prozessor und Speicher aufgeteilt. Auch wenn der ARM7 keinen getrennten Daten- und Programmspeicher besitzt, wurden diese beiden Speicher für das Modell getrennt dargestellt (Harvard-Architektur), wodurch die mathematische

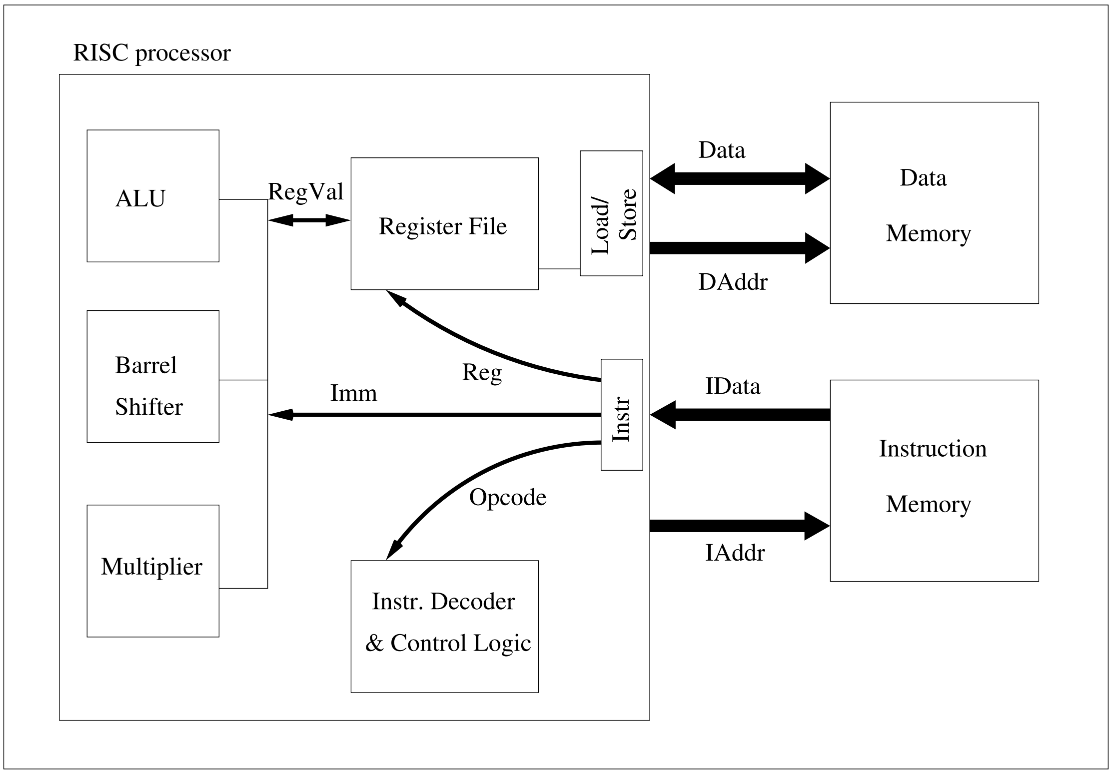


Abbildung 3.4: Blockdiagramm des modellierten Systems

Darstellung einfacher und klarer wird. Außerdem kann damit der Anwendungsbereich auf weitere Prozessorarchitekturen ausgeweitet werden. Für Prozessoren mit gemeinsamem Programm- und Datenspeicher können die betreffenden Gleichungen leicht angepasst werden.

Zwischen Prozessor und Speicher werden die Busse modelliert (siehe Abb. 3.4), für den Programmspeicher ein Adressbus (*IAddr*) und ein Datenbus (*IData*), und ebenso für den Datenspeicher ein Adressbus (*DAddr*) und Datenbus (*Data*). Die Adressbusse sind stets unidirektional, da nur der Prozessor Speicheradressen generiert. Ebenso ist der Instruktionsdatenbus (*IData*) unidirektional, da die Instruktionen darüber stets nur gelesen werden. Der Datenbus (*Data*) zum Datenspeicher transportiert als einziger Bus Informationen zum Lesen und zum Schreiben in beide Richtungen. Diese detaillierte Betrachtung der Busse trägt der Tatsache Rechnung, dass der Energieverbrauch dort relativ hoch ist und aus diesem Grund Optimierungen zur Reduzierung eingesetzt werden sollen.

Der Prozessor selbst wird anhand der frei verfügbaren technischen Unterlagen als Menge von Funktionseinheiten, die durch Busse miteinander verbunden sind, modelliert. Dies ermöglicht auch die Untersuchung von Registerwerten und den Aktivierungs- und Deaktivierungskosten von Funktionseinheiten sowie die Zerlegung der Befehlsworte in Opcode, Registernummern und Immediate-Werte. Das Blockdiagramm des Energiemodells in Abbildung 3.4 kann für RISC-Prozessoren mit Load/Store-Architektur angewandt werden. Bei zusätzlichen oder nicht vorhandenen Funktionseinheiten lässt sich das Modell einfach modifizieren. Pipeline-Stufen werden nicht separat betrachtet, da nicht zu erwarten ist, dass der Effekt durch die detaillierte Betrachtung der Pipeline-Stufen Einfluss auf die Codegenerierung im Compiler haben wird. Außerdem würde das Modell dadurch viel komplexer und es müssten gesonderte Messeinrichtungen entwickelt werden.

Um nun den Einfluss durch Instruktionen geeignet zu modellieren und auch die beiden Speicher getrennt zu betrachten, werden die Energiekosten in die folgenden Bestandteile zerlegt:

1. instruktionsabhängige Kosten innerhalb des Prozessors (E_{cpu_instr})
2. datenabhängige Kosten innerhalb des Prozessors (E_{cpu_data})
3. instruktionsabhängige Kosten im Programmspeicher (E_{mem_instr})
4. datenabhängige Kosten im Datenspeicher (E_{mem_data})

Diese Kosten ergeben zusammen die Gesamtenergiekosten des Systems:

$$E_{total} = E_{cpu\_instr} + E_{cpu\_data} + E_{mem\_instr} + E_{mem\_data}$$

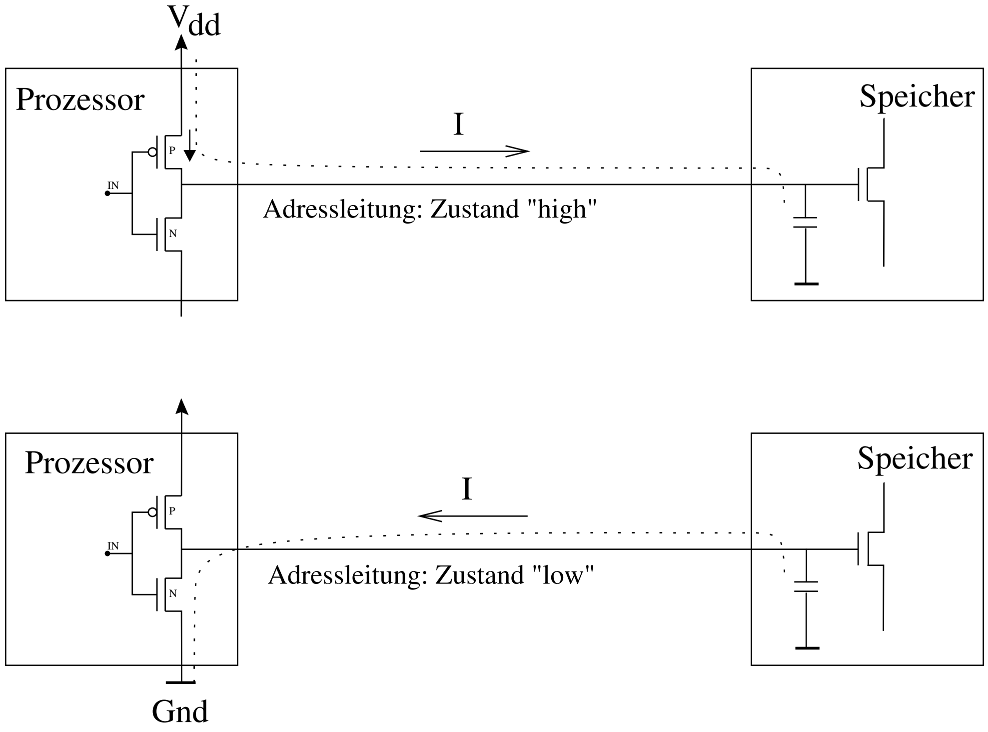


Abbildung 3.5: Stromfluss über die Busse in Abhängigkeit des Zustandes

Die Kosten, die außerhalb von Prozessor und Speichern auf den Bussen anfallen, werden entweder dem Prozessor oder einem der Speicher zugeordnet. Der Grund hierfür ist, dass die Busse von einer der beiden Komponenten gespeist und die Energiekosten bei einer dieser Komponenten in dem später vorgestellten Messverfahren gemessen werden (Abb. 3.5). Da der Stromverbrauch jeweils am V_{dd}-Eingang der Komponenten gemessen wird, wird beispielsweise der Strom für das Anlegen einer '1' auf dem Adressbus beim Prozessor gemessen und das Anlegen einer '0' beim Speicher, da dann der Strom durch den V_{dd}-Eingang des Speichers und die Adressbus-Pins des Bausteins über den Bus zum Prozessor fließt und dort durch die Ausgangstreiber zu Gnd weitergeleitet wird.

Die instruktionsabhängigen Kosten innerhalb des Prozessors sind abhängig von einem Immediate-Wert Imm, den Registernummern Reg, den Inhalten der bearbeiteten Register $RegVal$ und der Instruktionsadresse $IAddr$. Es muss nun eine Entscheidung darüber getroffen werden, wie diese Komponenten modelliert werden. Für dieses Modell wurde von einem additiven Anteil der jeweiligen w- und h-Anteile ausgegangen. Dies heißt, dass diese Anteile unabhängig von anderen Anteilen sind und der Energieverbrauch, der sich aus den einzelnen Komponenten ergibt, einfach addiert werden kann. Wenn also eine zusätzliche Leitung beispielsweise den Signalwert '1' führt, wird der w-Anteil auf den Energieverbrauch addiert. Andere Komponenten werden nicht dadurch beeinflusst. Dieser Ansatz kann auch - wie später anhand der Messwerte gezeigt wird - für den betrachteten ARM7-Prozessor als ausreichend betrachtet werden. Daher kann nun durch Proportionalitätskonstanten der Anteil durch w- und h-Funktion zu den anderen Energiekomponenten zuaddiert werden. In den folgenden Gleichungen werden nun diese Konstanten als Parameter α für die Anzahl der *Ones*, die mit der Funktion w berechnet werden, und die Konstante β für die Hammingdistanz, die mit der Funktion h bestimmt wird, hinzugefügt.

Da eine Instruktion potenziell mehrere Immediate-Werte und Registernummern beinhalten und Einfluss auf mehrere Registerinhalte haben kann, werden die j Immediate-Werte $Imm_{i,j}$, k Registernummern $Reg_{i,k}$ sowie k Registerinhalte $RegVal_{i,k}$ für eine Instruktion i summiert. Es ergibt sich daher für die Gesamtsumme E_{cpu_instr} für eine Folge von m Instruktionen mit s_i Immediate-Werten und t_i Registern die folgende Gleichung:

$$
\begin{aligned}
E_{cpu\_instr} = \sum_{i=1}^{m} \quad & \Big(BaseCPU(Opcode_i) + \\
& \sum_{j=1}^{s_i} (\alpha_1 * w(Imm_{i,j}) + \beta_1 * h(Imm_{i-1,j}, Imm_{i,j})) + \\
& \sum_{k=1}^{t_i} (\alpha_2 * w(Reg_{i,k}) + \beta_2 * h(Reg_{i-1,k}, Reg_{i,k})) + \\
& \sum_{k=1}^{t_i} (\alpha_3 * w(RegVal_{i,k}) + \beta_3 * h(RegVal_{i-1,k}, RegVal_{i,k})) + \\
& \alpha_4 * w(IAddr_i) + \beta_4 * h(IAddr_{i-1}, IAddr_i) + \\
& FUChange(Instr_{i-1}, Instr_i) \Big)
\end{aligned}
$$

Für den h-Anteil und die *FUChange*-Funktion wird auf die vorherigen Werte der Variablen, die für den Index 0 undefiniert sind, zugegriffen. Daher müssen der Immediate-Wert, die Registernummer, die Adressen oder Daten auf den Bussen für den Index 0 wie folgt definiert werden:

$$X_{o,y} := 0 \text{ für } X \in \{Imm, Reg, RegVal, IAddr, IData, Addr, Data\}$$

Entsprechend den instruktionsabhängigen Kosten ergeben sich die datenabhängigen Kosten innerhalb der CPU für *n* Datenspeicherzugriffe in Abhängigkeit der Datenadresse *DAddr*, der Daten *Data* selbst und der Richtung *dir* (read/write). Die Summe des Energieverbrauchs der *n* Zugriffe auf den Datenspeicher beträgt dann:

$$
\begin{aligned}
E_{cpu\_data} = \sum_{i=1}^{n} \quad & \Big(\alpha_5 * w(DAddr_i) + \beta_5 * h(DAddr_{i-1}, DAddr_i) + \\
& \alpha_{6,dir} * w(Data_i) + \beta_{6,dir} * h(Data_{i-1}, Data_i) \Big)
\end{aligned}
$$

Bei der Betrachtung des Speichers ergeben sich für den Programmspeicher Kosten durch die Basiskosten *BaseMem* in Abhängigkeit des Speichertyps *InstrMem* und der Wortbreite *Word_width* des jeweiligen Zugriffs *i*. Zusätzlich entstehen Kosten durch die jeweilige Instruktionsadresse *IAddr* und die zu lesenden

Instruktionen *IData*. Hieraus ergibt sich die folgende Summe für die Ausführung von m Programmspeicherzugriffen:

$$
\begin{aligned}
E_{mem\_instr} = \sum_{i=1}^{m} \quad & \Big(BaseMem(InstrMem, Word\_width_i) + \\
& \alpha_7 * w(IAddr_i) + \beta_7 * h(IAddr_{i-1}, IAddr_i) + \\
& \alpha_8 * w(IData_i) + \beta_8 * h(IData_{i-1}, IData_i)\Big)
\end{aligned}
$$

Entsprechend ergeben sich die datenabhängigen Kosten im Datenspeicher für *n* Datenzugriffe wie folgt:

$$
\begin{aligned}
E_{mem\_data} = \sum_{i=1}^{n} \quad & \Big(BaseMem(DataMem, dir, Word\_width_i) + \\
& \alpha_9 * w(DAddr_i) + \beta_9 * h(DAddr_{i-1}, DAddr_i) \\
& \alpha_{10,dir} * w(Data_i) + \beta_{10,dir} * h(Data_{i-1}, Data_i)\Big)
\end{aligned}
$$

Mit diesen Gleichungen lässt sich nun anhand der Basiskosten *BaseCPU* für jede Instruktion und der Basiskosten der Speicher *BaseMem* für den jeweiligen Speicher sowie der Parameter α_1 bis α_{10} und β_1 bis β_{10} und *FUChange* für ein beliebiges Programm der Energieverbrauch abschätzen. Dies kann auch schon während des Compiliervorgangs für einzelne Instruktionen oder Instruktionssequenzen durchgeführt werden. Zu diesem Zeitpunkt sind die Adressen *IAddr* und *DAddr* noch nicht bekannt oder können bestenfalls auf einen bestimmten Adressbereich eingegrenzt werden. In diesem Fall werden für die unbekannten Bits Durchschnittswerte angesetzt.

Von den beiden Methoden der Simulation und der Messung zur Bestimmung der Parameter eines individuellen Prozessors und -systems, wurde, um den ARM7-Prozessor verwenden zu können, die Methode der Messung gewählt und ein entsprechendes Messverfahren ausgearbeitet. Diese Methode ist notwendig, da eine detaillierte Hardwarebeschreibung nicht allgemein verfügbar ist und somit über eine Simulation der Schaltung der Energieverbrauch nicht bestimmt werden kann. Die Parameter des Modells $BaseCPU$, $BaseMem$, α_i, β_i und $FUChange$ müssen nun mithilfe verschiedener Messreihen für den jeweiligen Prozessor und das jeweilige System konkret bestimmt werden, was im folgenden Unterkapitel untersucht wird.

3.4 Messverfahren

Für die Messungen soll ein möglichst einfacher Hardwareaufbau ausreichen, damit diese auch ohne größeren Aufwand für weitere Prozessoren oder auch durch andere Forschungsgruppen wiederholt werden können. Es wurde daher darauf verzichtet, digitale Speicheroszilloskope oder speziell entwickelte Messeinrichtungen zu verwenden. Der hardwaremäßige Aufbau (siehe Abb. 3.6) ist identisch mit dem Aufbau für die Messungen des Energiemodells von Tiwari et al. [TMW94b], bezieht aber noch zusätzlich die Speicherbausteine mit ein. Die Messungen erfolgen abwechselnd für den Prozessorstrom I_{proc} und den Speicherstrom I_{mem}, damit ein einzelnes Messgerät ausreichend ist. Der Prozessorstrom beinhaltet allerdings auch den Strom für den Onchip-Speicher, der nicht getrennt gemessen werden kann, da er bei dem ausgewählten Baustein, dem AT91M40400, über dieselben V_{dd}-Leitungen versorgt wird. Durch zusätzliche geeignete Testmuster muss daher ermittelt werden, wie groß dieser Anteil ist. Es existieren weitere

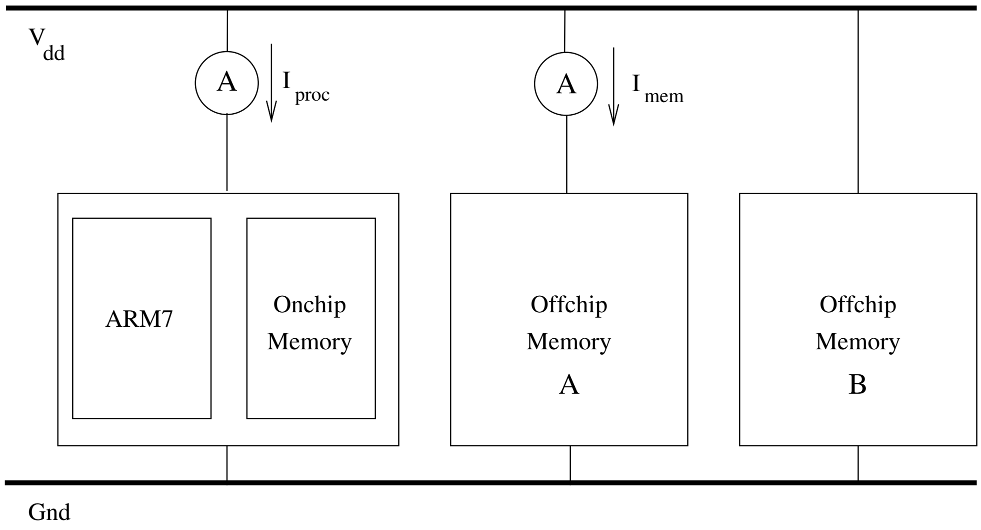


Abbildung 3.6: Messaufbau

Offchip-Speicher, die nicht für die Messungen vorbereitet sind und deren Energieanteil nicht gemessen werden kann.

Es ist noch anzumerken, dass mit dem Amperemeter der durchschnittliche Strom gemessen wird. Da diese Durchschnittsbildung nur über einen begrenzten Zeitraum erfolgen kann, muss die Länge der Testprogramme begrenzt werden. Das Zeit-Intervall für das verwendete Amperemeter wurde bei Theokharidis [The00] auf Instruktionssequenzen mit ca. 20 Instruktionen bestimmt. Daher muss bei der Entwicklung der Messreihen darauf geachtet werden, dass diese nicht mehr als 20 Instruktionen beinhalten. Dies reicht zur Validierung des Modells aus, verhindert jedoch die einfache Bestimmung des Energieverbrauchs für vollständige Programme.

Für die vorgestellte Hardware müssen nun geeignete Testmuster zur Bestimmung der Parameter entwickelt werden. Zusätzlich wird Software benötigt, die diese Testmuster auf der Hardware generiert und die Messungen für I_{proc} und I_{mem} durchführt.

Um die Parameter zu bestimmen, werden im Folgenden drei Schritte ausgeführt:

1. Planung und Durchführung der Messreihen
2. Datenumrechnung
   Umrechnung der Messergebnisse auf mehrere Speicherbausteine, Linearisierung sowie Transformation in Energiewerte
3. Bestimmung der Parameter des Modells

In den nachfolgenden Unterkapiteln werden diese Schritte im Detail beschrieben.

3.5 Planung und Durchführung der Messreihen

Parameter BaseCPU und BaseMem

Wie beim Messverfahren von Tiwari werden die zu messenden Befehle in einer Schleife ausgeführt. Für die Bestimmung der *Basiskosten* (hier als Strom gemessen) ist in der Schleife lediglich der zu messende Befehl *Instr1* (Abb. 3.7a) vorhanden. Hierdurch werden die Kosten durch das Umschalten zwischen

```
LblA Instr1
  Instr1
  ..
  ..
  Instr1
  JMP LblA
```

(a) Bestimmung Basiskosten

```
LblB Instr1
  Instr2
  Instr1
  Instr2
  ..
  JMP LblB
```

(b) Bestimmung Interinstruktionskosten

Abbildung 3.7: Messschleifen

Funktionseinheiten oder durch andere Operanden oder Register minimal. Für die Bestimmung der *Interinstruktionskosten* werden zwei Instruktionen *Instr1* und *Instr2* alternierend ausgeführt (siehe Abb. 3.7b) und somit die Kosten durch die Umschaltung bestimmt. Vom Messwert I_{mess} müssen aber noch anteilig die *Basiskosten* der beiden Instruktionen subtrahiert werden, da die *Interinstruktionskosten* nur den Aufschlag zur mehrmaligen Ausführung des gleichen Befehls *Instr1* bzw. *Instr2* repräsentieren:

$$InterInstrKosten(Instr1, Instr2) = I_{mess} - \frac{Basiskosten(Instr1) + Basiskosten(Instr2)}{2}$$

Das Messverfahren wurde im Detail von Theokharidis [The00] für alle Instruktionen des ARM7TDMI durchgeführt und ausgewertet. Für das vorgestellte Energiemodell reichen diese Messungen jedoch nicht aus. Sie dienen lediglich zur Bestimmung des Parameters *BaseCPU* in dem hier vorgestellten Modell. Der Parameter *BaseCPU* entspricht den *Basiskosten* des Modells von Tiwari abzüglich der Anteile, die durch andere im Vergleich zu Tiwari zusätzlichen Modellparameter bestimmt werden. Die Anteile für *FUChange* und die Parameter β_1, β_2, β_5 und β_8, die sich auf die Hamming-Distanz der Instruktionsworte beziehen, sind für die Bestimmung von *BaseCPU* nicht relevant, da bei mehrfacher Ausführung nur einer Instruktion diese Anteile den Wert 0 annehmen.

Für die Bestimmung des Parameters *BaseMem* muss berücksichtigt werden, dass bei dem gewählten ARM7-Prozessor ein gemeinsamer Speicher für Programm und Daten vorhanden ist. Allerdings existieren im betrachteten System (siehe Abb. 3.6) mehrere Speicher (die Offchip-Speicherbausteine A und B sowie ein Onchip-Speicher), von denen nur ein Speicherbaustein für die Messungen zur Verfügung steht. Programm-Speicherzugriffe können nun entweder durch die Ausführung von Instructionfetches gemessen werden, wenn das Programm mit der Messschleife im zu messenden Speicher (in Offchip-Speicher A oder im Onchip-Speicher) liegt und mögliche Datenzugriffe auf andere Speicher erfolgen. Oder es können alternativ Datenzugriffe gemessen werden, wenn das Programm mit der Messschleife in einem anderen Speicher liegt, dessen Strom nicht gemessen wird, und Datenzugriffe auf dem der Messungen zugehörigen Speicher vorgenommen werden.

Parameter FUChange

Die Funktion *FUChange* berechnet die zusätzlichen Energiekosten durch das Aktivieren oder Deaktivieren von Funktionseinheiten. Zur Bestimmung dieser Kosten müssen wie bei der Messung von Tiwari für die *Interinstruktionskosten* zwei Befehle abwechselnd ausgeführt werden, von denen einer die betrachtete Funktionseinheit verwendet und der andere Befehl nicht. Mithilfe dieser Messung kann durch Bildung der Differenz der Parameter für jeweils eine einzelne Funktionseinheit bestimmt werden. Die Funktion *FUChange* liefert dann die Summe aus den Aktivierungs- und Deaktivierungskosten der Funktionseinheiten, die von zwei aufeinander folgenden Befehlen verursacht werden.

```
  Adressbits:     0  1  2  3  4  5  6  7          0  1  2  3  4  5  6  7
                 -----------------------         -----------------------
Testpattern1      0  0  0  0  0  0  0  0          0  0  0  0  0  0  0  0
Testpattern2      1  0  0  0  0  0  0  0          1  0  0  0  0  0  0  0
Testpattern3      0  1  0  0  0  0  0  0          1  1  0  0  0  0  0  0
Testpattern4      0  0  1  0  0  0  0  0          1  1  1  0  0  0  0  0
Testpattern5      0  0  0  1  0  0  0  0          1  1  1  0  0  0  0  0
Testpattern6      0  0  0  0  1  0  0  0          1  1  1  1  1  0  0  0
Testpattern7      0  0  0  0  0  1  0  0          1  1  1  1  1  1  0  0
Testpattern8      0  0  0  0  0  0  1  0          1  1  1  1  1  1  1  0
Testpattern9      0  0  0  0  0  0  0  1          1  1  1  1  1  1  1  1
```

Abbildung 3.8: (a) wandernde *Ones* (b) steigende Anzahl *Ones*

Parameter α_1 bis α_{10}

Die Parameter α_1 bis α_{10} stellen die Gewichtung der *Ones* dar. *Ones* entsprechen dem Wert '1' einer Leitung und die *Ones*-Kosten entstehen dadurch, dass Energie aufgewendet werden muss, um eine Leitung auf den Wert '1' zu setzen. Das vorgestellte Modell geht von einer Gleichgewichtung der *Ones* aus, d. h. es treten die gleichen Zusatzkosten z. B. durch *n Ones* auf, unabhängig davon, welche der möglichen Leitungen die *Ones*-Werte annehmen. Um sicherzugehen, dass diese Annahme auch zutrifft, muss vorher eine Messreihe durchgeführt werden, die ein einzelnes *Ones* in mehreren Messreihen jeweils auf einer Leitung aktiviert (Abb. 3.8a). Die Messungen müssen gleiche Werte liefern, um diese Annahme des Modells zu bestätigen. Nach der Bestätigung dieser Annahme ist nur noch die Anzahl der *Ones* für den Energieverbrauch relevant. Danach können die eigentlichen Messreihen der Parameter durchgeführt werden.

Die Parameter α_1 bis α_{10} bilden Energiekosten in Zusammenhang mit der Anzahl der *Ones* nach. Zur Bestimmung dieser Parameter muss also für die jeweilige Komponente *Imm*, *Reg*, *RegVal*, *IAddr*, *DAddr*, *Data*, *IData* die Anzahl der *Ones* von 0 bis zur Wortbreite variiert werden (Abb. 3.8b) und durch jeweils zwei entsprechende unterschiedliche Instruktionen wie in Abbildung 3.7b gemessen werden.

Parameter β_1 bis β_{10}

Ähnlich wie bei den Parametern α_1 bis α_{10} wird zuerst die Annahme geprüft, ob alle Leitungen die gleiche Wertigkeit bezüglich dieser so genannten Hamming-Distanz besitzen. In verschiedenen Messreihen wird dafür der Zustand einer Leitung in aufeinander folgenden Befehlen geändert, beginnend bei der niederwertigsten bis zur höchstwertigsten Leitung (Abb. 3.8a). Auch hier müssen die Messwerte bis auf kleine tolerierbare Unterschiede identisch sein.

Die Parameter β_1 bis β_{10} bilden Energiekosten im Zusammenhang mit dem Zustandswechsel auf den Busleitungen nach. Im Gegensatz zu der Funktion *Ones* ist hier stets der Bezug zwischen dem alten und neuen Zustand einer Leitung relevant. Daher wird bei den Messungen immer abwechselnd ein Bezugswert, in diesem Fall ein '0'-Vektor (Testpattern1 in Abb. 3.8), angelegt und danach das aktuelle Muster.

In den nachfolgenden Messreihen kann dann jeweils für die Komponenten *Imm*, *Reg*, *RegVal*, *IAddr*, *DAddr*, *Data*, *IData* die Hamming-Distanz von 0 bis zum größtmöglichen Wert variiert werden (Abb. 3.8b), um den entsprechenden Parameter β zu bestimmen.

Anzahl der Messungen

Für die Bestimmung der Parameter des Adressbusses (α_4,β_4,α_5,β_5,α_7,β_7,α_9,β_9) kann der Suchraum bei einem n-Bit-Adressbus mit 2^nAdressen durch $4 * n$ Messungen abgedeckt werden. Diese Anzahl setzt sich aus $n + 1$ Messungen für die Überprüfung der Annahme der gleichen Wertigkeit aller Leitungen und $n + 1$ Messungen mit einer unterschiedlichen Anzahl von *Ones* zusammen. Zusätzlich sind nochmals die gleichen Messungen für die Hamming-Funktion notwendig (siehe Abb. 3.8). Weil die jeweils ersten beiden Testmuster übereinstimmen, sind $4 * n$ Messungen ausreichend. Weiterhin kann man ausnutzen, dass der Speicherbaustein, der für die Messungen des Stroms I_{mem} genutzt wird, nicht den vollen Adressbereich des Prozessors abdeckt. Die Anzahl der Messungen kann so bei einem Adressraum des Speicherbausteins von 2^mAdressen auf $4 * m$ Messungen noch weiter reduziert werden. Die Beschränkung auf m kann ohne Einschränkung der Qualität der Ergebnisse vorgenommen werden, weil durch die Testmuster (in Abb. 3.8a) jeweils die Linearität, die eine notwendige Voraussetzung dieses Modells ist, geprüft wurde. Insgesamt ergibt sich eine Anzahl der Messungen mit $m = 17$ für die 2 Adressbusse ($IAddr$ und $DAddr$) und jeweils für die CPU bzw. den Speicherbaustein von $4 * 17 * 2 * 2 = 272$ Messungen.

Entsprechendes gilt für einen k-Bit-Datenbus, der durch $4*k$ Messungen abgedeckt werden kann. Bei dem betrachteten Aufbau mit dem Evaluationboard liegt ein externer 16-Bit-Datenbus vor, an den 2 Speicherbausteine mit 8-Bit-Datenbus angeschlossen sind. Dadurch reduziert sich die Anzahl der notwendigen Messungen auf $4 * 8$. Für die restlichen Komponenten *Imm*, *Reg* und *RegVal* kann ebenfalls analog die Anzahl der Messungen bestimmt werden (siehe Tab. 3.2).

Parameter	Komponente	Bitbreite	Anzahl Messungen
BaseCPU	BaseCPU	-	95
BaseMem	BaseMem	-	4
α_4,β_4,α_5,β_5,α_7,β_7,α_9,β_9	IAddr, DAddr	2*17	272
α_6,β_6,α_{10},β_{10}	IData, Data	2*8	64
α_1,β_1	Imm	8	32
α_2,β_2	Reg	3	12
α_3,β_3	RegVal	32	128
Summe			607

Tabelle 3.2: Anzahl der Messungen

Die Details der Durchführung der Messreihen können bei Knauer [Kna01] nachgelesen werden. Insgesamt kann festgestellt werden, dass die Anzahl von 607 Messungen (Tab. 3.2) für den ARM7TDMI durch dieses Modell zwar hoch aber noch akzeptabel ist.

3.6 Datenumrechnung

Die bei der Durchführung der Messreihen erhaltenen Werte können noch nicht direkt als Parameter des Modells verwendet werden. Es müssen Eigenschaften der Messeinrichtung und der physikalischen Messungen berücksichtigt werden. Weiterhin ist zu berücksichtigen, dass durch die Messreihen nicht ausschließlich Werte für einen spezifischen Parameter erhoben werden können, sondern dass bei jeder Messreihe auch andere Parameter mitgemessen werden, die anschließend herausgerechnet werden müssen.

3.6.1 Hochrechnung auf mehrere Speicherbausteine

Bei 16-Bit-Zugriffen wird auf beide Speicherbausteine des verwendeten Evaluationboards parallel zugegriffen und 32-Bit-Zugriffe über zwei aufeinander folgende 16-Bit-Zugriffe realisiert. Um den Aufwand für den Aufbau der Messeinrichtung zu beschränken, wurde nur die V_{dd}-Leitung eines der beiden baugleichen 8-Bit-Speicherbausteine unterbrochen. Daher müssen die Werte des einen gemessenen Speicherbausteins auf beide Bausteine hochgerechnet werden, um die Energiewerte für einen vollständigen Speicherzugriff zu erhalten. Diese Hochrechnung geschieht für den Parameter $BaseMem$ durch die Verdopplung des Messwertes. Für die Parameter α und β ist diese Umrechnung komplexer. Auf dem Datenbus ist nur die Hälfte der Leitungen durch die Messvorrichtung einbezogen während die anderen Datenleitungen entsprechend hochgerechnet werden müssen. Weiterhin ist die niederwertigste Adressleitung für die Auswahl des Speicherbausteins zuständig und liegt daher nicht am untersuchten Speicherbaustein an. Die restlichen Adressleitungen sind bei beiden Speicherbausteinen parallel angeschlossen und führen zu einer einfachen Verdopplung der Messwerte.

Nachdem diese Umrechnungen durchgeführt wurden, liegen Werte für einen fiktiven 16-Bit-Speicherbaustein vor.

3.6.2 Linearisierung der Messwerte

Das Modell setzt einen linearen Zusammenhang zwischen der Anzahl der *Ones* bzw. des Hamming-Abstandes und dem Energieverbrauch voraus. Da bei den Messungen jedoch Messfehler auftreten und ebenso geringe Unterschiede zwischen den einzelnen Leitungen vorliegen, muss ein geeignetes statistisches Verfahren verwendet werden, um aus den Messwerten die Parameter zu errechnen. Als statistische Methode kann hierfür die *Regressionsanalyse* angewandt werden. Um bei einer Messreihe aus n Messungen mit den Messpunkten (x_1,y_1) bis (x_n,y_n) die Datenwerte durch eine Gerade zu approximieren, kann die lineare Regressionsmethode angewandt werden. Die so genannte *Regressionsgerade f* [Gro96] kann durch ein Polynom $y = a + bx$ mit den Regressionskoeffizienten a und b definiert werden. Die Güte des Polynoms wird durch die Summe über alle quadratischen Fehler δ_i^2 definiert. Diese Summe des quadratischen Fehlers Δ^2 soll minimal sein:

$$\Delta^2 = \sum_{i=1}^{n} \delta_i^2$$

Als Grundlage zur Berechnung werden die folgenden Definitionen eingeführt [Gro96]:

arithmetisches Mittel:

$$\overline{z} := \frac{1}{n} \sum_{i=1}^{n} z_i$$

Stichprobenkovarianz:

$$s_{XY} = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})(y_i - \overline{y})$$

Stichprobenvarianz:

$$s_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

daraus ergeben sich dann nach der Regressionsanalyse die Gleichungen für die Parameter a und b:

$$b = \frac{s_{XY}}{s_x^2} = \sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}) / \sum_{i=1}^{n}(x_i - \overline{x})^2$$

$$a = \overline{y} - b * \overline{x}$$

Die hier angewendete Regressionsmethode kann sich auf die y-Komponente als stochastische Größe beschränken, da die Anzahl der *Ones* bzw. des Hamming-Abstandes als x-Komponente keiner Ungenauigkeit unterliegt.

3.6.3 Umrechnung von Strom- in Energiewerte

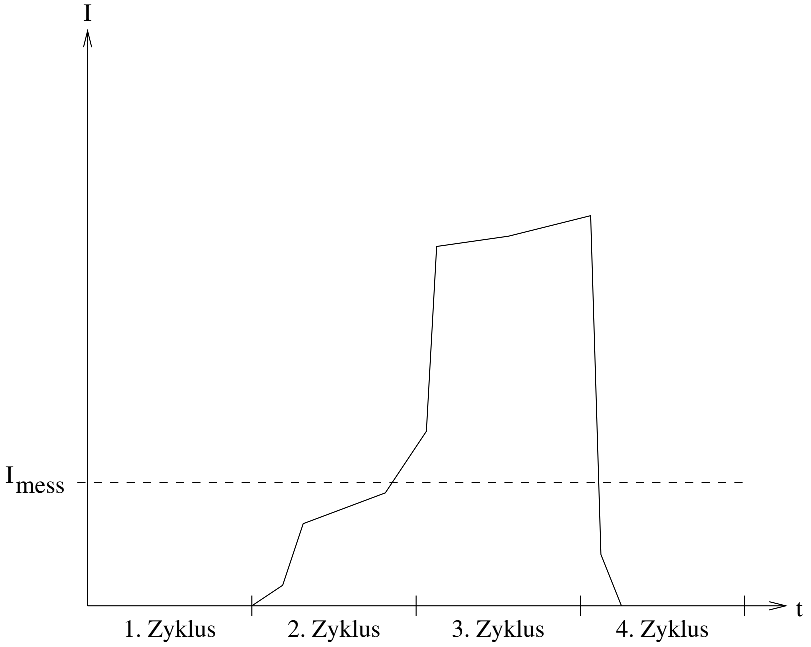


Abbildung 3.9: schematische Darstellung des Speicherstroms während eines Load-Zugriffes

In einer letzten Stufe müssen die mithilfe des Amperemeters gemessenen Stromwerte in Energiewerte umgerechnet werden, da letztendlich der Energieverbrauch modelliert werden soll. Grundsätzlich kann bei konstantem Strom I_{mess} und der Versorgungsspannung V_{dd} die Energie E_{access} mithilfe der Anzahl der Zyklen n und der Zykluszeit T berechnet werden:

$$E_{access} = U * I * t = V_{dd} * I_{mess} * t = V_{dd} * I_{mess} * n * T$$

Allerdings können einige Testmuster nur mithilfe von Befehlen angelegt werden, die mehr Zyklen benötigen als die zu messenden Ereignisse selbst. Beispielsweise muss für einen Zugriff auf den Datenspeicher ein Load- oder Store-Befehl verwendet werden, der z. B. mit 16-Bit-Zugriff (=Halfword) 4 bzw. 3 Zyklen benötigt. Der Zugriff auf den Datenspeicher erfolgt aber nur in einem Teil der 4 bzw. 3 Zyklen. Wenn man davon ausgeht, dass der Strom durch das Messgerät integriert wird, hat der zu messende Strom während des Zugriffs einen entsprechend höheren Wert (Abb. 3.9). Da der Zugriff aber nur in einem Teil der Zyklen stattfindet, ergibt sich die gleiche Formel wie oben:

$$E_{access} = V_{dd} * \int I(t)dt = V_{dd} * (I_{mess} * n) * t = V_{dd} * I_{mess} * n * T$$

Der Energieverbrauch E_{access} beim Speicherzugriff ist also identisch. Der in Wirklichkeit nicht konstante Strom I_{mess} wird durch das Messgerät als mittlerer Strom pro Load/Store-Befehl dargestellt, kann aber im Energiemodell als konstanter Strom I_{mess} eingesetzt werden.

3.7 Auswertung und Modellparameter

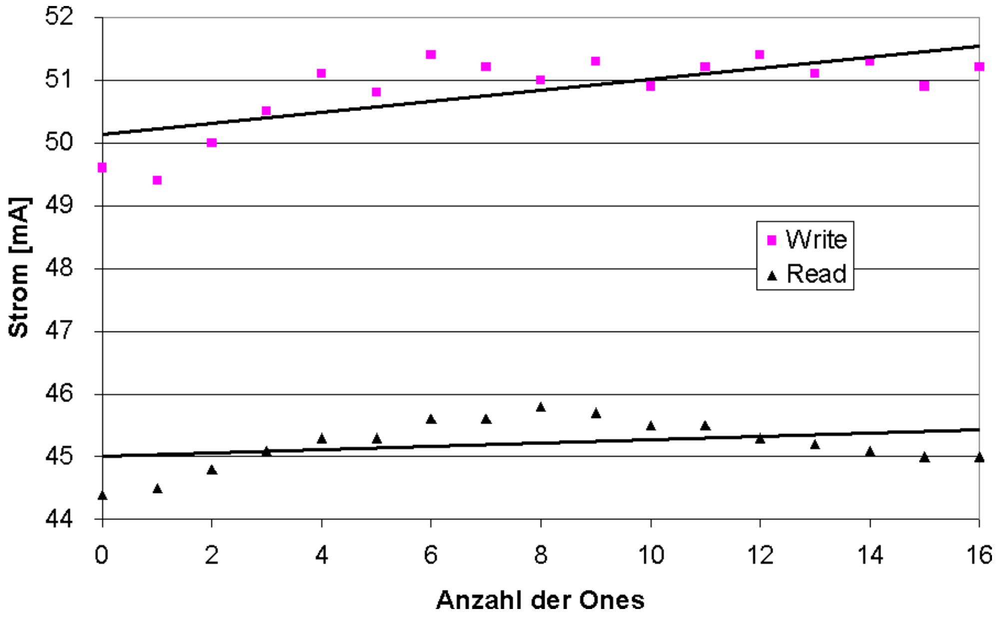


Abbildung 3.10: CPU-Strom in Abhängigkeit der Anzahl der *Ones* auf dem Datenbus

Einige beispielhafte Messreihen mit der zugehörigen Regressionsgeraden sind in den folgenden Abbildungen dargestellt. Die Abbildung 3.10 zeigt den gemessenen Strom des Prozessors in Abhängigkeit von der Anzahl der *Ones* auf dem Datenbus. Deutlich zu sehen ist, dass relevante Unterschiede zwischen Read- und Write-Zugriffen festzustellen sind. Es muss hier jedoch berücksichtigt werden, dass diese Darstellung nur den CPU-Strom enthält und der Speicherstrom stets mit betrachtet werden muss (Abb. 3.11). Hier ist die entgegengesetzte Steigung der Regressionsgeraden wie beim Prozessorstrom zu verzeichnen. Zur Energieoptimierung muss daher die Summe dieser beiden Messungen berücksichtigt werden (Abb. 3.12). Wenn ein Energiemodell nur den Energieverbrauch des Prozessors berücksichtigt, würde der Compiler versuchen, die Anzahl der *Ones* zu minimieren. Dies wäre jedoch für das gesamte System gesehen genau die falsche Optimierungsrichtung. Bei zusätzlicher Berücksichtigung der Energieverluste im Speicher kommt man zu der Erkenntnis, dass bei Betrachtung des gesamten Systems die Anzahl der *Ones* maximiert werden sollte. Anhand dieses praktischen Beispiels lässt sich die Notwendigkeit des Speichers als Bestandteil des Energiemodells darlegen. Man kann auch erkennen, dass diese Effekte einen nennenswerten Einfluss haben. Zwischen dem Strom bei einem Wert '0' und dem Wert '1' auf allen Datenleitungen besteht eine Differenz von fast 10%. Es ist daher zu prüfen, wie viel von diesem Potenzial durch mögliche Optimierungen genutzt werden kann.

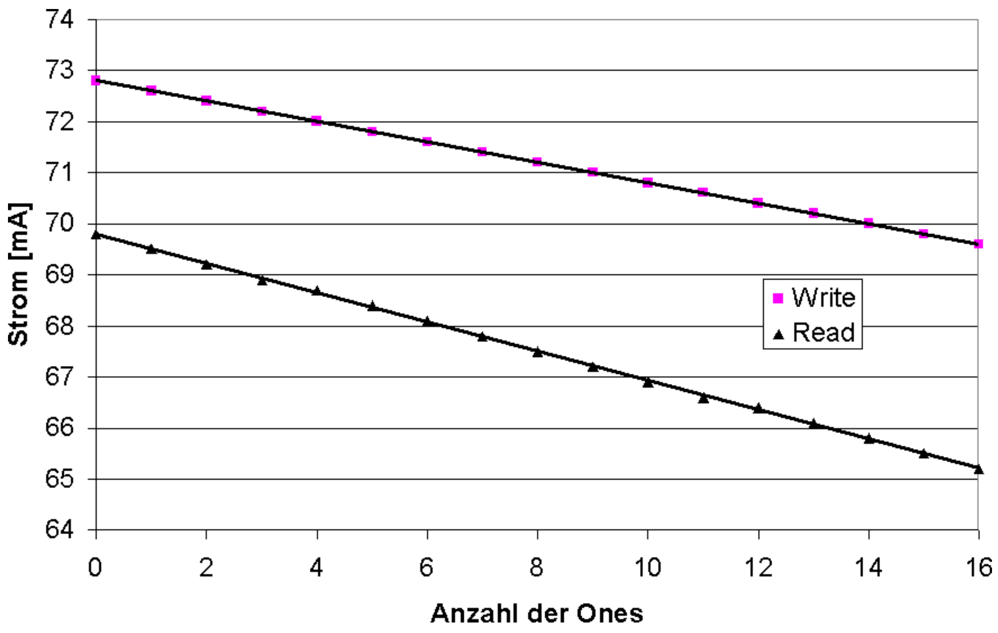


Abbildung 3.11: Speicherstrom in Abhängigkeit von der Anzahl der *Ones* auf dem Datenbus

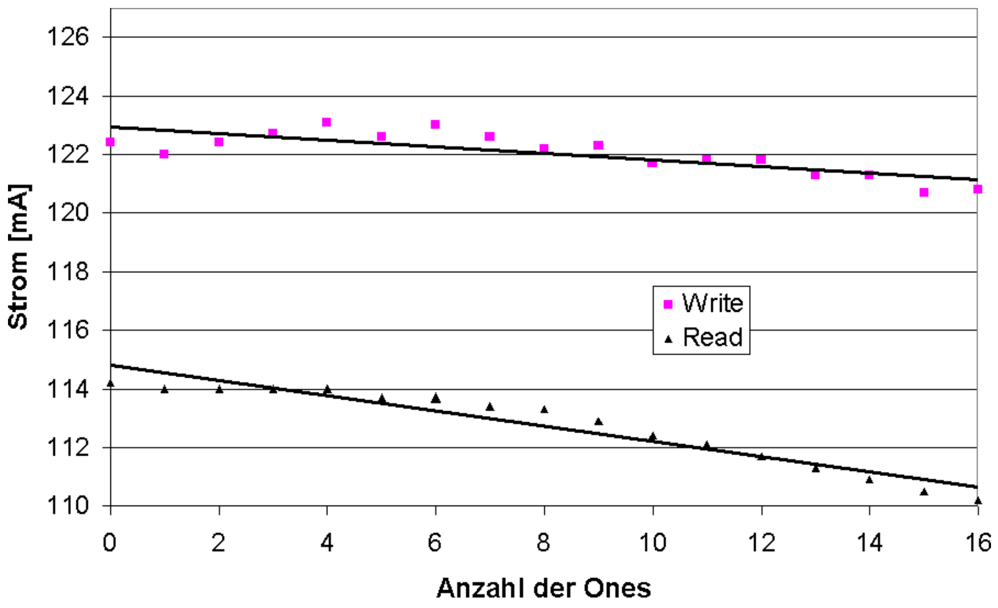


Abbildung 3.12: Gesamtstrom in Abhängigkeit von der Anzahl der *Ones* auf dem Datenbus

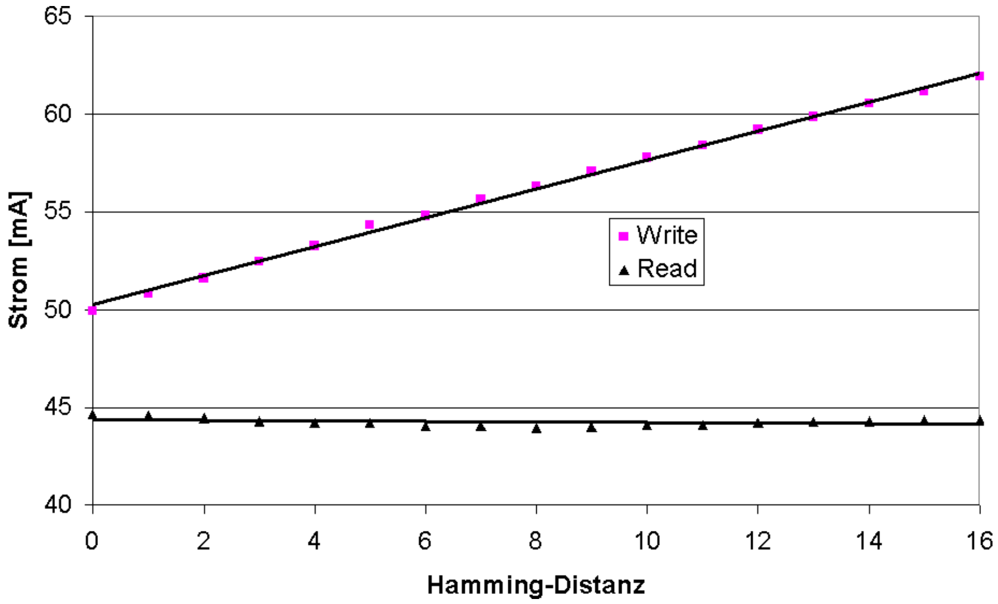


Abbildung 3.13: CPU-Strom in Abhängigkeit von der Hamming-Distanz auf dem Datenbus

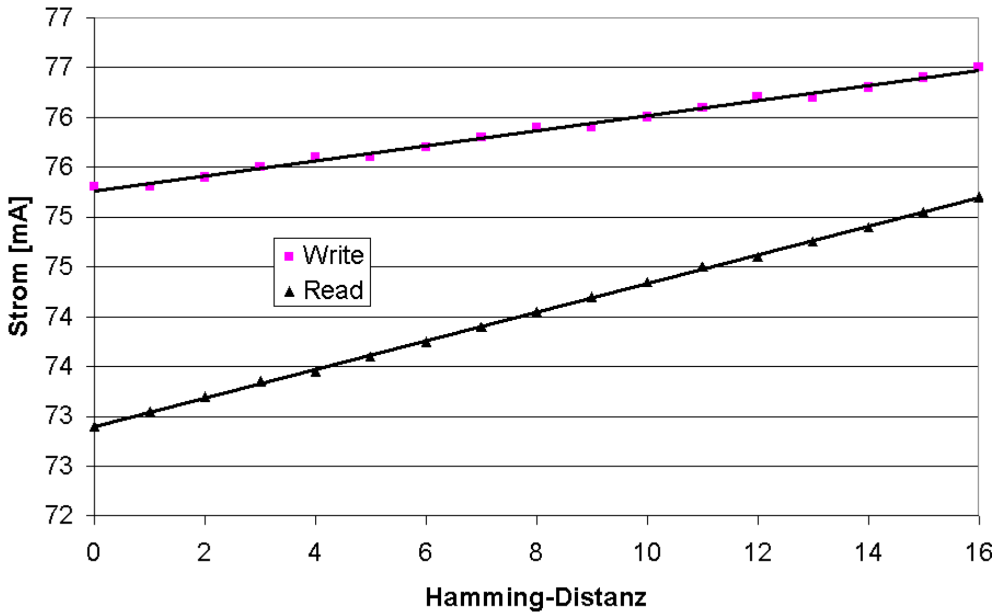


Abbildung 3.14: Speicherstrom in Abhängigkeit von der Hamming-Distanz auf dem Datenbus

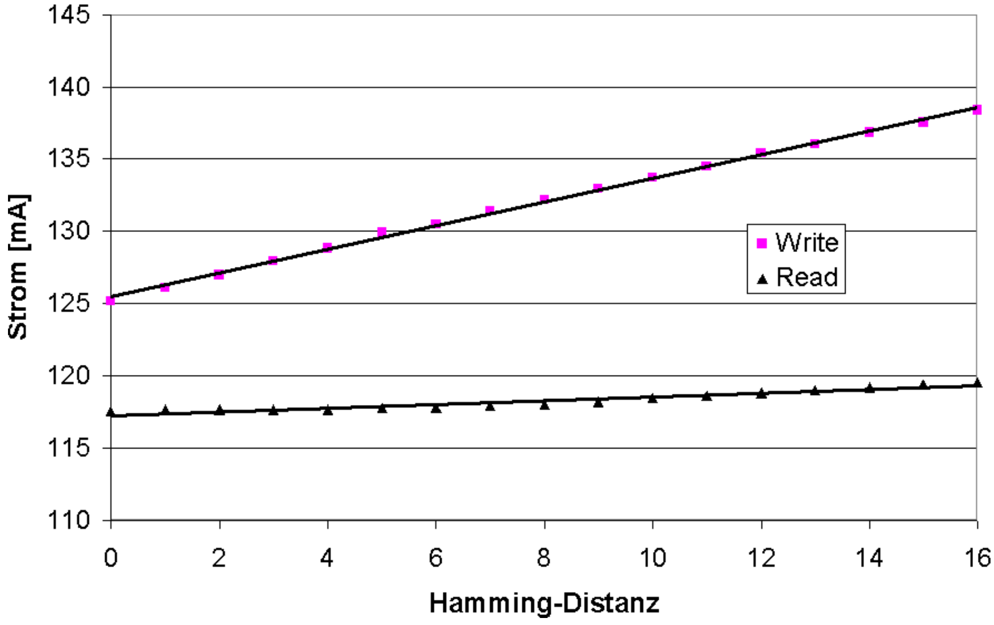


Abbildung 3.15: Gesamtstrom in Abhängigkeit von der Hamming-Distanz auf dem Datenbus

Nach der Betrachtung des Einflusses durch die Anzahl der *Ones* wird in Abbildung 3.13 der Hamming-Abstand auf dem Datenbus dargestellt. Hier kann festgestellt werden, dass bei einem Read-Zugriff keine Abhängigkeit des Energieverbrauchs vom Hamming-Abstand besteht. Bei einem Write-Zugriff steigt der Energieverbrauch mit steigendem Hamming-Abstand. Auch hier wäre die alleinige Betrachtung des Prozessorstroms nicht ausreichend, da man in der Abbildung 3.14 sehen kann, dass bei einem Read-Zugriff insbesondere der Speicherstrom mit der Hamming-Distanz des Datenbusses korreliert. Für den Gesamtstrom ergibt sich dadurch eine leicht positive Korrelation zwischen der Hamming-Distanz und dem Energieverbrauch bei einem Read-Zugriff und eine stärkere positive Korrelation beim Write-Zugriff (Abb. 3.15). Die Hamming-Distanz sollte daher auf dem Datenbus ebenfalls minimiert werden.

Neben dem Datenbus sollen auch die Abhängigkeiten des Prozessorstroms von der Codierung auf dem Adressbus untersucht werden. In Abbildung 3.16 wird der Energieverbrauch des Prozessors in Abhängigkeit von der Anzahl der *Ones* dargestellt. Hier sind die Regressionsgeraden des Read- und des Write-Zugriffs annähernd parallel, da der Adressbus unabhängig von der Datenrichtung arbeitet. Die kleineren Abweichungen in der Steigung ergeben sich durch die zur Messung verwendeten Load- und Store-Befehle, die eine unterschiedliche Zyklenzahl benötigen. Der Versatz zwischen Read und Write hat seine Ursache in anderen Komponenten des Prozessors und nicht in der Adressgenerierungslogik selbst. Leichte Differenzen in der Steigung finden sich hingegen bei der Hamming-Distanz (Abb. 3.17). Insgesamt gibt es aber eine klare positive Korrelation zwischen der Hamming-Distanz und dem Energieverbrauch. Für den Adressbus sollte daher sowohl die Anzahl der *Ones* als auch die Hamming-Distanz minimiert werden, um den Energieverbrauch zu reduzieren.

Bei der Bestimmung der Modellparameter muss die Reihenfolge der Berechnungen berücksichtigt werden. Beispielsweise muss für die Messung der Parameter *BaseCPU* oder *BaseMem* zwangsläufig ein Muster angelegt werden, welches auch *Ones* auf die Busse legt. Sinnvoll ist es daher, zuerst den Einfluss der *Ones* zu bestimmen und dies bei der Bestimmung der Parameter *BaseCPU* und *BaseMem* zu berücksichtigen.

Durch diese Parameter, die für den ARM7TDMI bestimmt wurden (siehe Tab. 3.3), kann auch ermittelt werden, wie viel Einfluss insgesamt durch *Ones*- und Hammingkosten anteilig in dem System aus Prozessor und Speicher anfallen. Jeweils für Read und Write haben die *Ones*-Kosten des Datenbusses einen Anteil von 3,7% und 1,1%, die Hamming-Kosten des Datenbusses hingegen 1,8% und 8%. Da der Adressbus un-

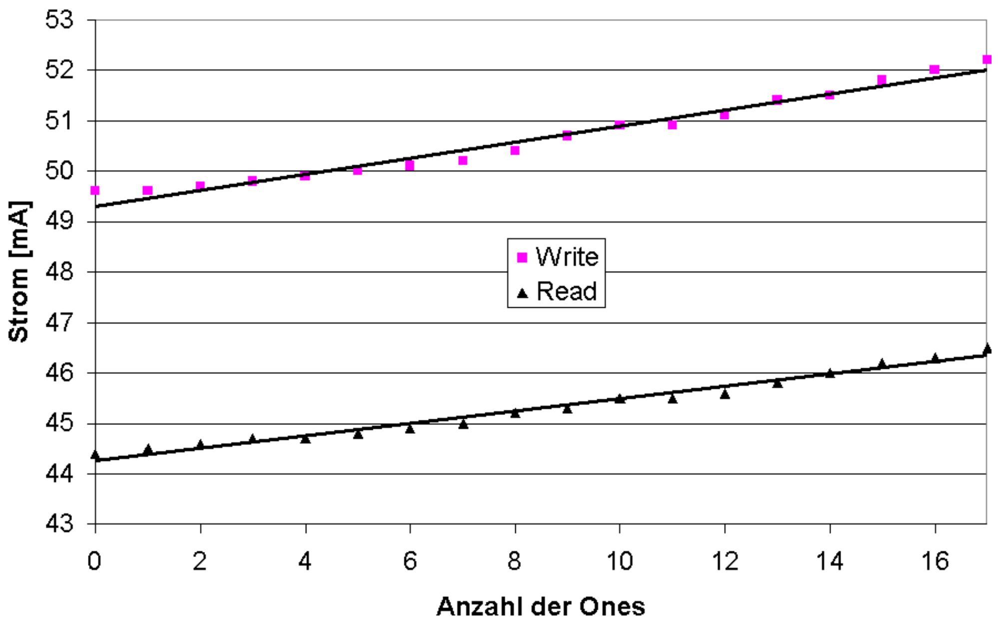


Abbildung 3.16: CPU-Strom in Abhängigkeit von der Anzahl der *Ones* auf dem Adressbus

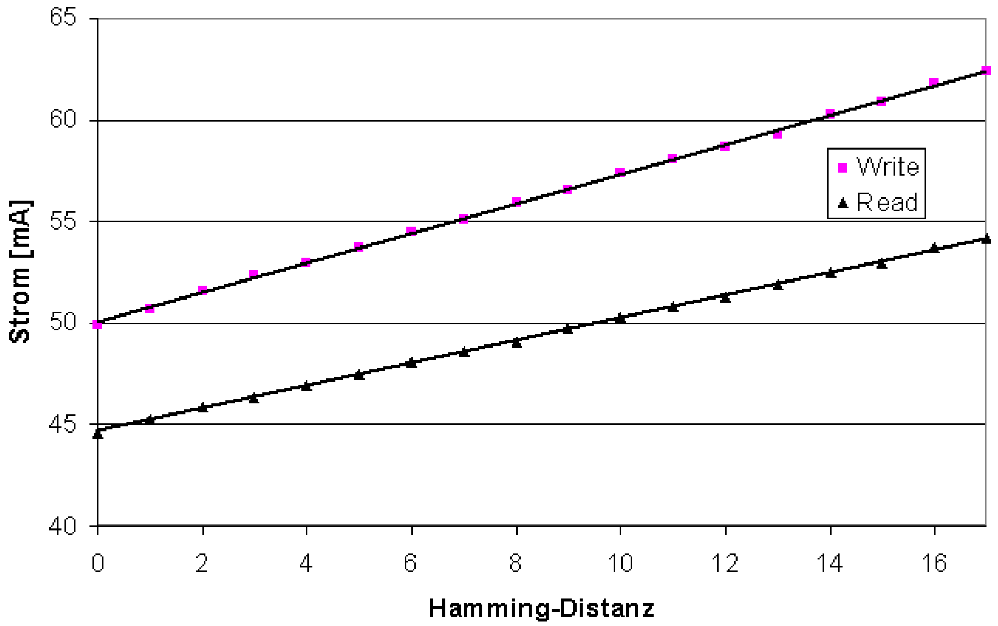


Abbildung 3.17: CPU-Strom in Abhängigkeit von der Anzahl der wechselnden Bits auf dem Adressbus

idirektional arbeitet, werden nur für den Write-Modus Werte zwischen 1% und 12,2% ermittelt. Dieses ist das Potenzial, welches den Codierungs-Optimierungen zur Verfügung steht. Im Energiemodell von Tiwari sind diese Anteile nicht getrennt modelliert, sondern Bestandteil der *Basiskosten*. Eine entsprechende Optimierung auf Basis des Energiemodells von Tiwari ist somit nicht möglich.

Es wurde auch die Genauigkeit des Modells überprüft [The00]. Dafür wurden unterschiedliche kleinere Instruktionssequenzen gemessen, die - wie bereits beschrieben - weniger als 20 Instruktionen umfassten, damit das Messgerät noch konstante Werte anzeigt. Die Sequenzen wurden in der Schleife ausgeführt und der Energieverbrauch bestimmt. Als Vergleichsmaßstab wurde mithilfe des Energiemodells auf Basis der Einzelwerte der Energieverbrauch für die Instruktionssequenz berechnet. Der Vergleich der beiden Ergebnisse zeigte, dass der Fehler mit 1,7% angegeben werden kann. Dies kann als ausreichend genau für den Anwendungsfall angesehen werden.

Parameter	Energie (pJ)		Parameter	Energie (pJ)	
	Read	Write		Read	Write
α_4, α_5	-	48,0	β_4, β_5	-	219,9
$\alpha_{6,dir}$	11,0	26,4	$\beta_{6,dir}$	-5,5	224,1
α_7, α_9	-	-19,2	β_7, β_9	-	138,9
α_8	-115,3	-	β_8	57,7	-
$\alpha_{10,dir}$	-115,3	-60,4	$\beta_{10,dir}$	57,7	22,8

Tabelle 3.3: Parameter des Energiemodells

3.8 Profiling

Nachdem die Parameter für das Energiemodell bestimmt sind, muss nun die Möglichkeit in der Toolkette geschaffen werden, auf der Basis dieses Modells Programme auf ihren Energieverbrauch hin zu bewerten. Einfache Modelle basieren bei einem aktiven Prozessor nur auf dem durchschnittlichen Stromverbrauch [SBM99, RJ98], was durchaus zu guten Ergebnissen für die Berechnung des Durchschnittsverbrauchs größerer Programme führen kann. Wenn jedoch auch Optimierungen beispielsweise der Codierung auf Daten- und Adressbus untersucht werden sollen, müssen Veränderungen der Codierung auch auf den berechneten Energieverbrauch Einfluss haben. Der Standard-ARM-Simulator liefert einen *Trace* aller ausgeführten Instruktionen und Speicherzugriffe sowie die Gesamtanzahl der ausgeführten Befehle und der benötigten Prozessorzyklen. Eine Berechnung des Energieverbrauchs findet nicht statt. Daher muss ein zusätzlicher Profiler - hier *enProfiler* genannt - auf der Basis des Simulationstraces diesen Energieverbrauch berechnen (Abb. 3.18). Die Informationen des Traces mit den ausgeführten Befehlen, Adressen und Daten auf den Bussen sind ausreichend, um eine Auswertung nach dem vorhergehend definierten Energiemodell vorzunehmen. Auf der Basis des aktuell ausgeführten Befehls, der Adresse und des Inhaltes auf dem Datenbus kann der Energieverbrauch des Prozessors berechnet werden. Weiter kann aufgrund der Adresse auf dem Adressbus festgestellt werden, welcher Speicherbaustein angesprochen wird. Dementsprechend kann der Energieverbrauch des Speichers aus den Konfigurationsdaten des Profilers ermittelt werden.

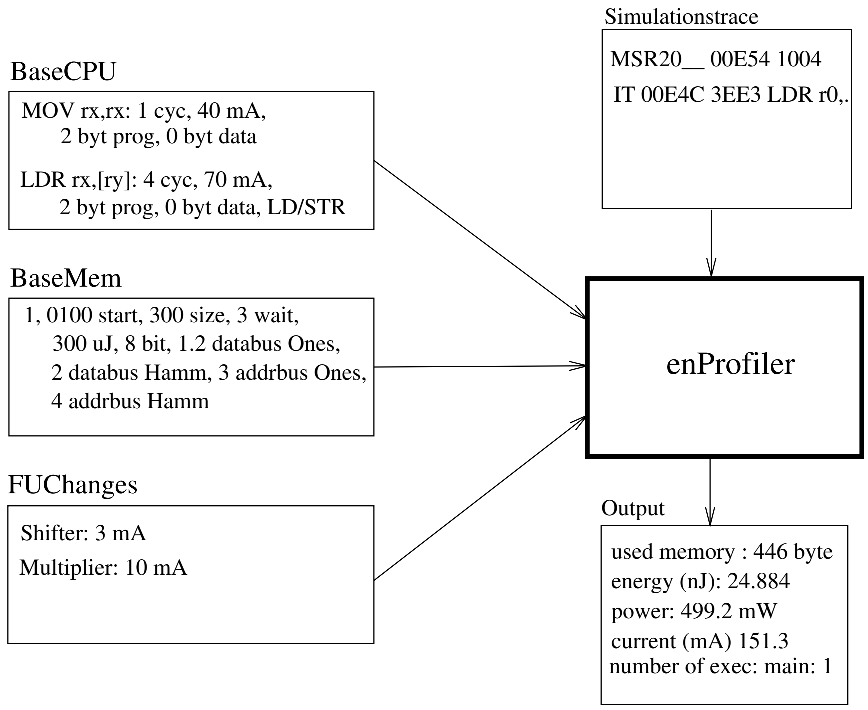


Abbildung 3.18: Profiler mit Ein-/ Ausgabedaten

Der Profiler erhält die Kennwerte des Energiemodells aus insgesamt drei Konfigurationsdateien:

1. Charakteristik der Prozessorbefehle

   In dieser Datei werden für jeden Prozessorbefehl Op die davon abhängigen Informationen definiert: die Anzahl der notwendigen Zyklen für die Befehlsausführung $Cycle$, der Basisanteil des Energieverbrauchs $BaseCPU$, die Größe der Instruktion $InstrSize$, die Größe eines möglichen Datentransfers $DataSize$ und eine Liste mit aktiven Funktionseinheiten.

   $$\forall Op \in InstrSet : \{Cycle, BaseCPU, InstrSize, DataSize, \{UsedFU\}\}$$

2. Charakteristik der Speicher

   In einer weiteren Konfigurationsdatei werden die Daten der verschiedenen Speicher mem im System jeweils für die unterschiedlichen Wortbreiten und Datenrichtungen angegeben: der Speicher hat die Größe $size$ und liegt im Adressbereich ab Adresse $start$ mit der Datenrichtung dir und der Wortbreite $width$. Bei dem Zugriff mit $wait$ Waitstates wird die Energie $BaseMem$ verbraucht, mit den zusätzlichen Parametern für die Anzahl der *Ones*, $data_1$, $addr_1$ und der Hamming-Distanz $data_h$, $addr_h$ auf dem Daten- und Adressbus.

   $$\forall mem_{width,dir} : \{start, size, wait, BaseMem, data_1, data_h, addr_1, addr_h\}$$

   Es ist darauf hinzuweisen, dass die Parameter $data_1$, $data_h$, $addr_1$und $addr_h$ sowohl von dem Prozessor als auch dem Speicher abhängig sind. Da die Parameter stets für einen spezifischen Prozessor bestimmt werden, ist es ausreichend, diese bei den verschiedenen Speichertypen zu annotieren.

3. Charakteristik der Funktionseinheiten

Da die Kosten für das Aktivieren oder Deaktivieren den Funktionseinheiten zugeordnet werden, enthält die letzte Konfigurationsdatei einen Eintrag für jede Funktionseinheit FU mit ihrem Aktivierungs- bzw. Deaktivierungsenergiebedarf $FUChange$:

$$\forall FU : \{FUChange\}$$

Auf der Basis dieser Konfiguration und des Simulationstraces werden durch den Profiler die folgenden Auswertungen vorgenommen:

- Gesamtanzahl der Prozessorzyklen
- Energieverbrauch aufgrund von *Ones* bzw. der Hamming-Distanz auf Daten- bzw. Adressbus für jeden Speichertyp
- Anzahl der Zugriffe auf einen Speicher mit Wortbreite und Richtung
- Gesamtenergieverbrauch im Speicher durch Holen von Instruktionen bzw. durch Datenzugriffe
- maximaler Bedarf an Speicher auf dem Stack
- absolute Aufrufhäufigkeit jeder Funktion und der dort anfallende Energieverbrauch
- absolute Anzahl von Ausführungen jedes Basisblocks und der jeweilige Energieverbrauch

Da für einige Optimierungen die Anzahl der Durchläufe von Basisblöcken relevant ist, muss diese entweder im Compiler auf der Basis einer statischen Analyse berechnet oder durch die Auswertung eines Programmlaufs mit Test-Daten bestimmt werden. Letzteres übernimmt der Profiler auf Basis des Simulationstraces und ermittelt die exakten Anzahlen von Durchläufen der Basisblöcke, die allerdings von den Eingabedaten abhängig sind. Um diese Ausführungshäufigkeit der Basisblöcke bei der Codegenerierung zu berücksichtigen, muss nach dem Durchlauf durch den Profiler das Back-End des Compilers erneut durchlaufen werden. Insgesamt ergibt sich die bereits in Abbildung 2.15 dargestellte Abfolge von Tools.

Der Profiler wurde als eigenständige Applikation entwickelt, enthält aber Komponenten, die ebenso im Compiler verwendet werden. Über entsprechende Kostenfunktionen können für einen Befehl die zugehörigen Energiekosten ermittelt werden. Nicht berücksichtigt werden beim Aufruf während des Compilerlaufes die Busse, da die Adressen noch nicht zugeordnet sind, ebenso wie die Muster auf dem Datenbus, die erst nach dem Assembleraufruf bekannt sind. Es ist daher bei Optimierungen stets zu prüfen, ob Einflüsse durch die Busse berücksichtigt werden müssen. Ist dies, wie beispielsweise bei der Optimierung der Codierung, der Fall, können durch entsprechende Aufrufe der Komponenten Kosten für einzelne Adressen oder Folgen von Adressen ermittelt werden. Eine beispielhafte Ausgabe des Profilers ist in Tabelle 3.4 dargestellt.

3.9 Zusammenfassung

In diesem Kapitel wurden die Anforderungen (hinreichende Genauigkeit der Energieabschätzung, Wiederverwendbarkeit, Modellierung der Art von Maschineninstruktionen, Scheduling von Instruktionen, Speicherhierarchie, Bitwechsel auf Bussen) an ein Energiemodell zur Unterstützung eines Compilers bei der Optimierung des Energieverbrauchs definiert. Auf dieser Grundlage wurden bekannte Energiemodelle auf

```
# Memory areas :
|  Area |      Start - End | Size(Byte) | Accesses | Energy(µJ) |
|-------|------------------|------------|----------|------------|
| Stack | 57ffcc - 57ffdf |         20 |        6 |      0.255 |

# Memory size :
|         Memoryunit |      Inst |      Data |
|--------------------|-----------|-----------|
|     offchip (Byte) |        44 |        28 |

# function/basic block accesses and energy :
| Type | Name                | Address |      Acc | Energy (µJ) |
|------|---------------------|---------|----------|-------------|
| F    | main                |  500000 |        1 |     0.59203 |
| BB   | main                |  500000 |        1 |     0.12731 |
| BB   | LL3_0               |  500012 |        4 |     0.42996 |
# basic block access scheme :
| Source - BB            --> Destination - BB         | Count     |
|-------------------------->-------------------------|-----------|
| main                   --> LL3_0                    |         1 |
| LL3_0                  --> LL3_0                    |         3 |
| LL3_0                  --> _M_4                     |         1 |

# Memory accesses :
| Memoryunit |     Acc | Width |    Inst |    Data | Energy (nJ) |
|------------|---------|-------|---------|---------|-------------|
| offchip    | read    | 2 Byte |      48 |       0 |        24.0 |
| offchip    | read    | 4 Byte |       0 |       1 |        49.3 |
| offchip    | write   | 4 Byte |       0 |       5 |        41.1 |
|------------|-----------------|---------|---------|-------------|
| SUM        | access  |       |      48 |      24 |             |

# CPU-Cycles :  129
# Energy values :
|              | SUM             | Instruction     | Memory          |
|--------------|-----------------|-----------------|-----------------|
| Energy (µJ)  | 1.999           | 0.592           | 1.407           |
| Power (mW)   | 511.3           | 151.4           | 359.9           |
| Current (mA) | 154.9           | 45.9            | 109.1           |

# Function unit-costs current :  40.9 mA
# Hamming distance- & Ones-costs :
|            | Hamming distance costs | Ones costs              |
|Memory type | Data bus      Addr bus | Data bus      Addr bus |
|------------|------------------------|-------------------------|
| 4 (mA)     |       71.6        18.8 |       48.6        40.8 |
| 5 (mA)     |        0.2         4.8 |        1.2        20.4 |
```

Tabelle 3.4: Ausgabe des Profilers

Instruktionsebene bewertet und als Ergebnis festgestellt, dass keines der verfügbaren Modelle alle Anforderungen erfüllt. Aus diesem Grunde wurde ein neues Modell entwickelt und vorgestellt, welches als Grundlage für energieoptimierende Compiler dienen kann. Es ist, ohne interne Kenntnisse der Schaltung zu besitzen, allgemein auf RISC-Prozessoren anwendbar. Ein Messverfahren, welches detailliert beschrieben wurde, ermöglicht die Bestimmung der notwendigen Parameter des Modells mithilfe eines einfachen Amperemeters. Das Modell bietet durch die Berücksichtigung von Speicherkomponenten und Codierungen auf den Bussen ein größeres Potenzial für Energieverbesserungen als das bekannte Modell von Tiwari.

Um das Energiemodell zur Bewertung von Programmen auf ihren Energieverbrauch zu verwenden, muss das Modell in die Software-Entwicklungsumgebung integriert werden, um die verschiedenen Energieoptimierungen des Compilers zu bewerten. Für diese Aufgabe wurde im letzten Unterkapitel der *enProfiler* vorgestellt.

Es können nun auf dieser Basis verschiedene Optimierungen im Compiler implementiert und bewertet werden. Die notwendige Infrastruktur und der Zusammenhang zur physikalischen Ebene wurden hiermit geschaffen.

Kapitel 4

Speicher

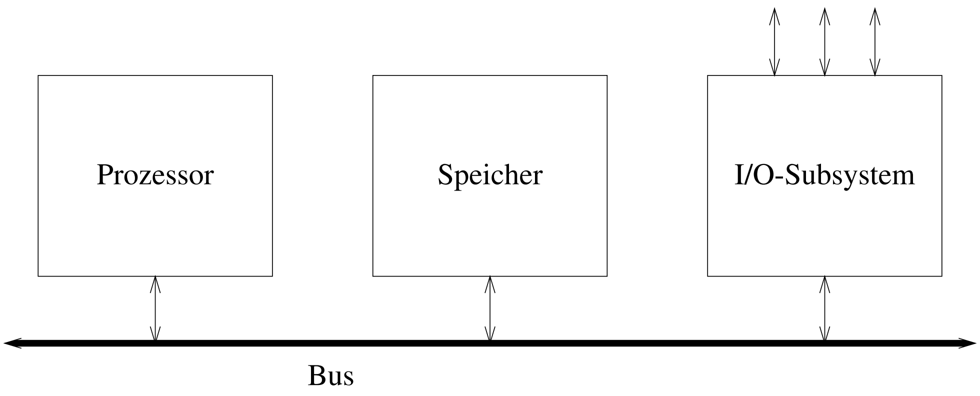


Abbildung 4.1: Speicher in der Systemarchitektur

Auf der Grundlage des im vorherigen Kapitel vorgestellten Energiemodells können nun die unterschiedlichen Einheiten eines Computersystems betrachtet und bewertet werden. Dadurch können Ansatzpunkte für Energieoptimierungen ermittelt und anschließend im Detail evaluiert werden. Das Ziel ist die Entwicklung von Energieoptimierungen zur Integration in den Compiler.

Allgemein besteht ein Computersystem aus den in Abbildung 4.1 dargestellten drei Komponenten [Mar00, Prz90]. Diese sind:

1. der Prozessor, der als Mittelpunkt für die Steuerung des Ablaufs dient und die Daten verarbeitet,
2. der Speicher, aus dem Instruktionen geholt und in dem Daten sowohl gelesen als auch geschrieben werden können und
3. das I/O-Subsystem, welches zur Kommunikation mit der Außenwelt dient.

Für die Ausführung des Programms werden kontinuierlich Instruktionen aus dem Speicher in den Prozessor geholt. Die Abarbeitung der Instruktionen bedingt eventuell noch das Holen von Eingangsdaten aus dem Speicher bevor die Verarbeitung der Daten ausgeführt werden kann. Anschließend werden gegebenenfalls die Ergebnisdaten im Speicher abgelegt. Die Ausführungszeit des Programms, die auch wesentlich für den Energieverbrauch verantwortlich ist, hängt von der Anzahl der auszuführenden Instruktionen, der Taktfrequenz, der mittleren Anzahl von Taktzyklen je Instruktionsausführung (CPI) und den mittleren Zugriffszeiten auf den Speicher oder das I/O-Subsystem ab.

Aufgrund der beschränkten Anzahl von Registern im Registerspeicher des Prozessors hat der Speicher des Computersystems eine große Bedeutung für die Geschwindigkeit und den Energieverbrauch. Ideal wäre, wenn der Speicher ausreichend groß und die Zugriffsgeschwindigkeit größer als die Verarbeitungsgeschwindigkeit der Daten durch den Prozessor wäre. Leider ist aufgrund der physikalischen Gegebenheiten bei größeren Speichern der Zugriff langsamer und der Energieverbrauch um ein Vielfaches größer als bei kleineren Speichern. Die besondere Notwendigkeit von Optimierungen im Bereich der Speicherhierarchie zeigen die unterschiedlichen Steigerungsraten der Prozessoren und Speicher. Während die Leistung von Prozessoren um 50% bis 100% pro Jahr wächst, ist bei den DRAM-Bausteinen nur eine Steigerung von unter 10% pro Jahr zu verzeichnen. Aus diesen unterschiedlichen Raten ergibt sich die so genannte *CPU-DRAM-Lücke*, die jährlich um ca. 50% ansteigt [HP90]. In Kapitel 4.1 wird erläutert, wie eine Kombination von unterschiedlich großen und schnellen Speichern in einer so genannten *Speicherhierarchie* die Gesamtgeschwindigkeit bei der Ausführung einer Applikation erhöhen und den Energieverbrauch reduzieren kann.

Für die Abarbeitung der jeweiligen Instruktionen sind Daten im Prozessor bereitzustellen. Entweder können diese direkt aus den Registern des Prozessors geholt oder aber aus dem Speicher des Computersystems angefordert werden. Daten, die über einen längeren Zeitraum vorgehalten werden müssen, werden stets im Speicher abgelegt, da der Speicherplatz im Prozessor in Form von Registern dafür i.d.R. nicht ausreichend dimensioniert ist. Daten mit kürzerer Lebensdauer werden bevorzugt in den Prozessorregistern gehalten, sofern genügend Register über den jeweiligen Zeitraum zur Verfügung stehen. In Kapitel 4.2 wird eine Compiler-Technik vorgestellt, die die Datenhaltung in Registern optimiert und dafür Zugriffe auf den Speicher reduzieren kann. Trotz einer Erhöhung der Anzahl ausgeführter Instruktionen kann durch die eingesparten Speicherzugriffe insgesamt der Energiebedarf reduziert werden.

Der Aufbau einer Speicherhierarchie ermöglicht darüber hinaus noch weitere Freiheitsgrade für die Optimierung durch einen Compiler. Daher wird als zweite wesentliche Optimierung in Kapitel 5 ein schneller, kleiner und frei adressierbarer Onchip-Speicher genutzt, um Hauptspeicherzugriffe durch kostengünstigere Onchip-Speicherzugriffe zu ersetzen. Dadurch können hohe Energieeinsparungen erzielt werden. Auch im Gegensatz zu weit verbreiteten Speicherhierarchien auf der Basis von Caches ist diese neu vorgestellte Methode vorteilhaft. Die Optimierung, die Speicherobjekte auf Hauptspeicher und Onchip-Speicher während des Compilierungsvorganges verteilt, wird in einer festen, statischen Belegung des Onchip-Speichers und alternativ in einer dynamischen Variante, die den regelmäßigen Austausch von Programmteilen während der Programmabarbeitung beinhaltet, vorgestellt.

Zur Einführung in die Thematik der Speicher werden verschiedene Speicherklassen sowie das Prinzip von Speicherhierachien erläutert.

4.1 Speicherhierarchie

Um den langsamen Hauptspeicher durch weitere schnellere Speicher zur Erhöhung der Geschwindigkeit und zur Reduktion des Energieverbrauchs zu ergänzen, soll zu Beginn ein Überblick über verfügbare Speichertypen gegeben werden. Grundsätzlich kann Speicher - wobei hier die Beschränkung auf Halbleiterspeicher gilt - nach den folgenden Kriterien, die in der Übersicht in Abbildung 4.2 dargestellt sind [Mar00, Hu01], klassifiziert werden:

1. Festwert- vs. Schreib/Lese-Speicher
   In einem Festwert-Speicher können nur Daten gehalten werden, die keiner Veränderung unterliegen. Neben dem eigentlichen Programm, welches sich nicht selbst verändern darf, sind dies konstante Daten, die nur als Eingabewerte dienen. Der Vorteil eines Festwert-Speichers ist insbesondere die kleinere und kostengünstigere Bauform, die schnellere und energiesparendere Zugriffe erlaubt.

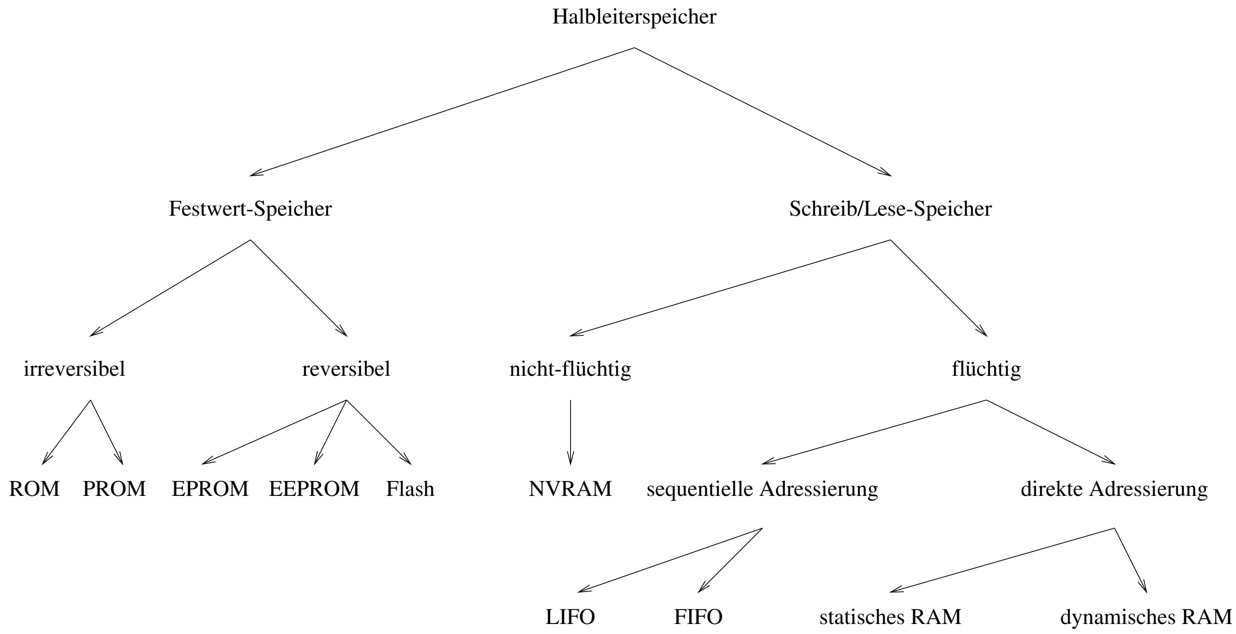


Abbildung 4.2: Klassifizierung von Speicherkomponenten [Mar00]

Schreib/Lese-Speicher sind im Gegensatz dazu bezogen auf diese Eigenschaften im Nachteil, dafür jedoch universell einsetzbar.

2. irreversibel vs. reversibel
   Die Klasse der Festwert-Speicher muss nochmals unterteilt werden. Irreversible Bausteine können für Inhalte eingesetzt werden, die mit sehr hoher Wahrscheinlichkeit nicht mehr geändert werden müssen (ROM, PROM). Tritt dieser Fall dennoch auf, muss der Baustein ausgetauscht werden. Reversible Bausteine könnnen bei gelegentlichen Änderungen des Inhalts, z. B. für Update-Zwecke des Programmes, geändert werden (EPROM, EEPROM).

3. flüchtiger vs. nicht-flüchtiger Speicher
   Flüchtige Speicher, wie beispielsweise dynamische RAMs (DRAM), die als Hauptspeicher eingesetzt werden, verlieren ihren Inhalt nach dem Ausschalten des Systems (= Abschalten der Versorgungsspannung des Speichers). Für die Speicherung von Informationen, die auch nach dem Neustart eines Systems wieder benötigt werden, muss dafür auf nicht-flüchtige Speicher, wie Festplatten oder batteriegepufferte RAM-Bausteine (NVRAM), zurückgegriffen werden.

4. direkte vs. sequentielle Adressierung
   Während die häufigste Art von Speichern einen direkten und wahlfreien Zugriff erlaubt, sind andere Speicherorganisationen auf sequentielle Zugriffe beschränkt. Insbesondere die First-In-First-Out (FIFO) Zugriffsmethode, die einer Queue entspricht, und die Last-In-First-Out (LIFO) Methode, die einen Stack realisiert, werden in der Praxis eingesetzt.

5. statisches vs. dynamisches RAM
   Für die Zugriffe durch den Prozessor sind Bauarten mit statischen und dynamischen Speicherzellen identisch zu behandeln. Intern jedoch erfordern statische RAMs pro Speicherzelle sechs Transistoren gegenüber einem einzigen Transistor bei dynamischen RAMs. Letztere sind platzsparender und stromsparender, erfordern jedoch eine aufwendige Refresh-Logik und ein internes Wiederbeschreiben beim Lesen von Zellen. Außerdem sind sie langsamer als die statischen Speicher.

Aus diesen vorhandenen Möglichkeiten können jetzt verschiedene Bausteine zur Realisierung einer Speicherhierarchie kombiniert und auch weitere spezielle Eigenschaften eines Speichertyps wie z. B. "Power-Down" Modi von DRAM-Speichern ausgenutzt werden. Diese Hardwarefunktionen erlauben z. B. die Ausnutzung einer Optimierung des Datenlayouts, indem ein Teil der Speicher vorübergehend in einen "Power-Down" Modus heruntergeschaltet werden kann und dadurch der Energieverbrauch signifikant reduziert wird [DKV$^+$01]. Weitere Freiheitsgrade liefern neue Technologien wie Embedded DRAM, womit auch größere dynamische Speicher direkt auf dem Prozessorchip realisiert werden können. Die Schwierigkeit liegt hier in der Notwendigkeit eines gemeinsamen Herstellungsprozesses für Prozessor und Onchip-Speicher. Durch Embedded DRAM sind größere Onchip-Speicher realisierbar, die Energieeinsparungen bis zum Faktor 10 erreichen können [WH98].

Vor der Auswahl der Bestandteile der Speicherhierarchie werden im Folgenden die theoretischen Grundlagen diskutiert.

Lokalität

Die Grundlage einer Hierarchie und die Basis für die Optimierung ist das Prinzip der Lokalität [HP90]. Dieses Prinzip besagt, dass Programme einen Teil ihres Adressraumes aufgrund der folgenden Eigenschaften häufiger nutzen:

- *zeitliche Lokalität*: wenn auf einen Eintrag im Speicher zugegriffen wurde, erfolgt mit hoher Wahrscheinlichkeit bald ein erneuter Zugriff auf ihn,
- *räumliche Lokalität*: wenn auf einen Eintrag zugegriffen wurde, erfolgt mit hoher Wahrscheinlichkeit bald ein Zugriff auf einen benachbarten Eintrag.

Da kleinere Speicher schneller und energiesparender sind, kann man sich die zeitliche Lokalität zunutze machen, indem verwendete Daten vorübergehend dort zwischengespeichert werden. Mit hoher Wahrscheinlichkeit wird nach einem ersten Zugriff auf ein Datum, der die Zwischenspeicherung im kleineren Speicher initiiert, bald erneut auf dieses Datum zugegriffen und dadurch anstatt eines langsamen und energieintensiven Zugriffs auf den großen Speicher ein Zugriff auf den sparsamen Zwischenspeicher ausgeführt. Die zusätzlichen Kosten, die für das Zwischenspeichern im kleineren Speicher entstehen, müssen durch die folgenden sparsameren Zugriffe auf den Zwischenspeicher wieder eingespart werden. Wären die Zugriffe über alle Speicheradressen zeitlich gleichverteilt, sodass es also keine zeitliche Lokalität gäbe, könnte der kleinere, energiesparende Speicher aufgrund der begrenzten Kapazität nie effizient genutzt werden und es würde insgesamt sogar ein höherer Energieverbrauch aufgrund der Aufwendungen für das Zwischenspeichern entstehen.

Die räumliche Lokalität kann genutzt werden, indem nicht nur ein einzelnes Datum aus dem großen Speicher in den kleineren Speicher kopiert, sondern dies gleich für mehrere nebeneinander liegende Daten ausgeführt wird. Dies ist auch deswegen profitabel, da zusammenhängende Adressen hintereinander schneller gelesen werden können als wenn dies durch einzelne Zugriffe erfolgt. Daher werden mehrere aufeinander folgende Daten zu so genannten *Blocks* zusammengefasst, die gemeinsam in der Speicherhierarchie verwaltet werden. Auch hier kann ein Nutzen nur eintreten, wenn nach dem Zugriff auf eine Adresse deren benachbarte Adressen auch mit höherer Wahrscheinlichkeit verwendet werden.

Funktionsweise von Speicherhierarchien

Das Prinzip der Lokalität wird nun ausgenutzt, um eine Speicherhierarchie bestehend aus Bausteinen mit unterschiedlich großer Kapazität, unterschiedlicher Zugriffsgeschwindigkeit und unterschiedlichen Energieverbräuchen aufzubauen. Eine Hierarchie wird mit dem Prozessor beginnend bis zum Hauptspeicher[1]

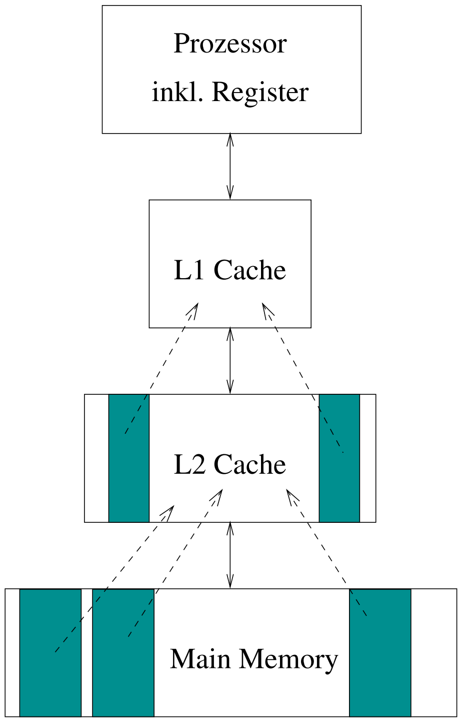


Abbildung 4.3: Speicherhierarchie

(Abb. 4.3) so aufgebaut, dass die Speicher mit zunehmender Entfernung vom Prozessor größer, langsamer, aber auch energiehungriger werden. In der gleichen Richtung werden andererseits die Anschaffungskosten pro Speicher-Bit geringer. Eine weitere Eigenschaft ist, dass man die Daten einer Speicherebene auch immer in allen darunter liegenden Ebenen findet, sodass eine Ebene jeweils einer Kopie eines Teils der darunter liegenden Ebene entspricht.

Wenn ein Datum vom Prozessor angefordert wird, beginnt die Suche zuerst in der nächsttieferen Ebene, dem First-Level-Cache (L1-Cache)[2]. Aufgrund des Prinzips der räumlichen und zeitlichen Lokalität wird das benötigte Datum mit einer höheren Wahrscheinlichkeit gefunden als es dem Verhältnis zwischen der Größe dieser Speicherebene und der Gesamtspeichergröße entspricht. Falls das Datum auf dieser Ebene nicht gefunden wird, wird die Anforderung an die nächsttiefere Ebene, den Second-Level-Cache (L2-Cache), weitergereicht. Dieses Durchreichen der Anforderung von Daten wird gegebenenfalls bis zum Hauptspeicher fortgeführt. Danach wird in allen darüber liegenden Ebenen der Block, zu dem das angeforderte Datum gehört, in den Speicher kopiert.

Wie man aus diesem beschriebenen Mechanismus ersehen kann, schickt der Prozessor über seinen Adressbus die Adresse des benötigten Datums, ohne zu wissen, in welcher Speicherebene dieses liegt. Die Speicherhierarchie bzw. die einzelnen Speicherebenen müssen selbst entscheiden können, ob das Datum in ihnen enthalten ist und, falls nicht, die Anforderung automatisch an die nächsttiefere Ebene weiterleiten. Dieser Mechanismus wird von Caches selbständig in ihrem Cache-Controller ausgeführt, indem anhand der Adresse überprüft wird, ob er den Inhalt dieser Adresse gespeichert hat. Dafür existiert neben dem eigentlichen Datenspeicher noch ein so genannter Tagspeicher, in dem die Adressen zu den Einträgen im Datenspeicher verwaltet werden. Im Detail werden Caches im Kapitel 4.3 vorgestellt.

Der Erfolg oder Misserfolg eines Cache-Zugriffs wird als Treffer (*hit*) oder Fehlzugriff (*miss*) bezeichnet. Die Trefferrate r_{hit} (*hit rate*) kennzeichnet den erfolgreichen Anteil der Zugriffe auf der ersten Ebene

[1] weitere Speicherebenen wie Festplatte oder Bänder werden hier nicht betrachtet.

[2] sofern dieser vorhanden ist.

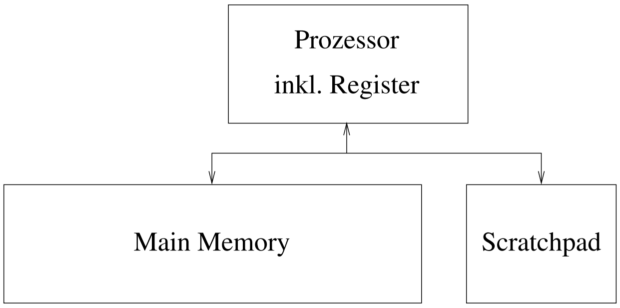


Abbildung 4.4: Hauptspeicher und Scratchpad

relativ zu der Anzahl der Gesamtzugriffe und die Fehlzugriffsrate $r_{miss,i}$ (*miss rate*) entsprechend die relative Häufigkeit der Fehlzugriffe auf der Ebene i. Um nun die Geschwindigkeit zu bestimmen, die durch die Einführung der Speicherhierarchie gesteigert werden sollte, muss noch die Trefferzeit t_{hit} (*hit time*) eingeführt werden. Dies ist die Zeit für den Zugriff auf die erste Ebene der Hierarchie einschließlich der Zeit für die Entscheidung, ob es sich um einen Treffer oder einen Fehlzugriff handelt. Entsprechend ist die Fehlzugriffszeit $t_{miss,i}$ die Zeit für die Entscheidung, ob das gewünschte Datum in der Ebene $i+1$ vorhanden ist zuzüglich der Zeit zum Ersetzen eines Blockes in der Ebene i mit einem Block aus der Ebene $i+1$. Mit diesen Parametern können wir nun die mittlere Speicherzugriffszeit $t_{mem,avg}$ für eine Speicherhierarchie mit n Ebenen definieren:

$$t_{mem,avg} = t_{hit} + \sum_{i=1}^{n-1} \prod_{j=1}^{i} r_{miss,j} * t_{miss,i}$$

Optimierung durch vertikale Aufteilung von Speicherebenen

Beim Entwurf einer Speicherhierarchie ist es das Ziel, diese mittlere Speicherzugriffszeit zu minimieren. Randbedingungen sind hierfür die Anzahl und Größe der Speicherebenen sowie weitere Parameter wie z. B. die Cache-Organisation. Diese Randbedingungen beeinflussen den Platzbedarf und die Kosten des Systems. Die zugehörige komplexe Berechnung des Energieverbrauchs wird im späteren Kapitel 4.3 vorgestellt.

Alternativ zur Einführung von Ebenen in Form einer Speicherhierarchie kann eine Speicherebene auch horizontal unterteilt werden (Abb. 4.4). Beispielsweise können in getrennten unterschiedlich großen Adressbereichen verschiedene Speicherarten eingesetzt werden. Ein kleinerer Speicher, der als Onchip-Speicher auch Scratchpad genannt wird, benötigt für einen Zugriff bei einer konstanten Zugriffszeit t_{SP} im Allgemeinen einen geringeren Energieverbrauch. Der große Speicher auf der selben Ebene in einem anderen Adressbereich benötigt jedoch die Zugriffszeit t_{main}. Bei einer statistischen Gleichverteilung des Zugriffs ergibt sich daraus eine Zugriffsrate r_{SP} (*scratchpad hit rate*) auf den Scratchpad mit der Größe $size_{SP}$ von:

$$r_{SP} = \frac{size_{SP}}{size_{SP} + size_{main}}$$

Es ergibt sich daraus die folgende durchschnittliche Speicherzugriffszeit:

$$t_{mem,avg} = (1 - r_{SP}) * t_{main} + r_{SP} * t_{SP}$$

Wenn durch eine geschickte Auswahl von Daten für den Scratchpad und dank des Prinzips der Lokalität mehr Zugriffe auf diesen kleineren Speicher erfolgen, ist die Scratchpad Zugriffsrate r_{SP} entsprechend größer. Dieses Prinzip der Anordnung von Speichern wird in Kapitel 5 verwendet, um häufig verwendete Programmteile und Daten in dem kleineren Speicher zu positionieren und dadurch insgesamt die durschnittliche Zugriffszeit $t_{mem,avg}$ und den Energieverbrauch zu reduzieren.

Nachdem nun die Grundlagen einer Speicherhierarchie mit der Unterteilung der vertikalen Dimension durch Caches und der horizontalen Dimension durch Scratchpad-Speicher vorgestellt wurden, sollen in den nachfolgenden Untersuchungen diese verschiedenen Speicherhierarchien durch entsprechende Unterstützung im Compiler zur Optimierung des Energieverbrauchs ausgenutzt werden. Vorab wird jedoch gezeigt, wie durch eine Compiler-Technik schon die Anzahl der Speicherzugriffe insgesamt reduziert werden kann, indem die Prozessorregister effizienter belegt werden.

4.2 Nutzung der Register

Neben dem Ansatz der Verlagerung von Speicherzugriffen vom langsamen und energieintensiven Hauptspeicher auf den schnelleren und energiesparsameren Scratchpad-Speicher, kann man auf der prozessornahen Ebene der Speicherhierarchie auch versuchen, Speicherzugriffe durch die effizientere Nutzung der Register zu ersetzen. Die hier beschriebene Technik *Registerpipelining (RP)* nutzt temporär verfügbare Register aus, um Speicherzugriffe vom Prozessor auf einen Onchip- oder Offchip-Speicher einzusparen und somit den Energieverbrauch der Applikation zu reduzieren.

Nach der Präsentation eines Beispiels für die Anwendung des RP wird im zweiten Abschnitt die grundsätzliche Funktionsweise vorgestellt und bisher veröffentlichte Arbeiten benannt. Im dritten Abschnitt werden detailliert die Techniken zur Analyse des Programms beschrieben, um Informationen zur optimalen Anwendung der Technik zu erhalten. Auf dieser Basis kann dann das RP, falls es Verbesserungen bezüglich des Optimierungsziels des Compilers liefert, angewendet werden. Die Vorstellung der Ergebnisse bezüglich des Energieverbrauchs schließen das Kapitel ab.

Beispiel Registerpipelining

```
for (i = 3; i < 120; i++) {
  b[i] = a[i] + a[i-3] + 5;
}
```

(a) vorher

```
R0 = a[0];
R1 = a[1];
R2 = a[2];
R3 = a[3];
for (i = 3; i < 120; i++) {
  R3 = a[i];
  b[i] = R3 + R0 + 5;
  R0 = R1;
  R1 = R2;
  R2 = R3;
}
```

(b) nachher

Abbildung 4.5: Anwendung des Registerpipelinings

Zur Veranschaulichung der Arbeitsweise von RP ist in Abbildung 4.5 ein Programm in C-Notation vor und nach der Anwendung von RP dargestellt. Dieses entspricht jeweils einem Maschinensprachenprogramm, wobei die Variablen `R0` bis `R3` Prozessorregister darstellen. Die C-Anweisungen stehen stellvertretend für Maschinenbefehle, um ein einfacheres Verständnis zu erzielen.

Das dargestellte Programm bearbeitet die Elemente eines Arrays `a` der Reihe nach bildet daraus die Inhalte des Arrays `b`. Bei dieser Implementierung werden auf das Array `a` insgesamt 2 * 117 Leseoperationen ausgeführt. Da die Arrays aufgrund der beschränkten Registeranzahl im Prozessor im Datenspeicher abgelegt ist, werden dementsprechend 2*117 Datenspeicher-Lesezugriffe ausgeführt. Falls noch 4 Prozessorregister innerhalb dieser Schleife zur Verfügung stehen, können diese Register nun genutzt werden, um durch eine modifizierte Implementierung mithilfe der Technik RP die Anzahl der Zugriffe zu verringern. In Abbildung 4.5b werden initial die ersten Elemente des Arrays `a` vor der Schleife in die Prozessorregister `R0` bis `R3` geladen. Innerhalb der Schleife enthalten diese Register dann jeweils die aktuellen Arrayelemente `a[i]` bis `a[i-3]`. Die Berechnung des Arrayelements `b[i]` benötigt dann nur noch eine Speicherleseoperation, um das Element `a[i]` aus dem Speicher zu laden. Am Ende der Schleife werden wegen der Inkrementierung von `i` in den Registern `R0` bis `R3` der Reihe nach die Arrayelemente durchgeschoben. Die Register bilden hier die so genannte *Registerpipeline*, die der Technik den Namen gegeben hat.

Die Anwendung der Technik hat die Anzahl der vorher insgesamt 234 Datenspeicherzugriffe auf 4+117 = 121 Datenspeicher-Lesezugriffe (=52%) reduzieren können. Dafür sind jedoch zusätzliche Befehle notwendig, um die Register initial zu laden und die Werte in der Registerpipeline durchzuschieben. Es hängt nun von mehreren Faktoren ab, ob die Anwendung des RP einen Vorteil liefert. Insbesondere für die Energieoptimierung ist dies häufig der Fall, da die eingesparten Hauptspeicherzugriffe sehr energieintensiv sind und diese Einsparung die Kosten für die zusätzlichen Befehle überwiegt.

Grundsätzliche Funktionsweise

Grundsätzlich ist die Technik RP auf die Anwendung in Schleifen beschränkt. Daher wird zu Beginn die Applikation auf vorhandene Schleifen hin analysiert. Ein wichtiges Charakteristikum der Schleifen sind die Induktionsvariablen mit Anfangswert, Endwert und Schrittweite. Jeweils innerhalb dieser Schleifen wird untersucht, welche Speicherzugriffe auftreten und mit welchen Zugriffsmustern diese auf ein Array zugreifen. Zusätzlich muss das Array untersucht werden, um den Datenfluss bezogen auf das Array zu berechnen. Mithilfe dieser Information und nach der Berechnung der Anzahl von Registern, die zur Zwischenspeicherung der Werte und Einsparung der Speicherzugriffe notwendig sind, kann die Schleife dann transformiert werden.

An dieser Stelle soll auf andere veröffentlichte Arbeiten auf dem Gebiet der Speicherzugriffsoptimierung durch Nutzung von Registern eingegangen werden. RP gehört zur Klasse von Compilertechniken, die den Energieverbrauch ohne Änderungen an der Hardware beeinflussen. Die Technik wurde zusammen mit weiteren Optimierungen zur Optimierung der Speicherhierarchie von Carr [Car92] vorgestellt. Arrayzugriffsoptimierungen durch die Nutzung von Registern sind bei Callahan et al. [CCK90] beschrieben. Ein allgemeiner Überblick über Compilerbau und aktuelle Techniken findet sich bei Muchnick [Muc97]. Veröffentlichungen über die Auswirkungen dieser Techniken auf den Energieverbrauch sind nicht bekannt.

Ein Sonderfall des RP ist die Technik *Redundant Load Elimination (RLE)*[3], die als Beispiel in Abbildung 4.6 dargestellt ist.

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

(a) vorher

```
R8 = a[1];
for (i = 1; i < 1000; i++) {
  R1 = R8;
  ..
  a[i+1] = R4;
  R8 = R4;
  ...
  R3 = R8;
}
```

(b) nachher

Abbildung 4.6: Anwendung der Redundant Load Elimination

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

Die Technik erkennt und optimiert zwei bestimmte Zugriffe:

1. Das Register `R3` lädt den Inhalt von `a[i+1]`, obwohl derselbe Inhalt noch in Register `R4` vorliegt, da dessen Wert einige Instruktionen vorher in `a[i+1]` gespeichert wurde. Eine Speicherleseoperation kann also dadurch eingespart werden, dass der Inhalt von `R4` in `R3` kopiert wird.

2. Der Zugriff mit `R1` auf das Arrayelement `a[i]` kann eingespart werden, da im vorherigen Schleifendurchlauf genau dieses Element gespeichert wurde. Lediglich für den ersten Durchlauf muss der Schleife ein einmaliges Laden des ersten Elementes vorangestellt werden. Diese zugehörige Transformation wird in Abbildung 4.6b durch das Register `R8` vorgenommen.

Durch diese Transformation kann in dem angeführten Beispiel die Anzahl von 1000 * 2 Speicher-Lesezugriffen auf einen einzigen Speicher-Lesezugriff reduziert werden.

Die Technik RP behandelt im Gegensatz zur RLE auch größere Iterationsdistanzen. Eine *Iterationsdistanz* bezeichnet den Abstand zweier Indexfunktionen in Schleifeniterationen. Bei der RLE wird nur die Iterationsdistanz 0 (im Beispiel für den Lesezugriff für Register `R3`) und die Iterationsdistanz 1 (im Beispiel für den Lesezugriff für Register `R1`) optimiert. Iterationsdistanzen größer als 1 können nur durch die auf Arrayzugriffe in Schleifen verallgemeinerte Methode des RPs optimiert werden.

Programmanalyse

Die Analyse des zu compilierenden Programms auf die Anwendungsmöglichkeit von RP erfolgt in mehreren Schritten:

1. Schleifenerkennung,
2. Bestimmung von Induktionsvariablen und ihrer Grenzwerte,
3. Suche der Load- und Store-Zugriffe,
4. Array-Datenflussanalyse,
5. Berechnung der Anzahl freier Register.

Die Anwendung der Technik erfolgt auf der LIR-Ebene und nicht z. B. auf C-Code-Ebene, da präzise Informationen über die notwendige Anzahl zusätzlicher Instruktionen und die Anzahl der frei verfügbaren Register notwendig sind. Auf C-Code-Ebene entsteht häufig die Situation, dass RP angewendet wird und anschließend ein Spilling auftritt, weil sich beim Übersetzen herausstellt, dass kein Register mehr frei war. Da ein Spilling aber mehr Instruktionen verursacht und damit auch die Energiekosten ansteigen, ist die Auswirkung der Technik sogar nachteilig. Auf LIR-Ebene lassen sich die Auswirkungen genauer vorhersagen, um in derartigen Fällen die Anwendung der Technik zu vermeiden.

Die einzelnen Schritte werden im Folgenden näher detailliert und erläutert:

1. Schritt: Schleifenerkennung
   Da RP Zwischenergebnisse in Registern hält, muss analysiert werden, wo ein Wert gespeichert und wo derselbe Wert wiederverwendet wird. Die Anwendung in Schleifen nutzt aus, dass Programme den Großteil ihrer Zeit verhältnismäßig wenig unterschiedliche Instruktionen ausführen [HP90, Seite 11]. Lokale Verbesserungen in Schleifen haben daher große Auswirkungen auf das Gesamtprogramm.

Für die Schleifenerkennung wird der Kontrollfluss des Programms analysiert und die zur Graphendarstellung des Kontrollflusses gehörenden Kanten mit einem Tiefensuch-Algorithmus [Muc97] klassifiziert. Alle ermittelten Rückkanten (=backedges) geben Aufschluss auf vorhandene Schleifen. Beschränkt sind die hier beschriebenen Untersuchungen auf *single-entry/single-exit* Schleifen, was aber in der Praxis keine größere Einschränkung darstellt.

2. Schritt: Bestimmung von Induktionsvariablen und ihren Grenzwerten

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

Abbildung 4.7: Beispielprogramm für die Darstellung der Array-Datenflussanalyse

Nachdem die Schleifen erkannt worden sind, fehlen noch Informationen über die Induktionsvariablen zusammen mit ihrem Anfangswert, der Schrittweite und ihrem Endwert. Induktionsvariablen sind dabei Variablen, deren Werte von der Anzahl der Schleifeniterationen abhängen. Eventuell sind algebraische Transformationen notwendig, um diese Werte aus dem Programm zu extrahieren, wenn diese nicht direkt als Konstanten Bestandteil des Schleifenbefehls sind. Zuletzt werden - falls vorhanden - mehrere Induktionsvariablen in eine einzige transformiert. Diese Transformationen sind insbesondere deswegen notwendig, da das angewendete Verfahren auf der LIR aufgesetzt wurde. Da die Befehle auf der LIR weniger komplex als beispielsweise in höheren Sprachen sind, und die Arrayzugriffe durch mehrere algebraische Operationen umgesetzt werden, ist diese intensive Analyse zusammen mit algebraischen Transformationen notwendig, um beispielsweise das Zugriffsmuster mehrerer Zugriffe erkennen zu können. Im Beispiel in Abbildung 4.7 muss erkannt werden, dass in der ersten Anweisung in der Schleife auf das Element i und das Element $i - 2$ zugegriffen wird. In der folgenden Anweisung wird in das Element $2*i-5$ geschrieben. Wichtig ist hier, die unterschiedlichen Schrittweiten zu erkennen und zu berücksichtigen. Für diese Analyse wird jeder Instruktion eine Ersetzungsregel zugeordnet, die das Verhalten der Instruktion widerspiegelt. Die Ersetzungsregeln werden dann umgekehrt zur Ausführungsreihenfolge auf die Indexfunktion angewandt, wobei Kontrollstrukturen berücksichtigt werden müssen.
Das Ergebnis dieser Transformationen ist dann die Indexfunktion. Eine Indexfunktion ist die Funktion, die angibt, wo in Abhängigkeit zu der Induktionsvariablen der Zugriff auf ein Array stattfinden soll. In den praktischen Arbeiten wurden die Arbeiten auf affine Indexfunktionen beschränkt. Affine Funktionen sind Funktionen des Typs $f(x) = ax + b$ mit zwei konstanten Zahlen a und b.

3. Schritt: Suche der Load- und Store-Operationen
Im Programm werden auf LIR-Ebene die Load- und Store-Operationen gesucht, die auf ein Array zugreifen. Für jeden dieser Zugriffe auf das jeweilige Array muss die zugehörige Indexfunktion bezogen auf den Schleifenanfang ermittelt werden.

4. Schritt: Array-Datenflussanalyse
Neben der Bestimmung der Indexfunktion wird in diesem Schritt analysiert, wie der Datenfluss zwischen den einzelnen Array-Referenzen erfolgt. In dem Beispiel in Abbildung 4.7 ist veranschaulicht, wie an verschiedenen Stellen mit unterschiedlichen Indizierungsausdrücken i, $i - 2$, $2 * i - 5$ auf ein Array zugegriffen werden kann. Die Array-Datenflussanalyse soll nun erkennen, wie groß die Iterationsdistanz zwischen den einzelnen Zugriffen ist.
Für die Optimierungstechnik des RP können prinzipiell verschiedene Array-Datenflussanalysen verwendet werden [Fra99]. Für den vorliegenden Anwendungsfall wurde das δ-Verfahren ausgewählt,

da es mit mittlerer Präzision affine Indexfunktionen in Schleifen mit *single-entry/single-exit* Bedingung analysieren kann. Es arbeitet auf einer Vorwärtsanalyse und behandelt das vorliegende must-Problem [Muc97]. Das Ergebnis der Array-Datenflussanalyse gibt an, welche Speicherinhalte über welche Iterationsdistanzen erhalten bleiben. Für nähere Angaben sei auf [Sch00] verwiesen.

5. Schritt: Berechnung der Anzahl freier Register
Die letzte Stufe der Analyse betrifft die Anzahl vorhandener freier Register. Dafür muss eine Lifetime-Analyse angewandt werden, wie sie standardmäßig in der Registerallokation des Compilers stattfindet. Diese berechnet für jedes Register und für jede Instruktion, ob das Register aktuell benötigt wird, also lebendig ("*life*") ist. Als Ergebnis liegt dann die Anzahl frei verfügbarer Register innerhalb der Schleife vor. Als Besonderheit beim betrachteten ARM7-Prozessor sind im Thumb-Modus nur die ersten acht Prozessor-Register allgemein im Befehlssatz verwendbar. Die höheren acht Register können nur mit wenigen Befehlen angesprochen werden und werden daher zum Beispiel im *tcc*, dem Thumb-C-Compiler der Fa. ARM, oder im hier entwickelten *encc*-Compiler nicht verwendet. Eine Nutzung dieser höheren Register kann aber sehr gut für das RP geschehen, da der für alle Register vorhandene Move-Befehl ausreichend ist. Das RP kann damit auf die ansonsten nicht genutzten höheren Register zugreifen und findet daher häufiger Anwendung als bei Prozessoren, bei denen alle für RP verfügbaren Register in der Codegenerierung verwendet werden.

Anwendung

Nachdem die Analysen stattgefunden haben, kann auf dieser Basis geprüft werden, ob es grundsätzlich für das betreffende Optimierungsziel sinnvoll ist, das RP anzuwenden und ggf. bis zu welcher Tiefe der Pipeline die Anwendung Vorteile liefert.

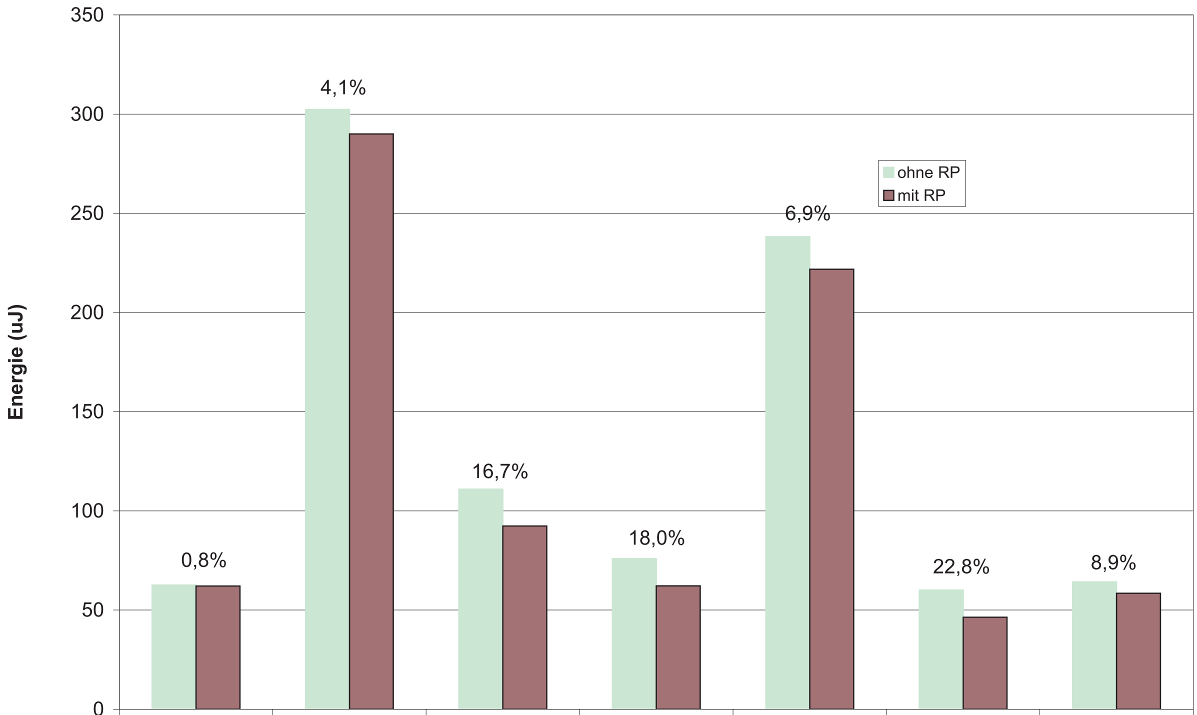
Ergebnisse

Für die Testläufe wurden Benchmarks aus mehreren Domänen ausgewählt:

- der Benchmark *biquad_N_sections* aus der DSPstone-Benchmark-Suite [ZVSM94],
- der *lattice*-Benchmark als Filterapplikation,
- und die Kernels *hydro fragment*, *tri-diagonal elimination*, *equation of state fragment*, *first sum* und *first difference* aus der Livermore-Benchmark-Suite [liv].

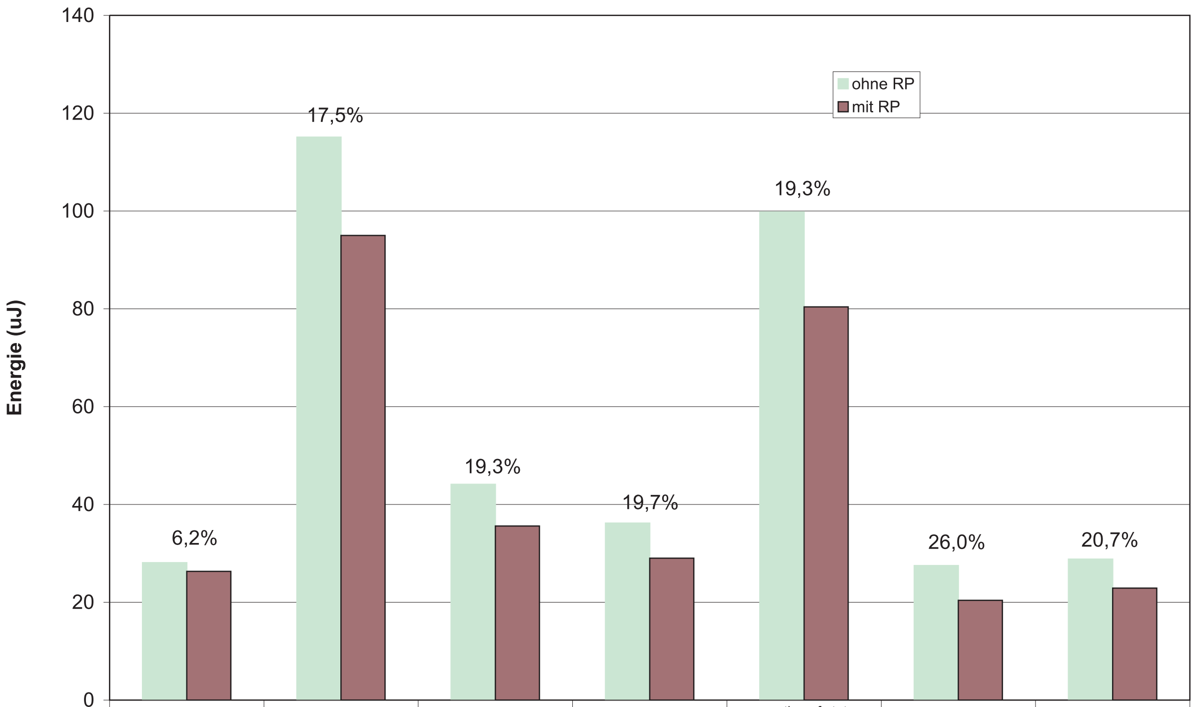
Da der ARM-Prozessor ohne *Floating Point Unit* typischerweise nicht für Applikation mit einer hohen Zahl von Floating Point Operationen eingesetzt wird, wurde in den Benchmarks der Datentyp *double* durch *integer* ersetzt. Die Wirkung der angewendeten Technik kann an den in Tabelle 4.1 dargestellten Ergebnissen deutlich gemacht werden. Dort ist für jeden Benchmark die Anzahl der freien Register vor und nach der Anwendung von RP gegenübergestellt.

Im Durchschnitt werden durch die Anwendung des RPs zwei zusätzliche Register belegt. Dadurch kann die Anzahl der Datenspeicherzugriffe der Benchmarks im Durchschnitt um 22,9% reduziert werden. Diese eingesparten Datenspeicherzugriffe bewirken eine relativ hohe Energieinsparung, da ein Speicherzugriff eingespart wird. Andererseits müssen für die Anwendung des RPs zusätzliche Befehle in die Schleifen eingefügt werden, die die Registerpipeline und das Durchschieben der Werte durchführen. Daher kann die Performance nicht im gleichen Verhältnis gesteigert werden, wie die Anzahl der Datenspeicherzugriffe reduziert wurde. Gleiches gilt für die Senkung des Energieverbrauchs.



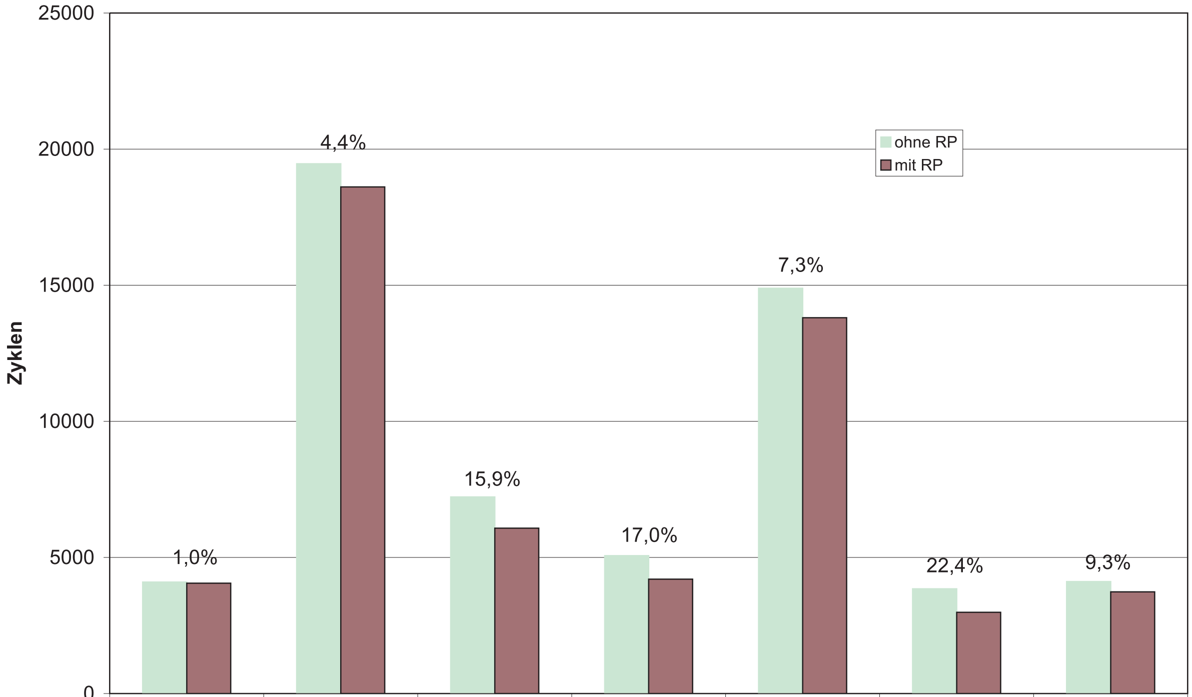
	biquad_N_sections	lattice (*10)	hydro fragment	tri-diagonal elimination	equation of state fragment	first sum	first difference
ohne RP	62,6	302,3	110,9	75,9	238,2	60,1	64,2
mit RP	62,1	290	92,4	62,2	221,8	46,4	58,5

Abbildung 4.8: Energieeinsparung durch Registerpipelining (Programm und Daten im Hauptspeicher)



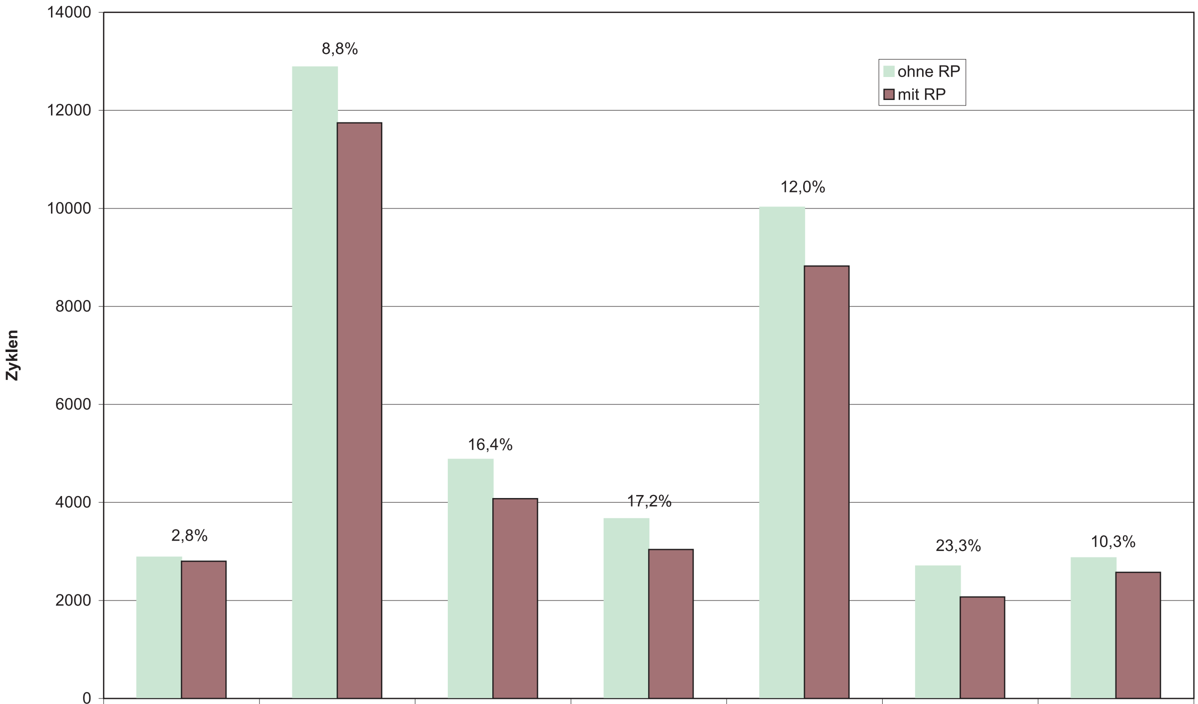
	biquad_N_sections	lattice (*10)	hydro fragment	tri-diagonal elimination	equation of state fragment	first sum	first difference
ohne RP	28,1	115,1	44,1	36,2	99,8	27,5	28,8
mit RP	26,3	95,0	35,6	29,0	80,4	20,4	22,9

Abbildung 4.9: Energieeinsparung durch Registerpipelining (Programm im Hauptspeicher, Daten im Scratchpad)



	biquad_N_sections	lattice (*10)	hydro fragment	tri-diagonal elimination	equation of state fragment	first sum	first difference
ohne RP	4093	19460	7223	5063	14890	3845	4115
mit RP	4054	18609	6073	4203	13808	2985	3733

Abbildung 4.10: Performancesteigerung durch Registerpipelining (Programm und Daten im Hauptspeicher)



	biquad_N_sections	lattice (*10)	hydro fragment	tri-diagonal elimination	equation of state fragment	first sum	first difference
ohne RP	2882	12885	4878	3668	10020	2700	2870
mit RP	2801	11746	4078	3038	8822	2070	2573

Abbildung 4.11: Performancesteigerung durch Registerpipelining (Programm im Hauptspeicher, Daten im Scratchpad)

Benchmark	freie Register		Differenz	Datenspeicherzugriffe (Bytes)		Reduzierung Datenspeicherzugriffe
	ohne RP	mit RP		ohne RP	mit RP	
biquad_N_sections	5	2	3	1.196	1.100	8,0%
lattice	3	0	3	43.020	31.620	26,5%
Hydro fragment	7	6	1	1.632	1.272	22,1%
Tri-diagonal elimination	7	6	1	1.472	1.152	21,7%
Equation of state fragment	4	0	4	4.048	3.000	25,9%
First sum	8	7	1	1.184	864	27,0%
First difference	7	6	1	1.224	872	28,8%
Durchschnitt			2			22,9%

Tabelle 4.1: Freie Register und Anzahl von Datenspeicherzugriffen

Um den Einfluss auf den Energieverbrauch der Applikation zu beurteilen, wurden für das ARM-System für die verschiedenen Benchmarks ebenfalls Vergleiche durchgeführt. Abbildung 4.8 zeigt, dass der Energieverbrauch des Gesamtsystems zwischen 0,8% und 22,8% verringert werden kann. Die Vorteile der Anwendung von RP hängen aufgrund der einzufügenden zusätzlichen Instruktionen vom Energieverbrauch des Programmspeichers und des Datenspeichers ab, auf den Zugriffe eingespart werden können. Da die Energiewerte von System zu System z. B. durch Einsatz von Onchip- und Offchip-Speichern schwanken können, wurde ein weiterer Vergleich in Abbildung 4.9 durchgeführt, bei dem das Programm im Hauptspeicher (=Offchip) und die Daten im Scratchpad (=Onchip) liegen. Auch dabei sind nennenswerte Einsparungen des Energieverbrauchs zwischen 6,2% und 26,0% zu erreichen, die höher sind als bei der vorherigen Messreihe, da die Zugriffe auf das Scratchpad weniger Energie als die Zugriffe auf den Hauptspeicher kosten.

Die positiven Auswirkungen der Technik können auch für die Performance festgestellt werden. Entsprechend wurden die gleichen Versuche hinsichtlich der Performance ausgewertet und in Abbildung 4.10 für Programm und Daten im Hauptspeicher sowie in Abbildung 4.11 für Programm im Hauptspeicher und Daten im Scratchpad dargestellt. Die Verbesserungen liegen im Bereich zwischen 1,0% und 22,4% bzw. zwischen 2,8% und 23,3%.

Auch wenn diese Ergebnisse den Anschein erwecken, dass die Optimierung mit RP auf Energie stets auch Performance-Optimierungen liefert, kann in einzelnen Fällen durchaus auch mit RP die Performance verschlechtert werden. Für das Programmbeispiel *time-energy* in Abbildung 4.12 sind die Ergebnisse in Tabelle 4.2 dargestellt. Während sich der Energieverbrauch wie erwartet um 17% verringert, steigt die Anzahl der benötigten Zyklen um 8,8% an. Die Ursache liegt in der langen Pipeline aus Registern, wodurch viele Instruktionen mit geringem Energieverbrauch eingefügt werden.

```
int a[100+7];

int main(void) {
  int i, b = 0, *c;
  for (i = 0; i < 100+7; i++) {
    a[i] = i;
  }
  c = a;
  for (i = 0; i < 100; i++) {
    b += *c;
    b += *(c+7);
    c += 1;
  }
  return(b);
}
```

Abbildung 4.12: Programmbeispiel *time-energy* für Unterschiede zwischen Energie- und Performance-Optimierung

	ohne RP	mit RP	Differenz
Anzahl von Zyklen	1958	2130	8,8%
Anzahl ausgeführter Instruktionen	795	1330	67%
Datenspeicherzugriffe	796 Bytes	492 Bytes	-38%
Energieverbrauch	19,33 μJ	16,02 μJ	-17%

Tabelle 4.2: Vergleich von Performance, Größe und Energieverbrauch für Programm *time-energy*

Fazit

In einem Compiler werden verschiedene und vielfältige Optimierungen eingesetzt. Für einen energieoptimierenden Compiler sind insbesondere die Optimierungen interessant, die sich bei einer Optimierung für Energie von einer Optimierung für Performance unterscheiden. Die vorgestellte Technik Registerpipelining zeigt beispielhaft dieses unterschiedliche Verhalten auf. Auch die Tatsache, dass RP die Speicherzugriffe optimiert, bietet aufgrund des hohen Energieverbrauchs von Speicherzugriffen ein größeres Einsparungspotenzial.

Es konnte auch gezeigt werden, dass ein Teil der Optimierungen auf unteren Ebenen - hier der LIR - angewendet werden muss, da die Optimierungen auf höheren Ebenen nicht die notwendige Genauigkeit in der Voraussage von Einsparungen liefern. Wichtig ist eine genügend präzise Bewertung des Energieverbrauches in allen Systemkomponenten, die auch durch die Optimierung bei den Entscheidungen Berücksichtigung finden.

Allgemein kann festgestellt werden, dass die bekannten Compiler-Optimierungen für die Anwendung in energieoptimierenden Compilern untersucht und gegebenenfalls entsprechend angepasst werden müssen.

Eine ausführlichere Beschreibung der RP-Technik sowie den Einfluss auf den Energieverbrauch findet sich in der Arbeit von Schwarz [Sch00] sowie in [SSWM01].

4.3 Nutzung von Caches

Nach der Einführung von Speicherhierarchien und der Anordnung und generellen Funktionsweise von Caches in der Hierarchie (siehe Kapitel 4.1), erfolgt nun eine detaillierte Beschreibung, um die Vor- und Nachteile dieses Speichertypes darzustellen und die neu entwickelten Strategien zur effizienten Speichernutzung in Kapitel 5 damit vergleichen zu können.

Tag	Index	Block-Offset

Abbildung 4.13: Bestandteile einer Speicheradresse beim Cachezugriff

Cacheorganisation

Neben der Größe und der Zugriffszeit eines Caches gibt es weitere entscheidende Parameter, die die Organisation betreffen und Auswirkungen auf die Arbeitsweise und Effizienz zeigen. Im Gegensatz zu anderen Speichertypen ist bei der Anfrage an einen Cache noch unbekannt, ob dieses Datum im Cache enthalten ist. Dies wird mittels Hardware zur Laufzeit geprüft und entweder der Zugriff auf das Datum im Cache eingeleitet oder eine Anforderung auf die nächste Speicherhierarchieebene erzeugt. In Abhängigkeit der Organisation des Caches müssen dafür ein oder mehrere Vergleiche zwischen der vom Prozessor gelieferten Adresse und den Adressen im Tagspeicher, die zu jeweils einer Cacheline gehören, vorgenommen werden. Es gibt die unterschiedlichen Cacheorganisationsformen *direct-mapped*, *set associative* und *full associative*, die nachfolgend erläutert werden. Mit einem einzelnen Vergleich kommt die Organisationsform *direct-mapped* aus, in der es für jede Adresse aber nur genau eine Cacheline gibt, in welcher der Inhalt gespeichert werden kann. Diese Organisation ist in Abbildung 4.14 dargestellt.

Um den Zugriffsmechanismus zu erläutern, ist in Abbildung 4.13 eine Adresse, die der Prozessor auf den Adressbus legt, und die Zerlegung in ihre Bestandteile für den Cachezugriff dargestellt. Der *Index* der

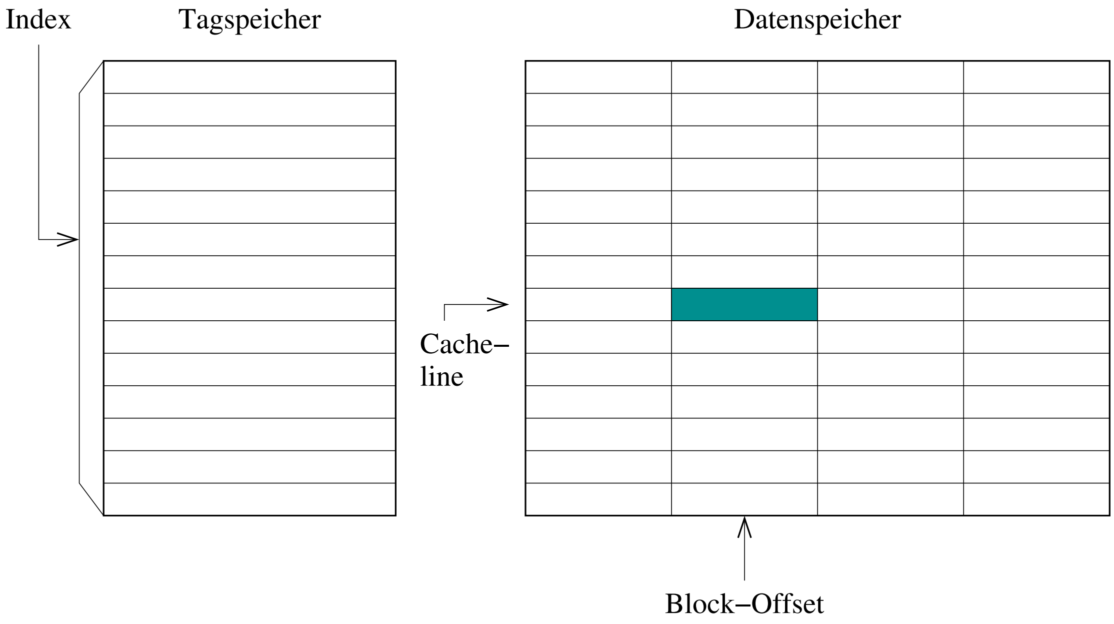


Abbildung 4.14: Direct Mapped Cache

Adresse legt fest, welche Cacheline dieser Adresse zugeordnet ist. Bei einem Cache des Typs *direct-mapped* gibt es nur genau eine Cacheline, die zum Speichern des Datums genutzt werden kann. Das *Tag* enthält die höheren Adressbits, die im Tagspeicher festgehalten werden und deren Inhalt mit der aktuellen Adresse verglichen wird, um festzustellen, ob die gewünschte Adresse aktuell im Cache vorgehalten wird. Der letzte Bestandteil, der *Block-Offset*, wählt innerhalb einer Cacheline aus, auf welches Datum zugegriffen wird.

Bei einem *direct-mapped* Cache kann häufiger der Fall auftreten, dass die betreffende Cacheline schon belegt ist. Die Wahrscheinlichkeit hängt von der Folge von Adressen ab, in der auf den Speicher zugegriffen wird. Falls die Cacheline schon belegt ist, muss der Inhalt ausgetauscht werden, auch wenn der Cache an anderer Stelle noch freie Cachelines besitzt. Um diese Wahrscheinlichkeit zu verringern, gibt es weitere Organisationsformen, in denen für jede Adresse mehrere Cachelines existieren, in denen die Informationen abgelegt werden können. Diese so genannten *set associative* Caches werden nach der Anzahl der Einträge im Tagspeicher für ein Set benannt. Ein *2-way set associative* Cache mit jeweils *2* Cachelines in einem Set für eine Adresse ist in Abbildung 4.15 dargestellt. Der Index als Bestandteil der Adresse wählt hier den Set aus und danach müssen parallel die in den zwei Tags des Sets abgelegten Adressen mit der Zugriffsadresse verglichen werden. Durch den größeren Freiheitsgrad bei der Auswahl einer Cacheline wird potenziell die Hitrate größer, der Hardwareaufwand durch das parallele Vergleichen von mehreren Tags mit der Zugriffsadresse insbesondere für den Energieverbrauch jedoch höher. Weiterhin sind die Bauarten von *4-way set associative* und *8-way set associative* Caches in der Praxis relevant, wobei die häufigsten Bauformen der *2-way* und der *4-way set associative* Cache sind.

Mehr theoretischer Natur ist der *full associative* Cache, in der jede Adresse in jeder Cacheline abgelegt werden kann. In der Praxis wird diese Bauform selten verwendet, da im Gegensatz zu den anderen Organisationsformen der Flächenbedarf stark ansteigt, ohne die Hitrate noch stark zu erhöhen.

Es gilt daher, für das jeweilige System eine geeignete Form zu ermitteln, die durch die Parameter des Flächenbedarfs (und damit auch der Herstellungskosten), der Hitrate und des Energieverbrauchs gekennzeichnet wird. Bei dem später betrachteten ARM710T-System wurde ein *4-way set associative* Cache integriert.

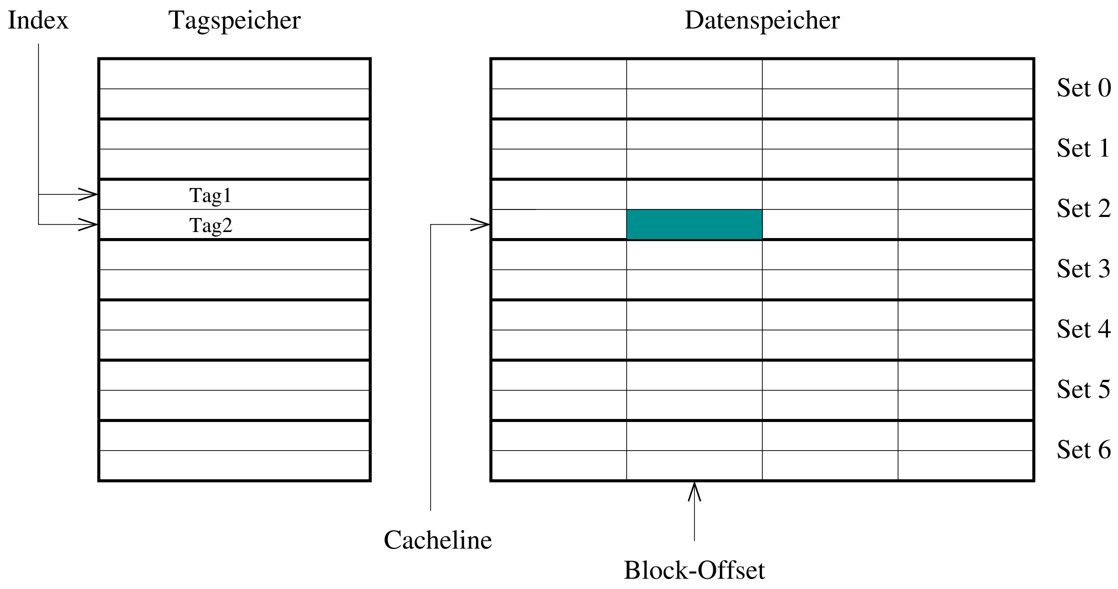


Abbildung 4.15: Set-Associative Cache

Ersetzungsstrategie

Als weiterer Parameter der Cacheorganisation ist die Ersetzungsstrategie für *set associative* Caches zu nennen. Wenn eine Adresse nicht im Cache gefunden wird, muss ein entsprechender Block nachgeladen (bei lesenden Zugriffen) sowie ein bereits existierender Block verdrängt werden. Welcher Block innerhalb des Sets verdrängt wird, wird mithilfe einer Ersetzungsstrategie entschieden. Die wesentlichen Strategien sind dabei *Least-recently used*, *Random* und *Round-Robin*:

- Random
  Die Zufallsstrategie versucht eine gleichmäßige Verteilung zu erreichen und wählt die zu verdrängenden Blöcke zufällig aus.

- Least-recently used (LRU)
  Bei dieser Ersetzungsstrategie wird der Block verdrängt, der am längsten nicht verwendet wurde. Dafür muss diese Information stets mitgeführt werden. Der Nutzen ergibt sich aus der zeitlichen Lokalität, da ein Datum, auf welches zugegriffen wird, i.d.R. auch mit höherer Wahrscheinlichkeit in näherer Zukunft verwendet wird.

- Round-Robin
  Bei Round-Robin werden reihum die Cachelines wieder neu belegt, ohne zu berücksichtigen, wann zuletzt auf eine Cacheline zugegriffen wurde.

Die gewählte Strategie hat Einfluss auf die Hitrate des Caches, wobei diese auch von den Zugriffsmustern, die sich aus der Applikation ergeben, abhängt. Andererseits ist für die Implementierung der Strategie eine unterschiedliche Chipfläche notwendig und letztendlich variiert auch der Energiebedarf beim Einsatz der unterschiedlichen Strategien.

Welche der Strategien im konkreten Fall bei dem Entwurf eines Caches eingesetzt wird, hängt daher von der komplexen Abwägung der beschriebenen Einflussfaktoren ab.

Instruktions- und Datencache

Grundsätzlich muss noch unterschieden werden, welche Informationen in einem Cache abgelegt werden. Handelt es sich nur um Instruktionen bzw. nur um Daten, bezeichnet man den Cache als *Instruktionscache* bzw. *Datencache*. Wegen der Trennung von Instruktionen und Daten heißt dieses Konzept auch *Split Cache*. Bei einem gemeinsamen Cache für Instruktionen und Daten bezeichnet man den Cache als *Unified Cache*.

Cache-Hierarchie

In einer Speicherhierarchie können mehrere Caches angeordnet werden. Näher zum Hauptspeicher sind die Caches größer und langsamer als direkt am Prozessor. In Abhängigkeit von der Entfernung zum Prozessor werden die Caches *First-Level*, *Second-Level* oder auch *Third-Level Cache* benannt. Dabei wird der First-Level Cache auch häufig direkt auf dem Chip, auf dem sich der Prozessor befindet, untergebracht und in diesem Fall auch als Onchip-Cache bezeichnet.

Schreibstrategie

Ein weiterer Unterschied von Cacheorganisationen besteht im Ablauf eines Schreibvorgangs. Es werden hier grundsätzlich zwei Strategien unterschieden:

1. Write-through
   Bei einem Schreibvorgang wird sowohl in den Cache als auch in die nächstniedrigere Speicherebene geschrieben. Der Vorteil dieser Strategie ist, dass die Strategie einfacher zu realisieren ist und der Hauptspeicher immer die aktuellste Kopie der Daten enthält, was insbesondere für Multiprozessor-Systeme und I/O wichtig ist.

2. Write-back
   Die Information wird nur in den Cache geschrieben, sodass das Schreiben auch mit Cache-Geschwindigkeit erfolgen kann. Die nächstniedrigere Speicherebene wird erst aktualisiert, wenn der Block im Cache ersetzt wird. Um beim Ersetzen eines Blockes nur die nächstniedrigere Speicherebene aktualisieren zu müssen, wenn dies auch notwendig ist, wird ein zusätzliches *Dirty*-Bit implementiert, welches angibt, ob der Block im Cache aktualisiert wurde. Dadurch, dass nur geänderte Blöcke auch wieder in die nächstniedrigere Speicherebene geschrieben werden müssen, ist auch die Speicherbandbreite geringer.

Falls beim Schreiben der betreffende Block nicht im Cache vorhanden ist, werden für diesen Fehlzugriff (Write Cache Miss) zwei weitere Fälle unterschieden:

1. Write-allocate
   Der Block wird vor dem Schreibvorgang noch aus der niedrigeren Speicherebene in den Cache geladen. Danach wird der eigentliche Schreibvorgang fortgesetzt. Es wird also automatisch eine Allokation einer Cacheline beim Write Cache Miss vorgenommen.

2. No-write-allocate
   Der Block wird nicht in den Cache geladen, sondern die Daten direkt in die niedrigere Speicherebene geschrieben. Die automatische Allokation einer Cacheline wird somit nicht durchgeführt.

Cache-Energiemodell

Nach der Erläuterung der Funktionsweise von Caches ist es für die weiteren Untersuchungen relevant, wie ein Cache sich bezüglich der Zugriffszeiten und des Energieverbrauchs verhält. Bei den Zugriffszeiten unterscheidet man - wie schon bei der Vorstellung der Speicherhierarchie in Kapitel 4.1 erläutert - die Fälle, wenn das Datum im Cache vorhanden ist, mit der Zeit $t_{hit,R}$ bzw. $t_{hit,W}$ und die Fälle, in denen bei einer Leseoperation in den Cache noch Daten geladen werden müssen mit $t_{miss,R}$ bzw. bei einer Schreiboperation die Daten direkt in den Hauptspeicher geschrieben werden müssen mit der Zugriffszeit $t_{miss,W}$. Es wird hier ein Cache mit der Schreibstrategie Write-through betrachtet. Für die Strategie Write-back fällt die Zeit $t_{miss,W}$ nur bei einem Teil der Zugriffe an, was durch eine Fallunterscheidung modelliert werden muss.

Zugriffsart	Cache Read ($N_{Ca,R}$)	Cache Write ($N_{Ca,W}$)	Hauptspeicher Read ($N_{H,R}$)	Hauptspeicher Write ($N_{H,W}$)
Read Hit	1	0	0	0
Read Miss	1	L	L	0
Write Hit	0	1	0	1
Write Miss	1	0	0	1

Tabelle 4.3: Anzahl der Cachezugriffe

Die einzelnen möglichen Zugriffsarten auf den Cache *Read Hit*, *Read Miss*, *Write Hit* und *Write Miss* werden in Tabelle 4.3 für das hier beispielhaft verwendete System dargestellt. Es wird für jede Zugriffsart die Anzahl der jeweils ausgeführten Schreib- und Lesezugriffe auf Cache und Hauptspeicher gegenübergestellt [BSL+01]. Bei dem betrachteten Cache wird davon ausgegangen, dass er einen Block innerhalb eines Zyklus lesen oder schreiben kann. Jede Cacheline, die aus *L* Worten besteht, kann in *L* Zugriffen aus dem Hauptspeicher nachgeladen bzw. in den Cache geschrieben werden. Folgende Zugriffsarten müssen nun unterschieden werden:

- Read Hit
  In diesem Fall reicht ein einziger Zyklus aus, um den Block aus dem Cache zu lesen. Dies ist auch der einzige Fall, wo ein Zugriff auf den Hauptspeicher nicht notwendig und somit eingespart werden kann.

- Read Miss
  Bei einem Read Miss wird zuerst versucht, die Information aus dem Cache zu lesen. Da bei dem Vergleich der betreffenden Tags festgestellt wird, dass das Datum nicht im Cache vorliegt, wird ein Lesen der Cacheline mit *L* Zugriffszyklen aus dem Hauptspeicher eingefügt. Diese vom Hauptspeicher gelesenen Daten werden dann mit *L* Zugriffen in den Cache geschrieben, bevor das vom Prozessor angeforderte Datum erfolgreich aus dem Cache geliefert werden kann.

- Write Hit
  Der betrachtete Cache basiert auf der Write-through-Schreibstrategie und erzeugt neben dem Schreiben in den Cache auch einen Schreibzugriff auf den Hauptspeicher.

- Write Miss
  Falls beim Schreiben festgestellt wird, dass das betreffende Datum nicht im Cache liegt, wird es in dem hier betrachteten System nur in den Hauptspeicher geschrieben, was der No-write-allocate-Strategie entspricht. Auch wenn kein Cache Read ausgeführt wird, muss doch die Adresse mit den Tags verglichen werden. Daher muss bei dieser Zugriffsart auch ein Cache Read bei den Zugriffen gezählt werden.

Die Berechnung der Zyklen für die verschiedenen Cachezugriffe bedeutet, dass auch die Anzahl der Wartezyklen für die Hauptspeicherzugriffe mit berücksichtigt werden muss. Für das betrachtete System mit einem 16-Bit breiten Datenbus sind folgende Wartezyklen aufgrund der Latenzzeit des Speicherbausteins anzusetzen:

Wortbreite	Wartezyklen Hauptspeicher
8 Bit	1
16 Bit	1
32 Bit	3

Auf dieser Basis können nun die Anzahl der Zyklen für die unterschiedlichen Zugriffe Read Hit, Read Miss, Write Hit und Write Miss ermittelt werden, die in Tabelle 4.4 für die Blockgröße L = 2 Worte dargestellt sind. Anzumerken ist, dass bei einem Write Hit die Zugriffe auf Cache und Hauptspeicher gleichzeitig erfolgen, sodass die Anzahl der Zyklen sich durch den Hauptspeicherzugriff bestimmt.

Zugriffsart	Cache	Hauptspeicher	Gesamtzyklen
Read Hit	1 Zyklus	0 Zyklen	1 Zyklus (=$Cycle_{Read,Hit}$)
Read Miss	1 Zyklus	1 Zyklus + 2 * (1 Zyklus + 3 Waitstates)	9 Zyklen (=$Cycle_{Read,Miss}$)
Write Hit	1 Zyklus	1 Zyklus + 3 Waitstates	4 Zyklen (=$Cycle_{Write,Hit}$)
Write Miss	1 Zyklus	1 Zyklus + 1 Zyklus + 3 Waitstates	5 Zyklen (=$Cycle_{Write,Miss}$)

Tabelle 4.4: Zugriffszyklen auf den Cache

Mit der berechneten Anzahl der Zyklen des Cachezugriffs kann nun die Performance des Systems und der Energieverbrauch des Prozessors bestimmt werden, der während dieser Zeit auf das Ende des Cachezugriffs warten muss. Es fehlt jetzt aber noch die Berechnung des Energieverbrauchs in der Speicherhierarchie. Die Häufigkeiten der Speicherzugriffe wurden in Tabelle 4.3 bereits betrachtet, sodass wir uns nun dem Energieverbrauch der verschiedenen Zugriffe zuwenden können.

Für den Hauptspeicher ist es bei den Zugriffen *Hauptspeicher Read* und *Hauptspeicher Write* kein Unterschied, ob der Prozessor oder der Cache lesend oder schreibend zugreifen. Der Energieverbrauch des Hauptspeichers ist daher gleich und kann getrennt vom Cacheverhalten betrachtet werden.

Ein *Cache Read* tritt in zwei Varianten auf. Beim *Hit* wird ein vollständiger Lesezugriff mit dem Energieverbrauch $E_{Ca,R}$ auf den Cache durchgeführt, bei dem die angelegte Adresse mit einem oder mehreren Tags verglichen wird, das angeforderte Datum aus dem Speicher geholt und durch die Ausgangstreiber auf den Datenbus gelegt wird. Im Gegensatz dazu wird bei einem Miss der Zugriff nur bis zum Vergleich der Tags ausgeführt. Ein Auslesen des Datums und das Aktivieren der Ausgangstreiber unterbleibt, wodurch die Anzahl der Zyklen unverändert ist, aber einen etwas reduzierten Energieverbrauch im Vergleich zum Hit bedeutet. Da aber ein großer Teil des Energieverbrauchs durch die Vergleiche entsteht und auch die Zyklenzahl bei *Hit* und *Miss* identisch ist, setzen wir für beide Fälle den gleichen Energieverbrauch $E_{Ca,R}$ an, um das Modell einfach zu halten.

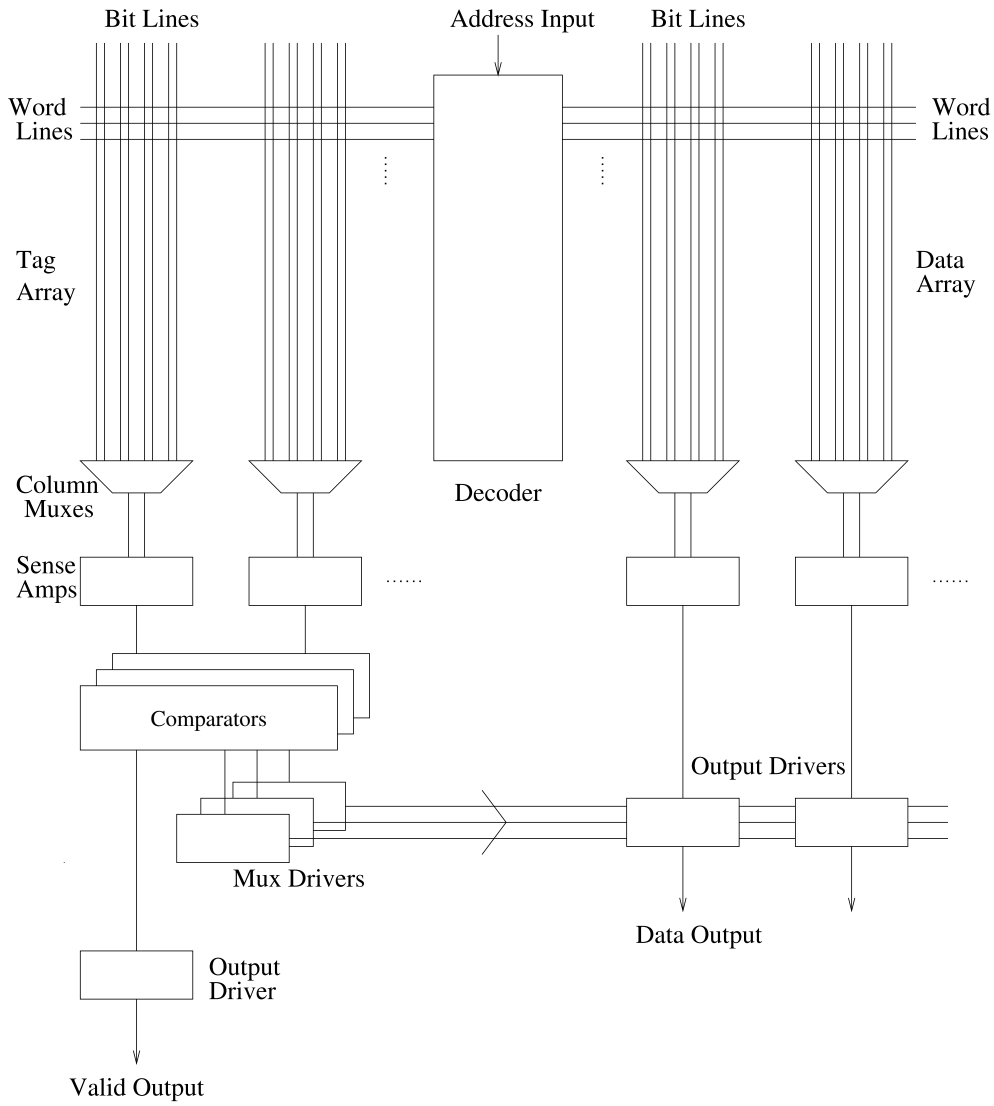


Abbildung 4.16: CACTI-Cache-Modell

Der letzte zu betrachtende Fall ist ein *Cache Write*, der entweder ein Schreiben eines einzelnen Datums durch den Prozessor oder das Schreiben einer kompletten Cacheline umfasst, wenn diese zuvor aus dem Hauptspeicher gelesen wurde. Der Energieverbrauch $E_{Ca,W}$ ist daher annäherungsweise proportional zur Anzahl der geschriebenen Datenworte und dementsprechend der Anzahl der Zyklen in Tabelle 4.4. Es ergibt sich nun der Energieverbrauch $E_{Mem,total}$ bei einem Cachezugriff durch die Multiplikation mit der jeweiligen Häufigkeit N der betreffenden Zugriffsart aus Tabelle 4.3:

$$E_{Mem,total} = E_{Ca,R} * N_{Ca,R} + E_{Ca,W} * N_{Ca,W} + E_{H,R} * N_{H,R} + E_{H,W} * N_{H,W}$$

Es fehlt nun noch die Bestimmung der Parameter $E_{Ca,R}$ und $E_{Ca,W}$, um ein vollständiges Modell zu besitzen.

Cache-Parameter	Wert
Technologiegröße	0,50 μm
Organisation	4-way set associative
Blockgröße	8 Bytes
Datenbusbreite	32 Bit
Cache-Adressbusbreite	24 Bit

Tabelle 4.5: Cache-Parameter

Als Modell für den Energieverbrauch bei einem einzelnen Zugriff auf den Cache wird hier das so genannte CACTI-Modell von Wilton et al. [WJ94, WJ96] gewählt. Es handelt sich hierbei um ein analytisches Modell, welches im Vergleich zur Bestimmung mit dem Tool Hspice auf Transistorebene eine Ungenauigkeit kleiner 10% zeigt, jedoch eine viel geringere Berechnungskomplexität besitzt. Mit Hilfe dieses Cache-Modells (Abb. 4.16) kann für eine gewählte Technologiegröße, Cachegröße und Cacheorganisation der Energieverbrauch (siehe Tab. 4.5) eines Cachezugriffes bestimmt werden. Der Energieverbrauch berechnet sich nach der folgenden Summenformel, in der der Verbrauch der Einzelkomponenten aufaddiert wird:

$$E_{Ca} = E_{DataArray} + E_{TagArray} + E_{Decoder} + E_{WordLines} + E_{BitLine} +$$
$$E_{ColumnMux} + E_{SenseAmp} + E_{Compare} + E_{ValidOutputDrv} + E_{OutputDrv}$$

Weitere Modell-Ergebnisse von CACTI für die Zugriffsgeschwindigkeit, die stets kleiner oder gleich der Zykluszeit des betrachteten ARM7-Systems liegt, sind für diese Untersuchungen nicht relevant. Für die Zugriffszeit ist der Prozessortakt der limitierende Faktor.

Für einen möglichst realistischen Vergleich unterschiedlicher Systeme wurden die Parameter für ein System mit Cache entsprechend einem real verfügbaren ARM710-System [ARM98] gewählt. Für diese Parameter können nun bei Variation der Cachegröße und der Organisationsform die Energieverbrauchswerte pro Zugriff bestimmt werden, die in Tabelle 4.6 dargestellt sind. Untersucht wurden sowohl ein direct-mapped Cache (DM), 2-way set bis 8-way set associative Caches (2X - 8X) als auch full associative Caches (FA). Die in Tabelle 4.6 mit einem "*" markierten Werte konnten nicht mehr mit dem CACTI-Modell berechnet werden und sind daher durch lineare Approximation ermittelt worden.

Lee [Lee01] hat die Performance und den Energieverbrauch in Abhängigkeit der Organisationsform und der Cachegröße bei verschiedenen Benchmarks untersucht. Zu diesen Benchmarks gehören die Filterapplikation *biquad_N_sections* aus der DSPstone-Benchmark-Suite [ZVSM94] und *quicksort* als häufig verwendeter Sortieralgorithmus. In den Abbildungen 4.17 und 4.18 wurden die Cachegrößen variiert und die Zyklen für die Ausführung des Benchmarks für unterschiedliche Cacheorganisationsformen dargestellt. Die Cachegröße "0" entspricht einem System ohne Cache und wurde zu Vergleichszwecken mit

Cachegröße	DM	2X	4X	8X	FA
64 Bytes	0,71 nJ	1,63 nJ*	2,87 nJ*	5,52 nJ*	1,20 nJ
128 Bytes	0,76 nJ	1,79 nJ	3,15 nJ*	5,87 nJ*	1,36 nJ
256 Bytes	0,86 nJ	1,90 nJ	3,32 nJ	6,24 nJ*	2,03 nJ
512 Bytes	0,98 nJ	2,05 nJ	3,48 nJ	6,63 nJ	2,47 nJ
1.024 Bytes	1,15 nJ	2,23 nJ	3,75 nJ	6.92 nJ	3,33 nJ
2.048 Bytes	1,47 nJ	2,55 nJ	4,04 nJ	7,37 nJ	6,00 nJ
4.096 Bytes	1,69 nJ	2,88 nJ	4,71 nJ	7,95 nJ	9,18 nJ
8.192 Bytes	2,67 nJ	3,57 nJ	5,39 nJ	8,89 nJ	17,24 nJ

Tabelle 4.6: Energieverbrauch pro Cachezugriff

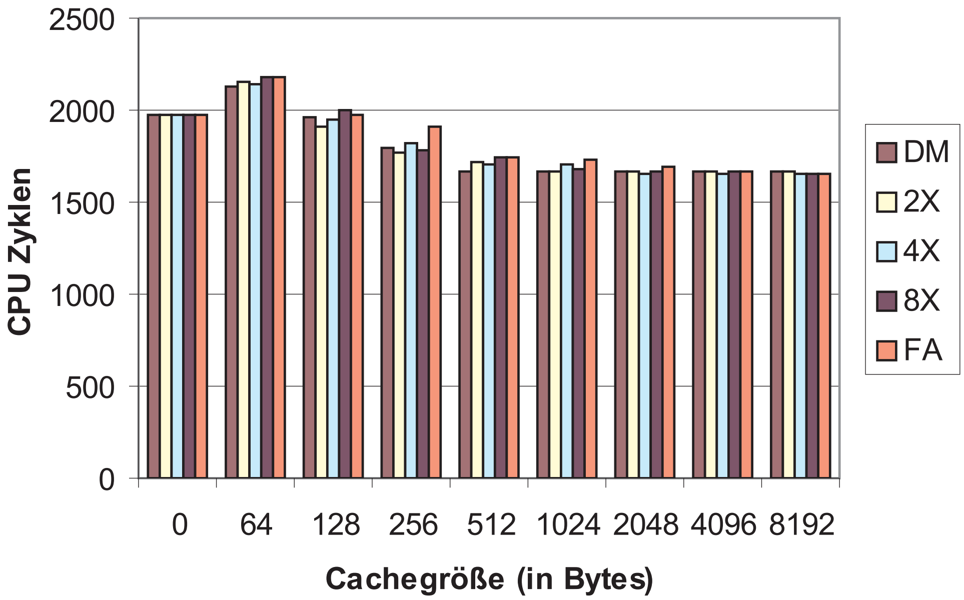


Abbildung 4.17: Prozessorzyklen bei unterschiedlichen Caches für die *biquad_N_sections*-Routine

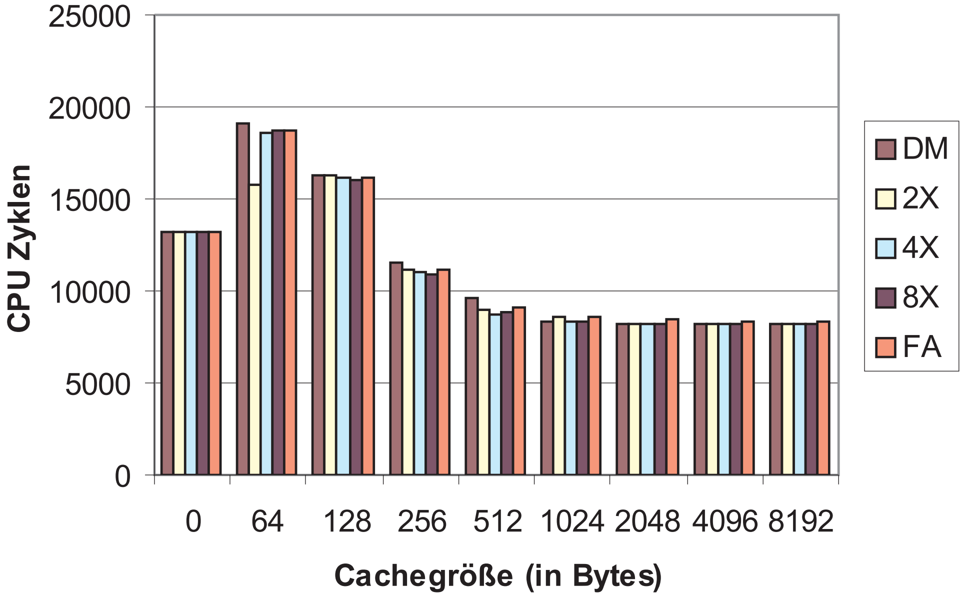


Abbildung 4.18: Prozessorzyklen bei unterschiedlichen Caches für die *quicksort*-Routine

aufgetragen. Für die anderen Cachegrößen kann man feststellen, dass kleinere Caches durchaus schlechter sein können als Systeme ohne Cache. Eine gewisse Mindestgröße für eine Applikation ist notwendig, da ansonsten eine hohe Zahl von *Cachemisses* entsteht und dieser nachteilige Effekt insgesamt den Nutzen durch den Cache übersteigt. Weiterhin kann ab einer bestimmten Cachegröße keine Veränderung mehr festgestellt werden, da dann die vollständige Applikation im Cache gespeichert werden kann.

Ein weiteres Ergebnis betrifft die unterschiedlichen Cacheorganisationen, deren optimale Form nicht allgemein ermittelt werden kann. Generell ist es zwar vorteilhaft für die Performance, wenn die Set-Assoziativität ansteigt, allerdings kann für Einzelfälle eine geringere Set-Assoziativität zu weniger Prozessorzyklen führen. Ähnlich ist es mit full associative Caches, die für die Performance allgemein am besten sind, allerdings auch einen höheren Flächenverbrauch und bei größeren Caches auch einen höheren Energieverbrauch aufweisen.

Für die Benchmarks wurden in den Simulationen die in Tabelle 4.6 angegebenen Energiewerte pro Cachezugriff berücksichtigt. Diese führen in den Abbildungen 4.19 und 4.20 zu Verschiebungen zu Gunsten von Cacheorganisationen mit geringerer Set-Assoziativität. Es ist zu erkennen, dass ein Cache ab einer gewissen Größe immer Vorteile liefert. Deutlich wird dies auch in Abbildung 4.21, wo die Energieanteile des Prozessors, des Hauptspeichers und des Caches einzeln dargestellt werden. Ohne Cache dominiert der Hauptspeicher, der ca. 2/3 des gesamten Energieverbrauchs verursacht. Schrittweise wird dies reduziert und in geringem Maße in den Cache verlagert. Bei einem 2KBytes großen Cache besteht dann ein ausgewogenes Verhältnis zwischen Cache und Hauptspeicher und nochmals dem gleichen Anteil an Prozessorenergie. Es wird auch deutlich, dass die Prozessorenergie aufgrund der reduzierten Anzahl von Hauptspeicherzugriffen abnimmt, da weniger Wartezyklen entstehen.

Weiterhin kann festgestellt werden, dass die Auswirkungen eines Caches auf den Energieverbrauch stärker sind als die Auswirkungen auf die Performance des Systems. Ein weiterer Effekt des Energieverbrauchs ist, dass ein größerer Cache, der sich nicht mehr vorteilhaft auswirkt, sogar nachteilig sein kann, da der Energieverbrauch pro Zugriff ansteigt.

Insgesamt kann festgestellt werden, dass die Cachegröße für ein System, das auf Energieverbrauch optimiert werden soll, genau auf die Applikation angepasst werden muss. Bei der Cacheorganisation sind im Vergleich zu Performance-optimierten Systemen eher Caches mit geringerer Set-Assoziativität zu bevorzugen.

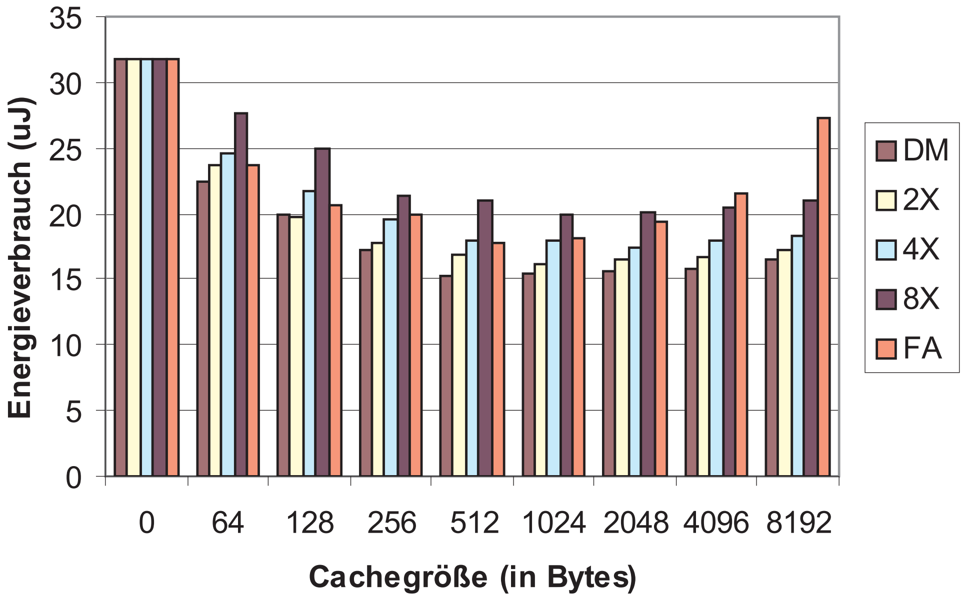


Abbildung 4.19: Energieverbrauch bei unterschiedlichen Caches bei *biquad_N_sections*

Verwandte Arbeiten

Vor der Vorstellung der neuen Strategien zur Minimierung des Energieverbauchs soll noch ein Überblick über bestehende Arbeiten gegeben werden. Weitere Untersuchungen über die beste Größe und Organisationsform eines Datencaches zur Minimierung des Energieverbrauchs wurden auch von Shiue et al. [SC99] durchgeführt. Für unterschiedliche Benchmarks wurden die Cachegröße, die Cachelinegröße, die Anzahl der Sets und die Tiling-Größe bestimmt. Unter Tiling versteht man hier die Bearbeitung eines Arrays in einzelnen Abschnitten (=Tiles), in die das Array aufgeteilt wird. Optimierungen des Caches auf Performance führten nicht automatisch zur Reduzierung der Energie. Daher muss Energie als eigenständiges Optimierungsziel betrachtet werden.

Eine Speicherhierarchie kann neben schon bestehenden Komponenten wie L1- und L2-Cache noch um weitere Bestandteile wie beispielsweise Buffer, die parallel zu den Caches eingesetzt werden, ergänzt werden. In [BAM98] wird gezeigt, dass in Abhängigkeit der Strategie für die Nutzung des Buffers wie *penalty*, *speculative* oder *non-temporal* weitere Energieeinsparungen für Daten- und/oder Instruktionscaches möglich sind. Die Verbesserungen reduzieren den Energieverbrauch in der Speicherhierarchie durchschnittlich zwischen 5% und 20%.

Die Wichtigkeit der Speicheroperationen wird auch bei Sinha et al. [SC01] deutlich, die Instruktionen für die Modellierung des Energieverbrauchs nach der Art von Speicherzugriffen klassifiziert haben. Weitere Arbeiten über den Zusammenhang und Einfluss verschiedener Optimierungen und die Cacheorganisationen finden sich bei Kandemir et al. [KVIY00].

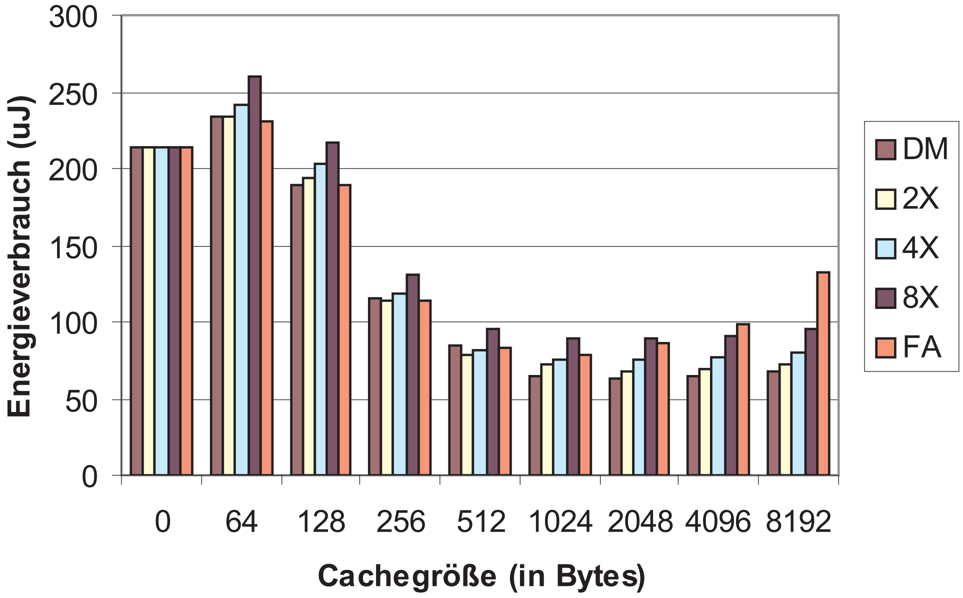


Abbildung 4.20: Energieverbrauch bei unterschiedlichen Caches bei *quicksort*

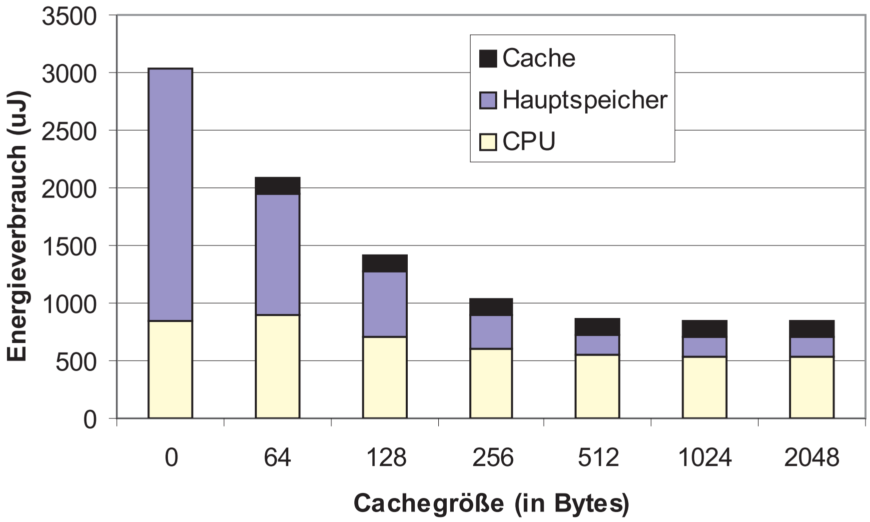


Abbildung 4.21: Energieverteilung im System bei *insertionsort*

Fazit

Der Einsatz von Caches bei Systemen, die auf Energieverbrauch zu optimieren sind, ist vorteilhaft und somit sinnvoll. Die möglichen Einsparungen sind größer als der Einfluss auf die Performance und führen zu einer deutlichen Reduzierung der Hauptspeicherenergie (siehe Abb. 4.21). Allerdings ist die optimale Größe des Caches von der Applikation abhängig. Ggf. kann sogar der gegenteilige Effekt erreicht werden. Für Energieoptimierungen ist bei der Cacheorganisation eine geringe Set-Assoziativität die meistens bessere Wahl als bei Performance-Optimierungen und insbesondere der direct-mapped Cache häufig die beste Form. Auswirkungen auf den Compiler und dessen Strategien hat die Wahl eines Caches kaum. Grundsätzlich ist das zu generierende Programm unabhängig vom Einsatz eines Caches, da die Cachesteuerung, die für die Software transparent ist, dies übernimmt. Ausnahmen bilden einige Optimierungen, die das Layout der Daten so optimieren, dass zum Beispiel die gleichzeitig verwendeten Elemente verschiedener Arrays gleichzeitig im Cache gehalten werden können [SC99].

Die hardwaremäßige Steuerung ist gleichzeitig aber auch der Schwachpunkt des Caches, da mit hohem Energieaufwand bei jedem Cachezugriff festgestellt werden muss, ob das gewünschte Datum im Cache aktuell vorhanden ist. Dieser hohe Energieeinsatz für das Holen der Tagadresse aus dem Tagspeicher und den Vergleich der Tags bilden ein Potenzial an Energieverbrauch, welches bei der Verwendung eines Scratchpad-Speichers zur Optimierung eingespart werden kann. Techniken, die sich dieses zunutze machen, werden im folgenden Kapitel vorgestellt.

Kapitel 5

Scratchpad-Organisation

Bisher wurde untersucht, wie sich die neben dem Hauptspeicher häufig eingesetzten Caches für die Reduzierung des Energieverbrauchs verwenden lassen. Im nächsten Schritt betrachten wir als alternative Lösung in der Speicherhierarchie die Verwendung eines Scratchpad-Speichers (siehe Abb. 4.4). Der Hauptvorteil eines Scratchpads ist der geringe Energieverbrauch für einen Zugriff im Vergleich zum Cache. Als Nachteil ist festzustellen, dass im Gegensatz zur in Hardware implementierten Steuerung des Caches diese Logik beim Scratchpad-Speicher in Software nachgebildet werden muss und daher auch Ausführungszeit und Energie kostet. Um die Laufzeit bei der Ausführung des Programms nicht unnötig zu verlängern und den Energieverbrauch zu erhöhen, ist es essentiell, dass die notwendigen Berechnungen und Entscheidungen so weit wie möglich bereits vor der Ausführung, d. h. zur Compilezeit, vorgenommen werden.

Grundidee

Die Grundidee des hier präsentierten neuen Verfahrens ist die Verlagerung von Programmteilen[1] in den Scratchpad-Speicher zusätzlich zu der schon bekannten Verlagerung von Daten. Diese Verlagerung geschieht als Alternative zu einem automatischen Einlagern in einen Cache. Das Programm wird im Hauptspeicher ausgeführt und bei Erreichen des betreffenden Programmteils durch einen zusätzlichen Sprung in den Scratchpad dort fortgesetzt. Nach Ausführung des Programmteils im Scratchpad erfolgt ein Rücksprung in den Hauptspeicher und die dortige Fortführung der Programmabarbeitung. Die Programmteile, die den größten Vorteil versprechen, werden zur Compilezeit ausgewählt und später während des Ladens bzw. während der Ausführung des Programms in den Scratchpad-Speicher kopiert.

Eine grundsätzliche Alternative besteht in der Verwendung des Scratchpad-Speichers. Es können 1. Teile des Programms dauerhaft in den Scratchpad verlagert werden oder 2. diese Programmteile während der Programmlaufzeit ausgetauscht werden. In den folgenden Abschnitten werden beide Alternativen vorgestellt:

1. statische Variante: Zur Compilezeit werden Programmteile ausgewählt, die bei einer dauerhaften Verlagerung die größtmögliche Energieeinsparung liefern. Die Verlagerung selbst geschieht zum Zeitpunkt des einmaligen Ladens des Programms ohne Zusatzkosten.
2. dynamische Variante: Zur Compilezeit werden Programmteile ausgesucht, die vorübergehend in den Scratchpad kopiert werden. Die optimale Auswahl und die Zeitpunkte zur Verdrängung durch andere Programmteile werden ebenfalls bestimmt. Allerdings muss das Kopieren des Programmteils bei der Kostenbestimmung berücksichtigt werden, damit nur Teile verlagert werden, die zur Energieeinsparung beitragen.

[1] Folgen von Instruktionen in Form von Basisblöcken oder Funktionen

Überblick

Vor der Präsentation der Verfahren zur Ausnutzung von Scratchpads wird zuerst im Unterkapitel 5.2 das bisherige Energiemodell für den Scratchpad erweitert, um den Energieverbrauch eines Cache-basierten Verfahrens mit einem Scratchpad-basierten Verfahren vergleichen zu können. Da die Verwendung von Cache und Scratchpad Alternativen darstellen, muss auch das neue Verfahren für den Scratchpad mit einem üblichen Cache-System bezüglich des Energieverbrauchs bewertet werden. Ein Vergleich der Scratchpad-Variante gegenüber einem System, welches lediglich einen Hauptspeicher enthält, wäre eindeutig vorteilhaft für den Scratchpad. Allerdings ist dieser Vergleich nicht relevant, da man in Systemen, in denen Onchip-Speicher verwendet werden können, heutzutage Caches einsetzt, die den reinen Hauptspeicher-basierten Systemen überlegen sind.

Im Unterkapitel 5.3 wird die Programmanalyse, die für alle Varianten des Verfahrens gleich ist, beschrieben. Hierzu gehört die Identifizierung der Objekte und die Berechnung der Aufrufhäufigkeit und des eingesparten Energieverbrauchs beim Verschieben des Objektes.

Nach diesen Vorarbeiten werden dann das statische Verfahren im Unterkapitel 5.4 und das dynamische Verfahren im Unterkapitel 5.5 präsentiert.

Vorab wird jedoch noch ein Überblick über den Stand der Forschung gegeben.

5.1 Verwandte Arbeiten

Die Betrachtung der effizienten Nutzung einer Speicherhierarchie wurde von Panda et al. [PDN97, PDN99] untersucht. Diese Arbeiten beschränken sich auf die Auslagerung von Daten in den Scratchpad-Speicher, allerdings ohne Berücksichtigung von Programmteilen. Die vorgestellte Technik partitioniert die skalaren und Arrayvariablen mit dem Ziel der Minimierung der Ausführungszeit. Ein weiterer Ansatz ist bei Sjödin et al. [SFL98] zu finden, der auf der Basis einer statischen Analyse ebenfalls eine Menge von Variablen in den Scratchpad verlagert. Es wird gezeigt, dass eine statische Analyse ausreichend präzise ist und keine dynamische Analyse während des Programmlaufs oder auf Basis ausgeführter Beispielprogramme benötigt wird.

Andere Ansätze beinhalten die Generierung oder Modifikation von Hardware und fallen damit in die Kategorie der applikationsspezifischen Hardwaresynthese. Ishihara et al. [IY00] zeigen einen Ansatz, in dem häufig ausgeführte Instruktionssequenzen identifiziert und zu einer Menge von zusätzlichen Instruktionen zusammgefasst werden. Die Instruktionssequenzen werden zur Laufzeit durch einen Decompressor wieder restauriert. Ein weiterer Ansatz von Benini et al. [BMMP00] generiert applikationsspezifische Speicher, die zusätzlich zu einem Scratchpad-Speicher noch Decoder zur Entscheidung über Hit oder Miss beinhalten. Der energieintensive Zugriff über einen Tag-Speicher wie bei einem Cache kann entfallen. Die Ergebnisse zeigen eine Reduzierung des Energieverbrauchs zwischen 12% und 16%.

Bisherige Arbeiten sind auf die Verlagerung von Daten beschränkt oder beinhalten auch die applikationsspezifische Modifikation von Hardware. Eine neuere Arbeit von Kandemir et al. [KRI$^+$01] optimiert Array-Zugriffe, indem Teile des Arrays dynamisch in einen Scratchpad verlagert werden. Diese Arbeit setzt einige Besonderheiten bezüglich der Schleifenkonstruktion voraus und betrachtet auch nur das Handling von Teilen des Arrays (so genannte Array-Tiles), die lokal zur Berechnung benötigt werden.

5.2 Aufbau und Energieverbrauch

Zuordnung von Funktionalitäten zwischen Cache und Scratchpad

Im Gegensatz zum Cache werden einige Aufgaben bei der Verwendung eines Scratchpad-Speichers nicht in Hardware ausgeführt, sondern müssen zur Compilezeit bereits entschieden und berücksichtigt werden. Es handelt sich um die folgenden Cache-Charakteristika:

1. Gültigkeit / Lokalität eines Datums
   Während der Cache überprüft, ob ein Datum derzeitig im Cache vorliegt, wird dies bei einem Scratchpad-Zugriff nicht berücksichtigt und muss an anderer Stelle entschieden werden. Der Energieverbrauch für diese Überprüfung im Cache kann potenziell eingespart werden.

2. Tagspeicher
   Der Tagspeicher dient zur Speicherung der Adressen der aktuell im Cache vorhandenen Daten. Wenn die Gültigkeit und ihre wiederholte Überprüfung an anderer Stelle geregelt wird, kann der Tagspeicher vollständig entfallen.

3. Cache-Misses
   Das Nachladen von Daten aus dem Hauptspeicher erfolgt beim Cache bei einem Lese-Zugriff auf ein Datum, welches aktuell nicht im Cache vorliegt. Dadurch wird ein automatisches Nachladen ausgelöst, welches in den meisten, aber nicht in allen Fällen hilfreich ist. Das Nachladen eines Datums, welches nur einmalig benötigt wird, ist beispielsweise von Nachteil, da es günstiger wäre, für diese einmalige Verwendung das Datum direkt aus dem Hauptspeicher in den Prozessor zu transferieren.
   Beim Schreib-Zugriff auf nicht im Cache vorhandene Daten erfolgt bei der hier betrachteten No-write-allocate-Organisation kein Cache-Zugriff. Die Daten werden lediglich in den Hauptspeicher geschrieben. Im Gegensatz dazu fordert die Write-allocate-Organisation in diesem Fall eine Cacheline an, in die die Daten dann geschrieben werden.

4. Cachelines
   Es wird beim Cache nicht nur ein einzelnes Datum, sondern stets ein Block nachgeladen. Dies ist in den meisten Fällen günstiger, da nur ein Tag im Cache für einen vollständigen Block benötigt und damit auch die Anzahl der Vergleiche bei jedem Cachezugriff reduziert wird. Abhängig vom System kann auch das Nachladen von aufeinander folgenden Adressen in einem Block aus dem Hauptspeicher günstiger sein. Allerdings kann auch der Fall eintreten, dass Daten mitgeladen werden, die niemals zur Verwendung gelangen. Dieses Potenzial der Einsparung von überflüssigen Einlagerungen in den Onchip-Speicher kann zur Optimierung genutzt werden.

5. Set-Assoziativität
   Die Adressen verschiedener Datenworte, die in den Cache geladen werden, können zur Kollision führen, d. h. zu einer Verdrängung. Wenn sich die Adressen zweier Datenworte in ihrer Indexadresse nicht unterscheiden, wird bei einem direct-mapped Cache bei einem Zugriff auf eines der beiden Daten jeweils das andere Datum verdrängt. In der nächsten Stufe kann bei einem set-associative Cache zwischen mehreren Lines ausgewählt werden. Die beliebige Kombination von Daten bis zur Kapazitätsgrenze des Caches kann nur bei einem full associative Cache erreicht werden. Andererseits hat ein full associative Cache, bzw. als Zwischenstufe die set-associative Caches, den Nachteil, dass diese gewonnene Freiheit durch zusätzliche Hardware erkauft wird und nicht nur die Chipfläche vergrößert, sondern dadurch auch den Energieverbrauch erhöht.

Diese aufgezeigten negativen Eigenschaften des Caches entfallen beim Scratchpad und bieten ein Potenzial zur Energieeinsparung. Durch das Entfallen dieser Eigenschaften und der entsprechenden Hardware-Komponenten reduziert sich der Energieverbrauch beim Zugriff auf den Scratchpad-Speicher gegenüber dem eines Caches, sodass dessen Energiekosten im Folgenden neu berechnet werden müssen. Später müssen aber noch die folgenden Aufgaben neu organisiert und zugeordnet werden:

1. Ein-/Auslagern von Daten in den Scratchpad
   Diese beim Cache automatisch ausgeführte Funktion muss entweder einmalig vor Start der Applikation oder aber auch während des Programmlaufs durch Software ersetzt werden.

2. Berücksichtigung des Speicherorts des Datums
   Während bei dem Einsatz eines Caches die Prozessoradressen unverändert bleiben können, bedeutet das Verschieben in den Scratchpad auch die Änderung der Adressen. Dies muss bei der Programmgenerierung durch den Compiler oder durch andere Mechanismen erreicht werden.

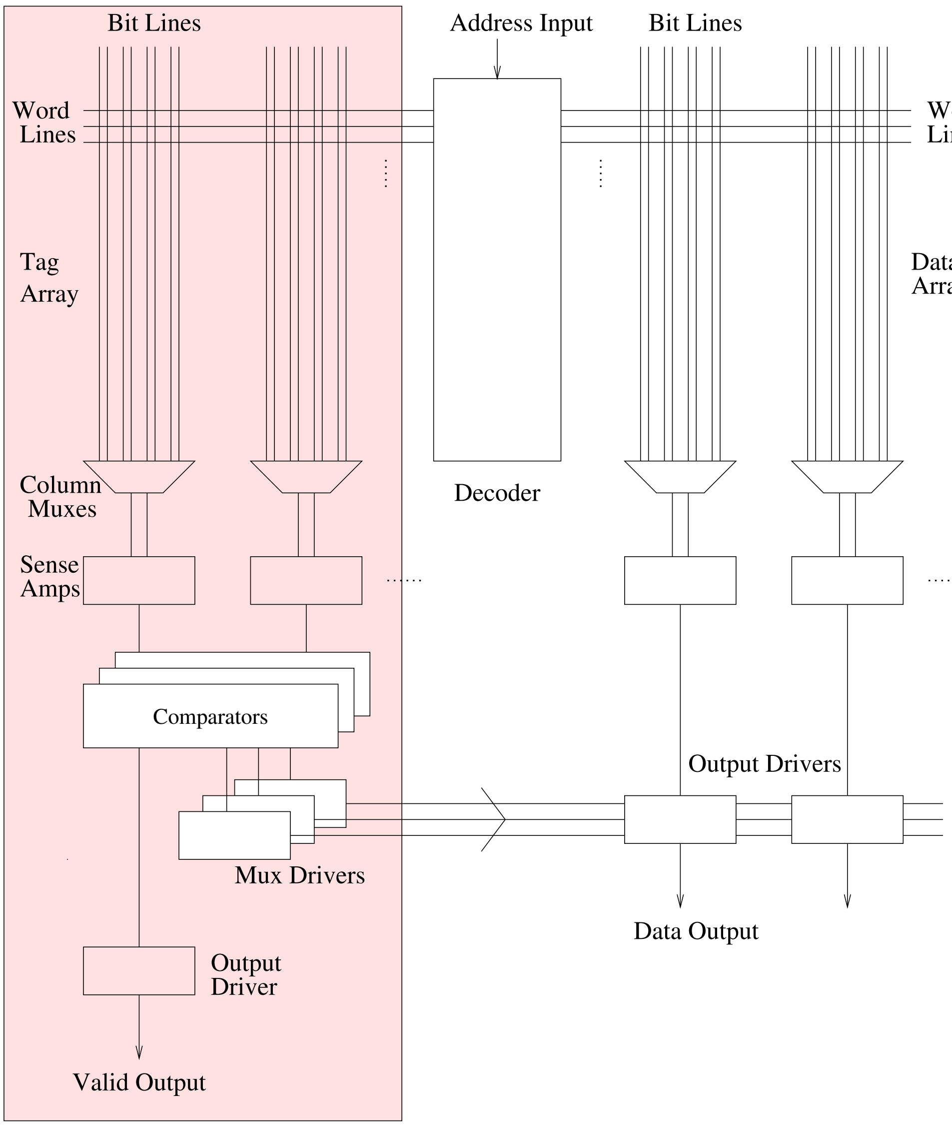


Abbildung 5.1: Scratchpad-Modell als Subset von CACTI

Methode zur Bestimmung des Energieverbrauchs

Der Energieverbrauch beim Zugriff auf den Speicher kann auf zwei unterschiedliche Arten bestimmt werden: durch eine Messung von realen Speicherbausteinen oder durch den Einsatz eines analytischen Modells wie beispielsweise CACTI. Die erste Methode, die Messung von realen Speicherbausteinen, müsste für einen Cache und für einen Scratchpad auf die gleiche Art und Weise erfolgen, um vergleichbare Verhältnisse zu erreichen, wobei darauf zu achten ist, dass beide in der gleichen Technologie hergestellt sein müssen. Diese Methode durch Messung ist im vorliegenden Fall nur sehr schwer möglich, da es sich bei den Speichern um Onchip-Speicher handelt und diese mit dem Prozessor zusammen auf einem Chip untergebracht sind. Daher kann die Messung nur gemeinsam mit dem Prozessor erfolgen, wenn nicht ausnahmsweise getrennte Stromversorgungspins zur Verfügung stehen, welches aber bei den betrachteten Chips nicht zutrifft. Wenn man den Stromverbrauch des Scratchpad zusammen mit dem Verbrauch des Prozessors misst, kommt es zwangsläufig zu Ungenauigkeiten, da der Verbrauchsanteil des Scratchpads im Verhältnis zum Prozessorverbrauch sehr gering ist. Weiterhin wurden keine Chips gefunden, die bei gleicher Technologie in beiden Varianten, mit Scratchpad und mit Cache, gefertigt werden. Aus diesen Gründen erfolgt die Berechnung des Energieverbrauchs des Scratchpad ebenso wie die des Caches mit Hilfe des analytischen Modells von CACTI, um eine Vergleichbarkeit zu gewährleisten.

Da das CACTI-Modell ursprünglich nur für Caches entwickelt wurde, müssen die Bestandteile, die im Scratchpad nicht enthalten sind, entfernt werden [BSL+01, BSL+02]. Dadurch entfallen die in Abbildung 5.1 grau hinterlegten Bestandteile bei der Berechnung des Energieverbrauchs des Speichers. Als Organisationsform für den Cache ist direct-mapped zu wählen, damit die Organisation der verbleibenden Komponenten einem Scratchpad-Speicher entspricht.

Speichergröße	Scratchpad	Cache (4X)	Verhältnis
64 Bytes	0,49 nJ	2,87 nJ	1:5,86
128 Bytes	0,53 nJ	3,15 nJ	1:5,94
256 Bytes	0,61 nJ	3,32 nJ	1:5,44
512 Bytes	0,69 nJ	3,48 nJ	1:5,04
1.024 Bytes	0,82 nJ	3,75 nJ	1:4,57
2.048 Bytes	1,07 nJ	4,04 nJ	1:3,78
4.096 Bytes	1,21 nJ	4,71 nJ	1:3,89
8.192 Bytes	2,07 nJ	5,39 nJ	1:2,60

Tabelle 5.1: Energieverbrauch pro 32-Bit-Speicher-Zugriff (0,5 μm Technologie)

Der Energieverbrauch berechnet sich analog zu der Cache-Modellierung nach der folgenden Summenformel, in der der Verbrauch der Einzelkomponenten aufaddiert wird. Einzelne Komponenten entfallen vollständig, andere repräsentieren nur noch einen Teil der entsprechenden Cache-Komponente, weil z. B. der Tagspeicher entfallen ist:

$$E_{SP} = E_{DataArray} + E_{Decoder} + E_{WordLines} + E_{BitLine} + E_{ColumnMux} + E_{SenseAmp} + E_{OutputDrv}$$

Durch die Summierung der verbliebenen Komponenten erhält man für die unterschiedlichen Größen des Scratchpads die Energiewerte in Tabelle 5.1 für einen einzelnen Speicherzugriff.

Zu Vergleichszwecken sind auch noch die Energieverbrauchswerte eines 4-way set-associative Caches sowie das Verhältnis des Energieverbrauchs zwischen diesen Onchip-Speichern dargestellt. Es ist deutlich zu erkennen, dass der Cache einen zwischen 2,6 und 5,9-fach höheren Energieverbrauch im Vergleich zum Scratchpad aufweist. Mit den im Folgenden vorgestellten Techniken wird diese Differenz ausgenutzt, um insgesamt eine Energiereduzierung bei der Abarbeitung eines Programmes zu erreichen. Allerdings muss ein Teil der Differenz investiert werden, um die entfallenen Hardware-Funktionalitäten des Caches durch Software nachzubilden.

Im folgenden Unterkapitel wird die Analysemethode des auszuführenden Programms zur Vorbereitung der Anwendung der Compilertechniken vorgestellt.

5.3 Analyse des Programms

Zur Vorbereitung auf die Techniken zur Verwendung eines Scratchpad-Speichers muss das auszuführende Programm zuerst analysiert werden. Bei der Beschreibung dieser Programmstruktur wird die Programmiersprache C beispielhaft verwendet. Die Übertragung auf andere imperative Programmiersprachen kann entsprechend erfolgen. Zu unterscheiden sind dafür die folgenden Objekte, die im Speicher angesiedelt sind:

```
int main(void) {
  convert();
  for (i = 0; i < 10; i++) {
    pin_down();
  };
  convert();
};

pin_down() {
  ..
  convert();
};

void convert() {
};
```

Abbildung 5.2: Beispielprogramm mit Funktionen

1. Funktionen
   Je nach Programmiersprache besteht ein Programm aus einer Menge von Funktionen bzw. Prozeduren. Eine ausgezeichnete Funktion “`main`” stellt die zuerst aufgerufene Funktion des Programms dar. Die Schnittstelle zwischen den Funktionen ist eindeutig geregelt. Beim Aufruf einer Funktion werden die Parameter übergeben, die in dem Funktionsaufruf mit angegeben werden. Diese können, je nach Art des Datentyps und der Deklaration, teilweise innerhalb der aufgerufenen Funktion verändert werden. Begonnen wird die Ausführung einer Funktion stets mit der ersten Instruktion. Die Beendigung der Abarbeitung der aufgerufenen Funktion erfolgt durch eine der möglicherweise mehrfach vorhandenen Rücksprunganweisungen. Sprünge von außerhalb zu einer anderen Instruktion als der ausgezeichneten ersten Instruktion einer Funktion sind nicht zugelassen bzw. nach üblichen Codierungsrichtlinien nicht möglich.

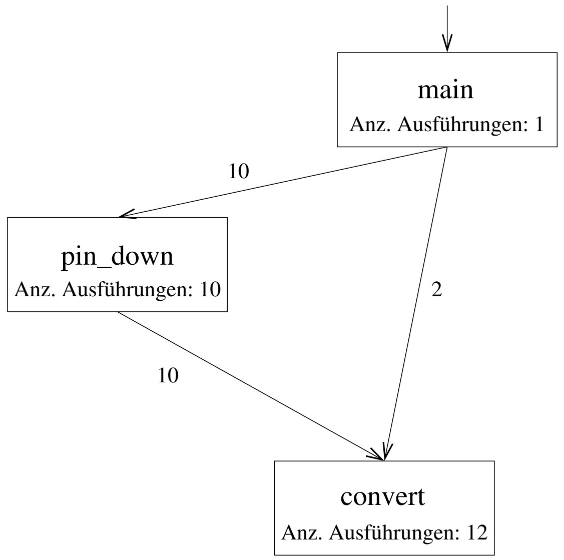


Abbildung 5.3: Funktionsaufrufgraph

Eine Besonderheit ist, dass eine Funktion den Gültigkeitsbereich von lokalen Daten bzw. Variablen darstellt, die von außerhalb nicht erreicht werden können. Diese werden später unter dem Stichwort "lokale Daten" behandelt.

Der Zusammenhang der Aufrufe zwischen Funktionen kann durch einen Funktionsaufrufgraphen repräsentiert werden. Ein Funktionsaufrufgraph, der die Funktionsaufrufe für das Beispielprogramm in Abbildung 5.2 zeigt, ist in Abbildung 5.3 dargestellt. Die Funktionsaufrufe können durch eine statische Analyse des Programmcodes ermittelt werden. Weiterhin ist die Häufigkeit der Ausführung jeder Funktion sowie der Anzahl der Aufrufe mit angegeben. Diese Häufigkeit kann auf unterschiedliche Arten ermittelt werden:

(a) statisch
Das Programm wird analysiert und zählt die Anzahl der Funktionsaufrufe. Erweiterte Algorithmen können auch die Häufigkeit von Schleifendurchläufen ermitteln, wie beispielsweise im Beispielprogramm in Abbildung 5.2, wo die for-Schleife 10-mal ausgeführt wird. Nur begrenzt möglich ist diese Analyse bei Datenabhängigkeiten.

(b) dynamisch
Für diese Variante wird das Programm übersetzt und mit einem Simulator ausgeführt. Ein hierbei erzeugter Trace wird ausgewertet und die Anzahl der Ausführungen der einzelnen Funktionsaufrufe und der Funktionen ermittelt. Die hierbei ermittelten Ausführungshäufigkeiten sind von den Eingabedaten abhängig und nur dafür gültig. Um verlässlichere Häufigkeiten zu erhalten, muss das Programm mit unterschiedlichen Eingabedaten simuliert und daraus der Mittelwert gebildet werden. Der Vorteil der dynamischen Variante ist die höhere Präzision der Daten im Vergleich zur statischen Bestimmung. Nachteilig ist die Notwendigkeit, dass der Aufwand für die Compilierung des Programms, die Simulation und die Auswertung des Traces investiert werden müssen.

Mit diesen Analysen liegen alle Informationen über die Größe und Häufigkeit einzelner Funktionen vor. Die Größe der Funktionen wird bei einem Verschieben nicht verändert, da sowohl Einsprung als auch Rücksprung unverändert gelassen werden können. Lediglich die Einsprungadresse muss beim Verschieben angepasst werden.

Es fehlt nun noch die Bestimmung des Energieverbrauchs, der jedoch von der Wahl des Verfahrens des Kopierens der Funktionen in den Scratchpad, statisch oder dynamisch, abhängt und daher in den betreffenden Kapiteln erläutert wird.

2. Basisblöcke

```
BB_main:         int main(void) {
                   convert();
BB_main2:          for (i = 0; i < 10; i++) {
BB_main_loop:        pin_down();
BB_main3:          };
BB_main4:          convert();
BB_main_end:     };

BB_pindown:      void pin_down() {
                   ..
BB_pindown2:       convert();
BB_pindown_end: };

BB_convert:      void convert() {
                 };
```

Abbildung 5.4: Beispielprogramm mit Basisblöcken

Funktionen bestehen wiederum aus einer Menge von Basisblöcken. Basisblöcke, deren Kontrollfluss linear ohne Ein- oder Aussprung ist, haben eine Ausführungshäufigkeit, die auch für alle enthaltenen Instruktionen identisch ist. Es kann innerhalb von Funktionen Basisblöcke geben, die sehr häufig durchlaufen werden und deren Kopieren in den Scratchpad-Speicher lohnend ist, während andere Basisblöcke, die nur selten durchlaufen werden, günstiger im Hauptspeicher verbleiben. Die Betrachtung auf Basisblockebene anstatt auf Funktionsebene erlaubt daher die Behandlung kleinerer Einheiten und eine bessere Differenzierung zwischen häufig und selten ausgeführten Instruktionen.

Am Beispiel des Programms in Abbildung 5.4 wird ein Kontrollflussgraph für Funktionen und die Zerlegung in Basisblöcke in Abbildung 5.5 dargestellt. Die unterschiedlichen Häufigkeiten der Ausführung der Basisblöcke sind jeweils an den Kanten annotiert. Beispielsweise wird innerhalb der Funktion "main" der Basisblock "BB_main_loop" 10-mal ausgeführt, hingegen der Basisblock "BB_main" nur einmal. Dadurch kann es sinnvoll sein, den Basisblock "BB_main_loop", aber nicht Basisblock "BB_main", zu verschieben. Weiterhin kann man am Beispiel des Basisblockes "BB_main_loop" sehen, dass er einmal vom vorhergehenden Basisblock "BB_main2" und 9-mal durch den Rücksprung aus Basisblock "BB_main3" angesprungen wird. Diese unterschiedlichen Vorgänger-Basisblöcke sind für die spätere Programm-Modifikation von Bedeutung und müssen daher unterschieden werden. Ebenso kann ein Basisblock ein oder zwei Nachfolger-Basisblöcke besitzen, wie man am Beispiel des Basisblocks "BB_main3" sehen kann. Er selbst wird 10-mal durch den Rücksprung aus der Funktion "pin_down" beginnend mit der ersten Instruktion ausgeführt und gibt die Programmkontrolle in 9 Fällen durch einen bedingten Sprungbefehl an "BB_main_loop" weiter oder, wenn der Schleifenzähler seinen Maximalwert erreicht hat, an den nachfolgenden Block "BB_main4".

Durch diese Analysen und die zusätzliche Angaben über die Größe der einzelnen Basisblöcke können die unterschiedlichen Optimierungen die bestmöglichen Programmteile auswählen, um eine maximale Energieeinsparung zu erzielen. Bei den Basisblöcken ist im Gegensatz zu den Funktionen jedoch zu berücksichtigen, dass sie nicht unbedingt durch einen Sprungbefehl angesprungen werden, sondern dass dies auch durch die lineare Abarbeitung der Instruktionen vom vorhergehenden Basisblock aus geschehen kann. Ebenso kann auch das Verlassen des Basisblockes ohne Sprungbefehl sequentiell in den nachfolgenden Basisblock geschehen. Eine Verschiebung des Basisblockes

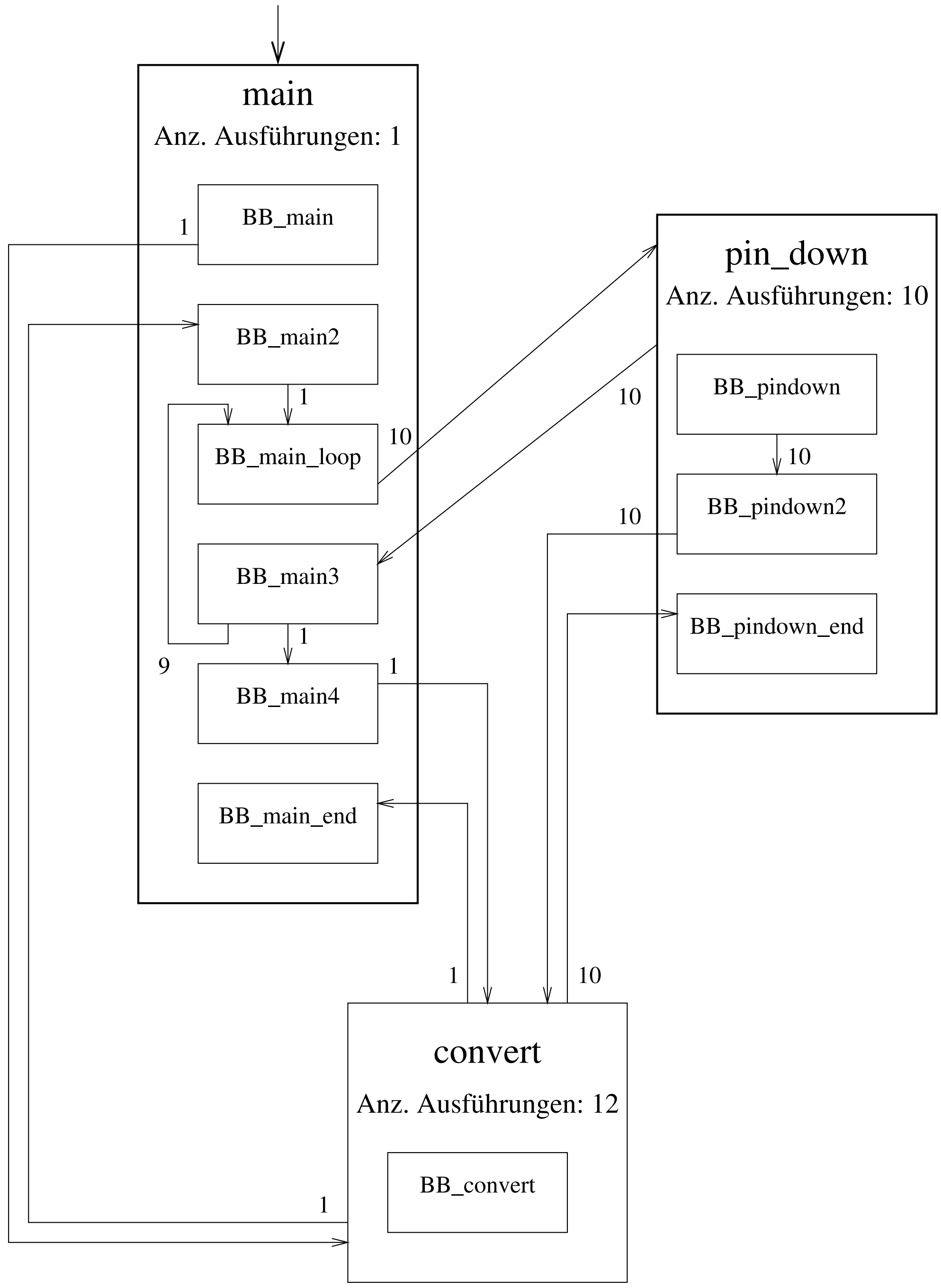


Abbildung 5.5: Kontrollflussgraph

erfordert in diesen Fällen zusätzliche Sprungbefehle oder auch bei einigen relativen Sprüngen, deren Sprungdistanz begrenzt ist, eine Verlängerung durch Einfügung weiterer Instruktionen. Diese Zusatzbefehle bedeuten einen Overhead und müssen bei der Energiebetrachtung berücksichtigt werden.

3. Lokale Daten

   Wie schon erwähnt, können Funktionen in der Programmiersprache "C" innerhalb von Funktionen lokale Variablen verwenden. Die Gültigkeit der Variablen ist auf die Funktion begrenzt. Bei hierarchischen Aufrufen einer Funktion, z. B. durch Rekursion, müssen auch für jede gültige Ebene der Funktion diese lokalen Variablen getrennt vorhanden sein. Die meisten Compilern legen daher diese lokalen Variablen auf dem Stack[2] ab, da man im aktuellen Stackframe während der Laufzeit vorübergehend leicht Platz für die Speicherung der lokalen Variablen erhalten kann und auch die mehrfache Instanziierung der Funktion durch den Auf-/Abbau des Stackframes automatisch berücksichtigt wird. Weiterhin erfolgt bei einem Stack der Zugriff auf die Variablen relativ zu einem Stackpointer, was einfach und effizient möglich ist. Die Verschiebung von lokalen Daten in den Onchip-Speicher ist daher nur indirekt über das Verschieben des Stackpointers möglich, sofern nicht die vollständige Adressierung der lokalen Variablen modifiziert werden soll. Es ist daher zu prüfen, ob der Stack vom Hauptspeicher auf den Onchip-Speicher verlagert werden soll. Dies kann vollständig geschehen oder auch erst ab einer gewissen Tiefe bzw. Funktion. Die weitere Behandlung dieser Thematik erfolgt später unter dem Thema Stack auf Seite 109.

4. Globale Variablen

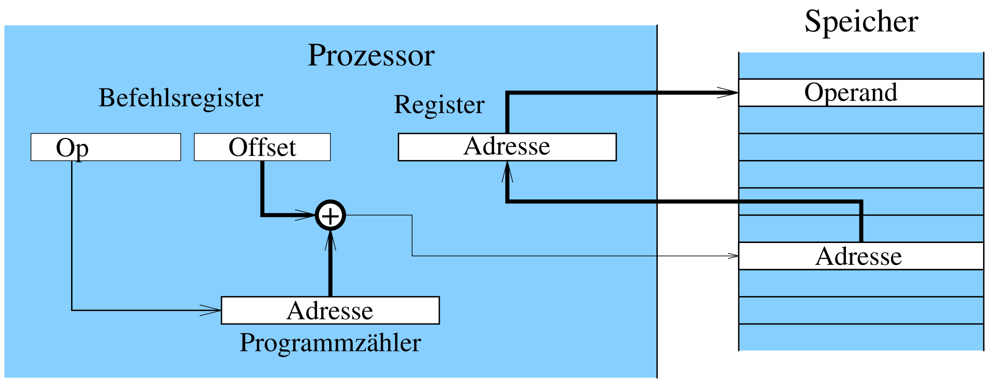


Abbildung 5.6: indirekte PC-relative Adressierung

   Neben der Möglichkeit, lokale Variablen zu deklarieren, können Programme auch globale Variablen verwenden. Globale Variablen sind von allen Funktionen aus zugreifbar und existieren nur in genau einer Instanz. Da diese Variablen während des kompletten Programmlaufs gültig sind, werden sie nicht nur temporär wie die lokalen Variablen während der Abarbeitung einer Funktion auf dem Stack angelegt. Der Compiler generiert vielmehr ein Datensegment im Speicher, in welchem die globalen Variablen gesammelt und vom Linker einem ReadWrite-Speicherbereich zugeordnet werden. Da das Programm selbst in einem Codesegment verwaltet wird, welches nur ReadOnly-Zugriffsrechte besitzt, und das Segment im Adressbereich weit entfernt liegen kann, erfolgt der Zugriff auf globale Variablen bei den hier betrachteten Compilern mit einer indirekten PC-relativen Adressierung (2-stufige Speicheradressierung [Bäh91]). In Abbildung 5.6 ist beispielhaft ein Zugriff auf einen

[2]Eine Datenstruktur zum Speichern von Objekten mit last-in first-out Ordnung (LIFO). Ein Stack wird verwendet, um die Reihe aufgerufener Funktionen eines Programmes zu verfolgen. Ein neues Objekt wird mit "Push" hinzugefügt und das oberste Objekt mit "Pop" herausgeholt [Com].

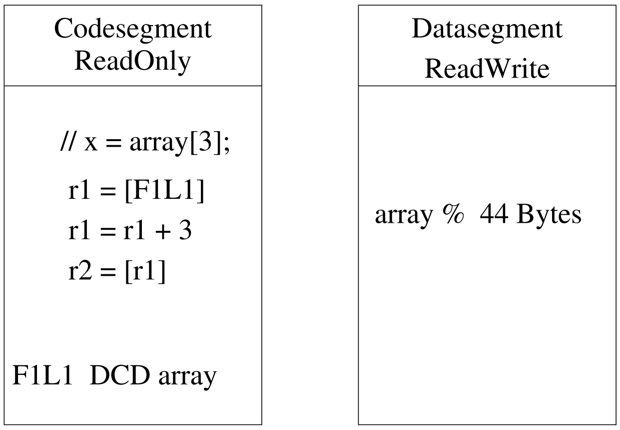


Abbildung 5.7: Arrayzugriff beim ARM7

globalen Operanden dargestellt. Zuerst muss die Adresse des Operanden aus dem Speicher in ein Register geladen werden, indem zum Programmzähler ein Offset addiert und aus dieser berechneten Adresse der Inhalt geladen wird. Mit dieser Adresse im Register kann dann im zweiten Schritt auf den zugehörigen Operanden an dieser Adresse zugegriffen werden.

Der konkrete Assemblercode für den ARM7TDMI bei einem Zugriff auf ein Arrayelement ist in Abbildung 5.7 dargestellt. Zur Adressierung wird ein Offset als Bestandteil des Instruktionswortes zum Programmzähler addiert. Als Ergebnis wird aus dieser Adresse *F1L1* die Startadresse des Arrays geladen und in das Register *R1* geholt. Um nun auf ein Element des Arrays mit dem Index *3* zuzugreifen, muss der entsprechende Offset *3* auf die im Register *R1* gehaltene Basisadresse addiert werden. Mit dieser Adresse kann nun ein Ladebefehl ausgeführt werden und das gewünschte Arrayelement aus dem ReadWrite-Datensegment in Register *R2* geholt werden.

Wenn nun ein Array in den Onchip-Speicher verschoben wird, bedeutet dies, dass das Array in einem anderen Datensegment, welches dem Onchip-Speicher zugeordnet wird, liegt. Weitere Anpassungen sind nicht notwendig. Im Gegensatz zur Verschiebung von lokalen Variablen in den Onchip können globale Daten ohne Veränderung der Instruktionsanzahl verschoben werden.

Für die Berechnung des Energievorteils durch Verschieben von globalen Daten in den schnelleren und energiesparenderen Onchip-Speicher entsteht der Unterschied allein bei den Load und Store-Zugriffen auf diese Daten. Dieser Unterschied zeigt sich ggf. beim Prozessor, der weniger Wartezyklen benötigt und damit auch weniger Energie, und beim Speicher, da der Onchip-Speicherzugriff weniger Energie benötigt als der Hauptspeicherzugriff.

5. Stack

Als letztes Objekt wird der Stack betrachtet, der vollständig oder in Teilen auch in den Scratchpad-Speicher verschoben werden kann. Der Stack dient zur Speicherung unterschiedlicher Daten, die im Folgenden einzeln betrachtet werden sollen:

- lokale Daten
  Wie schon beschrieben, enthält der Stack die lokalen Variablen, die für jede gerade ausgeführte Instanz einer Funktion vorhanden sein müssen.
- Funktionsparameter
  Funktionen können beim Aufruf Parameter erhalten, die in Registern oder ab einer gewissen Grenze über den Stack übergeben werden.

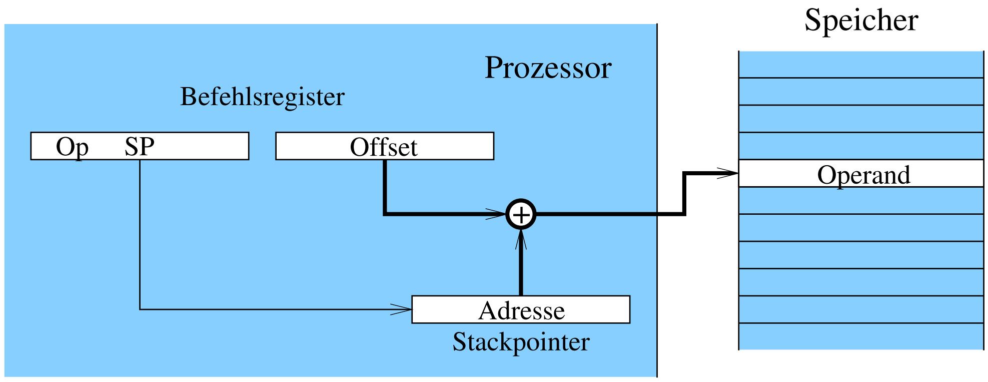


Abbildung 5.8: Stackpointer-relative Adressierung

- Rücksprungadresse
  Beim Aufruf einer Funktion muss die Adresse, an der nach Beendigung der Funktionsausführung die Programmabarbeitung fortgesetzt werden soll, gespeichert werden. Je nach Prozessor kann beispielsweise die Rücksprungadresse wie beim ARM-Prozessor in einem expliziten Register, dem Link-Register, gespeichert werden. In den Fällen, in denen aus dieser Funktion wiederum noch eine Funktion aufgerufen wird, muss der Inhalt des Link-Registers auf den Stack gesichert werden. Ansonsten kann für den Rücksprung am Ende der Funktion das Link-Register in den Programmzähler kopiert werden.
- Registerspilling
  Der letzte Typ von Daten, die auf dem Stack gesichert werden, entsteht durch Registerspilling. Register werden temporär auf den Stack ausgelagert, wenn die Anzahl der Register nicht ausreicht und daher ein Spilling erfolgen muss.

Die Verwaltung der Daten auf dem Stack ist einfach möglich. Für jede Instanz einer Funktion wird der Stackpointer um die benötigte Größe beim Einsprung bzw. bei der Beendigung der Funktion versetzt. Es wird nur für die Funktionen Speicher vorgehalten, die auch aktuell abgearbeitet werden. Daher ist der Speicherplatzbedarf auf dem Stack minimal. Ein letzter Vorteil ist die einfache Adressierung der Daten auf dem Stack mit Hilfe des Stackpointers (siehe Abb. 5.8). Hierzu muss lediglich zu dem Wert des Stackpointers ein Offset, der Bestandteil des Instruktionswortes ist, hinzuaddiert werden. Diese Berechnung und das Laden oder Speichern kann mit einem einzigen Befehl erfolgen.

Um ein Objekt einzeln in den Scratchpad zu verschieben, wäre die Konvertierung einer lokalen Variablen, deren zugehörige Funktion auch nur in maximal einer Instanz während der kompletten Programmabarbeitung vorliegt, in eine globale Variable im Onchip-Speicher notwendig. Wenn diese Variable nur in einer Instanz vorkommt, ist die Verwaltung als einzelne globale Variable möglich. Die Konvertierung ist jedoch aufgrund der dann notwendigen komplexeren Adressierung (siehe Abb. 5.7) insgesamt nachteilig. Es verbleibt noch die Möglichkeit, den Stack insgesamt oder aber ab einer gewissen Tiefe in den Onchip-Speicher zu verschieben.

Vor der Entscheidung, ob es lohnend ist, den Stack vollständig oder in Teilen zu verschieben, muss die Anzahl der Zugriffe auf den Inhalt des Stacks bestimmt werden. Diese Analyse kann ebenso wie die Bestimmung der Aufrufhäufigkeiten von Funktionen oder Basisblöcken statisch nur bedingt gelöst werden, u. a. wegen eines möglicherweise auftretenden Pointer Aliasing [Muc97].

Die alternative dynamische Analyse durch einen Simulationstrace ist datenabhängig, liefert aber nach einer Mittelwertbildung der Ergebnisse von mehreren Eingabedaten genügend gute Resultate. Es hängt dann von der Prozessorarchitektur und dem Instruktionssatz ab, ob es sich lohnt, ab einer

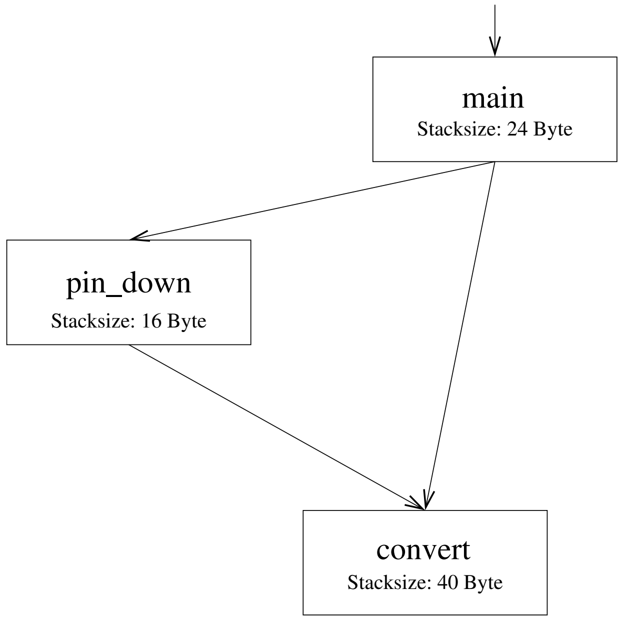


Abbildung 5.9: Stackgrößen-Berechnung

gewissen Stufe in der Funktionshierarchie den Stackpointer auf den Scratchpad zu versetzen und somit den restlichen Teil des Stacks dort zu verwalten. Beim ARM-Prozessor sind dafür sehr viele Instruktionen notwendig, sodass dies kaum Verbesserungen liefern dürfte. Einfacher und in der Praxis auch vorteilhaft ist es, den Stack vollständig in den Scratchpad zu versetzen. Für das Beispielprogramm P in Tabelle 5.2 ist in Abbildung 5.9 die Funktionshierarchie nochmals dargestellt. Der maximale Stackbedarf $MaxStackSize$ ergibt sich aus dem Maximum des Stackgrößenbedarfs über alle möglichen einfachen Wege w. Dies ist für einen Funktionsaufrufgraphen $FCG = (V, E, v_o)$ mit einer Menge von Funktionen V, einer Menge von Aufrufkanten E und einem expliziten Startknoten v_o wie folgt zu berechnen:

Definition einfacher Weg: *Ein einfacher Weg* $w = v_0, ..., v_s$ *ist eine Folge von Knoten mit* $(v_i, v_{i+1}) \in E$ *für* $0 \leq i \leq s-1$ *mit* $v_i \neq v_j$ *für* $i \neq j$ *[Sim92].*

Der Stackgrößenbedarf eines Weges $w \in W$ mit

$$W = \{w \,|w \; ist \, einfacher \;\; Weg \;\; innerhalb \;\; von \;\; FCG\}$$

beträgt:

$$StackSize(w) = \sum_{v \in w} StackSize(v)$$

Der maximale Stackgrößenbedarf des Programms P beträgt dann:

$$MaxStackSize(P) = max(x) \text{ mit } x \in \{StackSize(w) \,|\, w \in W\}$$

Im dargestellten Fall wäre das Maximum die Summe des Stackgrößenbedarfs der einzelnen Funktionen *main*, *pin_down* und *convert* mit $MaxStackSize = 24 + 16 + 40 = 80$ Bytes. Für Funktionsaufrufgraphen, die Wege enthalten, die nicht *einfach* sind, also Zyklen enthalten, kann der maximale Stackgrößenbedarf nur mit aufwendigeren Analysen durch die Bestimmung einer oberen Schranke abgeschätzt werden.

Beim Umsetzen des kompletten Stacks wird der Stackpointer von Anfang an auf einen anderen Startwert gesetzt. Wichtig ist es in diesem Fall nur, die maximale Stackgröße zu berechnen.

Fazit

In diesem Unterkapitel wurde eine Analyse des Programms präsentiert, die potenzielle Objekte zum Verlagern in einen Scratchpad untersucht. Bei diesen Objekten handelt es sich um Funktionen, Basisblöcke, globale Daten und den Stack. Mit Hilfe der gewonnen Analysedaten kann nun mit einem der im Folgenden vorgestellten Verfahren eine Auswahl von Objekten getroffen werden, um entweder statisch oder dynamisch die maximale Energiereduzierung durch Ausnutzung des Scratchpads zu erreichen.

5.4 Statische Verschiebung

5.4.1 Modifizierung der Objekte und Berechnung des Energiegewinns

Nach der Analyse eines Programms betrachten wir nun die verschiedenen Objekte Funktionen, Basisblöcke, globalen Datenobjekte und den Stack einzeln, um die Größe dieser Objekte nach der Verschiebung in den Scratchpad-Speicher zu bestimmen und die Differenz des Energieverbrauchs (= den Energiegewinn), die durch die Verschiebung des Objektes entsteht, zu berechnen [Zob01].

Funktionen

Beginnen wir zuerst mit einigen im weiteren Verlauf notwendigen Definitionen für Funktionen. Dazu betrachten wir eine Menge von Funktionen F, die zusammen das Programm P bilden.

Jede Funktion $f \in F$ hat eine Größe $S(f)$, die sich aus ihren n Instruktionen und der Größe $S(k)$ für eine einzelne Instruktion k ergibt:

$$S(f) = \sum_{\forall k \in f} S(k)$$

Das Holen einer Instruktion aus dem Speicher bewirkt einen Energieverbrauch sowohl im Speicher als auch im Prozessor. Letzteres entsteht durch das Aktivieren des Businterfaces und möglicher Wartezyklen, die beim Zugriff auf den Speicher entstehen. Die Energieeinsparung beim Instructionfetch E_{if} berechnet sich daher aus der Differenz der Zugriffe auf die beiden Speichertypen Hauptspeicher und Scratchpad-Speicher und basiert auf den unterschiedlichen Energieverbräuchen, die dabei im Speicher und Prozessor entstehen:

$$E_{if} = E_{offchip,if} - E_{onchip,if}$$

Jede Funktion $f \in F$ kann einzeln in den Scratchpad-Speicher verschoben werden, ohne dass die Instruktionen angepasst werden müssen. Daher bleibt die Anzahl der Instruktionen n und die Größe $S(f)$ unverändert. Eine Funktion $f \in F$ mit n Instruktionen, die jeweils m_k-mal ausgeführt werden, hat dann eine Energieeinsparung $E(f)$:

$$E(f) = \sum_{\forall k \in f} m_k * E_{if}$$

Sowohl die Einsprünge in die Funktion als auch die Rücksprünge bleiben bzgl. der Anzahl der Befehle unverändert. Lediglich die Adressen müssen angepasst werden. Über das Verschieben einer Funktion kann daher bei entsprechend vorhandenem Platz unabhängig von anderen Objekten entschieden werden.

Basisblöcke

Die weitere Zerlegung der Funktionen in Basisblöcke kann nun als Alternative zu der Verlagerung vollständiger Funktionen betrachtet werden. Die Funktionen $f \in F$ eines Programms P werden dafür in eine

Menge von Basisblöcken BB zerlegt. Die Größe $S(bb)$ eines Basisblockes bb berechnet sich analog zu der Größe der Funktionen:

$$S(bb) = \sum_{\forall k \in bb} S(k)$$

Ebenso kann analog der Energiegewinn $E(bb)$ durch Einsparungen beim Instructionfetch für den Basisblock bb, der insgesamt $n-$mal ausgeführt wird und aus m Instruktionen besteht, berechnet werden:

$$E(bb) = m * n * E_{if}$$

Die Ausführungshäufigkeiten der einzelnen Instruktionen des Basisblockes sind gleich und können daher

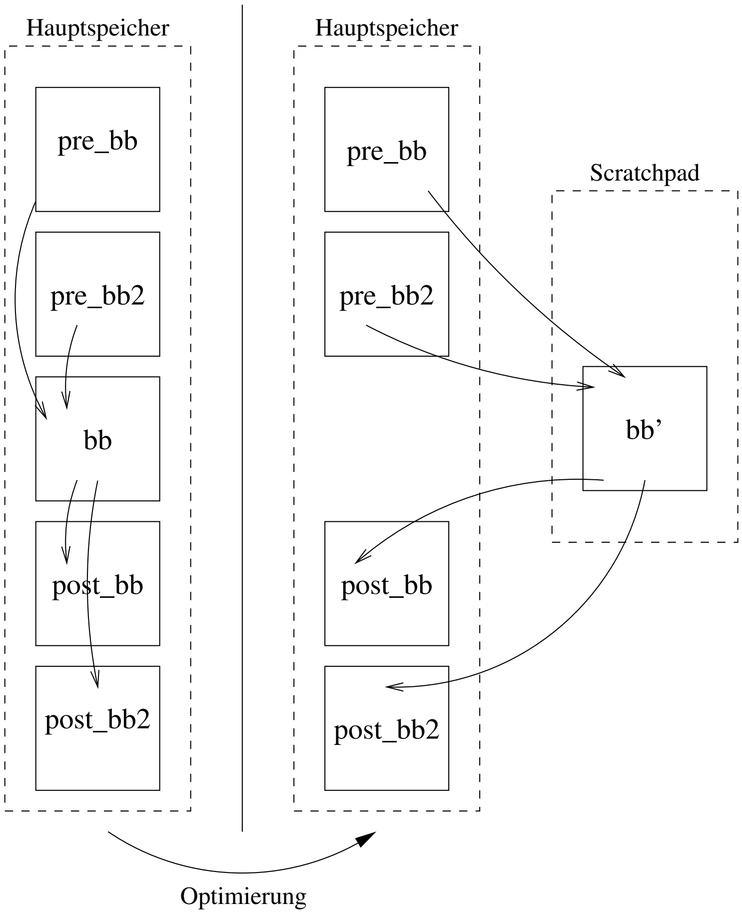


Abbildung 5.10: Verschiebung des Basisblocks bb in den Scratchpad

gemeinsam behandelt werden. Wichtig ist es nun, bei dem Verschieben eines Basisblockes bb die Sprünge in dem Beispiel in Abbildung 5.10 von den Basisblöcken pre_bb und pre_bb2 zu dem verschobenen Basisblock bb', der aus dem Basisblock bb im Hauptspeicher entstanden ist, sowie vom Basisblock bb' zu seinen Nachfolgern $post\_bb$ und $post\_bb2$ anzupassen. Dafür ist es notwendig, die im Instruktionssatz vorhandenen Sprungbefehle zu berücksichtigen. Einerseits müssen bereits vorhandene Sprungbefehle ggf. durch andere ersetzt und andererseits zusätzliche Sprungbefehle eingefügt werden. Die Details zur Anpassung der verschiedenen Sprünge werden im Folgenden vorgestellt.

Im verwendeten Thumb-Instruktionssatz sind wie in allen komprimierten Instruktionssätzen nur sehr begrenzte Möglichkeiten geboten. Folgende Sprungbefehle existieren:

1. ein unbedingter kurzer, relativer Sprung *shortJMP*
   Ein 16-Bit-Befehl, “BRA”, der einen relativen, kurzen Sprung ausführen kann und unbedingt ausgeführt wird. Die Reichweite beträgt +/- 2 KBytes.

2. mehrere bedingte kurze, relative Sprünge *condshortJMP*
   Es handelt sich hier ebenfalls um 16-Bit-Befehle, die einen relativen, kurzen Sprung ausführen, dies aber nur, wenn die betreffenden Flags der Conditioncodes entsprechend gesetzt sind. Die Reichweite beträgt +/- 256 Bytes.

3. ein unbedingter langer, relativer Sprung *longJMP*
   Bei einem 16-Bit-Befehlssatz kann nur eine geringe Reichweite bei einem Sprungbefehl implementiert werden. Daher gibt es im Thumb-Befehlssatz eine Ausnahme mit dem Befehl "Branch Link (BL)", der aus zwei 16-Bit-Worten zusammengesetzt ist. Dadurch stehen insgesamt 24 Bit für die relative Adressierung des Sprungziels zur Verfügung. Dieser Sprungbefehl ist insbesondere für Funktionsaufrufe ausgelegt, da er neben dem Ausführen des Sprungs die Ausgangsadresse noch zusätzlich im Link-Register für spätere Rücksprünge speichert.
   Da der Adressabstand zwischen Hauptspeicher und Scratchpad-Speicher größer als der maximale Adressbereich der kurzen Sprünge ist, ist dieser Befehl auch der einzige Sprungbefehl, der für einen Sprung zwischen den beiden Speichern eingesetzt werden kann.

Sowohl absolute Sprungbefehle als auch bedingte lange Sprungbefehle existieren nicht, um den Befehlssatz möglichst klein zu halten. Für andere Prozessoren mit anderen Kombinationen von Sprungbefehlen lässt sich diese Technik aber leicht anpassen.

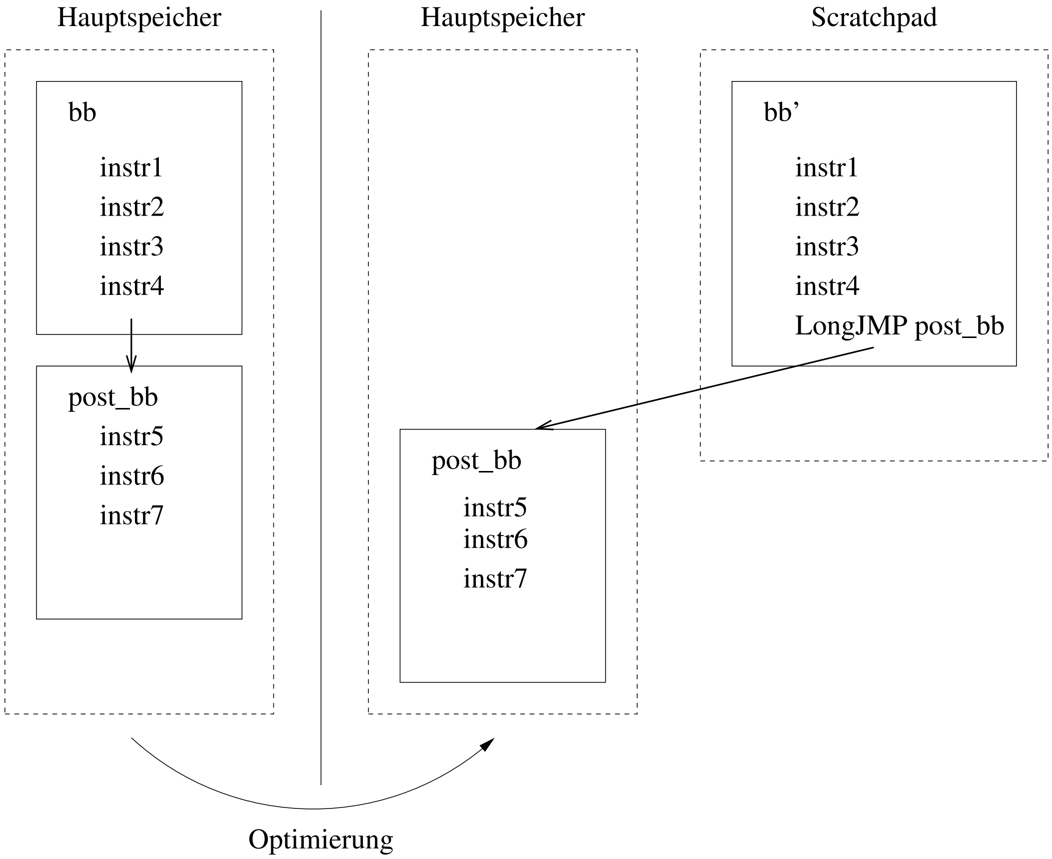


Abbildung 5.11: Basisblock *bb* ohne Sprungbefehl als letzte Instruktion

Im Folgenden werden nun die Fälle unterschieden, mit denen der in den Scratchpad-Speicher verschobene Basisblock *bb*’ verlassen wird. Neben der Möglichkeit, dies durch Sprungbefehle zu tun, kommt noch die Möglichkeit des impliziten Kontrollflusses hinzu (siehe Abb. 5.11, Übergang von *bb* zu $post\_bb$), d. h. der letzte Befehl des Basisblocks *bb* ist kein Sprungbefehl und es wird daher automatisch im Anschluss der

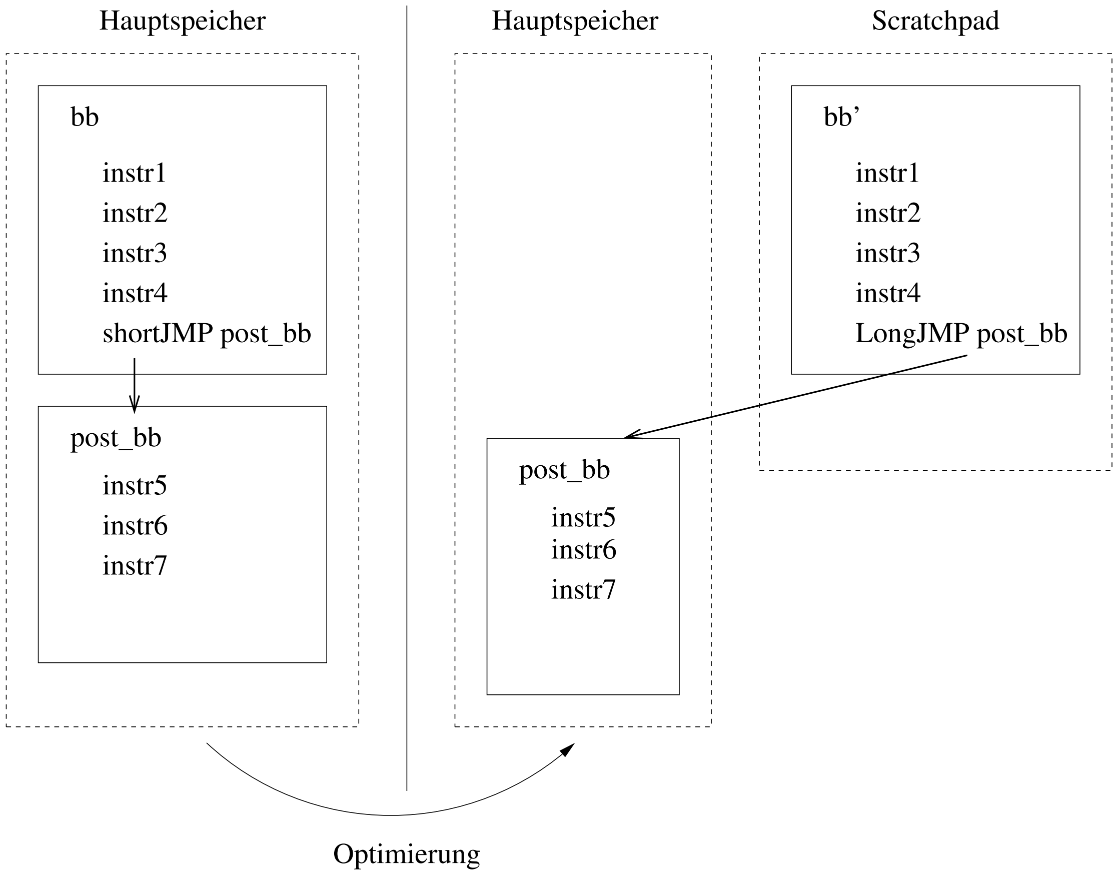


Abbildung 5.12: Basisblock *bb* mit kurzem Sprung als letzter Instruktion

erste Befehl des im Speicher nachfolgenden Basisblockes $post\_bb$ ausgeführt.

Folgende Anpassungen müssen nun für den ausgehenden Kontrollfluss des verschobenen Basisblockes bb' durchgeführt und für die unterschiedlichen Fälle der zusätzliche Platzbedarf *sizeOffset* und der zusätzliche Energieverbrauch für den Einsprung $EnergyOffset_{in}$ und den Rücksprung $EnergyOffset_{out}$ bestimmt werden. Dabei basieren die Berechnungen der Größe und der Energie des in den Scratchpad verschobenen Blockes bb' auf denen des ursprünglichen, nicht verschobenen Basisblockes *bb*:

$$S(bb') = S(bb) + sizeOffset(bb')$$

$$E(bb') = E(bb) + EnergyOffset_{in}(bb') + EnergyOffset_{out}(bb')$$

Folgende Szenarien müssen unterschieden werden:

1. Ende des Basisblocks *bb* ohne Sprungbefehl
   Für den Rücksprung muss ein langer Sprungbefehl ergänzt werden (siehe Abb. 5.11). Bei der Größenberechnung muss dieser Sprungbefehl berücksichtigt werden:

$$sizeOffset(bb') = S(longJMP)$$

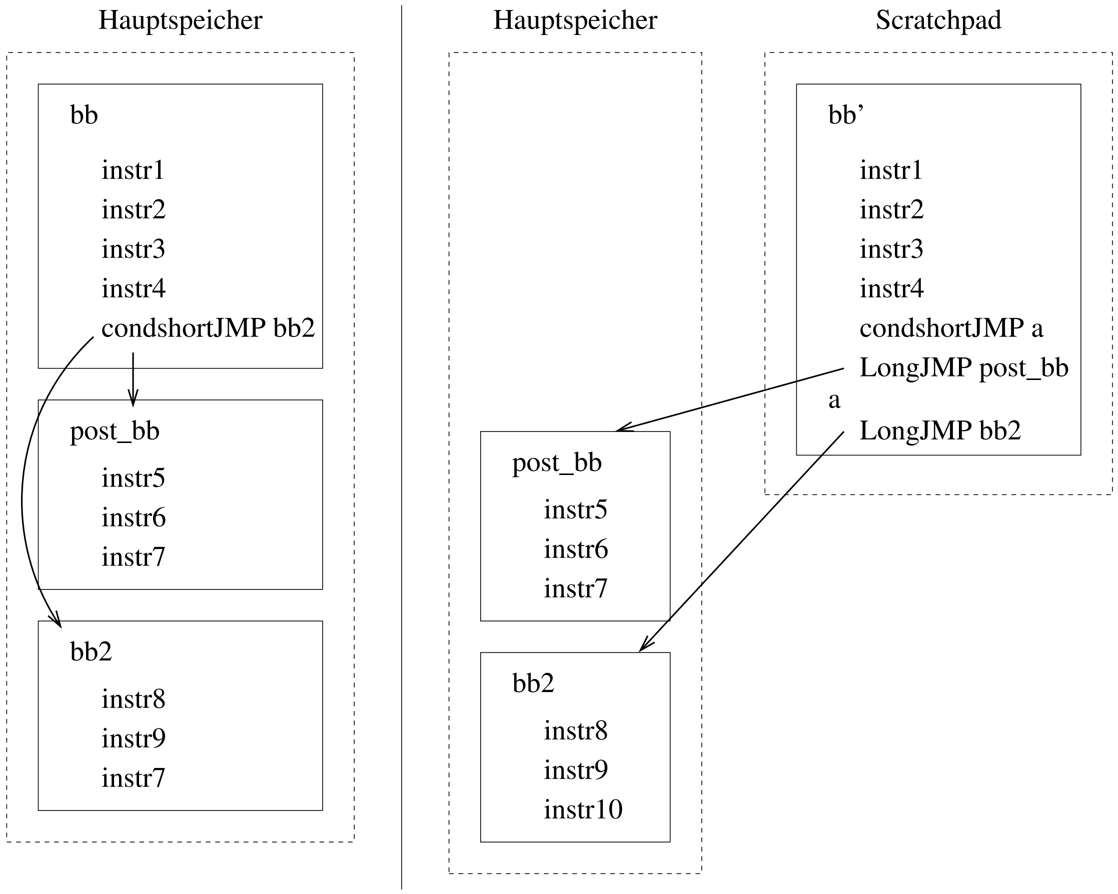


Abbildung 5.13: Basisblock *bb* mit bedingtem Sprung als letzter Instruktion

Der Energiegewinn reduziert sich wegen der zusätzlichen n-maligen Ausführung des langen Sprungbefehls:

$$\mathit{EnergyOffset}_{out}(bb') = -n * E(\mathit{longJMP})$$

2. Verlassen des Basisblocks bb durch einen unbedingten, kurzen Sprungbefehl *shortJMP*
   Der kurze Sprungbefehl muss durch einen langen Sprungbefehl ersetzt werden (siehe Abb. 5.12), da der Abstand zwischen Hauptspeicher und Scratchpad im Adressraum die Reichweite eines kurzen Sprungbefehls überschreitet. Die Größe verändert sich entsprechend:

   $$\mathit{sizeOffset}(bb') = S(\mathit{longJMP}) - S(\mathit{shortJMP})$$

   Der Energiegewinn muss ebenso den langen statt den kurzen Sprung berücksichtigen, der n-mal ausgeführt wird:

   $$\mathit{EnergyOffset}_{out}(bb') = -n * [E(\mathit{longJMP}) - E(\mathit{shortJMP})]$$

3. Verlassen des Basisblocks bb durch einen bedingten, kurzen Sprungbefehl *condshortJMP*
   Der bedingte Sprung muss durch einen zusätzlichen langen Sprung *longJMP* verlängert werden (siehe Abb. 5.13). Wenn die Bedingung nicht erfüllt ist, wird nicht gesprungen, sondern mit dem im Speicher folgenden Basisblock fortgesetzt. Für diesen Fall muss ein weiterer unbedingter langer Sprungbefehl *longJMP* ergänzt werden. Insgesamt ergibt sich dadurch der folgende Größenoffset:

   $$\mathit{sizeOffset}(bb') = 2 * S(\mathit{longJMP})$$

   Beim Energiegewinn ist zu berücksichtigen, dass bei jedem der n Durchläufe auch immer nur einer der beiden langen Sprünge *longJMP* ausgeführt wird, wodurch der Faktor n und nicht $2*n$ entsteht:

   $$\mathit{EnergyOffset}_{out}(bb') = -n * E(\mathit{longJMP})$$

4. Verlassen des Basisblocks bb durch einen oder zwei lange Sprungbefehle *longJMP*
   In diesem Fall ist keine Veränderung notwendig, da die langen Sprungbefehle weiterhin ausreichend sind. Daraus ergibt sich:

   $$\mathit{sizeOffset}(bb') = 0$$

   und

   $$\mathit{EnergyOffset}_{out}(bb') = 0$$

5. Verlassen des Basisblocks bb mit einem bedingten kurzen Sprungbefehl *condshortJMP* und einem langen Sprungbefehl *longJMP*
   Während der lange Sprungbefehl *longJMP* unverändert bleiben kann, muss der kurze, bedingte Sprung *condshortJMP* durch einen langen Sprungbefehl *longJMP* verlängert werden (s. Abb. 5.14). Dadurch ergibt sich für den Größenoffset:

   $$\mathit{sizeOffset}(bb') = S(\mathit{longJMP})$$

   Beim Energiegewinn muss aber nun unterschieden werden, welcher der beiden Sprünge durchlaufen wird, da ein zusätzlicher Energieaufwand nur entsteht, wenn die Bedingung zutrifft. Der Energiegewinn muss durch folgenden Offset korrigiert werden, wenn die Bedingung k-mal zutrifft:

   $$\mathit{EnergyOffset}(bb') = -k * E(\mathit{longJMP})$$

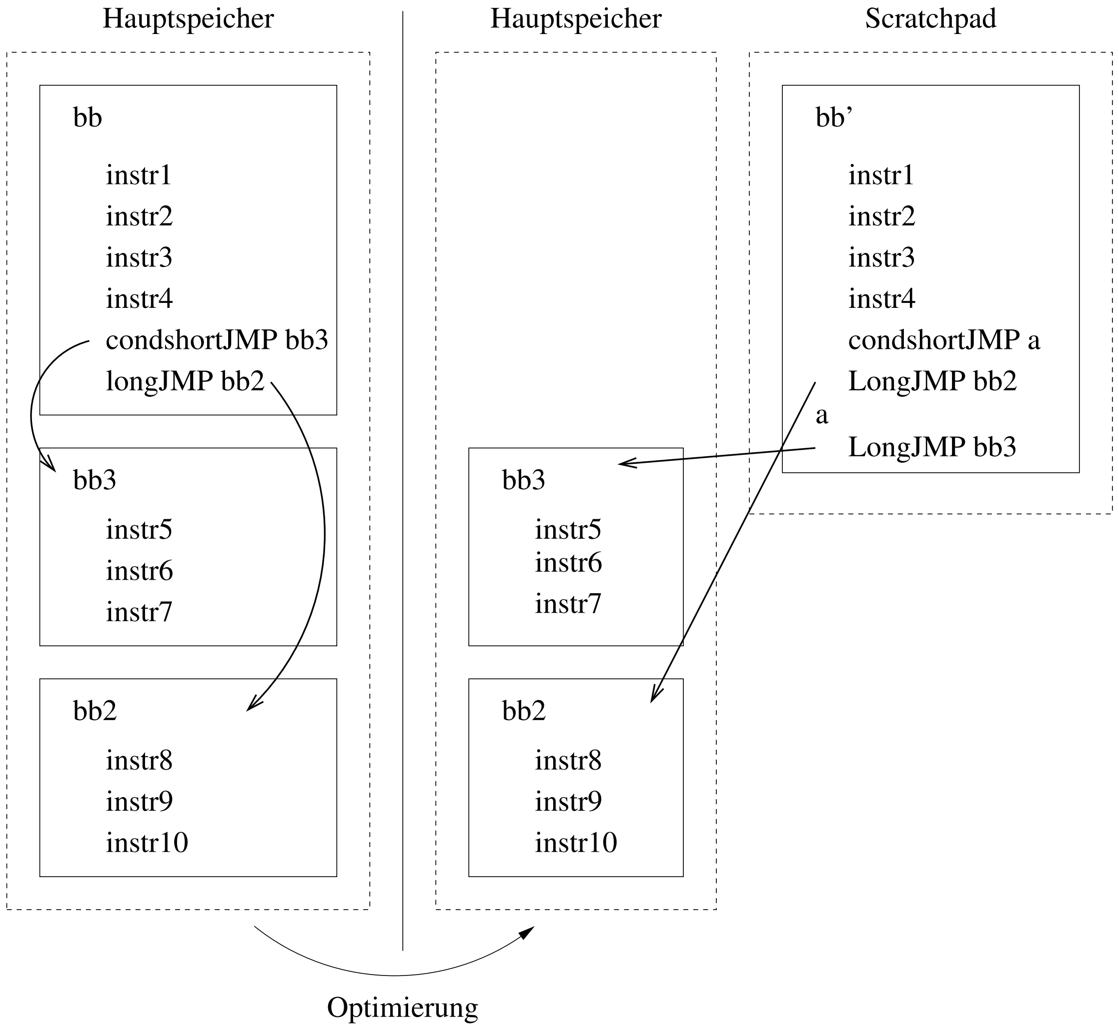


Abbildung 5.14: Ende mit kurzem bedingtem und langem Sprungbefehl

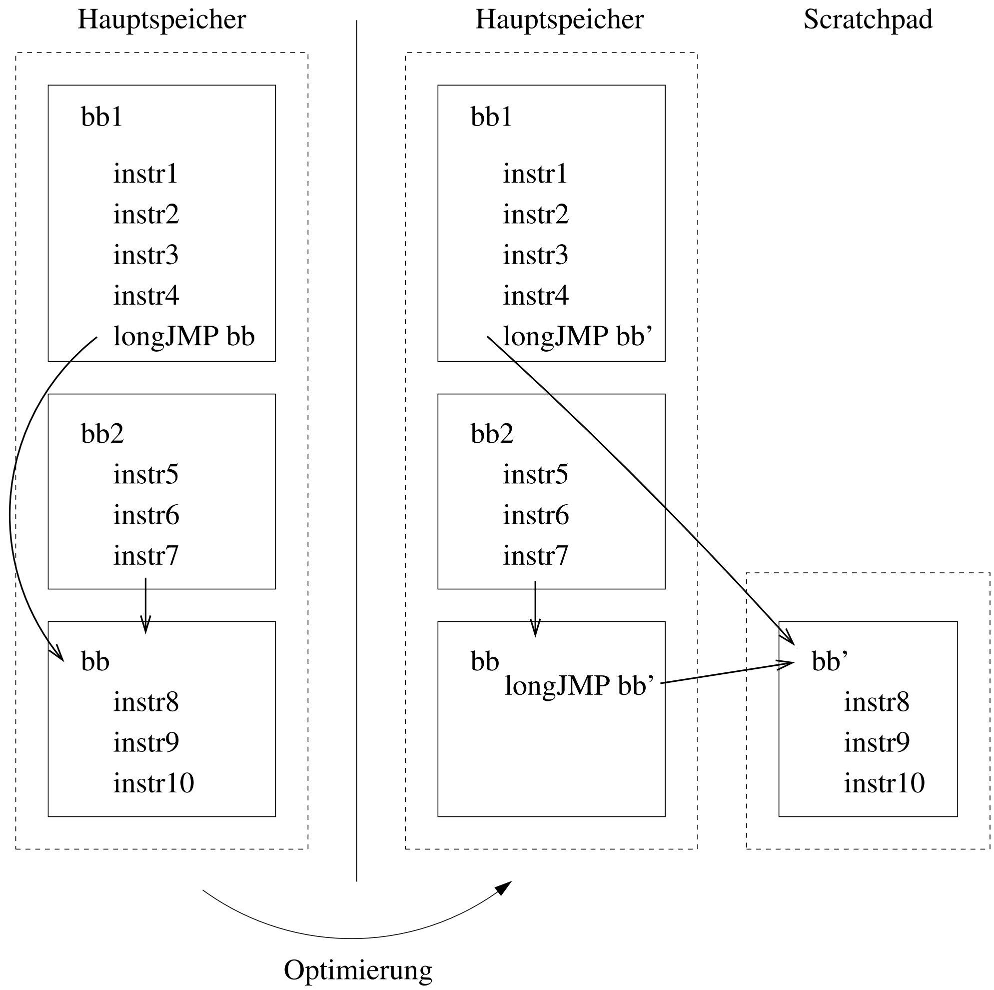


Abbildung 5.15: Einsprung in den verschobenen Basisblock bb'

Zwei mögliche Sonderfälle sind noch zu erwähnen:

1. Häufig erfolgt ein bedingter Sprung vom Ende eines Basisblockes wieder zu seinem Anfang. Bei Schleifenkonstrukten dieser Art muss der Sprung nicht modifiziert werden, da das Sprungziel innerhalb des Basisblockes liegt und die relative Sprungweite unverändert bleiben kann.

2. Wenn ein Basisblock am Ende einer Funktion einen Rücksprung ausführt, z. B. beim ARM-Prozessor durch Laden des Link-Registers in den Programmzähler, muss für den Ausgang aus dem Basisblock keine Modifikation durchgeführt werden. Daraus ergeben sich:

$$\textit{sizeOffset}(bb') = 0$$

und:

$$\textit{EnergyOffset}_{out}(bb') = 0$$

Nachdem die Größe und Energie entsprechend neu für das Verlassen des Basisblocks bb' berechnet wurde, muss analog dazu auch der Einsprung kalkuliert werden (siehe Abb. 5.15). Dafür kann entweder an die alte Stelle des nun verschobenen Basisblocks bb' im Hauptspeicher ein langer Sprungbefehl *longJMP* zur neuen Position im Scratchpad eingefügt werden, wodurch aber in allen Fällen der Ausführung des Basisblockes bb' ein zusätzlicher langer Sprung beim Energiegewinn subtrahiert werden muss. Effizienter ist die Lösung, wenn alle Kanten des Kontrollflusses zum Basisblock bb' einzeln betrachtet werden und dadurch das Potenzial ausgeschöpft wird, den Sprung eventuell effizienter zu gestalten, z. B. wenn vorher auch schon ein langer unbedingter Sprung vorgesehen war (Basisblock $bb1$ in Abb. 5.15).

Die eventuelle Einfügung zusätzlicher langer Sprünge im Hauptspeicher ist für die Größenberechnung im Scratchpad-Speicher nicht relevant, jedoch wird der Energiegewinn kleiner. Daher muss ein zusätzlicher Energieoffset $\textit{EnergyOffset}_{in}$ für den Einsprung berechnet und addiert werden, der von den unterschiedlichen Häufigkeiten und Sprungarten zum Basisblock bb' abhängt. Die Berechnung kann analog zu den angegebenen Berechnungen des Ausgangs von Basisblock bb' durchgeführt werden.

Globale Daten

Zu den globalen Daten gehören Konstanten und Variablen, skalare und nicht-skalare Typen, die jeweils einzeln als Objekt betrachtet werden. Jedes globale Datum gd benötigt einen bestimmten Platzbedarf im Hauptspeicher $S(gd)$. Da nur die Adressen dieses Datums ausgetauscht werden, verändert sich auch der Platzbedarf nicht, wenn ein globales Datum in den Scratchpad-Speicher verschoben wird.

Die Energiereduzierung beim Verschieben in den Onchip-Speicher hängt von der Gesamtanzahl der Zugriffe acc auf das jeweilige globale Datum gd ab. Die Gesamtanzahl berechnet sich aus den Zugriffen der jeweiligen Basisblöcke $acc_{bb}(gd)$. Innerhalb eines Basisblockes bb kann dann die Anzahl von Instruktionen (=statische Anzahl) $stat_{bb}(gd)$, die auf das Datum gd zugreifen, ermittelt und mit der Anzahl der Ausführungen n_{bb} des Basisblocks bb multipliziert werden:

$$acc(gd) = \sum_{\forall bb \in BB} acc_{bb}(gd) = \sum_{\forall bb \in BB} stat_{bb}(gd) * n_{bb}$$

Die Energiereduzierung durch Verschiebung des globalen Datums gd in den Scratchpad-Speicher ergibt sich dann auf Basis der Energieeinsparung eines einzelnen Zugriffs E_{data} durch eine Load- oder Store-Instruktion:

$$E(gd) = acc(gd) * E_{data}$$

Für jedes globale Datum kann auf diese Art unabhängig von anderen Objekten die Größe $S(gd)$ und die potenzielle Energiereduzierung $E(gd)$ berechnet werden.

Stack

Der Stack enthält - wie schon in Kapitel 5.3 beschrieben - unterschiedliche Arten von Informationen. In diesem Verfahren soll der Stack nur komplett als Objekt betrachtet werden, welches in den Scratchpad verschoben werden kann. Das Verfahren kann jedoch auch auf die Betrachtung von Teilen des Stackes erweitert werden.

Die Bestimmung der Größe, die sich auch durch die Verschiebung in den Scratchpad nicht verändert, wurde bereits dargelegt:

$$S(Stack) = MaxStackSize(P)$$

Da die statische Analyse der Zugriffe auf den Stack über die Summe der einzelnen Objekte nur schwer bzw. ungenau zu ermitteln ist, wurde hier das Verfahren der Simulation mit anschließender Zählung der Zugriffe auf den Adressbereich des Stacks gewählt ("dynamic access count"). Da die Stackgröße berechnet werden kann und die Startadresse ebenfalls bekannt ist, kann damit auch der Adressbereich des Stacks bestimmt werden. Die Auswertung des Simulationstraces liefert dann die Anzahl der Zugriffe $acc(Stack)$ auf deren Basis sich die Energiereduzierung wie bei den globalen Daten errechnet:

$$E(Stack) = acc(Stack) * E_{data}$$

Fazit

In diesem Abschnitt wurden die verschiedenen Speicherobjekte Funktionen, Basisblöcke, globale Daten und der Stack auf die Größenänderung aufgrund der Verschiebung in den Scratchpad-Speicher und auf ihre Energiereduzierung hin untersucht. Auf Basis dieser Daten gilt es nun in geeigneter Weise eine Auswahl an Objekten getroffen werden, sodass die Energiereduzierung maximal wird.

5.4.2 Auswahl der Objekte

Nach den Vorarbeiten in den letzten Abschnitten gilt es nun, die optimale Auswahl aus den vorhandenen Objekten zu treffen. Dafür muss zuerst das Problem klassifiziert werden, um - falls möglich - bekannte Lösungsverfahren anwenden zu können. Dieses Unterkapitel stützt sich wesentlich auf die theoretischen Grundlagen in [Weg00] und [Zob01] ab.

Wenn wir die Speicher-Objekte betrachten, stellen wir fest, dass sie:

1. voneinander unabhängig sind,
2. nicht teilweise sondern stets vollständig verschoben werden,
3. eine feste Größe besitzen, wenn sie in den Scratchpad-Speicher verschoben werden.

Weiterhin ist festzustellen, dass die Größe des Scratchpad ebenfalls konstant ist. Es geht nun darum, eine Menge von Objekten unter Berücksichtigung der begrenzten Scratchpad-Größe auszuwählen, sodass die Summe der Energieeinsparungen maximal wird.

Die vorgenannten Eigenschaften beschreiben das so genannte "Rucksack-Problem" (= Knapsack-Problem). Der Name entstammt dem entsprechenden Problem eines Wanderers, der nicht mehr als b kg Gewicht in seinem Rucksack mit sich tragen möchte. Es stehen ihm insgesamt n Objekte zur Verfügung, von denen er eine Auswahl treffen muss. Das i-te Objekt hat a_i kg Gewicht und für den Wanderer einen Nutzen von c_i. Er möchte nun den Nutzen seiner Rucksackbeladung maximieren, ohne das Gewichtslimit zu verletzen. Formal beschrieben kann das Knapsack-Problem wie folgt definiert werden:

Definition Knapsack: *Das Knapsack-Problem besteht aus der Aufgabe, für* $a_1, ..., a_n, c_1, ..., c_n, b \in N$ *unter allen Vektoren* $(x_1, ..., x_n) \in \{0,1\}^n$ *mit* $a_1x_1 + ... + a_nx_n \leq b$ *einen Vektor zu berechnen, der die Funktion* $W = c_1x_1 + ... + c_nx_n$ *maximiert.*

Auf das hier vorliegende Problem angewandt, wäre $a_i \in N$ dementsprechend die Größe S_i des Objektes i, der Nutzen c_i würde dem Energieeinsparpotenzial E_i des i-ten Speicherobjektes entsprechen und das Fassungsvermögen des Rucksacks b der limitierten Größe des Scratchpad-Speichers $SP \in N$.

Das Knapsack-Problem stellt ein ganzzahliges (da $x_i \in \{0,1\}$ sogar binäres) Optimierungsproblem dar, bei welchem die Anzahl der Restriktionen 1 beträgt und somit minimal ist. Diese Restriktion entspricht der Größenbeschränkung des Scratchpad-Speichers. Sowohl die Zielfunktion als auch die Restriktion sind linear. Bei einem Knapsack-Problem handelt es sich um ein NP-hartes Problem. Dies bedeutet unter der Annahme $NP \neq P$, dass es keinen polynomiellen (deterministischen) Algorithmus gibt, der das Optimum findet. Daher muss entschieden werden, welchen Ausweg man wählt, da das Problem für den allgemeinen Fall nicht effizient lösbar ist. Folgende Arten von Algorithmen stehen zur Wahl:

- Pseudopolynomielle Algorithmen, die für Eingaben, die aus kleinen Zahlen bestehen, effizient sind,
- Algorithmen, die im *worst case* exponentielle Rechenzeit haben, aber für viele Eingaben schnell stoppen,
- Polynomielle Algorithmen, die stets effizient sind, aber nicht in allen Fällen das beste Ergebnis liefern, wozu auch heuristische Algorithmen zählen.

Generell wird optimierenden Compilern für die Ausführung der im Vergleich zu einfacheren Compilern zusätzlich ausgeführten Optimierungen mehr Rechenzeit zugestanden. Allerdings werden Anwender eine Laufzeit von mehreren Tagen oder sogar noch längere Zeiten sicherlich nicht mehr akzeptieren. Andererseits ist generell ein durch Compiler generierter Maschinencode nicht optimal, sodass eventuell auf die Optimalität der einzelnen Lösungen verzichtet werden kann. Daher wird das folgende Vorgehen vorgeschlagen:

Algorithm 1 simples Verfahren für das Knapsack-Problem

```
E_sum := 0;
SP_akt := SP;
heapsort(obj[]);                        /* Sortierung der Objekte nach Effizienz */
for i := 1 to n do
if S(obj[i]) <= SP_akt then begin       /* Platz für Objekt i ausreichend ? */
  x_i := 1;                             /* Platz reicht aus für Objekt i */
  SP_akt := SP_akt - S(obj[i]);
  E_sum := E_sum + E(obj[i]); end
else begin
  x_i := 0;                             /* Platz reicht nicht aus für Objekt i */
end
```

Es ist eine Laufzeit bis zu einer oberen zeitlichen Schranke t_{limit} akzeptabel. Wenn diese Zeit für den Compiliervorgang nicht ausreicht, wird auch die bis dahin gefundene beste Lösung, die nahe am Optimum ist, vom Anwender akzeptiert.

Zuerst soll die notwendige Eigenschaft der Effizienz der betrachteten Objekte eingeführt werden:

Definition Effizienz: *Die Effizienz eff eines Objektes* obj *ist der Quotient aus dem Energiegewinn bei der Verschiebung des Objektes in einen günstigeren Speicher* $E(obj)$ *dividiert durch seine Größe* $S(obj)$*:*

$$eff(obj) = \frac{E(obj)}{S(obj)}$$

Ausgangslage

1	2	3	4	5
size: 24 Bytes	size: 20 Bytes	size: 12 Bytes	size: 12 Bytes	size: 10 Bytes
Gewinn: 13 nJ	Gewinn: 10 nJ	Gewinn: 5 nJ	Gewinn: 12 nJ	Gewinn: 8 nJ
eff=13/24=0,54	eff=10/20=0,5	eff=5/12=0,42	eff=12/12=1	eff=8/10=0,8

Nach Effizienz sortierte Objekte

4	5	1	2	3
size: 12 Bytes	size: 10 Bytes	size: 24 Bytes	size: 20 Bytes	size: 12 Bytes
Gewinn: 12 nJ	Gewinn: 8 nJ	Gewinn: 13 nJ	Gewinn: 10 nJ	Gewinn: 5 nJ
eff=1	**eff=0,8**	**eff=0,54**	**eff=0,5**	**eff=0,42**

Abbildung 5.16: Sortieren von Objekten für das Scratchpad

Ein sehr einfaches Verfahren (siehe Algorithmus 1) erhält man, wenn die einzelnen Objekte obj nach ihrer Effizienz $\mathit{eff}(obj)$ sortiert werden (siehe Abb. 5.16). Beginnend mit dem Objekt mit der höchsten Effizienz wird anschließend geprüft, ob das Objekt noch in den Scratchpad hineinpasst. Wenn ja, wird es hereingenommen, der Gesamtenergiegewinn E_{sum} der Lösung um den Energiegewinn dieses Objektes $E(obj)$ erhöht und die noch verbleibende und verfügbare Größe des Scratchpads SP_{akt} um die Größe des Objektes $S(obj)$ verringert. Danach ist das nächste Objekt in der sortierten Folge in der gleichen Art zu begutachten. Die Komplexität des Verfahrens für n Objekte besteht aus der Sortierung, deren Komplexität vom gewählten Sortierverfahren abhängt (z. B. für heapsort mit $O(n * \log(n)))$, und einem Anteil $O(n)$ für die Auswahl der Objekte. Dadurch ergibt sich insgesamt die folgende Komplexität:

$$O(n * log(n) + n)$$

Dieses sehr simple Verfahren findet in der Praxis sehr schnell eine gute Lösung. Die Abweichung der gefundenen Auswahl der Objekte von der optimalen Lösung liegt darin begründet, dass am Ende ein Objekt nicht mehr ganz in den Scratchpad-Speicher passt. Es kann sein, dass man auf ein schon hereingenommenes Objekt besser verzichtet und dafür eine Kombination von Objekten mit einer geringeren Effizienz gewählt hätte, weil sie den restlichen Platz besser ausnutzen würde. Auf das Beispiel aus Abbildung 5.16 kann man das simple Verfahren anwenden und gelangt zur in Abbildung 5.17 dargestellten Lösung für einen Scratchpad-Speicher, die insgesamt 46 Bytes von 56 Bytes ausnutzt und einen Energiegewinn von 33 nJ erzielt. In diesem Beispiel existiert auch der beschriebene Fall, dass es bessere Lösungen gibt:

- Die "optimale Lösung 1" nimmt das "Objekt 1" nicht und stattdessen "Objekt 2" und "Objekt 3". Die Speicherausnutzung erhöht sich auf 54 von insgesamt 56 Bytes und der Energiegewinn steigt um weitere 2 nJ auf insgesamt 35 nJ.

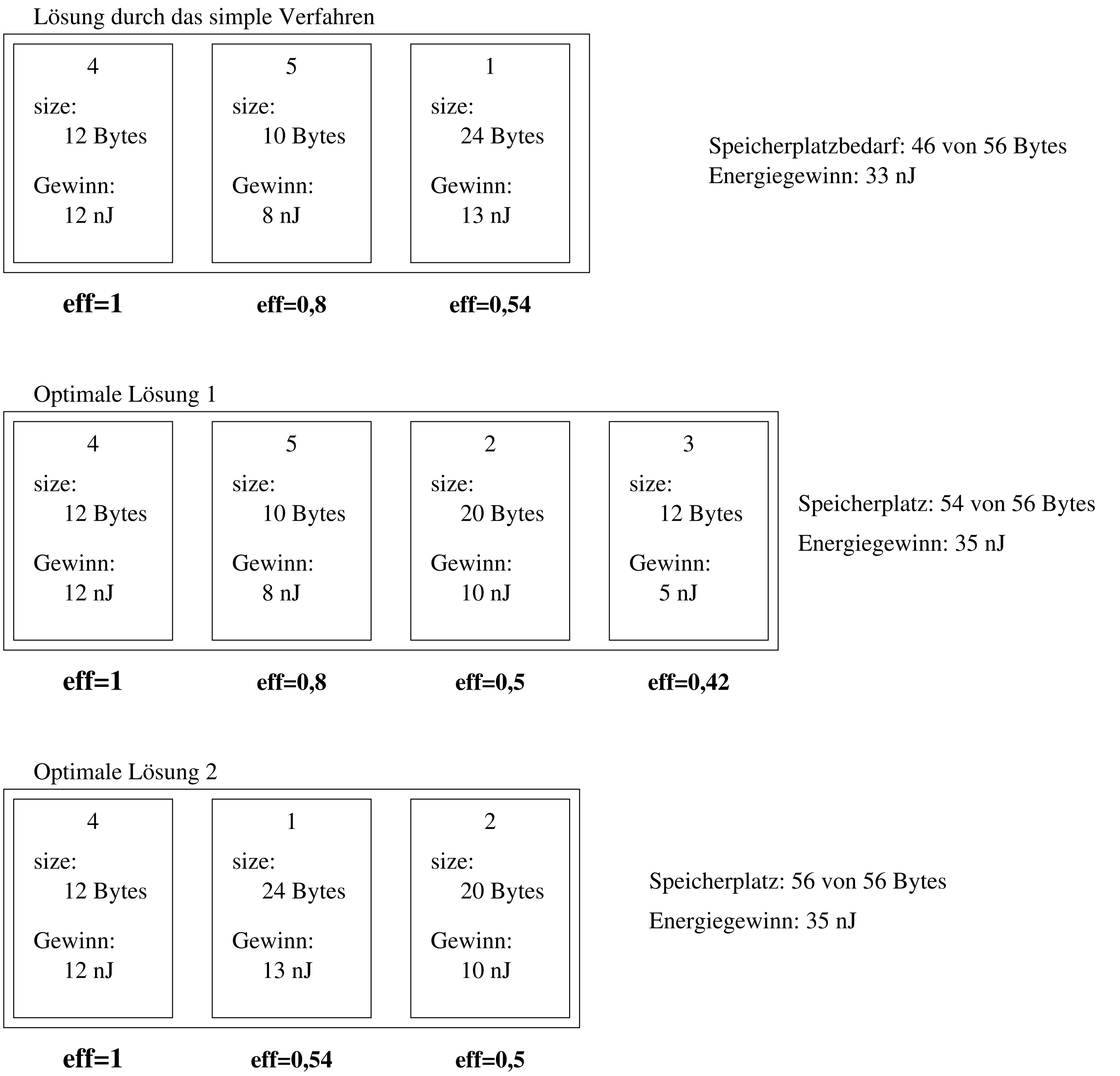


Abbildung 5.17: simple und optimale Lösungen

- Die "optimale Lösung 2" nimmt im Vergleich zur Lösung durch das simple Verfahren das "Objekt 5" nicht und stattdessen das "Objekt 2" in die Auswahl. Die Speicherausnutzung erhöht sich auf 56 von insgesamt 56 Bytes und der Energiegewinn steigt ebenfalls um weitere 2 nJ auf insgesamt 35 nJ.

Beide Lösungen sind als optimal zu bezeichnen, da sie den gleichen maximal möglichen Energiegewinn erzielen. Der Unterschied im belegten Speicherplatz im Scratchpad ist nicht relevant, da der Constraint der Speichergröße eingehalten wird. Es gilt nun, Verfahren zu finden, die über das simple Verfahren hinaus bessere und möglichst optimale Lösungen für das beschriebene Knapsack-Problem finden.

Dynamische Programmierung

Für die Suche der optimalen Lösung, falls dies in der zur Verfügung stehenden Zeit t_{limit} möglich ist, betrachten wir im nächsten Schritt einen rekursiven Algorithmus (siehe Algorithmus 2). Voraussetzung für die Anwendbarkeit dieses Algorithmus ist die mögliche Zerlegung des Gesamtproblems in Teilprobleme, wie dies auf das Knapsack-Problem zutrifft. Die optimale Lösung muss sich ferner aus der optimalen Verbindung von Lösungen kleinerer Teilprobleme ergeben (Bellmansches Optimalitätsprinzip). Als letzte Bedingung muss gelten, dass die Verbindung suboptimaler Lösungen nicht zu einer optimalen Lösung führen darf.

Algorithm 2 rekursiver Algorithmus für das Knapsack-Problem

```
knapsack(i,H) begin                      /* Aufruf für Objekt i mit Speichergröße H */
  if i = 0 then return 0;
  if H < S_i then begin                  /* Platz für Objekt i ausreichend ? */
    x_i:=0;                              /* wenn nein, Objekt i nicht wählen */
    return knapsack(i-1,H); end          /* rekursiver Aufruf für Objekt i-1 */
  else begin                             /* Platz reicht aus für Objekt i */
    A := knapsack(i-1,H);                /* berechnen für i-1-tes Objekt ohne Objekt i */
    B := knapsack(i-1,H-S_i)+E_i;        /* berechnen für i-1-tes Objekt mit Objekt i */
    if A >= B then begin                 /* beste Lösung auswählen */
      x_i:=0;                            /* i-tes Element ausschließen */
      return knapsack(i-1,H); end        /* Fortsetzung für Objekt i-1 */
    else begin
      x_i:=1;                            /* i-tes Objekt auswählen */
      return (E_i +                      /* Fortsetzung für Objekt i-1  */
           knapsack(i-1,H-S_i)) end;
  end;
end;                                     /* anschließend Rückgabe des Wertes von Objektes
                                            i-1 + Rückgabewert der Rekursion */
```

Der Algorithmus wird durch den initialen Aufruf der Funktion $knapsack(n, SP)$ für ein Knapsack-Problem mit der Anzahl n der Objekte und der Größe SP des Knapsacks (= Größe des Scratchpads) gestartet. Es wird jeweils rekursiv die Berechnung für die Lösung der Teilprobleme ohne Einbeziehung des letzten Objektes i aufgerufen. In Abhängigkeit davon, ob das Objekt i in den Knapsack genommen wird oder nicht, wird $knapsack(i-1, H-S_i)$ bzw. $knapsack(i-1, H)$ berechnet. Als Rückgabewert wird der Energiegewinn geliefert.

Mit diesem Algorithmus wird nun im Gegensatz zu dem vorher präsentierten simplen Verfahren die optimale Lösung des Problems gefunden. Der Nachteil dieses Ansatzes ist allerdings, dass einzelne Teilprobleme mehrfach berechnet werden und sich insgesamt eine Komplexität von $O(3^n)$ ergibt [Wid02].

Algorithm 3 dynamisches Programmieren beim Knapsack-Problem

```
for H := 0 to SP do                          /* Phase 1 : Berechnung der Teilergebnisse */
  if i = 0 then                              /* in Tabelle r[] */
    r[i,H] := 0;
  else if H < S_i then                       /* Objekt passt nicht in Scratchpad */
    r[i,H] := r[i-1,H];
  else                                       /* Wahl des besseren Ergebnisses der beiden */
    r[i,H] := max(r[i-1,H],                  /* Teilprobleme */
                  r[i-1,H-S_i]+E_i);
H := b;                                      /* Phase 2 : Start der Berechnung der Lösung */
for i := n downto 1 do                       /* auf Basis der Tabelle */
  if r[i,H] = r[i-1,H] then
    x_i := 0;                                /* Objekt gehört nicht zur Lösung */
  else begin
    x_i := 1;                                /* Objekt wird ausgewählt */
    H := H - S_i;
  end;
```

Ein weiteres Verfahren zur Lösung des Problems, dass eine geringere Komplexität besitzt, kann mithilfe des *dynamischen Programmierens* entwickelt werden. Der Begriff dynamisch steht in diesem Zusammenhang für sequentiell und der Begriff Programmierung bedeutet hier die Optimierung mit Nebenbedingungen. Diese Bezeichnung geht auf Richard Bellman (1957) zurück.

Die Idee des dynamischen Programmierens erweitert den Ansatz des Lösens von Teilproblemen des Algorithmus 2 durch die Speicherung der Zwischenergebnisse. In Algorithmus 3 [Wid02] ist dargestellt, wie eine Implementierung hierfür aussehen kann. In der ersten Phase wird bottom-up die Tabelle gefüllt, in der die Ergebnisse der Teilprobleme zwischengespeichert werden, und in der zweiten Phase die eigentliche Auswahl der Objekte durchgeführt.

Wenn die Methode des dynamischen Programmierens anwendbar ist, ist bekannt, dass sie zu einem effizienten Algorithmus führt, wenn:

- die Anzahl betrachteter Teilprobleme klein,
- die Anzahl betrachteter Kombinationen von Teilproblemen klein und
- die Berechnung des Wertes einer Kombination von Teilproblemen aus den Werten der Teilprobleme einfach ist.

Die Komplexität des Algorithmus beträgt $O(n * SP)$, welches einem linearen Wachstum der Funktion entspricht. Allerdings hängt die Komplexität stark von der Größe des Scratchpads SP ab, wodurch ein exponentieller Faktor entstehen kann. Von einem effizienten Algorithmus spricht man daher, wenn $n^2 < SP$ ist. Wenn man nun von einer Anzahl der Objekte n bei realistischen Applikationen zwischen 100 und 1.000 ($\Rightarrow$ 10.000 $< n^2 <$ 1.000.000) und einer Größe des Scratchpad SP von z. B. 8 KBytes ausgeht, ist die Effizienz daher nicht mehr gegeben. Ein weiterer Nachteil ist, dass bei Abbruch des Algorithmus wegen Erreichens der zeitlichen Schranke t_{limit} keine Lösung vorliegt, die in einem solchen Fall nahe dem Optimum liegt und weiterverwendet werden kann.

Branch-and-Bound

Als alternativen Algorithmus betrachten wir die *Branch-and-Bound* Methode [Weg00]. Branch-and-Bound Algorithmen werden auf NP-harte Optimierungsalgorithmen angesetzt, sodass man auf eine polynomielle

Schranke für die *worst case* Rechenzeit nicht hoffen kann. Es besteht aber die Hoffnung, dass die Rechenzeit in vielen Fällen klein und akzeptabel ist. Dies geschieht durch eine geschickte Einschränkung des Suchraums. Ein weiterer Vorteil des Branch-and-Bound Algorithmus besteht darin, dass bei vorzeitigem Abbruch des Algorithmus (etwa zur Begrenzung der Rechenzeit auf t_{limit}) für die bis dahin beste berechnete Lösung ein maximaler Abstand vom Optimum geliefert werden kann.

Ein primitives Verfahren zur Suche der optimalen Lösung könnte derart arbeiten, dass alle 2^n x-Vektoren ausprobiert werden, um zu prüfen, ob sie zulässige Lösungen enthalten und um die Zielfunktion W zu berechnen. Darauf baut der Branch-and-Bound Algorithmus auf und versucht, den Suchbereich durch einfache Tests wesentlich zu verkleinern. Die geschickte Bestimmung einer unteren und einer oberen Schranke für die Lösung des Problems sowie einer guten Suchstrategie führen dazu, dass man in der Regel relativ schnell die optimale Lösung finden kann. Der Algorithmus basiert auf effizienten Funktionen für die folgenden Teilaufgaben:

1. **Upper Bound Modul:** Berechnung einer (möglichst guten) oberen Schranke U für die Lösung des Problems. Die obere Schranke muss dabei selbst keine zulässige Lösung des Problems darstellen.

2. **Lower Bound Modul:** Berechnung einer (möglichst guten) unteren Schranke L für die Lösung des Problems. Dabei soll der Wert L eine zulässige Lösung darstellen.

3. **Branching Modul:** Zerlegung des Problems in (möglichst) disjunkte Teilprobleme (genauer: Zerlegung der Menge zulässiger Lösungen in (möglichst) disjunkte Teilmengen), die wieder vom gleichen Typ (hier Knapsack-Problem) sind.

Mit Hilfe dieser Module kann die Grundidee des Branch-and-Bound beschrieben werden:

Der Algorithmus basiert auf der Datenstruktur eines Branch-and-Bound Baumes BBB, dessen Knoten jeweils Teilprobleme P_k zugeordnet werden. Die Teilprobleme P_k, die den Blättern zugeordnet sind, bilden stets eine disjunkte Zerlegung des Gesamtproblems. Damit ist die größte untere (obere) Schranke an den Blättern eine untere (obere) Schranke für das Gesamtproblem. Für jedes entstehende Problem P_k werden die Schranken L_k und U_k mit den Modulen zur Berechnung des *Upper Bound* und *Lower Bound* berechnet. Die Schranken L_k und U_k für das Teilproblem berücksichtigen die vorher getroffenen Entscheidungen von der Wurzel des Branch-and-Bound Baumes bis zum Knoten, der das Problem P_k repräsentiert. In Abbildung 5.18 ist ein Branch-and-Bound Baum für ein Beispiel mit $n = 5$ Objekten dargestellt. Der Knoten für das Problem P_y wird nur erreicht, wenn vom Wurzelknoten beginnend, das erste Objekt nicht in den Knapsack hingenommen ($x_1 = 0$) und das zweite Objekt hineingenommen wird ($x_2 = 1$). Die weiteren Entscheidungen über x_3 bis x_4 sind nicht getroffen. Die Schranken L_y und U_y werden unter Berücksichtigung von x_1 und x_2 errechnet und gelten auch für alle Knoten, die in dem Teilbaum mit der Wurzel P_k liegen.

Der Ablauf des Algorithmus ist wie folgt:

1. Sortierung aller Objekte nach Effizienz *eff*.

2. Initialisierung des Baumes BBB mit einem Knoten, der das Gesamtproblem P_0 darstellt.

3. Berechnung der aktuellen Werte für die Schranken L und U mit den Modulen *Lower Bound* und *Upper Bound* als Maximum aller L_i bzw. U_i an den Blättern des Baumes BBB.

4. Falls $L = U$, ist die zulässige Lösung, die zu der unteren Schranke L gehört, eine optimale Lösung des Knapsack-Problems. Der Algorithmus kann terminieren.

5. Falls $L < U$, wird ein Problem P_k an seinem kritischen Objekt in zwei Teilprobleme zerlegt, die im Baum BBB Söhne von P_k werden. Fortsetzung mit Schritt 3.

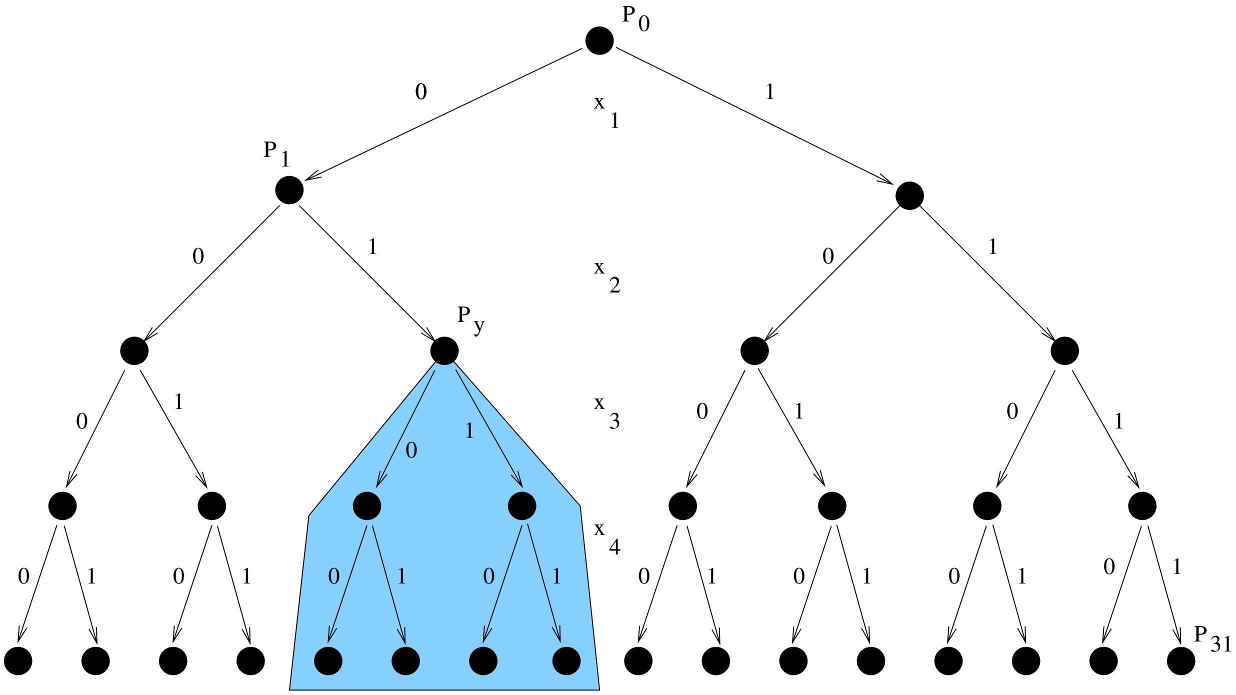


Abbildung 5.18: Branch-and-Bound Baum

Die Module für UpperBound, LowerBound und Branching arbeiten im Detail für den Knoten k wie folgt:

- Upper Bound (k)
  Dieses Modul berechnet für den Knoten k auf der Ebene l des Baumes die Obergrenze jedes Lösungswertes, der ab diesem Knoten erreichbar ist. Diese Obergrenze bedeutet, dass im gesamten Lösungsbaum, dessen Wurzelknoten k ist, keine Lösung existiert, die diese Obergrenze überschreitet. Eine derartige Aussage hat großen Einfluss auf die Entscheidung, an welchem Knoten des Baumes weitergerechnet wird und trägt somit zur Verkürzung der Laufzeit des Gesamtprogramms bei. Die Berechnung geschieht wie folgt:

  1. $SP^+ = \sum_{i=1}^{l-1} x_i * S_i$ und $W_k = \sum_{i=1}^{l-1} x_i * e_i$ (Diese Werte können auch iterativ jeweils auf Basis des Vaterknotens bestimmt werden)

  2. Bestimmung des größten m, sodass $SP^+ + \sum_{i=l}^{m} S_i \leq SP$ ist, und Bestimmung des verbleibenden Platzes $SP^* := SP - (SP^+ + \sum_{i=l}^{m} S_i)$.
  Bei der Bestimmung von m müssen zwei Fälle unterschieden werden:

  Fall 1: Falls $m = n$, dann ist $U_k = W_k + \sum_{i=l}^{n} e_i$. (Alle restlichen Objekte passen in den Knapsack)

  Fall 2: Falls $m < n$, berechne $\delta = \frac{SP^*}{S_{m+1}}$ und setze $U_k = W_k + \sum_{i=l}^{m} e_i + \delta e_{m+1}$.

  Der δ-Anteil im Fall 2 entspricht hierbei dem Teil von S_{m+1}, der noch in den Scratchpad genommen wird. Dass dies tatsächlich die obere Schranke U_k für alle möglichen Nutzwerte (=Energiegewinne) aller zur Verfügung stehenden Objekte ist, folgt aus der Sortierung der Objekte nach ihrer Effizienz. Die Effizienz aller Objekte nimmt mit fortschreitendem m ab, sodass der Platz $SP - SP^* = SP^+ + \sum_{i=l}^{m} S_i$ optimal genutzt wird. Weiterhin lässt sich feststellen, dass der restliche Platz SP^* durch den δ-Anteil von S_{m+1} mit Nutzen $\delta * e_{m+1}$ ebenfalls optimal genutzt wird.

Die Komplexität dieser Funktion beträgt $O(n)$.

- Lower Bound (k)
  Dieses Modul berechnet für den Knoten k auf der Ebene l des Baumes einen garantierten Mindestwert einer Lösung, der sich innerhalb des Lösungsraums befindet, dessen Wurzelknoten der Knoten k ist. Die Komplexität beträgt ebenfalls $O(n)$. Dabei wird wie folgt vorgegangen:

  1. Setze $SP^+ = \sum_{i=1}^{l-1} x_i * S_i$ und $L = \sum_{i=1}^{l-1} x_i * e_i$

  2. Für $i = l, ..., n$: Falls $SP^+ + S_i \leq SP$ dann $SP^+ := SP^+ + S_i$; $L_k := L_k + e_i$

  Am Ende des einmaligen Durchlaufs durch die Objekte hat L_k den Wert der unteren Schranke des Knotens k.

- Branching (k)
  Dieses letzte Modul entscheidet am Knoten k auf der Ebene l über die Variable x_l des l-ten Objektes der sortierten Folge, die als nächstes fixiert und wodurch das Problem in zwei disjunkte Teilprobleme zerlegt wird. Der entscheidende Schritt im gesamten Lösungsansatz ist das Fixieren einer Entscheidungsvariablen x_l auf einen festen Wert und die Bildung von zwei Teilproblemen. Es entstehen nun zwei Probleme:

  1. Fall (=Teilproblem) Hineinnahme:
  Durch Hineinnahme des Objektes l ($x_l = 1$) in den Knapsack wird unter allen noch verbliebenen zulässigen Lösungen die optimale Lösung gesucht. Weil sich nun erzwungenermaßen das Objekt l im Knapsack befindet, muss der verbliebene freie Platz um S_l reduziert und der Wert W_k der bisherigen Lösung im Knoten k für den entsprechenden Sohnknoten auf $W_k + e_l$ erhöht werden.
  Falls durch die Hineinnahme des Objektes l der Platz des Knapsacks nicht mehr ausreicht, kann an diesem Teilproblem die Berechnung eingestellt werden, da keine zulässigen Lösungen mehr generiert werden können. Die weitere Berücksichtigung dieses Teilbaums kann entfallen.

  2. Fall (=Teilproblem) Ausschluss:
  Die Entscheidungsvariable x_l für Objekt l wird auf 0 gesetzt, d. h. es wird unter allen zulässigen Lösungen mit $x_l = 0$ eine optimale Lösung gesucht. Der restliche Platz im Knapsack bleibt unverändert und der Wert der bisherigen Lösung W_k wird in den Sohn-Knoten übernommen.
  Anmerkung zur Suchstrategie: Grundsätzlich ist es möglich, jede der verbliebenen Entscheidungsvariablen zu wählen. Als vorteilhaft hat sich allerdings erwiesen, die Aufspaltung des Problems am "kritischen" Objekt vorzunehmen. Das kritische Objekt ist das Objekt $m+1$, das bei der Berechnung vom *Upper Bound* nicht mehr vollständig in den Knapsack passte und mit seinem δ-Anteil zu U_k beitrug.

Die Laufzeit des Algorithmus beträgt $O(n \log(n) + kn)$ mit k betrachteten Knoten, wobei für k gilt: $k \leq 2^{n+1} - 1$.

Fazit

Dieser hiermit vorgestellte Branch-and-Bound Algorithmus liefert in den meisten Fällen schnell die optimale Lösung für die Belegung des Scratchpad. Er ist daher den vorher präsentierten Verfahren (simpel, rekursiv, dynamische Programmierung) vorzuziehen. Falls die Zeitschranke t_{limit} überschritten wird, kann er terminiert werden, und mit der bis dahin gefundenen besten Lösung der Compiliervorgang fortgesetzt werden.

Es existiert allerdings noch eine Einschränkung, die bisher nicht betrachtet wurde. Es wurde davon ausgegangen, dass die Objekte - und somit auch Funktionen und Basisblöcke - voneinander unabhängig sind. Das trifft aber nicht vollständig zu. Jede Funktion lässt sich in eine Menge von Basisblöcken zerlegen. Da ein Basisblock bb, der zu einer Funktion f gehört, nicht doppelt in den Scratchpad-Speicher verschoben werden darf (1. als einzelner Basisblock, 2. als Bestandteil der Funktion), ist diese bisherige Annahme verletzt. Der bisherige Ansatz kann daher nur entweder angewendet werden auf:

1. die Menge der Funktionen oder
2. die Menge der Basisblöcke.

Die Ebene der Basisblöcke umfasst auch die Betrachtung der Funktionen, allerdings ist es in der Praxis oft schwierig, die genauen Größen und Energiegewinne für die Menge von Basisblöcken, die die Zerlegung einer Funktion darstellen, in der gleichen Genauigkeit zu berechnen, wie für die Funktion insgesamt. Es ist daher sinnvoll den bisherigen Ansatz so zu erweitern, dass sowohl Basisblöcke als auch Funktionen gleichzeitig als Objekte betrachtet werden können. In dem zuletzt vorgestellten Verfahren ist dies nur möglich, indem einzeln geprüft wird, ob eine gefundene Lösung zu den gültigen Lösungen gehört (= kein Verschieben eines Basisblockes, der gleichzeitig zu einer verschobenen Funktionen gehört).

Im nächsten Unterkapitel wird daher ein weiterer Ansatz präsentiert, bei dem diese Einschränkung der Lösung durch eine weitere Spezifikationsmethode des Problems einfach definiert werden kann.

5.4.3 Auswahl mit Integer Linear Programming

Ein Problem, in dem eine Funktion f zu maximieren oder zu minimieren ist und die weiteren Constraints (= Randbedingungen) unterliegt, wird als *mathematical programming problem* bezeichnet. Wenn es sich bei dieser Funktion f zusätzlich um eine lineare Funktion handelt, spricht man auch von *linear programming* (LP) [NW88]. Falls weiterhin die Variablen dieser Funktion f vom Typ *integer* sind, spricht man von *integer linear programming*. Mithilfe dieses integer linear programming-Ansatzes lässt sich das Problem des Knapsack einfach spezifizieren. Außerdem kann ein ILP-Problem durch am Markt verfügbare ILP-Solver gelöst werden, die beispielsweise intern mit Branch-and-Bound Verfahren arbeiten, um die optimale Lösung zu finden. Auch der hier verwendete ILP-Solver CPLEX [CPL] beinhaltet ein Branch-and-Bound Verfahren.

Die Verwendung der ILP-Technik kann hier erfolgen, weil die Formulierung des Problems mit linearen Gleichungen möglich ist und alle Variablen binäre Variablen darstellen. Man kann als Zielfunktion W bei dem hier vorliegenden Problem den Nutzen (= Energiegewinn) basierend auf dem einzelnen Nutzen E_i des Objektes i wie folgt definieren:

$$\textit{Zielffkt}: \; W = E_1 x_1 + ... + E_n x_n$$

und die Einschränkung durch die Größe des Scratchpad SP auf Basis der Größe S_i für jedes Objekt i:

$$S_1 x_1 + ... + S_n x_n \leq SP$$

Diese (Un-)Gleichungen allein reichen aus, wenn zusätzlich noch die Variablen $x_1, ..., x_n$ als binäre Entscheidungsvariablen deklariert werden. Wir definieren nun ein ILP-System mit Hilfe dieser (Un-)Gleichungen, indem wir die folgenden Typen von Memory-Objekten eines Programmes P unterscheiden:

1. die Menge aller Funktionen F,

2. die Menge von Basisblöcken BB,
3. die Menge von Variablen V,
4. den Stack St.

Für jedes Memory-Objekt x werden drei Funktionen für den jeweiligen Energiegewinn und die Größe definiert:

$$\begin{aligned} E(x) &= \text{Energieeinsparung durch Verschieben von } x \text{ in den Scratchpad} \\ S(x) &= \text{Größe von } x \text{ im Scratchpad} \end{aligned}$$

sowie eine Variable $m(x)$:

$$m(x) = \begin{cases} 1, & \text{wenn } x \text{ in den Scratchpad verschoben wird} \\ 0, & \text{sonst} \end{cases}$$

Damit kann die zu maximierende Zielfunktion für den Energiegewinn wie folgt definiert werden:

$$\begin{aligned} sav = \sum_{f_i \in F} m(f_i) * E(f_i) &\quad + \\ \sum_{bb_j \in BB} m(bb_j) * E(bb_j) &\quad + \\ \sum_{v_k \in V} m(v_k) * E(v_k) &\quad + \\ m(St) * E(St) & \end{aligned}$$

Der einzige Constraint, der unmittelbar mit einem Knapsack-Problem verbunden ist, ist die Größeneinschränkung SP durch den Scratchpad-Speicher:

$$\begin{aligned} \sum_{f_i \in F} m(f_i) * S(f_i) &\quad + \\ \sum_{bb_j \in BB} m(bb_j) * S(bb_j) &\quad + \\ \sum_{v_k \in V} m(v_k) * S(v_k) &\quad + \\ m(St) * S(St) &\quad \leq \quad SP \end{aligned}$$

Die bisherige Einschränkung im Standard-Branch-and-Bound Verfahren, dass Funktionen und Basisblöcke nicht gleichzeitig verteilt werden können, kann beim ILP-Ansatz einfach berücksichtigt werden, indem die folgenden Gleichungen für alle Basisblöcke ergänzt werden:

$$\forall bb_x \in BB : \; m(bb_x) + m(f_i) \leq 1 \, mit \, f_i \ni bb_x$$

Hiermit wird sichergestellt, dass ein Basisblock nur entweder als bb_x oder als Bestandteil der Funktion f_i ausgewählt wird.

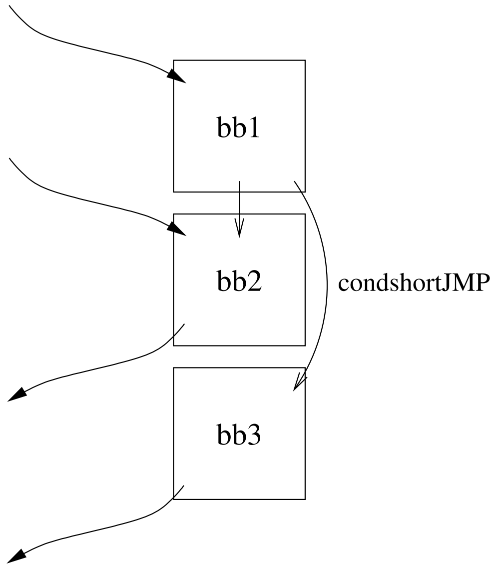


Abbildung 5.19: Zusammenhang zwischen Basisblöcken im Scratchpad

5.4.4 Ergänzung der Multibasisblöcke

Eine weitere Eigenschaft von Basisblöcken ist es, dass es entgegen der bisher im ILP-System beschriebenen Gleichungen doch Beziehungen zwischen den Basisblöcken gibt, die die Größe und den Energiegewinn bei Verschiebung in den Scratchpad-Speicher beeinflussen. In Abbildung 5.19 ist ersichtlich, dass durch entsprechende Zusammenhänge durch Sprungbefehle oder eine direkte Aufeinanderfolge im Adressraum Abhängigkeiten entstehen. Dies wurde auch schon ausführlich in Kapitel 5.4.1 betrachtet. Wenn die Basisblöcke einzeln verschoben werden, müsste mehr Platz und geringerer Energiegewinn aufgrund der dann notwendigen Verlängerung oder dem zusätzlichen Einfügen von Sprungbefehlen kalkuliert werden, d. h. die in Kapitel 5.4.1 präsentierten Abschätzungen müssen in das ILP-System integriert werden. Dieses Problem wird im Folgenden nochmals betrachtet. Bei der Bewertung des Platzbedarfs des Basisblockes $bb1$ in Abbildung 5.19 wird nicht berücksichtigt, dass, wenn Basisblock $bb2$ auch in den Scratchpad verschoben wird, ein impliziter Kontrollfluss entsteht und man einen langen relativen Sprungbefehl $longJMP$ von $bb1$ zu $bb2$ einsparen kann. Die bisherige defensive Abschätzung im ILP-System kalkuliert diesen langen relativen Sprungbefehl mit ein, sowohl beim notwendigen Platzbedarf $S(bb1)$ als auch beim Energiegewinn $E(bb1)$ aufgrund zusätzlicher Zyklen für die Ausführung des Sprungbefehls.

Diese Unterschiede haben sich als durchaus nicht vernachlässigbar gezeigt, da es ansonsten keinen "Anreiz" für das ILP-System gibt, aufeinander folgende Basisblöcke auszuwählen. Um diese Ungenauigkeit in der Abschätzung des Speicherbedarfs und des Energiegewinns in der Modellbildung des ILP-Systems zu verringern, wurde der bisherige Ansatz um so genannte "Multibasisblöcke" $mbb \in MBB$ erweitert.

Diese Multibasisblöcke mbb bestehen aus mehr als einem Basisblock, beinhalten aber andererseits auch nur Basisblöcke aus einer einzigen Funktion, da kurze Sprünge oder auch ein impliziter Kontrollfluss in den folgenden Basisblock nur innerhalb einer Funktion auftreten können. Wir generieren daher zu dem Kontrollflussgraph $CFG = (V, E)$ des Beispiels in Abbildung 5.19 einen neuen ungerichteten Graphen $CFG_{Mult} = (V, E')$ mit $E' := \{(v_1, v_2)|(v_1, v_2) \in E \wedge (v_1, v_2) \neq longJMP\}$ (siehe Abb. 5.20), bei dem die Kanten aufgrund von langen Sprungbefehlen nicht enthalten sind. Eine Kante e steht daher für einen kurzen Sprung oder den impliziten Kontrollfluss zweier Basisblöcke bei der Hintereinanderpositionierung im Adressraum. Wir bilden nun alle Teilgraphen $g \subseteq CFG_{Mult}$, die zusammenhängend sind. Jeder Teilgraph $g = (V'', E'')$ stellt einen Multibasisblock $mbb = V''$ dar. Für das Beispiel werden daher Multibasisblöcke mbb für die folgenden Kombinationen gebildet, die in Abbildung 5.20b, c und d

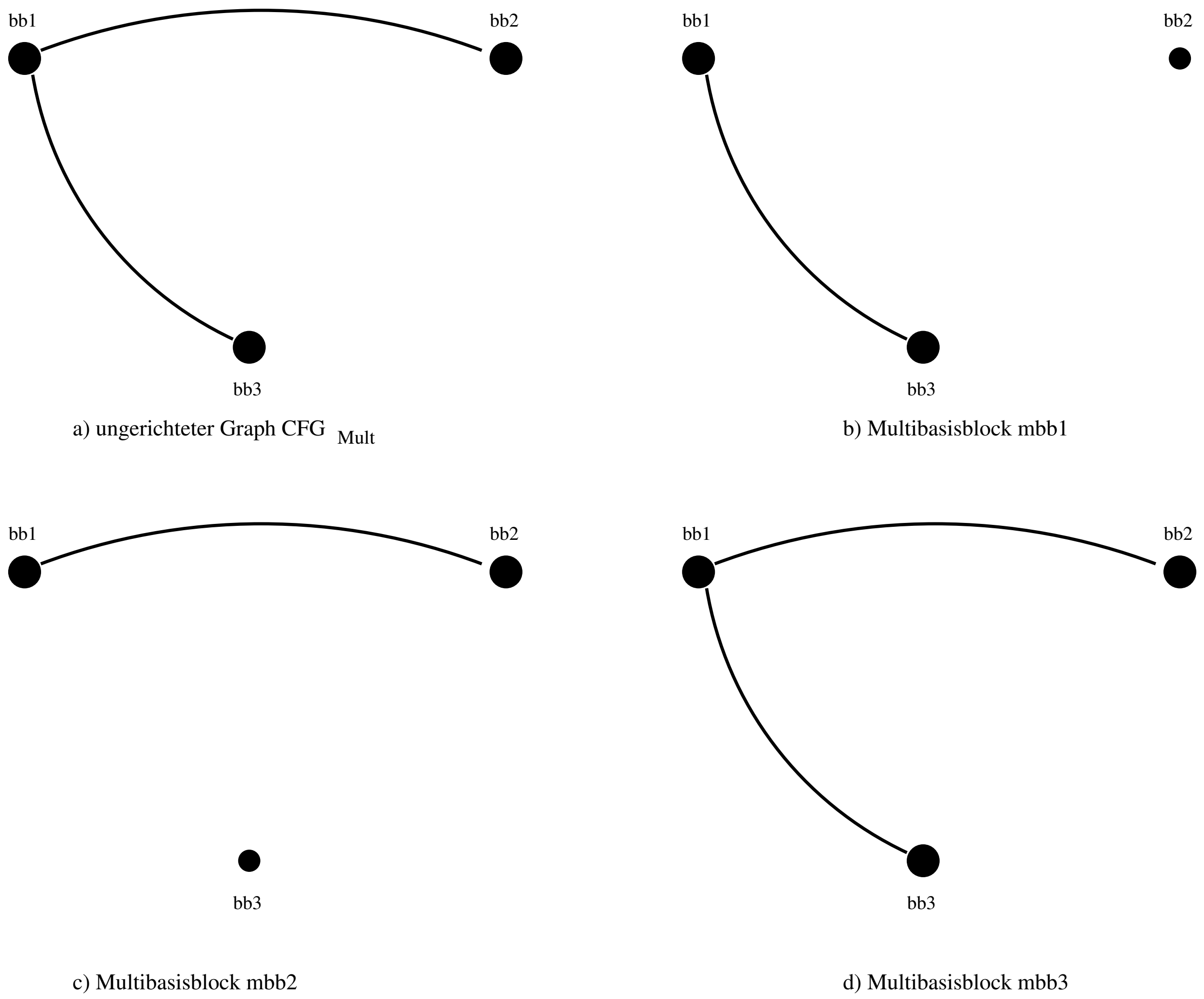


Abbildung 5.20: Kontrollflussgraph für Multibasisblöcke

dargestellt sind:

$$MBB = \{\{bb1, bb2\}, \{bb1, bb3\}, \{bb1, bb2, bb3\}\}$$

Die Kombination $\{bb2, bb3\}$ entfällt, da die Basisblöcke $bb2$ und $bb3$ zwar im Adressraum aufeinander folgen, jedoch kein Zusammenhang im Kontrollfluss besteht.

Zur Berücksichtigung dieser Multibasisblöcke müssen die bisherigen Gleichungen des ILP-Systems erweitert werden.[3] Wir definieren einen *vollständigen Multibasisblock* als einen Multibasisblock, der nicht echte Teilmenge eines anderen Multibasisblockes ist:

Definition vollständiger Multibasisblock: *ein Multibasisblock $x \in MBB$ ist vollständig $\Leftrightarrow \nexists mbb \in MBB : x \subset mbb$*

Von den betrachteten Multibasisblöcken ist nur der Block $\{bb1, bb2, bb3\}$ ein vollständiger Multibasisblock. Bei Compilern, die auf Standard-Codegenerierungstechniken basieren, wird ein vollständiger Multibasisblock alle Basisblöcke einer Funktion, zu der er gehört, umfassen.

Die Größe des Multibasisblockes mbb berechnet sich wie folgt:

$$S(mbb) = \sum_{bb \in mbb} S(bb) + \mathit{sizeOffset}(mbb)$$

Der $\mathit{sizeOffset}(mbb)$ berechnet sich analog wie in Kapitel 5.4.1 für die Basisblöcke beschrieben und besteht aus zusätzlichem Platz für Sprungbefehle aus dem Scratchpad heraus. Für vollständige Multibasisblöcke gilt, dass $\mathit{sizeOffset}(mbb) = 0$ gilt. Dies liegt darin begründet, dass von dem Scratchpad keine kurzen Sprünge aus diesem Multibasisblock heraus zu anderen Basisblöcken existieren.

Der Energiegewinn eines Multibasisblockes mbb beträgt:

$$E(mbb) = \sum_{bb \in mbb} E(bb) + \mathit{EnergyOffset}(mbb)$$

Der $\mathit{EnergyOffset}(mbb)$ entspricht der Summe des Energieverbrauchs aller Sprünge zu einem der Basisblöcke in mbb und von einem der Basisblöcke in mbb zu einem Basisblock außerhalb von mbb. Die Berechnung erfolgt analog der Beschreibung in Kapitel 5.4.1.

Das erweiterte ILP-System besteht nun aus den folgenden Elementen:

Zielfunktion:

$$\begin{aligned} sav = \sum_{f_i \in F} m(f_i) * E(f_i) & \quad + \\ \sum_{bb_j \in BB} m(bb_j) * E(bb_j) & \quad + \\ \sum_{mbb_n \in MBB} m(mbb_n) * E(mbb_n) & \quad + \\ \sum_{v_k \in V} m(v_k) * E(v_k) & \quad + \\ m(St) * E(St) & \end{aligned}$$

[3] Bei einem angewendeten Function Inlining muss dieser Ansatz erweitert werden, da dann auch kurze Sprünge zwischen mehreren Funktionen auftreten können.

Größen-Constraint des Speichers:

$$\begin{aligned}
\sum_{f_i \in F} m(f_i) * S(f_i) &\quad + \\
\sum_{bb_j \in BB} m(bb_j) * S(bb_j) &\quad + \\
\sum_{mbb_n \in MBB} m(mbb_n) * S(mbb_n) &\quad + \\
\sum_{v_k \in V} m(v_k) * S(v_k) &\quad + \\
m(St) * S(St) &\quad \leq \quad SP
\end{aligned}$$

der ergänzten Einschränkung für den gegenseitigen Ausschluss von Objekten, die einen gemeinsamen Basisblock besitzen:

$$\forall bb_x \in BB : \, m(bb_x) + m(f_i) + \sum_{mbb_n \ni bb_x} m(mbb_n) \leq 1 \, mit \, f_i \ni bb_x$$

5.4.5 Ergebnisse

Nach der theoretischen Vorstellung wird nun in einer Reihe von Experimenten anhand von Beispielprogrammen die Einsparung nachgewiesen. Als Vergleichsmaßstab zu einem ARM7T-Prozessor mit Scratch-

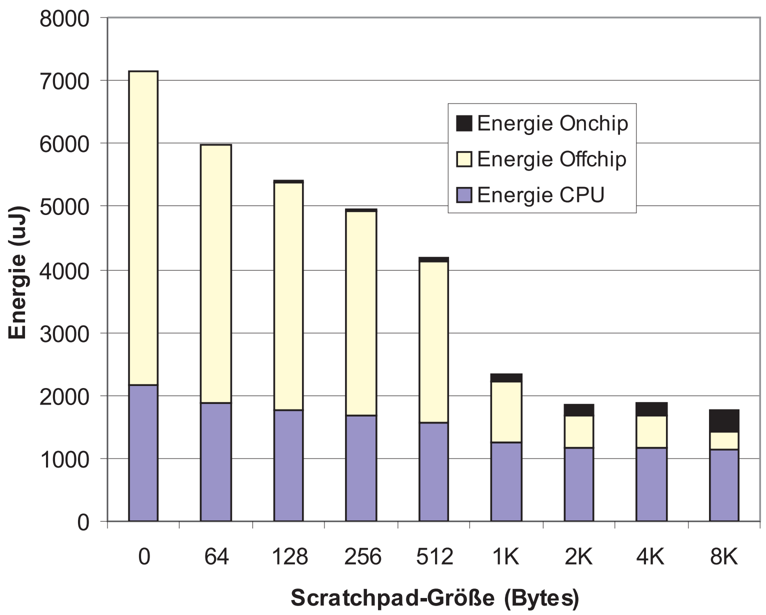


Abbildung 5.21: *fast-idct* Benchmark

pad wird ein ARM7T-Prozessor mit gleich großem Onchip-Cache betrachtet, da die vorgestellte Methode in Zusammenhang mit einem Scratchpad insbesondere der Ersetzung eines Cache-Systems dienen kann.

In Abbildung 5.21 sind die Energiewerte für ein Scratchpad-System dargestellt, bei dem die Größe des Scratchpads variiert wird. Beispielhaft wird das Programm *fast_idct* betrachtet, bei welchem es sich um eine inverse diskrete Cosinus-Transformation handelt. Diese Transformation ist z. B. in einem MP3-Decoder enthalten, der zur Klasse der typischen mobilen Applikation gehört [Hül02]. Beginnend mit der Speichergröße 0, also einem System ohne Onchip-Speicher, wird anschließend ein 64 Bytes großer Scratchpad-Speicher in seiner Kapazität jeweils verdoppelt. Ohne Onchip-Speicher wird mit ca. 70 % der weit überwiegende Teil der Energie im Speicher verbraucht. Durch die Nutzung des Scratchpad-Speichers können schrittweise immer mehr Anteile des Programms und der Daten in diesen verlagert werden, sodass die Anzahl der Speicherzugriffe auf den Offchip-Speicher und somit auch der Energieverbrauch stark reduziert werden können. Da der Onchip-Speicher pro Zugriff mit einem geringeren Energieverbrauch als der Offchip-Speicher auskommt, kann der Energieverbrauch bei Nutzung eines Scratchpad-Speichers von 8 KBytes um insgesamt 75% reduziert werden, obwohl zu beachten ist, dass der Scratchpad bei ansteigender Größe pro Zugriff intern auch mehr Energie verbraucht. Der Energieverbrauch des Prozessors reduziert sich ebenfalls, da der Onchip-Speicher schneller ist und die Anzahl der Wartezyklen bei den Speicherzugriffen geringer wird. Weiterhin müssen der externe Adress- und der Datenbus für den Speicherzugriff nicht angesteuert werden, wodurch ebenfalls der Energieverbrauch reduziert wird.

Im Folgenden soll eine obere Grenze für die Energieeinsparung bestimmt werden, um feststellen zu können, wie nahe das theoretische Maximum approximiert wird. Als Parameter verwenden wir hierfür den Energieverbrauch beim Instructionfetch aus dem Scratchpad $E_{onchip,if}$ und aus den Hauptspeicher $E_{offchip,if}$, wobei diese beiden Werte sowohl die Speicherenergie als auch die Prozessorenergie umfassen. Für die Einsparung bei Verschiebung vom Offchip- in den Onchip-Speicher ergibt sich:

$$E_{rel} = \frac{Energiegewinn}{vorheriger\ Energieverbrauch} = \frac{E_{offchip,if} - E_{onchip,if}}{E_{offchip,if}}$$

Weiterhin ist noch die Anzahl der Zyklen pro Instruktion relevant (= CPI), da nicht in jedem Zyklus ein Speicherzugriff für einen Instructionfetch ausgeführt wird. Die maximale Energieeinsparung ergibt sich bei Instruktionen mit einem CPI-Wert von 1, der hier auch angesetzt wurde, da dann in jedem Zyklus ein Instructionfetch erfolgt. Durch die durchgeführten Messungen mit dem betrachteten ARM7T-System sind die benötigten Instructionfetch-Kosten $E_{onchip,if}$ und $E_{onchip,if}$. mit $38,68\,nJ$ und $5,61\,nJ$ bekannt. Für dieses System existiert eine obere Schranke für die maximale Einsparung bei Verschiebung von Programmteilen daher:

$$max.\,Einsparung = \frac{38,68 - 5,61}{38,68} nJ = 85\%$$

Die beim *fast_idct* erreichte Energieeinsparung von 75% ist daher schon recht nahe an dieser oberen Schranke, da auch die zusätzlichen Sprungbefehle mit berücksichtigt werden müssen.

Verschiebung von Programmteilen

Nach der Einschätzung der maximal möglichen Energieeinsparung bei genügend großem Speicher sollen nun die Ergebnisse bei Anwendung des Verfahrens einzeln präsentiert und diskutiert werden. Neben dem *fast-idct* Benchmark wird in den folgenden Untersuchungen der *heapsort* Benchmark verwendet, der als typischer Sortieralgorithmus bekannt ist. Als erste Klasse von Memory-Objekten wird in den Abbildungen 5.22 und 5.23 betrachtet, wie sich die alleinige Verschiebung von Funktionen darstellt. In Konkurrenz dazu können später Basisblöcke allein oder die Kombination von Funktionen und Basisblöcken betrachtet werden. Die Anzahl der ausgeführten Instruktionen bleibt bei der alleinigen Verschiebung von Funktionen konstant, da keine zusätzlichen Sprungbefehle eingefügt werden müssen. In den beiden Abbildungen ist zu erkennen, dass nicht bei jeder Vergrößerung des Speichers auch dieser entsprechend genutzt werden kann. Da nur ganze Funktionen verschoben werden, hängt es von der Größe der Funktionen ab, wann wieder eine weitere Funktion zusätzlich in den Scratchpad verschoben werden kann. Mit hoher Wahrscheinlichkeit ist

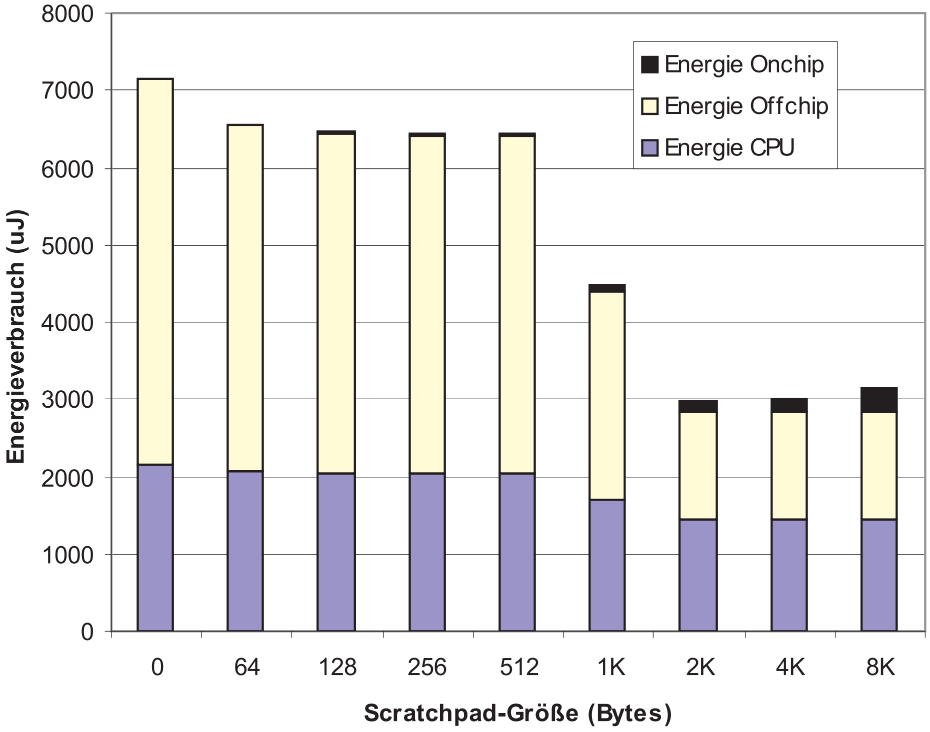


Abbildung 5.22: *fast-idct* Benchmark mit Funktionen im Scratchpad

daher stets ein Teil des Speichers ungenutzt. Alternativ kann statt der Funktionen die Zerlegung in Basisblöcke betrachtet werden. Dies geschieht in den beiden Abbildungen 5.24 und 5.25. Es ist zu sehen, dass häufiger Veränderungen in den Werten auftreten, was durch die geringere Größe der Basisblöcke im Vergleich zu den Funktionen zu erklären ist. Da jedoch teilweise zusätzliche Sprünge beim Verschieben eines Basisblockes eingefügt werden müssen, wird die Anzahl der ausgeführten Instruktionen nochmals speziell betrachtet. Die Anzahl der ausgeführten Instruktionen ist für das Beispiel des *fast_idct* in Abbildung 5.26 dargestellt. Die ausgeführten Instruktionen "wandern" vom Offchip-Speicher in den Onchip-Speicher. Wenn sich ein Teil des Programms im Offchip- und ein anderer Teil im Onchip-Speicher befindet, kann es notwendig sein, zusätzliche Sprünge einzufügen, um den Anschluss an einen in das Scratchpad verschobenen Basisblock wiederherzustellen. Bei der Scratchpad-Größe von 512 Bytes und 1024 Bytes ist dies ersichtlich. Die zusätzlichen 498 bzw. 744 Instruktionen (142.656 - 142.158 = 498 bzw. 142.902 - 142.158 = 744) entsprechen diesen zusätzlich ausgeführten Sprungbefehlen. Allerdings beträgt der Anteil dieser zusätzlichen Instruktionen an der Gesamtzahl der Instruktionen weniger als 0,5% und ist daher für die Gesamtbewertung nicht erheblich. Er muss aber trotzdem bei der Berechnung des Energiegewinns für ein einzelnes Memory-Objekt berücksichtigt werden, wenn zusätzliche Sprungbefehle notwendig sind. Auf dieser Ebene kann der Anteil der Sprungbefehle erheblich sein und entscheidenden Einfluss auf die Wahl der Memory-Objekte zeigen.

Nach der Einzelbetrachtung der Programm-Objekte der Funktionen und Basisblöcke erfolgt nun ein direkter Vergleich dieser beiden Alternativen. In den Abbildungen 5.27 und 5.28 werden diese verschiedenen Ansätze gegenübergestellt. Die Wahl von Basisblöcken erfolgt in kleineren Schritten, allerdings ist beim *fast-idct* Benchmark für die Speichergrößen 256 bis 1024 Bytes der Energiegewinn geringer. Ursache sind hierfür die zusätzlichen Sprünge bei der Variante für die Basisblöcke und die Ungenauigkeiten in den Abschätzungen des Energiegewinns. Es kann daher günstiger sein, die Funktionen auch als Ganzes zu

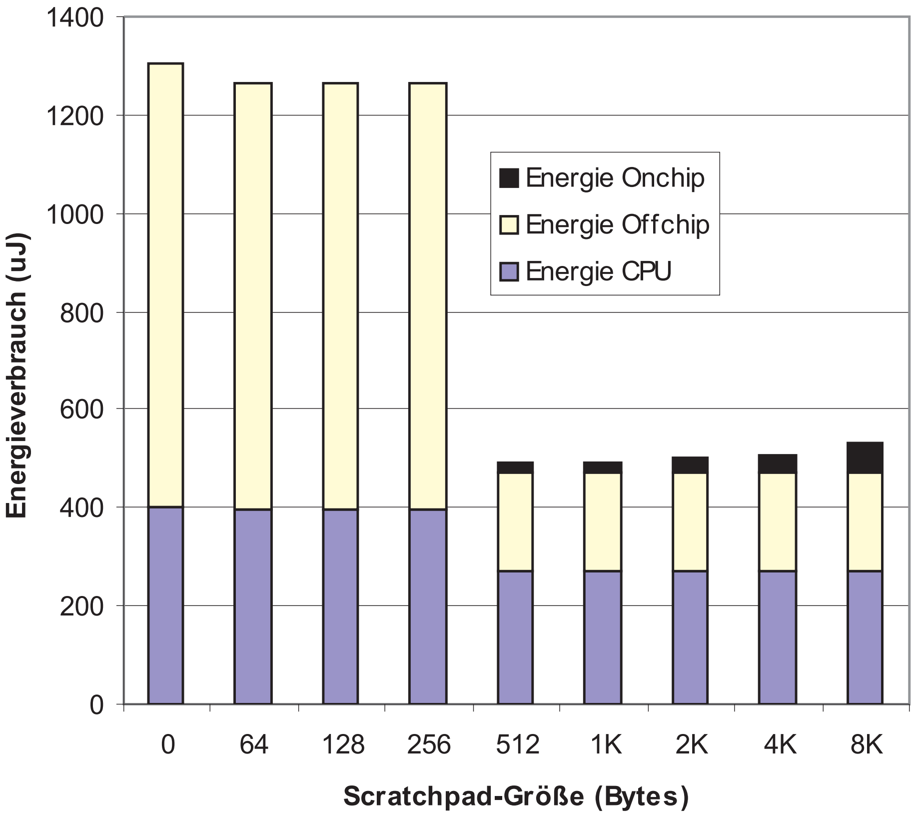


Abbildung 5.23: *heapsort* Benchmark mit Funktionen im Scratchpad

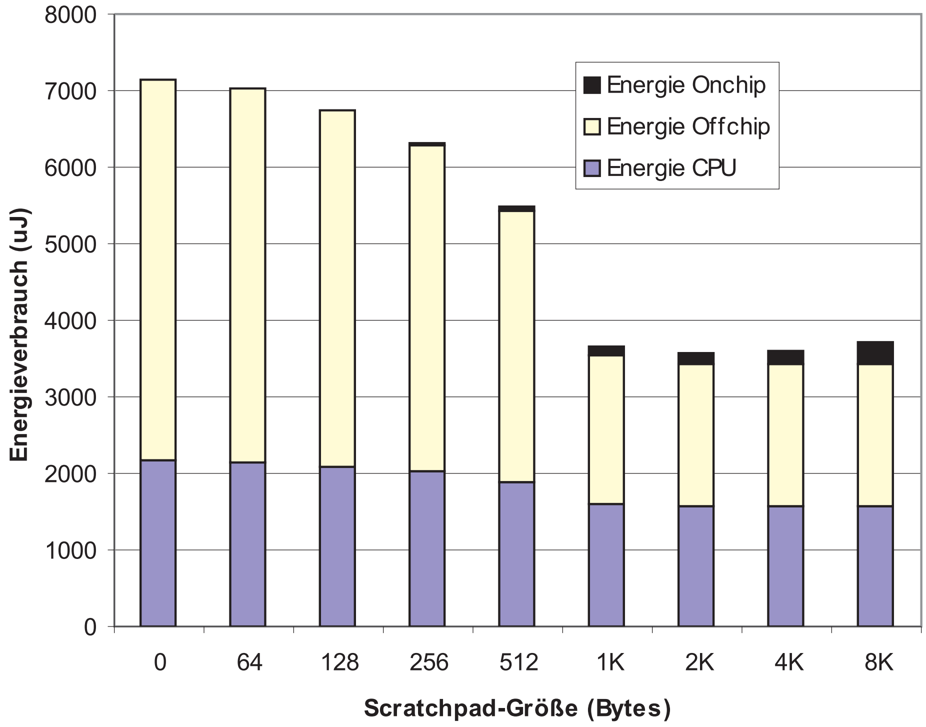


Abbildung 5.24: *fast-idct* Benchmark mit Basisblöcken im Scratchpad

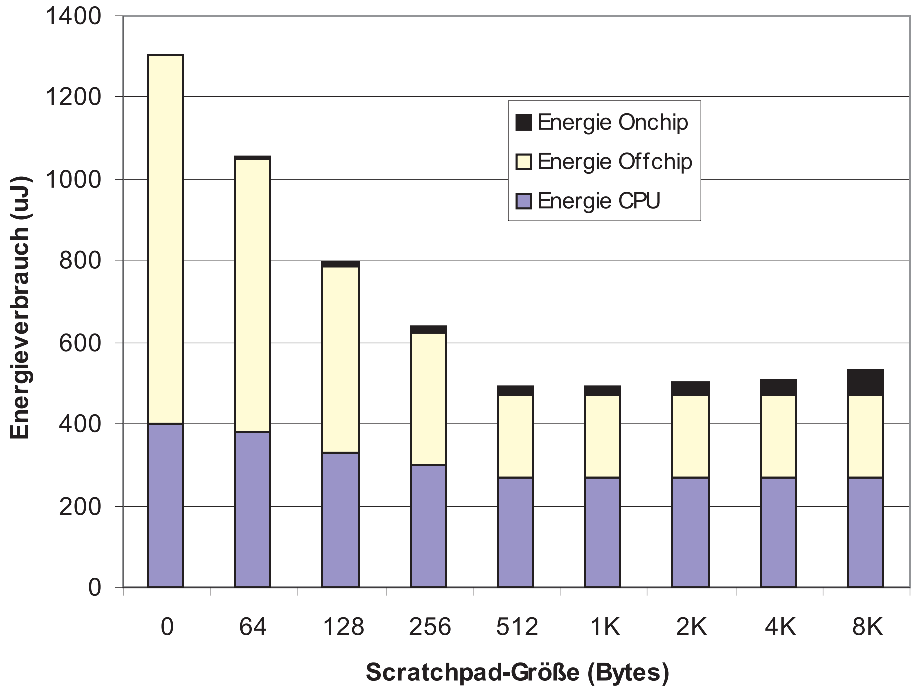
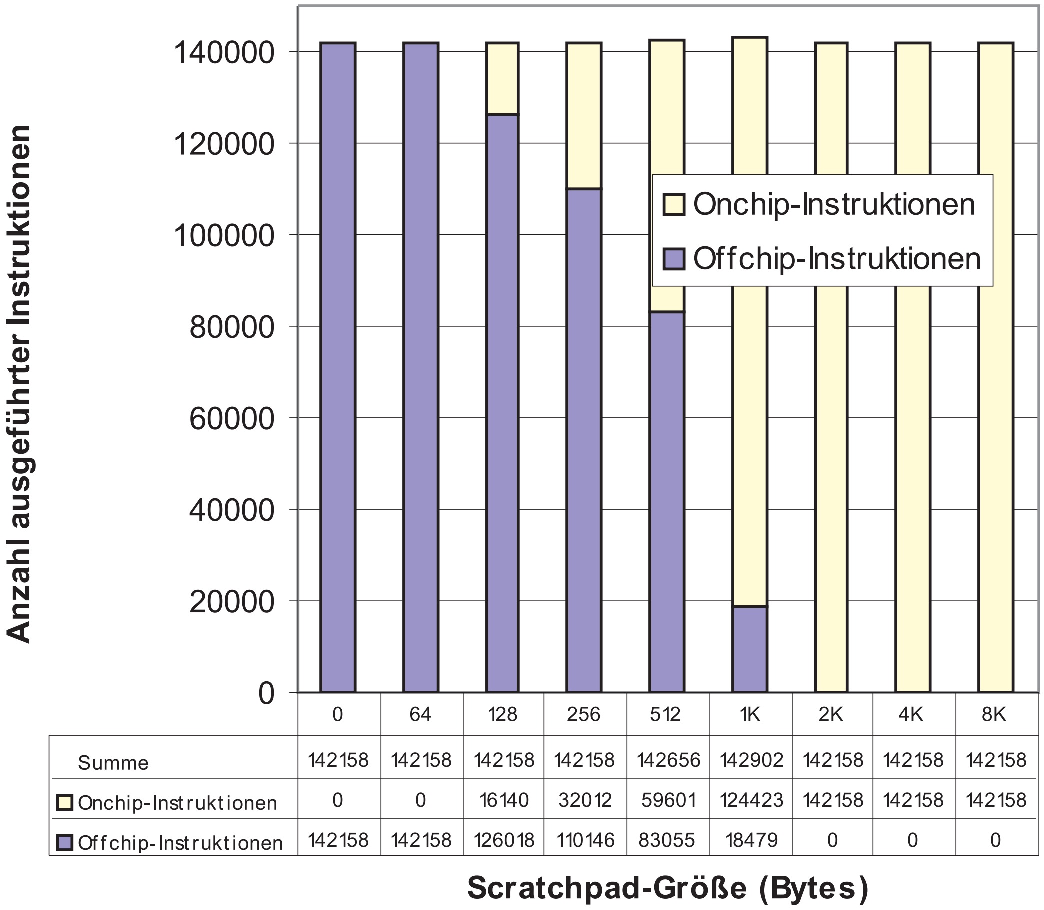
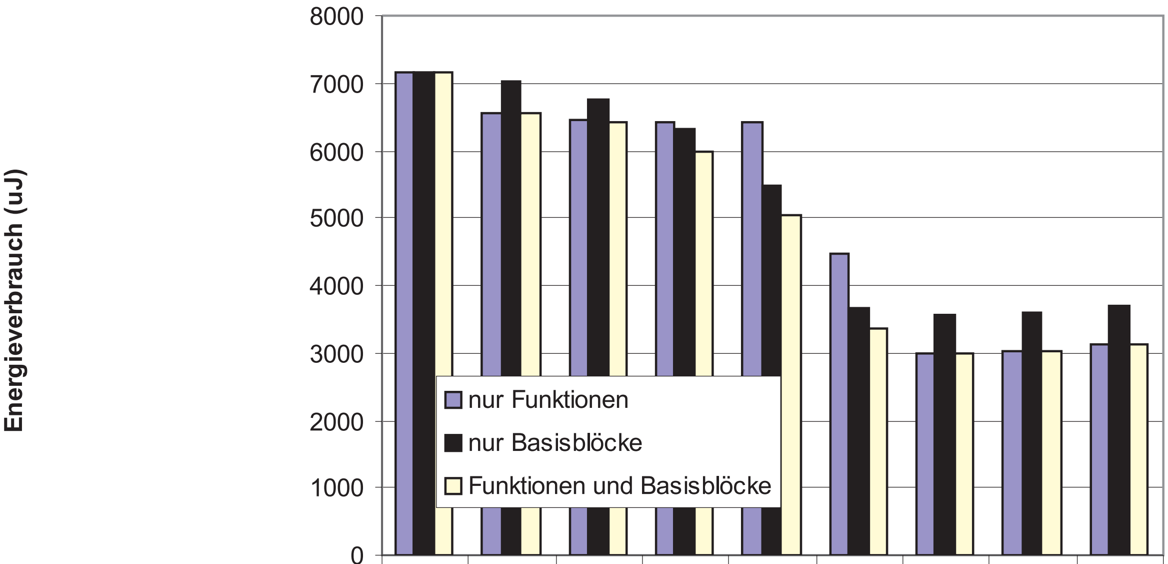


Abbildung 5.25: *heapsort* Benchmark mit Basisblöcken im Scratchpad



	0	64	128	256	512	1K	2K	4K	8K
Summe	142158	142158	142158	142158	142656	142902	142158	142158	142158
Onchip-Instruktionen	0	0	16140	32012	59601	124423	142158	142158	142158
Offchip-Instruktionen	142158	142158	126018	110146	83055	18479	0	0	0

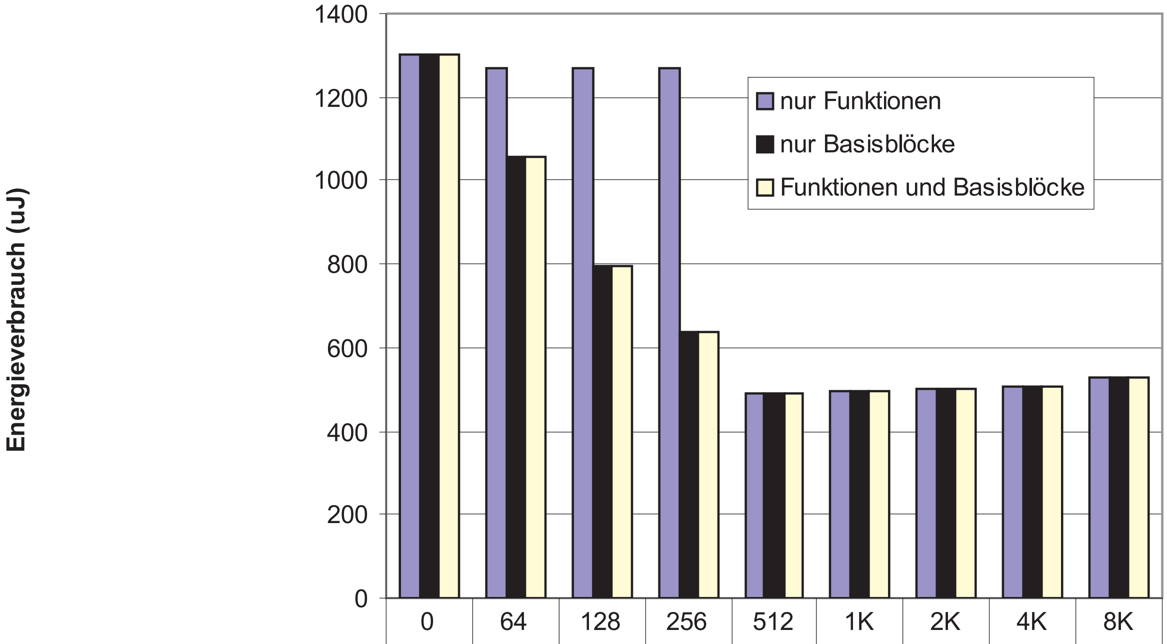
Abbildung 5.26: *fast-idct* Benchmark mit Anzahl ausgeführter Instruktionen



	0	64	128	256	512	1K	2K	4K	8K
nur Funktionen	7149,1	6553,5	6458,1	6426,7	6428,6	4477	2990,5	3010,9	3136,7
nur Basisblöcke	7149,1	7028,7	6740,5	6316,1	5483,2	3650	3575,3	3593,2	3703,1
Funktionen und Basisblöcke	7149,1	6.553,5	6433,2	5979,5	5026	3363,7	2990,5	3010,9	3136,7

Scratchpad-Größe (Bytes)

Abbildung 5.27: *fast-idct* Benchmark mit alternativen Objekten Funktionen, Basisblöcken sowie deren Kombination



	0	64	128	256	512	1K	2K	4K	8K
nur Funktionen	1303,7	1266,8	1266,8	1266,9	489,96	493,75	501,05	505,14	530,26
nur Basisblöcke	1303,7	1057,2	796,98	637,66	489,96	493,75	501,05	505,14	530,26
Funktionen und Basisblöcke	1303,7	1057,2	796,98	637,66	489,96	493,75	501,05	505,14	530,26

Scratchpad-Größe (Bytes)

Abbildung 5.28: *heapsort* Benchmark mit alternativen Objekten Funktionen, Basisblöcken sowie deren Kombination betrachten. Am besten schneidet insgesamt die Kombination aus Memory-Objekten für Funktionen und für Basisblöcke ab. Für den *fast-idct* Benchmark bei einer Scratchpad-Größe von 512 Bytes ist damit beispielsweise eine Kombination möglich, die sowohl besser als eine nur auf Funktionen als auch besser als eine nur auf Basisblöcken vorgenommene Optimierung ist.

Das gleiche Ergebnis ist auch beim *heapsort* Benchmark in Abbildung 5.28 festzustellen. Insbesondere fällt hier die bessere Granularität der Basisblöcke im Vergleich zu den Funktionen auf. Während die erste Funktion erst bei einer Größe von 512 Bytes verschoben werden kann, können Basisblöcke schon ab 64 Bytes verlagert und der Energieverbrauch verringert werden. In diesem Beispiel ist die Verschiebung von Basisblöcken und alternativ die Kombination aus Basisblöcken und Funktionen die beste Lösung.

Verschiebung von Daten

Wir haben somit die Möglichkeiten für die Verschiebung von Programm-Objekten betrachtet und wenden uns als Nächstes den Datenobjekten zu.

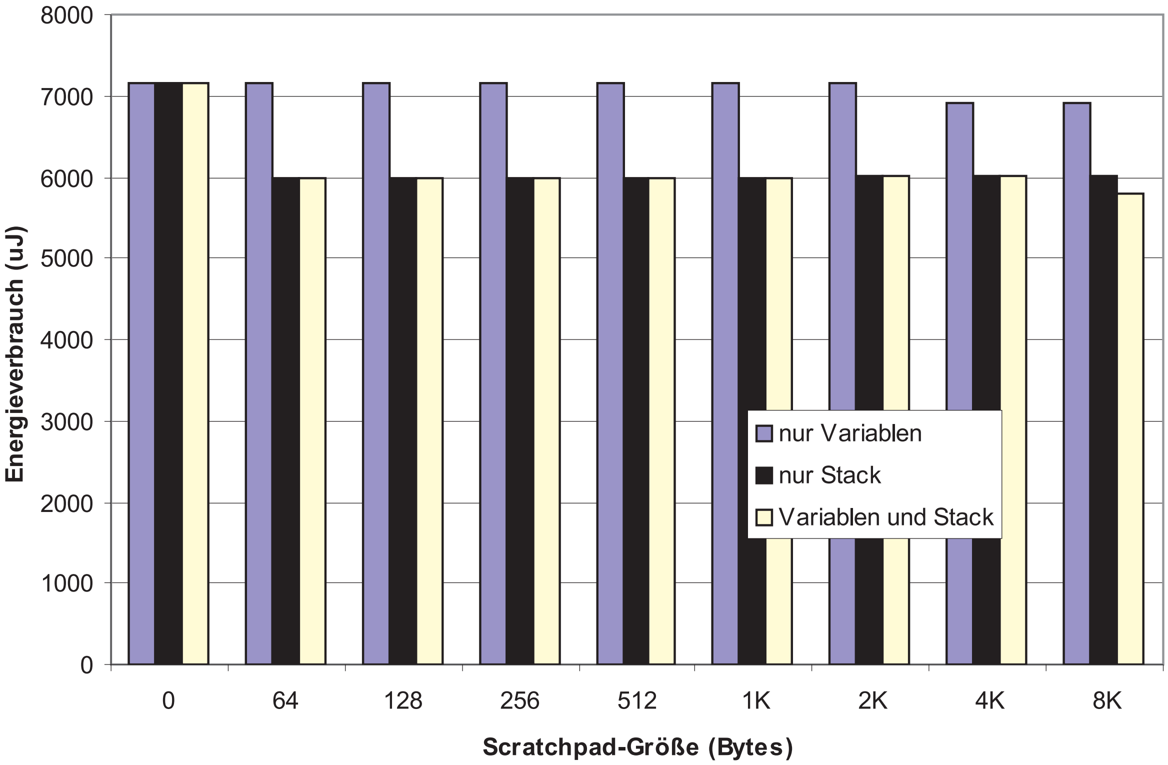


Abbildung 5.29: *fast-idct* Benchmark mit alternativen globalen Variablen und Stack-Objekten

Bei den Datenobjekten unterscheiden wir im Wesentlichen lokale Objekte, die sich auf dem Stack befinden, und globale Objekte. Bei den schon bisher betrachteten Benchmarks werden in den Abbildungen 5.29 und 5.30 die vorhandenen globalen Daten verschoben. Beim *fast-idct* Benchmark existieren insgesamt drei globale Objekte: ein Array mit 32*32 Einträgen vom Typ *short* mit einem Speicherbedarf von insgesamt 2.048 Bytes, ein Array mit 1.024 Elementen vom Typ *integer* mit einem Speicherbedarf von insgesamt 4.496 Bytes und eine skalare Variable mit 4 Bytes Speicherbedarf. Der *heapsort* Benchmark besitzt lediglich ein globales Speicherobjekt, ein Array mit 100 Elementen bestehend aus jeweils 4 Bytes und somit einem Gesamtspeicherbedarf von insgesamt 400 Bytes.

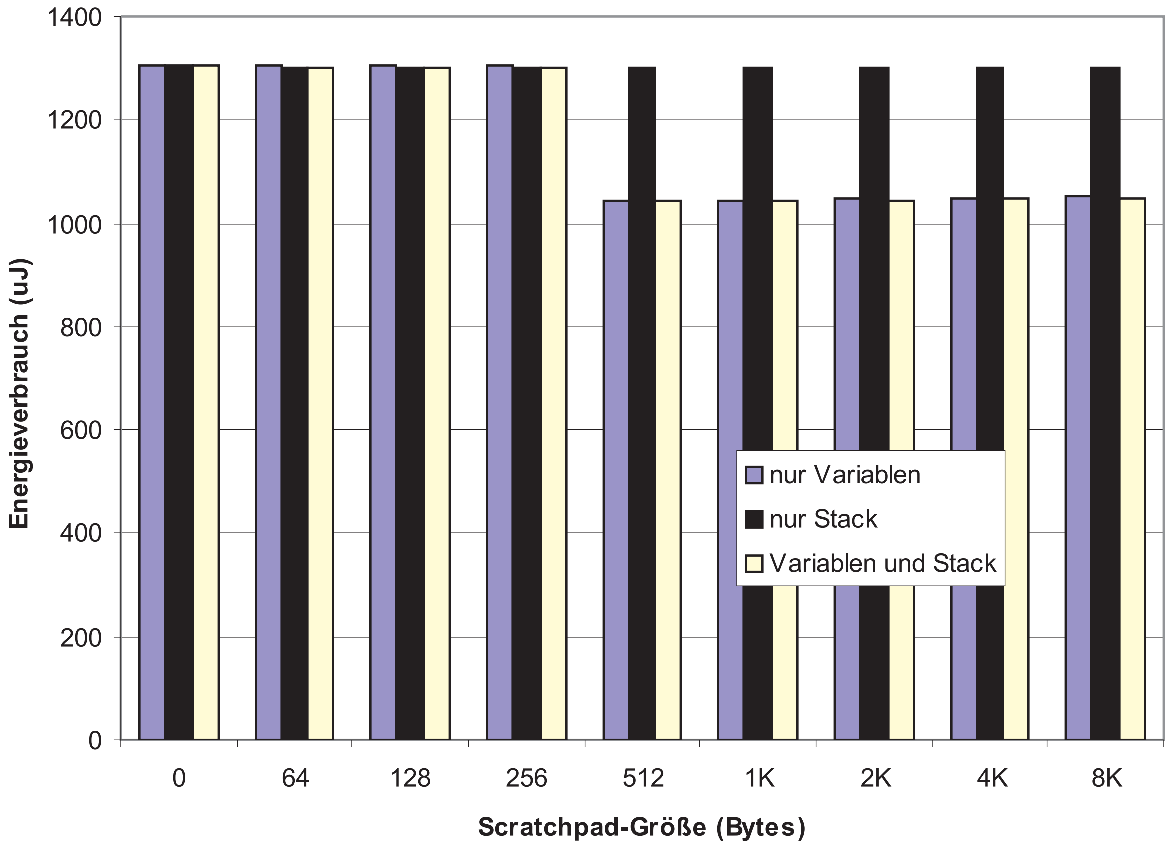


Abbildung 5.30: *heapsort* Benchmark mit alternativen globalen Variablen und Stack-Objekten

Beim *fast-idct* Benchmark in Abbildung 5.29 wird schon bei 64 Bytes Onchip-Speichergröße der Energieverbrauch durch die Verschiebung des Stacks stark verringert. Für die Verschiebung der globalen Variablen ist ein größerer Onchip-Speicher notwendig, sodass erst ab einer Größe von 4 KBytes eine sichtbare Einsparung möglich wird. Die Vorteile der Kombination gegenüber der alleinigen Verschiebung von Stack oder globalen Variablen treten bei 8 KBytes auf. Beim *heapsort* Benchmark in Abbildung 5.30 wird nur durch die Verschiebung des globalen Speicherobjektes eine Energiereduzierung erreicht, da auf dem Stack keine größeren Aktivitäten zu verzeichnen sind. Insgesamt wird durch das Verfahren auch bei Datenobjekten die optimale Menge an Objekten durch die Kombination der verschiedenen Klassen von Objekten ausgewählt und verschoben.

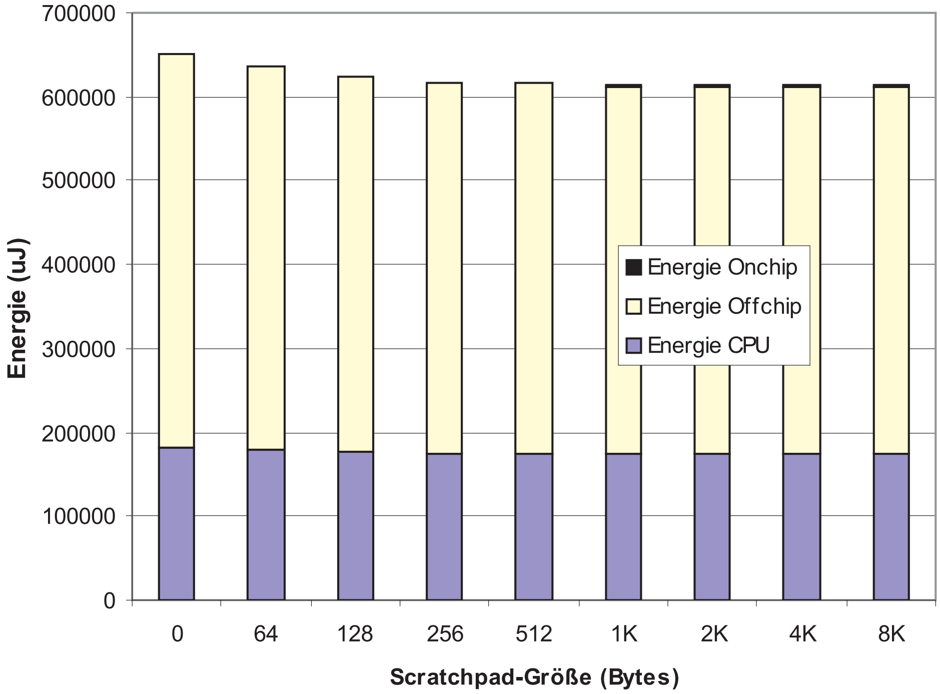


Abbildung 5.31: *ref-idct* Benchmark mit Bibliotheksfunktionen

Einschränkungen

Die Voraussetzung für die Anwendbarkeit dieses Algorithmus ist, dass das Programm dem Compiler vollständig bekannt sein muss. Diese trivial scheinende Forderung gilt beispielsweise nicht für Bibliotheksfunktionen. Diese werden erst beim Linkvorgang mit eingebunden; während des vorherigen Compilerlaufs sind nur die Aufrufe bekannt. Informationen über die Größe von Bibliotheksfunktionen und ihre innere Struktur sind nicht verfügbar. Programme wie *ref_idct* (siehe Abb. 5.31), die einen großen Teil ihrer Laufzeit in Bibliotheksfunktionen verbringen, erfordern eine entsprechende Berücksichtigung dieser Bibliotheksfunktionen, da die Nutzung des Scratchpads ansonsten kaum Vorteile liefert und sich der Energieverbrauch bei einem 8 KBytes großen Scratchpad nur um ca. 5% verringert. Die Ursache ist der zugrunde liegende Datentyp *double float*, der fast ausschließlich verwendet wird. Da Floating Point Operationen in Ermangelung einer Floating Point Unit im verwendeten Prozessor durch Softwarefunktionen in der Standard-Bibliothek ausgeführt werden, wird der überwiegende Teil der Laufzeit und der Energie in der Bibliothek verbracht. Um Programme dieser Art optimieren zu können, muss daher das Verfahren die Häufigkeiten des Aufrufs von Bibliotheksfunktionen sowie den durchschnittlichen Energieverbrauch in diesen Funktionen und deren Größe ermitteln. Mit diesen Informationen können auch Bibliotheksfunktionen als Memory-Objekte betrachtet und das vorstehende ILP-System um diese Objekte erweitert werden.

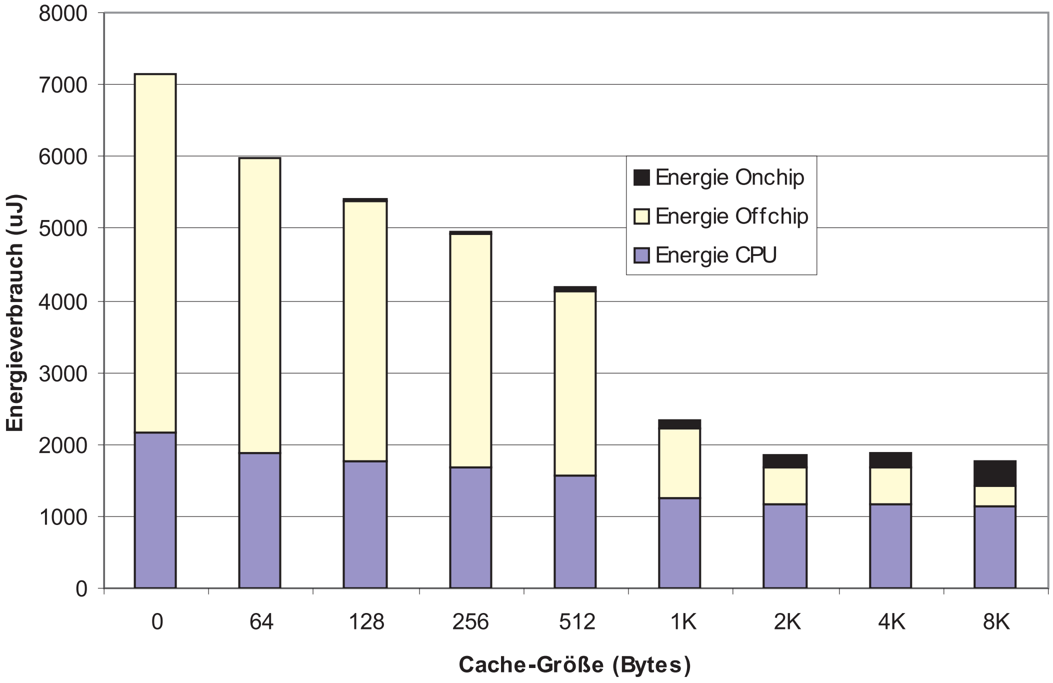
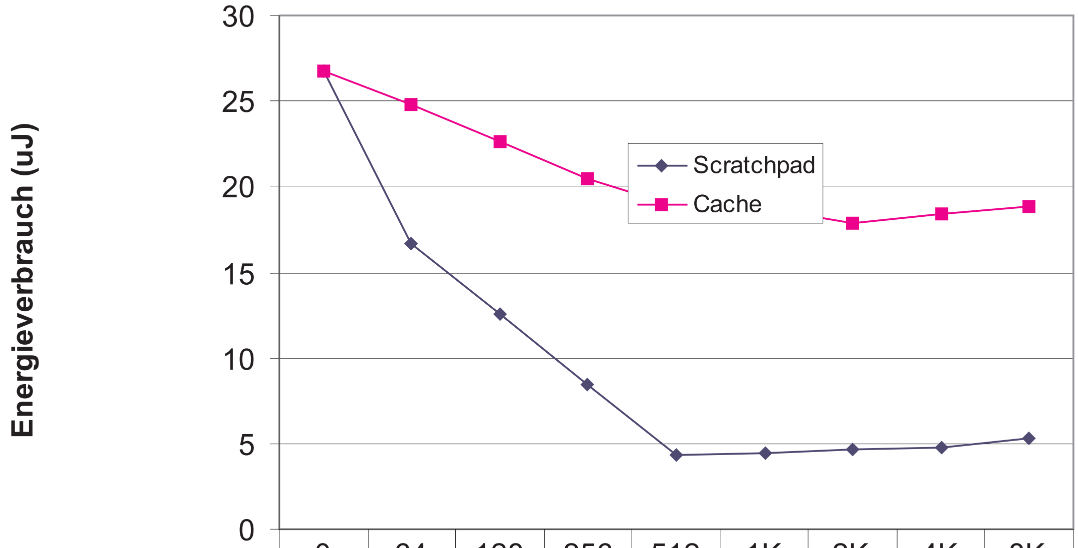


Abbildung 5.32: *fast-idct* Benchmark mit Cache

Vergleich mit einem Cache-System

Nach der Einzelbetrachtung der Klassen von Memory-Objekten folgt nun der Vergleich mit dem alternativen Cache-System. Dieser Vergleich ist für verschiedene Benchmarks in den Abbildungen 5.35 bis 5.40 zu finden. Bei den Benchmarks handelt es sich um bekannte und häufig verwendete Sortierverfahren *bubblesort*, *heapsort*, *insertionsort*, *quicksort* und *selectionsort* sowie die Filterapplikationen *biquad_N_sections* und *lattice*. In Abbildung 5.32 wird der *fast_idct* Benchmark mit den Energiewerten für unterschiedliche Cache-Größen dargestellt. Es ist ersichtlich, dass bei sehr kleinen Caches die Energieverbrauchswerte gegenüber Systemen ohne Caches ansteigen. Die Ursache ist die hohe Anzahl von Cache-Misses, die zum häufigen Austausch von Cache-Inhalten mit hohen Energiekosten führen. Entscheidend ist der Gesamtvergleich der Kurven, bei dem man einen klaren Vorteil für die Scratchpad-Variante erkennen kann. Nur in sehr wenigen Fällen liegt der Cacheenergiewert unterhalb des Scratchpads. In Tabelle 5.2 sind für die verschiedenen Benchmarks die günstigsten Energiewerte einmal für den Scratchpad und zum anderen für den Cache aufgeführt. Der größte untersuchte Onchip-Speicher von 8 KBytes kann von einigen Benchmarks nicht vollständig genutzt werden. Daher ist ein Vergleich mit verschiedenen, den jeweiligen Benchmarks entsprechenden Onchip-Speichergrößen geeigneter. Die gewählte Onchip-Größe in Tabelle 5.2 wurde so bestimmt, dass sich für den Cache ein minimaler Gesamtenergieverbrauch ergibt. Für den entsprechenden Vergleich mit dem Scratchpad-Speicher wurde die gleiche Größe wie beim Cache gewählt.

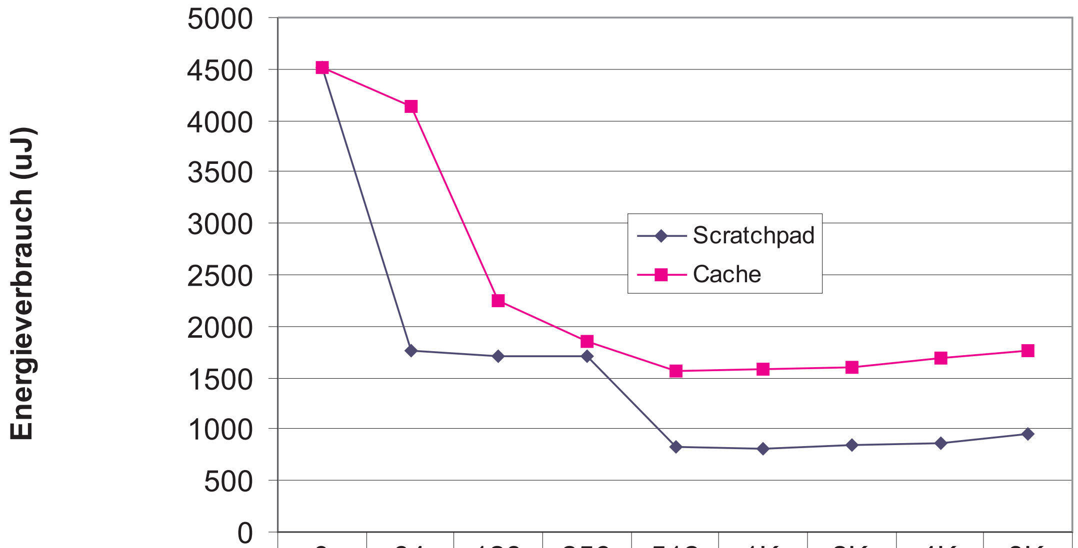
Die Ergebnisse zeigen die geringste Verbesserung des Energieverbrauchs durch den Scratchpad im Vergleich zum Cache in Höhe von 8% beim Benchmark *insertionsort* und einer Onchip-Größe von 512 Bytes. Bei Betrachtung des vollständigen Vergleichs dieses Benchmarks in Abbildung 5.37 ist zu erkennen, dass



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	26,73	16,72	12,53	8,481	4,383	4,469	4,634	4,727	5,296
Cache	26,73	24,78	22,6	20,49	18,83	18,75	17,83	18,4	18,9

Onchip-Größe (Bytes)

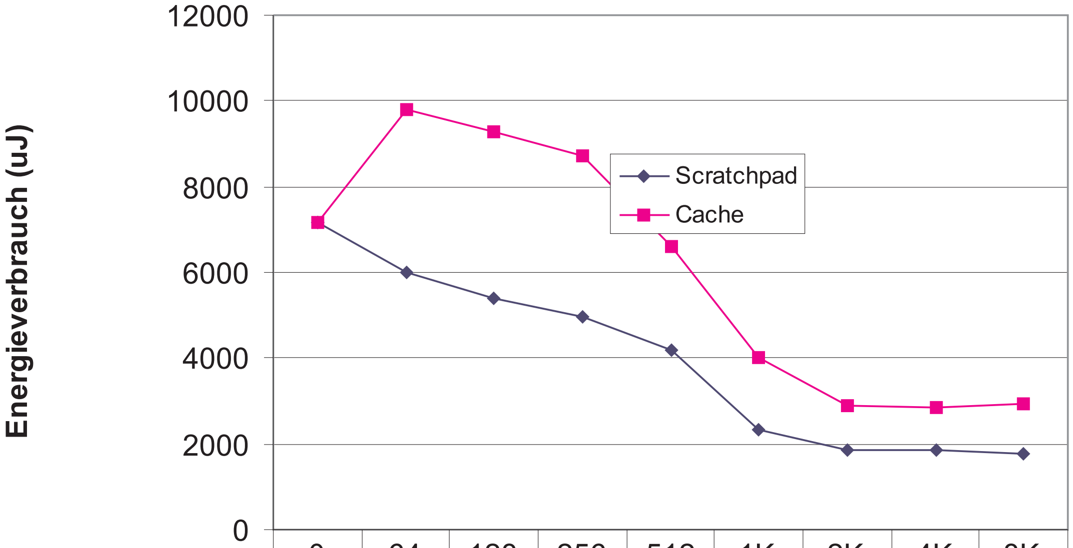
Abbildung 5.33: *biquad_N_sections* Benchmark Scratchpad vs. Cache



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	4508	1770	1712	1708	818,8	817,6	846,5	862,6	961,9
Cache	4508	4130	2243	1852	1559	1577	1607	1685	1763

Onchip-Größe (Bytes)

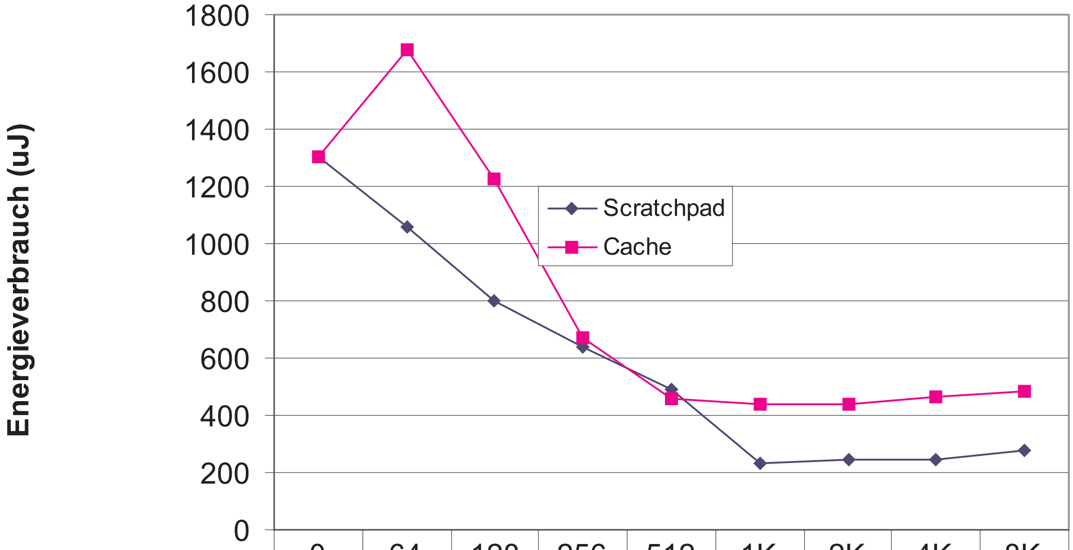
Abbildung 5.34: *bubblesort* Benchmark Scratchpad vs. Cache



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	7149	5993	5398	4959	4176	2334	1845	1868	1773
Cache	7149	9794	9282	8708	6587	4020	2888	2846	2947

Onchip-Größe (Bytes)

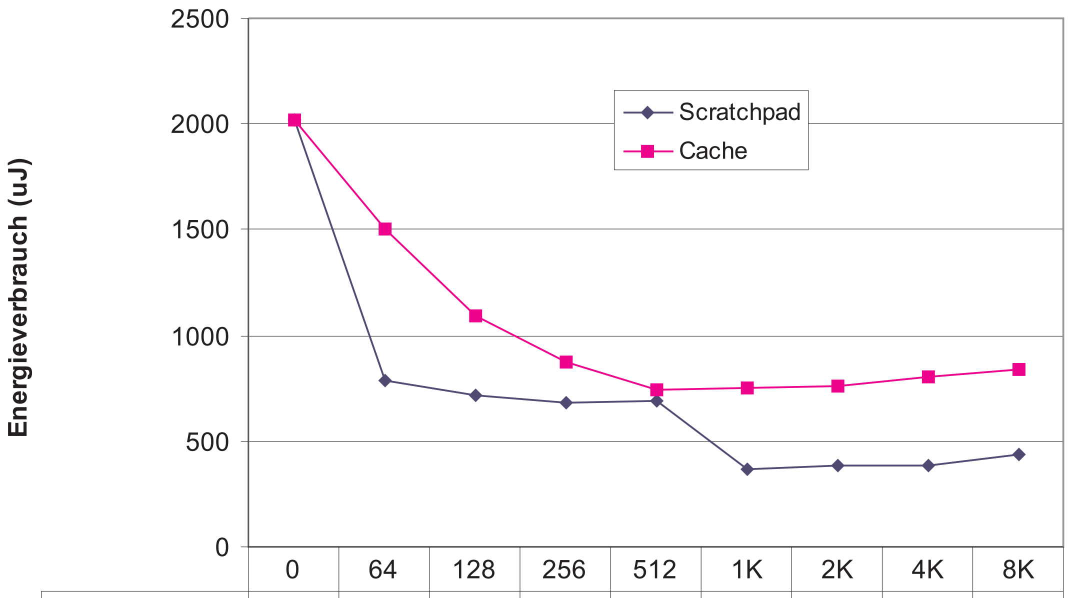
Abbildung 5.35: *fast-idct* Benchmark Scratchpad vs. Cache



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	1304	1057	797	637,7	489,1	234,5	242,8	247,5	276,2
Cache	1304	1677	1223	673,5	458,2	437,3	440,8	462	484,9

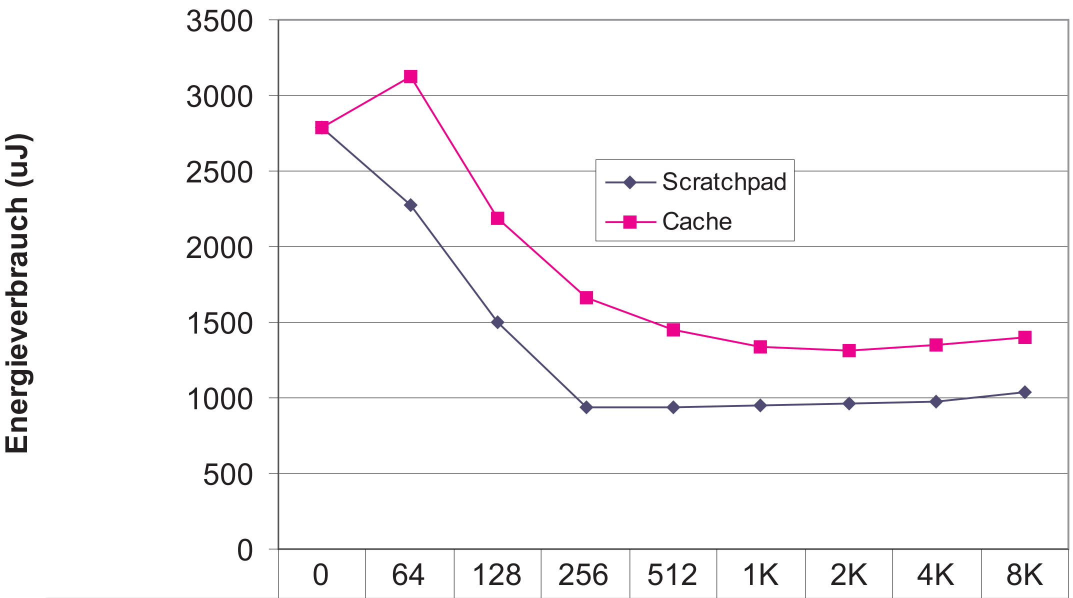
Onchip-Größe (Bytes)

Abbildung 5.36: *heapsort* Benchmark Scratchpad vs. Cache



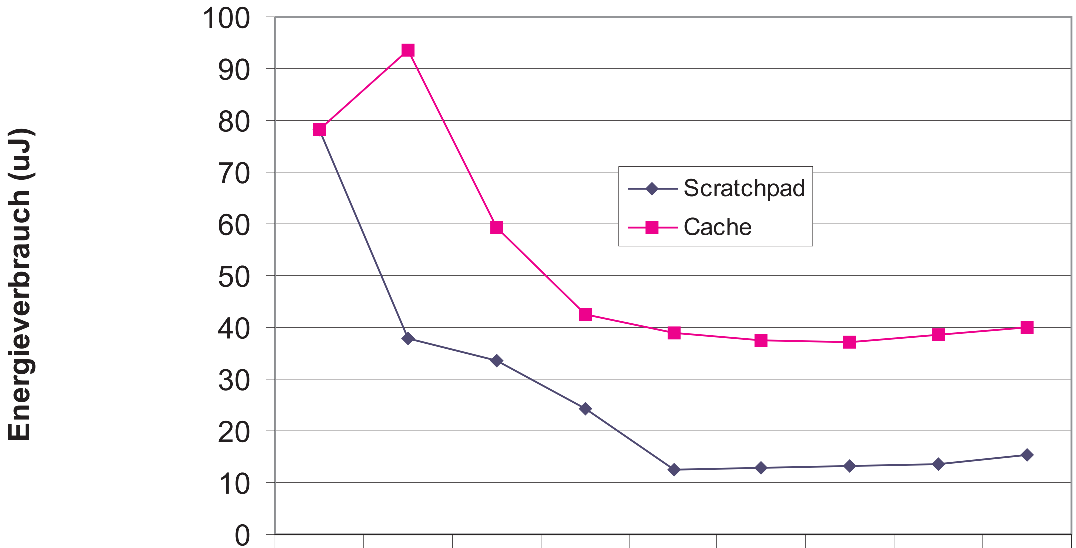
	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	2019,9	786,69	715,07	683,29	687,12	367,69	381,01	388,46	434,27
Cache	2019,9	1501,1	1095,9	874,14	746,97	752,28	764,44	799,85	836,31

Abbildung 5.37: *insertionsort* Benchmark Scratchpad vs. Cache



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	2782	2270	1495	939,2	942,5	951,1	967,6	976,9	1034
Cache	2782	3129	2188	1662	1454	1341	1309	1351	1398

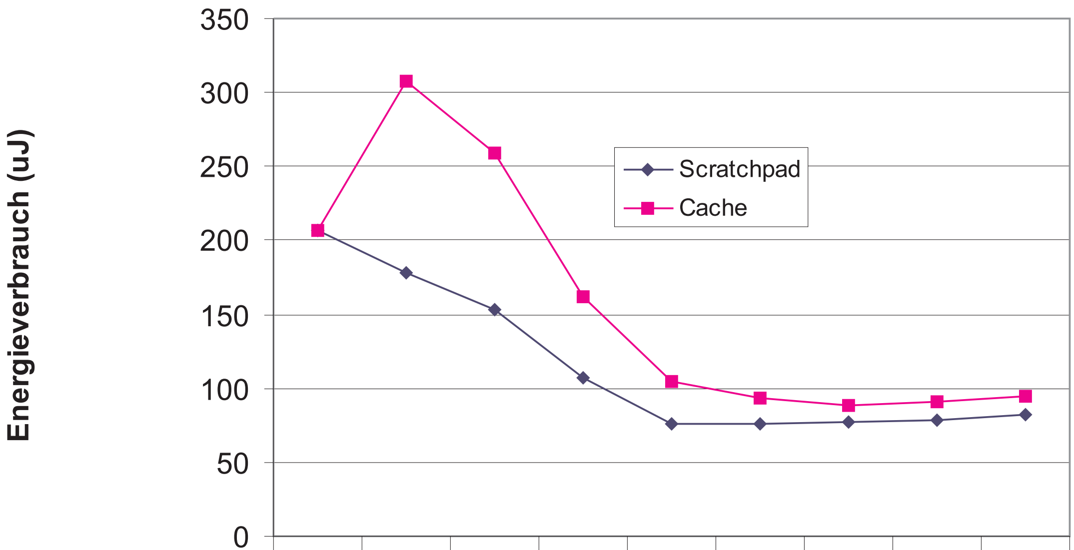
Abbildung 5.38: *lattice* Benchmark Scratchpad vs. Cache



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	78,09	38	33,72	24,31	12,59	12,85	13,34	13,62	15,32
Cache	78,09	93,65	59,19	42,43	38,87	37,38	37,19	38,44	39,86

Onchip-Größe (Bytes)

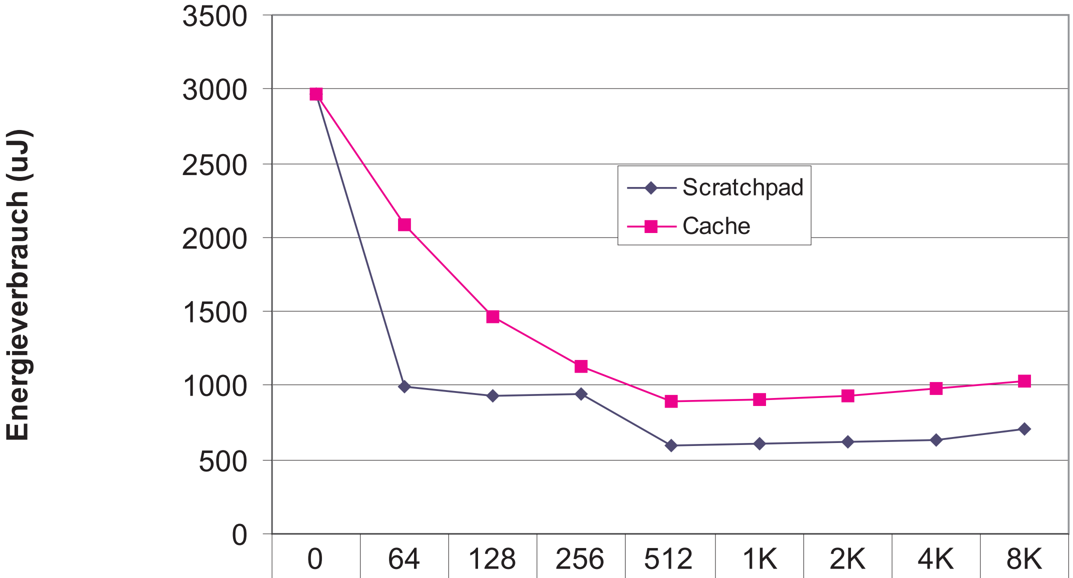
Abbildung 5.39: *matrixmult* Benchmark Scratchpad vs. Cache



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	206,5	178,6	152,9	107,7	75,74	76,29	77,35	77,94	81,59
Cache	206,5	307,3	259,4	161,3	105,2	92,83	88,75	91,01	94,21

Onchip-Größe (Bytes)

Abbildung 5.40: *quicksort* Benchmark Scratchpad vs. Cache



	0	64	128	256	512	1K	2K	4K	8K
Scratchpad	2964	993,2	936,5	941,7	597,8	607,2	626,7	638,7	704,9
Cache	2964	2079	1461	1131	896,2	907,3	927,6	980,2	1033

Onchip-Größe (Bytes)

Abbildung 5.41: *selectionsort* Benchmark Scratchpad vs. Cache

gerade bei dieser Speichergröße der Scratchpad-Algorithmus im Vergleich zur Speichergröße von 256 Bytes diesen zusätzlichen Speicherraum nicht ausnutzen konnte. Es ist vorstellbar, dass bei weiteren Verbesserungen des Algorithmus dieses Potenzial noch ausgeschöpft werden kann. Beispielsweise könnten größere Datenobjekte teilweise in den Scratchpad verschoben werden (siehe auch Verma et al. [VSM03]).

Insgesamt kann für die untersuchten Benchmarks eine durchschnittliche Verbesserung von 38,8% festgestellt werden. Bei diesem Wert muss berücksichtigt werden, dass als Vergleichsmaßstab die Onchip-Größe gewählt wurde, die für den Cache am besten geeignet ist, sodass in dieser Hinsicht der Vergleich den Scratchpad etwas benachteiligt. Das Suchen der optimalen Lösung mit dem ILP-Modell benötigte für die verwendeten Benchmarks eine Laufzeit von durchschnittlich unter 0,1s.

Es ist weiterhin zu berücksichtigen, dass die Kosten für die beiden Systemarchitekturen unterschiedlich sind. Da der Cache eine nennenswert größere Fläche benötigt [BSL+01, BSL+02], sind die Kosten für einen Prozessor mit Onchip-Cache höher als bei einem Prozessor mit Scratchpad gleicher Speichergröße. Insbesondere für Systeme mit hoher Stückzahl ist dies oft ein entscheidender Gesichtspunkt.

Neben der Optimierung auf Energieverbrauch ist es interessant zu untersuchen, wie die Verfahren sich auf die Performance auswirken. Für die gleichen Benchmarks und Onchip-Größen werden daher in Tabelle 5.3 die Performance-Werte dargestellt. Die Analyse zeigt eine durchschnittliche Steigerung der Performance um 21,8%. Daher ist dieser Algorithmus auch für die weit verbreitete Performance-Optimierung einsetzbar und liefert dort ebenfalls nennenswerte Verbesserungen, wenn auch die Verbesserungen des Energieverbrauchs höher sind.

Ein zusätzlicher Energiegewinn kann durch Ausnutzen dieses Performancegewinns erreicht werden, indem ein Reduzieren der Spannung am Prozessor (so genanntes *Voltage Scaling*) - sofern der jeweilige Prozessor diese Eigenschaft besitzt - den Prozessor verlangsamt, wodurch der Energieverbrauch in etwa quadratisch im Verhältnis zur Spannung reduziert wird.

Benchmark	Onchip-Größe (Bytes)	Scratchpad-Energie (μJ)	Cache-Energie (μJ)	Verbesserung (%)
biquad_N_sections	512	4,383	18,827	76,7
bubblesort	512	818,82	1.558,6	47,5
fast-idct	4K	1.868,1	2.845,0	34,3
heapsort	1K	234,45	437,26	46,4
insertionsort	512	687,12	746,97	8,0
lattice	2K	967,6	1.308,6	26,1
matrixmult	2K	13,343	37,191	64,1
quicksort	2K	77,348	88,746	12,8
selectionsort	512	597,84	896,25	33,3
Durchschnitt				38,8

Tabelle 5.2: Scratchpad- vs. Cache-Energie

Benchmark	Onchip-Größe (Bytes)	Scratchpad-Perform. (Zyklen)	Cache-Perform. (Zyklen)	Verbesserung (%)
biquad_N_sections	512	843	1733	51,4
bubblesort	512	161.973	205.344	21,1
fast-idct	4K	247.944	310.825	20,2
heapsort	1K	45.821	57.499	20,3
insertionsort	512	88.800	96.813	8,3
lattice	2K	114.966	145.191	20,8
matrixmult	2K	2.426	3.927	38,2
quicksort	2K	9.190	9.787	6,1
selectionsort	512	122.937	136.222	9,8
Durchschnitt				21,8

Tabelle 5.3: Scratchpad- vs. Cache-Performance

Fazit

In diesem Kapitel wurde das Verfahren zur statischen Verlagerung von Programmteilen und Daten mit den Phasen der Analyse sowie der Berechnung der Energiewerte und Größen vorgestellt. Das Problem wurde mit unterschiedlichen theoretischen Ansätzen modelliert, von denen sich ein ILP-Modell als das am besten geeignete Verfahren zeigt. Die Wahl der Objekte für eine statische Belegung des Scratchpad wurde daher optimal mit einem ILP-Ansatz vorgenommen (sofern die Zeitschranke t_{limit} nicht erreicht wurde) und die Ergebnisse im letzten Abschnitt zeigen die Vorteile, die insbesondere gegenüber heute verwendeten Cache-Systemen vorliegen. Beim Vergleich der Performance zwischen einem System mit Scratchpad und einem System mit Cache kann mit dem Scratchpad eine zwischen 6,1% und 51,4% reduzierte Laufzeit erzielt werden. Die durchschnittliche Performance-Steigerung für die betrachteten Benchmarks beträgt 21,8%. Beim Energieverbrauch ist das Verhältnis zu Gunsten des Scratchpad noch höher, mit Werten zwischen 12,8% und 76,7% und einer durchschnittlichen Energieeinsparung von 38,8%. Die Ergebnisse wurden auch durch Steinke et al. [SWLM02] veröffentlicht.

Der Ansatz der Nutzung des Scratchpad ist viel versprechend und bietet neben der Energieeinsparung insbesondere auch den Vorteil des geringeren Flächenbedarfs. Einzige Bedingung ist eine entsprechende Erweiterung des verwendeten Compilers um das vorgestellte Verfahren.

Eine Eigenschaft des Scratchpad-Algorithmus soll im Folgenden noch näher betrachtet werden. Bisher werden die Speicherobjekte zur Compilezeit ausgewählt und beim Laden in den Speicher einmalig in den Scratchpad geladen. Ein Austausch von Objekten, wie es beim Cache geschieht, findet nicht statt. Es ist jedoch denkbar, dass ein Austausch während der Programmlaufzeit vorteilhaft ist, wenn die folgenden Bedingungen zutreffen:

1. Der Benchmark besitzt mehrere Hotspots, von denen jeder erst häufig durchlaufen wird, bevor die Programmkontrolle zum anderen Hotspot wechselt

2. Die Hotspots passen nur einzeln in den Scratchpad aber nicht zusammen

Im nachfolgenden Kapitel soll daher der bisherige statische Ansatz zu einem dynamischen Ansatz weiterentwickelt werden.

5.5 Dynamische Verschiebung

5.5.1 Auswahl des Verfahrens

Wir betrachten nach dem statischen Verfahren mit der festen Zuordnung von Objekten zu Offchip- oder Onchip-Speicher nun ein Kopieren von Objekten während der Programm-Laufzeit in den Scratchpad. Dies ist vergleichbar mit dem Cache-Verhalten, wobei dort das Einlagern hardwaremäßig gesteuert wird. Dieses Einlagern einer Cacheline bzw. eines -blockes geschieht bei einem Cache sehr effizient, da im Gegensatz zur Softwarelösung für den Scratchpad-Ansatz nicht alle Daten einzeln in den Prozessor und von dort aus wieder in den Scratchpad geladen werden müssen. Der Cache ist beim Einlagern im Vorteil, wobei dafür die Entscheidung, welche Objekte wann und in welcher Größe hereingeholt werden, nur sehr einfach getroffen wird. Ein Objekt wird im Allgemeinen bei seiner Verwendung zusammen mit der vollständigen Cacheline in den Cache geholt. Dies kann durch einen software-gesteuerten Algorithmus gezielter geschehen und eventuell die Nachteile durch einen längeren Kopiervorgang beim Scratchpad wieder ausgleichen.

Zuerst ist zu entscheiden, welche Memory-Objekte für ein dynamisches Austauschen zur Verfügung stehen. Es sind insbesondere die Programmteile, wie Funktionen, Basisblöcke oder auch Multibasisblöcke hierfür geeignet. Das Austauschen von globalen Variablen, wie es bereits betrachtet wurde, ist grundsätzlich auch möglich, wird aber zurückgestellt. Die letzte Klasse von Memory-Objekten, der Stack, kann nicht verlagert werden, da auf ihn sehr häufig von vielen Stellen im Programm aus zugegriffen wird. Wir fokussieren die Analysen und Betrachtungen daher auf die Programmteile und geben später einen Ausblick für eine Einbeziehung von globalen Variablen.

Für den Entwurf eines Verfahrens zur Verlagerung von Memory-Objekten in den Speicher und das Austauschen während des Programmlaufes gibt es mehrere unabhängige Design-Entscheidungen zu treffen. Diese insgesamt vier Dimensionen des Suchraums sind in Abbildung 5.42 dargestellt und werden nachfolgend im Detail betrachtet.

Die Dimensionen des Entscheidungsraums für das dynamische Verlagern der Objekte sind:

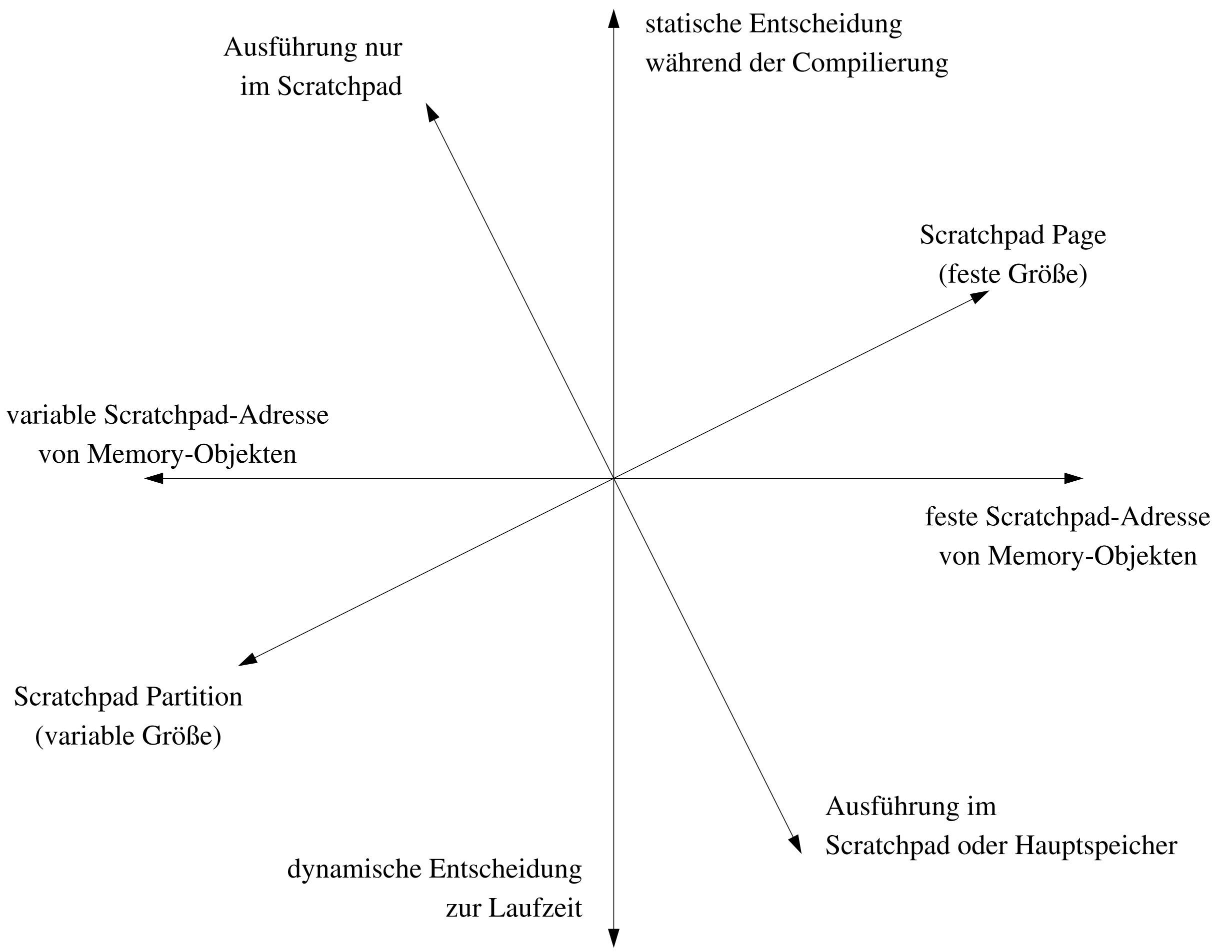


Abbildung 5.42: Suchraum für dynamisches Einlagern

1. der Zeitpunkt für die Scheduling-Entscheidung,
2. die Wahl fester oder wechselnder (=variabler) Adressen für die Memory-Objekte im Scratchpad,
3. die Aufteilung des Scratchpad-Speichers in variable Größen oder in feste Seitengrößen,
4. die Ausführung der Memory-Objekte nur im Scratchpad oder auch alternativ im Hauptspeicher.

Zeitpunkt der Scheduling-Entscheidung

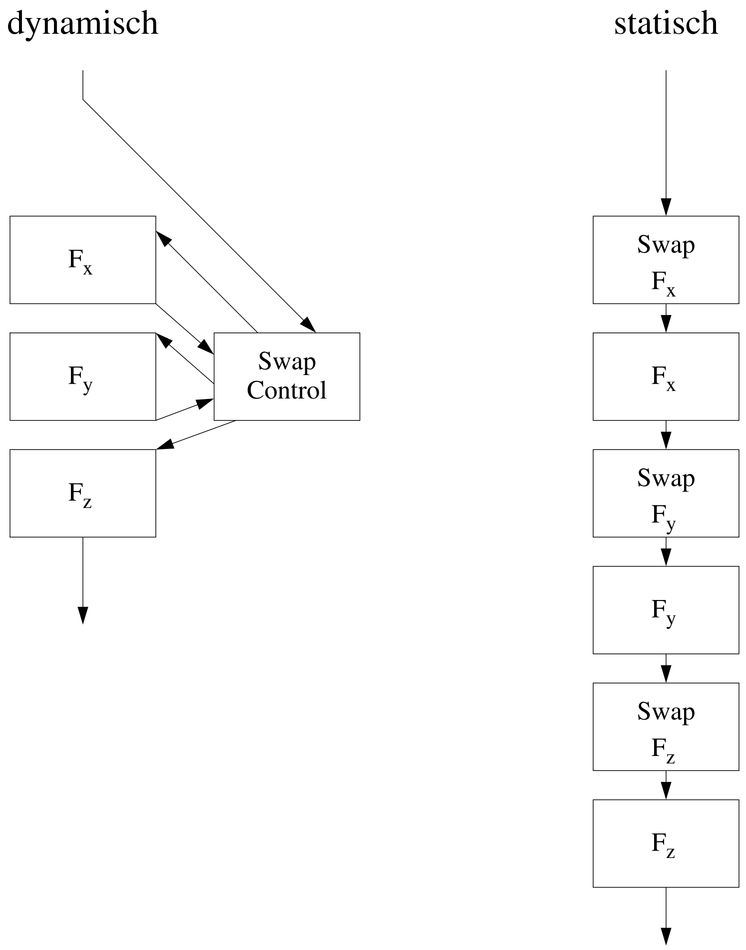


Abbildung 5.43: dynamische vs. statische Entscheidung

Welche Objekte zu welchem Zeitpunkt ein- oder ausgelagert werden, kann am besten während der Programmlaufzeit entschieden werden. Dann liegen die umfangreichsten und detailliertesten Informationen vor. Es kann beispielsweise vor der Ausführung eines Objektes entschieden werden, dass ein Verlagern lohnend ist und eventuell vor einer erneuten Ausführung des gleichen Objektes unter anderen Umständen, dass das Objekt im Hauptspeicher verbleibt. Dieser Ansatz der Entscheidung zur Laufzeit ist von Betriebssystemen und der Speicherverwaltung bekannt, wo ebenfalls während der Laufzeit über das Ein- und Auslagern entschieden werden muss. In Abbildung 5.43a ist ein Programm mit drei Funktionen F_x, F_y und F_z dargestellt, die hintereinander ausgeführt werden, und eine Einheit $Swap\,Control$, die jeweils die Entscheidung zu treffen hat, ob eine Funktion oder alternativ ein Basisblock oder Multibasisblock in den Scratchpad einzulagern ist. Das Kopieren der Funktion in den Scratchpad-Speicher wird auch durch die $Swap\,Control$ durchgeführt.

Die Alternative ist eine Entscheidung vorab zur Compile-Zeit. Da diese Entscheidung nur einmalig vor der Programmausführung getroffen wird (siehe Abb. 5.43b), ist der Zusatzaufwand während des Programmlaufs bedeutend geringer. Es verbleiben lediglich fest eingefügte Kopierfunktionen $Swap\,F_x$, $Swap\,F_y$ und $Swap\,F_z$, die das Ein- und Auslagern von Objekten übernehmen. Diese Kopierfunktionen sind fest im Kontrollfluss eingebaut. Falls nur ein Teil der Funktionen F_x, F_y und F_z eingelagert werden soll, können die anderen Kopierfunktionen vollständig entfallen. Dadurch fällt ein Overhead nur dann an, wenn eine Funktion auch tatsächlich kopiert wird.

Da die Laufzeit und der Energieverbrauch der betrachteten Objekte im Vergleich zu einer zentralen Einheit $SwapControl$ (siehe Abb. 5.43a), die für die Verwaltung aller Objekte zuständig ist, relativ klein sind, wird der Nutzen der dynamischen Entscheidung im Vergleich zur statischen Entscheidung viel geringer sein. Es wird sogar häufig der Fall sein, dass das ganze Verfahren nicht lohnend ist und es insgesamt zu Energieverlusten führt, da die $SwapControl$ regelmäßig ausgeführt werden muss, auch wenn letztendlich kein Objekt verschoben wird. Der Grenzwert, ob es überhaupt Konfigurationen gibt, bei denen sich der dynamische Ansatz als vorteilhaft erweist, wird von dem Verhältnis zwischen Zugriffskosten des Scratchpad bezogen auf die Kosten des Hauptspeichers, der Anzahl der Instruktionen, die in der $SwapControl$ durchschnittlich ausgeführt werden müssen, der Anzahl der Instruktionen in den kopierten Objekten sowie deren Ausführungshäufigkeit abhängen.

Für das betrachtete System wurde aus diesem Grunde die Entscheidung für das statische Scheduling-Verfahren getroffen, die damit Grundlage für die folgenden Ausführungen sein wird. Der zusätzliche Energieaufwand für die dynamische Scheduling-Entscheidung scheint hier nicht durch den Nutzen gerechtfertigt. Die detaillierte Kosten/Nutzenanalyse soll hier aber nicht weitergeführt werden.

Im Anschluss an die praktischen Ergebnisse kann nochmals geprüft werden, ob die erzielten Energiegewinne aufwendigere Entscheidungsverfahren zur Laufzeit ermöglichen.

Feste oder variable Adressen für Memory-Objekte im Scratchpad

```
// Programm
longJMP jtab+Off_M4
..
..
// Sprungtabelle
jtab+Off_M4:  longJMP  0x012308     /* aktuelle Adresse von M4 */
jtab+Off_M5:  longJMP  0x0ffe00     /* aktuelle Adresse von M5 */
jtab+Off_M6:  longJMP  0x158ef8     /* aktuelle Adresse von M6 */
```

Tabelle 5.4: Sprungtabelle

Der zweite Freiheitsgrad betrifft die Zuordnung von Adressen zu Memory-Objekten. Wenn ein Memory-Objekt mehrmals zu unterschiedlichen Zeitpunkten in den Scratchpad kopiert wird, kann dies so implementiert werden, dass dies jeweils immer zur gleichen Adresse im Scratchpad geschieht (dies würde bei einem vergleichbaren Cache der direct-mapped Organisation entsprechen) oder es kann, um den Speicher besser auszunutzen, die Adresse bei jedem Kopieren frei gewählt werden (vergleichbar einem full associative Cache). Da dann die Adresse eines Objektes im Scratchpad zum Zeitpunkt des Compilierens und Linkens nicht feststeht, muss eine Sprungtabelle eingeführt werden (siehe Tab. 5.4), die jeweils gepflegt wird, um stets die aktuelle Adresse der Objekte $M4$, $M5$ und $M6$ im Scratchpad zu enthalten. Falls beispielsweise das Objekt $M4$ in den Scratchpad verschoben wird, muss der Sprungtabelleneintrag `jtab+Off_M4` auf die neue Adresse von $M4$ angepasst werden. Der Overhead beim Ansprung eines Objektes im Scratchpad-Speicher beträgt im Vergleich zu der Variante mit festen Adressen genau einen zusätzlichen Sprungbefehl $longJMP$. Im Vergleich zur geringen Größe der Objekte und dem in unserem Beispielsystem vorhandenen Verhältnis zwischen Energiekosten für Scratchpad im Vergleich zum Hauptspeicher ist dieser zusätzliche Sprungbefehl sehr teuer. Weiterhin ist zu berücksichtigen, dass die Sprungtabelle bei jedem Kopieren eines Objektes in den Scratchpad-Speicher aktualisiert werden muss.

Als Ergebnis lässt sich feststellen, dass die festen Adressen mehr Energiegewinn versprechen, da der Overhead durch das Pflegen und Verwenden einer Sprungtabelle kaum durch den Nutzen durch bessere Speicherausnutzung aufgewogen werden wird. Für die weiteren Betrachtungen wird daher von der Wahl fester Adressen im Scratchpad für jedes Memory-Objekt ausgegangen.

Ausführung der Memory-Objekte nur im Scratchpad oder auch alternativ im Hauptspeicher

```
 void f(int result) {
  printf("Result:  %d\n",result);
};

int main(void) {
int a = 5; int i;

  f(a);
  ..
  for (i = 0; i < 10000; i++) {
    g(i);
  }
  ..
  for (i = 0; i < 100; i++) {
    f(i);
  }
};
```

Tabelle 5.5: Beispielprogramm mehrerer Funktionsaufrufe mit unterschiedlicher Häufigkeit

In diesem Freiheitsgrad unterscheiden sich die Möglichkeiten des Scratchpad von bekannten Verfahren zur Verwaltung von Speicherhierarchien. Bei einer Speicherhierarchie müssen grundsätzlich Programmteile, die ausgeführt werden sollen, in den Cache kopiert werden. Eine Ausführung beispielsweise direkt aus dem Hauptspeicher ist bei zwischengeschaltetem Cache im Allgemeinen nicht möglich. Bei dem hier betrachteten Einsatzgebiet kann aber eine Programmausführung im Hauptspeicher oder alternativ im Scratchpad erfolgen. Für die letztgenannte Möglichkeit ist es Voraussetzung, dass das Objekt noch im Scratchpad vorhanden ist oder gesondert dafür in diesen kopiert werden muss. Dieser Vorteil der Möglichkeit der Ausführung in beiden Speichern kann genutzt werden, wenn ein Objekt an einer Stelle im Programm nur wenige Male durchlaufen wird und sich deshalb ein Kopieren nicht lohnt, es an anderer Stelle im Kontrollfluss wieder häufig ausgeführt wird und sich dann ein Kopieren und Ausführen im Scratchpad als vorteilhaft erweist.

Die Frage ist nun, in welchen Fällen diese Situation auftreten kann, dass ein Programmteil in einem Abschnitt häufiger und in einem anderen Abschnitt weniger häufig ausgeführt wird. Im Wesentlichen trifft dies auf Funktionen zu. Funktionen sind neben der Möglichkeit der Strukturierung durch Kapselung dafür gedacht, Programmteile abzubilden, die an unterschiedlichen Stellen im Programm genutzt, d. h. aufgerufen werden können. Ein konkreter Fall (siehe Tab. 5.5) wäre also eine Funktion f, die an zwei unterschiedlichen Stellen im Programm aufgerufen wird, einmal mit wenigen Durchläufen (Anfang der `main`-Funktion), an anderer Stelle mit vielen Durchläufen (innerhalb der for-Schleife) und zwischen diesen beiden Aufrufen der Aufruf einer weiteren Funktion g, die die Funktion f aus dem Scratchpad verdrängen würde.

Ein Verfahren, welches sowohl die Möglichkeit der Ausführung im Scratchpad oder auch im Hauptspeicher erlaubt, ermöglicht daher höhere Energieeinsparungen, ohne dass, abgesehen vom höheren Aufwand bei der Implementierung des Verfahrens, Nachteile erkennbar sind. Wir werden daher im Folgenden davon ausgehen, dass Programmteile auch in beiden Speichern ausgeführt werden können und dies durch das Verfahren berücksichtigt wird.

Aufteilung des Scratchpad-Speichers in variable Größen oder in feste Seitengrößen

Der letzte Freiheitsgrad im Suchraum für ein geeignetes Verfahren zur dynamischen Ein- und Auslagerung von Programmteilen beschäftigt sich mit der Speicherverwaltung im Scratchpad. Wie aus der Speicher-

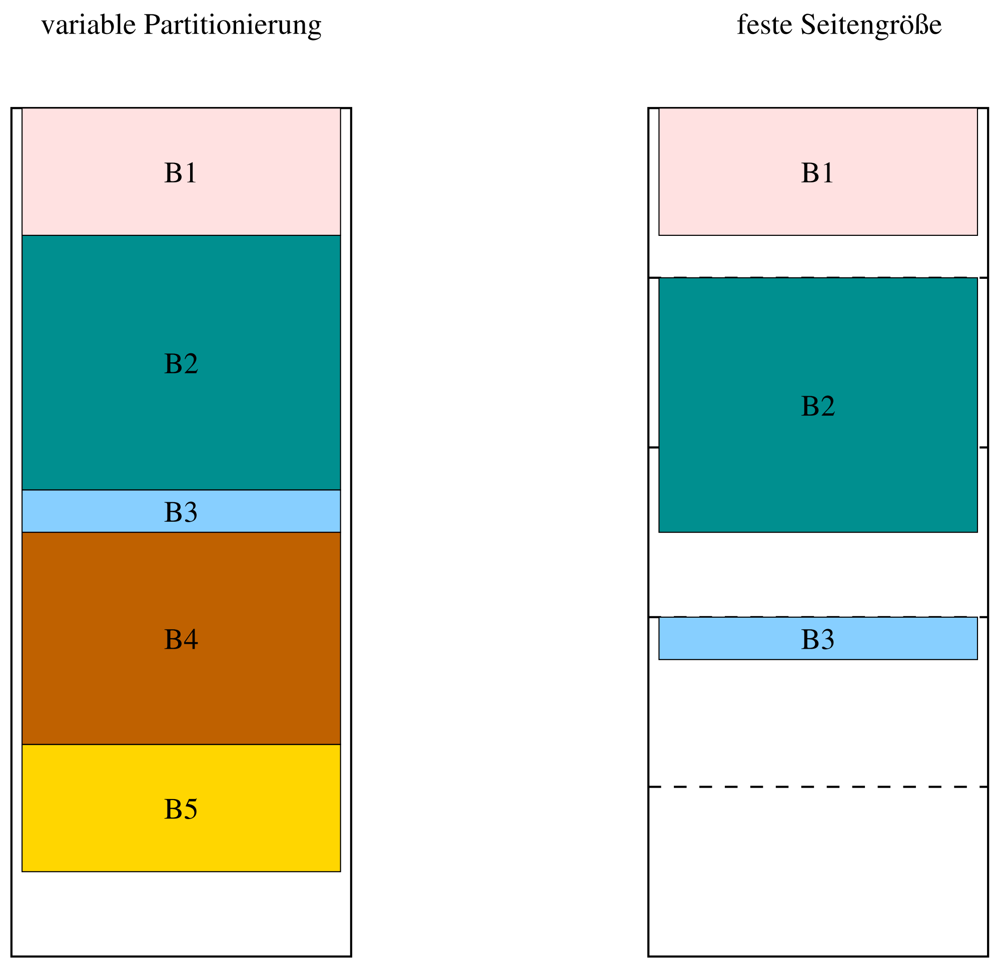


Abbildung 5.44: variable Partitionierung vs. feste Pagegrößen

verwaltung z. B. von Betriebssystemen bekannt ist, können entweder variable Partitionen oder eine feste Seitengröße gewählt werden (siehe Abb. 5.44). Bei variablen Partitionen kann mehr in das Scratchpad gepackt werden, da zwischen den einzelnen Objekten keine Speicherzellen ungenutzt gelassen werden. Dafür kann es aber zu einer externen Fragmentierung kommen.

Die andere Möglichkeit der festen Seitengröße kann potenziell zu interner Fragmentierung führen. Außerdem können weniger Objekte in den Scratchpad-Speicher kopiert werden. Der Vorteil ist, dass eventuell das Kopieren effizienter implementiert werden kann, da die Seitengröße fest ist.

Insgesamt ist für dieses Verfahren aufgrund der geringen Scratchpad-Größe der Ansatz von variablen Partitionen zu bevorzugen.

Zusammenfassend kann festgehalten werden, dass das Verfahren auf folgenden Eckpunkten basieren soll:

- eine statische Scheduling-Entscheidung über das Kopieren von Memory-Objekten in den Scratchpad,
- feste Adressen für ein einzelnes Objekt im Scratchpad,
- Ausführung eines Programmteils sowohl im Hauptspeicher als auch im Scratchpad,
- variable Partitionsgröße.

Es gilt im Folgenden also ein Verfahren zu entwickeln, welches unter Berücksichtigung der genannten Eckpunkte eine optimale Wahl von Memory-Objekten trifft und optimale Punkte im Kontrollfluss, wo diese Memory-Objekte in den Scratchpad kopiert werden sollen.

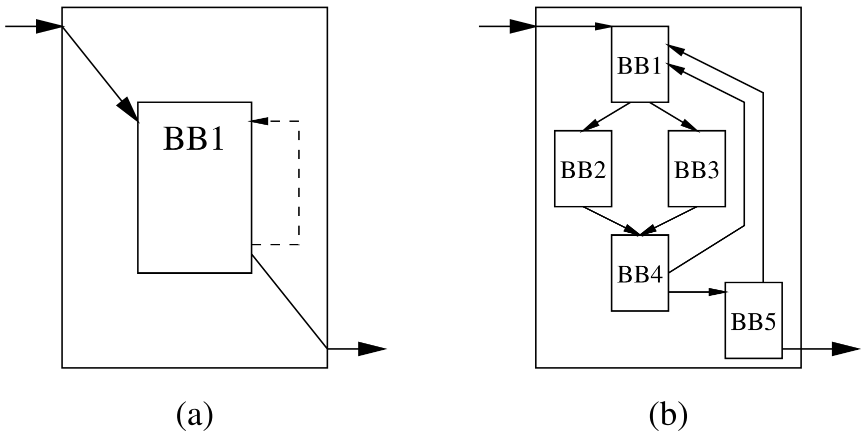


Abbildung 5.45: Superblöcke

5.5.2 Mathematisches Modell

Vorab eine Definition des Begriffes Superblock, der für das Verfahren für die Zerlegung des Kontrollflusses grundlegend ist.

Definition Superblock: *Ein Superblock besteht aus einem oder mehreren Basisblöcken und hat die Eigenschaft, dass*

1. *der Kontrollfluss von außerhalb stets an genau einer Stelle im Superblock beginnt,*

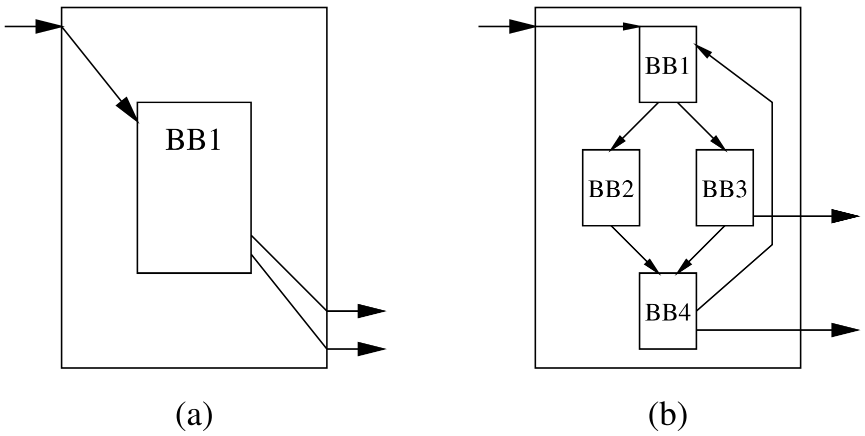


Abbildung 5.46: keine Superblöcke

2. *beim Verlassen des Superblocks nur genau eine nachfolgende Instruktion existiert,*
3. *Basisblöcke nur vollständig hinzugehören und keine einzelnen Instruktionen.*

Der kleinste Superblock besteht aus einem einzelnen Basisblock (Abb. 5.45a), der nur einen Austrittspunkt hat und deswegen die drei Bedingungen erfüllt. Dieser Basisblock kann auch eine Schleife darstellen, wobei der Rücksprung dann zum selben Basisblock erfolgen muss. Ein Beispiel für eine komplexere Struktur einer Schleife, die auch einen Superblock darstellt, zeigt Abbildung 5.45b, in der auch die beiden Bedingungen für den Eintritt und den Austritt erfüllt werden. Gegenbeispiele für Strukturen, die keinen Superblock darstellen, zeigt Abbildung 5.46, wo in Struktur (a) der Basisblock zu zwei unterschiedlichen Nachfolge-Instruktionen verzweigt. Gleiches gilt für die Struktur (b), wo von den Basisblöcken $BB3$ und $BB4$ der Superblock zu unterschiedlichen Basisblöcken verlassen wird.

```
// -- copy point KF 1 --
for (i = 0; i < 100; i++) {
  ..// Basic block 1

  // -- copy point KF 2 --
  for (j = 0; j < 20; j++) {
    ..// Hotspot 1
  }
  // -- copy point KF 3 --
  for (k = 0; k < 30; k++) {
    ..// Hotspot 2a
    if (..)  {
      ..// Hotspot 2b
    }
  }
}
```

Abbildung 5.47: Beispielprogramm für die dynamische Nutzung des Scratchpad

Durch eine Zerlegung in Superblöcke wird sichergestellt, dass es genau einen Eintrittspunkt und einen Austrittspunkt gibt, sodass die vollständige Kontrolle und Verfügbarkeit über den Scratchpad-Speicher zwischen diesen beiden Punkten vorhanden ist. Die Komplexität der Kontrollflussanalyse wird durch die oben gegebene Definition der Superblöcke reduziert.

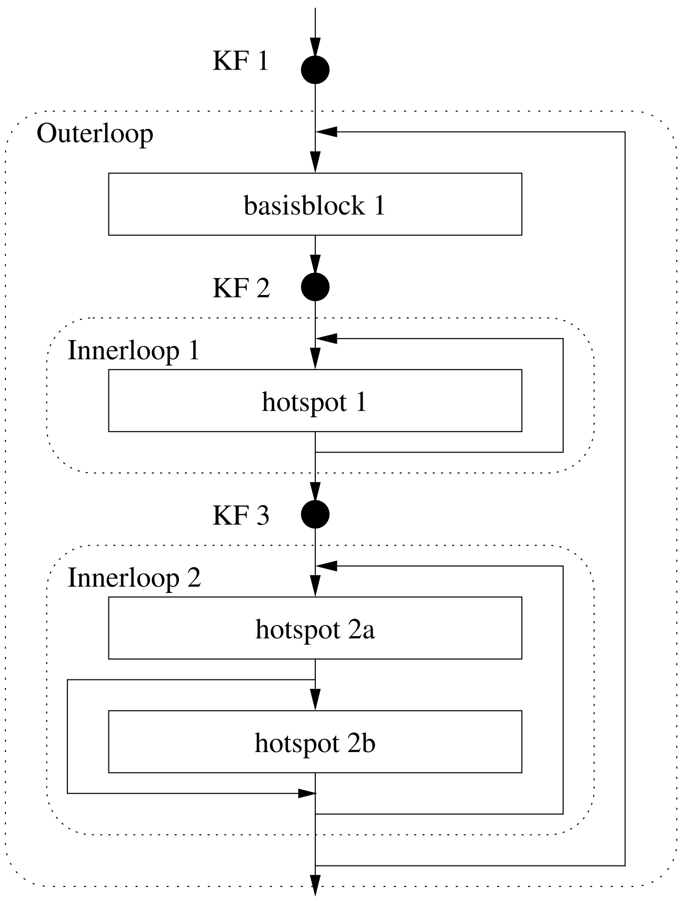


Abbildung 5.48: Kontrollfluss des Beispielprogramms

5.5.3 Wahl der Objekte und Clustern der Kopierfunktionen

Zum Einlagern von Programmteilen in das Scratchpad werden an verschiedenen Punkten im Programm Kopierfunktionen eingefügt. Es gilt nun, die optimalen Positionen dieser Kopierfunktionen und die jeweils optimal zu kopierenden Programmblöcke zu berechnen.

Ansatz für die Position der Kopierfunktion

1. Annahme: Ein Einlagern ist nur sinnvoll, wenn die eingelagerten Blöcke danach mehrfach ausgeführt werden

Begründung: Ein Einlagern bedeutet, dass die Befehle in der Kopierfunktion mit einem Load-Befehl in den Prozessor geladen und mit einem Store-Befehl im Scratchpad wieder gespeichert werden müssen. Eine einmalige Ausführung des Programms außerhalb des Scratchpad benötigt lediglich ein einmaliges Lesen in der Instructionfetch-Phase des Prozessors. Der Aufwand durch das Kopieren ins Scratchpad lohnt sich daher nur, wenn die verschobenen Programmteile häufiger als die Kopierfunktion aufgerufen werden.

2. Annahme: Ohne Einschränkung der Optimalität genügt genau eine Kopierfunktion an jedem Schleifenanfang

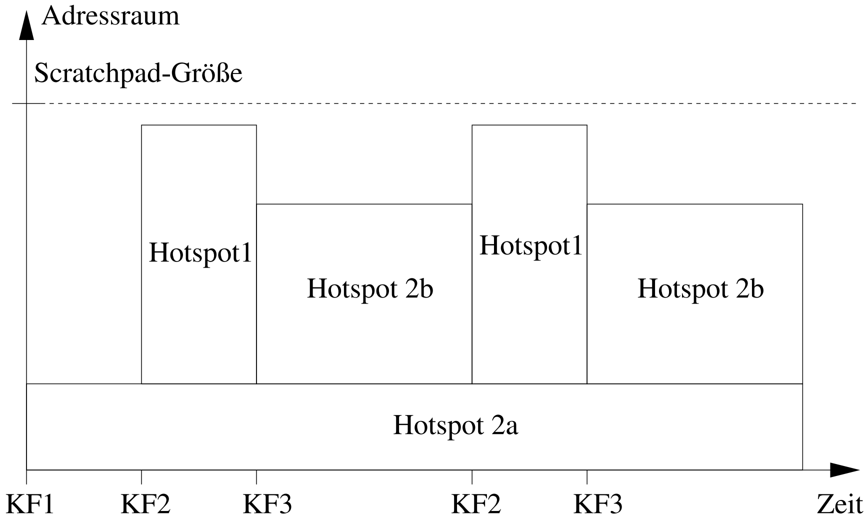


Abbildung 5.49: Speicherbelegung

Begründung: Da nur in Schleifen höhere Ausführungshäufigkeiten als bei vorangehenden Basisblöcken entstehen, kann die Betrachtung dieser Optimierung auf diese beschränkt werden. Ohne weitere Einschränkung kann dann die Kopierfunktion direkt an den Schleifenanfang positioniert werden.

Insgesamt ist es für die optimale Anzahl und Position von Kopierfunktionen ausreichend, genau eine Kopierfunktion an jedem Schleifenanfang zu generieren. Damit ist jedoch noch nicht eindeutig festgelegt, welcher Basisblock welcher Kopierfunktion zugeordnet wird.

Wir betrachten das Programmbeispiel in Abbildung 5.47, in dem insgesamt drei Schleifen existieren. Weiterhin findet in der zweiten inneren Schleife eine Kontrollflussverzweigung durch den if-Befehl statt. Der Kontrollfluss zu diesem Beispiel ist in Abbildung 5.48 dargestellt, wobei die erste Schleife dem Superblock "Outerloop" entspricht, da es genau einen Eintrittspunkt gibt und auch die Programmkontrolle nach Verlassen der Schleife an der ersten nachfolgenden Instruktion fortsetzt. Die zweite Schleife liegt eine Ebene tiefer und wird im Kontrollfluss mit "Innerloop 1" bezeichnet. Auch sie erfüllt die Kriterien eines Superblockes, da der Rücksprung wieder direkt auf denselben Basisblock erfolgt. Der vorangestellte

"Basisblock 1" wird durch einige zusätzlich nicht im Detail dargestellte Instruktionen zusammen mit der Schleifeninitialisierung für die "Innerloop 1" gebildet.

Die dritte und letzte Schleife "Innerloop 2" bildet gleichzeitig einen weiteren Superblock, der die beiden Hotspots "Hotspot 2a" und "Hotspot 2b" umfasst. Diese beiden Hotspots werden zur Vereinfachung des Beispiels selbst auch wiederum als Superblock betrachtet, obwohl sie Kriterium 2 der Superblock-Definition nicht erfüllen.

Die Kontrollflussanalyse liefert insgesamt drei Schleifen. Für diese sind drei Kopierfunktionen jeweils zu Beginn der Schleifen ausreichend, die in der Abbildung 5.48 mit *KF1*, *KF2* und *KF3* dargestellt werden. Zur Vereinfachung wird der Basisblock zur Initialisierung der dritten Schleife nicht dargestellt.

Mit der Definition der Position der Kopierfunktion ist aber noch nicht festgelegt, ob beispielsweise der "hotspot 1" der Kopierfunktion *KF1* oder *KF2* zugeordnet wird. Bei hierarchischen Schleifenkonstrukten ergeben sich entsprechende Freiheitsgrade. Es liegt hier ein Tradeoff vor zwischen der Ausführungshäufigkeit und der zeitlichen Dauer der Speicherbelegung. Wenn "Hotspot 1" durch *KF1* kopiert wird, entsteht weniger Energieaufwand für das Kopieren, da *KF1* seltener ausgeführt wird als *KF2*. Andererseits belegt "Hotspot 1" einen Teil des Scratchpad-Speichers während der vollständigen Schleifenausführung von "Outerloop". Im Gegensatz dazu kann bei der Zuordnung von "Hotspot 1" zu *KF2* der Speicher nach der "Innerloop 1" wieder anderweitig, z. B. für "Hotspot2a" oder "Hotspot2b", genutzt werden.

Die Entscheidung über die beste Zuordnung von Hotspots zu Kopierfunktionen stellt daher einen Teil des Optimierungsproblems dar.

Ein Beispiel für eine mögliche Belegung des Speichers für das betrachtete Programmbeispiel wird in Abbildung 5.49 gezeigt. Auf der Zeitachse sind die Zeitpunkte für das Ausführen der einzelnen Kopierfunktionen *KF1*, *KF2* und *KF3* aufgetragen und auf der Y-Achse die Belegung des Adressraums des Scratchpad-Speichers. Begrenzt ist der Adressraum aufgrund der Größe des Scratchpad-Speichers. Eine mögliche Belegung des Speichers ergibt sich durch das Kopieren des Basisblocks "Hotspot 2a" durch die Kopierfunktion *KF1*. Bei Erreichen der Kopierfunktion *KF2* wird der "Hotspot 1" in den Scratchpad kopiert, was im Gegensatz zu *KF1* mehrfach geschieht, aber die Nutzung des zugehörigen Speicherplatzes zwischendurch durch den "Hotspot 2b" zulässt, der durch die Kopierfunktion *KF3* kopiert wird.

Nach der Berechnung der Hotspots und der zugehörigen Kopierfunktionen sollen im nächsten Schritt nun die Energiekosten für das dynamische Verfahren betrachtet werden.

5.5.4 Energiebetrachtung

Bei der Energieberechnung betrachten wir als erste Funktion die Kopierkosten E_{single_cp} für das Kopieren einer einzelnen Instruktion aus dem Hauptspeicher in den Scratchpad-Speicher. Für das Kopieren während der Programmlaufzeit des Memory-Objektes mo mit k Instruktionen durch die Kopierfunktion kf_{su} mit der Ausführungshäufigkeit $n(kf_{su})$ ergibt sich daraus:

$$E_{cp}(mo) = n(kf_{su}) * E_{single\_cp} * k$$

Diese Kopierkosten $E_{cp}(mo)$ müssen im Vergleich zum statischen Verfahren zusätzlich vom Energiegewinn subtrahiert werden und reduzieren den Energievorteil durch die Einsparungen bei den Instructionfetches E_{if}. Daraus ergibt sich für das Kopieren einer Funktion f_i mit der Ausführungshäufigkeit n_k der Instruktion k der folgende Energiegewinn $E(f_i)$:

$$E(f_i) = \sum_{\forall k \in f_i} n_k * E_{if} - E_{cp}(f_i)$$

Im Gegensatz zu den Funktionen wird bei einem Basisblock jede der k Instruktionen gleich häufig, nämlich n-mal, ausgeführt. Weiterhin kann es notwendig sein, ähnlich wie beim statischen Verfahren, zusätzliche Sprungbefehle anzufügen, die dann l-mal jeweils mit Energiekosten von E_{jmp} ausgeführt werden. Es ergibt sich dann ein Energiegewinn $E(bb_j)$ für einen verschobenen Basisblock bb_j:

$$E(bb_j) = k * n * E_{if} - l * E_{jmp} - E_{cp}(bb_j)$$

In Abhängigkeit der Kopierkosten eines Befehls E_{single_cp}, der Einsparung bei einem einzelnen Instructionfetch E_{if} und den Ausführungshäufigkeiten der Kopierfunktion kf_{su} und einem Memory-Objekt mo kann abgeschätzt werden, ob eine Verschiebung grundsätzlich Gewinn bringen kann oder nicht.

Für einen möglichen Gewinn muss folgende Ungleichung erfüllt sein:

$$\frac{n(mo)}{n(kf_{su})} \geq \frac{E_{single\_cp}}{E_{if}}$$

Für einen Basisblock sind in dieser Ungleichung noch nicht die zusätzlichen Sprünge berücksichtigt, sodass es sich hierbei nur um eine notwendige jedoch nicht hinreichende Bedingung handelt. Bei Funktionen ist der Wert $n(mo)$ ein Durchschnittswert über die Ausführungshäufigkeit aller enthaltenen Instruktionen. Vorrangig kann diese Bedingung dazu verwendet werden, um schon frühzeitig Memory-Objekte auszuschließen, die keinen Gewinn bringen können.

Im nächsten Schritt wird ein ILP-Modell zur Spezifikation des Problems entwickelt.

5.5.5 ILP-Modell

Zu Beginn werden die notwendigen Variablen für das Modell präsentiert. Zuerst wird ein *dynamisches Programm-Objekt po* definiert, welches ein Tupel aus einem Memory-Objekt $mo \in MO$ und einem Superblock $su \in SU$ darstellt, zu dem das Memory-Objekt gehört:

$$po := (mo, su)$$

Die *Menge aller dynamischen Programm-Objekte* PO ist dann wie folgt für ein Programm P bestehend aus einer Menge von Basisblöcken BB, einer Menge von Funktionen F und der Menge von Superblöcken SU definiert:

$$PO := \{(mo, su) | mo \in MO \wedge su \in SU \wedge mo \subseteq su\}$$

Die Menge von dynamischen Programm-Objekten für einen einzelnen Superblock su ist definiert als:

$$PO_{su} := \{po \in PO | \exists mo : po = (mo, su)\}$$

Mit der Funktion $m(po)$ wird definiert, ob ein dynamisches Programm-Objekt po ausgewählt wurde:

$$m(po) := \begin{cases} 1, & \text{wenn } po \text{ in den Scratchpad verschoben wird} \\ 0, & \text{sonst} \end{cases}$$

Die zu maximierende Zielfunktion für den Energiegewinn sav des ILP-Modells lautet nun:

$$sav = \sum_{po \in PO} E(po) * m(po)$$

Die Menge der während der Ausführung eines Superblockes im Scratchpad liegenden Blöcke darf die Scratchpad-Größe $size$ nicht überschreiten. Die Menge der innersten Superblöcke, die auf der untersten Hierarchieebene stehen, ist wie folgt definiert:

$$SU_{bottom} := \{su \in SU | \{y \in SU | y \subset su\} = \emptyset\}$$

Daher ist für jeden Superblock $su \in SU_{bottom}$ der untersten Ebene genau ein Constraint zu generieren, der sicherstellt, dass zu jedem Zeitpunkt der Speicherplatz $size$ eingehalten wird. Constraints für Superblöcke der höheren Hierarchiestufen können entfallen, da sie Bestandteil der Constraints der unteren Hierarchiestufen sind.

$$\forall su \in SU_{bottom} : \sum_{y \supseteq su} \sum_{po \in PO_y} S(po) * m(po) \leq size$$

Weiterhin ist sicherzustellen, dass jeder Basisblock bb nur von einer Kopierfunktion kopiert wird:

$$\forall bb : \sum_{po \in \{PO_{su} | bb \in su \wedge (bb, su) \in PO_{su}\}} m(po) \leq 1$$

Hiermit wurde ein vollständiges ILP-Modell aufgestellt, welches die optimale Lösung zum dynamischen Einlagern von Programmteilen in einen Scratchpad ermittelt.

Die Anzahl der Variablen in diesem ILP-System entspricht den Programm-Objekten po. Die Anzahl der Objekte entspricht den aus dem statischen Ansatz bekannten Programm-Objekten der Basisblöcke, Multibasisblöcke und Funktionen, jeweils multipliziert mit der Anzahl der Superblöcke, zu denen diese Objekte gehören. Da nur Objekte relevant sind, die Bestandteil von Schleifen bzw. Superblöcken sind, können andere Objekte von vornherein ausgeschlossen werden, um den Aufwand und die Komplexität zu reduzieren.

5.5.6 Ergebnisse

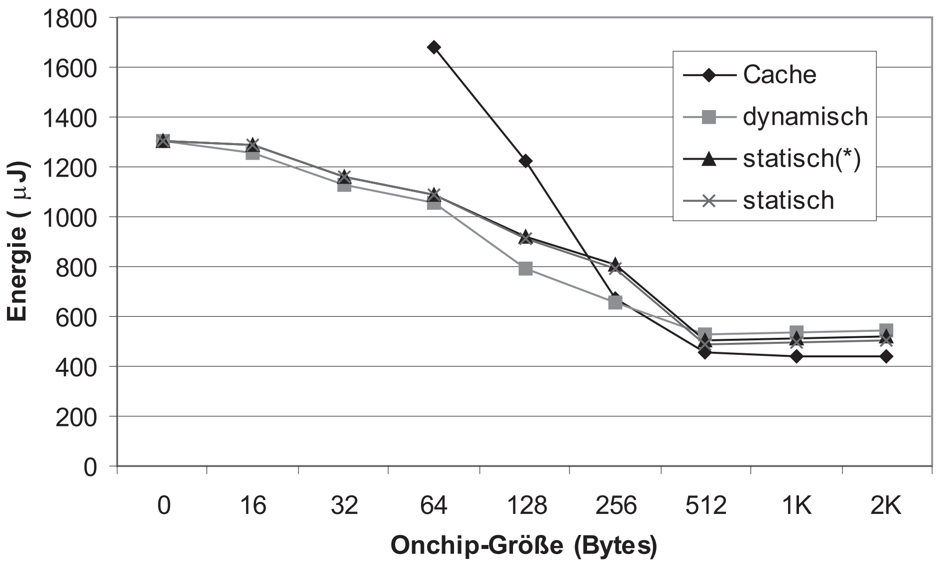


Abbildung 5.50: *heapsort* Benchmark mit dynamischem Einlagern

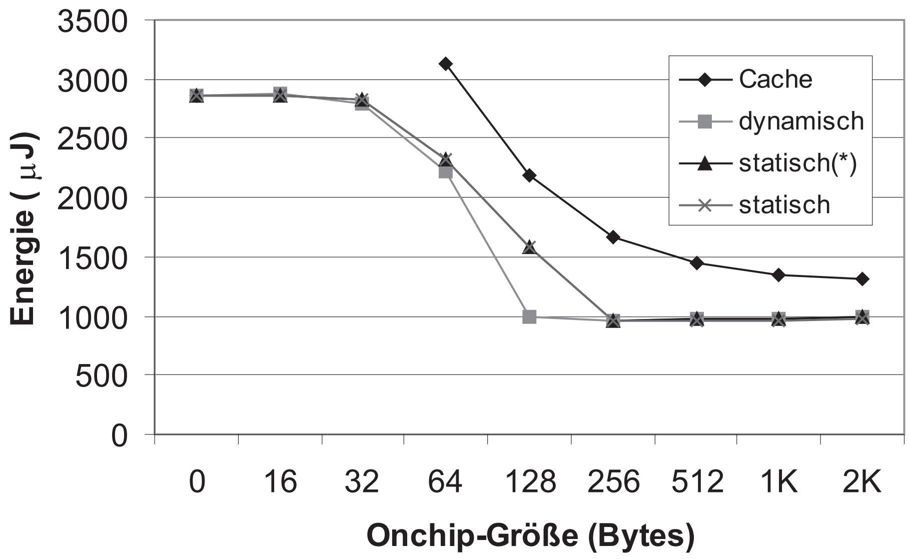


Abbildung 5.51: *lattice* Benchmark mit dynamischem Einlagern

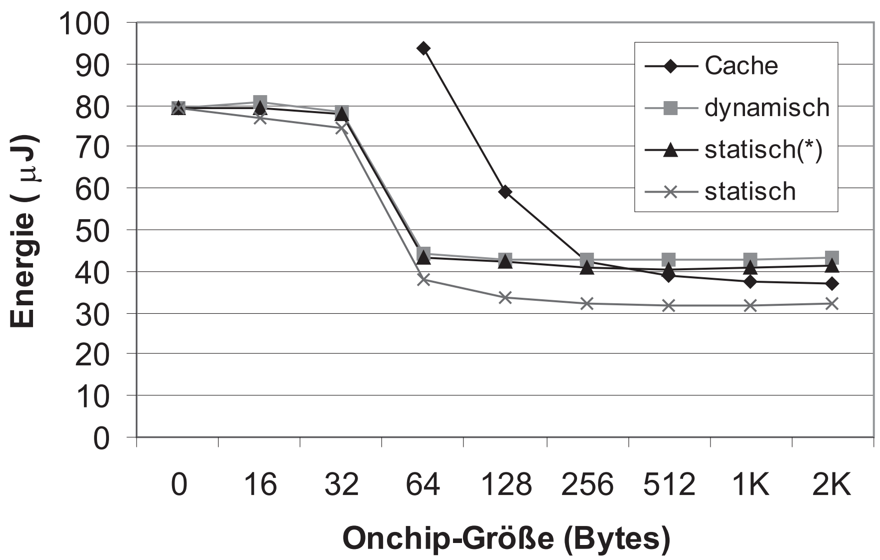


Abbildung 5.52: *matrixmult* Benchmark mit dynamischem Einlagern

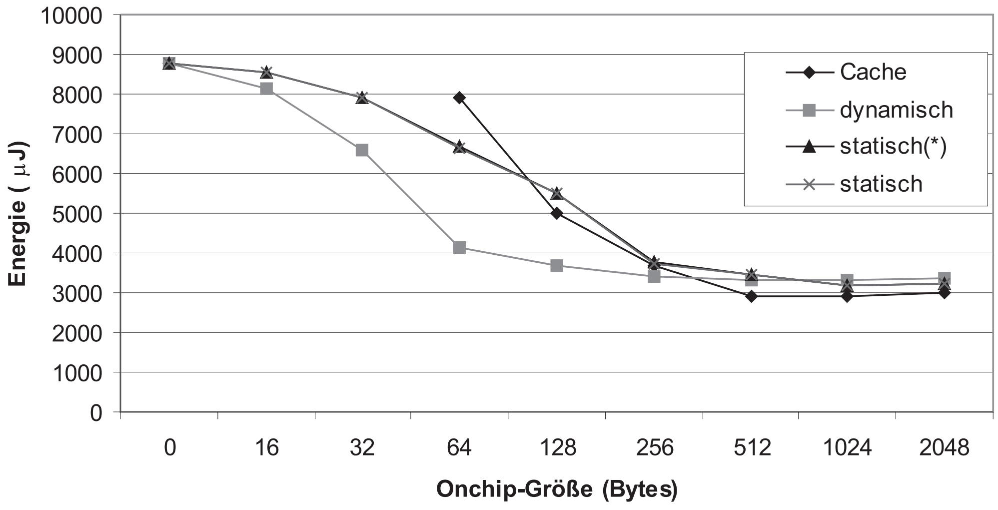


Abbildung 5.53: *multisort* Benchmark mit dynamischem Einlagern

Dieses ILP-Verfahren wurde mit verschiedenen Benchmarks untersucht, wobei die Ergebnisse der Benchmarks *heapsort*, *lattice* und *matrixmult* hier beispielhaft dargestellt werden. Ein zusätzlicher konstruierter Benchmark *multisort* aus der Konkatenation von mehreren Sortierverfahren soll insbesondere das Verhalten bei mehreren Hotspots zeigen, die nicht zusammen in den Scratchpad passen und nacheinander bzw. wechselweise ausgeführt werden.

In den Abbildungen 5.50 bis 5.53 wurden die Ergebnisse für unterschiedliche Compilerläufe dargestellt. Neben dem Cache als Vergleichsmaßstab wurde auch das statische Verfahren in zwei Varianten gemessen. Da bei Benchmarks mit geringer Programmlaufzeit auch das einmalige Kopieren in den Scratchpad bei der dynamischen Variante einen hohen Anteil hat und dies bei der statischen Variante nicht gezählt wurde, da es Bestandteil des Loader-Mechanismus ist, wurde das statische Verfahren einmal ohne Berücksichtigung der durch den Loader geladenen Scratchpad-Anteile als "statisch" und einmal mit Berücksichtigung des Ladens des Scratchpads als "statisch*" visualisiert.

Der erste Benchmark *heapsort* in Abbildung 5.50 zeigt, dass die beiden Kurven "statisch" und "statisch*" nahe beieinander liegen. Das bedeutet, dass der Anteil des einmaligen Ladens in den Scratchpad relativ geringen Anteil hat und hier nicht weiter relevant ist. Das dynamische Verfahren "dynamisch" liegt für die relevanten Scratchpad-Größen bis 256 Bytes unterhalb des statischen Verfahrens und verbraucht somit weniger Energie. Bei einer Programmgröße von 332 Bytes stellt der Bereich oberhalb von 256 Bytes kein realistisches Verhältnis mehr dar. Das dynamische Verfahren kann keinen Vorteil mehr bringen, da das vollständige Programm in den Scratchpad passt und durch den Overhead des Verfahrens selbst muss es für solche Situationen immer schlechter als die statische Variante sein. Der letzte Vergleich betrifft den Cache, für den Werte aufgrund des verwendeten Tools erst für Speichergrößen ab 64 Bytes vorliegen. Im interessanten Speicherbereich bei 64 und 128 Bytes ist das dynamische Verfahren eindeutig besser. Selbst beim eher unrealistischen Verhältnis bei 256 Bytes, wo das Scratchpad 77% des Programms aufnehmen kann, sind Cache und das dynamische Verfahren gleichauf, während das statische Verfahren schlechter abschneidet.

Der nächste betrachtete Benchmark *lattice* in Abbildung 5.51 zeigt ebenfalls keinen nennenswerten Einfluss durch den Lade-Vorgang beim statischen Verfahren. Der Vergleich zwischen dynamischem und statischem Verfahren zeigt Vorteile insbesondere im relevanten Speicherbereich für das dynamische Verfahren. Ebenfalls eindeutig ist der Vergleich mit dem Cache, der für alle Speichergrößen eindeutig schlechtere Werte liefert.

Als letzter kleinerer Benchmark wird *matrixmult* in Abbildung 5.52 dargestellt. Hier ist ein Unterschied beim statischen Verfahren durch das Laden zu verzeichnen, sodass man eine geringe Programmlaufzeit feststellen kann. Das dynamische Verfahren liegt von der Größenordnung her sehr nah beim statischen Verfahren inklusive dem Energieaufwand für das Laden.

Als Letztes ist in Abbildung 5.53 im *multisort* Benchmark ein Fall konstruiert, in dem die speziellen Eigenschaften des dynamischen Verfahrens besonders gegenüber der statischen Variante im Vorteil sind. Aufgrund der großen Laufzeit ist der Einfluss durch das einmalige Laden beim statischen Verfahren vernachlässigbar. Das dynamische Austauschen von Blöcken ist im relevanten Speichergrößenbereich viel günstiger. Im Vergleich zum Cache ist das dynamische Austauschen bis zu einer Größe von einschließlich 256 Bytes besser. Eine Speichergröße über 256 Bytes ist bei einer Programmgröße von 712 Bytes ein in der Praxis unwahrscheinliches Verhältnis des Onchip-Speichers im Vergleich zur Gesamtprogrammgröße.

Benchmark	Onchip (Bytes)	Cache (μJ)	Scratchpad (μJ)	Einsparung (%)
heapsort	64	1.677	1.053	37
heapsort	128	1.223	786	36
lattice	64	3.129	2.210	29
lattice	128	2.188	983	55
matrixmult	64	93,6	43,8	53
matrixmult	128	59,2	42,1	29
multisort	64	9.589,9	4.154,6	57
multisort	128	6.704,8	3.675,3	45
multisort	256	5.093,6	3.416,0	33
Durchschnitt				42

Tabelle 5.6: Verbesserung des Energieverbrauchs durch dynamisches Kopieren

Zusammenfassend werden die Werte für verschiedene realistische Speichergrößen und die Energieeinsparung im Vergleich zum Cache in Tabelle 5.6 dargestellt. Es ergibt sich eine durchschnittliche Einsparung von 42% des Energieverbrauchs im Vergleich zu einem verwendeten Cache.

Erwähnt werden muss allerdings auch der Nachteil des dynamischen Verfahrens, welches die Programmgröße aufgrund der eingebauten Kopierfunktionen stark erhöht. Die Scratchpad-Größe ist begrenzt, aber der Teil im Hauptspeicher, zu dem auch die Kopierfunktionen gehören, wächst beträchtlich. Für die in Tabelle 5.6 betrachteten Kombinationen ergibt sich ein durchschnittliches Anwachsen der Programmgröße um 150%.

Auch wenn der Schwerpunkt des vorgestellten Verfahrens auf die Energieverbrauchsreduzierung abzielt, hat die Methode auch Einfluss auf die Performance eines Programms. Analog zu den Energiewerten der

Benchmark	Onchip (Bytes)	Scratchpad (Zyklen)	Cache (Zyklen)	Einsparung (%)
heapsort	64	86.227	135.134	36
heapsort	128	72.794	106.287	32
lattice	64	164.432	264.282	38
lattice	128	120.594	203.516	41
matrixmult	64	4.180	7.507	44
matrixmult	128	4.057	5.337	24
multisort	64	484.988	861.178	44
multisort	128	456.835	663.168	31
multisort	256	435.372	552.657	21
Durchschnitt				35

Tabelle 5.7: Verbesserung der Performance durch dynamisches Kopieren

Tabelle 5.6 werden die Performance-Werte in Tabelle 5.7 dargestellt. Es zeigt sich, dass auch die Ausführungszeit nennenswert um durchschnittlich 35% gesenkt werden kann. Dies ist insbesondere beachtlich, da sich die Programmgröße sehr erhöht hat. Weitere Benchmarks und ausführlichere Ergebnisse finden sich bei Grunwald [Gru02] und wurden auch durch Steinke et al. [SGW+02] veröffentlicht.

Nach der Analyse der Ergebnisse wird deutlich, dass sowohl die statische als auch die dynamische Verschiebung ihre Vorteile besitzen. Das statische Verfahren behandelt Daten wie globale Daten und den Stack, die nur dauerhaft verschoben werden können, hat aber Nachteile bei mehreren Hotspots. Im Gegensatz dazu kann das dynamische Verfahren die Programmteile während der Laufzeit verschieben, aber wiederum keine Daten behandeln. Die Frage ist, ob es die Möglichkeit der Kombination der beiden Verfahren gibt, wobei sich natürlich die Vorteile kombinieren sollten und das neue Verfahren immer die Werte erzielt wie das bessere der beiden bisherigen Verfahren.

Diese Kombination ist durch eine Erweiterung des dynamischen Verfahrens einfach möglich. Man führt einen zusätzlichen Superblock ein, der das vollständige Programm umfasst und setzt die Kopierkosten für die Kopierfunktion dieses Superblockes auf 0. Dadurch entspricht es dem statischen Verfahren, in dem nur einmal die dieser Kopierfunktion zugeordneten Programmteile während der Programmausführung kopiert werden. Da das Laden dieser Programmteile nur einmal durch den Loader vorgenommen wird, sind diese Kopierkosten auch mit 0 anzusetzen. Diese Erweiterung würde die Vorteile der beiden Verfahren in geeigneter Weise verknüpfen. Der einzige Nachteil entsteht durch die aufwendiger gewordene Auswahl der Objekte mit dem ILP-Verfahren und damit einer erhöhten Laufzeit des ILP-Solvers.

Fazit

Insgesamt kann festgestellt werden, dass sowohl die statische als auch die dynamische Verschiebung erhebliche Energiereduzierungen bewirken. Der Ansatz durch die Reduzierung der Hauptspeicherzugriffe hat selbst gegenüber bewährten Methoden wie dem Einsatz von Caches nennenswerte weitere Einsparungen erreicht. Die Erweiterung durch das dynamische Verfahren hat insbesondere die Energiewerte für Applikationen mit mehreren um das Scratchpad konkurrierenden Hotspots verbessern können.

Kapitel 6

Codierung

Die betrachteten Optimierungen haben unterschiedlichen Einfluss auf die Optimierungsziele Performance, Programmgröße und Energieverbrauch. Zu den Optimierungen für den Energieverbrauch ohne Einfluss auf die Performance gehört die Optimierung der über Busse transportierten Daten, die in diesem Kapitel intensiver betrachtet werden soll.

6.1 Überblick und Klassifizierung

Der Einfluss der Zustandswechsel auf den Busleitungen erfordert, wie bei der Entwicklung des Energiemodells gezeigt, aufgrund der hohen Kapazitäten der Busleitungen und der angeschlossenen Eingänge höhere Ströme. Es werden in diesem Kapitel diese Signalwechsel auf den Bussen durch eine optimierte Codierung verringert. Dies hat bei den vorgestellten Optimierungen keinen Einfluss auf die Geschwindigkeit oder Codegröße.

Wir betrachten eine Harvard-Architektur mit einem ARM-Prozessor und getrennten Speichern für Programm und Daten. Für die Codierung sind nun die Daten und ihre Optimierung auf den verschiedenen internen und externen Bussen relevant. Eine Optimierung z. B. der Codierung der Daten im Speicher (siehe auch [vdW]) wird hier nicht weiter betrachtet, da dies nicht Gegenstand dieser Arbeit ist. Auch das Energiemodell berücksichtigt diesen Effekt nicht.

Im Blockschaltbild in Abbildung 6.1, welches bereits in Kapitel 3 präsentiert wurde, sind die verschiedenen Busse nochmals dargestellt. Nachfolgend werden sie in der Reihenfolge ihrer Nummerierung einzeln betrachtet, die Einflussmöglichkeiten durch einen Compiler untersucht und das Optimierungspotenzial bewertet.

1. Adressbus für Datenspeicher

   Der Adressbus zur Adressierung des Datenspeichers (*DAddr*) wird bei den Datenzugriffen durch Load-, Store-, Push- und Pop-Befehle verwendet. Es muss zwischen dem Einfluss durch die *Ones*- und die Hamming-Funktion unterschieden werden. Eine Optimierung bzgl. der *Ones*-Funktion kann ohne Betrachtung des vorhergehenden oder nachfolgenden Zugriffs die Zahl der *Ones* in Abhängigkeit von den betrachteten Systemeigenschaften minimieren oder maximieren. Dies kann durch eine veränderte Zuordnung von Adressen zu Daten mit einer höheren oder niedrigeren Anzahl von *Ones* in der jeweiligen Adresse geschehen. Im Gegensatz dazu müssen bei der Hamming-Distanz die vorhergehenden und nachfolgenden Zugriffe berücksichtigt werden. Eine Verringerung kann durch eine Umsortierung oder ebenfalls geänderte Zuweisung von Adressen zu einzelnen Daten durchgeführt werden.

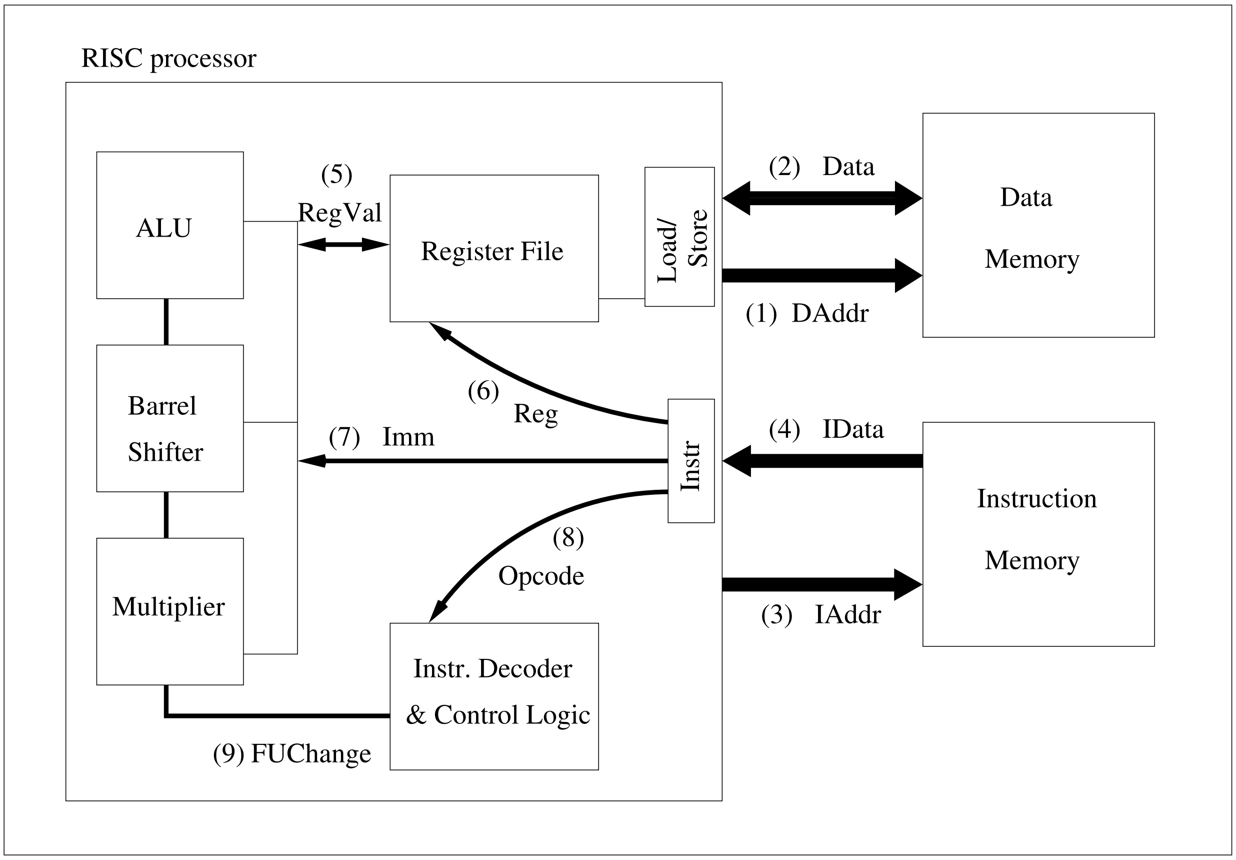


Abbildung 6.1: Blockschaltbild mit Bussen

Beides bedeutet eine detaillierte Analyse der Datenzugriffe mit Auswertung der Häufigkeit und für die Hamming-Distanz auch die Auswertung der Reihenfolge der Zugriffe. Aufgrund dieser Analyse kann das Memorylayout der Daten im Speicher optimiert werden. Einschränkend ist zu berücksichtigen, wie die Adressen generiert werden und dass unter Umständen zusätzliche Befehle für die Generierung oder das Laden größerer oder weiter entfernt liegender Adressen notwendig sind.

Für die Berechnung des Einsparungspotenzials werden folgende Annahmen auf empirischer Basis getroffen:

- jeder fünfte Befehl führt einen Zugriff auf den Datenspeicher aus
- durchschnittlich entstehen 25% der maximalen *Ones*-Kosten und Hamming-Kosten bei einem Datenspeicherzugriff
- maximal 25% der *Ones*- und Hamming-Kosten können eingespart werden

Daraus ergibt sich folgendes maximales Einsparungspotenzial bei einem maximalen Gesamtanteil am Energieverbrauch der *Ones*-Kosten von 1% sowie einem maximalen Anteil der Hamming-Kosten von 12% (siehe Messergebnisse in Kapitel 3):

$$\frac{1}{5} * 25\% * 25\% * (1\% + 12\%) = 0{,}1625\%$$

2. Datenbus für Datenspeicher

   Analog zur Optimierung des Adressbusses werden hier die gelesenen und geschriebenen Daten auf dem Datenbus des Datenspeichers (*Data*) betrachtet. Es muss unterschieden werden zwischen variablen Daten und konstanten Daten. Erstere werden dynamisch während des Programmlaufs verändert und ihre Werte können nur sehr schwer - wenn überhaupt - durch den Compiler vorher berechnet werden.

   Bei konstanten Daten ist die Analyse einfacher. Der Compiler hat hier die Kenntnis des konstanten Wertes. Eine Optimierung kann auf zwei verschiedene Arten erfolgen:

   (a) Modifikation von nicht relevanten Bits
   Wenn in dem konstanten Wert Bits enthalten sind, die für den späteren Programmlauf nicht relevant sind (*don't care*), kann entsprechend der *Ones*-Funktion und der Hamming-Distanz in Relation zu dem vorhergehenden und dem nachfolgenden Datum auf dem Datenbus das optimale Muster für die *don't care*-Bits ermittelt werden. Dies könnte beispielsweise beim Laden einer 32-Bit-Konstante auftreten, von der nur ein Byte genutzt wird. Eine in dieser Arbeit nicht weiter verfolgte Erweiterung ist die Zulassung einer Fehlerrate, die es ermöglicht, zusätzliche Bitwechsel zu unterdrücken, obwohl dadurch die Daten innerhalb einer Toleranz verfälscht werden [MF99].

   (b) Auswahl des optimalen Wertes aus einer Menge möglicher Werte
   Der andere seltene Optimierungsfall wäre die Wahl einer Konstanten aus einer Menge möglicher Muster. Es müsste eine Bewertung aller Muster auf der Basis der *Ones*- und der Hamming-Funktion vorgenommen und das beste Muster ermittelt werden. Beispiele von Mengen äquivalenter Daten sind applikationsabhängig.

   Von folgenden Annahmen wird bei der Einschätzung des Einsparungspotenzials aufgrund empirischer Untersuchungen ausgegangen:

   - jeder dritte Befehl enthält eine Konstante
   - jede Konstante enthält durchschnittlich 25% don't care-Bits bzw. Bits, die aufgrund der Wahl eines alternativen Wertes aus der Menge der möglichen Werte eingespart werden können

- aufgrund der Optimierung können 25% der *Ones*- und Hamming-Kosten reduziert werden

Aufgrund der maximalen Kosten für *Ones* entsteht ein Anteil von 3,7% und für die Hamming-Kosten von 1,8% (siehe Messergebnisse in Kapitel 3) mit folgendem sich ergebenden Einsparungspotenzial:

$$\frac{1}{3} * 25\% * 25\% * (3,7\% + 1,8\%) = 0,115\%$$

3. Adressbus für Instruktionsspeicher

   Die Adressen für den Instruktionsspeicher (*IAddr*) werden durch den Kontrollfluss des generierten Programms bestimmt. Sie bestehen aus der einfachen Abarbeitung von hintereinander liegenden Instruktionen und den verschiedenen Sprungbefehlen. Eine mögliche Optimierung besteht darin, häufig ausgeführte Instruktionen so im Programmspeicher zu positionieren, dass ihre Adressen bzgl. der *Ones*-Kosten und der Hamming-Distanz minimale Kosten auf dem Adressbus *IAddr* verursachen. Dies kann durch eine Optimierung des Memorylayouts der Basisblöcke und Funktionen erreicht werden.

   Von folgenden Annahmen wird bei der Abschätzung des Einsparungspotenzials ausgegangen:

   - 25% der *Ones*- und der Hamming-Kosten können optimiert werden
   - bei häufig ausgeführten Instruktionen entstehen 25% der möglichen *Ones* in den Instruktionen und 25% der maximal möglichen Hamming-Distanz zwischen aufeinander folgenden Instruktionen

   Mit dem maximalen Anteil von 1% der *Ones*- und 12% der Hamming-Kosten am Gesamtenergieverbrauch ergibt sich folgendes Potenzial:

$$25\% * 25\% * (1\% + 12\%) = 0,8125\%$$

4. Datenbus für Instruktionsspeicher

   Über den Datenbus des Instruktionsspeichers (*IData*) werden die Instruktionen in den Prozessor geholt. In Abhängigkeit des Kontrollflusses werden entweder die unmittelbar nachfolgenden Instruktionen geladen oder es wird bei Sprüngen zu den Adressen des Sprungzieles gewechselt und dort die nächsten Instruktionen geladen.

   Die Sprünge in einem Programm werden von einem Compiler minimiert, da bei Sprüngen typischerweise die Pipeline neu gefüllt werden muss, was zu einem beträchtlichen Zusatzaufwand führt. Da die Sprünge deshalb viel seltener auftreten und auch die Vorhersage, ob ein bedingter Sprung ausgeführt wird, nur schwer berechnet werden kann, werden hier bei der Optimierung die Sprünge nicht berücksichtigt. Der Schwerpunkt dieser Optimierung liegt daher auf den aufeinander folgenden Instruktionsworten, die durch entsprechende Anordnung bzgl. der *Ones*- und Hammingkosten optimiert werden.

   Diese Optimierung wird ausführlicher im Unterkapitel 6.2 auf Seite 178 beschrieben, da sie ein größeres Einsparungspotenzial verspricht.

5. Registerinhalte

   Die Inhalte der Register *RegVal* sind nur sehr schwer - wenn überhaupt - während des Compilerlaufes vorherzuberechnen. Eine Optimierung ist ebenfalls nur sehr eingeschränkt möglich. Da weiterhin diese Werte nur über interne Busse transportiert werden (mit Ausnahme von Load- und Store-Befehlen, die über die externen Busse auf den Speicher zugreifen), ist das Optimierungspotenzial als marginal einzustufen.

6. Registernummern

   Die von einen Befehl verwendeten Registernummern werden im Instruktionswort kodiert. Sie sind daher Bestandteil des Instruktionswortes auf dem Datenbus des Instruktionsspeichers (*IData*) beim Laden der Instruktion. Außerdem werden sie auch prozessorintern bei der Adressierung des Registerfiles (*Reg*) verwendet. Eine Änderung der Registernummern kann nur gemeinsam für *IData* und *Reg* vorgenommen werden. Weiterhin wird davon ausgegangen, dass der Effekt auf dem externen Bus *IData* aufgrund der größeren Kapazitäten einen stärkeren Effekt zeigt als auf dem internen Bus *Reg*. Deshalb werden die Registernummern nur indirekt über die Optimierung der Instruktionsworte optimiert. Teilweise wird dies mit dem Wert auf dem internen Bus *Reg* korrelieren, wenn bei den *Ones*-Kosten für *IData* und *Reg* die gleiche Polarität (= höhere Anzahl der *Ones* ist besser oder umgekehrt) vorliegt. Ebenso korrelieren die beiden Optimierungen, wenn bei der Hamming-Distanz die Registernummern der aufeinander folgenden Instruktionen an den gleichen Bitpositionen stehen. Falls dies nicht der Fall ist, hat die Optimierung des Instruktionswortes *IData* aufgrund des größeren Potenzials Vorrang. Diese Optimierung, die im Wesentlichen einer Umsortierung der Reihenfolge der Instruktionen entspricht, wird im Unterkapitel 6.2 auf der nächsten Seite erläutert.

   Die Optimierung der Registernummern durch Verändern der Registerzuordnung der einzelnen Instruktionen wie sie bei [MOI$^+$97] beschrieben wird, ermöglicht bei dem betrachteten System nur marginale Verbesserungen, wie Untersuchungen gezeigt haben.

7. Immediate-Werte

   Ebenso wie die Registernummern sind die Immediate-Werte (direkt im Instruktionswort enthaltene Konstanten) sowohl Bestandteil des Instruktionswortes *IData* auf dem externen Bus als auch eine eigenständige Komponente als interner Bus *Imm*. Da auch hier das Potenzial des externen Busses überwiegt, hat die Optimierung des Instruktionswortes Vorrang, die im Unterkapitel 6.2 auf der nächsten Seite beschrieben wird. Nur in sehr seltenen Fällen ist die Alternative vorhanden, dass mehrere unterschiedliche Immediate-Werte für die gleiche Semantik möglich sind. Ein Beispiel wäre ein Immediate-Wert, von dem später nur ein Teil der Bits verwendet wird. Aufgrund der geringen Häufigkeit sind die Auswirkungen einer entsprechenden Auswahl als marginal anzusehen.

8. Opcodes

   Der interne Bus *Opcode* entspricht den höheren Bits des Instruktionswortes *IData*. Auch hier gilt der Vorrang für den größeren Einfluss auf dem externen Bus. Da die Optimierung des Instruktionswortes auch direkt die Werte auf dem internen Bus *Opcode* mitverändert, ist eine getrennte Optimierung auf Basis des *Opcode*-Wertes nicht sinnvoll.

9. FUChange

   Die Kosten durch das Aktivieren oder Deaktivieren von Funktionseinheiten können durch eine andere Wahl einer Funktionseinheit oder durch eine andere Reihenfolge von Instruktionen verändert werden. Der Parameter *FUChange* ist unabhängig von Werten, die über externe Busse transportiert werden. Trotzdem bietet es sich an, diesen Parameter zusammen mit den anderen Codierungen zu optimieren, da er indirekt über das Instruktionswort kodiert wird. Zwei Fälle können hier unterschieden werden:

   (a) Eine Optimierung durch die Wahl einer alternativen Instruktion ist unter dem Begriff “Strength Reduction” bekannt und bedeutet z. B. das Ersetzen einer Multiplikation mit "2" durch ein Schieben der Binärdarstellung um eine Stelle nach links. Dies kann bereits während der Instruktionsauswahl, wie in dem entwickelten energieoptimierenden Compiler *encc*, erfolgen und muss an dieser Stelle nicht mehr betrachtet werden. Bei der Strength Reduction werden aber nicht die vorherige oder die nachfolgende Instruktion berücksichtigt. Es kann bezogen auf das

Kostenmaß bei der Strength Reduction - typischerweise Zyklenanzahl oder hier Energieverbrauch der verwendeten Funktionseinheiten - mehrere gleichwertige oder annähernd gleichwertige alternative Instruktionen geben. Von diesen Alternativen kann nun diejenige Instruktion ausgewählt werden, die auch unter der Berücksichtigung ihrer Vorgänger- und Nachfolger-Instruktion die geringsten Kosten *FUChange* aufweist.
Aufgrund der äußerst seltenen Existenz dieser Alternativen, ist das Optimierungspotenzial marginal.

(b) Im Gegensatz dazu ist der Effekt durch eine Änderung der Reihenfolge bei der Optimierung der Datencodierung der Instruktionen relevant und wird im Unterkapitel 6.2 betrachtet.

Nach der Betrachtung der vielen verschiedenen Möglichkeiten, auf den Bussen die Codierung zu optimieren, muss festgestellt werden, dass fast alle Optimierungen ein Potenzial aufweisen, dass unterhalb von 1% des Gesamtenergieverbrauchs liegt. Es wird daher im nächsten Abschnitt nur die Optimierung der Datencodierung der Instruktionen weiterverfolgt, da diese einen nennenswerten Energiegewinn verspricht.

6.2 Datencodierung der Instruktionen

Die Betrachtung der Bitwechsel auf den Bussen ist bei der Energieoptimierung schon seit längerem ein Forschungsgegenstand. Dies liegt unter anderem auch daran, dass hierfür kein Compiler mit der entsprechenden Compilerbau-Technologie notwendig ist, sondern die Optimierung selbst schon bei der Betrachtung von Bussen relevant ist.

In [STD94] werden zwei neue Techniken zur Reduzierung der Switching-Aktivität vorgestellt:

- *Gray Code* Adressierung
  Durch die zusätzlichen Codierer und Decodierer wird eine binäre Darstellung in *Gray Code* umgewandelt. Es ergeben sich dadurch Reduzierungen zwischen 30 und 50% der Bitwechsel, die eingespart werden können. Diese Technik ist jedoch nur anwendbar, wenn Hardwaremodifikationen vorgenommen werden.
- *Cold Scheduling*
  Unter der Technik des *Cold Scheduling* wird eine Umsortierung der Instruktionen verstanden, indem die Bitwechsel minimiert werden. Die Optimierung beschränkt sich auf Änderungen der Software, was auch der Zielsetzung und Beschränkung dieser Arbeit entspricht. In [STD94] wird von Reduzierungen zwischen 20 und 30% im Kontrollpfad des Prozessors bei Anwendung des Cold Scheduling berichtet. Allerdings ergeben sich Performance-Einbußen in Höhe von 2 bis 4%.

Bei [TL98] wurde für einen 32-Bit-Embedded Microcontroller das Verfahren des Cold Scheduling von [STD94] angewandt. Es wurden aber keine signifikanten Reduzierungen des Energieverbrauchs durch diese Optimierung der Instruktionsanordnung festgestellt. Die Ursache für die unterschiedlichen Ergebnisse von [TL98] und [STD94] können durch die unterschiedlichen Basis der Vergleiche begründet sein. Während bei [TL98] der gesamte Prozessor gemessen wird, bei dem der Kontrollpfad nur einen möglicherweise geringen Anteil ausmacht, beschränkt sich der Vergleich von [STD94] allein auf den Einfluss der Optimierung auf den Kontrollpfad.

Speziell mit dem ARM-Prozessor und der Optimierung der Bitwechsel beschäftigen sich Sinevriotis et al. [SS99, SS01]. Es wird das Energiemodell von Tiwari verwendet und durch Instruktionsanordnung die Anzahl der Bitwechsel reduziert sowie einzelne Befehle durch äquivalente Befehle ersetzt (Strength Reduction). Die Energieeinsparung beträgt für einen IEEE 802.11 Protokoll-Benchmark 9,17%. Zu berücksichtigen ist allerdings, dass die Werte nicht nachvollziehbar waren, da der Gesamteinfluss nur 2,2%

beträgt, und das in diesem Projekt entstandene Tool eine Instruktionsanordnung ohne Berücksichtigung der Datenabhängigkeiten durchgeführt hat. Außerdem muss zwischen dem Energieverbrauch des Prozessorcores und dem Gesamtverbrauch eines Systems einschließlich Prozessor-I/O und Speicher unterschieden werden. Verbesserungen bezüglich des Prozessorcores werden im Verhältnis zum Gesamtsystem viel niedriger liegen.

Weitere Arbeiten zur Instruktionsanordnung finden sich in [MC95, BR95, CN00, LLHT00, TCR98].

Für das betrachtete ARM7T-System wurde geprüft, inwieweit die vorgeschlagenen Optimierungstechniken Einfluss zeigen. Dafür wurde ein Move-Befehl, der aus dem Hauptspeicher geladen wurde, als Maßstab genommen und untersucht, wie sich der Wechsel von allen 16 Bits auswirkt. Wenn zwischen verschiedenen Zugriffen alle 16 Bits geändert werden, entspricht dies lt. [Kna01] einer Energiedifferenz von 0,84 nJ. Der Energieverbrauch des Prozessors beträgt für einen Move-Befehl 8,86 nJ und für den Hauptspeicher unserer Platine 29,00 nJ, sodass insgesamt ein Systemenergieverbrauch von 37,86 nJ entsteht. Dadurch entsteht eine obere Schranke von 0,84 / 37,86 = 2,2%, die maximal für diesen Befehl eingespart werden kann. Da die anderen Befehle fast ausschließlich einen höheren Verbrauch aufweisen, entspricht der Wert für den Move-Befehl auch annähernd einer oberen Schranke aller Befehle.

Im nächsten Schritt wurden für verschiedene Benchmarks mit dem encc-Compiler die folgenden zwei Testreihen durchgeführt:

Testreihe 1:

Jedes Programm wurde auf Bitwechsel (ohne Berücksichtigung der FU-Wechsel) mit einem simplen Verfahren optimiert, indem einzeln für jeden Befehl die optimale Position innerhalb des Basisblocks gesucht wurde. Dies wurde der Reihe nach für alle Instruktionen eines Basisblockes durchgeführt. Da jeder Befehl nur einzeln betrachtet und nicht gleichzeitig alle Befehle optimiert wurden, erhält man insgesamt nicht die optimale Lösung, sondern eine untere Schranke. Diese untere Schranke zeigt ein Verbesserungspotenzial bei der Anzahl der Bitwechsel von 3,15%. Man erhält unter Berücksichtigung des Anteils der Bitwechsel von 2,2% am Gesamtenergieverbrauch des Systems eine Reduzierung des Energieverbrauchs um 3,15% * 2,2% = 0,0693% als untere Schranke.

Testreihe 2:

In einer weiteren Testreihe wurde eine obere Schranke berechnet, in dem die Bitwechsel durch optimales Platzieren innerhalb eines Basisblockes ohne Berücksichtigung der Datenabhängigkeiten minimiert wurden. Dieser Wert kann daher nur als theoretisches Maximum angesehen werden, um eine obere Schranke zu ermitteln. Die Werte unter Berücksichtigung der Datenabhängigkeiten werden daher eher bei der unteren Schranke zu erwarten sein. Für verschiedene Benchmarks wurde bei dieser Testreihe insgesamt eine Reduzierung der Anzahl der Bitwechsel um 41,89% erzielt. Für den Energieverbrauch des Systems ergibt sich daraus mit 41,89% * 2,2% = 0,92% die obere Schranke.

Da sich durch die Optimierung der Bitwechsel der Instruktionsworte lediglich ein Einsparungspotenzial des Systemenergieverbrauchs in der Größenordnung zwischen 0,0693% und 0,92% ergibt, wird diese Optimierung nicht weiter verfolgt.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Das Ziel dieser Arbeit ist die Untersuchung des Energieeinsparpotenzials durch die Anwendung energieoptimierender Compilertechniken. Für diese Untersuchungen wird ein energieoptimierender Compiler vorgestellt und systematisch alle Möglichkeiten des Energiesparens durch Compileroptimierungen untersucht. Die besonders profitablen Optimierungen, die im Wesentlichen auf der Speicherhierarchie arbeiten, werden detailliert präsentiert und auf ihre Einsparung hin bewertet. Durch die Arbeit können die Notwendigkeit einer gesonderten Berücksichtigung des Energieverbrauchs in Compilern offengelegt und des Weiteren neue Wege aufgezeigt werden, große Einsparungen zu erreichen. Teilweise führen diese Optimierungen zu Performanceverbesserungen des generierten Programms. Im Folgenden wird der Inhalt und die Ergebnisse der einzelnen Kapitel zusammengefasst.

Zuerst werden die Ursachen des Energieverbrauchs in elektronischen Schaltungen und im Besonderen in elektronischen CMOS-Schaltkreisen beschrieben. Der wesentliche Anteil des Energieverbrauchs entsteht durch die Umschaltvorgänge der CMOS-Transistoren, die daher das größte Einsparungspotenzial darstellen. Ein Ansatzpunkt im Entwurfsablauf von eingebetteten Systemen ist die Optimierung durch den Compiler, da dadurch ohne zusätzliche Entwicklungsphasen und zusätzlichen Designaufwand Verbesserungen erzielt werden können.

Bisherige Compiler berücksichtigen allerdings lediglich die Laufzeit und Programmgröße und müssen daher für die Energieoptimierung entsprechend erweitert werden. Dafür werden die unterschiedlichen Phasen eines Compilers erläutert und notwendige Ergänzungen zur Berücksichtigung des Energieverhaltens eingebaut. Ansatzpunkte sind im Wesentlichen im Back-End zu finden, da erst dort ein genauerer Bezug zu den generierten Maschineninstruktionen und ihrem Energieverbrauch hergestellt werden kann. Als Beispiel für die Untersuchungen wurde ein RISC-Prozessor gewählt und für diesen ein neues Back-End entwickelt.

Für die Berücksichtigung des Energieverbrauchs eines Prozessors und des zugehörigen Speichers müssen innerhalb des Compilers diese Komponenten als Modell nachgebildet werden. Hierfür werden existierende Energiemodelle vorgestellt und bewertet. Um die vorgestellten Anforderungen zu erfüllen, muss ein neues Energiemodell entwickelt werden, welches durch verschiedene Messreihen am Beispielsystem des ARM7TDMI mit Daten gefüllt wird. Der Compiler und ein gesonderter Profiler werden an das Energiemodell angepasst, sodass sowohl während der Programmgenerierung als auch nach einem Simulationslauf durch den neu entwickelten Profiler der Energieverbrauch bewertet werden kann.

Bei den betrachteten RISC-Prozessoren bietet der Energieanteil des Speichers ein großes Potenzial für Einsparungen. Als Grundlage für spätere Optimierungen wird die prinzipielle Arbeitsweise einer Speicherhierarchie einschließlich Caches und Scratchpad-Speichern vorgestellt. Beispielhaft können Speicherzugriffe durch eine effizientere Nutzung der internen Prozessorregister reduziert werden.

Eine wesentliche Reduzierung des Energieverbrauchs bei eingebetteten Systemen kann die Nutzung eines Scratchpad-Speichers anstatt eines Caches liefern. Die profitabelsten Programmteile und Variablen werden dafür in den Scratchpad verlagert, wodurch gegenüber Systemen mit Caches eine Energiereduzierung zwischen 8,0% und 76,7% erreicht werden kann. Um auch komplexere Programme mit mehreren Hotspots verwalten zu können, wird die vorgestellte Technik auf den Austausch der Programmteile während der Programmausführung erweitert. Auch hier werden Energieeinsparungen gegenüber Caches zwischen 29% und 55% erreicht. Ein weiterhin interessanter Aspekt ist die gleichzeitige Steigerung der Performance zwischen 21% und 44%.

Im letzten Kapitel wird der Ansatzpunkt der Zustandswechsel auf Bussen auf sein Energieeinsparungspotenzial hin untersucht. Insgesamt ist das Einsparungspotenzial für das betrachtete System jedoch gering und die erzielten Einsparungen größtenteils marginal.

Insgesamt konnte dargelegt werden, dass durch eine Erweiterung herkömmlicher Compiler die Optimierung des Energieverbrauchs in beträchtlichem Maße möglich ist. Durch die Integration der vorgestellten Compilertechniken kann der Energieverbrauch zwischen 8,0% und 76,7% verringert werden.

Im Folgenden soll ein Ausblick über zukünftige Ansatzpunkte und die mögliche Fortsetzung dieser Arbeit gegeben werden.

7.2 Ausblick

Durch diese Arbeit wird eine Plattform und Basis geschaffen, die fortführende Untersuchungen gestattet. Dazu gehört insbesondere die Betrachtung komplexerer Speicherhierarchien, die sowohl die Kombination aus Caches und Scratchpads oder auch mehrere Cache-Ebenen und Scratchpads umfassen. Hierdurch sind weitere nennenswerte Verbesserungen zu erwarten. Die vorgestellten Algorithmen können nicht nur auf die in diesem Beispielsystem betrachtete Kombination von Offchip- und Onchip-Speicher angewandt werden, sondern auch auf andere Speicherkombinationen, wie beispielsweise einen kleinen zusammen mit einem großen Onchip-Speicher.

Die vorgestellte dynamische Technik umfasst die Behandlung von Programmteilen, ohne die bei der statischen Verlagerung zusätzlich berücksichtigten Variablen. Die Weiterentwicklung der dynamischen Technik kann daher in der Integration der Variablen und in der Zusammenfassung der statischen und der dynamischen Technik bestehen.

Generell kann die Grundidee der Behandlung von Basisblöcken dahingehend erweitert werden, dass der letzte nicht vollständig in das Scratchpad passende Basisblock aufgeteilt und nur teilweise in das Scratchpad verlagert wird.

Ein weiterer Ansatz, der die Möglichkeit des Hardwareentwurfs umfasst, besteht in der Realisierung der Kopierfunktion von Basisblöcken durch eine Hardwarelogik. Das effizientere Kopieren der Blöcke verspricht weitere größere Einsparungen.

Zusammenfassend kann festgestellt werden, dass hiermit eine Basis geschaffen wurde, aus der noch vielfältige Ansätze heraus entwickelt werden können.

Literaturverzeichnis

[ADR98] A. Appel, J. Davidson, and N. Ramsey. *The Zephyr Compiler Infrastructure*. Princeton University and University of Virginia, Nov 1998.

[AG98] A. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[All01] *Allensbacher Computer- und Telekommunikations-Analyse*. Institut für Demoskopie Allensbach, 2001.

[ARM] ARM. *Advanced RISC Machines Ltd.* http://www.arm.com.

[ARM95a] *ARM7TDMI Data Sheet*. ARM DDI 0029E, ARM Ltd., http://www.arm.com, 1995.

[ARM95b] *An Introduction to Thumb*. ARM DVI 0001A, ARM Ltd., http://www.arm.com, 1995.

[ARM98] *ARM710T Data Sheet*. ARM DDI 0086B, ARM Ltd., http://www.arm.com, 1998.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilerbau*. Addison-Wesley, 1988.

[BAM98] R. I. Bahar, G. Albera, and S. Manne. Power and Performance Tradeoffs using Various Caching Strategies. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 64–69, Monterey, CA, Aug 1998.

[Bel00] F. Bellosa. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. In *Proceedings of 9th ACM SIGOPS European Workshop*, pages 37–42, Kolding, Denmark, Sep 2000.

[Bäh91] H. Bähring. *Mikrorechner-Systeme*. Springer, 1991.

[BM98] L. Benini and G. De Micheli. *Dynamic Power Management - Design Techniques and CAD Tools*. Kluwer Academic Publishers, 1998.

[BMMP00] L. Benini, A. Macii, E. Macii, and M. Ponicino. Synthesis of Application-Specific Memories for Power Optimization in Embedded Systems. In *Proceedings of the Design Automation Conference*, pages 300–303, Los Angeles, CA, Jun 2000.

[BR95] V. Bala and N. Rubin. Efficient Instruction Scheduling Using Finite State Automata. In *Proceedings of the International Symposium on Microarchitecture*, pages 46–56, Ann Arbor, MI, Nov 1995.

[BSL+01] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption. Technical Report 762, University of Dortmund, 2001.

[BSL+02] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems. In *Proceedings of the International Symposium on Hardware/Software Codesign*, pages 73–78, Estes Park, CO, May 2002.

[Car92] S. Carr. *Memory-Hierarchy Management*. PhD Thesis, CRPC-TR92222-S, Rice University, 1992.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, NY, Jun 1990.

[CDK+02] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002.

[CKL00] N. Chang, K. Kim, and H. G. Lee. Cycle-Accurate Consumption Measurement and Analysis: Case Study of ARM7TDMI. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 185–190, Rapallo, Italy, Jul 2000.

[CN00] B. R. Childers and T. Nakra. Reordering Memory Bus Transactions for Reduced Power Consumption. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 146–161, Vancouver, Canada, Jun 2000.

[Com] *Computer User Dictionary*. http://www.computeruser.com/resources/dictionary.

[CPL] *ILOG CPLEX optimizer*. http://www.ilog.com, ILOG Inc.

[CWG+98] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology - Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[DKV+01] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM Energy Management Using Software and Hardware Directed Power Mode Control. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 159–169, Nuevo Leone, Mexico, Jan 2001.

[EIA] *Annual Energy Outlook 2002, DOE/EIA-0383*. Energy Information Administration. http://www.eia.doe.gov/oiaf/aeo.

[Fal02] H. Falk. Control Flow Optimization by Loop Nest Splitting at the Source Code Level. Technical Report 773, University of Dortmund, 2002.

[FH95] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc., 1995.

[FHP92] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. *Engineering a Simple, Efficient Code Generator Generator*. AT & T Bell Laboratories, 1992.

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

[Fre97] J. Frenkil. Embedded Tutorial: Tools and Methodologies for Low Power Design. In *Proceedings of the Design Automation Conference*, pages 76–81, Anaheim, CA, Jun 1997.

[gcc] Free Software Foundation, GCC Compiler. http://gcc.gnu.org.

[Gro96] G. Grosche. *Teubner-Taschenbuch der Mathematik*. B.G. Teubner, 1996.

[Gru02] N. Grunwald. *Energieminimierung eingebetteter Programme durch die dynamische Nutzung eines Scratchpad-Speichers, Diplomarbeit*. Universität Dortmund, 2002.

[Hül02] T. Hüls. *Energieoptimierung einer MPEG-Applikation, Diplomarbeit*. Universität Dortmund, 2002.

[HP90] J. L. Hennessy and D. A. Patterson. *Rechnerarchitektur - Analyse, Entwurf, Implementierung, Bewertung*. Friedr. Vieweg und Sohn, 1990.

[Hu01] Y. H. Hu. *Programmable Digital Signal Processors - Architecture, Programming, and Applications*. Vol. 13, Marcel Dekker Inc., New York, 2001.

[Int98] *Mobile Power Guidelines 2000*. Intel Corporation, Dec 1998.

[Int00] *The Intel XScale Microarchitecture Technical Summary*. Intel Corporation, 2000.

[Int02] *ISSCC: McKinleys Cache, 5-GHz-Chip und ovonische Speicher*. 2002. http://www.heise.de/newsticker/data/wst-05.02.02-002/.

[ITR01] International Technology Roadmap of Semiconductors. http://public.itrs.net, 2001.

[IY96] T. Ishihara and H. Yasuura. Experimental Analysis of Power Estimation Models of CMOS VLSI Circuits. *IEICE TRANS. Fundamentals*, E00-A(6), Jun 1996.

[IY00] T. Ishihara and H. Yasuura. A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 617–623, Paris, France, Mar 2000.

[KG02] A. Krishnaswamy and R. Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *Proceedings of Joint Conference on Languages, Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 56–64, Berlin, Germany, Jun 2002.

[Kna01] M. Knauer. *Codierungsverfahren zur Reduktion des Energiebedarfs von Programmen, Diplomarbeit*. Universität Dortmund, 2001.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1978.

[KRI$^+$01] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proceedings of the Design Automation Conference*, pages 690–695, Las Vegas, NV, Jun 2001.

[KVIY00] M. Kandemir, N. Vijakrishnan, M. J. Irwin, and W. Ye. Influence of Compiler Optimizations on System Power. In *Proceedings of the Design Automation Conference*, pages 304–307, Los Angeles, CA, Jun 2000.

[LAN] LANCE retargetable Compiler. http://ls12-www.cs.uni-dortmund.de/lance, University of Dortmund.

[lcc] Free Software Foundation, LCC Compiler. http://www.cs.princeton.edu/software/lcc.

[Lee01] B.-S. Lee. *Vergleich des Energieverbrauchs von Cache- und Scratch-Pad-Speichern für den ARM7-Prozessor, Diplomarbeit*. Universität Dortmund, 2001.

[LEMC01] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–10, Snowbird, UT, Jun 2001.

[liv] Livermore Benchmarks. http://scicomp.ewha.ac.kr/netlib/benchmark/livermorec.

[LLHT00] C. Lee, J. Lee, T. Hwang, and S. Tsai. Compiler Optimization on Instruction Scheduling for Low Power. In *Proceedings of the International Symposium on System Synthesis*, pages 55–60, Madrid, Spain, Sep 2000.

[LM01] R. Leupers and P. Marwedel. *Retargetable Compiler Technology for Embedded Systems*. Kluwer Academic Publishers, 2001.

[Mar00] P. Marwedel. *Skript zur Vorlesung Rechnerarchitektur*. University of Dortmund, 2000.

[MC95] E. Musoll and J. Cortadella. Scheduling and Resource Binding for Low Power. In *Proceedings of the International Symposium on System Synthesis, Cannes, France*, pages 104–109, Apr 1995.

[MF99] R. Murgai and M. Fujita. On Reducing Transitions Through Data Modifications. In *Proceedings of Design, Automation and Test in Europe Conference*, pages 82–88, Munich, Germany, Mar 1999.

[MIP99] *MIPS32 Architecture*. MIPS Technologies, Inc., 1999.

[MOI+97] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for Low Energy Software. In *Proceedings of the International Symposium of Low Power Electronics and Design*, pages 72–75, Monterey, CA, Aug 1997.

[MPS98] E. Macii, M. Pedram, and F. Somenzi. High-Level Power Modeling, Estimation, and Optimization. *IEEE Transactions on CAD of ICs and Systems*, 17(11):1061–1079, Nov 1998.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, NY, 1988.

[OIY99] T. Okuma, T. Ishihara, and H. Yasuura. Real-Time Scheduling for a Variable Voltage Processor. In *Proceedings of the International Symposium on Systems Synthesis*, pages 24–29, San Jose, CA, Nov 1999.

[PDN97] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of European Design and Test Conference*, pages 7–11, Paris, France, Mar 1997.

[PDN99] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.

[Prz90] S. A. Przybylski. *Cache and Memory Hierarchy Design - A Performance-Directed Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[RJ98] J. T. Russell and M. F. Jacome. Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors. In *Proceedings of International Conference on Computer Design*, pages 328–333, Austin, TX, Oct 1998.

[SBM99] T. Simunic, L. Benini, and G. De Micheli. Cycle-Accurate Simulation of Energy Consumption in Embedded Systems. In *Proceedings of the Design Automation Conference*, pages 867–871, New Orleans, LA, Jun 1999.

[SC99] W.-T. Shiue and C. Chakrabarti. Memory Exploration for Low Power, Embedded Systems. In *Proceedings of the Design Automation Conference*, pages 140–145, New Orleans, LA, Jun 1999.

[SC01] A. Sinha and A. P. Chandrakasan. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Proceedings of the Design Automation Conference*, pages 220–225, Las Vegas, NV, Jun 2001.

[SCG95] S. Segars, K. Clarke, and L. Goudge. Embedded Control Problems, Thumb, and the ARM7TDMI. *IEEE Micro*, pages 22–30, Oct 1995.

[Sch00] R. Schwarz. *Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining, Diplomarbeit*. Universität Dortmund, 2000.

[Seg97] S. Segars. ARM7TDMI Power Consumption. *IEEE Micro*, pages 12–19, Jul/Aug 1997.

[Seg01] S. Segars. Low Power Design Techniques for Microprocessors. In *Proceedings of the International Solid-State Circuits Conference*, San Francisco, CA, Feb 2001.

[SFL98] J. Sjödin, B. Fröderberg, and T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. In *Proceedings of the Workshop on Compiler and Architectural Support for Embedded Computer Systems*, Washington DC, Dec 1998.

[SGW+02] S. Steinke, N. Grunwald, L. Wehmeyer, , R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *Proceedings of the International Symposium on System Synthesis*, pages 213–218, Kyoto, Japan, Oct 2002.

[Sim92] K. Simon. *Effiziente Algorithmen für perfekte Graphen*. B. G. Teubner, 1992.

[SKWM01] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proceedings of the International Workshop - Power and Timing Modeling, Optimization and Simulation*, Yverdon-Les-Bains, Switzerland, Sep 2001.

[SPA92] *The SPARC Architecture Manual, Version 8*. SPARC International Inc., 1992.

[SS99] G. Sinevriotis and T. Stouraitis. Power Analysis of the ARM 7 Embedded Microprocessor. In *Proceedings of the International Workshop Power and Timing Modeling, Optimization and Simulation*, pages 261–270, Kos Island, Greece, Oct 1999.

[SS01] G. Sinevriotis and T. Stouraitis. *SOFLOPO - Low Power Software Development for Embedded Applications: Public Final Report*. University of Patras, ATMEL Hellas, Greece, Jan 2001.

[SSWM01] S. Steinke, R. Schwarz, L. Wehmeyer, and P. Marwedel. Low Power Code Generation for a RISC Processor by Register Pipelining. Technical Report 754, University of Dortmund, Mar 2001.

[STD94] C.-L. Su, C.-Y. Tsui, and A. M. Despain. Low Power Architecture Design and Compilation Techniques for High-Performance Processors. In *Proceedings of IEEE COMPCON*, pages 489–498, Feb 1994.

[Sti02] A. Stiller. Die ARM-Story. In *c't, Magazin für Computertechnik*, pages 70–73, 2/2002.

[SUI] SUIF Compiler System. http://suif.stanford.edu.

[SWLM02] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of Design Automation and Test in Europe*, pages 409–415, Paris, France, Mar 2002.

[Syn96] *Power Products Reference Manual*. Synopsys, Sep 1996.

[TCR98] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction Scheduling for Low Power Dissipation in High Performance Microprocessors. In *Proceedings of Power Driven Microarchitecture Workshop in conjunction with the ISCA*, Barcelona, Spain, Jun 1998.

[The00] M. Theokharidis. *Energiemessung von ARM7TDMI Prozessor-Instruktionen, Diplomarbeit*. Universität Dortmund, 2000.

[Tiw96] V. Tiwari. *Logic and System Design for Low Power Consumption, PhD Thesis*. Princeton University, November 1996.

[TL98] V. Tiwari and M. T.-C. Lee. Power Analysis of a 32-bit Embedded Microcontroller. *VLSI Design Journal*, 7(3), 1998.

[TMW94a] V. Tiwari, S. Malik, and A. Wolfe. Compilation Techniques for Low Energy: An Overview. In *Proceedings of the Symposium on Low Power Electronics*, San Diego, CA, Oct 1994.

[TMW94b] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step towards Software Power Minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, Dec 1994.

[TMW96] V. Tiwari, S. Malik, and A. Wolfe. Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing Systems*, 13(2):223–238, Aug-Sep 1996.

[Tri] *Trimaran - An Infrastructure for Research in Instruction-Level Parallelism*. http://www.trimaran.org.

[vdW] Rik van de Wiel. *The Code Compaction Bibliography*. http://www.extra.research.philips.com/ccb.

[VSM03] M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of the Asia and South Pacific Design Automation Conference*, Kitakyushu, Japan, Jan 2003.

[WE94] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 1994.

[Weg00] I. Wegener. *Skript zur Vorlesung Effiziente Algorithmen*. Universität Dortmund, 2000.

[WH98] N. Wehn and S. Hein. Embedded DRAM Architectural Trade-Offs. In *Proceedings of Design Automation and Test in Europe Conference*, pages 704–708, Paris, France, Feb 1998.

[Wid02] A. Widiger. *Skript zur Vorlesung Programmierungstechnik*. Universität Rostock, 2002.

[WJ94] S. J. E. Wilton and N. P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, Western Research Laboratory, Jul 1994.

[WJ96] S. J. E. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[XSc] *Intel XScale Microarchitecture*. Intel Corporation, http://developer.intel.com/design/ intelxscale/.

[Yea98] G. K. Yeap. *Practical Low Power Digital VLSI Design*. Kluwer Academic Publishers, 1998.

[Zep] *Tools for a National Compiler Infrastructure*. http://www.cs.virginia.edu/zephyr/.

[Zob01] C. Zobiegala. *Energieeinsparung durch compilergesteuerte Nutzung des On-Chip-Speichers, Diplomarbeit*. Universität Dortmund, 2001.

[ZVSM94] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of Signal Processing Applications & Technology*, pages 715–720, Dallas, TX, Oct 1994.

Index