**Diplomarbeit**

# Energieeinsparung durch compilergesteuerte Nutzung des On-Chip-Speichers

Christoph Zobiegala

Diplomarbeit

am Lehrstuhl 12

des Fachbereichs Informatik

der Universität Dortmund

24. September 2001

**Betreuer:**

Dipl.-Inf. Stefan Steinke

Prof. Dr. Peter Marwedel

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Seit Beginn des Einzugs von Mikrocontroller-basierten Systemen vor über 50 Jahren gibt es fortlaufend und in immer kürzeren Perioden Steigerungen in punkto Leistung, Miniaturisierung und umgekehrt proportional dazu im Stromverbrauch. Doch sind die beiden ersten Aspekte forcierte Entwicklungen, ergibt sich der geringer gewordene Verbrauch eher zwangsläufig durch verbesserte Technologien. Diese Entwicklungen führen dazu, dass immer mehr prozessorbasierte Systeme in immer mehr Bereiche vordringen. Waren früher aufgrund der Grösse und der geringen Rechenleistung mobile Anwendungen nicht möglich, so hat sich das in den letzten Jahren grundlegend geändert. Es wurden mobile Bereiche in PDA's, Mobiltelefonen, eBooks und Kameras erschlossen.

So ist heutzutage nicht mehr die Grösse oder die Rechenleistung der Hauptaspekt, der den Einsatz solcher Systeme einschränkt, sondern der die Einsatzdauer limitierende Verbrauch an Energie. Mit weiteren Entwicklungen auf der Hardwareseite wird versucht, den Verbrauch zu senken. Immer umfangreichere Anwendungen, die auf den mobilen Systemen zum Einsatz kommen, treiben diesen wiederum in die Höhe.

Durch verbesserte Algorithmen kann die Effizienz der Programme sicherlich gesteigert werden. Das macht sich dann auch in der Laufzeit und oft im Energieverbrauch positiv bemerkbar.

Da heutige Programme durch ihre Komplexität und ihren Umfang mitlerweile alle ausschliesslich in Hochsprachen wie C, C++, Smalltalk geschrieben werden, sie jedoch auf Maschinen, die nur Maschinencode verarbeiten, ausgeführt werden, bedarf es eines Compilers um Hochsprachenprogramme in Maschinencode zu übersetzen. Compiler ihrerseits haben erheblichen Einfluss auf den Energieverbrauch der von ihnen übersetzten Programme. Sie sind es, die Besonderheiten der Zielmaschine kennen und versuchen diese bestmöglich zu nutzen. Die Besonderheiten können Speicherdesign, Befehlssatz usw. sein.

Im Zusammenhang mit Speicher weiss man, dass Hauptspeicherzugriffe einen sehr hohen Anteil am Gesamtenergieverbrauch eines Programms haben. Oft stellt die Zielhardware mehrere unterschiedliche Speicherarten mit unterschiedlichen Zugriffskosten zur Verfügung. In solchen Fällen kann ein optimierender Compiler diese Tatsache ausnutzen und den Gesamtenergieverbrauch senken.

In dieser Diplomarbeit soll eine solche Optimierung vorgestellt werden. Sie wurde für den experimentellen "encc" Compiler implementiert und hat als Ziel, den Energieverbrauch durch Nutzung des On-Chip Speichers des AT91M40400 Controllers, im weiteren Scratch-Pad genannt, zu senken. Aus der Tatsache, dass der On-Chipspeicherzugriff 1/8 der Energie eines Hauptspeicherzugriffs benötigt, ergibt sich ein enormes Einsparpotenzial.

## 1.1 Motivation

Mit zunehmender Mobilität des Menschen und zunehmender Anzahl mobiler Geräte, die diesen Trend begleiten, hat Energiesparsamkeit und niedriger Verbrauch eine neue Bedeutung bekommen. Wenn auch die "alten" Argumente wie Energiekosten und Resourcenschonung immer noch relevant und wichtig sind, so kommen neue Gründe, die die Verringerung des Energieverbrauchs zwingend vorschreiben, hinzu.

Es sind:

- kritische Wärmeentwicklung
  Ein hoher Energieverbrauch hat eine Erhöhung der Betriebstemperatur zufolge. Es entsteht zusätzlicher Kühlungsbedarf. Dadurch steigt der Platzbedarf der Systeme ebenso wie die Kosten. Ein weiteres Problem der Wärmeabführung entsteht.

- hohe Zuverlässigkeit
  Die im vorigen Punkt angesprochene Wärmeentwicklung führt auf die Dauer zu höherer Materialbeanspruchung, die sich in letzter Konsequenz auf die Lebensdauer des Gesamtsystems negativ auswirkt. Da die Zahl eingebetteter Systeme steigt, die auch zunehmend in kritischen Bereichen (Autopiloten, Überwachungssysteme in Atomkraftwerken) eingesetzt werden, ist der Faktor Zuverlässigkeit enorm wichtig.

- begrenzte Energiezufuhr
  Der begrenzte Energievorrat mobiler Systeme führt zwangsläufig zur einer geringeren Einsatzdauer. Diese könnte durch Reduzierung des Energieverbrauchs verlängert werden und somit vielleicht marktentscheidende Vorteile bringen.

- Energiekosten
  Auch wenn man bei den mobilen Systemen mit geringem Energiever-

brauch pro Einheit zu tun hat, so macht ihre ständig steigende Zahl in der Summe einen immer grösseren Betrag aus. Dabei sind Energiekosten nicht einzig und allein die Kosten ihrer Bereitstellung, sondern alle mit der Bereitstellung anfallenden Folgekosten.

Wie man hieraus ersehen kann, gibt es einige gute Gründe, die für die Reduktion des Energieverbrauchs sprechen. Sie umfassen nicht nur die Ökonomie aber auch die Ökologie, die nicht ausser Acht zu lassen ist.

## 1.2 Ziele der Diplomarbeit

Wie in der Einleitung breits kurz erwähnt, ist ein Ziel dieser Diplomarbeit die Implementierung und Untersuchung eines Optimierungsverfahrens zur Senkung des Energiebedarfs von Programmen. Dieses Ziel soll durch die effiziente Nutzung des Scratch-Pad Speichers erreicht werden. Der Scratch-Pad ist ein meist wenige kB kleiner Speicher, dessen Energieverbrauchswerte deutlich unter denen des Hauptspeichers liegen. Er ist in den meisten Fällen nicht gross genug, um vollständig ein Programm aufzunehmen. So muss man sich Strategien überlegen, Teile des gesamten Programmes dorthin zu verschieben.

Um dieses Ziel zu verwirklichen, wurde im Rahmen dieser Diplomarbeit die Optimierungstechnik Memory Allocation implementiert und in den am Lehrstuhl 12, Fachbereich Informatik der Universität Dortmund für Forschungszwecke entwickelten "encc" Compiler integriert. Für die Umsetzung werden Analysen des Programmverhaltens, die auf dem Callfunction Graph durchgeführt werden, herangezogen. Der durch den Compiler mit der Optimierungsmethode Memory Allocation modifizierte Code wurde anschliessend untersucht. Zur Bewertung der Auswirkungen auf den Energieverbrauch des Prozessors wurde auf eine bestehende Arbeitsumgebung, die im Rahmen vorangegangener Diplomarbeiten entwickelt wurde, zurückgegriffen. Das sind einerseits ein Traceanalyzer, dessen Aufgabe es ist, das durch einen Simulator simulierte Programm zu bewerten, und andererseits ein Energiemodell mit realistischen Verbrauchswerten des Prozessors, der Grundlage der Bewertung des Traceanalyzers ist.

## 1.3 Überblick

Der weitere Teil der Diplomarbeit gibt mit dem Kapitel 2.1 und 2.2 Einblick in Themenrelevante Gebiete. Der erste Teil beschäftigt sich mit den elektrotechnischen Grundlagen. Der zweite Teil des Kapitels führt in das komplexe Thema Compiler und die energiereduzierenden Massnahmen in diesem Umfeld ein. Es wird auch ein Einblick in die Standardoptimierungen und speziell die energieverbrauchssenkenden Optimierungen gegeben.

Kapitel 3.1 stellt den für diese Arbeit zur Verfügung stehenden ARM7TDMI Prozessor und sein Speicherlayout im Hinblick auf den Energieverbrauch und das für alle Messungen zugrundeliegende Energiemodell vor.

Kapitel 4 beschreibt zu Anfang die auf dem Gebiet der effizienten Nutzung von On-Chip Speicher bereits gemachten Untersuchungen und stellt im Anschluss daran die verbesserte Optimierungsmethode *Memory Allocation* mit den ihr zugrundeliegenden Konzepten und Analysen vor. Zur Verwirklichung der Optimierungsmethode *Memory Allocation* muss ein NP-vollständiges Problem gelöst werden.

Kapitel 5 befasst sich mit eben diesem Problem, der Einordnung in die Problemklasse wie auch mit den möglichen Lösungsmöglichkeiten. Im weitern Teil des Kapitels wird eingehend die Methode, die tatsächlich in dieser Diplomarbeit zur Lösung benutzt wurde, besprochen. Am Ende werden Ergebnisse, die an gängigen Benchmarks mit der Methode erzielt wurden, vorgestellt.

Im Kapitel 6 wird das zuvor auf eine Dimension reduzierte Problem mit Hilfe von *Integer Linear Programming* als mehrdimensionales Knapsack-Problem dargestellt und mit CPLEX, einen ILP-Solver, gelöst. Abschliessend werden auch hier die Ergebnisse der Untersuchungen dargestellt.

Die Zusammenfassung sowie weitere Ansätze für zukünftige Untersuchungen kommen im Kapitel 7 zur Sprache.

# Kapitel 2

# Grundlagen

Warum beschäftigt man sich mit dem Thema Energieverbrauch speziell mit der Senkung desselben, wenn man doch hört, dass die Energiewirtschaft einen Überschuss an Energie produziert? Es muss andere Gründe geben, die Entwicklerteams dazu veranlassen sich auf diesem Gebiet zu betätigen.

Im Zeitalter der Mikroelektronik und Fertigungstechnologien von VLSI-Schaltungen im Nanometer-Bereich ist der Energieverbrauch, der damit gefertigten Prozessoren und Speicherbausteine stetig nach unten gegangen. Die damit verbundene Reduktion der Grösse hat allerdings die Steigerung der Anzahl der Transistoren auf einem Chip zur Folge gehabt und die Erhöhung der Taktraten im Gigahertz-Bereich ermöglicht. Diese Entwicklung verlief im Hinblick auf den Energieverbrauch äusserst kontraproduktiv, sodass man sich bei gesteigerten Möglichkeiten der Prozessoren heute immer noch dem Problem desEnergieverbrauchs gegenüber steht.

So ist heute stärker denn je Ziel der Entwicklungen, neben der Steigerung der Geschwindigkeit den Energieverbrauch zu reduzieren. Waren diese Optimierungen früher ausschliesslich auf Seiten der Hardwareentwicklung beachtete Ziele, so hat man vor einigen Jahren auch auf der Softwareseite die Bemühungen verstärkt, ebenfalls durch Modifikation von Programmen, bei gleichbleibendem Verhalten, den Energieverbrauch zu reduzieren.

Da heutige Programme in Sprachen geschrieben werden, die stark von der Maschine auf der sie ablaufen abstrahiert sind, so ist der Einfluss des Programmierers begrenzt, wenn es darum geht, die Zielhardware optimal auszunutzen. An dieser Stelle tritt der Compiler als letzte Instanz auf, die Programme so zu optimieren, um das bestmögliche Ergebnis zu erzielen.

## 2.1 Energieverbrauch in eingebetteten Systemen

Um mögliche Energie-Einsparpotenziale voll ausschöpfen zu können, bedarf es eines eingehenden Blickes auf die Hardwaregegebenheiten. Dazu muss man das Gesamtsystem mit seinen Komponenten auf ihren Verbrauch hin untersuchen. Ein Gesamtsystem besteht im wesentlichen aus folgenden Komponenten:

- Motherboard
  - Prozessor
    - Prozessorcore
    - On-Chip SRAM
  - Off-Chip DRAM
  - I/O
- Festplatte
- Floppy-Laufwerk
- LCD/VGA
- Netzteil

Die Teilkomponenten tragen aber unterschiedlich viel zum Gesamtverbrauch bei. Die Abbildung 2.1 verdeutlicht diese Verhältnisse. Da der Einflussbereich der Software auf den Energieverbrauch, speziell des hier im Vordergrund stehenden Compilers, sich auf wenige dieser Komponenten auswirkt, werden im Folgenden nur der Prozessor und Speicher näher betrachtet. Um zu verstehen wie sich die Energiekosten zusammensetzen, muss man auf die Bauweise der Komponenten im Detail schauen. Beide, Prozessor und Speicher, sind, geht man von heutigen Standards aus, durch CMOS[1]-Schaltungen realisiert. CMOS ist heute die dominante Halbleitertechnologie für Mikroprozessoren, Speicher (SRAM und DRAM) und anwenderspezifische integrierte Schaltungen (ASIC's)[2]. Hauptvorteil der CMOS-Gatter gegenüber der NMOS oder bipolarer Technologie ist der, dass sie viel stromsparender sind und fast keinen statischen Energieverbrauch haben. Abbildung 2.3 zeigt schematisch eine solche.

[1]Complementary Metal-Oxid Semiconductors
[2]Applikation specific integrated circuit

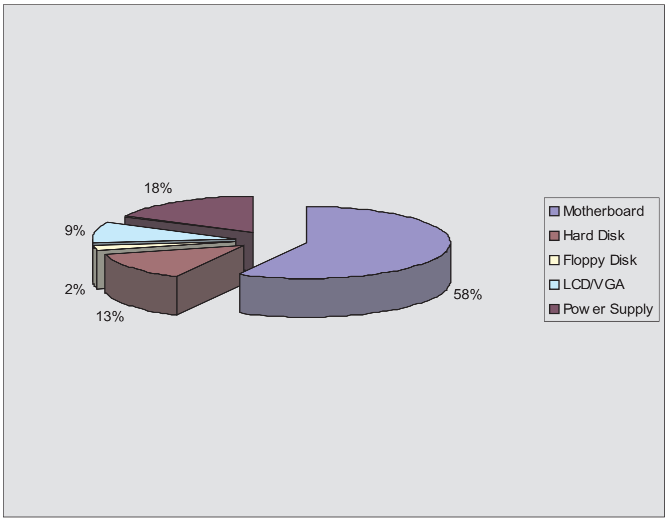



Abbildung 2.1: Energiebilanz eines mobilen Systems

### 2.1.1 Physikalische Grundlagen

Das folgende Kapitel ist dazu gedacht, einige Begriffe der Physik wie Leistung, Energie uns mit ihren physikalischen Definitionen näher zu bringen. Die Leistung $P$ ist definiert als Produkt aus der Spannung $U$ und dem Strom $I$.

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

Die Einheit der Leistung $P$ ist 1W(Watt) = 1V(Volt) * 1A(Ampere). Energie ist demgegenüber das Integral der Leistung über die Zeit. Abbildung 2.2 zeigt den Zusammenhang dieser beiden Grössen.

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

Die Einheit der Energie ist $J$(Joule)= $Ws = VAs$. Die Energie kann somit aus dem Produkt von Spannung, Strom und Zeit gebildet werden.

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

### 2.1.2 Energieverbrauch und seine Ursachen

Um das Energieverbrauchsverhalten von Prozessor und Speicher besser beeinflussen zu können, muss man sich fragen, wo und wodurch Energieverbrauch entsteht. Grundsätzlich kann man das “wo” auf der Ebene der CMOS-Schaltungen sehen und das “wobei” auf der Ebene des Prozessors oder Speichers bei der Ausführung von Instruktionen eines Programms.

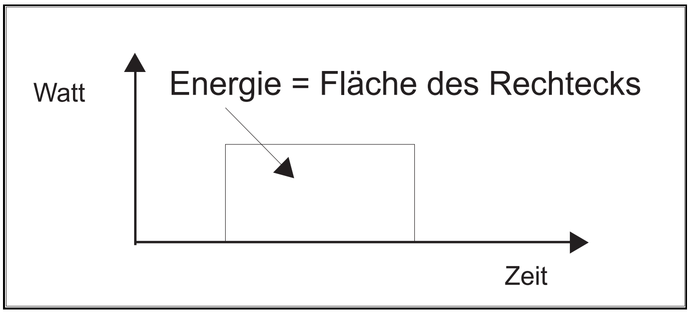



Abbildung 2.2: Leistung und Energie

### CMOS

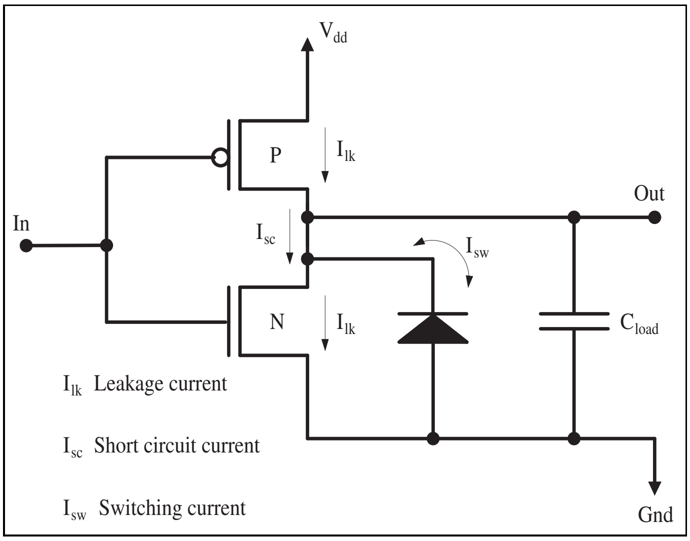



Abbildung 2.3: Inverter in CMOS-Technologie [SY96]

Betrachtet man die CMOS-Schaltungen im Detail so gibt es die drei folgenden Gegebenheiten, die Energie verbrauchen.

- Switching Power
  Beim Laden und Entladen von Kapazitäten während eines Schaltvorgangs wird ein Energiebetrag von $E = C * U^2$ aufgenommen, der zur Hälfte in Verlustwärme, wegen der in der Kapazität gespeicherten Energie, umgesetzt wird. Geschieht das mit einer Frequenz $f$, so erhöht sich die Energieaufnahme um diesen Faktor.

- Short Circuit Power

Die zeitlichen Abläufe des leitend werdenden und des zu sperrenden Transistors überschneiden sich derart, dass für kurze Zeit beide leitend sind. Diese Verlustleistung hängt von der Frequenz $f$, von der Flankendauer und von dem Quadrat der Spannung ab.

- Leakage Power
  Die statische Leistungsaufnahme in CMOS-Schaltungen ist gegenüber den anderen beiden Werten vernachlässigbar klein. Ihre Grösse von nur 1% bei aktiver Schaltung hat keinen Einfluss auf den Gesamtumsatz. Ruht die Schaltung, so ist Leakage Power die einzig messbare Grösse und trägt zu 100% zum Momentanverbrauch bei.

**Prozessor**

Geht man nun auf die Ebene des Prozessors so kann man für ihn folgende Energiebilanz aufstellen

$$E_{cpu} = \sum E_{BasicCosts} + \sum E_{Inter-InstructionCosts} \tag{2.4}$$

Dabei sind $\sum E_{BasicCosts}$ diejenigen Kosten, die ein Prozessorbefehl verursacht. Die meisten Befehle haben allerdings nicht nur Auswirkungen auf den Prozessorstrom, sondern ebenfalls auf den Peripheriestrom der Speicherbausteine. Wie wir später sehen werden, ist dieser Anteil in Abhängigkeit von der Speicherart ein entscheidender Faktor für den Verbrauch.
Die Gesamtkosten sind dann also:

$$E_{gesamt} = E_{cpu} + E_{speicher} \tag{2.5}$$

Die Inter-Instruktion-Costs $E_{Inter-InstructionCosts}$ sind Kosten, die zur erhöhten Schaltaktivität des Prozessors zwischen der Ausführung von einzelnen Befehlen führen, die verschiedene Resourcen des Prozessors benötigen. Die genauere Betrachtung dieser Werte folgt im Kapitel 3.3.

### 2.1.3 Einflussnahme auf den Energieverbrauch

Die Einflussnahme auf den Energieverbrauch kann wie im vorherigen Kapitel auf zwei Ebenen erfolgen. Die Ebene der CMOS-Schaltungen ist dabei nur für Hardware-Entwickler von Interesse. Die Möglichkeiten der Software sind auf dieser Ebene stark eingeschränkt.

Nichtsdestotrotz werden sich in der Zukunft auch auf diesem Gebiet Möglichkeiten für die Software bieten. Wie wir im Kapitel 2.1.2 gesehen haben, hat die Versorgungsspannung $U$ grossen Einfluss auf den Verbrauch einer CMOS-Schaltung. Sie kommt in allen drei Phasen einer aktiven bzw. passiven Schaltung als Zweierpotenz vor. Das Stichwort hierbei sind Prozessoren mit mehreren Versorgungsspannungen. Die Idee, die dahinter steckt, ist die Folgende. Hat man zum Zeitpunkt $t$, $n$ verschiedene Tasks mit $n$

Deadlines gegeben, so kann ein Scheduler eine Ausführungsreihenfolge so errechnen, dass alle Deadlines erfüllt und dabei eine niedrigere Versorgungsspannung ausgewählt wird.

Spannung ist also eine Grösse, die zur Reduktion des Verbrauchs beitragen kann. Die Verringerung der Frequenz wäre eine weitere, ist aber kein Ziel heutiger Hardwareentwicklung. Aus der Formel **??** wo Energie das Produkt der Spannung, des Stroms und der Zeit ist, bleiben also noch Strom und Zeit als beeinflussbare Parameter.

Diese beiden Parameter sind im entscheidenden Maße von der Software beeinflussbar. Dies geschieht im hohen Anteil durch die Güte des Programms (Algorithmenwahl, Laufzeitkomplexität) aber auch durch den Compiler, auf den im nächsten Kapitel näher eingegangen wird.

## 2.2 C Compiler

Die Entwicklung von immer komplexeren und somit auch vom Codeumfang grösseren Programmen erfolgt heutzutage nicht wie früher in Assembler, sondern in Hochsprachen wie C, C++, Java u.ä. .

Diese funktionalen, objektorientierten Sprachen haben einen wesentlich höheren Abstraktionslevel als frühere imperative Programmiersprachen wie z.B Lisp oder Pascal. Der Vorteil eines höheren Abstraktionslevels liegt klar auf der Hand. Probleme lassen sich besser auf dieser Weise darstellen. Der Nachteil ist die semantische Lücke zur Zielsprache - den Maschineninstruktionen, die auf der Zielhardware zur Ausführung kommen. Und genau diese Lücke wird von einem Compiler geschlossen, der nichts weiter ist als ein Analyse-Synthese-Programm. Ein Programm, dessen Komplexität die meisten Programme übertrifft, die er übersetzt. Eine weitere wichtige Aufgabe des Compilers ist das Melden der Fehler, die sich erfahrungsgemäss immer beim Schreiben des Quellprogramms einschleichen.

### 2.2.1 Genereller Aufbau eines Compilers

Der Aufbau heutiger Compiler wird in Phasen unterteilt. Diese Phasen umfassen alle Schritte, die nötig sind, ein Quellprogramm in Objektcode zu überführen. Unter Umständen kann es jedoch sein, dass sehr umfangreiche Quellprogramme in mehreren Dateien abgelegt sind. Diese müssen manchmal zuerst an einen Präprozessor übergeben werden, der diese zusammensetzt und eventuell zusätzlich Makros[3] expandiert.

Abbildung 2.4 zeigt eine typische Compilerumgebung. Wie man sieht, sind nach dem Compilierungsprozess oft einige Schritte nötig, um einen lauffähigen Maschinencode zu erhalten. So erzeugt der in der Abbildung 2.4 gezeigte Compiler einen Assemblercode, der im nächsten Schritt vom

[3] Kürzel, oft kleine Anweisungssequenzen

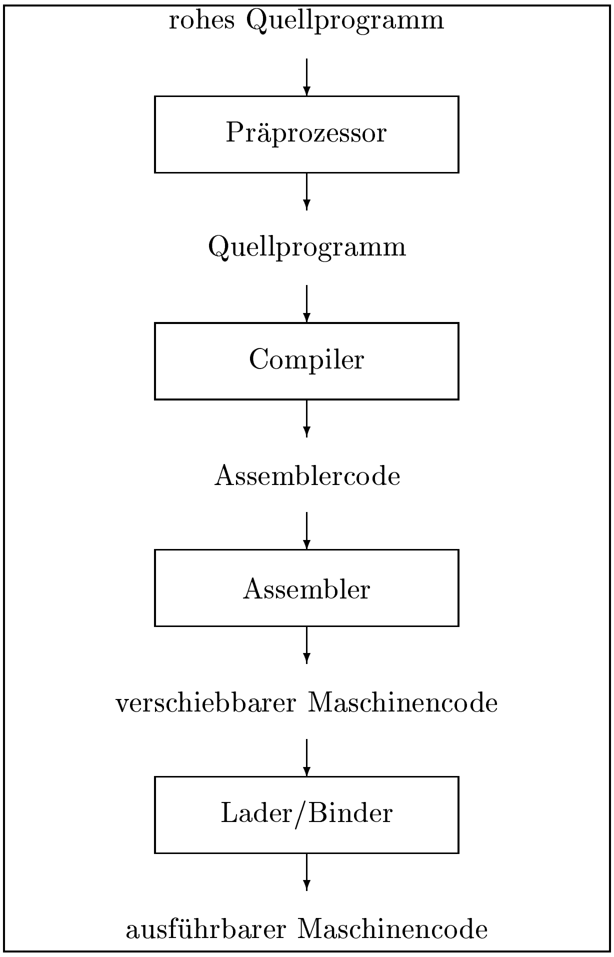



Abbildung 2.4: Compilerumgebung [AH88a]

Assembler in Maschinencode umgesetzt wird, um zum Schluss durch den Lader/Binder zum Objektcode zusammengebunden zu werden.

### Die Phasen des Compilers

Im Allgemeinen kann man die Phasen des Compilers in zwei Teile gliedern. Der erste Teil, der überwiegend aus Analysephasen besteht, unterscheidet sich von dem zweiten Teil hauptsächlich dadurch, dass er von der Quellsprache abhängt und weitgehend von der Zielmaschine unabhängig ist. Dazu zählen die

- lexikalische und syntaktische Analyse
- semantische Analyse

- Erzeugung von Zwischencode
- Erstellung der Symboltabelle

Diese Phasen werden oft in dem sogenannten vorderen Teil oder *Front-End* zusammengefasst. Die Erstellung der Symboltabelle ist keine Phase im eigentlichen Sinne, sondern eine phasenübergreifende Massnahme, die sich über alle Phasen hinwegzieht. Auch die Phasen des Back-End sind an der Erstellung und Modifikation der Symboltabelle beteiligt. In erster Linie aber nutzen sie die im Front-End erstellten Eingaben.

Der zweite Teil, *Back-End* genannt, umfasst all die Phasen, die sich auf die Zielmaschine beziehen und weitestgehend mit der Zielcodeerzeugung zu tun haben. Es sind die

- Befehlsauswahl
- Registervergabe
- Instructionscheduling

Darüberhinaus wird, was in dieser Diplomarbeit von grosser Wichtigkeit ist, der Zielcode durch den *Memory Allocator* auf unterschiedliche Speicher ( im Falle des ARM7TMI auf den On- und Off-Chip ) verteilt.

Die Zweiteilung des Compilers in das Front- und Back-End mit der Schnittstelle Zwischencode, hat sich mittlerweile im Compilerbau durchgesetzt, da es einige Vorteile mit sich bringt. Durch diesen Ansatz wird erreicht, dass eine andere Zielsprache dadurch erzeugt werden kann, indem man an ein bestehendes Front-End ein neues, für eine andere Zielmaschine spezifisches, Back-End anfügt. Das erspart die erneute Implementierung der Front-End-Phasen. Ein weiterer Vorteil besteht darin, dass man bereits auf dem Zwischencode, der das Ausgabeformat des Front-Ends darstellt, einen maschinenunabhängigen Code-Optimierer anwenden kann.

### 2.2.2 Das Front-End

Zu Anfang jedes Compilierungsprozesses steht dem Compiler, wie in Abb. 2.4 dargestellt, der Quellcode zur Verfügung. Dieses in Hochsprache geschriebene Programm ist für den Compiler nichts anderes als ein Zeichenstrom, der zunächst keinen direkt verwertbaren Informationsgehalt hat, den man in ausführbaren Maschinencode übersetzen könnte.

Hier übernimmt die erste Phase des Front-End, “lexikalische Analyse” genannt, den Transformationsprozess des Quellcodes in eine Form, die durch die nachfolgende Phase weiterverarbeitet werden kann. Mit fortschreitender Übersetzung verändert sich dann die Repräsentation des Quellcodes im Front-End, bis es schliesslich die Zwischencodedarstellung erreicht, worauf wiederrum das Back-End aufsetzt.

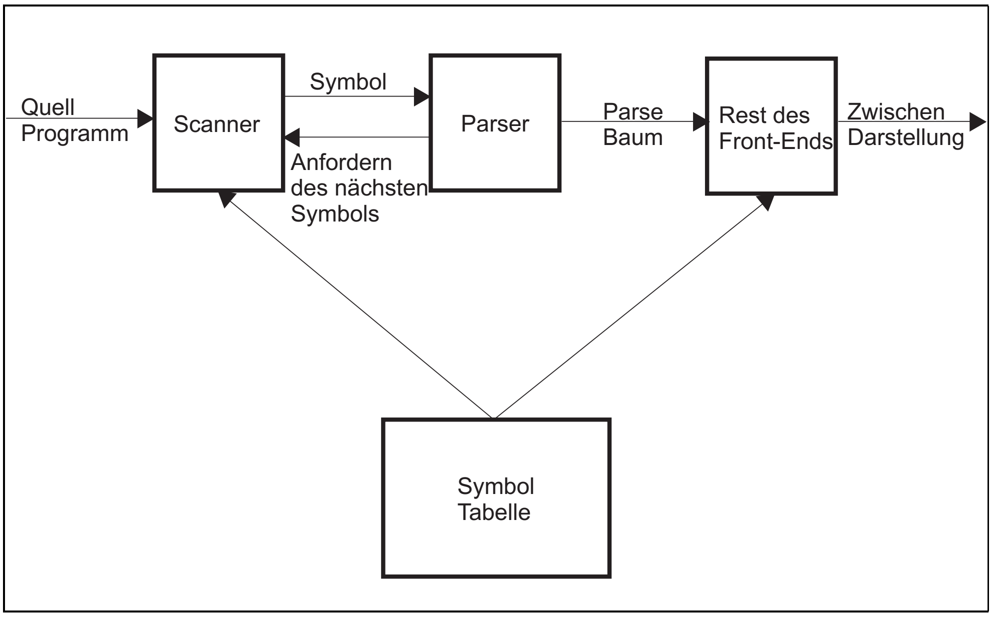



Abbildung 2.5: Front-End

### Lexikalische Analyse

Die lexikalische Analyse, engl. auch *scanning* genannt, ist eine der zeitintensivsten Phasen eines Compilers. Der scanning-Prozess liest nacheinander jedes einzelne Zeichen der Eingabe, in unserem Fall des Quellprogramms und wandelt es in einen Strom von Symbolen um. Dabei werden Kommentare und überflüssige Leerzeichen entfernt. Ein Leerraum, der vom Scanner eliminiert wurde, braucht vom Parser[4] nicht mehr behandelt zu werden, was zur vereinfachten Implementierung eines Parsers beiträgt. Die Symbole stellen bereits kleine logische Einheiten als Folge von zusammengehörigen Zeichen dar und werden Lexeme genannt. Sie werden nach folgenden Kategorien eingeteilt:

- Schlüsselwörter (`main, class, while`)
- Satzzeichen ({}`, ; usw.`)
- Bezeichner (`myvariable, ...`)
- Operatoren (`:=, >, &&`)

Manche Symbole erhalten zusätzlich einen “lexikalischen Wert”. Wenn ein Bezeichner gefunden wird, dann erzeugt der Scanner nicht nur ein Symbol, sondern er trägt das Lexem auch in die Symboltabelle ein, wenn es dort noch

[4]Syntaxanalyse engl. *parsing*

nicht enthalten ist. Der zum entsprechenden Symbol gehörige lexikalische Wert zeigt auf den Symboltabelleneintrag für das Lexem.

Ein Beispiel für die Wandlung einer Anweisung durch den Scanner stellt Abbildung 2.5 dar. Es wäre möglich, die nun folgende Phase der syntaktischen Analyse mit der lexikalischen Analyse zu koppeln, doch es gibt eine Reihe guter Gründe diese beiden Phasen aufzuteilen. Das vielleicht wichtigste Argument ist die Vereinfachung des Entwurfs. Ein weiterer ist die Effizienz der Implementierung und die Portabilität des Compilers. Besonderheiten anderer Eingabealphabete können auf den Scanner beschränkt werden, so z.B das $\uparrow$ in Pascal, das kein standardisiertes Symbol ist.

### Syntaktische Analyse

In der Phase der syntaktischen Analyse wird der nun durch den Scanner gewandelte Quellcode auf seine Struktur hin untersucht. Ihre Aufgabe besteht darin, die Symbole des Quellprogramms zu grammatikalischen Sätzen zusammenzufassen.

Jedes Programm einer Programmiersprache ist nach gewissen Regeln aufgebaut. Diese definieren die Wohlgeformtheit der Programme und werden Syntax genannt. Sie lassen sich durch kontextfreie Grammatiken beschreiben. Zu einer kontextfreien Grammatik gehören vier Komponenten:

1. $T$, dem endlichen Alphabet, über dem die zu erzeugende Sprache definiert ist (Terminalzeichen)
2. $V$, einer endlichen, zu $T$ disjunkten Menge von Hilfszeichen (Nichtterminalen)
3. $S \in V$, dem Startsymbol
4. $P$, einer endlichen Menge von Ableitungsregeln. Eine Ableitungsregel ist ein Paar $(l, r)$ (linke und rechte Seite der Regel), wobei $l \in (V \cup T)^+$ und $r \in (V \cup T)^*$ ist

Eine Grammatik beschreibt somit die hierarchische Struktur von Programmiersprachen. Die Darstellung der hierarchischen Struktur erfolgt durch einen Parse-Baum. Eine gebräuchliche interne Repräsentation dieser Darstellung ist der Syntax-Baum, dessen Darstellung kompakter ist als die des Parse-Baums.

### Semantische Analyse

Die Phase der syntaktischen Analyse überpüft das Quellprogramm auf seine semantischen Fehler und sammelt Typ-Informationen für die anschliessende Phase der Zwischencode-Generierung. Sie nutzt die hierarchische Struktur, die während der Syntaxanalyse ermittelt wurde, um die Operatoren und Operanden von Ausdrücken und Anweisungen zu bestimmen.

Ein weiteres wichtiges Element der semantischen Analyse ist die Typüberprüfung. Diese Typüberprüfung nennt man *statische Überprüfung*, um sie von der *dynamischen Überprüfung*, die während der Ausführung des Programms geschieht, zu unterscheiden. Beispiele für statische Überprüfungen während der semantischen Analyse sind:

- *Typüberprüfungen.* Anwendung von Operatoren auf kompatible Operanden, z.B. keine Subtraktion von Funktions- und Arrayvariablen.
- *Überprüfungen des Kontrollflusses.* Anweisungen, die bewirken, dass der Kontrollfluss eines Konstrukts verlassen wird, z.B. *break* in Case-Anweisungen
- *Überprüfung der Eindeutigkeit.* Eindeutige Deklaration von Objekten vor ihrem Gebrauch, keine Wiederholungen bei Aufzählungstypen
- *Namensbezogene Überprüfungen.* Bei Konventionen, wo gleiche Namen am Anfang und Ende eine Konstrukts erscheinen müssen

### Symboltabelle

Wie aus der Bezeichnung bereits hervorgeht ist die Symboltabelle eine Tabelle, in welcher der Compiler für jeden Bezeichner einen Eintrag mit allen dazugehörigen Informationen erzeugt. Diese Informationen, Attribute genannt, sind der Speicherbedarf, der Typ und sein Gültigkeitsbereich. Für Funktionen, die ebenfalls Bezeichner sind, werden darüberhinaus noch die Anzahl der Argumente, die Methode ihrer Übergabe und der Typ des Rückgabewertes eingetragen.

### Erzeugung des Zwischencodes

Der Zwischencode ist eine Darstellung des Quellprogramms, die der Assemblersprache für eine Maschine recht ähnlich ist. Es gibt mehrere Formen, die die Darstellung einnehmen kann. Die geläufigste Form ist die des *Drei-Adress-Code.* Seine allgemeine Form sieht wie folgt aus:

$$x := y \ op \ z$$

wobei die Buchstaben für Namen, Konstanten oder vom Compiler generierte temporäre Werte stehen und *op* für irgendeinen logischen oder arithmetischen Operator.
Darüberhinaus existieren noch folgende weitere Drei-Adress-Befehle:

- `x` := `op z` , wobei *op* eine unäre Operation ist.
- `x` := `y` , Kopier-Instruktion

- `goto` $L$ , unbedingte Sprung-Anweisung, wobei `L` eine Marke ist, die mit einer anderen Drei-Adress-Anweisung assoziiert wird und als nächstes auszuführen ist
- `param` $x_{1-n}$ und `call` $p, n$ , Prozeduraufrufe mit $n$ Parametern
- `x[i]` := `y` und `x` := `y[i]` , indizierte Zuweisungen
- `x` := &`y` und `x` := ∗`y` , Adress- und Zeiger-Zuweisungen

Der Name Drei-Adress-Code kommt daher, weil jede Anweisung gewöhnlich drei Adressen enthält, eine für das Ergebnis und zwei für die Operanden. Bei dieser Form der Darstellung fällt auf, dass jeder Drei-Adress-Befehl neben dem Zuweisungsoperator höchstens noch einen weiteren Operator haben darf.

Grundlage der Übersetzung in den Drei-Adress-Code stellt der bereits nach der syntaktischen Analyse vorliegende Syntax-Baum dar. Hat man Ausdrücke, die mehrere Operatoren besitzen, kann keine direkte Entsprechung im Format des Drei-Adress-Code niedergeschrieben werden, denn wie wir gesehen haben, ist dieser auf einen Operator (abgesehen von der Zuweisung) beschränkt. Dieses Problem wird durch Generieren von temporären Zwischenergebnissen gelöst. Nachfolgendes Beispiel verdeutlicht den Mechanismus:

$$\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * \mathbf{id}_4$$

Der dazugehörige Drei-Adress-Code sieht dann folgendermassen aus:

$$\begin{array}{l} \texttt{temp}_1 := \texttt{id}_4 \\ \texttt{temp}_2 := \texttt{temp}_1 * \texttt{id}_3 \\ \texttt{temp}_3 := \texttt{temp}_2 + \texttt{id}_2 \\ \texttt{id}_1 := \texttt{temp}_3 \end{array}$$

Zum Zeitpunkt, in dem der Compiler den Zwischencode erzeugt, muss er die Reihenfolge festlegen, in der die Operationen auszuführen sind. Deshalb muss der Rang der Operationen berücksichtigt werden. In unserem Fall erfolgt die Auswertung der Multiplikation vor der Addition. Die Übersetzung der Ausdrücke ist allerdings nur ein Teil der Arbeit, die die Phase der Code-Erzeugung zu leisten hat. Sie muss darüberhinaus auch Zwischencode-Befehle generieren, die den Kontrollfluss und Unterprogrammaufrufe realisieren, also typische programmiersprachliche Konstrukte. Eine graphische Darstellung von Drei-Adress-Befehlen wird Flussgraph genannt. Er spiegelt den Kontrollfluss in einem Programm wieder, wobei die Knoten die Berechnungen und die Kanten den Fluss darstellen. Dieser Kontrollflussgraph ist als Hilfsmittel für weitere Phasen der Code-Erzeugung sehr wichtig. Viele

Analyseverfahren an die sich Optimierungen des Codes anschliessen, nutzen diese Darstellung des Programms.

In diesem Zusammenhang möchte ich noch den Begriff *Basisblock* erwähnen. Er stellt eine Folge fortlaufender Anweisungen dar, in die der Kontrollfluss am Anfang eintritt und am Ende verlässt, ohne zwischendurch zu verzweigen.

Für die von mir untersuchte Optimierungsmethode *Memory Allocation* ist der Flussgraph mit seiner Einteilung in Basisblöcke die Darstellungsform des Quellprogramms, auf der alle Analysen durchgeführt werden.

### 2.2.3 Das Back-End

Das Back-End stellt nun die Phase des Compilers dar, die die *"eigentliche"* Arbeit, nämlich die der Maschinen- oder Assemblercode-Erzeugung, leistet. Als Eingabe fungiert hier die Zwischendarstellung des Front-End und die Symbolltabelle, die zur Bestimmung von Laufzeitadressen für Datenobjekte benutzt wird. Die Ausgabe des Back-End ist das Zielprogramm.

Im folgenden werden die einzelnen Phasen des Back-End: Befehlsauswahl, Registervergabe und Wahl der Auswertungsreihenfolge beschrieben.

#### Befehlsauswahl

Die Befehlsauswahl ist diejenige Phase des Back-End, die jede Anweisung des Zwischencodes einem, meist aber einer Folge von Befehlen zuordnet. Dabei spielen Eindeutigkeit und Vollständigkeit des Befehlssatzes eine wichtige Rolle. Werden beispielsweise nicht alle Datentypen in eindeutiger Weise behandelt, so muss eine Ausnahmebehandlung erfolgen. Gute Befehlsauswahl zeichnet sich dadurch aus, dass sie die Ausführungszeiten und Inanspruchnahme der Ressourcen der einzelnen Befehle berücksichtigt und die effizientesten auswählt. Beispiel einer naiven und im Gegensatz dazu einer "guten" Übersetzung der oft im Quellprogramm vorkommenden Anweisung
x := x + 1:

```
MOV := x,R0          INC x
ADD := #1,R0
MOV := R0,x
```

Wie man anhand dieses Beispiels sieht, gibt es durchaus mehrere Implementationsmöglichkeiten für eine Übersetzung, doch kann die Nichtbeachtung der Besonderheiten des Befehlssatzes zu ineffizientem Zielcode führen.

#### Registervergabe

Die effiziente Nutzung der Register des Prozessors ist für den Energieverbrauch und die Ausführungsdauer des Zielcode von enormer Wichtigkeit. Befehle mit Registeroperanden sind kürzer und schneller als solche mit Speicheroperanden. Die Zahl der Register ist jedoch stark beschränkt und schwankt vom Prozessor zu Prozessor recht stark.

Während der Befehlsauswahl wird so getan, als ob eine unendliche Anzahl von Registern zur Verfügung stehen würde. Dadurch erreicht man, das Namen im Zwischencode zu *symbolischen* Registernamen und die Drei-Adress-Befehle zu Maschinenbefehlen werden. Um nun die real zur Verfügung stehenden physikalischen Register den symbolischen zuzuordnen, muss für jede Funktion ein *Register Interferenz Graph* erzeugt werden. Bei diesem Graphen stellen die Knoten symbolische Register dar. Zwei Knoten $a,b$ werden miteinander verbunden, wenn zu einem bestimmten Zeitpunkt $a$ aktiv ist und $b$ definiert wird.

Den so erzeugten Graphen versucht man im Anschluss mit $r$ Farben einzufärben, wobei $r$ die Anzahl der tatsächlich vorhandenen physikalischen Register ist. Jede Farbe stellt ein Register dar. Die Knoten werden nun so eingefärbt, dass niemals zwei miteinander verbundene Knoten die gleiche Farbe besitzen. Ist es nicht möglich den Graphen so zu färben, so bedeutet dies, dass die Anzahl der Register nicht ausreicht und man, um ein Register frei zu bekommen, die Daten eines Registers in den Speicher zurückschreiben muss. Dieses Problem gehört wie auch einige andere im Umfeld von Compilern zu den NP-vollständigen Problemen. Diese Problematik wird im Kapitel 4.1 näher erläutert.

### 2.2.4 Optimierungen

Es gibt Compiler, die in einigen Phasen des Compilierungsprozesses bestimmte Veränderungen an der aktuellen Darstellung der Programmcodes vornehmen. Diese Veränderungen haben den Zweck, den Zielcode dahingehend zu verbessern, dass er:

- kompakteren Zielcode ergibt,
- schnellere Ausführungszeiten erreicht oder
- weniger Energie verbraucht.

Letztere werden *Power Aware Compiler* genannt. Einige Optimierungstechniken, die zum Ziel die Energiereduktion der Programme haben, werden nachfolgend vorgestellt.

#### Energieoptimierungen

- Strength Reduction: ist eine Optimierungstechnik, die einzelne Befehle die "viel" Energie verbrauchen, durch solche mit geringeren Energieverbrauchs-Werten ersetzt. Z.B. kann man bei der Multiplikation mit Faktoren $f = 2^n, n \in \mathbb{N}$, anstatt einer "teuren" `MULT`-Operation eine "billige" `SHIFT`-Operation verwenden.

- Registerpipelining[SC00]:Da Speicherzugriffe hohe Schaltaktivitäten auf den Bussen verursachen, ist es vorteilhaft ihre Anzahl zu reduzieren. Die Optimierungstechnik Registerpipelining versucht durch das Halten von Speicherwerten in freien Registern, das nochmalige Laden dieser Werte zu vermeiden.

- Instruction Scheduling: Bei sehr hohem Registerdruck, das heisst es werden mehr Register für eine Berechnung gebraucht als momentan zur Verfügung stehen (siehe Kapitel 2.2.3 Registervergabe), können durch geschickte Umordnung der Befehle (Voraussetzung die Semantik bleibt erhalten) Register frei werden. Dadurch entfällt das in solchen Fällen praktizierte *Spilling* [5].

Darüberhinaus gibt es Optimierungen, die alle Compiler standardmässig durchführen. Exemplarisch sollen hier die wichtigsten sogenannten “funktionserhaltenden Transformationen” kurz erwähnt werden.

### Standardoptimierungen

- Common subexpression elimination
  Wird ein Vorkommen eines gemeinsamen Teilausdrucks (engl. common subexpression) in gleichen oder verschiedenen Grundblöcken erkannt, kann man auf die erneute Berechnung dieses Ausdrucks verzichten, indem man dem zweiten Ausdruck den Wert des ersten zuweist. Voraussetzung für diese Zuweisung ist, dass der Wert eines Faktors in dem Ausdruck sich zwischendurch nicht verändert hat.

- dead code elimination
  Diese Optimierungstechnik versucht durch Code-Analysen herauszufinden, ob Codebereiche nie erreicht werden. Sowas nennt man passiven (engl. dead) Code. Kommt eine solche Codesequenz vor, kann sie ersatzlos gestrichen werden.

- Schleifenoptimierungen
  Da man weiss, dass Programme viel Rechenzeit in Schleifen verbringen, ist es immer lohnenswert, Code innerhalb von Schleifen zu optimieren. Zwei Vertreter dieser Art sind:

  - loop invariant code motion
    Hier wird Code von innerhalb der Schleife herausgenommen und vor die Schleife platziert. Dadurch erspart man sich $n-1$ Mal die Ausführung dieser Anweisung, wenn der Laufindex der Schleife $n$ beträgt.

[5] Auslagern der Registerinhalte in den Speicher

- Elimination von Induktionsvariablen
  Oft kommt es vor, dass eine Induktionsvariable $i$ für lineare Funktionen, die relative Adressen $t$ einer Feldkomponente berechnen, gebraucht werden. In diesem Fall kann man, wenn $i$ als Index zum Abbruch benutzt wurde, den Test auf $t$ übertragen und $i$ streichen.

# Kapitel 3

# Arbeitsumgebung

Für die Auswirkungen der Optimierungstechnik Memory Allocation auf den Energieverbrauch wurde im Rahmen dieser Diplomarbeit auf der Hardwareseite der ARM7TDMI-Prozessor und auf der Softwareseite der experimentelle “encc” Compiler, in welchen die Methode integriert werden sollte, eingesetzt.

Da Memory Allocation bestimmte Architekturmerkmale des Prozessors ausnutzt, wird an dieser Stelle der Prozessor wie auch das Evaluationboard, auf dem dieser sich befindet, vorgestellt.

## 3.1 ARM7TDMI

Der ARM7TDMI-Prozessor gehört zu der Familie der RISC[1]-Prozessoren. Seine geringe Chip-Fläche, geringer Verbrauch und die Verarbeitungsgeschwindigkeit von 117 MIPS/Watt, machen ihn besonders für mobile Anwendungen äusserst interessant. Als weitere Besonderheit unterstützt der ARM7TDMI zwei Befehlssätze. Der ARM-Instruction-Set hat eine Instruktionswortbreite von 32 Bit und umfasst 80 Kernelbefehle.

Demgegenüber hat der Thumb-Instruction-Set eine Befehlswortbreite von 16 Bit und gerade mal 36 Befehle. Diese Einschränkung hat den Vorteil einenr sehr hohen Codedichte. Als Nachteil kann sich allerdings, durch die eingeschränkte Wortbreite, die Reduktion der Adressierungsarten erweisen.

[1] Reduced Instruction Set Computer

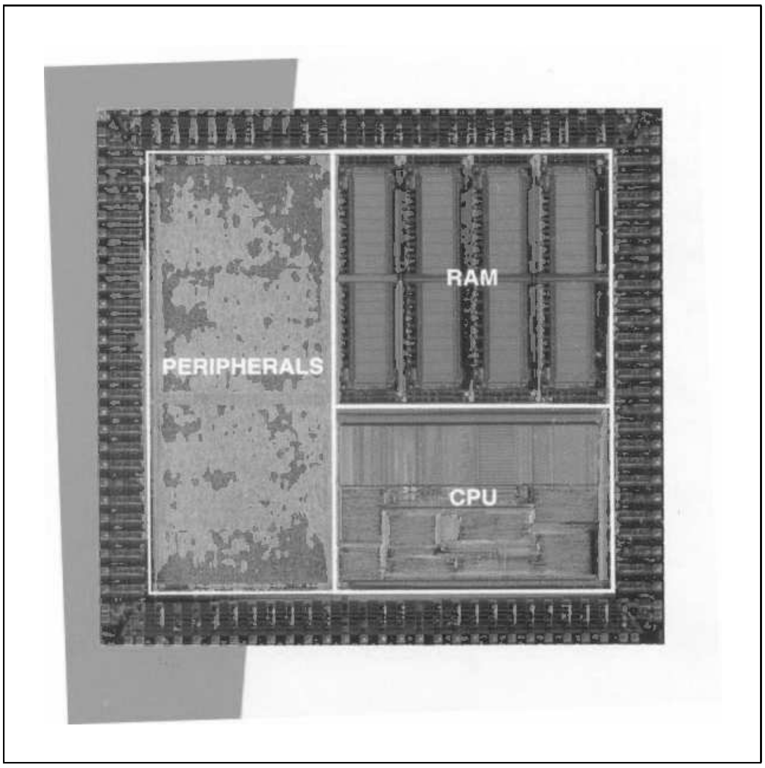



Abbildung 3.1: AT91M40400-Controller

Der “encc” Compiler generiert ausschliesslich Code im Thumb-Format, da hier das Hauptaugenmerk auf den Energieverbrauch gelegt ist. Der ARM-Instruction-Set bietet zwar Vorteile bei der Ausführungsgeschwindigkeit, ist aber im Hinblick auf den Energieverbrauch und Codedichte dem Thumb-Instruction-Set unterlegen. Hier machen sich die 32-Bit Wortbreite und damit verbundenen Schaltaktivitäten auf dem Bus negativ bemerkbar.

Grundsätzlich ist aber das Umschalten zwischen den beiden Befehlssätzen jederzeit möglich.

Der ARM7TDMI weist folgende Charakteristika auf:

- 32-Bit RISC
- 31 × 32-Bit-Register plus 6 zusätzliche Status Register
- Load-/Store Architektur
- dreistufige Instruction-Pipeline
- 32-Bit ALU
- separater Barrelshifter
- 32-Bit Multipizierwerk
- 32-Bit Adress- und Datenbus

## 3.2 Evaluationboard

Die Arbeitsumgebung des Prozessors ist ein Evaluationboard der Firma ATMEL [Atm99a]. Der AT91M40400 Microcontroller 3.2 besitzt auf seinem Chip:

- den ARM7TDMI-Prozessorcore
- zusätzliche Peripherie
- einen 4-KB-RAM Speicher

Darüberhinaus stehen extern auf dem Evaluationboard ein:

- 512-KB-RAM Speicher
- 128-KB-ROM Speicher

zur Verfügung. Weitere Eigenschaften des AT91EB01 sind:

- 16-Bit-Datenbus, 24-Bit-Adresssbus
- 2 serielle Schnittstellen
- JTAG ICE Debug Interface
- Angel Debug Monitor

Aus der Sicht der Optimierungstechnik Memory Allocation stellt der 4-KB-RAM Speicher eine wichtige Eigenschaft des Mikrocontrollers dar. Dieser kleine interne Speicher, der über den gleichen Bus wie der 512-KB-RAM Speicher angesprochen wird, ist erstens schneller und zweitens "günstiger" in seinem Energieverhalten als der 512-KB-RAM Speicher. Standardmässig wird der 512-KB-RAM zur Aufnahme der Programme und Daten verwendet.

Auch das On-Chip RAM, in diesem Falle Scratch-Pad, weisst diese Eigenschaft auf und kann sowohl Programmcode als auch Daten speichern.

## 3.3 Energieverbrauch des AT91M40400

Die Motivation für die Optimierungsmethode Memory Allocation war das Vorhandensein des Scratch-Pad Speichers in dem für diese Arbeit zur Verfügung stehenden Mikrocontroller. Aus einer früheren Diplomarbeit von Michael Theokharidis [TH00] mit dem Thema "Energiemessung von ARM7TDMI Prozessor-Instruktionen" ging hervor, dass sich die Kosten von Instruktionen wie folgt aufteilen. Diese für Instruktionen mit externen Speicherzugriffen ungünstige Energiebilanz hat zweierlei Ursachen. Es sind:

- Zyklenanzahl

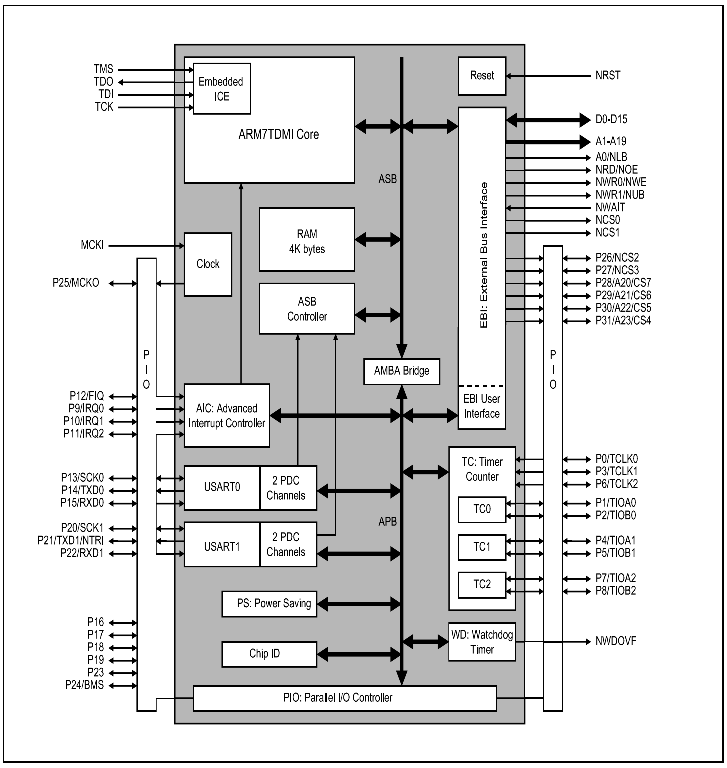



Abbildung 3.2: Evaluationboard ATMEL AT91EB01

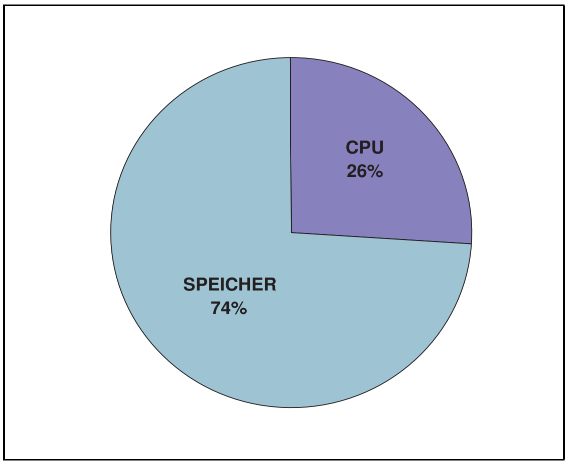



Abbildung 3.3: Durchschnittliche Energieverteilung Prozessor vs. Speicher

- Stromverbrauch externer Speicherbausteine

Man sieht, dass durch Reduktion der Speicherzugriffe sich sehr hohe Energieeisparungen erzielen lassen. Dass diese nicht immer zu vermeiden sind dürfte klar sein, deshalb ist die Wahl von Speicherarten, die diesen Anteil minimieren, vorzuziehen.

Im Falle der Speicher Off-Chip und Scratch-Pad ist der Energiebedarfsunterschied enorm. Abbildung 3.4 verdeutlicht diesen Unterschied. Zwar ist der Stromverbrauch des Prozessors für Befehle, die auf dem On-Chip-Speicher liegen, im Durchschnitt um 10-15 % höher als bei Verwendung von Befehlen, die sich im Off-Chip-Speicher befinden, doch entfallen die Gesamtkosten fast vollständig auf die Energiekosten des externen Speichers und der erhöhten Zyklenanzahl der einzelnen Instruktionen.

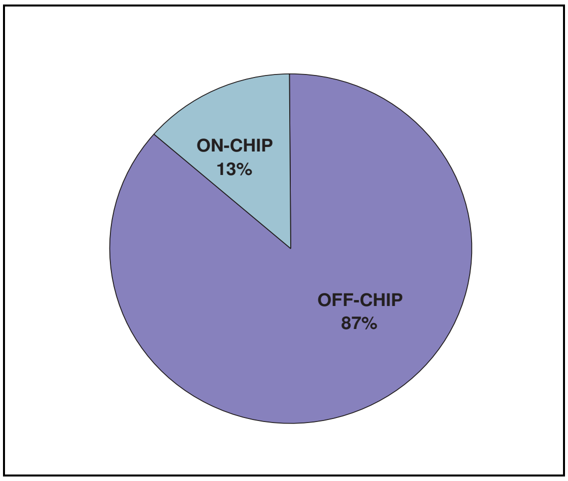



Abbildung 3.4: Durchschnittliche Energieverteilung Off-Chip vs. On-Chip

## 3.4 Trace-Analyzer

Ebenfalls wichtige Grundlage dieser Diplomarbeit ist die im Rahmen zurückliegender Arbeiten [SC00] implementierte Bewertung der simulierten Ausführung des ARM7TDMI-Prozessors durch den ARMulator. Diese Bewertung Trace-Analyzer genannt, ermittelt folgende Charakteristika des simulierten Programms:

- Energieverbrauch
- Taktzyklen
- Leistung
- benötigter Speicherplatz für:

  - Daten
  - Programm

- Anzahl der ausgeführten Instruktionen
- Anzahl der Speicherzugriffe, wobei hier die Art der Speicher (Off-Chip,On-Chip) berücksichtigt und separat ausgegeben wird.

Für die Bewertung der Energiekosten wird Modell von Tiwari [TI94] herangezogen, die die Gesamtenergie wie folgt bewertet:

$$E_P = \sum_i (B_i * N_i) + \sum_{i,j} (O_{i,j} * N_{i,j}) + \sum_{k=1}^{h} E_h \qquad (3.1)$$

Die drei Summanden der Gleichung haben folgende Bedeutung:

$\sum_i (B_i * N_i)$

$B_i$: Instruktionskosten für eine Instruktion $i$.
$N_i$: Häufigkeit, mit der $i$ ausgeführt wird.
Der erste Summand stellt die Grundkosten der einzelnen Befehle dar und wird mit dieser Summe berücksichtigt. An dieser Stelle kann ebenfalls die Operandenabhängigkeit der Befehle eingehen.

$\sum_{i,j} (O_{i,j} * N_{i,j})$

$O_{i,j}$: Mehrverbrauch an Energie beim Wechsel von Instruktion $i$ nach $j$.
$N_{i,j}$: Häufigkeit, mit der das Instruktionspaar $(i, j)$ vorkommt.
Dieser Summand gibt die Inter-Instruction-Costs wieder, die aus Schaltaktivitäten für Zustandswechsel zwischen zwei Instruktionen resultieren.

$\sum_{k=1}^{h} E_h$

Hierunter finden alle anderen Kosten ihre Berücksichtigung, z.B. pipeline stalls u.ä. Diese werden in dieser Arbeit nicht berücksichtigt.

Der Trace-Analyzer betrachtet jeden Befehl $B_i$ des simulierten Durchlaufs einzeln. Für jeden solchen Befehl gibt es in einer Datei einen Eintrag, der die Verbrauchswerte der Befehle enthält. Die Häufigkeit seines Auftretens spiegelt der Faktor $n$ wieder. Zu jedem Befehl wird ebenfalls die Summe der durchgeführten Speicherzugriffe $S_i$ gebildet. Für den gesamten Trace ergibt sich der folgende Gesamtverbrauch:

$$E_P = \sum_i (B_i * n) + S_i \qquad (3.2)$$

Der so ermittelte Wert wird als Gesamtverbrauch der Programme in den Ergebniss-Kapiteln 5.4 und 6.3 für die Bewertung der Optimierungsmethode verwendet.

Beispiel einer Ausgabe des Traceanalyzers zeigt die nachfolgende Datei.

```
OBJ/heap_sort.report2
:::::::::::::
Memory Size:
        Memoryunit   Prog          Data
        offchip      548           436
        onchip       56              0

  executed Instructions: 39157
  access to datamemory : 24532 Byte
        Memoryunit      Inst    Data    Energy/10^9
        offchip         read    2 Byte  33498   0       24.0
        offchip         read    4 Byte  0       4488    49.3
        onchip          read    2 Byte  5659    0       0.3
        offchip write 4 Byte    0       1645    41.1

CPU-Cycles : 118977
Energy : 1639.755/10^6 Ws = 545.318/10^6 Ws (Instruction)
                          +1094.438/10^6 Ws (Memory)
Power : 454.8 mW = 151.3 mW (Instruction) +303.6 mW (Memory)
```

Abbildung 3.5: Ausgabe des Traceanalyzers für Heap-Sort

# Kapitel 4

# Optimierung durch Nutzung des Scratch-Pad

Die Ausführungen der letzten beiden Kapitel haben dargestellt, wie sich der Energieverbrauch eines Programms zusammensetzt und die Möglichkeiten der Software und Hardware aufgezeigt, diesen zu reduzieren. Insbesondere die effiziente Nutzung von Registern und Besonderheiten des Befehlssatzes, können dazu beitragen. Liegen die Einsparungen an Energie dieser Methoden im einstelligen Prozentbereich, können durch optimierte Nutzung des Scratch-Pad bis zu 80 % des Gesamtverbrauchs eingespart werden.

In diesem Kapitel wird eine neue Optimierungstechnik *Memory Allocation* vorgestellt. Sie nutzt dabei den Vorteil des auf dem Kontroller AT91M40400 untergebrachten RAM-Speicher, der wie im Kapitel 3.2 beschrieben, etwa 1/8 der Energie des Off-Chip Speichers braucht. Anders als ein Cache, der unter der Verwaltung von Hardwarelogik steht, kann Scratch-Pad durch den Compiler frei genutzt werden. Diese Freiheit, Daten und Programmteile dorthin zu verschieben, macht die Speichervariante für Energieoptimierungen durch den Compiler äußerst interessant.

In weiterem Verlauf des Kapitels wird der gesamte Prozess von der Programmanalyse bis zur Verteilung der Memory-Objekte in den Scratch-Pad vorgestellt.

## 4.1 Bisherige Untersuchungen

Die Idee, die Vorteile des Scratch-Pads zu nutzen, ist nicht neu. In den Arbeiten von Sjödin [SJ98] und Panda [PA97] werden einige Verfahren zur Reduktion des Energieverbrauchs von Programmen durch Nutzung des Scratch-Pads vorgestellt.

Bei Sjödin werden nach statischer und dynamischer Analyse des Gesamtprogramms globale Daten mit der höchsten Zugriffshäufigkeit in den Scratch-Pad verlagert. Aus den Untersuchungen diese Arbeiten ging hervor, dass bereits eine statische Analyse ausreicht, um gute Ergebnisse zu erzielen. Panda untersuchte darüberhinaus die gleichzeitige Verwendung der beiden On-Chip Varianten, die des Scratch-Pad und des Caches. Dabei konnte er eine 30% Steigerung der Energieeinsparungen im Vergleich zur Nutzung der einen oder anderen Speicherart separat erreichen.

## 4.2 Programmanalyse

Die für die Programmanalyse benutzten Variablen werden in der nachfolgende Tabelle definiert und gelten auch im weiterem Verlauf der Arbeit.

| | |
|---|---|
| $BB_n$ | Basic-Block $n$ |
| $F_n$ | Funktion $n$ |
| $var_n$ | Variable $n$ |
| $E_{var_n}$ | Energieverbrauch aller Lade- und Speicherbefehle mit $var_n$ |
| $E_{off/onBB}$ | Energieverbrauch von $BB$ für das Holen der Instruktionen im Off-Chip oder Scratch-Pad (On-Chip) |
| $S_{BB_n}$ | Grösse eines $BB$ in Byte |
| $S_{F_n}$ | Grösse einer Funktion $F_n$ in Byte |
| $A_{Obj}$ | Ausführungshäufigkeit eines gegebenen Objekts $Obj$ |

Die Untersuchung der Eigenschaften von Variablen, Funktionen und Basic-Blöcken erfolgt auf Basis eines Call Graphs. Der Call Graph ist eine graphische Darstellung jedes Programms und stellt seine hierarchische Struktur dar. Er spiegelt den Kontrollfluss des Programms wieder, stellt also die Aufrufreihenfolge bis auf die Basic-Block-Ebene dar. Diese Struktur spiegelt ebenfalls die "ist enthalten in" Relation wieder, die für Kapitel 5 und 6 eine Grundlage zur Bildung von Restriktionsgleichungen für das mehrdimensionale Knapsackproblem, darstellt.

Der Rootknoten des Call Graphs ist die Funktion "main". In ihm als einzigen stehen globale deklarierte Variablen. Jede von ihm ausgehende Kante geht zu einer, durch "main" aufgerufenen Unterfunktion oder zu einem Basic-Block, der Bestandteil der Funktion "main" ist. Abbildung 4.1 zeigt eine solche Struktur.

Weitere Informationen, die die Knoten des Call Graphs enthalten, sind:

- Grösse des Objekts
- Eigenschaft
  - Funktion
  - Basic-Block
- Aufrufhäufigkeit

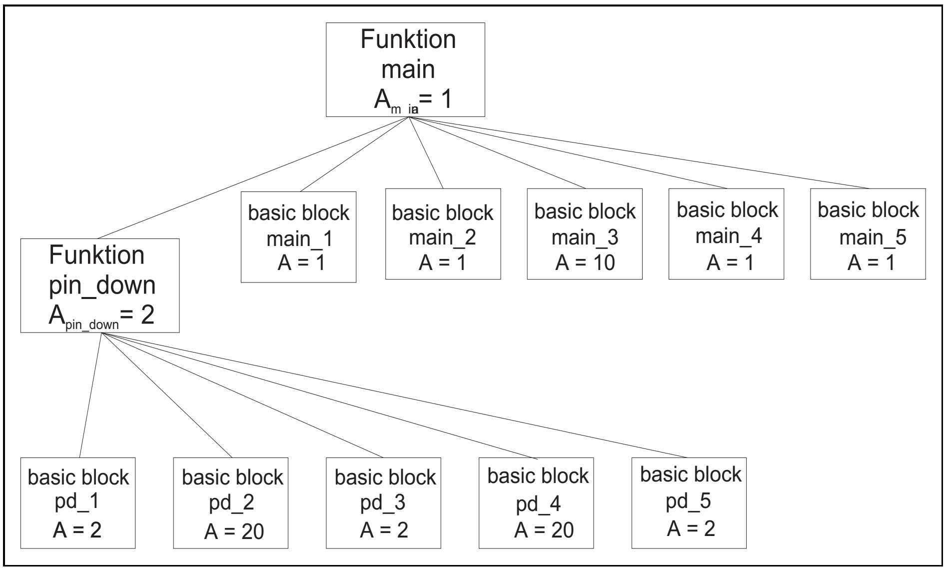



Abbildung 4.1: Aufbau eines Call Graphs ([ST01])

- eine Reihe weiterer Informationen, hier nicht relevant

Ein Basic-Block 4.2 ist hierbei eine Folge von Befehlen, der zwei wichtige Eigeschaften besitzt.

1. Der erste Befehl eines Basic-Blocks ist immer mit einer Marke, Label genannt, als Zeichen für den Beginn des Basic-Blocks versehen. (Das Label wird als Sprungziel bei bedingten oder unbedingten Verzweigungen des Kontrollflusses benutzt.)

2. Innerhalb eines Basic-Blocks $BB_i$ gibt es nur dann eine Sprunganweisung $stmt_{jump}$, wenn sie die letzte Anweisung des Basic-Blocks ist. Hat ein $BB_i$ am Ende keine Sprunganweisung, dann muss der nachfolgende Befehl des Programms $stmt_{next}$ Ziel eine Spunganweisung sein, sonst würde $stmt_{next}$ zu $BB_i$ gehören usw.

Diese beiden Eigenschaften haben zur Folge, dass, wenn ein Basic-Blocks betreten wird, alle in ihm enthaltenen Befehle einmal sequentiell ausgeführt werden und mit dem letzten Befehl $stmt_j$ der Basic-Block verlassen wird. Dabei kann das Ziel ein anderer Basic-Block oder die Sprungmarke *label* und somit $stmt_1$ sein. Ist dies der Fall so muss $stmt_j$ ein konditionaler Sprung sein, andererseits wird mit dem nachfolgendem Befehl das Programm fortgesetzt.

Um nun Variablen, Funktionen und Basic-Blöcke in den Scratch-Pad verschieben zu können, bedarf es einer genauen Analyse aller in Frage kommenden Objekte. Die Analyse bezieht sich auf die im Call Graph enthaltenen

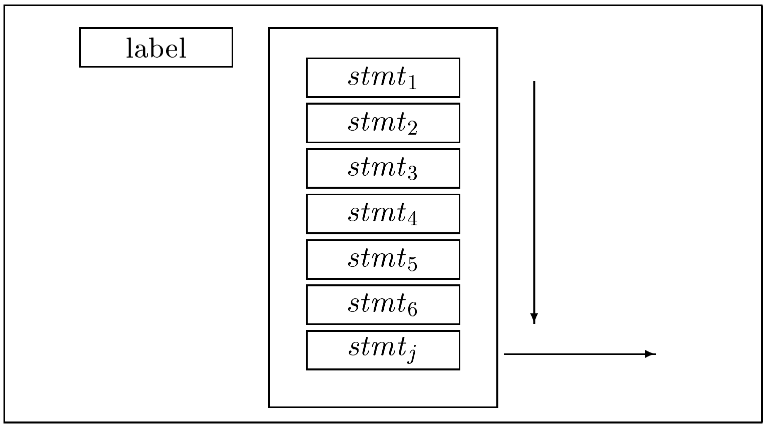



Abbildung 4.2: Definition eines Basic-Blocks

Informationen über Grösse und Ausführungshäufigkeit. Sie erfolgt statisch, d.h. es findet keine Berücksichtigung des dynamischen Laufzeitverhaltens statt. Darüber hinaus wird die Datenabhängigkeit für den Verbrauch eines Befehls ebenfalls nicht berücksichtigt. Diese Informationen dienen der Berechnung der Energieeinsparpotenziale $E_{save}$ der einzelnen Speicher-Objekte und werden nach der Formel 4.1 berechnet.

$$\begin{aligned} E_{save} &= A_X(E_{on} - E_{off}) \ - \ Offset_X \\ \text{für} \quad X &\in \{BB_i \mid 0 \leq i < n\} \cup \{F_j \mid 0 \leq j < m\} \end{aligned} \tag{4.1}$$

Da der Parameter *Offset* in der oben stehenden Gleichung von der Art des Objektes abhängig ist, wird er in den jeweiligen Kapiteln einzeln behandelt.

Alle $E_{save}$-Werte der Speicherobjekte werden anschließend in Bezug zu ihrer Grösse gesetzt, um so die energieeffizientesten bezogen auf eine Platzeinheit, hier ein Byte, zu bekommen. Die Berechnung der sich verändernden Grösse der Objekte, die je nach Art ebenfalls unterschiedlich ausfallen kann, wird auch in den betreffenden Kapiteln für Variablen, Funktionen und Basic-Blöcke erklärt.

### 4.2.1 Funktionen und Variablen

Durch die Eigenschaft des Scratch-Pad Daten und Programm oder Teile davon aufnehmen zu können, ist man in der Lage, auch Funktionen und Variablen darin unterzubringen.

Bei Funktionen bedarf es keinerlei Veränderungen des Codes um sie in den Scratch -Pad verschieben zu können. So ist ihr Energieersparniss laut Formel 4.1 gleich der Summe der $E_{save}$-Werte, der in ihm enthaltenen BB. Der Offset-Parameter ist aus oben erwähnten Gründen gleich Null. Seine Grösse bleibt ebenfalls erhalten. Variablen bedürfen genauso wenig einer

Veränderung und können direkt in den Scratch-Pad gemappt werden. Die Situation bei den Basic-Blöcken stellt sich dagegen grundverschieden dar und wird im folgendem Kapitel vorgestellt.

### 4.2.2 Basic-Blöcke

Genau wie Funktionen sind Basic-Blöcke Teile des gesamten Programms. Ihre Verschiebung in den Scratch-Pad erfordert jedoch im Gegensatz zu Variablen und Funktionen einige Modifikationen des Codes und die Anpassung des Grösseparameters, d.h. nach vollzogener Verschiebung entsteht ein $BB$ mit veränderter Grösse mit zusätzlichen Anweisungen.

Durch die Definition des $BB$ in Abbildung 4.2 gibt es grundsätzlich drei Möglichkeiten, die man unterscheiden muss, wenn man einen $BB$ verschieben möchte. Diese Unterscheidung richtet sich nach dem letzten Befehl des betreffenden Basic-Blocks. Es kann sein:

1. eine Sprunganweisung
   - konditionaler Sprung
   - nicht-konditionaler Sprung
2. eine Anweisung, die keine Verzweigung des Programmablaufs verursacht (es wird mit dem nächsten Befehl des Programms fortgesetzt, d.h. dem ersten Befehl des nächsten $BB$).
3. Sonderbefehle wie: MOV PC,LR oder POP<reglist, PC>.

Die Veränderung der Grösse entsteht durch die notwendige Hinzunahme von zusätzlichen Sprunganweisungen und mit der Besonderheit des ARM-Befehlssatzes bei Sprüngen. Alle Sprungbefehle bis auf Aufrufe von Unterfunktionen, die mit BL realisiert werden, werden durch Sprungbefehle, die die Zieladresse relativ zum PC adressieren, mit einem maximalen Sprungbereich von 256 Speicherzellen implementiert. Diese Einschränkung ergibt sich durch den Thumb-Mode, der mit der Befehlswortbreite von zwei Byte keinen Platz für grössere Adressräume als besagte 256 bietet. Für Sprünge vom Scratch-Pad in den Off-Chip und umgekehrt müssen Adressen angesprungen werden, die sich mit der Einschränkung auf $2^8 = 256$ (8 Bit für die Adresse) Speicherzellen nicht mehr realisieren lassen. Deshalb müssen alle (2 Byte) Sprünge in "long branch with link" (BL *label*), übersetzt werden. Die Besonderheit dieses Befehles im Thumb-Mode liegt darin, dass er Adressräume von bis zu 4 MB adressieren kann. Dieser Vorteil wird mit dem Nachteil erkauft, dass dieser Befehl mit zwei Thumb-Instruktionen und somit 4 Byte implementiert wird.

Das Hinzufügen zusätzlicher Befehle hat ebenso Einfluss auf den Energieverbrauch des einzelnen $BB$ und somit auch des gesamten Programms, wie auch auf die seiner Grösse. Wie sich dieser Verbrauch darstellt, wird für die oben angeführten Fälle nachfolgend betrachtet.

### Konditionaler Sprung

Hat ein $BB$ als letzten Befehl einen konditionalen Sprung, so stellt sich die Situation folgendermassen dar:

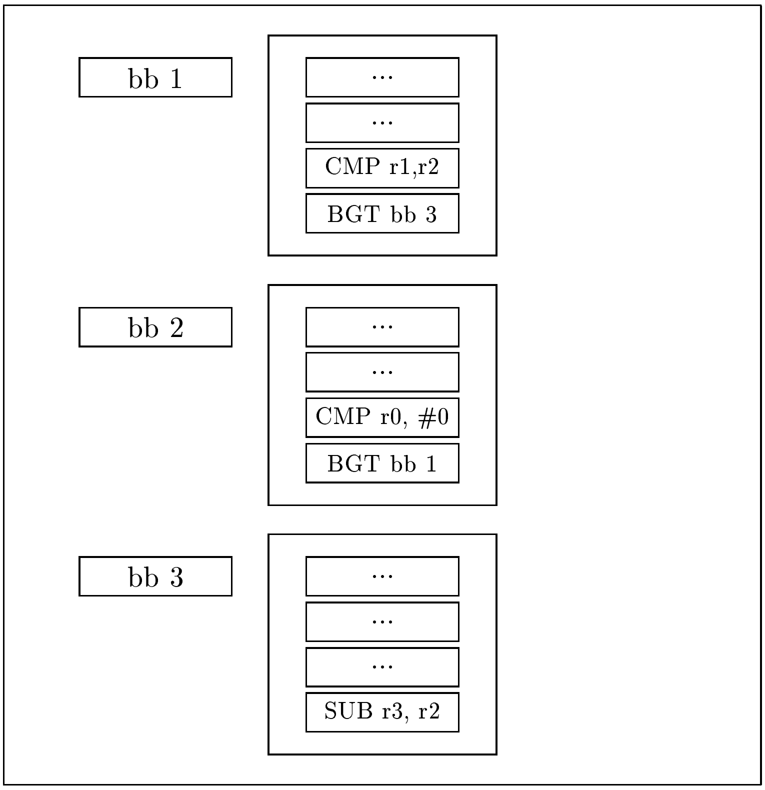



Abbildung 4.3: Konditionaler Sprung - Speicherlayout: Single Memory

Der zu verschiebende $BB$ ist $bb2$. Es sollen seine Grösse, Energieverbrauch und der Parameter *Offset* (s. 4.1) berechnet werden, wenn er in den Scratch-Pad verschoben wird. Der Kontrollfluss dieses Beispiels, geht man von $bb2$ aus, verzweigt im Falle $r0 > 0$ nach $bb1$, ansonsten wird $bb3$ ausgeführt. Um diesen zu erhalten, müssen zwei Maßnahmen getroffen werden:

1. der Übergang vom $bb1$ nach $bb2$, der hier bei nicht erfüllter Bedingung vollzogen wird und ohne weiteren Kodierungsaufwand durch Überspringen der Branch-Anweisung realisiert wird, muss sichergestellt werden.

2. Befindet sich $bb2$ im Scratch-Pad, so müssen die Ansprünge der Folgeziele, in diesem Fall $bb1$ und $bb3$, realisiert werden.

In beiden Fällen bedeutet diese Maßnahme zusätzlichen Code, da die

Übergänge von $bb1$ nach $bb2$ (Fall 1) und $bb2$ nach $bb1$ und $bb3$ mit Branch Link-Befehlen kodiert werden müssen.

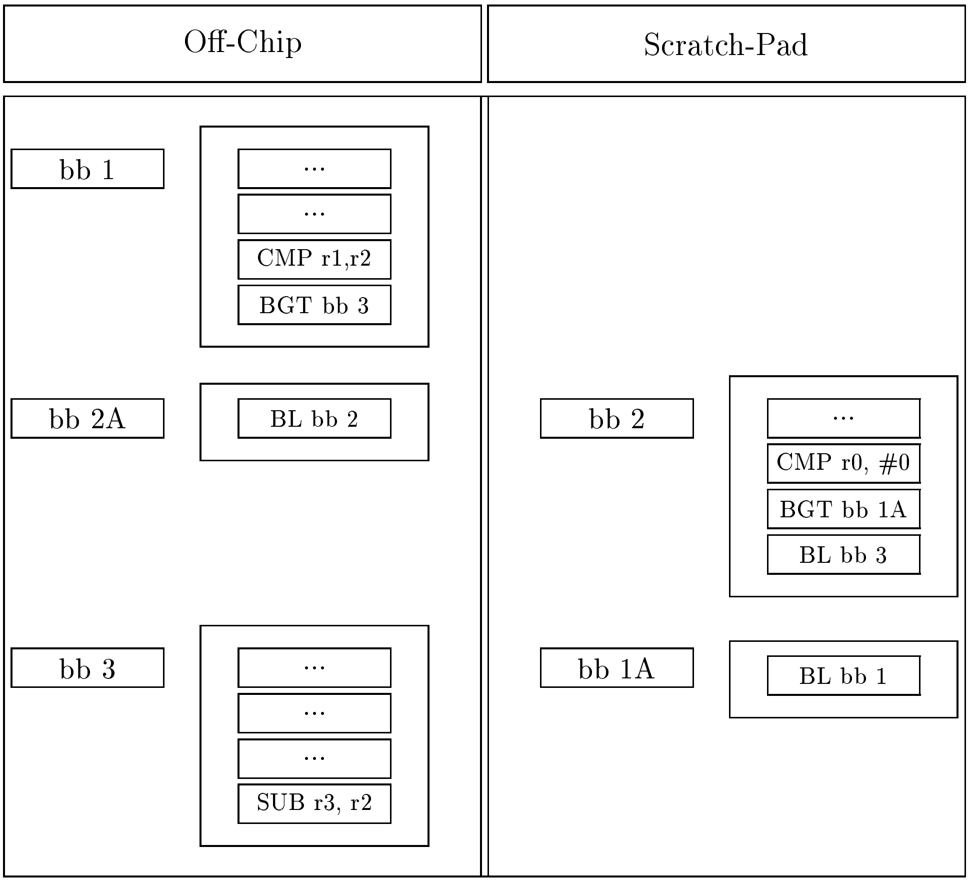



Abbildung 4.4: Konditionaler Sprung - Speicherlayout: Multiple Memory

Wie man aus den Abbildungen 4.3 und 4.4 ersehen kann, bedarf es bei konditionalen Sprüngen zweier BL im Scratch-Pad und eines im Off-Chip mehr als bei der Speicherart Single Memory. Darüberhinaus muss jeder Branch-Befehl, der das Label $bb2$ als Ziel hat, dahingehend geändert werden, dass das Ziel auf den neu erzeugten $BB\quad bb2A$ im Off-Chip gesetzt wird.

Nach diesen Erkenntnissen kann zusammenfassend für die Grösse, Energiebedarf und Offset folgendes festgestellt werden:

- Der Platzbedarf im Scratch-Pad erhöht sich um 2 BL-Befehle, also insgesamt 8 Byte. Der zusätzliche Platz für den Off-Chip-Sprung von 4 Byte ist hier nicht relevant.

$$S_{bb_2} = S_{bb_2} + 8 \tag{4.2}$$

- Der Vorteil beim Energieverbrauch gegenüber der Ausführung im Off-Chip ist laut Formel 4.1 zu berechnen. Der Offsetanteil ist dabei:

$$Offset_{bb_2} = 2 \cdot A_{bb_2} \cdot E_{on}(BL) + A_{bb_1} \cdot E_{off}(BL) \qquad (4.3)$$

### Nicht-konditionaler Sprung

Ist der letzte Befehl eines $BB$ ein nicht-konditionaler Sprung, so gestaltet sich seine Verschiebung einfacher als bei einem konditionalen Sprung. Der Kontrollfluss zum im Speicher nachfolgenden BB ist auf jeden Fall unterbrochen, d.h. es wird nie der nachfolgender $BB$ ausgeführt. Die Weiterführung des Kontrollflusses zu einer anderen Stelle des Programms (z.B. Unterprogrammaufruf, Case-Anweisung) wird mit einem Branch-Befehl "B *label*" realisiert. Diese Situation zeigt die Abbildung 4.5:

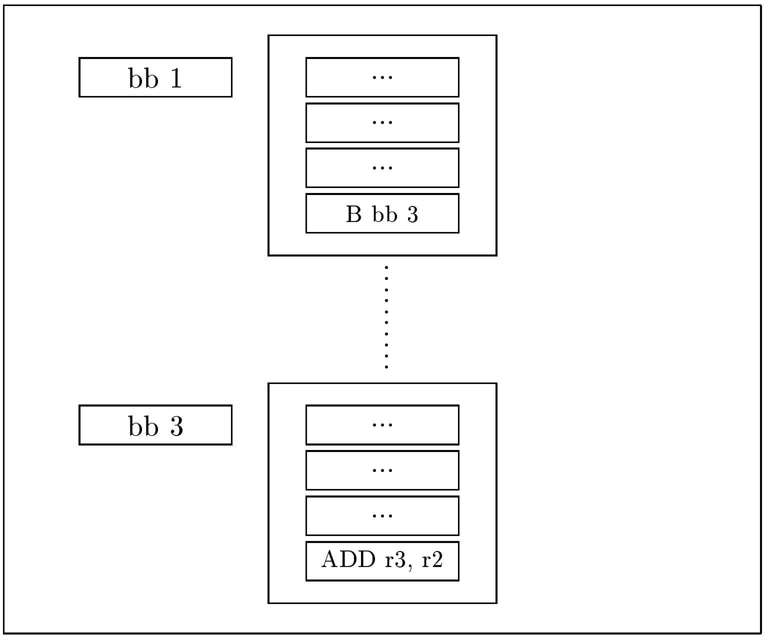



Abbildung 4.5: Nicht-konditionaler Sprung - Speicherlayout: Single Memory

Die Herausnahme des $bb1$ aus dem Off-Chip wird genauso realisiert wie bei konditionalem Sprung durch das Hinterlassen eines neuen Labels $bb1A$ mit einem BL $bb1$, der den Ansprung in den Scratch-Pad realisiert. Die Aufgabe des Rücksprungs in den Off-Chip lässt sich dann einfach durch das Ersetzen des "B $bb3$" durch "BL $bb3$" bewerkstelligen. Der Code muss wie in Abbildung 4.6 angepasst werden.

Ebenfalls wie im Falle des konditionalen Sprungs müssen alle Label die zuvor $bb1$ hiessen, auf $bb1A$ gesetzt werden.

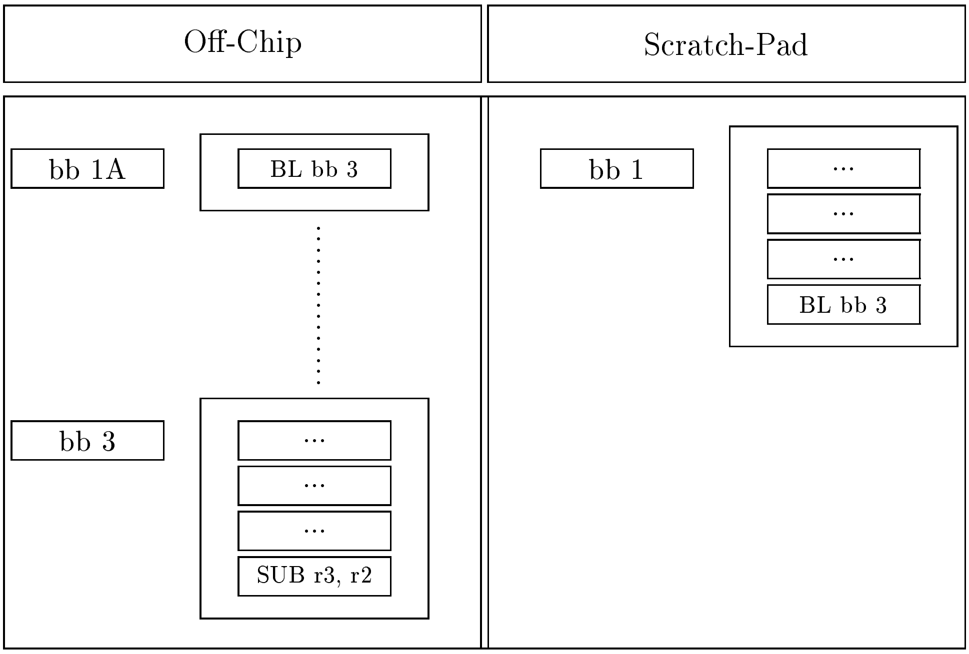



Abbildung 4.6: Nicht-konditionaler Sprung - Speicherlayout: Multiple Memory

Der benötigte Platz im Scratch-Pad ist durch den Austausch des Befehls "B"durch den Befehl "BL" lediglich um 2 Byte grösser geworden.

$$S_{bb_1} = S_{bb_1} + \text{ 2 Byte} \tag{4.4}$$

Die Abschätzung der Energieeinsparung durch Verschiebung eines solchen $BB$ in den Scratch-Pad lässt sich durch die Formel 4.1 zzgl. des folgenden Offsets angeben:

$$Offset_{bb_1} = A_{bb_1} \cdot E_{on}(BL) - A_{bb_1} \cdot E_{on}(B) + A_{bb_1} \cdot E_{off}(BL) \tag{4.5}$$

Der negative Wert $A_{bb_1} \cdot E_{on}(B)$ entsteht durch den Wegfall des Branch-Befehls aus dem urspünglichen $BB$. In der Formel 4.1 geht jedoch der vollständige $BB$ in die Berechnung mit ein $(A_X(E_{on} - E_{off}))$, was bei nicht-konditionalem Sprung nicht der Fall ist.

### Sonstiger Befehl

Mit *Sonstiger* ist hierbei ein Befehl gemeint, der keine Verzweigung des Kontrollflusses verursacht. Nach Ausführung dieses Befehls wird das Programm mit der nächsten Anweisung im nachfolgenden BB fortgesetzt.

Die Analyse und Anpassung des Codes gestalten sich ähnlich wie bei nicht-konditionalen Sprüngen, was Abbildung 4.7 verdeutlicht.

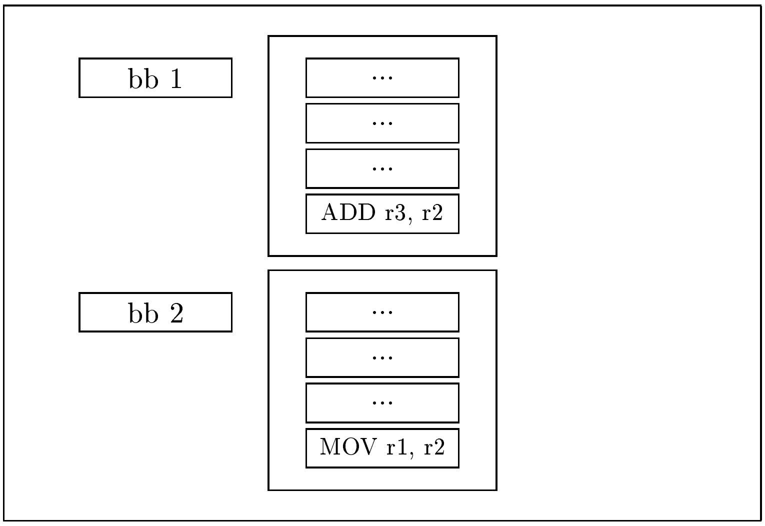



Abbildung 4.7: "Sonstiger" Befehl - Speicherlayout: Single Memory

Wird nun $bb1$ in den Scratch-Pad verschoben, muss lediglich sichergestellt werden , dass Ansprünge zum Label $bb1$ das richtige Ziel finden und der Rücksprung zum $bb2$ stattfindet. Den ersten Teil der Aufgabe realisiert der neue BB $bb1A$, den zweiten Teil der neu eingefügte Befehl BL am Ende von $bb1$ mit Ziel $bb2$.

Die Veränderung des Codes nach erfolgter Verschiebung zeigt Abbildung 4.8.

Die Grösse des Speicherobjektes im Scratch-Pad ist folglich:

$$S_{bb_1} = S_{bb_1} + \text{ 4 Byte} \tag{4.6}$$

und der Offset Anteil der Gleichung 4.1:

$$Offset_{bb_1} = A_{bb_1} \cdot E_{on}(BL) + A_{bb_1} \cdot E_{off}(BL) \tag{4.7}$$

## 4.3 Auswahl geeigneter Objekte

Die Auswahl derjenigen Speicherobjekte mit dem grössten Einsparpotenzial in Bezug auf seine Grösse erfolgt nach der Berechnung der $E_{save}$ und $S$-Werte für alle in Frage kommenden Objekte mit dem bekannten Knapsack-Ansatz. Er wird im Kapitel 5 für den eindimensionalen und Kap. 6 für den mehrdimensionalen Fall ausführlich vorgestellt.

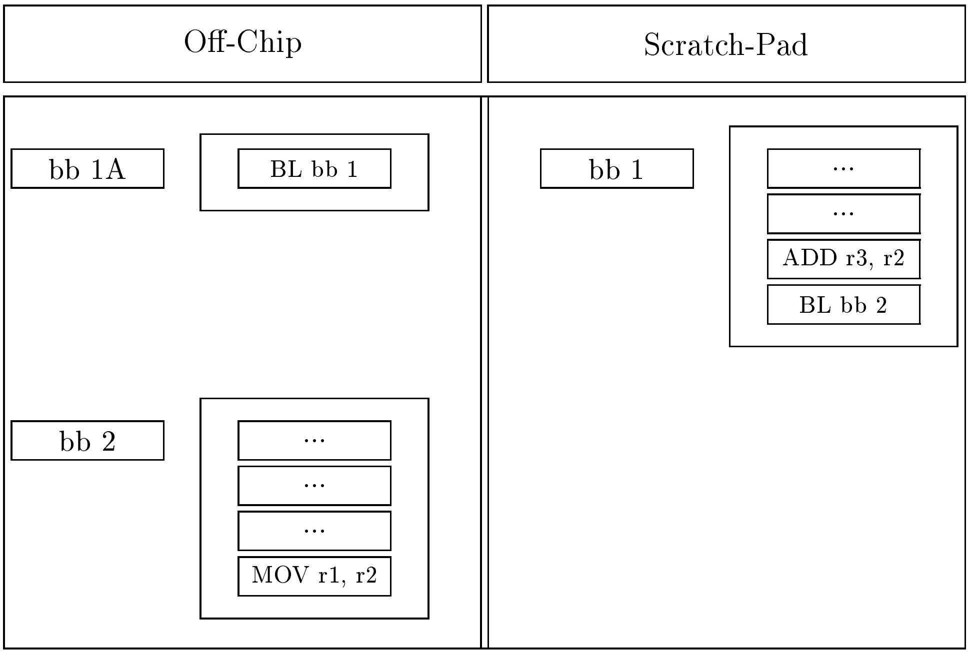



Abbildung 4.8: "Sonstiger" Befehl - Speicherlayout: Multiple Memory

## 4.4 Verteilung von Memory-Objekten

Nach vollzogener Anpassung des Assembler-Codes müssen die Memory-Objekte auf die verschiedenen Speicher verteilt werden, um Aussagen über den Einfluss dieser Änderungen machen zu können. Dieser Schritt erfordert eine Modifikation einiger Programmteile, die für die Verteilung des Programms auf die verschiedenen Speicher zuständig sind.

Dazu zählt die Änderung einer bestehenden Library und das Hinzufügen eines eigens dafür geschriebenen Initialisierungscodes, der beim Linken dem gesamten Programm hinzugefügt wird. Er ist dann zur Laufzeit dafür zuständig, die Daten und Programmteile in die verschiedenen Speicherbereiche zu kopieren. Die Verteilung des Programmcodes erfolgt mit Hilfe einer Datei. Sie gibt die drei möglichen Bereiche des Speichers an. Es sind im einzelnen:

1. Flash-ROM-Speicher, der bei der Adresse 0x00400000 beginnt,

2. Scratch-Pad (32-Bit-RAM) an der Adresse 0x00300000,

3. externer Speicher (16-Bit-RAM) bei Adresse 0x00500000

Ein Beispiel für eine Speicheraufteilungsdatei (**??**).

```
FLASH 0x004000000 0x00400000 {
   FLASH 0x00400000
   {
       *(+RO)
   }
   32BitRAM 0x00300000
   {
       test.o (+RO)
       test.o (Daten1,+RW)
   }
   16BitRAm 0x00500000
   {
       * (+RW,+ZI)
   }
}
```

# Kapitel 5

# Eindimensionales Knapsack-Problem

Im vorhergehenden Kapitel wurden alle in Frage kommenden Speicherobjekte errechnet. Sie sind nach der Analysephase mit zwei Parametern ausgestattet. Es ist zum einen die Grösse und zum anderen das Energie-Einsparpotenzial. Um den grösst möglichen Einspareffekt zu erzielen, muss man für eine fest vorgegebene On-Chip-Grösse der Zahl $b \in \mathbb{N}$ eine Auswahl der Speicherobjekte treffen, die den maximalen Nutzen bringen. Diese Problemstellung ist in der Informatik unter dem Namen Knapsack als NP-vollständiges Optimierungsproblem bekannt. Die formale Darstellung dieses Problems ist wie folgt definiert [WE00].

**Definition:** Das Knapsack-Problem besteht in der Aufgabe, für $a_1, ..., a_n, c_1, ..., c_n, b \in \mathbb{N}$ 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 $W = c_1x_1 + ... + c_nx_n$ maximiert. Für die Entscheidungsvariante des Knapsackproblems ist zusätzlich ein $d \in \mathbb{N}$ gegeben. Es soll nun entschieden werden, ob es einen Vektor $(x_1, ..., x_n)^n \in \{0,1\}$ mit $a_1x_1+...+a_nx_n \leq b$ und $c_1x_1+...+c_nx_n \geq d$ gibt.

Auf das hier vorliegende Problem übertragen, ist $a \in \mathbb{N}$ die Grösse der Speicherobjekte, $c$ das Energieeinsparpotenzial $E_{save}$ und $b$ die Grösse des Scratch-Pad-Speichers.

Das Knapsack-Problem ist ein ganzzahliges (da $x_i \in \{0,1\}$ sogar binäres) Optimierungsproblem. Probleme lassen sich in der Informatik in bestimmte Kategorien einteilen. Diese Kategorien werden im nächsten Kapitel näher beleuchtet.

## 5.1 Exkurs in die Komplexitätstheorie

Auch im Sinne der Energieersparnis ist man bei der Lösung von Problemen stehts bemüht, effiziente Algorithmen zu entwerfen. Denn oft sind ineffiziente Lösungen nicht mehr wert als gar keine. Man ist darin übereingekommen, dass Probleme dann effiziente Lösungen haben, wenn ihre Berechnung nicht mehr als polynomiell viele Rechenschritte benötigt, oder anders ausgedrückt, ihre obere Zeitschranke, die Zeit also die man für ihre Berechnung braucht, sich nach oben hin durch einen polynome beschränken lässt. Die Frage, die sich hier stellt, ist: Gibt es für alle Probleme polynomielle Algorithmen ? Die Antwort ist leider "nein". Grundsätzlich gibt es zwei Klassen von Problemen [WE92]. Es sind:

1. Klasse P, die alle die Probleme enthält, die sich von einer deterministischen Turingmaschine[1] in polynomieller Rechenzeit lösen lassen,

2. Klasse NP aller anderen Probleme, für die es keine polynomielle Lösung gibt

Es gibt mittlerweile über 2000 NP-vollständige Probleme, darunter das bereits in Kapitel 2.2.3 Registervergabe angesprochene Färbbarkeitsproblem in Graphen.

## 5.2 Lösungsansätze

Es ist vollkommen unbefriedigend, naive Algorithmen für solche Probleme zu schreiben, deren Zeitverhalten sich exponentiell zur Eingabegrösse verhält. Schon für kleine Eingabegrössen kann die Rechenzeit alle vertretbaren Zeitschranken sprengen und eine inakzeptable Antwortzeit erreichen. Auch schneller werdende Prozessoren bringen hier keine nennenswerten Verbesserungen. Bei Verdopplung oder gar Verzehnfachung der Rechenleistung reicht die Hinzunahme einiger wenige Elemente mehr in die Problemmenge aus, um die Rechenzeit auf die gleichen Werte anzuheben wie zuvor. Und exponentieller Rechenzeitbedarf ist keinesfalls die obere Schranke für NP-vollständige Probleme. Es existieren Probleme, die zu ihrer Lösung $2^{2^n}$ Schritte benötigen, d.h. dass sie auch schon für $n \geq 10$ praktisch unlösbar sind.

Die Problematik, die sich daraus ergibt, ist, dass es viele praxisrelevante Probleme gibt und man trotz ihrer Komplexität auf vertretbare Rechenzeit angewiesen ist.

Anfang der 80er Jahre erkannte man immer mehr die Praxisrelevanz der Komplexitätstheorie als immer mehr Probleme in den Bereichen Datenbanken, Betriebssysteme, VLSI-Entwurf mit ineffizienten Algorithmen gelöst

[1] Ein uniformer, sequentieller Rechner in der theoretischen Informatik, der deterministisch arbeitet

werden mussten und man sich fragte, ob es nicht bessere Lösungsansätze gibt.

Die Forschung auf diesem Gebiet ergab eine Klassifizierung der Lösungsalgorithmen in drei Klassen:

1. Pseudopolynomielle Algorithmen, die für Eingaben, die aus kleinen Zahlen bestehen, effizient sind.

2. Algorithmen, die im *worst case* exponentielle Rechenzeit haben, für viele Eingaben aber schnell stoppen.

3. Polynomielle Algorithmen, die definitionsgemäss effizient sind, aber nicht in allen Fällen das beste Ergebnis liefern, wozu die heuristischen Algorithmen zählen.

In den nächsten drei Unterkapiteln werden die drei Lösungsklassen mit je einem Vertreter vorgestellt und die Vor- und Nachteile gegeneinander abgewogen.

### 5.2.1 Dynamisches Programmieren

Die Methode des dynamischen Programmierens fällt in die Klasse der pseudopolynomiellen Algorithmen. Pseudopolynomiell bedeutet in diesem Zusammenhang, dass diese Algorithmen für kleine Eingabedaten stets zu schnellen und optimalen Lösungen führen. Die Einschränkung der Eingabe auf kleine Werte ist hier allerdings ein grosser Nachteil. Praxisrelevante Probleme übersteigen in ihrer Eingabegrösse die bei dieser Methode noch akzeptablen Werte.

Dieser Ansatz versucht, durch das Lösen von Subproblemen mit eingeschränkten Parametern, die Lösung des Gesamtproblems zu erhalten. Diese Einschränkungen, beziehungsweise ihre Anzahl, richten sich nach der Grösse der Eingabeparameter. Bei dem Knapsack-Problem wären das die Anzahl der Objekte $n$ und die Kapazität $b$ des Knapsacks.

Die Laufzeit des Lösungsansatzes wird somit mit $O(n * b)$ nach oben hin abgeschätzt. Wie man an der Formel sehen kann, kommt es bei der Laufzeit stark auf den zweiten Faktor $b$ an. Das Wachstum der Funktion ist linear, was augenscheinlich dem exponentiellem Charakter der NP-vollständigen Probleme widerspricht, aber genau in $b$ kann sich der exponentielle Faktor verbergen.

### 5.2.2 Branch and Bound-Methode

Die Methode, die Lösungsmenge durch den Branch und Bound-Baum zu suchen, zählt zu den heuristischen Methoden. Sie ist "naiven" Heuristiken durch die Einschränkung des Suchbereichs überlegen. Die Einschränkung geschieht dabei durch einfache Tests, dessen Laufzeitverhalten bei $O(n)$ liegen. Durch diesen Overhead kann die zur Lösung benötigte Laufzeit $O(n \log n + kn)$ betragen, wobei $k$ die Anzahl der Knoten im Lösungsbaum ist. Wie man weiss, hat der vollständig besetzte binäre Baum $2^n - 1$ Knoten. Somit kann der Summand $kn$ maximal $2^{n+1} - n$ werden. Daraus folgt für die Gesamtlaufzeit eine obere Schranke von $O(n \log n + 2^{n+1} - n)$.

Eine allgemeingültige Aussage über die Laufzeit für ein gegebenes Problem $P$ lässt sich somit, wie man sieht, nicht machen. Trotzdem ist diese Methode für die Praxis sehr interessant. Die Unsicherheit der Laufzeit wird mit der Aussicht auf eine "schnelle" optimale Lösung oft in Kauf genommen. Experimentelle Untersuchungen haben gezeigt, dass die Dauer der Suche oft von den Zahlenwerten der Konstanten $b$ (die Kapazität des Knapsack) und der Grösse der Objekte abhängt.

Für das in dieser Diplomarbeit zu lösende Problem sind diesbezüglich die Voraussetzungen sehr günstig. Die Grösse der Speicherobjekte und der Konstanten $b$ ist immer eine gerade Zahl was der schnellen Lösung sehr zuträglich ist.

### 5.2.3 Polynomielle Approximationsalgorithmen

Polynomielle Aproximationsalgorithmen gehören zu einer Klasse von Methoden, die duch eine Erweiterung der Eingabe mit einer Konstanten $\varepsilon$ in polynomieller Zeit eine $\varepsilon$-optimale Lösung des Problems $P$ liefern. Die Variable $\varepsilon$ ist eine Art Güteparameter für die Lösung des Problems, denn sie definiert den Zeitpunkt bei der Berechnung, in dem die $\varepsilon$-optimale Lösung sich um den Faktor $\varepsilon^{-1}$ an die optimale Lösung angenähert hat.

Die Laufzeit solcher Algorithmen wird hauptsächlich von dem Parameter $\varepsilon$ bestimmt und lautet $O(n^k)$ für $k = \varepsilon^{-1}$. Wie zu Eingang des Kapitels bereits erwähnt lautet die Definition für einen effizienten Algorithmus, polynomielle Laufzeit. Setzt man diesen Maßstab an eine $\varepsilon = \frac{1}{2}$ optimale Lösung an, so muss laut Definition die Bewertung effizient lauten. Formal betrachtet heisst es aber: Für eine Lösung die sich um den Faktor $\frac{1}{2}$ unterscheidet, nimmt man eine Laufzeit von $O(n^3)$ in Kauf. Da liefert jede Heuristik in Zeit $O(n)$ eine bessere Lösung. Steigert man hingegen die Güte der Lösung, so explodiert durch den Parameter $k$ die Laufzeit. Für eine bis auf $\varepsilon = \frac{1}{100}$ genaue Lösung ergibt das eine Laufzeit von $= O(n^{100})$.

Wie man sieht, ist diese Methode nicht ganz so gut wie ihr Name das hätte vermuten lassen.

## 5.3 Lösung durch Branch and Bound

Die Entscheidung, das Knapsack-Problem durch die Branch and Bound Methode zu lösen, hatte mehrere Gründe. Im Laufe der Ausführungen, die die Lösung beschreiben, werden alle vorgestellt.

Die Ausgangssituation bei der Implementierung dieser Methode ist die, dass die im Kapitel 4 beschriebene Analyse das Datenmaterial bereitstellt. Es sind eine Menge $M$ von Speicherobjekten mit zwei Parametern:

- Grösse der Speicherobjekte, ein Wert $a = size(n) \in \mathbb{N}$
- Potenzial der Energieeinsparung, $c = val(n) \in \mathbb{N}$

wobei $s$ize(n) und $v$al(n) zwei Funktionen sind, die für ein gegebenes Objekt $n \in M$ die Grösse und das Energie-Einsparpotenzial von $n$ liefern. Zur Erinnerung sei noch mal erwähnt, dass die Objektgrösse sich für

- globale Variablen
  - skalare, durch die Grösse des Basistyps
  - Arrays $A[l]$, als $\sum_{i=1}^{n}$ über den Basistypen
- Grundblöcke, $\sum_{i=1}^{k}$ über alle $k$ Statements, und
- Funktionen, als $\sum_{i=1}^{j}$ über alle $j$ Grundblöcke

definiert. Auf die Berechnung der Werte für das Energie-Einsparpotenzial, das komplexer ist als die Berechnung der Grösse, verweise ich auf Kapitel 4.2 Programmanalyse.

Ein weiterer Parameter, der als Eingabe für das Knapsack-Problem nötig ist, ist eine Konstante $b \in \mathbb{N}$. Diese Zahl $b$ definiert die Grösse des Knapsack und ist davon abhängig, wie gross der für die Speicherobjekte zur Verfügung stehende Scratch-Pad ist. Für experimentelle Untersuchungen verschiedener Speichergrössen kann dieser Parameter natürlich variabel verändert werden. Mit diesen Ausgangswerten, die die vollständige Eingabemenge für das Knapsack-Problem darstellen, kann die Lösung angegangen werden.

### 5.3.1 Suche der Lösungsmenge

Die Effizienz der Lösung durch die Branch und Bound-Methode ist entscheidend davon abhängig wie effektiv der nicht zulässige Lösungsraum des gesamten Lösungsraums von $2^n$ Vektoren vom zulässigen abgetrennt werden kann. Die Suche der Lösung findet in einem binären Baum statt, in dem jeder innere Knoten auf der Ebene $e$ ein Teilproblem mit $e$ gesetzten und $n-e$ freien, nicht entschiedenen Variablen darstellt.

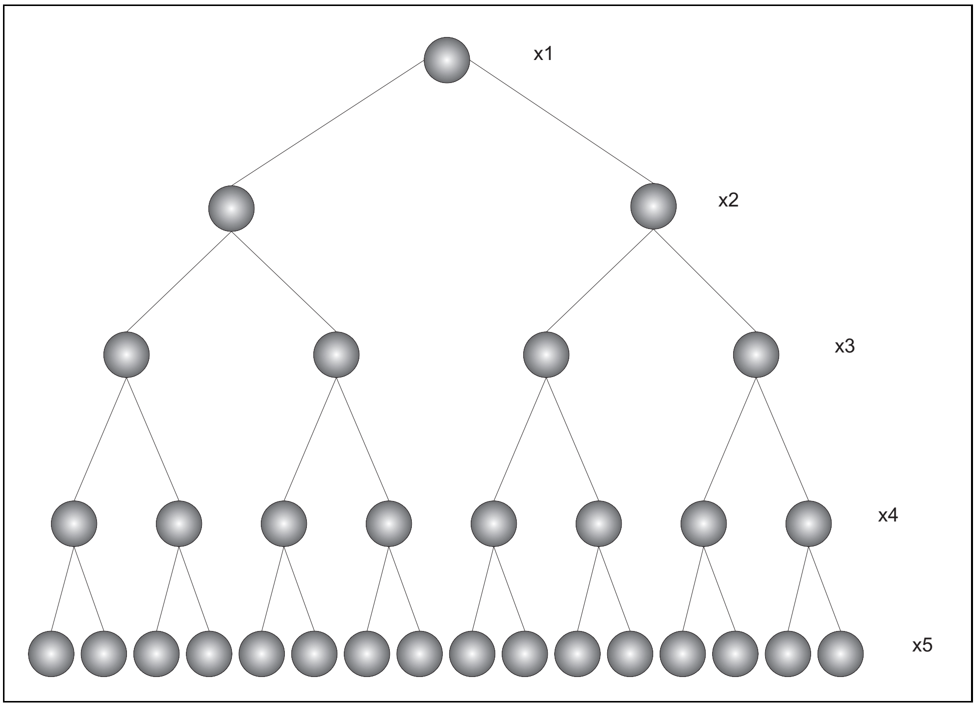



Abbildung 5.1: Beispiel eines vollständigen Lösungsraums für $n = 5$

Der Startknoten stellt das gesamte Problem dar, an welchem der Lösungsraum $2^n$ mögliche Lösungen hat. Da auf jeder Ebene des binären Baums die Entscheidung über eine Variable gefällt wird und wir $n$ Variablen entscheiden müssen, hat man spätestens nach $n$ Ebenen die Lösung des gesamten Problems. Dies stellt den Worst Case dar und soll durch das Abschneiden grosser Lösungsräume nicht eintreten.

Warum hofft man, grosse Bereiche des Lösungsbaums abschneiden und somit nicht berechnen zu müssen. Die Antwort darauf gibt das folgende Beispiel:

Die mathematische Definition des Beispiels erfolgt durch eine Zielfunktion der Form:

$$W = 11x_1 + 14x_2 + 16x_3 + 8x_4 + 7x_5 \tag{5.1}$$

und eine Restriktionsungleichung, die die Grösse des Knapsack berücksichtigt.

$$17x_1 + 16x_2 + 21x_3 + 8x_4 + 12x_5 \leq 36 \tag{5.2}$$

Dadurch, dass die Objekte $a_1, ..., a_5$ nicht teilbar sind, und entweder ganz oder gar nicht in den Knapsack hineingenommen werden, kommen noch fünf weitere "Ganzzahligkeits-Restriktionen" für $x_1, ..., x_n$ hinzu, dessen Definition wie folgt aussieht:

$$x_i = \begin{cases} 1 & : \text{ Mitnahme des Gegenstandes } i \\ 0 & : \text{ Nichtmitnahme des Gegenstandes } i \end{cases} \tag{5.3}$$

mit $i \in (1, ..., 5)$

Die Variablen $x_i$ sind also in unserem Fall binäre Entscheidungsvariablen. Das hier vorliegende Problem $P_1$ hat fünf Unbekannte. Ein naheliegender Versuch, diese Gleichung zu lösen, wäre, eine der Variablen auf einen Wert (das eine Mal auf 1, das andere auf 0) zu fixieren. Diese Maßnahme reduziert die Anzahl der Unbekannten von 5 auf 4 und vereinfacht das Problem $P_1$. Sie schafft aber gleichzeitig, dass wir zwei Unterprobleme $P_2$ und $P_3$ bekommen, die man auf genau die gleiche Weise behandeln kann, bis alle Variablen fixiert sind. Irgendeine Gleichung hat dann den maximalen Wert und genügt der Restriktionsgleichung. Sie ist die Lösung des Problems.

Die Verdopplung der Probleme bei der Entscheidung einer Variablen ergibt die uns bekannte Anzahl von $2^n$ Problemen und ist der Branch-Schritt der Branch and Bound-Methode.

Es ist die erschöpfende Suche über den gesamten Lösungsraum und bringt keine Vorteile. Der Vorteil ergibt sich durch das weiter oben erwähnte Abschneiden von Lösungsräumen des Baums und wird anhand des Beispiels erklärt werden.

### Effizienz von Objekten

Der erste Schritt auf diesem Wege ist die Errechnung der Effizienz der zur Verfügung stehenden Objekte. Tatsache ist, dass manche Objekte effizienter sind als andere. Ihre Effizienz $e_i$ ergibt sich dabei aus dem Quotienten von Wert $c_i$ und Grösse $a_i$.

$$e_i = \frac{c_i}{a_i} \tag{5.4}$$

Für unser Beispiel bedeutet das:

| | Wert | Grösse | e |
|---|---|---|---|
| x1 | 11 | 17 | 0.65 |
| x2 | 14 | 16 | 0.875 |
| x3 | 16 | 21 | 0.76 |
| x4 | 8 | 8 | 1.0 |
| x5 | 7 | 12 | 0.58 |

Tabelle 5.1: Datenbasis für das Beispielproblem

Nach der Berechnung von $e_i$ werden diese anschliessend sortiert. D.h., dass unter Umständen die Objekte umnumeriert werden müssen. Die Allgemeinheit der Lösung wird durch diesen Schritt nicht beeinträchtigt.

Die Berechnung der Effizienzen $e_i$ geschieht als Vorarbeit für drei Funktionen, die entscheidend für die Verkleinerung des Lösungsraums verantwortlich sind. Im einzelnen sind es:

1. Upper Bound
   Diese Funktion berechnet für jeden Knoten $k$ die Obergrenze eines jeden Lösungswertes, der ab diesem Knoten zu erreichen ist. Diese Obergrenze besagt, dass im gesamten Lösungsbaum, dessen Rootknoten $k$ ist, es keine Lösung gibt, die diese Obergrenze überschreitet. Eine Aussage dieser Art hat, wie später ausgeführt wird, grossen Einfluss auf die Entscheidung, an welchem Knoten des Baumes weiter gerechnet wird und trägt somit zur Verkürzung der Laufzeit des Gesamtprogramms bei.

2. Lower Bound
   Sie berechnet für jeden Knoten $k$ einen garantierten Mindestwert einer Lösung, der sich innerhalb des Lösungsraums befindet, dessen Rootknoten der Knoten $k$ ist.

3. Branch
   Die Funktion entscheidet über die Variable $x_i$, die als nächstes fixiert wird und dadurch das Problem in zwei disjunkte Teilprobleme zerlegt. Die Güte der gesamten Methode hängt davon ab, den *zulässigen* Lösungsraum in möglichst disjunkte Teilräume aufzuteilen.

**Upper Bound**

Die Berechnung der oberen Schranke $U$ für einen gegebenen Knoten $k$ geschieht nach folgender Vorschrift:

1. Bestimme das grösste $i$, so dass $a_1 + ... + a_i \leq b$ ist und berechne $b^* := b - (a_1 + ... + a_i)$.
   Bei der Bestimmung von $i$ können zwei Fälle auftreten:

Fall 1 Falls $i = n$, dann ist $U = c_1 + ... + c_n$.

Fall 2 Falls $i < n$, berechne $\delta = b^*/a_{i+1}$ und setze
$U = c_1 + ... + c_i + \delta c_{i+1}$.

Die Zeitkomplexität dieser Funktion beträgt $O(n)$. Dass dies tatsächlich die obere Schranke $U$ für alle möglichen Nutzwerte aller zur Verfügung stehenden Objekte ist, folgt aus der Sortierung der Objekte nach ihrer Effizienz. Da die Effizienz aller Objekte mit fortschreitendem $i$ abnimmt, wurde der Platz $b - b^* = \sum_{k=1}^{i} a_i$ optimal genutzt. Darüberhinaus lässt sich sagen, dass der verbliebene Platz $b^*$ durch den $\delta$-Anteil von $a_{i+1}$ mit Nutzen $\delta c_{i+1}$ ebenfalls optimal genutzt wird.

**Beweis**: Jedes der Objekte, die $i + 1$ nachfolgen, haben eine kleinere Effizienz pro Platzeinheit und somit einen kleineren $\delta$-Wert für den Nutzen. Das Auffüllen des $b^*$-Platzes mit dem Teilstück des Objektes $i_{i+1}$ ist für die Lösung wegen der Ganzzahligkeitsresriktion nicht erlaubt, liefert aber die obere Schranke $U$.

### Lower Bound

Die Berechnung von $L$, der unteren Schranke für den Nutzen, hat ebenfalls die Zeitkomplexität $O(n)$. Dabei geht man wie folgt vor:

1. Setze $b^+ = 0$ und $L = 0$

2. Für $i = 1, ..., n$
   Falls $b^+ + a_i \leq b$ dann $b^+ := b^+ + a_i;\ L := L + c_i$

Am Ende des einmaligen Durchlaufs durch das Array hat $L$ den Wert der unteren Schranke. Der Wert von $L$ für das Beispiel beträgt $L = 29$ und kommt durch die Hineinnahme von $i = \{1, 2, 5\}$ zustande.

### Branch

Der entscheidende Schritt im gesamten Lösungsansatz ist das Fixieren einer Entscheidungsvariable $x_i$ auf einen festen Wert und die Bildung von zwei Subproblemen. Dafür ist es möglich, jede verbliebene Entscheidungsvariable zu wählen. Als günstig hat sich allerdings die Aufspaltung des Problems am "kritischen" Objekt vorzunehmen, erwiesen. Das kritische Objekt ist das Objekt $i + 1$, das bei der Berechnung von Upper Bound nicht mehr vollständig in den Knapsack passte und mit seinem $\delta$-Anteil zur $U$ beitrug.

Es entstehen nun zwei Probleme:

Problem 1 Durch Hineinnahme von Objekt $i+1$ (Setzen der Entscheidungsvariable $x_{i+1}$ auf 1) in den Knapsack wird unter allen noch verbliebenen zulässigen Lösungen die optimale Lösung gesucht. Hierbei muss man, weil sich nun erzwungenermassen ein Objekt im Knapsack befindet, den verbliebenen freien Platz $b$ um $a_{i+1}$ reduzieren und den Wert $W_k$ der bisherigen Lösung im Knoten $k$, auf $W_{k+1} + c_{i+1}$ erhöhen. Wird $b$ durch diesen Schritt negativ, so können wir an diesem Teilproblem die Berechnung einstellen, da er keine zulässigen Lösungen mehr generieren kann. Das Objekt $i + 1$ wird in die Lösungsmenge aufgenommen. Implementiert wurde das durch ein Array `bool localopt[n]` in dem zu Anfang alle Werte auf 0 initialisiert wurden. Die Hineinnahme bedeutet nun, dass der Wert von `localopt[n + 1]` auf 1 gesetzt wird.

Problem 2 Die Entscheidungsvariable $x_{i+1}$ für Objekt $i+1$ wird auf 0 gesetzt, d.h. es wird unter allen zulässigen Lösungen mit $x_{i+1} = 0$ eine optimale Lösung gesucht. Die Gewichtsschranke $b$ bleibt unverändert und der Wert der bisherigen Lösung $W_k$ wird in den Knoten $k+2$ übernommen.

In beiden Fällen wird das Objekt $i + 1$ aus der Menge der zur Verfügung stehenden Objekte gestrichen. Bei der Implementierung wurde das durch ein Array `bool current[n]` realisiert, in dem zu Beginn alle Werte auf 1 gesetzt wurden. Mit jedem Entscheidungsschritt ist das "kritische" Objekt auf 0 zu setzen.

### Branch and Bound-Algorithmus

Der Algorithmus operiert auf einem binären Baum, dessen erster Knoten das Gesamtproblem darstellt. Er arbeitet rekursiv und steuert den Rekursionsschritt (an welchem Problem $P_{left}$ oder $P_{right}$ des aktuellen Knoten $k_{akt}$ weitergearbeitet wird) in Abhängigkeit von den $U$- und $L$-Werten der Probleme $P_{left}$ und $P_{right}$. Jeder Knoten beinhaltet eine Datenstruktur, die folgende Parameter festhält:

- Menge zur Verfügung stehender Objekte (`bool current[n]`)
- Menge bereits fixierter Variablen $x_i$ $i \in \{1, ..., n\}$ (`bool localopt[n]`)
- untere Schranke $L$
- obere Schranke $U$
- Wert der bisherigen Lösung

Die einzelnen Schritte des Algorithmus sehen wie folgt aus:

1. Sortiere alle Objekte nach Effizienz $e_i$ und initialisiere beide Arrays `bool localopt[n]` und `bool current[n]`
2. berechne $U$ mit Upper Bound und $L$ mit Lower Bound im aktuellen Knoten $k_{akt}$.

3. $U = L$, so wird die aktuell beste Lösung $S_{best}$ mit $L$ verglichen und ggf. aktualisiert. Die Belegung der Variablen, die zu $L$ führt, ist Lösung des Problems. STOP

4. $U \neq L$: Vergleiche $U$ des $k_{akt}$ mit $S_{opt}$. Ist $S_{opt} > U$ STOP.
Ansonsten zerlege Problem im Knoten $k_{akt}$ in Probleme $P_{left}$ und $P_{right}$

5. Wähle $P_u \quad u \in \{left, right\}$ mit grösserem $U$-Wert zum $k_{akt}$.
Aktualisiere `bool localopt[n]` und `bool current[n]`, siehe Branch-Schritt und fahre fort mit Schritt 2.

Die Lösung wird nun exemplarisch an einem Beispiel verdeutlicht.
In dem Beispiel ergibt das nach erfolgter Sortierung von $e_i$ und Umnummerierung der Objekte folgende Werte für $i$ , $U$, $b^*$ und $\delta$:
$i = 3, \quad U = 31,14 \qquad b^* = 12 \qquad \delta_3 = 12/21$

Im ersten Schritt, wird also über die Variable $x3$ entschieden, da Element drei nach vorgestellten Algorithmus als erstes Element nicht mehr in den Knapsack passt. Es entstehen nun zwei Probleme $P_1$ und $P_2$ durch inklusion bzw. exklusion des dritten Elements. Im ersten Fall senkt man die Kapazität des Knapsack auf Knapsacksize (KS)=15 und erhöht den Wert $W$ der Lösung um 16. In anderen Fall bleiben beide Parameter unverändert. Durch die folgende Berechnung der Upper- und Lower-Werte in den neu entstandenen Problemen stellt man fest, dass das durch Inklusion von $x3$ erzeutes Problem den höheren Upper-Bound besitzt und folglich an diesem weiter zu verfahren ist. An diesem Knoten wird das Problem weiter aufgespalten indem die nächst Variable ($x2$) gesetzt wird. Bereits an diesem Punkt stellt man fest, dass die Inklusion von $x2$ nicht mehr möglich ist, d.h. der Entscheidungsbaum wird an der Stelle bereits beschnitten. Die weitere Berechnung wird so weiter fortgeführt, bis alle Variablen entschieden sind und Upper- und Lower-Wert eines Knotens gleich sind.

| | Wert | Grösse | e |
|---|---|---|---|
| x1 | 8 | 8 | 1.0 |
| x2 | 14 | 16 | 0.875 |
| x3 | 16 | 21 | 0.76 |
| x4 | 11 | 17 | 0.65 |
| x5 | 7 | 12 | 0.58 |

Tabelle 5.1: Datenbasis nach Umstellung der Elemente

### 5.3.2 Erweiterung des Konzepts

Die Erweiterung des Konzeptes beruht auf der Berücksichtigung der Abhängigkeiten zwischen Funktionen und Basic-Blöcken. Im ersten Lösungsansatz war die Eingabemenge auf:

1. skalare Variablen und Basic-Blöcke, oder
2. skalare Variablen und Funktionen

eingeschränkt.

Der Grund für diese Einschränkung war, dass keine Unterschiede zwischen Funktionen und Basic-Blöcken gemacht wurden [ST01a]. Die Auswahl der Objekte durch den Algorithmus, die in den Scratch-Pad verschoben werden sollten, konnte ergeben, dass eine Funktion $F_m$ gleichzeitigt mit einem Basic-Block $BB_m$, der Teil von $F_m$ ist, erfolgte. Da aber bei der Verschiebung der Funktion $F_m$ ebenfalls alle in ihm enthaltenen $BB_{m_{1,\ldots,n}}$ mit verschoben werden, führt dies zu Inkonsistenzen innerhalb der beiden Dateien, die der Linker auf die beiden Speicher Scratch-Pad und Off-Chip verteilt. Die Inkonsistenz beruht auf dem doppelten Vorkommen von Labeln innerhalb des Assembler-Codes.

Mit der Hinzunahme von weiteren Restriktion beim Entscheidungsprozess, ob ein Objekt in die Lösungsmenge aufgenommen werden kann, konnte diese Einschränkung behoben werden. Die einzige Einschränkung bisher bei der Hineinnahme eines Objekts $i$ in den Knapsack (Fixieren der Entscheidungsvariable in $\texttt{localopt}[\texttt{i}]$ auf 1) war die Grössenrestriktion:

$$\sum_{k=1}^{n} a_k + a_i \leq b, \quad \text{mit} \quad \texttt{current}[\texttt{k},\texttt{i}] = \texttt{1} \tag{5.5}$$

Die zweite Restriktion beruht auf dem Prinzip des gegenseitigen Ausschlusses.
Hat die Upper Bound-Berechnung für einen Knoten $k_{akt}$ das Objekt $i$ als kritisches Objekt ergeben, so gibt es zwei Fälle die per Exceptionhandling behandelt werden müssen:

1. Objekt $i$ ist ein Basic-Block, dann:
   Untersuche alle Objekte, für die gilt: `localopt[]=1` auf die Eigenschaft IsFunction. Ist ein solches Objekt vorhanden, dann prüfe ob BB $\in$ Func. Wenn
   - Ja, dann setze `current[i]=0` und führe Lower- und Upper Bound Berechnungen für aktuellen Knoten $k_{akt}$ noch mal durch.
   - Nein, dann nehme Objekt $i$ in die Menge der Lösungsobjekte, d.h. `localopt[i]=1`

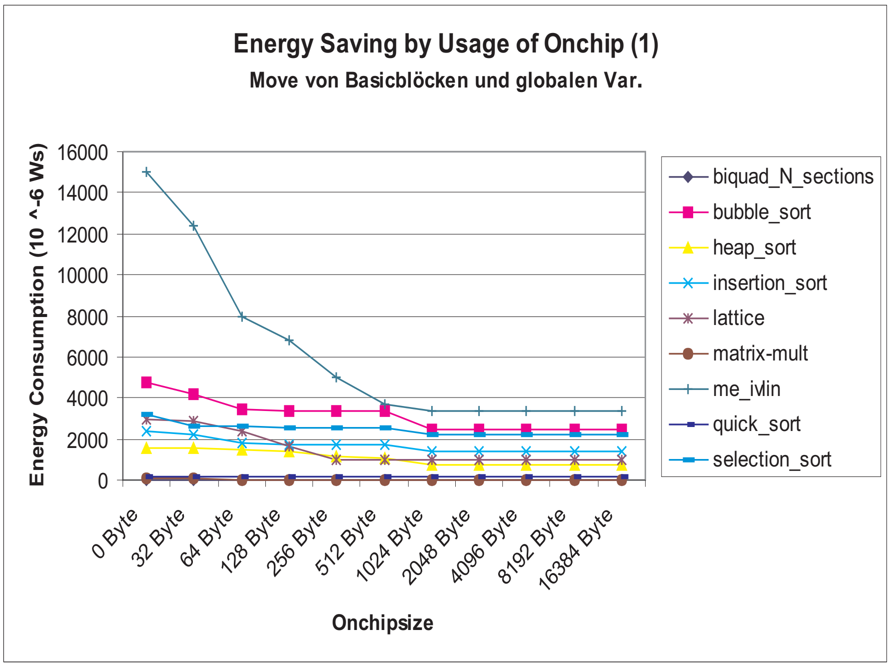
2. Objekt $i$ ist eine Funktion, dann:
   Untersuche alle Objekte, für die gilt: `localopt[]=1` auf die Eigenschaft IsBasicBlock und BB $\in$ Funktion. Eigenschaften treffen zu

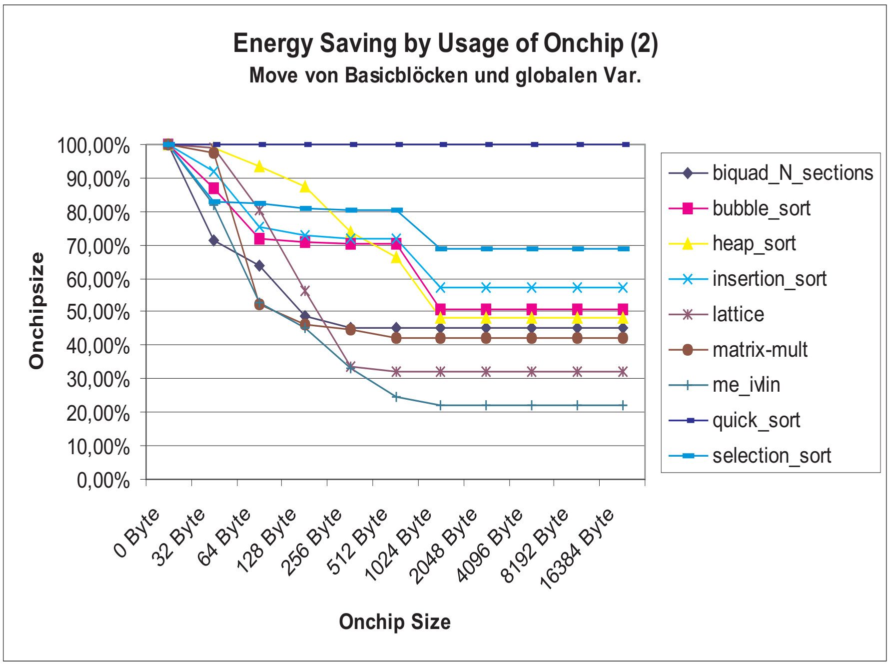
   - Ja, dann setze `current[i]=0` und führe Lower- und Upper Bound Berechnungen für aktuellen Knoten $k_{akt}$ noch mal durch.
   - Nein, dann nehme Objekt $i$ in die Menge der Lösungsobjekte, d.h. `localopt[i]=1`

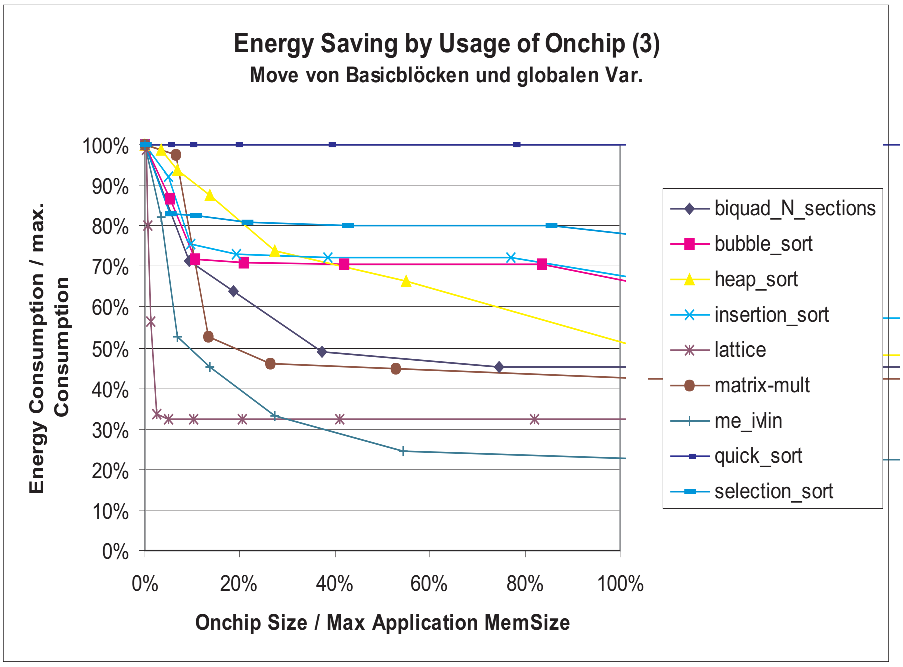
Diese zusätzliche Restriktion, ließ die Erweiterung der Objektmenge auf Funktionen und Basic-Blöcke, als mögliche Kandidaten für die Auswahl in den Scratch-Pad zu.

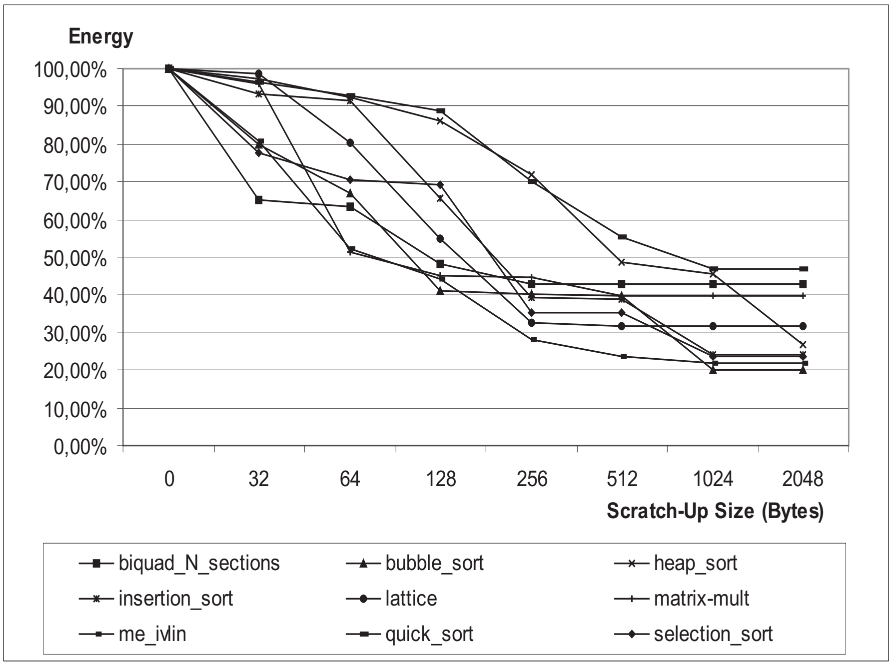
## 5.4 Ergebnisse

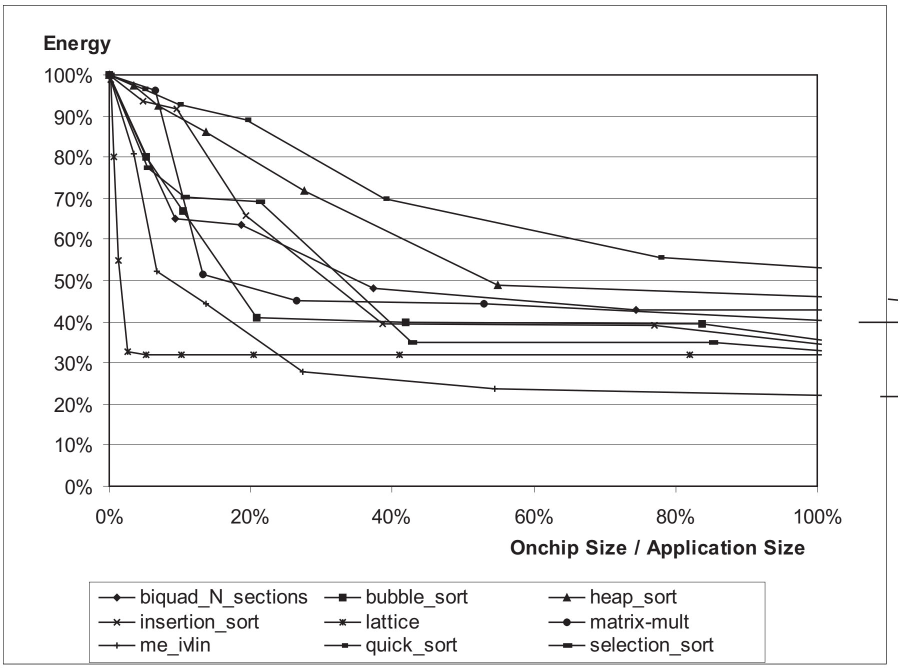
Für die Generierung der Ergebnisse wurde der im Kapitel 3.4 beschriebene Trace-Analyzer eingesetzt.




Energy Saving by Usage of Onchip (2)
Move von Basicblöcken und globalen Var.
Onchipsize
100,00%
90,00%
80,00%
70,00%
60,00%
50,00%
40,00%
30,00%
20,00%
10,00%
0,00%
0 Byte
32 Byte
64 Byte
128 Byte
256 Byte
512 Byte
1024 Byte
2048 Byte
4096 Byte
8192 Byte
16384 Byte
Onchip Size
biquad_N_sections
bubble_sort
heap_sort
insertion_sort
lattice
matrix-mult
me_ivlin
quick_sort
selection_sort


Energy Saving by Usage of Onchip (3)
Move von Basicblöcken und globalen Var.
Energy Consumption / max. Consumption
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0%
20%
40%
60%
80%
100%
Onchip Size / Max Application MemSize
biquad_N_sections
bubble_sort
heap_sort
insertion_sort
lattice
matrix-mult
me_ivlin
quick_sort
selection_sort


Energy
100,00%
90,00%
80,00%
70,00%
60,00%
50,00%
40,00%
30,00%
20,00%
10,00%
0,00%
0
32
64
128
256
512
1024
2048
Scratch-Up Size (Bytes)
biquad_N_sections
bubble_sort
heap_sort
insertion_sort
lattice
matrix-mult
me_ivlin
quick_sort
selection_sort


Energy
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0%
20%
40%
60%
80%
100%
Onchip Size / Application Size
biquad_N_sections
bubble_sort
heap_sort
insertion_sort
lattice
matrix-mult
me_ivlin
quick_sort
selection_sort

# Kapitel 6

# Mehrdimensionales Knapsack-Problem

Die Verschiebung von Basic-Blöcken in den Scratch-Pad hat natürlich Auswirkungen auf die Grösse des Basic-Blocks und den Energieverbrauch des Gesamtprogramms.

Pro verschobenem Basic-Block erfordert das im Worst Case:

- 1 OFF-Chip BL, der mit 2 Thumb Befehlen kodiert wird (4 BYTE)
- 2 On-Chip BL (8 Byte)

insgesamt 12 Byte zusätzlichen Code, davon 8 in Scratch-Pad und 4 Byte im Off-Chip.

Während der Ansprung einer Funktion oder einer Variable keinen zusätzlichen Platz bedarf im Scratch-Pad verursachen, benötigt man für BB die beiden BL-Rücksprünge. Einer der beiden BL ist nicht immer nötig. Er kann dann weggelassen werden, wenn der nachfolgende Basic-Block ebenfalls in den Scratch-Pad verschoben wurde. Der Platz, den man dann für beide Basic Blöcke benötigt, wäre um den einen BL-Befehl geringer und stünde anderen Memory-Objekten zur Verfügung.

Die konsequente Fortführung dieser Überlegung führt zur Einführung von *Multibasicblock Objekten*, die als Konkatenation von hintereinander liegenden Basic Blöcken zu verstehen sind.

In einigen Fällen ist das konkatinieren von Basic Blöcken nicht sinnvoll oder gar unmöglich und zwar, wenn der letzte Befehl des Basic Blocks ein:

- MOV PC, LR
  Dieser Befehl kopiert das link register in den programm counter und ist quasi ein Sprungbefehl.
- POP <reglist, PC>
  Ebenfalls ein Rück*sprung*befehl, der am Ende von Unterfunktionen aufgerufen wird

- B label, (unconditional branch)
  Ein unbedingter Sprung verzweigt irgendwo in den Programmcode, führt also nicht den im Speicher ihm nachfolgenden nächten Befehl aus

Die Bildung von *Multi-Memory Basic-Blöcken* führt nicht nur wie beschrieben zu Reduktion des benötigten Platzbedarfs sondern auch zur Verringerung des Energieverbrauchs durch das Auslassen des Sprungbefehls zwischen den $BB_{mm}$ im Scratch-Pad (BL). Dieser kann sich für mehrere konkatinierte Basic Blöcke bis auf den in der Formel angegebenen Wert steigern.

$$E_{save_{BB_{mm}}} = \sum_{k=1}^{n} E_{save.BB_k} + (n-1)E_{on}(BL) \tag{6.1}$$

$E_{save_{BB_{mm}}}$ = Energieeinsparpotenzial für einen Multi-Memory Basic Block
$E_{saveBB_k}$ = Energieeinsparpotential für ein Basic-Block $k$
$E_{on}(BL)$ =Energieverbrauch für ein long branch im On-Chip

Das Verschieben einzelner hintereinander liegender $BB$ in den Scratch-Pad benötigt wie im Kapitel 4.2.2 dargestellt bei konditionalen Sprüngen 2 BL für den Rücksprung. In den beiden Abbildungen 6.1 als mögliche Folge von $BB$ und Abb. 6.2, die die Multi Memory Variante darstellt, wird das Weglassen der BL zwischen $BB_{mm}$ vorgestellt.

Die Einsparung des einen Sprung-Befehls erfordert einige Veränderungen im Code, wie man beim Vergleich der beiden Abbildungen 6.1 und 6.2 am $bb2$ sehen kann. Der Kontrollfluss in diesem Beispiel geht bei erfüllter Bedingung in $bb2$ nach $bb4$, ansonsten nach $bb3$ dem nachfolgendem BB. Werden nun beide BB in den Scratch-Pad verschoben, müsste der Sprung nach $bb4$ unter Beibehaltung der Sprungbedingung (BGT bb 4) mit einem zusätzlichen BB im Scratch-Pad (s. Kapitel 4.2.2) realisiert werden, wegen der Beschränkung der Sprungweite des Befehls BGT (Branch Greater Then). Da man aber bei $BB_{mm}$ weiß, dass der nachfolgende BB, der bei nicht erfüllter Bedingung ausgeführt wird, sich ebenfalls in Scratch-Pad befindet, kann man den in der Sprungweite benachteiligten BGT zum Ansprung des $bb3$ nutzen. Dazu muss lediglich die Bedingung des Sprungs negiert werden. Trifft nun die Bedingung nicht zu, so wird mit dem nächsten Befehl (BL bb 4) fortgefahren, der den Sprung nach $bb4$ realisiert. Die Neuberechnung der Grösse von $BB_{mm}$ erfordert die Neuberechnung der$E_{save_{BB_{mm}}}$ und ergibt zwangsläufig einen neuen $e_i$.

$$S_{BB_{mm}} = \sum_{k=1}^{n} a_{BB_k} - (n-1) \cdot S(BL) \tag{6.2}$$

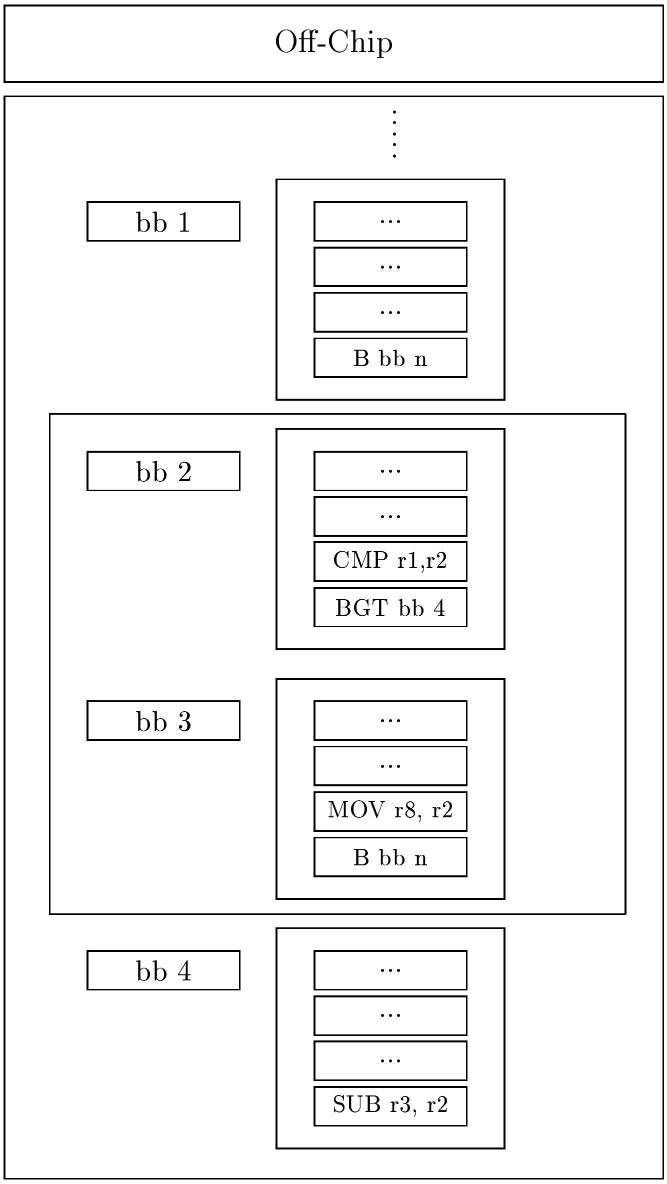



Abbildung 6.1: Beispiel möglicher $BB_{mm}$ - Speicherlayout: Singel Memory

Die Effizienz eines $BB_{mm}$ ist dann:

$$e_{BB_{mm}} = \frac{\sum_{k=1}^{n} E_{save.BB_k} + (n-1)E_{on}(BL)}{\sum_{k=1}^{n} a_{BB_k} - (n-1)S(BL)} \tag{6.3}$$

Bei der Erweiterung der Eingabemenge durch die Multi-Memory Basic-Blöcke, im weiterem $BB_{mm}$ genannt, kommt es zu einem bekannten Problem, das im Kapitel 5.3.2 Erweiterung des Konzepts, angesprochen wurde.

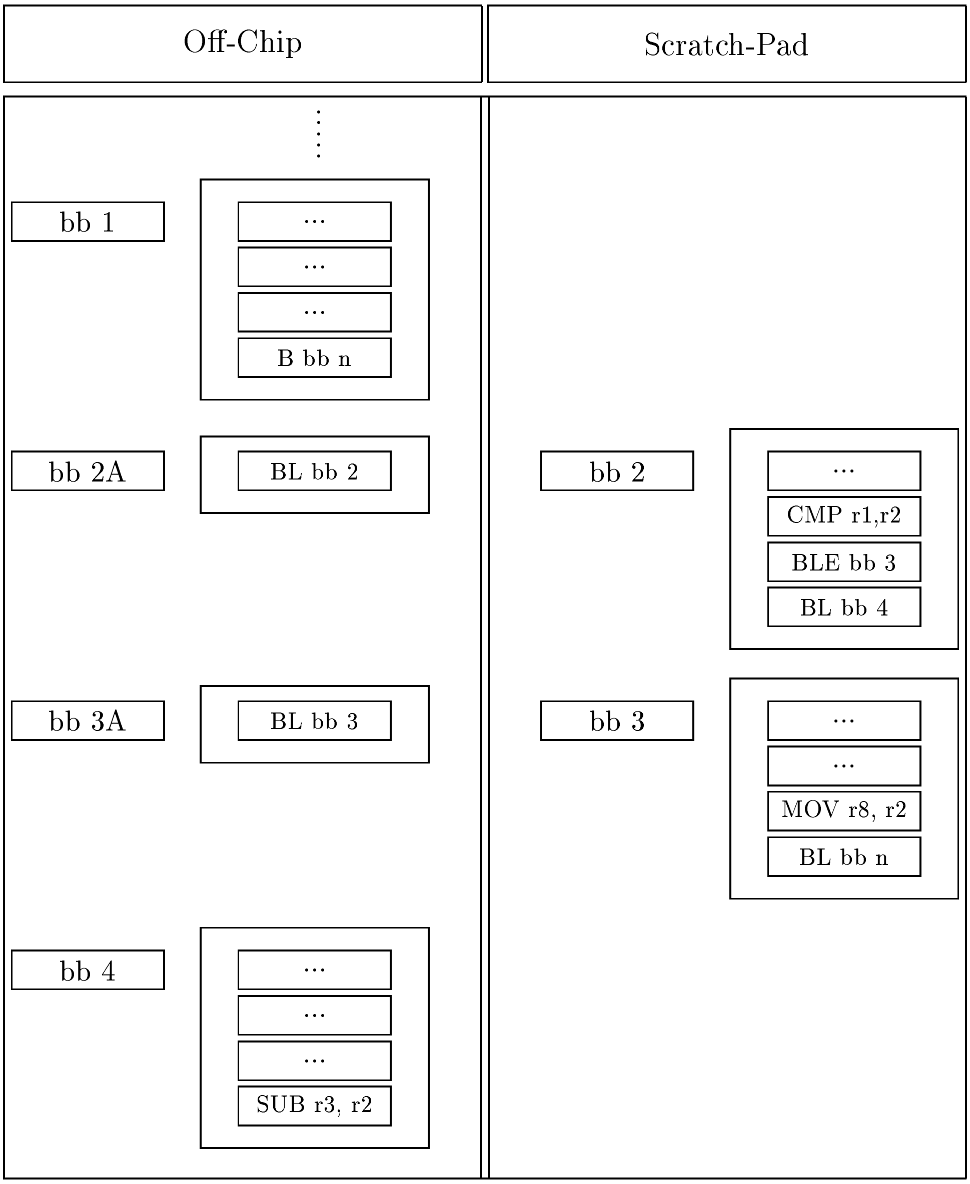



Abbildung 6.2: Einsparung von BL - Speicherlayout: Multiple Memory

Waren da nur Abhängigkeiten zwischen Funktionen und den in ihr enthaltenen Basic-Blöcken zu berücksichtigen, kommen hier die der $BB_{mm}$ innerhalb einer Funktion dazu.

Wegen der neu hinzugekommenen Abhängigkeiten, die sich mit ihrer Anzahl nicht mehr vernünftig von Hand haben lösen lassen, wurde zur ihrer Lösung das Konzept des Integer Linear Programming bemüht. Dafür musste das Problem in ein entsprechendes Format umgeschrieben werden. Anschließend wurde das so formulierte Problem mit einem ILP-Solver gelöst.

Die Vorgehensweise wird in den nächsten beiden Kapiteln beschieben. Das nächste Kapitel beschreibt dabei die Methode "Integer Linear Programming", das übernächste den Solver.

Hierbei wurde der ILP-Solver CPLEX ver. 6.0 der Firma ILOG verwendet.

## 6.1 Integer Linear Programming

Lineare Programmierung ist eine der wichtigsten und am meisten benutzten Optimierungs Techniken im Bereich von Operation Research. Sie umfasst zwei Klassen von mathematischen Problemen, die sich durch lineare Gleichungen formulieren lassen. Sie charakterisieren sich durch drei Merkmale:

- eine Zielfunktion
- eine Menge von expliziten Constraints
- eine Menge von impliziten Constraints

Eine allgemeine Darstellung solcher Probleme sieht wie folgt aus:

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n = k$$

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n \leq b_n \end{array}$$

$$x_1, x_2, \cdots, x_n \geq 0$$

Dass das Problem der Verteilung von Memory-Objekten in den Scratch-Pad, sich in diese Darstellung überführen lässt ist nicht schwer einzusehen.

Die erste Gleichung, sogenannte Zielfunktion, ist die aus Kapitel 5 als Optimierungsgleichung für das Knapsack-Problem bekannt. Sie soll maximiert werden. Die folgenden Gleichungen in der allgemeinen Darstellung des linearen Problems sind Restriktionsgleichungen. Für das eindimensionale Knapsack-Problem belief sich ihre Zahl auf "eins" und brachte die Größeneinschränkung zum Ausdruck. Wie wir vorhin festgestellt haben, hat sich die Zahl der Restriktionen, im folgenden Constraints genannt, auf einen Wert $\gg 1$ erhöht. Die Constraints werden dazu benutzt werden, Abhängigkeiten zu definieren und gegenseitigen Ausschluß zwischen nicht erlaubten Kombinationen von Funktionen, BB und den neu hinzugekommenen $BB_{mm}$ zu garantieren.

Schließlich dient die letzte Zeile um die Wertebereiche der Variablen zu Definition, die je nach Wertebereich unterschiedliche Klassen von Problemen bedingen.

Es gibt nun zwei Möglichkeiten, welche Werte die Variablen $x_1, ..., x_n$ annehmen können. In Abhängigkeit davon unterscheidet man Arten von linearen Problemen. Es sind:

1. integer linear programming Probleme
   Die Variablen können nur ganzzahlige, nichtnegative Werte annehmen. Diese Einschränkung bedeutet für viele Probleme den Ausschluß, um mit diese Methode gelöst zu werden. Im vorliegenden Fall ist die Klasse mächtig genug, da für die Variablen $x_i,\ x_i \in \{0, 1\}$ gilt. Sie kann zu lösung des Problems herangezogen werden.

2. linear programming Probleme
   Die Problemklasse ist eine echte Obermenge von ILP-Problems ($LP \subset ILP$) und erlaubt Variablenwerte $x_i$ für die gilt $x_i \in \mathbb{R}$

### 6.1.1 Definition von Constraints

Schon der einfache Fall, der im Kapitel 5.3.2 beschrieben ist, führte zu der Zunahme von Constraints $C$ von einem - Grössenkapazität des Knapsack, auf:

$$C_{all} = \sum_{P} BB \tag{6.4}$$

wobei $P$ das gesamte Programm und $BB$ die in ihm enthaltenen Basic Blöcke sind. Da jeder Basic Block einer Funktion angehört, muss über ein Constraint sichergestellt werden, dass keine Funktion $F_n$ gleichzeitig mit einem BB $BB_i$ mit $BB_i \in F_n$ in den Scratch-Pad verschoben wird. Diese Bedingung erfüllt die folgende Ungleichung:

$$F_n + \sum_{BB \in F_n} BB \leq 1 \tag{6.5}$$

Darüberhinaus treten bei Multi-Memory Basic-Blöcken Abhängigkeiten zwischen einzelnen $BB_{mm}$. Hat man eine Folge von mehreren hintereinander liegenden BB $bb_1, ..., bb_n$ die zu einem $BB_{mm}$ zusammengefasst werden können, so ergeben sich daraus $k\ \ BB_{mm}$, wobei:

$$k = \sum_{i=1}^{n-1} i \tag{6.6}$$

Die Zunahme der Speicherobjekte in Form von $BB_{mm}$ erfordert die Konstruktion von zusätzlichen Constraints, die den gegenseitigen Ausschluss sich überlappender $BB_{mm}$ gewährleisten. Zu diesem Zweck werden folgende Definitionen vorgenommen:

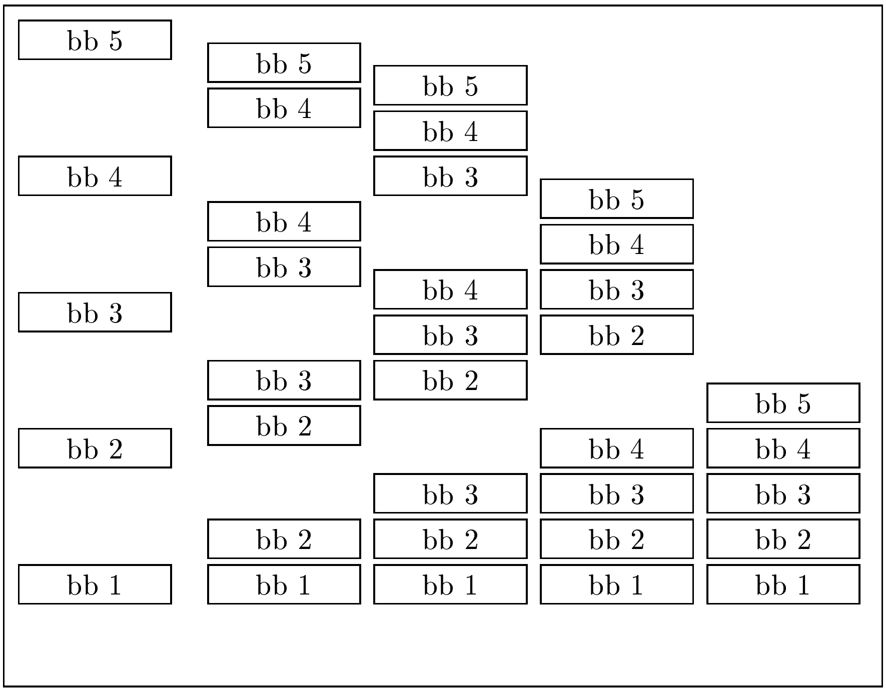



Abbildung 6.3: Alle möglichen $BB_{mm}$ eines BB-Blocks

| | |
|---|---|
| m($BB_i$) | = 1 wenn $BB_i$ in den Scratch-Pad verschoben wurde, sonst 0 |
| m($F_i$) | = 1 wenn $F_i$ in den Scratch-Pad verschoben wurde, sonst 0 |
| m($BB_{mm}$) | = 1 wenn $BB_{mm}$ in den Scratch-Pad verschoben wurde, sonst 0 |

Die formale Definition dieser Constraints lautet:

$$m(F_n) + \sum m(BB_{mm}) \leq 1 \qquad \text{mit} \qquad BB \in F_n \wedge BB \in BB_{mm} \tag{6.7}$$

für $BB_{mm}$ die mit Funktionen gemeinsame BB besitzen und:

$$m(BB_i) + \sum m(BB_{mm}) \leq 1 \qquad \text{mit} \qquad BB_i \in BB_{mm} \tag{6.8}$$

für $BB_i$ die in $BB_{mm}$ enthalten sind. Eine weitere Constraint-Gleichung für die oben erwähnten sich überlappenden $BB_{mm}$ lautet:

$$m(BB_{mm_j}) + m(BB_{mm_k}) \leq 1 \text{ mit } BB \in BB_{mm_j} \wedge BB \in BB_{mm_k} \tag{6.9}$$

Was im Kapitel 5.3.2 progammiertechnisch umgesetzt wurde, wird beim ILP-Ansatz über oben definierte Constraints erreicht.

## 6.2 CPLEX - ILP Solver

Das Programm CPLEX der Firma ILOG Inc. ist ein ILP- und LP Solver. Er wurde eingesetzt, um das erweiterte Knapsack-Problem zu lösen.

Die Eingabe der Daten an das Programm erforderte die Wahl eines geeigneten Eingabeformats, in die das Problem zuerst überführt werden musste. Die Wahl fiel auf das LP-Format und umfasst 14 Regeln. Die wichtigsten, die zur Erzeugung der Eingabedatei benutzt wurden, werden hier wiedergegeben:

- Die Datei muss mit einem der folgenden Schlüsselwörter beginnen:
  - MINIMIZE oder MINIMUM
  - MAXIMIZE oder MAXIMUM

  Sie definieren den Anfang der Zielfunktion.
- Variablen dürfen nicht mehr als 16 Buchstaben haben und mit einer Zahl beginnen. Längere Namen werden abgeschnitten.
- Der Constraint-Bereich muss mit einem "SUBJECT TO" beginnen und jeder Constraint muss in einer neuen Zeile begonnen werden.
- Die Angabe der Gültigkeitsbereiche muss mit einem Schlüsselwort BOUND beginnen.
- Variablen, die binäres Fortmat haben müssen mit BINARY eingeleitet werden.
- Das Ende der Datei ist mit einem END anzuzeigen. Dieses Schlüsselwort ist nicht zwingend vorgeschrieben. Nachfolgend ein Beispiel einer solchen Datei.

Die Einbindung des Programms in den Compilerlauf erfolgte über einen Systemcall. Mit dem Aufruf wurde eine Datei an das Programm übergeben die es veranlasste, eine vorher generierte, standardisierte Datei "problem.lp" einzulesen, sie zu bearbeiten und die Ergebnisse in einer Textdatei abzulegen. Die Datei, die mit dem Systemcall übergeben wurde, beinhaltet folgende Befehle, die CPLEX ausführen soll:

- READ $\{filename\}\{fileformat\}$
  Dieser Befehl veranlasst CPLEX eine Datei $\{filename\}$ einzulesen. Der Parameter $\{filename\}$ spezifiziert dabei den Pfad und die Datei. Hat sie eine Endung, die CPLEX kennt, dann entfällt die Angabe des Formats $\{fileformat\}$.
- MIPOPT
  Mit diesem Befehl wird das Programm veranlasst, das im ersten Schritt eingelesene Mixed Integer Problem zu lösen.

```
MAXIMIZE
x1 - 2 x2 + 3 x3
SUBJECT TO
x2 + x3 - x1 <= 20
- 3 x2 + x3 + x1 <= 30
BOUNDS
0 <= x1 <= 40
BINARY
x2
x3
END
```

Abbildung 6.4: Beispiel einer Cplex-Eingabedatei

- DISPLAY SOLUTION VARIABLE ALL
  Hier werden alle Variablen, die nicht Null sind, als Lösungsvariablen ausgegeben
- QUIT
  Beenden des Programms und übergabe der Kontrolle an den Compiler

Alle Ausgaben, die das Programm während seiner Tätigkeit macht werden in eine Ausgabedatei geschrieben und im Anschluss analysiert. Die Ausgabe vom CPLEX-Solver hat folgendes Format:

```
Welcome to CPLEX Linear Optimizer 6.5.2
  with Mixed Integer & Barrier Solvers
Copyright (c) ILOG 1997-1999
CPLEX is a registered trademark of ILOG

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> Problem 'myCplex.lp' read.
Read time =    0.02 sec.
CPLEX> Tried aggregator 1 time.
MIP Presolve eliminated 137 rows and 12 columns.
Reduced MIP has 94 rows, 36 columns, and 222 nonzeros.
Presolve time =    0.01 sec.
```

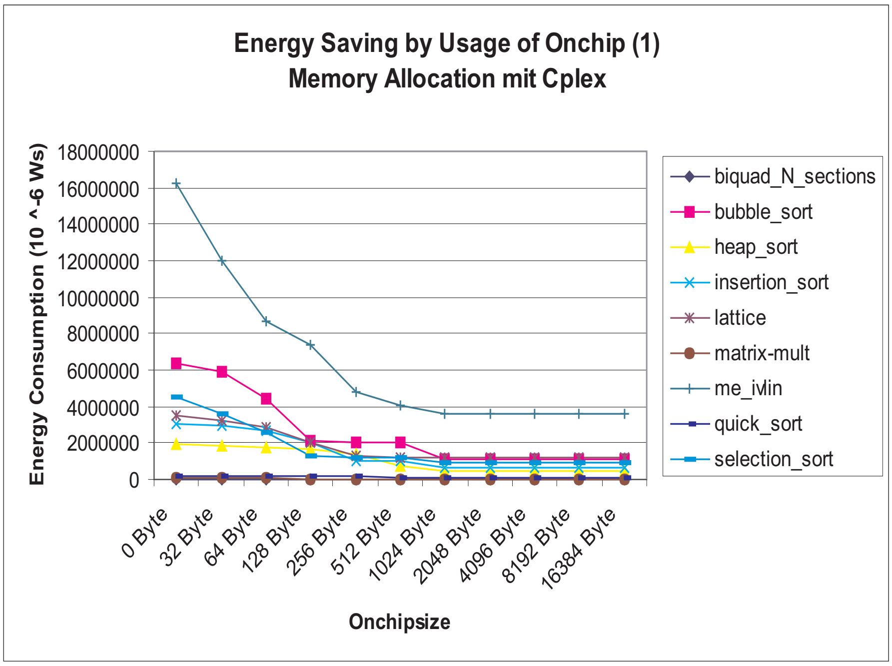
```
Clique table members: 19
Root relaxation solution time =    0.00 sec.

        Nodes                                               Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node

      0     0  1251763.1176     1                1251763.1176
               1165921.6667     1                    Cuts:  2
*              1116309.0000     0  1116309.0000      Cuts:  2

GUB cover cuts applied:  1
Clique cuts applied:  1
Cover cuts applied:  2

Integer optimal solution:  Objective =    1.1163090000e+06
Solution time =    0.01 sec.  Iterations = 4  Nodes = 0

CPLEX> Variable Name           Solution Value
%0                            1.000000
%5                            1.000000
LL37                          1.000000
_M_29                         1.000000
All other variables in the range 1-48 are zero.
```

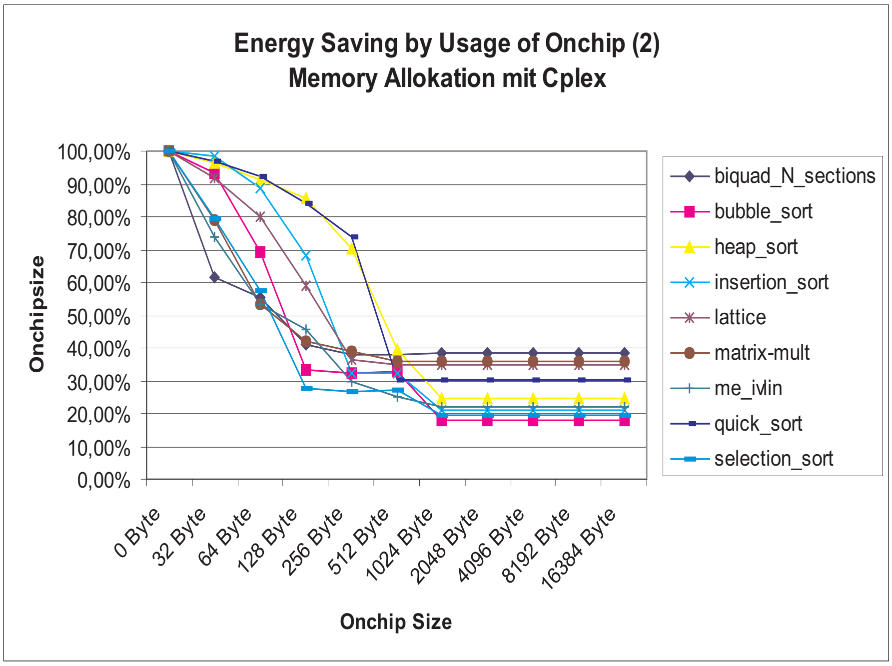
Die Analyse der Ausgabe beschränkt sich auf die Extraktion der Lösungsvariablen, die in der Augabedatei mit der Zeile "CPLEX> Variable Name" beginnen. Aufgrund von Beschränkungen, die die Läne der Variablen auf 16 Zeichen begrenzen, wurden alle $BB_{mm}$ in eine Form transferiert, die dieser Restriktion genügt. Die ersten beiden Memory-Objekte %0 und %5 sind demnach $BB_{mm}$, die letzten einzelne Basic-Blöcke LL37 und _M_29.

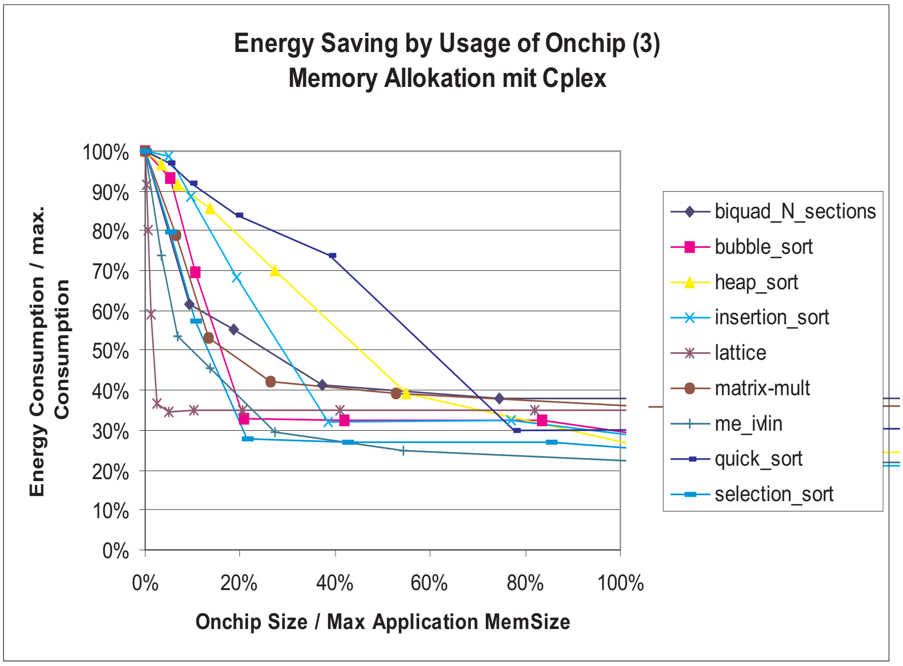
## 6.3 Ergebnisse

Die hier vorgestellten Resultate umfassen 9 Benchmarks.

Tabelle Nr. 1 stellt den absoluten Energieverbrauch aller Benchmarks


Energy Saving by Usage of Onchip (1)
Memory Allocation mit Cplex
Energy Consumption (10 ^-6 Ws)
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
0 Byte
32 Byte
64 Byte
128 Byte
256 Byte
512 Byte
1024 Byte
2048 Byte
4096 Byte
8192 Byte
16384 Byte
Onchipsize
biquad_N_sections
bubble_sort
heap_sort
insertion_sort
lattice
matrix-mult
me_ivlin
quick_sort
selection_sort


Energy Saving by Usage of Onchip (2)
Memory Allokation mit Cplex
Onchipsize
100,00%
90,00%
80,00%
70,00%
60,00%
50,00%
40,00%
30,00%
20,00%
10,00%
0,00%
0 Byte
32 Byte
64 Byte
128 Byte
256 Byte
512 Byte
1024 Byte
2048 Byte
4096 Byte
8192 Byte
16384 Byte
Onchip Size
biquad_N_sections
bubble_sort
heap_sort
insertion_sort
lattice
matrix-mult
me_ivlin
quick_sort
selection_sort



Energy Saving by Usage of Onchip (3)
Memory Allokation mit Cplex
Energy Consumption / max. Consumption
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0%
20%
40%
60%
80%
100%
Onchip Size / Max Application MemSize
biquad_N_sections
bubble_sort
heap_sort
insertion_sort
lattice
matrix-mult
me_ivlin
quick_sort
selection_sort

# Kapitel 7

# Zusammenfassung und Ausblick

Dieses Kapitel soll eine kurze Zusammenfassung der vorliegenden Diplomarbeit sein. Darüberhinaus soll ein Ausblick auf Verbesserungs- und Erweiterungsmöglichkeiten sowie eine mögliche Fortsetzung der Forschungen im Bereich "Nutzung des Scratch-Pad" gegeben werden.

## 7.1 Zusammenfassung

Wie in der Einleitung dargelegt, sind Optimierungen im Bereich der Reduktion des Energieverbrauchs auf Hardware- und Software-Seite eminent wichtig und gewinnen immer mehr an Bedeutung. Die Forderung nach Energiereduktion ergibt sich aus der Tatsache, dass immer mehr mobile Systeme entwickelt werden, deren Kapazität der Akkus nicht für die gewünschten Zeiträume zur Verfügung steht, die man als Nutzer erwartet. Dabei richtet sich der Fokus immer mehr auf Software und Optimierungen, die hierbei möglich sind.

Optimierende Compiler sind eine der wichtigsten Säulen der Softwareoptimierungen, da sie die Schnittstelle zwischen den Programmen in Hochsprache und der Hardware bilden. Sie können nicht nur zur Verbesserung des Laufzeitverhaltens sondern auch zur optimalen Nutzung der zur Verfügung stehenden Hardware genutzt werden. So kann, wie im Falle des ARM7TDMI-Prozessors, der Scratch-Pad Speicher, der deutlich weniger Energie pro Instruktions-Fetch verbraucht als der Off-Chip Speicher, dazu verwendet werden, den Energiebedarf von Programmen zu senken.

Die Optimierungstechnik Memory-Allokation versucht dazu, den On-Chip Speicher optimal zu nutzen, indem sie häufig zugegriffene Variablen und Teile von Programmen dorthin verschiebt.

Um dies zu erreichen musste ein NP-Vollständiges Problem, das des Knapsack, gelöst werden. Dazu wurden zwei Methoden näher untersucht.

Die erste Methode errechnete die Lösung mit einem pseudopolynomiellen Algorithmus, der für alle untersuchten Benchmarks zu schnellen Ergebnissen mit optimalen Lösungen führte. Als Problem hat sich, nach Erweiterung der Eingabemenge auf Multi-Basic-Blöcke, die mehrdimensionalität des Knapsack erwiesen, sodass eine weiterer Methode zur Lösung herangezogen wurde. Es war die des Integer Linear Programming, bei welcher ein Solver der Firma ILOG eingesetzt wurde. Mit Hilfe dieser Methode konnte auch das mehdimensionale Knapsack-Problem gelöst werden. Die Ergebnisse brachten eine erwartete Steigerung der vorher erzielten Resultate.

Diese Art der Optimierung hat häufig den Nachteil, dass dadurch die nötig gewordenen Änderungen des Codes das Programm insgesamt grösser machen. Das ist auch hier der Fall. Die Optimierung kann also nur zur Energie- und Laufzeitoptimierung benutzt werden. Die untersuchten Benchmarks zeigen teilweise enorme Differenzen in der Energieaufnahme. Diese Vorteile sind allerdings darauf zurückzuführen, dass sie mit der Energieaufnahme von Programmen verglichen werden, die gänzlich ohne den Scratch-Pad Speicher ausgeführt werden. So ist der Vergleich nicht ganz fair, zeigt aber dennoch die Vorteile von optimierenden Compilern.

Es wäre daher interessant diese Methode, mit der Speicherallokation mit Prozessoren mit Cache zu vergleichen, wo die Ein- und Auslagerung von Blöcken hardwaremässig gesteuert wird. Erst dann kann sich zeigen, wie effektiv diese Art der Allokation des On-Chip Speichers im Vergleich mit bekannten Cache-Ersetzungsstrategien ist. Zeigt sich, dass der optimierende Compiler hier bessere Ergebnisse erzielt, könnte eine Folgerung daraus sein, bestimmte eingebettete Systeme mit Prozessoren mit Scratch-Pad anstatt Cache-Speichern auszurüsten.

## 7.2 Ausblick

Die vorgestellte Methode basiert auf einer statischen Analyse des Programms. Es werden alle möglichen Pfade des Kontrollflusses mit 50%-Wahrscheinlichkeit belegt. Schleifen werden mit einer festen Ausführungshäufigkeit versehen. Das ist nicht der beste Analyseansatz. Eine Verbesserung der Genauigkeit könnte dadurch erzielt werden, die Analyse des Programms dynamisch durchzuführen.

Ein Ansatz zur Verbesserung der erzielten Ergebnisse könnte die dynamische Allokation des Speichers darstellen. Nach statischer oder dynamischer Analyse müsste untersucht werden, inwiefern das Ein- und Auslagern von Speicherobjekten zur Laufzeit, mit dem Overhead für das Kopieren der zu transferierenden Objekte, zur Steigerung der Effizienz beitragen kann.

# Anhang A

# Thumb-Instruktionssatz

| Assemblerinstruktion | Takt-Zyk-len | Speicher(Bytes) Instruk tion | Daten | Auswirkungen |
|---|---|---|---|---|
| ADC Rd,Rs | 1 | 2 | 0 | Rd=Rd + Rs + C-Bit |
| ADD Hd,Hs | 1 | 2 | 0 | Hd=Hd + Hs |
| ADD Hd,Rs | 1 | 2 | 0 | Hd=Hd + Rs |
| ADD Rd,#Offset8 | 1 | 2 | 0 | Rd=Rd + imm8 |
| ADD Rd,Hs | 1 | 2 | 0 | Rd=Rd + Hs |
| ADD Rd,PC,#Imm | 1 | 2 | 0 | add from adress from PC |
| ADD Rd,Rs#Offset3 | 1 | 2 | 0 | Rs=Rs + imm3 |
| ADD Rd,Rs,Rn | 1 | 2 | 0 | Rd=Rs + Rn |
| ADD Rd,Rs | 1 | 2 | 0 | Rd=SP + imm8$\cdot$ 4 |
| ADD Rd,Rs | 1 | 2 | 0 | SP=SP + imm7$\cdot$ 4 |
| ADD Rd,Rs | 1 | 2 | 0 | SP=SP - imm7$\cdot$ 4 |
| AND Rd,Rs | 1 | 2 | 0 | Rd=Rd AND Rs |
| ASR Rd,Rs,#Offset5 | 1 | 2 | 0 | arithmetic shift right |
| ASR Rd,Rs | 2 | 2 | 0 | arithmetic shift right |
| B | 3 | 2 | 0 | Rd=unconditional branch |
| BIC Rd,Rs | 1 | 2 | 0 | Rd=Rd AND NOT Rm |
| BL | 4 | 4 | 0 | long branch with link |
| BX Hs | 3 | 2 | 0 | R15=Hs AND 0xFFFFFFFF |
| BX Rs | 3 | 2 | 0 | R15=Rs AND 0xFFFFFFFE |
| Bxx | 1 | 3 | 0 | conditional branch |
| CMN Rd,Rs | 1 | 2 | 0 | compare negativ |
| CMP Hd,Hs | 1 | 2 | 0 | compare registers |
| CMP Hd,Rs | 1 | 2 | 0 | compare registers |
| CMP Rd,#Offset8 | 1 | 2 | 0 | compare immediate |
| CMP Rd,Rs | 1 | 2 | 0 | compare registers |
| EOR Rd,Rs | 1 | 2 | 0 | Rd=Rd EOR Rs |
| LDMIA Rb!,Rlist | 2-10 | 2 | 0-32 | load list of registers |
| LDR Rd,[PC,#Imm] | 3 | 2 | 4 | load PC-relativ |
| LDR Rd,[Rb,#Imm] | 3 | 2 | 4 | Rd=[Rb+imm5 $\cdot$ 4] |
| LDR Rd,[Rb,Ro] | 3 | 2 | 4 | Rd=[Rb+Ro](word) |
| LDR Rd,[SP,#Imm] | 3 | 2 | 4 | load PC-relativ |
| LDRB Rd,[Rb,#Imm] | 3 | 2 | 1 | load PC-relativ |
| LDRH Rd,[Rb,Ro] | 3 | 2 | 2 | Rd=[Rb+Ro] (halfword) |

| Assemblerinstruktion | Takt-Zyklen | Speicher(Bytes) | | Auswirkungen |
|---|---|---|---|---|
| | | Instruktion | Daten | |
| LDRB Rd,[Rb,Ro] | 3 | 2 | 1 | load PC-relativ |
| LDRH Rd,[Rb,#Imm] | 3 | 2 | 2 | Rd=[Rb+imm5·2] |
| LDSRB Rd,[Rb,Ro] | 3 | 2 | 1 | load signed byte |
| LDSRB Rd,[Rb,Ro] | 3 | 2 | 2 | load signed halfword |
| LSL Rd,Rs,#Offset5 | 1 | 2 | 0 | Rd=Rs << imm5 |
| LSL Rd,Rs | 2 | 2 | 0 | Rd=Rd << Rs |
| LSR Rd,Rs,#Offset5 | 1 | 2 | 0 | Rd=Rs << imm5 |
| LSR Rd,Rs | 2 | 2 | 0 | Rd=Rd << Rs |
| MOV Hd,Hs | 1 | 2 | 0 | Hd=Hs |
| MOV Hd,Rs | 1 | 2 | 0 | Hd=Hs |
| MOV Hd,#Offset8 | 1 | 2 | 0 | Rd=imm8 |
| MOV Rd,Hs | 1 | 2 | 0 | Rd=Hs |
| MUL Rd,Rs | 2-5 | 2 | 0 | Rd=Rs·Rd |
| MVN Rd,Rs | 1 | 2 | 0 | Rd=NOT Rs |
| NEG Rd,Rs | 1 | 2 | 0 | Rd=-Rs |
| ORR Rd,Rs | 1 | 2 | 0 | Rd=Rd OR Rs |
| POP Rlist,PC | 5 | 13 | 4-36 | pop and return |
| POP Rlist | 2-10 | 2 | 0-32 | pop register from stack |
| PUSH RegList,LR | 2-10 | 2 | 4-36 | push LR and register |
| PUSH RegList | 1-9 | 2 | 4-46 | push register onto stack |
| ROR Rd,Rs | 2 | 2 | 0 | rotate right |
| SBC Rd,Rs | 1 | 2 | 0 | Rd=Rd - Rs + Rs + C-Bit |
| STMIA Rb!,Rlist | 1-9 | 2 | 0-32 | store list of registers |
| STR Rd,[Rb,#Imm] | 2 | 2 | 4 | [Rb+imm5·4]=Rd |
| STR Rd,[Rb,Ro] | 2 | 2 | 4 | [Rb+Ro]=Rd (word) |
| STR Rd,[SP,#Imm] | 2 | 2 | 4 | store SP-relativ |
| STRB Rd,[Rb,Ro] | 2 | 2 | 4 | [Rb+imm5]=Rd (byte) |
| STRH Rd,[Rb,#Imm] | 2 | 2 | 4 | [Rb+Ro]=Rd (byte) |
| STRH Rd,[Rb,#Imm] | 2 | 2 | 4 | [Rb+imm5·2]=Rd |
| SUB Rd,#Offset8 | 1 | 2 | 0 | [Rb+Ro]=Rd (halfword) |
| SUB Rd,Rs,#Offset3 | 1 | 2 | 0 | store SP-relativ |
| SUB Rd,Rs,Rn | 1 | 2 | 0 | Rd=Rs-Rn |
| SWI Value8 | 3 | 2 | 0 | software interrupt |
| TST Rd,Rs | 1 | 2 | 0 | test bits |

# Literaturverzeichnis


[ARML99] ADVANCES RISC MACHINES LTD, ARM: *Thumb Instructions Set Quick Reference Card.* ARM QRB 0001D,Cambridge, England, 1999.

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

[AH88a] A. Aho, R. Sethi, J. D. Ullman: *Compilerbau-Teil 1*, Addison-Wesley Verlag, 1988.

[AH88b] A. Aho, R. Sethi, J. D. Ullman: *Compilerbau-Teil 2*, Addison-Wesley Verlag, Massachusetts, 1988.

[KU74] E. Kühn: *Handbuch TTL- und CMOS-Schaltungen*, 4 Auflage, Hüting Buch Verlag Heidelberg, 1993.

[PA97] P. R. Panda, N. Dutt, A. Nicolau: *Effizient utilization of scratch-pad memory in embedded processor application*, In Proc. of European Design and Test Conference, Paris, France, March 1997.

[PA99] P. R. Panda, N. Dutt, A. Nicolau: *Memory Issues in Emdedded Systems-On-Chip.*, Kluwer Academic Publishers, Norwell, MA, 1999.

[SC00] R.Schwarz: *Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Diplomarbeit, 2000.

[SJ98] J. Sjödin, B. Frödeberg, T. Lindgren: *Allocation of Global Data Object in On-Chip RAM.* In Proc. Workshop on Compiler and Architectural Support for Embedded Computer Systems, Washington DC. 1998. ACM.

[ST01] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, P. Marwedel: *Comparison of cache- and scratch-pad-based memory systems with respect to performance, area and energy consumption*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Technical Report, TR 762, 2001.

[ST01a] S. Steinke, Ch. Zobiegala, L. Wehmeyer, P. Marwedel: *Moving Program Objects to Scratch-Pad Memory for Energy Reduction*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Technical Report, TR 756, 2001.

[SY96] Synopsys, Inc.: *Power Product Reference Manual*, Version 3.5, 1996.

[TH00] M. Theokharidis: *Energiemessung von ARM7TDMI Prozessor-Instruktionen* Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Diplomarbeit, 2000.

[TI94] V.Tiwari, S. Malik, A. Wolfe: *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization*, IEEE Transaktion on VLSI System, 2(4):1994.

[WE92] I. Wegener: *Grundbegriffe der theoretischen Informatik*, Universität Dortmund, Fakultät Informatik, Lehrstuhl II, 1992.

[WE00] I. Wegener: *Effiziente Algorithmen*, Universität Dortmund, Fakultät Informatik, Lehrstuhl II, 2000.

[ZI74] S. Zionts: *Linear and Integer Programming*, Prentice-Hall Incorporation, Upper Saddle River, NJ, 1974.