Diplomarbeit

Compilergestützte Optimierung von Zugriffen auf partitionierte Speicher

Urs Helmig

Diplomarbeit
am Lehrstuhl 12
des Fachbereichs Informatik
der Universität Dortmund

30. März 2004

Betreuer:
Dipl.-Inf. Lars Wehmeyer
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung in die Thematik

Technische Entwicklungen und gesellschaftliche Bedürfnisse stehen seit jeher in einem wechselseitigen Verhältnis. Mehr Flexibilität und Mobilität kennzeichnen die aktuellen Anforderungen der modernen Informationsgesellschaft. Der moderne Mensch muss immer und überall erreichbar sein, sich und andere über aktuelle Ereignisse und Planungsvorgänge informieren. Es ist mittlerweile jedem offensichtlich, dass unser Leben von zahlreichen technischen Systemen abhängig geworden ist, derer wir uns jedoch im alltäglichen Gebrauch oft gar nicht bewusst sind.

Gezielt eingesetzte Werbung und der zunehmende harte Wettbewerb zwischen den Anbietern schüren Bedürfnisse, die zuvor nicht existierten. Diese Entwicklung kann vor allem bei den Privatkonsumenten beobachtet werden. War die Hauptaufgabe eines Mobiltelefons ursprünglich das Telefonieren, sind heute Funktionen wie die integrierte Kamera und die Möglichkeit Fotos per MMS[1] zu verschicken marktprägend.

Aufgrund dieser Tatsache kommen die Entwicklungsabteilungen in Zugzwang, den geweckten Bedürfnissen in immer kürzeren Zyklen nachzukommen. Obwohl die technische Entwicklung in den letzten Jahrzehnten rapide Fortschritte erzielt hat, ist es bis heute unmöglich ein Produkt zu entwickeln, das allen Anforderungen gerecht wird.

Diesbezüglich entstehen bei der Integration der unabhängig entwickelten Komponenten zahlreiche Probleme. So müssen zum Beispiel die Hersteller von Mobiltelefonen Kameras in ihre Produkte integrieren, wobei die Telefone gleichzeitig nicht größer, schwerer und teurer werden dürfen. Neben diesen offensichtlichen Problemen existieren weitere weitreichendere Probleme, die zunehmend in den Blickpunkt des Kundeninteresses gelangen. Dies betrifft insbesondere die Benutzerfreundlichkeit des jeweiligen Gerätes, was sich sehr anschaulich am Beispiel

[1] Multimedia Messaging Service

der Betriebsdauer eines Mobiltelefons darstellen lässt. War es vor einigen Jahren durchaus üblich, dass die Akkus gute fünf Tage den Betrieb ermöglichten, ist es heutzutage aufgrund der zunehmenden Verkleinerung und des steigenden Energiehungers der Applikationen kaum noch möglich, das Gerät länger als ein paar Stunden ohne Netz zu betreiben [Opi03]. Die zusätzlichen neuen Komponenten tun ihr übriges.

Aus diesem Grunde stehen die Anbieter vor einem Problem. Die Kunden verlangen neue Applikationen und zusätzliche Komponenten unter Beibehaltung der alten Leistungsmerkmale. Dies ist technisch jedoch nahezu unmöglich, da eines der Probleme darin besteht, dass die Entwicklung neuer Akkus mit größerer Kapazität nicht in der Geschwindigkeit erfolgt wie der Energiebedarf der neuen Produkte steigt. Dieser Trend wird zusätzlich durch die Forderung nach immer kleineren Produkten verstärkt, da die Kapazität eines Akkus maßgeblich von dessen Größe beeinflusst wird. Da es also offensichtlich ist, dass die Energiezufuhr nicht erhöht werden kann, besteht als Ausweg lediglich die Option Energie zu sparen.

In der Vergangenheit wurde die Lösung dieses Problems den Hardwareherstellern zugeschrieben. Diese allein sind mit dieser Aufgabe jedoch überfordert, da immer mehr Funktionalität aus Kosten- und Flexibilitätsgründen in Software realisiert wird. Hardwareseitige Energiesparmaßnahmen können demnach nur in Zusammenarbeit mit einer ressourcensparenden Software erfolgreich ihr Ziel erreichen. Aus diesem Grunde sind die Softwarehersteller ebenfalls gefordert, ihren Beitrag zur Senkung des Energiebedarfs beizusteuern.

Durch den zunehmenden Kostendruck, dem sich die Softwarehersteller beugen müssen, ist es ihnen allerdings nicht möglich, jeweils einen Spezialisten für jede verwendete Komponente einzustellen. Speziell angepasste Software sowie handoptimierte Assemblerprogramme stellen somit im allgemeinen keinen gangbaren Weg mehr dar, da dieser Lösungsansatz in der Regel viel zu kostspielig wäre.

Als Alternative verbleibt nur die Möglichkeit, die Komponenten in Hochsprachen zu programmieren, wobei die Hochsprache anschließend von einem Compiler in die Sprache der Komponente übersetzt werden muss. Dies hat den Nachteil, dass allgemeine Compiler im allgemeinen sehr schlechten Code erzeugen, da viele Komponenten für ihr Anwendungsgebiet speziell entwickelt worden sind und die Compiler somit keine ausreichenden Kenntnisse über die zur Verfügung stehende Spezial-

hardware besitzen. Einen Ausweg aus diesem Dilemma bieten speziell angepasste Compiler, die mit bestimmten Optimierungsmethoden versuchen, die Spezialhardware für das zu lösende Problem auszunutzen. Dies ist selbstverständlich nur möglich, wenn die Compiler permanent weiterentwickelt werden, neue Architekturen schnell unterstützt werden und die Forschung neue, bessere Optimierungmethoden entwickelt und zur Verfügung stellt.

Zusammenfassend lässt sich sagen, dass der Energieverbrauch mobiler Systeme ein bislang ungelöstes Problem darstellt, wobei die folgenden Aspekte des Energieverbrauchs neben den bereits erwähnten ebenfalls nicht zu vernachlässigen sind [Zob01, Gru02]

- **Kritische Wärmeentwicklung**
  Elektronische Systeme, die Energie aufnehmen, müssen diese in Form von Wärme wieder abgeben, wobei sie bedingt durch die steigende Energieaufnahme immer wärmer werden. Diese Wärme muss durch aufwendige Kühlungsmaßnahmen abgeführt werden, da ansonsten kein zuverlässiger Betrieb der Systeme möglich wäre. Diese Tatsache schränkt die Einsatzmöglichkeiten der Systeme ein, da zum Beispiel niemand während einer Zugfahrt mit einem zu heißen Laptop auf dem Schoß arbeiten möchte. Des weiteren verursacht die übliche Luftkühlung durch Ventilatoren Lärm, der in leisen Umgebungen als sehr störend empfunden wird.

- **Abnehmende Zuverlässigkeit**
  Die große Wärmeentwicklung der Systeme führt zu einer sehr hohen Materialbeanspruchung, die sich langfristig auf die Zuverlässigkeit auswirkt. Ohne aufwendige Kühlungsmaßnahmen würde kein System längere Zeit zuverlässig arbeiten. Mittlerweile sind jedoch die Kühlungsmaßnahmen selbst zu einem kritischen Faktor für die Zuverlässigkeit geworden, da ein Lüfterausfall durchaus den Ausfall des gesamten Systems bewirken kann.

- **Energiekosten**
  Jedes einzelne mobile System verbraucht für sich genommen in der Regel nur wenig Energie. Bedingt durch die ständig steigende Anzahl ist der Gesamtenergieverbrauch jedoch nicht mehr zu vernachlässigen.

Bei Untersuchungen, die die Ursachen des Energieverbrauchs genauer untersucht haben, hat sich gezeigt, dass ein großer Teil der Energie im Speichersystem elektronischer Systeme verbraucht wird. Dieser Energieverbrauch lässt sich jedoch, wie die Arbeiten [Zob01, Gru02] gezeigt haben, durch entsprechende Software reduzieren. Aus diesem Grunde versucht diese Arbeit einen Beitrag zur Lösung dieses Problems zu liefern, indem sie die in [Zob01] vorgestellte Compiler-Optimierungsmethode mit neuen Ideen weiterentwickelt.

1.1 Inhalt der Arbeit

»Compilergestützte Optimierung von Zugriffen auf partitionierte Speicher« lautet das Thema dieser Arbeit. Das Ziel ist die Untersuchung des Energiesparpotentials, das durch die Nutzung partitionierter Speicher gewonnen werden kann. Dieses Energiesparpotential entsteht, da kleinere Speicher bei einem Zugriff weniger Energie benötigen als größere. Aus dieser Tatsache entstand die Idee, die vorhandenen Speicher in mehrere kleine Speicher zu zerteilen und einen Compiler mit einer Optimierungsmethode, die in der Lage ist ein Programm auf die kleineren Speicher zu verteilen, auszustatten.

Aus diesem Grunde wurde im Rahmen dieser Arbeit eine entsprechende Compiler-Optimierungsmethode entwickelt und das mögliche Energiesparpotential untersucht. Um diese Untersuchungen durchzuführen mussten zuvor einige Überlegungen über sinnvolle mögliche Partitionierungsmöglichkeiten durchgeführt werden.

1.2 Aufbau der Arbeit

Bezogen auf die Zielsetzung dieser Arbeit »Compilergestützte Optimierung von Zugriffen auf partitionierte Speicher« sind einige technische Erläuterungen unabdinglich, da sie grundlegende Zusammenhänge veranschaulichen sowie das entsprechende Fachvokabular einführen. Daher ist dem Entwicklungs- und Evaluationsteil dieser Arbeit das Kapitel 2 vorangestellt. Selbiges gilt für eine Anzahl ausgewählter forschungsverwandter Arbeiten; auch sie gehen dem eigentlichen Hauptteil dieser Arbeit voraus, da sie den Ausgangspunkt für die vorzustellende Optimierungsmethode beinhalten.

Der eigentliche Hauptteil der Arbeit besteht aus zwei Teilen. Der erste Teil stellt die Idee, die hinter der Optimierungsmethode steckt, detailliert vor (Kapitel 4, 5 und 6). Er wurde zugleich als Konzept zur Implementierung der Optimierungsmethode verwendet. Der zweite Teil hingegen stellt die durch Simulation gewonnenen Ergebnisse vor und wertet diese hinsichtlich der Problemstellung aus (Kapitel 7).

2 Technische Grundlagen

Dieses Kapitel erläutert die für das Verständnis der entwickelten Compiler-Optimierungsmethode notwendigen technischen Grundlagen und stellt die Zusammenhänge zwischen den einzelnen Themengebieten dar.

2.1 Eingebettete Systeme

Unter »Eingebetteten Systemen«, kurz *ES*, versteht man Systeme, die in der Regel ohne Bildschirm und Tastatur auskommen und ihre Arbeit normalerweise im Hintergrund erledigen [Mar00].

Durch die fortschreitende Entwicklung wird die oben genannte Definition jedoch immer mehr aufgeweicht, da viele technische Systeme wie Mobiltelefone oder Geldautomaten sehr wohl als *ES* zu bezeichnen sind. Eine bessere Definition könnte demnach wie folgt lauten:

> Eingebettete Systeme sind informationsverarbeitende Systeme, die in größere Systeme eingebettet sind und bei denen die Informationsverarbeitung dem Benutzer – anders als beim PC – in der Regel nicht direkt sichtbar ist. Eingebettete Systeme reagieren meist zustandsabhängig (sie sind „reaktiv“). Sie stellen besondere Anforderungen an die Zuverlässigkeit, die Sicherheit, das Einhalten von Zeitbedingungen und die Effizienz (zum Beispiel hinsichtlich des Energieverbrauchs, des Gewichts und der Kosten). Zur Steuerung und Regelung eingesetzte eingebettete Systeme verfügen meist über Sensoren und Aktoren [Mar00].

Im Gegensatz zu normalen PCs werden an *ES* vielfach sehr hohe Anforderungen gestellt. Diese variieren je nach dem Anwendungsgebiet jedoch sehr stark. Zusammenfassend können jedoch folgende Eigenschaften als typisch für eingebettete Systeme bezeichnet werden [Mar00]:

- *ES* sind für bestimmte Anwendungen entworfen, wobei in der Regel keine vollkommen neuen Anwendungsprogramme zu späteren Zeitpunkten hinzukommen.

- *ES* müssen effizient sein. Das heißt, dass zum Beispiel mobile Geräte mit wenig Energie auskommen müssen.

Des weiteren werden wegen der Bedeutung, die die *ES* in der heutigen Gesellschaft spielen – auch wenn sich die meisten Menschen dieser Bedeutung nicht bewusst sind – sehr hohe Anforderungen an die eingesetzten Systeme gestellt. Unter anderen sind dies [Mar00]:

- **Zuverlässigkeit**
  Die Wahrscheinlichkeit eines System-Ausfalls muss klein sein.

- **Sicherheit**
  Falls ein System ausfällt, dürfen keine gefährlichen Situationen eintreten.

- **Wartbarkeit**
  Ein ausgefallenes System muss schnell wieder zur Verfügung stehen.

- **Verfügbarkeit**
  Die Wahrscheinlichkeit eines nicht arbeitsfähigen Systems muss klein sein.

- **Datensicherheit**
  Die Vertraulichkeit und Authentizität von Daten muss zu jedem Zeitpunkt gewährleistet sein.

Aus den oben dargestellten Eigenschaften und Anforderungen ergeben sich für die Entwicklungsabteilungen einige Probleme, die es im Rahmen des üblichen Budgets zu bewältigen gilt. Handoptimierte Lösungen beziehungsweise Speziallösungen, die die oben genannten Anforderungen erfüllen, sind deshalb normalerweise nicht zu realisieren. Aus diesem Grunde bietet sich der Einsatz standardisierter Entwicklungswerkzeuge an, die mit aufwendigen Optimierungsmethoden und vorgefertigten Softwarekomponenten versuchen, die Entwickler zu entlasten.

2.2 Hardware elektronischer Systeme

Heutige Systeme bestehen aus einer Vielzahl von Komponenten, wobei die zentralen Stellen meistens durch Prozessoren besetzt sind, die über Bus-Systeme mit anderen Komponenten verbunden sind. Speziell entwickelte Hardware wird immer seltener eingesetzt, da Prozessoren durch entsprechende Programmierung sehr flexibel und kostengünstig eingesetzt werden können. Die Verfügbarkeit von Entwicklungswerkzeugen und Compilern ist ebenfalls gewährleistet. Des weiteren lassen sich Fehler durch Softwareupdates schnell und einfach beheben [Mar00].

2.2.1 Prozessoren

Durch die zunehmende Beliebtheit von Prozessoren stehen heute viele, sehr unterschiedliche Prozessoren zur Verfügung, so dass sich für ein zu lösendes Problem eigentlich immer ein geeigneter Prozessor finden lassen sollte, wobei traditionell folgende Arten von Prozessoren unterschieden werden [Sch00b]:

- **Allgemeine Prozessoren**
  Es gibt eine Vielzahl von unterschiedlichen Prozessoren, die neben der CPU unter anderem meist Speicher und Schnittstellen enthalten, wobei die Prozessormerkmale je nach Anwendungsgebiet sehr stark variieren. Im PC-Bereich ist hauptsächlich die Taktfrequenz wichtig, da die Marketingabteilungen die Taktfrequenz als hauptsächliches Werbeargument benutzen. In anderen Bereichen hingegen sind Eigenschaften wie die Effizienz wichtiger, wobei der Begriff Effizienz sich auf das jeweilige Anwendungsgebiet bezieht. Bei mobilen Systemen würde er zum Beispiel den Energiebedarf beschreiben.

- **Mikro-Controller**
  Bei Mikro-Controllern handelt es sich um leicht veränderte Prozessoren, die in *ES* verwendet werden und im Gegensatz zu den allgemeinen Prozessoren zusätzliche Elemente wie Timer, I/O-Anschlüsse oder Analog/Digital-Wandler enthalten. Sie werden sehr häufig für Steuerungsaufgaben eingesetzt, bei denen normalerweise nur verhältnismäßig geringe Anforderungen an den Datendurchsatz gestellt werden.

- **Signalprozessoren**
  Signalprozessoren sind Prozessoren, die spezielle Algorithmen für gezielte Anwendungen implementieren. Neben dem eigentlichen Prozessor enthält ein Rechner heute oft mehrere Signalprozessoren. Sie sind spezialisierter als Allgemeine Prozessoren und übernehmen sehr unterschiedliche Aufgaben. So gibt es Signalprozessoren zum Beispiel für die Audioverarbeitung sowie für die Netzwerkanbindung.

Diese traditionelle Unterscheidung wird jedoch durch die Miniaturisierung und die fortschreitende Veränderung der Prozessoren zu bloßen Modulen für kundenspezifische Anforderungen aufgeweicht, so dass bei modernen Prozessoren eine eindeutige Zuordnung in eine der genannten Kategorien nicht immer möglich ist.

2.2.2 Speicher

Ein ideales System besitzt einen beliebig großen Speicher, der gleichzeitig eine beliebig kleine Zugriffszeit hat, so dass sich kein negativer Einfluss auf die Systemleistung ergibt. Allgemein gilt, dass ein Speichersystem mit hoher Zugriffszeit zu einer hohen Ausführungszeit der Programme und damit zu einer geringen Effizienz des Systems führt [Sch00b].

Aus diesem Grunde wäre ein Rechner, der ein Speichersystem mit folgenden Eigenschaften hätte, wünschenswert:

- beliebig große Kapazität
- beliebig kleine Zugriffszeit
- beliebig großer Datendurchsatz
- minimale Kosten (sowohl bei der Produktion als auch im Betrieb)

Um dieses Ziel zu erreichen wird ein sehr großer technischer Aufwand betrieben, da bedingt durch die schnelle technische Weiterentwicklung der Prozessoren auf der einen Seite und der nicht mithaltenden Entwicklung der Speichersysteme auf der anderen Seite, das grundsätzliche Problem der Zugriffszeit immer größer wird. Dieses Phänomen wird als *Memory Performance Gap* (siehe Abbildung 2.1 [Sch01a])

bezeichnet. Bezüglich des Datendurchsatzes und der Kapazität existieren diese Probleme nicht, da die Speicherentwicklung in diesem Bereich die Anforderungen der Prozessoren erfüllt.

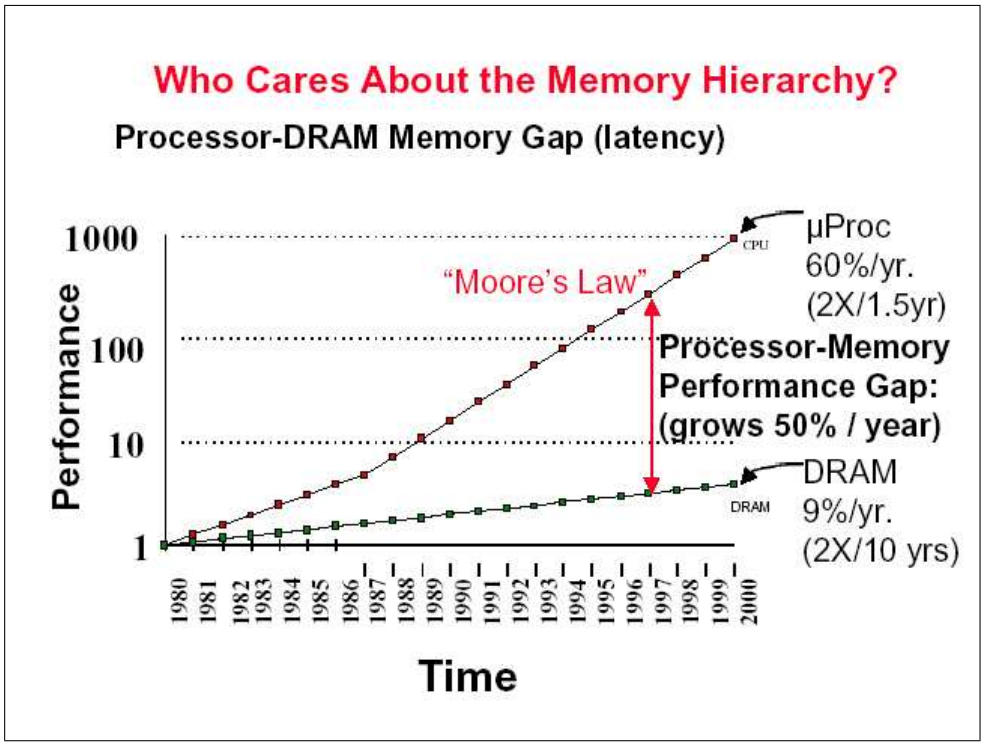
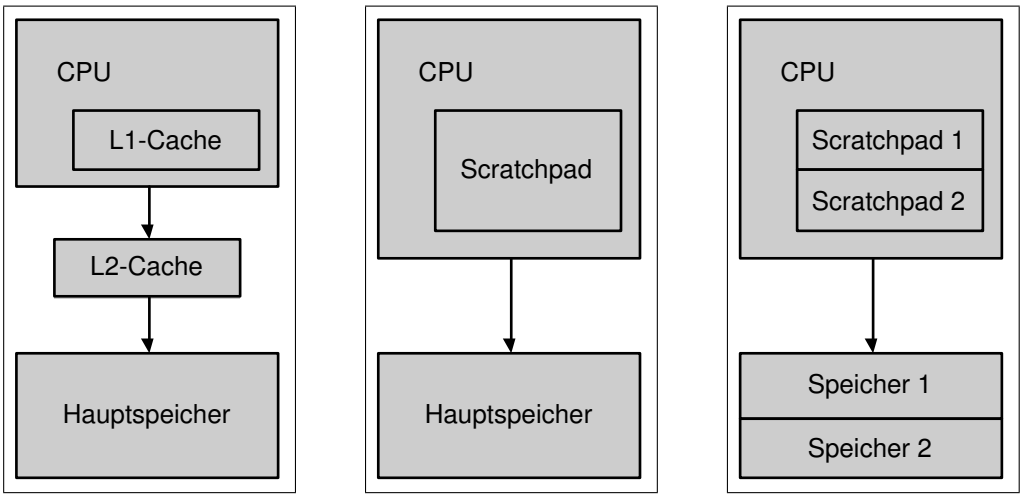


Abbildung 2.1: Memory Performance Gap [Sch01a]

Wie aus der Abbildung 2.1 ersichtlich wird, ist der Entwicklungsfortschritt bezüglich der Geschwindigkeit bei Prozessoren und DRAM-Speicher sehr unterschiedlich. Das heißt, dass die Prozessorgeschwindigkeit ca. 60% pro Jahr wächst. Die Geschwindigkeit von DRAM-Speicher hingegen wächst nur ca. 9% pro Jahr. Aus diesen beiden Tatsachen ergibt sich das Problem, dass die Prozessorgeschwindigkeit ca. 50% schneller zunimmt als die Speichergeschwindigkeit. Es ist also nur eine Frage der Zeit, dass die Prozessoren ihre Zeit nur noch mit Warten verbringen. Die Systemgeschwindigkeit würde dann nur noch von der Speichergewindigkeit bestimmt [WM94]. Die Forschung versucht deshalb bessere und schnellere Speichersysteme zu entwickeln, wobei diese Versuche bislang nicht den gewünschten Erfolg erzielt haben und von daher keine baldige Trendwende zu erwarten ist.

Aus diesem Grunde können nur die Methoden, die das Problem bislang kompen-

siert haben, verbessert und weiterentwickelt werden. Die Abbildung 2.2 zeigt, wie unterschiedliche Speichertechnologien in Form von Speicherhierarchien gemeinsam zur Erhöhung der Systemleistung eingesetzt werden können.



(a) Klassisch (b) Atmel AT91M40400 (c) Konzeptionell

Abbildung 2.2: Speicherhierarchien

Der Einsatz dieser Methoden geht jedoch mit neuen Problemen einher, da diese das grundsätzliche Problem nicht beheben. Es wird lediglich versucht, die Auswirkungen zu verstecken, indem zum Beispiel wie in Abbildung 2.2(a) dargestellt, Caches die durchschnittliche Wartezeit bei einem Speicherzugriff reduzieren sollen. Ihre Funktionsweise ist in dem gleichnamigen Unterkapitel beschrieben.

Eine andere, hardwaretechnisch einfachere Lösung, die vor allem im Bereich der *ES* immer beliebter wird, ist die Integration des Arbeitsspeichers in den Prozessor (siehe Abbildung 2.2(b)). Diese Lösung setzt jedoch speziell angepasste Software voraus, da sich ansonsten keine Vorteile aus dem integrierten Arbeitsspeicher ergeben. Die Funktionsweise dieser Lösung ist ebenfalls in einem der folgenden Unterkapitel beschrieben.

Die Evaluation der in Abbildung 2.2(c) dargestellten Möglichkeit ist das Thema dieser Arbeit und wird detailliert im Kapitel 4 beschrieben.

Caches

»Caches« sind kleine schnelle Zwischenspeicher, die sich typischerweise auf dem Prozessor-Chip befinden und den langsamen Zugriff auf den großen Hauptspeicher verstecken sollen. Um die Wirkungsweise von Caches zu erklären muss als erstes die Struktur von Speicherzugriffen bei typischen Programmen näher betrachtet werden. Dabei fallen folgende Eigenschaften auf [HP96]:

- **Zeitliche Lokalität**
  Falls ein Datum oder eine Instruktion gerade verwendet wurde, wird es oder sie mit hoher Wahrscheinlichkeit bald wieder verwendet werden.
- **Örtliche Lokalität**
  Falls ein Datum oder eine Instruktion verwendet wurde, wird wahrscheinlich bald ein benachbartes Datum oder eine benachbarte Instruktion verwendet werden.

Diese Eigenschaften ermöglichen es, kleine Programmteile beziehungsweise Datenausschnitte in kleine Zwischenspeicher zu kopieren. Bei nachfolgenden Zugriffen muss dann nur auf den entsprechenden Zwischenspeicher zugegriffen werden, da die Wahrscheinlichkeit, dass die Daten sich im Zwischenspeicher befinden, sehr groß ist. Nachteilig ist jedoch, dass die Verwaltungslogik sehr aufwendig ist. Sie muss unter anderem feststellen, ob die Daten sich bei einem Zugriff bereits im Zwischenspeicher befinden. Des weiteren muss sie entscheiden, welche Daten in den Speicher zurückgeschrieben werden müssen, wenn der Zwischenspeicher keine ausreichende Kapazität mehr zur Verfügung hat. Eine genauere Darstellung dieser Problematik kann zum Beispiel [HP96] entnommen werden.

Scratchpad-Speicher

In den letzten Jahren kann im Bereich der *ES* der Trend zur Integration des Arbeitsspeichers in den Prozessor beobachtet werden. Dies wird von der Logik-Seite kommend als »On-Chip-Memory« oder »Embedded RAM« bezeichnet, während die Speicherhersteller dagegen von »intelligentem RAM« sprechen [Fre98]. Neben diesen Bezeichnungen sind die Bezeichnungen »Scratchpad-Speicher«, kurz

SPM [Mar03] oder »Compiler-Controlled-Memory«, kurz CCM [CH98], ebenfalls gebräuchlich.

Aus der Sicht des Prozessors ist ein Scratchpad-Speicher lediglich ein frei adressierbarer Speicherbereich, der an einer bestimmten Basisadresse beginnt und der eine bestimmte Größe hat. Im Gegensatz zu Caches benötigen Scratchpad-Speicher allerdings keine aufwendige Verwaltungslogik. Sie sind deshalb wesentlich einfacher herzustellen, nehmen weniger Siliziumfläche ein und verbrauchen weniger Energie.

Durch die fehlende Verwaltungslogik profitieren jedoch nur speziell angepasste Programme von den Scratchpad-Speichern, da diese die Adressierung übernehmen müssen. Die Anpassung der Programme kann manuell durch einen Programmierer oder automatisch durch einen Compiler geschehen.

2.3 Energie

Für die Entwicklung einer Optimierungsmethode ist ein Modell, das das reale Problem beschreibt und formalisiert, unabdingbar. Aus diesem Grunde beschreibt dieses Kapitel die Ursachen des Energieverbrauchs in elektronischen Systemen und gibt einige Anregungen, wie sich der Energieverbrauch reduzieren lässt.

Bei einer Betrachtung gängiger Hardware fällt auf, dass die meisten Systeme aus Komponenten zusammengesetzt sind. Typische Komponenten sind unter anderen:

- Stromversorgung
- Motherboard mit Prozessor, Speicher und I/O-Funktionen
- Massenspeicher wie Festplatten und Floppy-Laufwerke
- Bildschirm und Tastatur

Der mögliche Einfluss einer Compiler-Optimierungsmethode zur Reduktion des Energieverbrauchs beschränkt sich jedoch auf die Komponenten, die unmittelbar mit der Programmausführung in Verbindung stehen. Dies sind vor allem der Prozessor und der Speicher, die unterschiedlich viel Energie in Abhängigkeit vom

ausgeführten Programm benötigen. Die Energieaufnahme der anderen Komponenten wie Bildschirm und Tastatur lassen sich durch den Compiler nicht direkt beeinflussen.

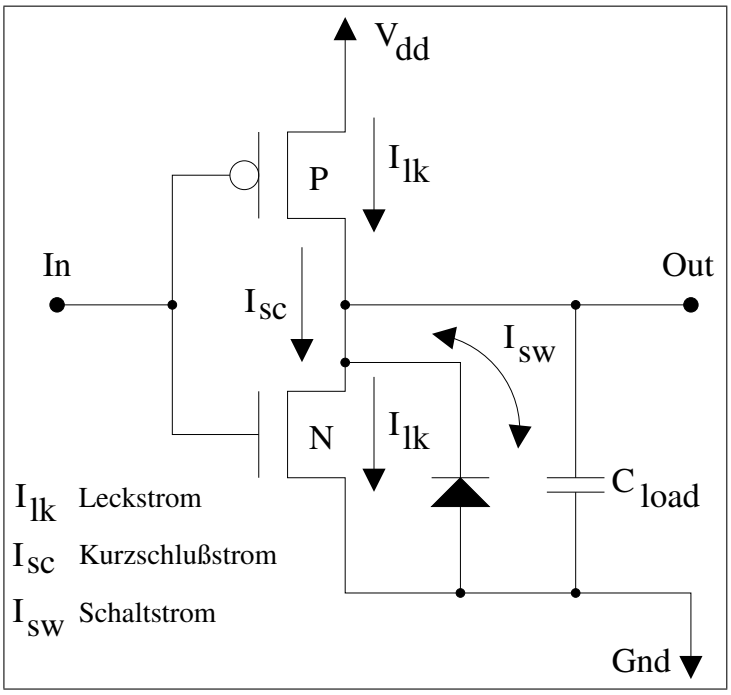


Abbildung 2.3: CMOS-Inverter

Aus diesem Grunde werden im folgenden die Ursachen, die für den Energieverbrauch dieser Komponenten, die typischerweise in CMOS-Technologie realisiert sind, betrachtet. Diese Ursachen sind im folgenden beschrieben und für ein besseres Verständnis in der Abbildung 2.3 dargestellt [Gru02]:

- **Schaltströme**
  Der Schaltstrom ist der Strom, der beim Schalten der Transistoren durch das Laden und Entladen der Lastkapazitäten (z.B. die Eingänge anderer Chips), die an den Ausgängen der Transistoren angeschlossen sind, fließt. Für den Energieverbrauch von CMOS-Schaltungen ist er maßgeblich verantwortlich, wobei eine Reduzierung des Energieverbrauchs zum einen durch eine Senkung der Versorgungsspannung erfolgen kann, da diese quadratisch in die Leistung und somit in den Energieverbrauch eingeht[Gru02]. Eine andere

Möglichkeit den Energieverbrauch zu senken besteht in einer Reduzierung der Schalthäufigkeit.

- **Kurzschluss-Ströme**
  Bei jedem Schaltvorgang der Transistoren fließt kurzzeitig ein Kurzschlussstrom, dessen Länge von der Schaltzeit der Transistoren abhängt, wobei die Energie, die durch solche Ströme verbraucht wird, einen Anteil von bis zu 30% am Gesamtenergieverbrauch haben kann[Gru02].

- **Leckströme**
  Leckströme sind unerwünschte, sehr kleine Ströme, die bei angelegter Spannung durch isolierendes Material hindurchfließen (z.B. durch einen sperrenden Transistor). Mit 1% haben Leckströme lediglich einen geringen Anteil am Energieverbrauch. Zukünftig wird dieses Problem jedoch vermehrt auftreten, da durch die zunehmende Verkleinerung der Prozessorstrukturen und der damit einhergehenden Senkung der Versorgungsspannung das Problem der Leckstöme prozentual ansteigen wird[Gru02].

Die Möglichkeiten eines Compilers, Einfluss auf den Energieverbrauch eines Systems zu nehmen, sind somit sehr begrenzt, da Compiler nicht in der Lage sind an den oben beschriebenen Ursachen des Energieverbrauchs unmittelbare Veränderungen herbeizuführen. Dies wäre die Aufgabe von Hardewareentwicklern, die zum Beispiel durch Verkleinerung der Strukturen oder der Senkung der Versorungsspannung Energieeinsparungen erreichen können. Diese Einsparungen werden jedoch meistens durch die steigende Anzahl der Transistoren, die auf einem Chip untergebracht werden, wieder aufgefressen. Ein Compiler hingegen ist in der Lage, Programme durch Optimierungsmethoden so zu verbessern, dass sie bedingt durch eine intelligentere Ressourcen-Nutzung (z.B. durch die Verringerung der Schalthäufigkeit) weniger Energie verbrauchen.

Die Leistung in elektronischen Systemen basiert auf der Spannung U und dem Strom I und wird nach Formel $p(t) = u(t) \cdot i(t)$ berechnet. Diese Größen gehen dann direkt in die Energiebilanz des Systems ein, da sich die Energie nach der Formel $E = \int p(t)dt$ berechnen lässt. Da ein Compiler jedoch keinen Einfluss auf die Versorgungsspannung des Prozessors hat[1], kann er zur Reduzierung des Energieverbrauchs nur die beiden anderen Größen beeinflussen [Sch00a]:

[1] Das Abschalten von Schaltungsteilen findet in dieser Betrachtung keine Berücksichtigung

- **Strom**
  Bei der Umsetzung des Zwischencodes in den zu erzeugenden Assemblercode sollte der Compiler Instruktionen auswählen, die eine möglichst geringe Leistungsaufnahme verursachen. Da die Versorgunsspannung normalerweise konstant ist, entspricht dies einer Auswahl von Instruktionen, die einen möglichst geringen durchschnittlichen Strom verursachen.

- **Zeit**
  Das typische Optimierungsziel der Programm-Performance ist weitgehend mit dem Optimierungsziel der Energieminimierung identisch, da ein wichtiger Bestandteil der Energie die benötigte Zeit ist. Ein Compiler sollte deswegen ein Programm generieren, das möglichst wenig Prozessor-Zeit für seine Ausführung benötigt.

Nachdem nun die Grundlagen, wie sich die Energieaufnahme von Systemen reduzieren lässt, besprochen worden sind, werden im folgenden einige allgemeine Möglichkeiten dargestellt, die den Energieverbrauch positiv beeinflussen können[Sch00a]:

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

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

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

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

2.4 Compiler

Der Entwurf von Hardware und den dazugehörigen Compilern ist untrennbar verbunden, da die Entwicklung komplexer und vom Codeumfang großer Programme nur durch den Einsatz von Hochsprachen wie C, C++ oder Java zu bewerkstelligen ist. Eine Programmierung in Assembler ist aus Zeit- und Kostengründen nahezu unmöglich. Aus diesem Grunde sind die Hersteller von Hardware tendenziell bemüht, passende Compiler zu ihrer Hardware zu entwickeln, die die in Hochsprachen geschriebenen Programme in entsprechende Assemblerprogramme übersetzen. Dies schließt die Entwicklung spezieller an die Hardware angepasster Optimierungsmethoden mit ein, da die Hardware nur im Zusammenspiel mit der Software ihren Zweck erfüllt. Leistungsvergleiche vergleichen demzufolge nicht nur die Hardwarekomponenten, sondern auch die Güte der Software. Das heißt, dass durch den Einsatz optimierender Compiler auch nachträglich durch Softwareupdates Leistungsreserven der Hardware freigesetzt werden können [Sch01c].

2.4.1 Aufbau eines Compilers

Ein Compiler hat die Aufgabe, eine Quellsprache in eine Zielsprache zu übersetzen. Im Kontext dieses Kapitels werden jedoch nur Compiler betrachtet, die in Hochsprachen geschriebene Programme in Assembler-Proramme übersetzen [AU99, Muc97]. Moderne Compiler benutzen normalerweise die in der Abbildung 2.4 dargestellte Struktur, die das Übersetzungsproblem in mehrere unabhängige Schritte aufteilt. Diese können dann durch entsprechende Algorithmen durchgeführt werden.

Des weiteren hat sich die Einteilung der Schritte in zwei Teile bewährt, wobei der erste Teil hauptsächlich aus Analyseschritten besteht, die von der Hochsprache abhängig und von der Zielarchitektur weitgehend unabhängig sind. Der zweite Teil hingegen ist nur von der Zielarchitektur abhängig. Als Schnittstelle zwischen diesen beiden Teilen wird ein Zwischencode verwendet, der durch seine einfache Struktur gut für Analysen und Optimierungen geeignet ist.

- **Das Front-End**
  Für den Compiler ist das in einer Hochsprache geschriebene Programm zu-

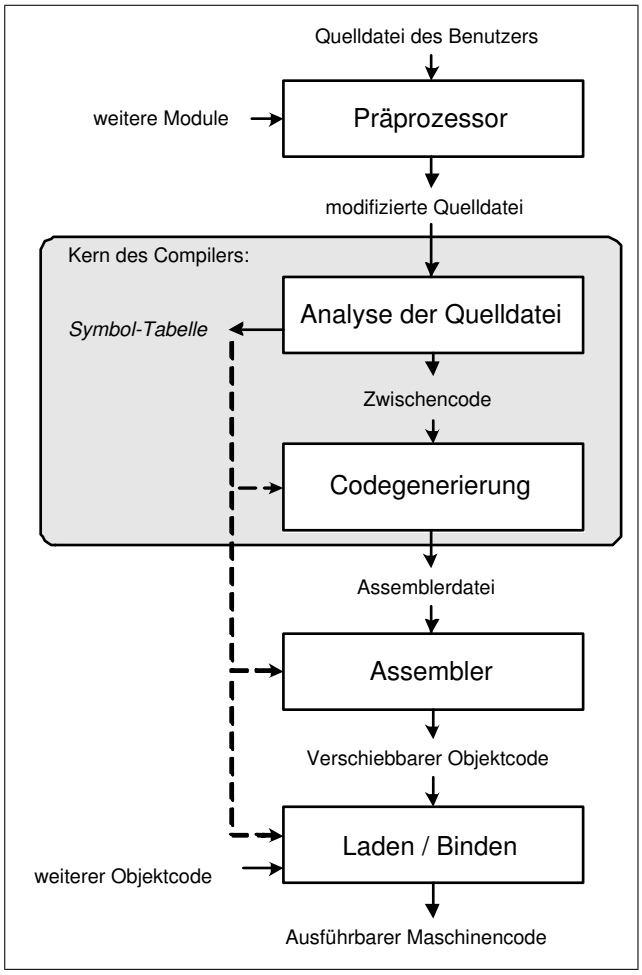


Abbildung 2.4: Ablauf eines typischen Compilevorgangs [Sch01c]

nächst nur ein Strom von Eingabezeichen, den er während der »lexikalischen Analyse« in einen Strom bekannter Symbole umwandelt, wobei Kommentare und Formatierungen verloren gehen. Diese Symbole werden dann durch die »syntaktische Analyse« zu grammatikalischen Sätzen, die durch die Grammatik der Hochsprache vorgegeben sind, zusammengefasst. Anschließend kann die »semantische Analyse« diese Sätze analysieren und den Wörtern der Sätze Bedeutungen zuordnen. Dies ermöglicht die Unterscheidung von Anweisungen, Ausdrücken und Operatoren.

- **Das Back-End**
  Das Back-End ist für die eigentliche Generierung des Assemblercodes zuständig. Aus diesem Grunde benötigt es sehr detaillierte Informationen über

die Möglichkeiten der Zielhardware. Als erstes hat es die Aufgabe, die Befehle des Zwischencodes in die Befehle der Zielhardware zu transformieren. Da es normalerweise jedoch keine eindeutige Abbildung zwischen diesen Befehlssätzen gibt, muss der Compiler eine gute Befehlsauswahl treffen, die möglichst schonend mit den Ressourcen der Zielhardware umgeht und deren Besonderheiten ausnutzt. Dies ist selbstverständlich nur möglich, wenn der Compiler entsprechend detaillierte Informationen über die Zielhardware besitzt. Nach der durchgeführten Befehlsauswahl muss der Compiler in der Regel noch die Registervergabe durchführen, die darüber entscheidet, welche Daten in den Prozessorregistern und welche im Speicher gehalten werden sollen. In diesem Zusammenhang ist vor allem die Vermeidung beziehungsweise die Minimierung des so genannten Spillings von Registern zu nennen, die Zwischenspeicherung von Registerinhalten in den Speicher. Diese Zwischenspeicherungen werden notwendig, wenn für einen Programmabschnitt nicht genügend Prozessorregister zur Verfügung stehen. Diese Entscheidungen sind von sehr großer Bedeutung, da Zugriffe auf den Speicher im Vergleich zu Registerzugriffen sehr kostspielig sind.

Diese Zweiteilung des Compilers in Front-End und Back-End liegt hauptsächlich darin begründet, dass es für die Compilerhersteller einfacher ist, neue Zielarchitekturen beziehungsweise neue Hochsprachen zu unterstützen, da sie lediglich den entsprechenden Teil des Compilers anpassen müssen. Das heißt, dass sich der Aufwand für die Unterstützung von n Front-Ends und m Back-Ends auf $n+m$ Module reduziert (Ansonsten $m \cdot n$). Des weiteren sollten Compiler während der Übersetzung Änderungen vornehmen, die die Semantik des Programms jedoch nicht verändern dürfen. Solche Änderungen dienen der Verbesserung der Codequalität, die üblicherweise durch einen der folgenden Punkte definiert wird:

- kleinerer, kompakterer Code
- geringere Ausführungszeit
- geringerer Energieverbrauch

Diese Optimierungsziele können durch unterschiedliche Optimierungsmethoden erreicht werden, wobei sich die unterschiedlichen Methoden in folgende Kategorien einteilen lasssen:

- **High-Level-Optimierungen**
  Die High-Level-Optimierungen sind maschinenunabhängig und werden auf der Quellsprache oder dem Zwischencode durchgeführt.

- **Low-Level-Optimierungen**
  Die Low-Level-Optimierungen sind maschinenabhängig und finden nach der eigentlichen Code-Generierung statt.

Im allgemeinen haben Optimierungsmethoden das Bestreben, die einzelnen Operationen so anzuordnen, dass keine unnötigen Operationen durchgeführt werden und dass alle Resourcen der Zielarchitektur möglichst gleichmäßig und vollständig ausgenutzt werden. Einer beliebigen Umordnung der Operationen stehen jedoch Abhängigkeiten zwischen den Operationen entgegen, die vom Compiler in einem Kontrollflussgraphen, kurz *CFG*, verwaltet werden (siehe Abbildung 2.5), wobei die Knoten in einem *CFG* sogenannte Basis-Blöcke sind.

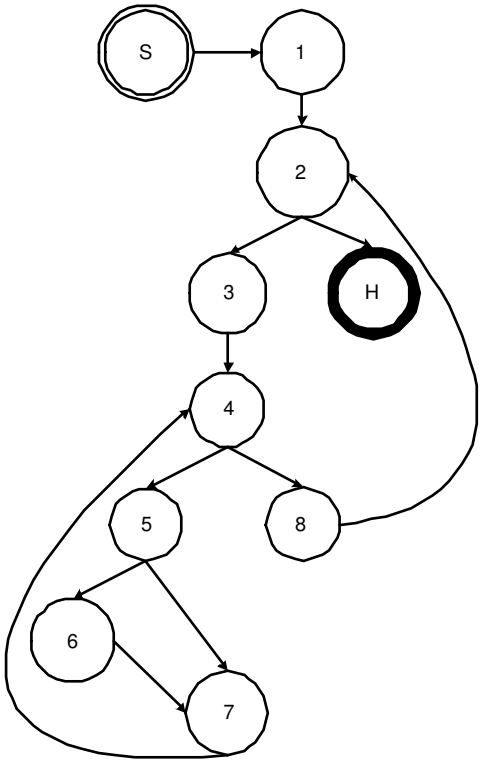
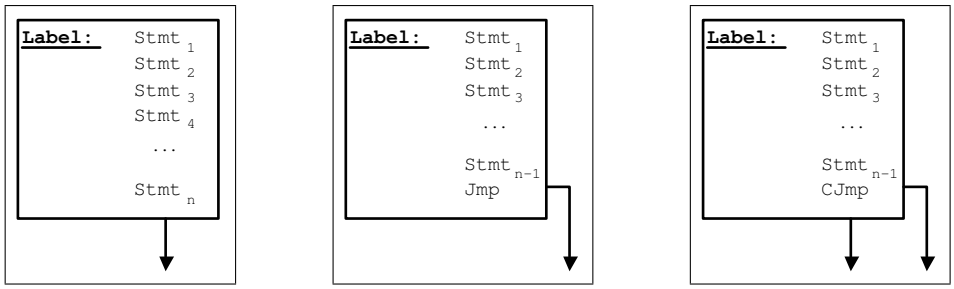


Abbildung 2.5: Kontrollflussgraph

Basis-Blöcke zeichnen sich dadurch aus, dass sie aus einer Menge von Instruktionen bestehen, die immer hintereinander ausgeführt werden. Sprünge zu einer Instruktion gehen immer zu der ersten Instruktion in einem Basis-Block. Des weiteren dürfen innerhalb eines Basis-Blocks keine Sprungbefehle auftreten. Diese sind nur als letzte Instruktion in einem Basis-Block zulässig. Das heißt, dass ein Basis-Block maximal 2 Nachfolger haben kann. Dieser Sachverhalt ist in Abbildung 2.6 anschaulich dargestellt.



(a) ohne Sprungbefehl (b) unbedingter Sprung (c) bedingter Sprung

Abbildung 2.6: Mögliche Varianten von Basis-Blöcken

2.5 Integer Linear Programming

»Integer Linear Programming«, kurz *ILP*, ist eine Technik zur Lösung mathematischer Probleme, die sehr häufig zur Lösung von Optimierungsproblemen im Bereich des Operation Research eingesetzt wird. Aus diesem Grunde gibt es unterschiedliche am Markt verfügbare Produkte, die in der Lage sind, *ILP*-Probleme zu lösen. Der Vorteil von *ILP* als Optimierungmethode liegt in der Flexibilität begründet, da sich sehr viele Probleme durch lineare Gleichungssysteme darstellen lassen. Es muss also lediglich das zu lösende Problem in ein lineares Gleichungssystem transformiert werden. Der Entwicklungsaufwand für einen spezialisierten Algorithmus entfällt, da verfügbare *ILP*-Pakete zur Lösung verwendet werden können. Aus diesem Grunde wird es auch für die während dieser Arbeit entwickelte Compiler-Optimierungsmethode eingesetzt.

Formal lässt sich ein *ILP*-Problem wie folgt beschreiben, wobei die Zielfunktion entweder *maximiert* oder *minimiert* werden kann. Des weiteren können die Variablen des Gleichungssystems in ihrem Wertebereich eingeschränkt werden, so dass die Variablen zum Beispiel als binäre Entscheidungsvariablen definiert werden können.

- Zielfunktion:

$$c_1 x_1 \quad + \quad c_2 x_2 \quad + \quad \ldots \quad + \quad c_n x_n \quad = \quad k$$

- Nebenbedingungen:

$$\begin{array}{rcrcrcrcl}
a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & \leq & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & \leq & b_2 \\
 & & & & \vdots & & & & \\
a_{m1}x_1 & + & a_{m2}x_2 & + & \ldots & + & a_{mn}x_n & \leq & b_m
\end{array}$$

3 Verwandte Arbeiten

Es gibt zu dem Problem, Energie in *ES* einzusparen, zahlreiche Forschungsarbeiten, die von dieser Arbeit weitergeführt und ergänzt werden. Einen ersten Überblick über das Themenfeld der Speicheroptimierung liefert [PN99], wobei aufgrund der Komplexität des Themenfeldes nicht alle Aspekte behandelt werden. In neuerer Zeit sind weitere Betrachtungsweisen und Aspekte hinzugekommen, die das Themenfeld der Speicheroptimierung mit dem Themenfeld der Energieoptimierung verbinden. Als Beispiel sei hier die Arbeit [AC03] genannt, die ein ähnliches Themengebiet wie das der vorliegenden Arbeit behandelt; der Ansatz der Betrachtungsweise ist jedoch ein anderer.

Im Kontext des hier diskutierten Themas sind allerdings die zwei Diplomarbeiten [Zob01, Gru02] zur Nutzung eines Scratchpad-Speichers von besonderer Bedeutung, da beide Arbeiten untersucht haben (die eine den statischen, die andere den dynamischen Fall) ob und wie viel Energie sich durch die Nutzung eines Scratchpad-Speichers einsparen lässt. Diese beiden Arbeiten sind später durch [Ste02] zusammengefasst und erweitert worden. Nicht zu vergessen ist, dass die hier vorliegende Arbeit in dem breiten Kontext der Lehrstuhlforschung[1] steht, so dass diesbezüglich zahlreiche Verknüpfungen und Vergleiche getroffen werden können. Ein Beispiel hierfür ist das ASPDAC-Konferenzpaper „Fast, Predictable, and Low-energy Memory References through Architecture-aware Compilation“ [PM04], das unter anderem einen positiven Effekt der Onchip-Speichernutzung auf die Vorhersagbarkeit der *Worst-Case-Execution-Time* aufzeigt. Diese Vorhersagbarkeit ist vorallem im Bereich der *ES*, die unter Echtzeitbedingungen arbeiten, von Bedeutung [Mar03].

[1] `http://ls12-www.cs.uni-dortmund.de`

3.1 Die compilergesteuerte Nutzung eines Scratchpad-Speichers

Zobiegala hat sich in seiner Diplomarbeit [Zob01] mit dem Problem beschäftigt, den Energiebedarf von Programmen durch die automatische Nutzung eines Scratchpad-Speichers zu reduzieren. Um dieses Ziel zu erreichen, hat er eine Compiler-Optimierungsmethode entwickelt, die in der Lage ist, ausgewählte Basis-Blöcke statisch in den Scratchpad-Speicher zu verschieben, wobei mit der statischen Verschiebung gemeint ist, dass die ausgewählten Basis-Blöcke beim Programmstart an die entsprechende Position im Scratchpad-Speicher verschoben werden und dort an unveränderter Stelle bis zum Programmende verbleiben. Dieser Sachverhalt ist in der Abbildung 3.1 noch einmal eingänglich dargestellt.

Der Vorteil dieser Optimierungsmethode ist, dass sie vollständig automatisch abläuft und somit alle Anwendungen ohne nennenswerten zusätzlichen Entwicklungsaufwand von ihr profitieren können. Um dies zu erreichen bewertet der Algorithmus der Optimierungsmethode den Energiebedarf der einzelnen Basis-Blöcke und berechnet aus dem Energieunterschied (Off-Chip versus On-Chip) einen *Vorteil*, der angibt, wie hoch die Energieeinsparung wäre, wenn der bewertete Basis-Block in den Scratchpad-Speicher verschoben würde. Anschließend können die globalen Variablen wie die Basis-Blöcke bewertet werden, da für die Energiebewertung nur die Energie der Speicherzugriffe berücksichtigt wird. Nach der Bewertung ist das Optimierungsproblem mit dem bekannten Rucksackproblem (*Knapsack-Problem*) identisch. Zobiegala verwendet zur Lösung dieses Problems sowohl einen Branch-and-Bound Algorithmus sowie einen *ILP*-Solver, wobei er für den zweiten Ansatz das Problem in ein lineares Gleichnungssystem transformiert. Die Lösung des *ILP*-Problems wird dann durch den *ILP*-Solver *CPLEX* der Firma ILOG [ILO] durchgeführt. Die eigentliche Verteilung der Basis-Blöcke und der globalen Variablen übernimmt dann der »Programm-Loader« mitteles Scatter-Loading [ARM], da der Compiler zwei Assemblerdateien generiert, die der Linker anschließend zu dem endgültigen Programm zusammenfasst. Dabei versieht der Linker die Programmteile, die in den unterschiedlichen Assemblerdateien liegen, mit den entsprechenden Basis-Adressen der gewünschten Speicherbereiche (Off-Chip oder On-Chip).

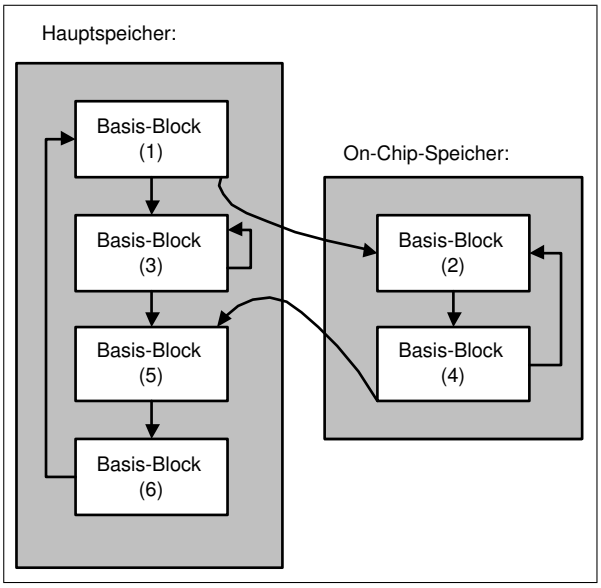


Abbildung 3.1: Die Einlagerung ausgewählter Basis-Blöcke in einen Scratchpad-Speicher (Abbildung aus [Lee01])

Als Ergebnis der Arbeit ergab sich eine erhebliche Energieeinsparung von bis zu 80% bei gleichzeitiger Performancesteigerung gegenüber einem System ohne Scratchpad-Speicher. Dieser hohe Wert ergibt sich aus der Tatsache, dass die untersuchten Programme größtenteils vollständig in den Scratchpad-Speicher verlagert werden konnten. Reale Programme hingegen sind normalerweise wesentlich größer und werden somit in der Praxis diese hohe Enerergieeinsparung nicht erreichen. Dennoch wurde gezeigt, dass durch eine sinnvolle Verwendung eines Scratchpad-Speichers sehr viel Energie gespart werden kann.

Dieses Konzept wurde später durch die Diplomarbeit [Gru02] aufgegriffen und erweitert. Die neue Optimierungsmethode ist in der Lage, die Belegung des Scratchpad-Speichers dynamisch während der Laufzeit zu verändern und kann somit das Problem, dass nur sehr kleine Programmteile in den Scratchpad-Speicher passen, gut kompensieren. Als Ergbniss ergaben sich ähnlich gute Werte wie bei der statischen Optimierungsmethode. Bei wachsender Programmgröße wirken sich die Nachteile der kleinen Scratchpad-Speichergröße bei diesem Ansatz jedoch nicht so gravierend aus. Aus diesem Grunde ist die dynamische Methode in der Praxis deutlich besser verwendbar.

4 Gegenstand der Arbeit

Damit ein Compiler in der Lage ist, ein Programm in ein Assembler-Programm zu übersetzen, muss er genaue Kenntnisse über die Hardwareeigenschaften des Systems besitzen. Da diese Arbeit jedoch eine konsequente Fortsetzung der am Lehrstuhl vorangegangenen Arbeiten ist (siehe Kapitel 3) und diese bereits den speziell entwickelten C-Compiler *enCC* eingesetzt und erweitert haben, kann sich diese Arbeit auf das Kernproblem, eine neue Optimierungmethode zu entwickeln, konzentrieren.

Für die Entwicklung von Compiler-Optimierungsmethoden steht am Lehrstuhl eine Arbeitsumgebung zur Verfügung, die es ermöglicht, diese durch die Verwendung einer Simulationsumgebung zu bewerten. Aus diesem Grunde konnte auf aufwendige Versuchsanordnungen, die zum Beispiel für die Ermittlung des Energieverbrauchs der Hardware benötigt würden, verzichtet werden. Für die Interpretation der Ergebnisse ist ein Verständnis der Originalhardware jedoch unabdingbar.

Bevor nun die Konzepte der neu entwickelten Compiler-Optimierungsmethode vorgestellt werden, sei aus den oben genannten Gründen ein Überblick über die Originalhardware und die zur Verfügung stehende Arbeitsumgebung gegeben.

4.1 Der Prozessor Atmel AT91M40400

Der Prozessor Atmel AT91M40400 [Atm] basiert auf der ARM7TDMI-Architektur [ARM], die wegen ihrer Eigenschaften, die der folgenden Aufzählung entnommen werden können, sehr häufig im Bereich der *ES* eingesetzt wird.

- 32-Bit ALU
- 32-Bit RISC Befehlssatz für maximale Performance und Flexibilität

- 16-Bit Thumb Befehlssatz für eine gesteigerte Codedichte
- Dreistufige Pipeline
- Niedriger Energieverbrauch

Das besondere an dieser Architektur ist, dass sie zwei unterschiedliche Befehlssätze beherrscht. Der normale 32-Bit Befehlssatz ermöglicht es, den Prozessor als vollwertigen 32-Bit RISC-Prozessor zu verwenden, der eine hohe Performance erreicht. Der zweite sogenannte 16-Bit Thumb Befehlssatz entsteht durch Abbildung eines 32-Bit Befehlswortes auf ein 16-Bit-Befehlswort, wobei aus Platzgründen bewusst auf einige Möglichkeiten des 32-Befehlssatzes verzichtet wurde [ARM]. So stehen statt der ursprünglichen 16 Register nur noch 8 direkt adressierbare Register zur Verfügung und die Möglichkeit der bedingten Ausführung von Instruktionen entfällt. Während der Ausführung eines Programms wandelt der ARM-Prozessor diese 16-Bit Befehle dynamisch in 32-Bit Befehle um. Der Vorteil dieses zweiten Befehlssatzes liegt in der hohen Codedichte begründet, da weniger Programmspeicher als bei dem 32-Bit Befehlssatz benötigt wird [Mar03]. Aus diesem Grunde ergibt sich auch eine Verminderung des Energieverbrauchs, da die Reduzierung der Wortbreite eine verminderte Schaltaktivität auf den Bussen zur Folge hat [Ste02]. Diese Vorteile können die Nachteile, die durch die Reduzierung der Wortbreite und den damit verbundenen Befehlssatzeinschränkungen verbunden sind, gut kompensieren [ARM].

Eine weitere Besonderheit des Atmel AT91M40400 im Vergleich zu anderen ARM7-TDMI-Implementierungen ist der 4 Kilobyte große Scratchpad-Speicher. Bei diesem handelt es sich um einen in den Prozessor integrierten Speicher, der von Programmen wie ein normaler Hauptspeicher benutzt werden kann. Da er jedoch in den Prozessor integriert ist und wesentlich kleiner als der normale Hauptspeicher ist, sind Zugriffe bezüglich der benötigten Energie wesentlich günstiger (vgl. Kapitel 4.3.3), da zum einen kleine Speicher weniger Energie pro Zugriff benötigen und zum anderen keine langen Busleitungen benötigt werden, die ebenfalls für einen hohen Energieverbrauch verantwortlich sind.

4.2 Die vorhandene Arbeitsumgebung

Dieses Kapitel beschreibt die am Lehrstuhl zur Verfügung stehende Simulationsumgebung, die die einfache Weiterenwicklung des Compilers *enCC* ermöglicht. Das aufwendige Arbeiten mit dem Evaluationsboard sowie die entsprechenden Energiemessungen an diesem können entfallen, da der Simulator in Kombination mit dem »*enProfiler*«, der aus dem »*Traceanalyzer*« hervorgegangen ist [Sch00a], eine weitere Beschäftigung mit der Hardware überflüssig macht.

Die eigentliche Simulation der vom Compiler erstellten Programme geschieht mit einer Familie von Programmen, die unter dem Namen ARMulator[1] zusammengefasst werden. Diese Programmfamilie gehört wie der Assembler und der Linker zum »Software Development Toolkit«, das von [ARM] zur Verfügung gestellt wird.

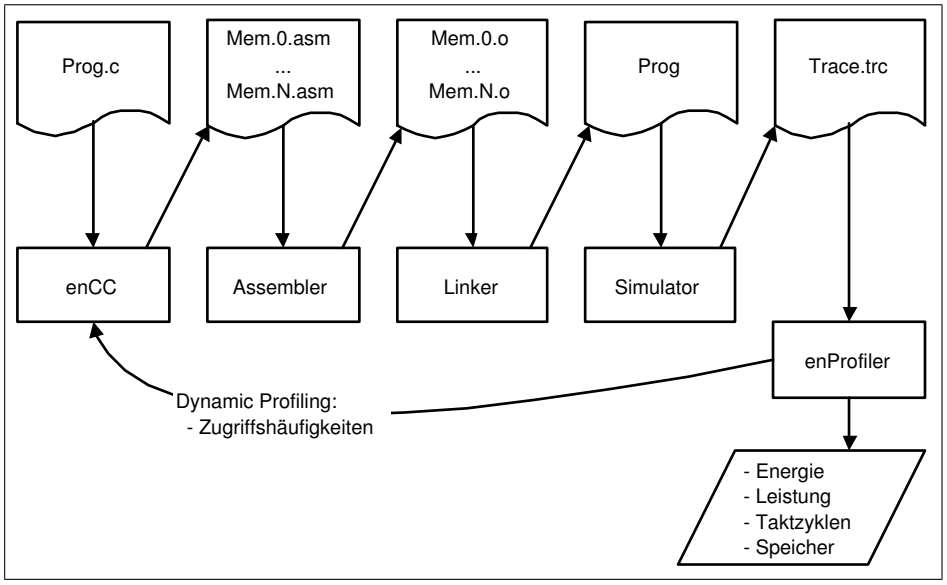


Abbildung 4.1: Simulationsumgebung

Wie aus der Abbildung 4.1 hervorgeht, muss für die Simulation zuerst das C-Programm vom Compiler in ein oder mehrere ASM-Dateien übersetzt werden, wobei die Anzahl der zu erzeugenden ASM-Dateien von der Anzahl der konfigurierten Speicherpartitionen abhängt. Diese ASM-Dateien werden dann vom Assembler zu Objektdateien übersetzt, die anschließend vom Linker zu einem ausführbaren

[1] ARM Instruction-Level Simulator

Programm zusammengefasst werden. Das so erstellte Programm wird dann mit dem *ARMulator* simuliert, wobei der Programmverlauf als Trace-Datei festgehalten wird. Durch die Analyse dieser Tracedatei können viele sehr unterschiedliche Informationen gewonnen werden.

Für die in dieser Arbeit vorgestellte Optimierungsmethode ist diese Tracedatei in zweierlei Hinsicht von Bedeutung. Zum einen ermöglicht sie das Profiling (siehe Kapitel 4.3.2) der Zugriffshäufigkeiten auf die Programmobjekte, die für die Optimierungsmethode wie in Kapitel 4.3 beschrieben, von großer Bedeutung sind. Zum anderen ermöglicht die Tracedatei dem *enProfiler* die Ausgabe wichtiger Informationen, die für die Bewertung der Optimierungmethoden benötigt werden. Unter anderem sind dies der Energieverbrauch, die Laufzeit (Anzahl der Taktzyklen) und die benötigte Speichergröße (Daten und Programm).

Vor der Simulationsdurchführung müssen jedoch einige Konfigurationseinstellungen vorgenommen werden, die das Verhalten der einzelnen Programme bestimmen. Dabei sind die Abhängigkeiten zwischen den Programmen zu beachten, da nur das Zusammenspiel aller Programme zu einem sinnvollen Ergebnis führt (siehe Abbildung 4.1).

Bei dem Compiler können zum Beispiel die zu verwendenden Optimierungsmethoden konfiguriert werden. Einige der Optimierungsmethoden benötigen jedoch weitere Konfigurationsdateien (siehe Kapitel 6.1), die inhaltlich mit den Konfigurationsdateien des Linkers und des Simulators übereinstimmen müssen. Bei der in dieser Arbeit vorgestellten Optimierungsmethode ist dies die Konfiguration der Speicherpartitionierung, die allen Programmen in der Tool-Chain bekannt sein müssen.

4.3 Das Optimierungsproblem

Ziel dieser Arbeit ist die Untersuchung des Energiesparpotentials, das durch die Nutzung partitionierter Speicher gewonnen werden kann. Dieses Energiesparpotential entsteht, da kleinere Speicher bei einem Zugriff weniger Energie benötigen als größere (vgl. Tabelle 4.1). Aus dieser Tatsache entstand die Idee, die vorhandenen Speicher in mehrere kleine Speicher zu zerteilen und den Compiler mit einer

Optimierungsmethode, die ein Programm auf die kleineren Speicher verteilt, auszustatten. Aus diesem Grunde wurde im Rahmen dieser Arbeit eine Compiler-Optimierungsmethode entwickelt, die die in [Zob01] dargestellte Methode erweitert. Dies ist anschaulich in Abbildung 4.2 dargestellt.

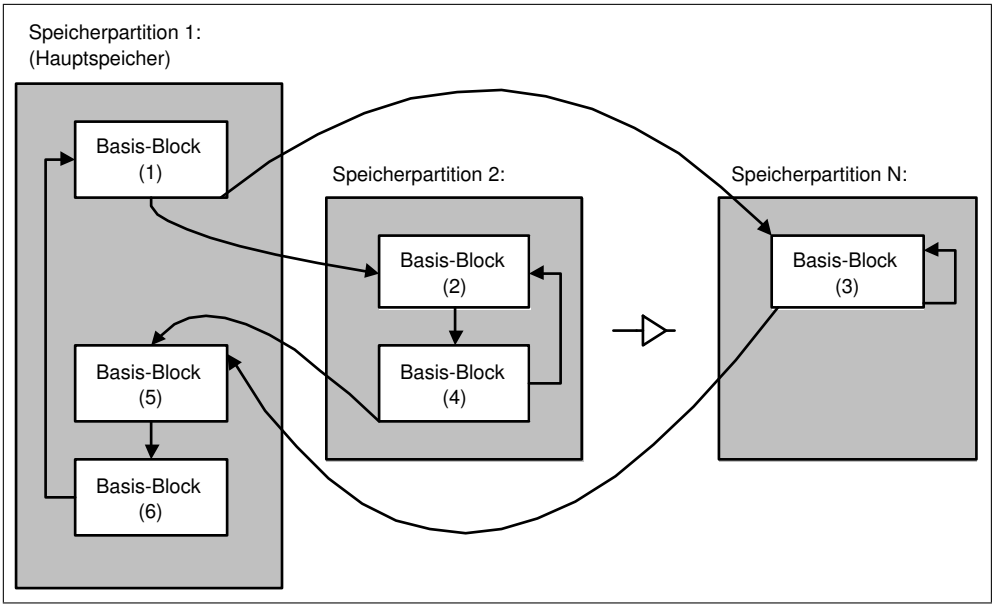


Abbildung 4.2: Das Verschieben ausgewählter Basis-Blöcke in unterschiedliche Speicherpartitionen

Für die Verteilung eines Programms auf mehrere Speicher muss ein Programm zuerst in seine Bestandteile zerlegt werden. Diese Teile können dann von der Compiler-Optimierungsmethode neu zusammengestellt und in der entsprechenden Speicherpartition abgelegt werden. Das zu lösende Problem der Optimierungsmethode ist somit mit dem bekannten Rucksackproblem (*Knapsack-Problem*) verwandt.

Bei dem Rucksackproblem [Weg99] hat jeder Gegenstand, der im Rucksack untergebracht werden soll, einen bestimmten Nutzen und ein bestimmtes Gewicht bzw. Volumen. Da ein Rucksack jedoch nur eine begrenzte Gewichts- bzw. Volumenkapazität hat, muss für die Lösung des Problems ein Inhalt gefunden werden, der die Kapazitätsgrenzen des Rucksacks nicht überschreitet und gleichzeitig einen maximalen Nutzen erreicht. Für unser Optimierungsproblem bedeutet dies, das mehrere Rucksäcke, die den Speicherpartitionen entsprechen, zur Verfügung stehen und die Größe der Speicherpartitionen der jeweiligen Kapazitätsgrenze entspricht.

Die Nutzenfunktion muss nun so gewählt werden, das sie dem zu erreichenden Optimierungsziel entspricht. Die möglichen Gegenstände entsprechen den »Programmobjekten« des Compilers, die in der folgenden Aufzählung aufgeführt sind:

- **Globale Variablen:**
  Globale Variablen werden an einer bestimmten Position im Speicher abgelegt (Basis-Adresse) und haben bedingt durch Ihren Typ eine feste Größe. Aus diesem Grunde kann die Verschiebung in eine andere Speicherpartition durch die einfache Veränderung der Basis-Adresse erfolgen.

- **Lokale Variablen (Stack):**
  Lokale Variablen haben im Gegensatz zu den globalen Variablen keine Basis-Adresse, da sie auf dem Stack abgelegt werden. Das heißt, dass sie je nach Programmverlauf an unterschiedlichen Adressen abgelegt werden. Eine Verschiebung einzelner lokaler Variablen ist somit nicht ohne weiteres möglich. Es besteht jedoch die Möglichkeit, den gesamten Stack in eine entsprechend große Speicherpartition zu verschieben. Da der Compiler *enCC* jedoch keine zuverlässige Methode anbietet, die die maximale Stackgröße und die Anzahl der Zugriffe auf den Stack bestimmt, werden lokale Variablen bei der Entwicklung der Compiler-Optimierungsmethode nicht berücksichtigt.

- **Funktionen:**
  Funktionen können genau wie die globalen Variablen durch Veränderung ihrer Basis-Adresse in andere Speicherpartitionen verschoben werden.

- **Basis-Blöcke:**
  Das Verschieben von Basis-Blöcken ist in der Abbildung 4.2 dargestellt. Es ist jedoch zu beachten, dass der Kontrollfluss durch das Einfügen von Sprungbefehlen, die die Programmausführung in der gewünschten Speicherpartition fortsetzen, anzupassen ist [Zob01]. Diese zusätzlichen Sprungbefehle müssen bei der Aufstellung des Modells berücksichtigt werden, da sie zum einen die Größe der Basis-Blöcke verändern. Zum anderen muss die Nutzenfunktion ebenfalls angepasst werden, da sich der Nutzen durch das Einfügen der benötigten Sprungbefehle verändert.

Für die Lösung des Problems muss bei den oben aufgeführten Programmobjekten jedoch beachtet werden, dass Funktionen und Basis-Blöcke keine unabhängigen

Objekte sind (vgl. Kapitel 2.4.1). Diese Abhängigkeiten müssen bei der Lösung des Optimierungsproblems berücksichtigt werden. Zur Lösung des Problems wird deshalb ein *ILP* -Solver eingesetzt. Dies hat den Vorteil, dass die Abhängigkeiten durch einfache Nebenbedingungen im *ILP*-Modell modelliert werden können, welches detailliert im Kapitel 5 beschrieben wird.

Mit dem bisher beschriebenen Konzept lassen sich nun unterschiedliche Optimierungsmethoden realisieren, da das Optimierungsziel lediglich durch die gewählte Nutzenfunktion beschrieben wird. Im Mittelpunkt dieser Arbeit steht die Untersuchung des Energiesparpotentials. Denkbar wären jedoch auch andere Optimierungsziele, da sich zum Beispiel Geschwindigkeitsverbesserungen durch die Ausnutzung partitionierter Speicher ergeben könnten.

Für die Aufstellung einer Nutzenfunktion, die den Nutzen für die oben gennanten Programmobjekte berechnet, ist die Ermittlung der Zugriffs- und Ausführungshäufigkeiten (siehe Kapitel 4.3.2) von zentraler Bedeutung, da der Nutzen im allgemeinen von diesen abhängig ist. Des weiteren wird für eine Energieoptimierung eine Nutzenfunktion benötigt, die in der Lage ist, jene Energieanteile, welche auf die Programmobjekte entfallen, vorherzusagen. Eine detailliertere Auseinandersetzung hiermit folgt in Kapitel 4.3.3.

Nachdem nun alle benötigten Grundlagen dargestellt worden sind, kann nun die Vorgehensweise der Optimierungsmethode vorgestellt werden. Es handelt sich um eine Low-Level-Optimierungsmethode (vgl. Kapitel 2.4.1), da der Nutzen und die Größe der Programmobjekte erst nach der Transformation des Programms in die Ziel-Assemblersprache berechnet werden kann.

Die einzelnen Schritte der Compiler-Optimierungsmethode sind:

1. Zusammenstellen der für die Verschiebung in Frage kommenden Programmobjekte
2. Berechnung des Nutzens für jedes Programmobjekt in Abhängigkeit der zur Verfügung stehenden Speicherpartitionen
3. Transformation des Problems in ein *ILP*-Problem
4. Lösung des *ILP*-Problems mit dem *ILP*-Solver CPlex der Firma ILOG [ILO]

5. Einlesen der *ILP*-Lösung
6. Setzen der neuen Basis-Adressen
7. Transformation des Kontrollflussgraphen (Sprunganpassung)

4.3.1 Das Teilen von Basis-Blöcken

Für das Erreichen des Optimierungsziels kann es sinnvoll sein große Basis-Blöcke in mehrere kleine Basis-Blöcke zu zerteilen. Dies vergrößert auf der einen Seite zwar die Anzahl der zu verteilenden Programmobjekte, auf der anderen Seite gibt diese Zerlegung dem *ILP*-Solver jedoch die Möglichkeit, geeignete Basis-Blöcke in die bestmöglichste Speicherpartition zu verschieben. Dies wäre ohne eine Teilung unter Umständen nicht möglich, da zu große Basis-Blöcke nicht in kleinen Speicherpartitionen abgelegt werden können. Das Konzept, wie entsprechende Basis-Blöcke geteilt werden können, ist in der Abbildung 4.3 dargestellt. Es basiert darauf, dass es lediglich 3 Typen von Basis-Blöcken gibt (vgl. Abbildung 2.6). Jeder dieser Basis-Block-Typen kann jedoch in zwei Basis-Blöcke zerlegt werden, wobei der erste der beiden neuen Basis-Blöcke ein Basis-Block ohne Sprungbefehl ist. Dieser wird dann durch eine sequentielle Kante im Kontrollflußgraphen mit dem zweiten neuen Basis-Block, der vom ursprünglichen Typ ist, verbunden. Diese Vorgehensweise kann theoretisch solange rekursiv wiederholt werden, bis jeder Basis-Block nur noch aus genau einem Befehl besteht.

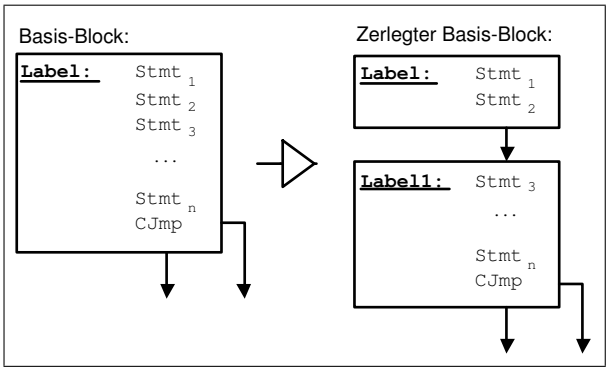


Abbildung 4.3: Die Teilung eines Basis-Blocks

Eine konsequente Umsetzung dieser Idee würde es wahrscheinlich sogar ermöglichen, einen effizienten Algorithmus für das Optimierungsproblem zu finden. In

dieser Arbeit steht jedoch die Untersuchung des Energiesparpotentials partitionierter Speicher im Vordergrund. Deshalb wurde das Optimierungsproblem mit einem *ILP*-Solver umgesetzt, da bei diesem Ansatz der Entwicklungsaufwand für einen spezialisierten Algorithmus entfällt (vgl. Kapitel 2.5). Die längere Laufzeit des *ILP*-Solvers wird bewusst in Kauf genommen.

4.3.2 Ermittlung der Zugriffs- und Ausführungshäufigkeiten

Der Compiler *enCC* bietet die Möglichkeit, die Ausführungshäufigkeit von Funktionen und Basis-Blöcken sowie die Zugriffshäufigkeit auf Variablen zu bestimmen. Diese Analysen sind für einige Optimierungsmethoden unabdingbar. Aus diesem Grunde wurden in der Vergangenheit zwei unterschiedliche Methoden zur Vorhersage dieser Häufigkeiten in den Compiler integriert, wobei beide Methoden ihre Vor- und Nachteile haben.

- **Statische Analyse**
  Diese Methode basiert auf einer »Kontrollflussanalyse«, wobei die Ausführungshäufigkeiten unabhängig von den Daten bestimmt werden, die das Programm verarbeitet. Der Compiler *enCC* nimmt bei diesem Verfahren an, dass jede Schleife zehnmal durchlaufen wird. Des weiteren nimmt er für jede Programmverzweigung eine Wahrscheinlichkeit von 50% an. Dabei stößt dieses Verfahren sehr schnell an seine Grenzen, wenn das Programm datenabhängige rekursive Funktionsaufrufe oder datenabhängige Schleifenabbruchbedingungen verwendet. In diesen Fällen können die vorhergesagten Häufigkeiten sehr stark von den realen Häufigkeiten abweichen.

- **Dynamisches Profiling**
  Diese Methode basiert auf einem Profiling des in einem ersten Compilerlauf generierten Programms. Das heißt, dass das Programm mit dem *ARM-Simulator* ausgeführt wird und der *enProfiler* anschließend die Zugriffe auf die Programmobjekte, die im Simulationsprotokoll festgehalten sind, zählt. Bei einem zweiten Compilerlauf kann der Compiler diese Daten einlesen und für seine Optimierungen verwenden. Die Nachteile dieses Verfahrens sind, dass zum einen zwei Compilerläufe benötigt werden. Zum anderen kann auch dieses Verfahren das Problem der Datenabhängigkeit nicht lösen, da

bei datenabhängigen Programmen die ausgeführten Programmteile und deren Ausführunghäufigkeiten sehr unterschiedlich sein können. Andererseits ermittelt es für gegebene Eingabedatenmengen die tatsächlichen Ausführunghäufigkeiten mit hoher Genauigkeit.

4.3.3 Energieberechnung

Wie im Kapitel 2.3 dargestellt, ist der mögliche Einfluss einer Compiler-Optimierungsmethode zur Reduktion des Energieverbrauchs hauptsächlich auf den Prozessor und das Speichersystem begrenzt. Aus diesem Grunde verwendet die im Zuge dieser Arbeit entwickelte Compiler-Optimierungsmethode ein Energiemodell auf Instruktionsebene, dass den Energiebedarf anhand der ausgeführten Instruktionen berechnet. Da es jedoch zu aufwendig wäre, während einer Optimierung immer den gesamten Energiebedarf des Programms zu berechnen, ist es sinnvoll, das in [Ste02] vorgestellte Modell insofern anzupassen, dass es nur die Energieanteile, die optimiert werden sollen, berücksichtigt.

Das in [Ste02] vorgestellte Modell modelliert die Energie, die pro ausgeführter Instruktion im Prozessor benötigt wird, wie folgt:

$$E_{cpu} = \sum E_{BasicCosts} + \sum E_{InterInstrCosts} \tag{4.1}$$

Dabei beschreibt der Energieanteil $E_{BasicCosts}$ die Energiekosten, die durch die Ausführung eines Befehls entstehen. Bedingt durch den Aufbau von Prozessoren entstehen jedoch zusätzliche Energiekosten, wenn zwei unterschiedliche Befehle aufeinanderfolgend ausgeführt werden. Dieser Anteil wird durch den Anteil $E_{InterInstrCosts}$ modelliert. Des weiteren benötigt das Speichersystem ebenfalls Energie. Neben einem Fixanteil, den der Speicher zum Beispiel für die periodischen Refreshs benötigt, wird für jeden Speicherzugriff zusätzliche Energie benötigt. Diese Zugriffe werden zum einen durch das Holen der Programminstruktionen aus dem Speicher verursacht (E_{ifetch}). Zum anderen werden sie durch das Programm, das auf benötigte Daten zugreift (E_{dfetch}), hervorgerufen. Die Höhe der Energie hängt dabei von der Speichergröße ab und kann der Tabelle 4.1 entnommen werden.

Aus diesen beiden Kostenarten lässt sich nun eine Formel, die den Energiebedarf

pro Instruktion beschreibt, ableiten.

$$E_{gesamt} = E_{cpu} + E_{ifetch} + E_{dfetch} \tag{4.2}$$

Speichergröße [Byte]	**Energy** $[nJ]$
64	0,49493
128	0,56830
256	0,60362
512	0,68883
1024	0,84299
2048	1,04730
4096	1,44001
8192	2,14199
16384	4,04672
32768	6,52916
65536	11,87150

Tabelle 4.1: Der Energieverbrauch von Speicherzugriffen

Da die im Zuge dieser Arbeit entwickelte Compiler-Optimierungsmethode lediglich das Speicherlayout des Programms verändert, also das eigentliche Programm bis auf die Positionierung im Speicher nicht verändert, ist der Energieanteil E_{cpu} konstant und spielt somit für die Optimierung keine Rolle. Daher wird im Modell lediglich E_{ifetch} und E_{dfetch} berücksichtigt. Die Energie für die Programmobjekte, die für die Verschiebung in Frage kommen, kann demnach wie folgt berechnet werden:

- **Globale Variablen**
  Die Energie, die die Compiler-Optimierungsmethode einer globalen Variablen zuordnen soll, entspricht der Energie, die durch die Zugriffe auf die Speicherzellen im Speichersystem entstehen. Aus diesem Grunde kann die für die Optimierung zu berücksichtigende Energie wie folgt berechnet werden:

$$E_g = E_{dfetch} \cdot (\text{Anz. Zugriffe}) \tag{4.3}$$

- **Basis-Blöcke**
  Bei den Basis-Blöcken wird der Energiebedarf durch die Ausführung der Instruktionen, aus denen die Basis-Blöcke bestehen, verursacht. Der Energieanteil, der bei der Ausführung im Speichersystem entsteht, kann durch folgende Formel berechnet werden:

$$E_{bb} = E_{ifetch} \cdot (\text{Anz. Ausführungen}) \cdot (\text{Anz. Instruktionen} + 2) \tag{4.4}$$

  Dabei ist jedoch zu beachten, dass bedingt durch den 16-Bit Thumb Befehlssatz des ARM-Prozessors, die Sprungdistanz normaler Sprungbefehle nicht ausreicht, um zu einem Basis-Block in einer anderen Speicherpartition zu springen. Dieses Problem kann durch das Einfügen von Sprungbefehlen, die die Distanz überbrücken können, gelöst werden. Für die Energieberechnung bedeutet dies, dass die Anzahl der Instruktionen um zwei erhöht werden muss, da für jeden nachfolgenden Basis-Block ein zusätzlicher Sprungbefehl eingefügt werden muss (vgl. Kapitel 2.4.1).

- **Funktionen**
  Bei den Funktionen wird der Energiebedarf durch die Ausführung der Instruktionen, aus denen die Funktion besteht, verursacht. Der Energieanteil, der bei der Ausführung einer Funktion im Speichersystem entsteht kann durch folgende Formel berechnet werden:

$$E_f = E_{ifetch} \cdot (\text{Anz. Ausführungen}) \cdot (\text{Anz. Instruktionen}) \tag{4.5}$$

  Diese ist weitgehend mit der Formel, die den Energieanteil für die Basis-Blöcke berechnet, identisch. Im Gegensatz zu den Basis-Blöcken müssen bei den Funktionen jedoch keine zusätzlichen Sprungbefehle eingefügt werden, da der Befehl zum ausführen einer Funktion auch im 16-Bit Thumb Befehlssatz die Sprungdistanz zu den anderen Speicherpartitionen überbrücken kann. Das Verschieben ganzer Funktionen hat somit einen entscheidenden Vorteil gegenüber dem Verschieben einzelner Basis-Blöcke, da keine zusätzlichen Sprungbefehle eingefügt werden müssen.

5 Formalisierung des Problems

In diesem Kapitel werden zwei alternative Modelle, die der im Zuge dieser Arbeit entwickelten Compiler-Optimierungsmethode zugrunde liegen, vorgestellt. Zuvor werden jedoch einige Vorüberlegungen angestellt sowie allgemeine Systemeigenschaften modelliert.

5.1 Vorüberlegungen

5.1.1 Speicherpartitionen

Aus der Sicht des Prozessors ist eine Speicherpartition lediglich ein Speicherbereich, der an einer bestimmten Basisadresse beginnt und der eine bestimmte Größe hat. Es können beliebig viele Speicherpartitionen existieren. Aus diesem Grunde werden die einzelnen Speicherpartitionen in der Menge MP zusammengefasst.

$$\text{Menge } MP := \{mp_1, \ldots, mp_m\} \tag{5.1}$$

Die Eigenschaften der Speicherpartitionen können dann durch Funktionen abgefragt werden. Für die im Zuge dieser Arbeit entwickelte Compiler-Optimierungsmethode ist jedoch nur die Funktion $Size(mp)$ von Bedeutung.

$$Size(mp) := \text{ Größe der Speicherpartition } mp \tag{5.2}$$

Die weiteren Eigenschaften, wie die Basisadresse, die später für die Codegenerierung benötigt wird, haben keine unmittelbare Bedeutung für das *ILP*-Modell und werden aus diesem Grunde hier nicht aufgeführt.

5.1.2 Programmobjekte

Wie im Kapitel 4.3 beschrieben, muss eine Compiler-Optimierungsmethode, die ein Programm auf mehrere Speicher verteilen möchte, das Programm zuerst in kleine Teile zerlegen. Die für eine Zerlegung in Frage kommenden Programmobjekte können mathematisch durch folgende Mengen modelliert werden:

- Menge der globalen Variablen:

  $G := \{g_1, \ldots, g_n\}$

- Menge der Funktionen:

  $F := \{f_1, \ldots, f_n\}$

- Menge der Basis-Blöcke:

  $BB := \{bb_1, \ldots, bb_n\}$

Diese Mengen können dann, je nachdem welche Programmobjekte von der Compiler-Optimierungsmethode berücksichtigt werden sollen, zu der Menge der Programmobjekte O zusammengefasst werden.

$$\text{Menge } O \subseteq F \cup BB \cup G = \{o_1, \ldots, o_n\} \tag{5.3}$$

Die Eigenschaften der Objekte können dann wie bei den Speicherpartitionen durch Funktionen abgefragt werden.

$$Size(o) := \text{ Größe des Programmobjekts } o \tag{5.4}$$

Da Funktionen und Basis-Blöcke jedoch keine unabhängigen Objekte sind (vgl. Kapitel 2.4.1), müssen die Abhängigkeiten modelliert werden. Dies kann zum einen durch zwei Hilfsfunktionen geschehen, die die Abfrage der Abhängigkeiten zwischen Funktionen und Basis-Blöcken ermöglichen. Alternativ können die Abhängigkeiten durch die Kanten des Kontrollflussgraphen modelliert werden.

$$\sigma(f, bb) = \begin{cases} 1, & f \in F, bb \in BB, bb \text{ gehört zur Funktion } f \\ 0, & \text{sonst} \end{cases} \qquad f \in O, bb \in O \tag{5.5}$$

$$BBC(f) = \sum_{i=1}^{n} \sigma(f, o_i) \qquad f \in O \tag{5.6}$$

Die Funktion $\sigma(f, bb)$ gibt an, ob der Basis-Block bb zur Funktion f gehört und die Funktion $BBC(f)$ bestimmt die Anzahl der zur Funktion f gehörenden Basis-Blöcke. Beide Funktionen sind so allgemein definiert, dass sie als Argumente Objekte der Menge O akzeptieren.

Um die Abhängigkeiten durch die Kanten des Kontrollflussgraphen zu modellieren, wird die Menge der Kanten V definiert.

$$\text{Menge } V := \{v_1, \ldots, v_p\} \tag{5.7}$$

Des weiteren benötigt das Modell eine Möglichkeit, die Zusammenhänge zwischen den Basis-Blöcken und den verbindenden Kontrollflusskanten abzufragen. Aus diesem Grunde wird die Funktion $V_k(i, x)$ definiert, die als Parameter zwei Indizes benötigt, die auf zwei Programmobjekte in der Menge O verweisen. Als Ergebniss liefert die Funktion einen Index in die Menge V, der es ermöglicht, die verbindende Kante aus der Menge V zu erhalten.

$$V_k(i, x) = \begin{cases} k, \ o_i \in BB \text{ und } o_x \in BB \text{ sind durch die Kante } v_k \in V \text{ verbunden} \\ -1, \ \text{sonst} \end{cases} \tag{5.8}$$

5.1.3 Nutzenfunktionen

Alle Optimierungsmethoden benötigen Bewertungsfunktionen, die ihnen die Analyse des Programms im Hinblick auf ihr Optimierungsziel ermöglichen. Aus diesem Grunde benötigt die während dieser Arbeit entwickelte Optimierungsmethode eine Bewertungsfunktion, die Aussagen über den Energiebedarf einzelner Programmobjekte ermöglicht. Da dies in Abhängigkeit der Speicherpartition, in der das Programmobjekt abgelegt werden soll, geschehen muss, wird die Funktion $Energy(o, mp)$ definiert, wobei die Grundlagen der Enrgieberechnung, die der Funktion zugrundeliegen, bereits im Kapitel 4.3.3 dargestellt worden sind. Ziel der Optimierungsmethoden wäre es demnach, die Summe der Energie zu minimieren, die die Programmobjekte benötigen.

$$Energy(o, mp) := \text{ Energie, die } o \text{ benötigt, wenn es in } mp \text{ abgelegt wird} \quad (5.9)$$

Eine äquivalente Möglichkeit, mit der einige Optimierungsmethoden einfacher implementiert werden können, ist die Maximierung einer Benefit-Funktion. Eine solche kann durch die Spiegelung der Energie-Funktion $(-Energy(o, mp))$ definiert werden. Da es jedoch zweckmäßiger wäre, eine rein positive Benefit-Funktion zu definieren, muss der Bildbereich der Funktion durch Addition einer Konstante in den positiven Bereich verschoben werden. Dies ist möglich, da die Energiefunktion für ein gegebenes Programm eine obere Schranke hat. Die benötigte Konstante kann somit wie folgt berechnet werden $(c = \max(Energy(o, mp)) + 1)$.

$$Benefit(o, mp) = -Energy(o, mp) + c \quad (5.10)$$

5.1.4 Umsetzung der Problembeschreibung in ein ILP-Modell

Da das Optimierungsproblem aus Flexibilitätsgründen mit Hilfe der *ILP*-Technik (siehe Kapitel 2.5) gelöst wird, muss das zu lösende Problem in ein lineares Gleichungssystem transformiert werden, wobei die Lösung des Gleichungssystems die

Speicher-Partitionen, in der die Programmobjekte abgelegt werden sollen, liefern muss.

Zu diesem Zweck werden die binären Entscheidungsvariablen der Menge $\tilde{O}$ verwendet, die angeben, ob ein Programmobjekt in einer bestimmten Speicherpartition abgelegt werden soll. Die Aufgabe des *ILP*-Solvers ist es somit, die Werte der binären Entscheidungsvariablen der Menge $\tilde{O}$ entsprechend zu berechnen.

$$\textit{ILP}\text{-Entscheidungsmatrix } \tilde{O} = \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix}$$

$$\text{mit } \tilde{o}_{i,j} = \begin{cases} 1, & o_i \text{ wird in Speicherpartition } mp_j \text{ abgelegt} \\ 0, & \text{sonst} \end{cases} \tag{5.11}$$

Nachdem nun feststeht, dass der *ILP*-Solvers die Werte der binären Entscheidungsvariablen der Menge $\tilde{O}$ berechnen muss, kann nun eine entsprechende Zielfunktion aufgestellt werden.

$$\text{Minimize} \sum_{i=1}^{n} \sum_{j=1}^{m} [Energy(o_i, mp_j) \cdot \tilde{o}_{i,j}] \tag{5.12}$$

Es ist jedoch zu beachten, dass jedes Programmobjekt o nur in einer Speicherpartition mp abgelegt werden darf. Dies wird durch die folgenden Nebendingungen erreicht.

$$\forall i \text{ mit } 1 \leq i \leq n : \sum_{j=1}^{m} \tilde{o}_{i,j} = 1 \tag{5.13}$$

Des weiteren dürfen natürlich nur so viele Programmobjekte in einer Speicherpartition abgelegt werden, wie diese bedingt durch ihre Größe aufnehmen kann. Dies lässt sich durch weitere Nebenbedingungen sicherstellen.

$$\forall j \text{ mit } 1 \leq j \leq m : \sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] \leq Size(mp_j) \tag{5.14}$$

Bisher sind noch keine Abhängigkeiten (siehe Kapitel 4.3) zwischen den Programmobjekten berücksichtigt. Da es jedoch mehrere Möglichkeiten gibt, diese Abhängigkeiten zu modellieren, wurden im Rahmen dieser Diplomarbeit zwei unterschiedliche Modelle ausgearbeitet. Diese sind in den folgenden Kapiteln beschrieben.

5.2 Das Top-Down Modell

Dieses Modell entstand direkt aus dem Modell, das in der Diplomarbeit [Zob01] vorgestellt wurde (siehe Kapitel 3.1). Es berücksichtigt, dass Funktionen aus Basis-Blöcken bestehen und somit vom *ILP*-Modell entweder die Funktion oder alle enthaltenen Basis-Blöcke ausgewählt werden müssen. Der Vorteil der gesonderten Betrachtung von Funktionen liegt in der Tatsache begründet, dass für die Energieberechnung eines Basis-Blocks bis zu zwei zusätzliche Sprungbefehle angenommen werden müssen. Dieses Modell liefert dem *ILP*-Solver somit einen Anreiz, komplette Funktionen auszuwählen, da die Energie einer kompletten Funktion niedriger ist als die Energiesumme der enthaltenen Basis-Blöcke (vgl. Kapitel 4.3.3). Im Gegensatz zu dem ursprünglichen Modell [Zob01] berücksichtigt dieses Modell nicht, dass kleine Funktionsteile[1] zusammenhängend abgelegt werden sollten. Dies hätte zwar den Vorteil, das einige Sprungbefehle, die die Sprungdistanz zwischen den Speicherpartitionen überbrücken, eingespart werden könnten. Durch den Aufbau dieses Modells, das von den Funktionen ausgehend die Abhängigkeiten modelliert (Top-Down), würde die Beschreibung dieser Zusammenhänge jedoch sehr aufwendig werden. Aus diesem Grunde wurde für das Top-Down Modell auf die Modellierung dieser Abhängigkeiten verzichtet und das Alternative Bottom-Up Modell (siehe Kapitel 5.3) entwickelt.

Das Top-Down Modell modelliert die Abhängigkeiten zwischen Funktionen und Basis-Blöcken im *ILP*-Modell durch Nebenbedingungen. Um dies zu ermöglichen müssen jedoch die Nebenbedingungen der Gleichung 5.13 durch die folgenden

[1] z.B. zwei sequentiell verbundene Basis-Blöcke einer Funktion

Nebenbedingungen ersetzt werden, da die ursprüngliche Form fordert, dass jedes Programmobjekt genau einmal ausgewählt werden muss.

$$\forall i \text{ mit } 1 \leq i \leq n : \sum_{j=1}^{m} \tilde{o}_{i,j} \leq 1 \tag{5.15}$$

Die neue Form hingegen fordert nur noch, dass jedes Objekt maximal einmal ausgewählt werden darf. Dies würde jedoch zwangsläufig dazu führen, das der *ILP*-Solver kein einziges Programmobjekt mehr auswählt, da nun die triviale Lösung, in der alle Entscheidungsvariablen der Menge $\tilde{O}$ gleich 0 sind, eine gültige Lösung wäre. Dieses Problem kann jedoch durch die Umkehrung der Zielfunktion behoben werden.

$$\text{Maximize} : \sum_{i=1}^{n} \sum_{j=1}^{m} [Benefit(o_i, mp_j) \cdot \tilde{o}_{i,j}] \tag{5.16}$$

Durch die Maximierung wählt der *ILP*-Solver nun jedes Programmobjekt aus, wenn dies nicht durch eine Nebenbedingung verhindert wird.

Nach diesen Änderungen ist es somit möglich, die Forderung, dass entweder die Funktion oder alle enthaltenen Basis-Blöcke ausgewählt werden, durch weitere Nebenbedingungen zu realisieren. Die folgenden Gleichungen verhindern, dass der *ILP*-Solver gleichzeitig die Entscheidungsvariable der Funktion o_i und eine der Entscheidungsvariablen der enthaltenen Basis-Blöcke o_x auf 1 setzt.

$(\forall i,\ 1 \leq i \leq n, o_i \in F)$, $(\forall j,\ 1 \leq j \leq m)$, $(\forall y,\ 1 \leq y \leq m)$:

$$(BBC(o_i) \cdot \tilde{o}_{i,j}) + \underbrace{\sum_{x=1}^{n} [\sigma(o_i, o_x) \cdot \tilde{o}_{x,y}]}_{\text{Anzahl einzeln ausgewählter Basis-Blöcke der Funktion } o_i} \leq BBC(o_i) \tag{5.17}$$

Zusammenfassung:

- Entscheidungsvariablen:

$$\tilde{O} = \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix}$$

- Zielfunktion:

$$\text{Maximize} : \sum_{i=1}^{n} \sum_{j=1}^{m} [Benefit(o_i, mp_j) \cdot \tilde{o}_{i,j}]$$

- Jedes Objekt maximal einmal auswählen:
  $(\forall i, 1 \leq i \leq n)$

$$\sum_{j=1}^{m} \tilde{o}_{i,j} \leq 1$$

- Partitionsgrößen beachten:
  $(\forall j, 1 \leq j \leq m)$

$$\sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] \leq Size(mp_j)$$

- Wähle entweder die Funktion oder alle enthaltenen Basis-Blöcke aus:
  $(\forall i,\ 1 \leq i \leq n, o_i \in F)$, $(\forall j,\ 1 \leq j \leq m)$, $(\forall y,\ 1 \leq y \leq m)$:

$$(BBC(o_i) \cdot \tilde{o}_{i,j}) + \sum_{x=1}^{n} [\sigma(o_i, o_x) \cdot \tilde{o}_{x,y}] \leq BBC(o_i)$$

5.3 Das Bottom-Up Modell

Das Bottom-Up Modell orientiert sich im Gegensatz zu dem Top-Down Modell an dem Kontrollflussgraphen des Programms. Das heißt, dass dem *ILP*-Solver die Funktionen nicht als Objekte der Menge O bekannt gemacht werden. Der Zusammenhang zwischen den Basis-Blöcken wird über Nebenbedingungen, die die Kanten des Kontrollflussgraphen beschreiben, hergestellt. Dies hat zur Folge, dass dieses Modell im Vergleich zum Bottom-Up Modell eine geringere Beschreibungskomplexität aufweist und die Ergebnisse im allgemeinen besser sind, da auch kleine Funktionsteile zusammenhängend abgelegt werden.

Wie bei den Objekten der Menge $\tilde{O}$ benötigt das *ILP*-Modell für die Kanten ebenfalls eine Entscheidungsmatrix. Das Modell sieht vor, dass es für jede Kante des Programms, die keinen Wechsel der Speicherpartition verursacht, einen Energievorteil $StmtEnergy_{jmp}$ in der *ILP*-Zielfunktion berücksichtigt. Aus diesem Grunde enthält die Entscheidungsmatrix $\tilde{V}$ für jede Kante v und jede Speicherpartition mp eine binäre Entscheidungsvariable $\tilde{v}_{k,j}$, die angibt, ob die entsprechende Kante ohne Wechsel der Speicherpartition ausgeführt wird.

$$\tilde{V} = \begin{pmatrix} \tilde{v}_{1,1} & \cdots & \tilde{v}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{v}_{p,1} & \cdots & \tilde{v}_{p,m} \end{pmatrix} \tag{5.18}$$

$$\text{mit } \tilde{v}_{k,j} = \begin{cases} 1, & mp_j \text{ wird beim Ausführen der Kante } v_k \text{ nicht verlassen} \\ 0, & \text{sonst} \end{cases}$$

Nach der Definition der zusätzlichen Entscheidungsvariablen kann nun eine modifizierte Zielfunktion aufgestellt werden. Wie im Kapitel 4.3.3 beschrieben, wird bei der Energieberechnung eines Basis-Blocks davon ausgegangen, dass zwei zusätzliche Sprungbefehle an den Basis-Block angefügt werden müssen. Durch die neuen Entscheidungsvariablen ist es jedoch Möglich, die so berechnete Worst-Case-Energie zu korrigieren.

$$\text{Minimize} : \sum_{i=1}^{n}\sum_{j=1}^{m}[Energy(o_i, mp_j)\cdot\tilde{o}_{i,j}] - \sum_{k=1}^{p}\sum_{j=1}^{m}[StmtEnergy_{jmp}\cdot\tilde{v}_{k,j}] \tag{5.19}$$

Im Gegensatz zu dem Top-Down-Modell brauchen die Nebenbedingungen, die sicherstellen, das jedes Objekt genau einmal ausgewählt wird, nicht angepasst zu werden, da es zwischen den Entscheidungsvariablen der Programmobjekte keine Abhängigkeiten gibt. Da durch die Entscheidungsvariablen V jedoch die wirklich benötigten zusätzlichen Sprungbefehle bekannt sind, können die Partitionsgrößen-Nebenbedingungen ähnlich der neuen Zielfunktion angepasst werden.

$$\forall j \text{ mit } 1 \leq j \leq m : \sum_{i=1}^{n}[Size(o_i)\cdot\tilde{o}_{i,j}] - \sum_{k=1}^{p}[StmtSize_{jmp}\cdot\tilde{v}_{k,j}] \leq Size(mp_j) \tag{5.20}$$

Die folgenden Nebenbedingungen müssen nun dafür sorgen, dass die Entscheidungsvariablen V entsprechend berechnet werden. Dies ist ohne weiteres möglich, da die entsprechende Entscheidungsvariable $\tilde{v}_{k,j}$ genau dann nicht gesetzt werden darf, wenn die beiden durch die Kante v_k verbundenen Basis-Blöcke nicht in der gleichen Speicherpartition mp_j abgelegt werden. Dies wird durch die folgenden Nebenbedingungen sichergestellt. Sollten die folgenden Nebenbedingungen das Setzen einer Entscheidungsvariablen nicht verhindern, wird der *ILP*-Solver die Entscheidungsvariable auf 1 setzen, da er durch die minimierende Zielfunktion einen Anreiz hat, dies zu tun.

$(\forall k \neq -1, k = V_k(i,x) \text{ mit } 1 \leq i \leq n, 1 \leq x \leq n)$, $(\forall j, 1 \leq j \leq m)$:

$$\tilde{o}_{i,j} + \tilde{o}_{x,j} - 2\tilde{v}_{k,j} \geq 0 \tag{5.21}$$

Zusammenfassung:

- Entscheidungsvariablen:

$$\tilde{O} = \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix} \qquad \tilde{V} = \begin{pmatrix} \tilde{v}_{1,1} & \cdots & \tilde{v}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{v}_{p,1} & \cdots & \tilde{v}_{p,m} \end{pmatrix}$$

- Zielfunktion:

$$\text{Minimize} : \sum_{i=1}^{n}\sum_{j=1}^{m}[Energy(o_i, mp_j) \cdot \tilde{o}_{i,j}] - \sum_{k=1}^{p}\sum_{j=1}^{m}[StmtEnergy_{jmp} \cdot \tilde{v}_{k,j}]$$

- Jedes Objekt genau einmal auswählen:
  $(\forall i, 1 \leq i \leq n)$

$$\sum_{j=1}^{m} \tilde{o}_{i,j} = 1$$

- Partitionsgrößen beachten:
  $(\forall j, 1 \leq j \leq m)$

$$\sum_{i=1}^{n}[Size(o_i) \cdot \tilde{o}_{i,j}] - \sum_{k=1}^{p}[StmtSize_{jmp} \cdot \tilde{v}_{k,j}] \leq Size(mp_j)$$

- Edge-Constrains:
  $(\forall k \neq -1, k = V_k(i,x)$ mit $1 \leq i \leq n, 1 \leq x \leq n)$, $(\forall j, 1 \leq j \leq m)$:

$$\tilde{o}_{i,j} + \tilde{o}_{x,j} - 2\tilde{v}_{k,j} \geq 0$$

5.4 Mögliche Modell-Erweiterungen

In den vorangegangenen Kapiteln wurden zwei Modelle vorgestellt, die in der Lage sind, beliebige Programmobjekte auf unterschiedliche Speicherpartitionen zu verteilen. Der Hauptaspekt bei der Entwicklung der Modelle war die Energieoptimierung. Nichtsdestotrotz sind beide Modelle durch das Austauschen der Nutzenfunktion in der Lage, andere Optimierungen durchzuführen. Das Optimierungspotential hängt jedoch hauptsächlich davon ab, ob das neue Optimierungsziel durch eine Speicherpartitionierung erreicht werden kann. Zum Beispiel wäre es denkbar, dass durch die beiden Modelle eine Geschwindigkeitsverbesserung erreicht werden kann. Eine Größenoptimierung wäre hingegen nicht realisierbar.

Wie bereits im Kapitel 4.3.3 dargestellt, gibt es neben den dynamischen Zugriffskosten Fixkosten, die zum Beispiel durch die periodischen Refreshs einiger Speichersysteme verursacht werden. Aus diesem Grunde erscheint es sinnvoll, diese Fixkosten in den Modellen ebenfalls zu berücksichtigen. Ein mögliches Anwendungsszenario wäre zum Beispiel das Abschalten von temporär nicht benötigten Speicherpartitionen.

Um die Fixkosten der Speicherpartitionen bei der Lösung der Modelle zu berücksichtigen, wird eine Funktion, die die jeweiligen Fixkosten der Speicherpartitionen liefert, benötigt. Für eine Energieoptimierung kann diese folgendermaßen definiert werden.

$$FixEnergy(mp) := \text{ Energie, die } mp \text{ permanent verbraucht} \tag{5.22}$$

Da die Fixkosten jedoch nur auftreten, wenn das zu optimierende Programm die Speicherpartition auch tatsächlich benutzt[2], werden für das Modell binäre Variablen (Menge $\widetilde{MP} = \{\widetilde{mp}_1, \ldots, \widetilde{mp}_m\}$) benötigt, die angeben, ob der *ILP*-Solver der Speicherpartion Programmobjekte zugeordnet hat. Diese Variablen können durch Nebenbedingungen in den Modellen entsprechend berechnet werden. Zur Veranschaulichung wird im folgenden das ursprüngliche Modell, das noch keine Abhängigkeiten zwischen den Programmobjekten berücksichtigt, erweitert, wobei die benötigten Änderungen in **fett** hervorgehoben sind.

[2]Es wird davon ausgegangen, dass eine nicht verwendete Speicherpartition abgeschaltet wird und somit keine Kosten verursacht.

- Entscheidungsvariablen:

$$\tilde{O} = \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix} \qquad \widetilde{\mathbf{MP}} = \{\widetilde{\mathbf{mp}}_\mathbf{1}, \ldots, \widetilde{\mathbf{mp}}_\mathbf{m}\}$$

- Zielfunktion:

$$\text{Minimize} : \sum_{i=1}^{n}\sum_{j=1}^{m}[Energy(o_i, mp_j) \cdot \tilde{o}_{i,j}] + \sum_{\mathbf{j=1}}^{\mathbf{m}}\left[\mathbf{FixEnergy}(\mathbf{mp_j}) \cdot \widetilde{\mathbf{mp}}_\mathbf{j}\right]$$

- Jedes Objekt genau einmal auswählen:
  $(\forall i, 1 \leq i \leq n)$

$$\sum_{j=1}^{m} \tilde{o}_{i,j} = 1$$

- Partitionsgrößen beachten:
  $(\forall j, 1 \leq j \leq m)$

$$\sum_{i=1}^{n}[Size(o_i) \cdot \tilde{o}_{i,j}] \leq Size(mp_j)$$

- **Entscheidungsvariablen der Menge $\widetilde{\mathbf{MP}}$ berechnen:**
  $(\forall \mathbf{j}, \mathbf{1} \leq \mathbf{j} \leq \mathbf{m})$

$$\sum_{\mathbf{i=1}}^{\mathbf{n}}[\tilde{\mathbf{o}}_{\mathbf{i,j}}] - \mathbf{n} \cdot \widetilde{\mathbf{mp}}_\mathbf{j} \leq \mathbf{0}$$

6 Planung der Simulation

Nachdem in den vorangegangenen Kapiteln alle Details der Compiler-Optimierungsmethode besprochen worden sind, können nun die Möglichkeiten und Grenzen der Methode untersucht werden. Da diese Arbeit eine Verallgemeinerung der in [Zob01] vorgestellten Methode vornimmt, bietet es sich – nicht zuletzt wegen der Vergleichbarkeit der Ergebnisse – an, die gleiche Simulationsumgebung zu verwenden.

Wie im Kapitel 4.2 dargestellt, müssen vor der Simulationsdurchführung einige Konfigurationseinstellungen vorgenommen werden, die das Verhalten der einzelnen Programme bestimmen. Aus diesem Grunde werden die möglichen Konfigurationseinstellungen im nächsten Kapitel detailliert besprochen. Anschließend folgt im Kapitel 6.2 eine Evaluation der möglichen Simulationsparameter, da die in dieser Arbeit vorgestellten Modelle komplexere Parametrisierungsmöglichkeiten bieten und aus diesem Grunde eine Übernahme aus den vorhergegangenen Arbeiten nicht möglich ist. Die Vergleichbarkeit der Ergebnisse bleibt dennoch gewährleistet.

6.1 Konfiguration der Simulationsumgebung

6.1.1 Die Konfiguration der Speicherpartitionierung

Die Konfigurationsdatei `boardconfig.dat` enthält die Konfigurationsdaten der Speicherpartitionen. Zu diesen Daten gehören unter anderem die Basis-Adresse, der Partitionsname sowie die Zugriffsenergie, die bei einem Speicherzugriff benötigt wird. Diese Datei bildet die Grundlage für die entwickelte Compiler-Optimierungsmethode. Des weiteren benötigt der *enProfiler* diese Datei, um detaillierte

Informationen über den Programmverlauf auszugeben. Die Bedeutung der einzelnen Felder ist in der Datei angegeben. Weitere Informationen können jedoch der Diplomarbeit [Kna01] entnommen werden.

Die Konfigurationsdatei `boardconfig.dat`:

```
; 1:number [int], 0 = undef, numbers have to be incremented
; 2:start  [32Bit(hexadezimal)]
; 3:size   [32Bit(hexadezimal)]
; 4:width in bytes [int]
; 5:waitstates     [int]
; 6:energy *3,3V/33Mhz/1000 [float]
; 7:read or write
; 8:Konstante für Hammingdistanz auf Datenbus
; 9:Konstante für Anzahl 1'en auf Datenbus
;10:Konstante für Hammingdistanz auf Adressbus
;11:Konstante für Anzahl 1'en auf Adressbus
;12:Name des Speichers als String

;1      2         3      4  5   6        7   8    9   10   11   12
;-------------------------------------------------------------------
00,  00400000, 00020000, 1, 1, 154.8,    r, 0.2, 0.1, 0.2, 0.1, ROM
01,  00400000, 00020000, 2, 1, 240.0,    r, 0.2, 0.1, 0.2, 0.1, ROM
02,  00400000, 00020000, 4, 3, 493.2,    r, 0.2, 0.1, 0.2, 0.1, ROM
03,  00500000, 00200000, 1, 1, 154.8,    r, 0.2, 0.1, 0.2, 0.1, offchip
04,  00500000, 00200000, 2, 1, 240.0,    r, 0.2, 0.1, 0.2, 0.1, offchip
05,  00500000, 00200000, 4, 3, 493.2,    r, 0.2, 0.1, 0.2, 0.1, offchip
06,  00700000, 00000040, 1, 0,   4.9493, r, 0.2, 0.1, 0.2, 0.1, mem64.1
07,  00700000, 00000040, 2, 0,   4.9493, r, 0.2, 0.1, 0.2, 0.1, mem64.1
08,  00700000, 00000040, 4, 0,   4.9493, r, 0.2, 0.1, 0.2, 0.1, mem64.1
...
```

6.1.2 Kommandozeilenparameter des Compilers enCC

Die Konfiguration des Compilers geschieht mit Kommandozeilenparametern, die im folgenden dargestellt sind. Für die im Zuge dieser Arbeit entwickelte Compiler-Optimierungsmethode ist vor allem der Parameter `-m` von Bedeutung, da mit ihm die zu verwendende Speicherpartitionierung konfiguriert wird. Er akzeptiert als Argument zum einen eine einzelne Zahl, die die zu verwendende Scratchpad-

Speichergröße angibt[1]. Andererseits akzeptiert er eine kommaseparierte Liste von Partitionsnamen, die es ermöglicht, mehrere Speicherpartitionen zu konfigurieren (vgl. `boardconfig.dat`).

Die Kommandozeilenparameter des Compilers *enCC*:

```
Usage:  ir2asm_ARM [-o optimize_option]
           [-s set option] [-u unset option]
           [-m (onchipsize | 'memName1, memName2, ...']
           [-b maxBBSize]
           [-c <cachewords>:
               <cacheassociativity>:
               <cacheblocks>]
               <IR-filename>

  optimize_option = time | energy | power | size
  option          = highreg | registerpipelining | instructionscheduling |
                    licm | dynamicprofiling | overlay | gentrace | optCache |
                    coding | debug | bigendian | mlold
```

Die beiden anderen wichtigen Paramter `-b` und `-s mlold` ermöglichen eine genauere Steuerung des Verhaltens der entwickelten Optimierungsmethode, wobei der Parameter `-b` es ermöglicht, die maximale Größe der Basis-Blöcke auf eine bestimmte Größe einzuschränken (siehe Kapitel 4.3.1). Der Parameter `-s mlold` hingegen erzwingt die Verwendung des alternativen »Top-Down Modells«.

6.1.3 Die Linker-Konfigurationsdatei

Die Konfigurationsdatei `scat.txt` konfiguriert über den sogenannten Scatter-Loading-Mechanismus [ARM] die Speicheradressen, an denen eine bestimmte Objektdatei geladen werden soll. Wie im Kapitel 4.2 dargestellt, überführt die Arbeitsumgebung ein Programm in mehrere Objektdateien, die anschließend vom Linker zu einem ausführbaren Programm zusammengefasst werden. Dabei werden die Inhalte der Objektdateien mit Adressen versehen, an denen selbige bei der Programmausführung geladen werden. Aus diesem Grunde muss die Konfigurationsdatei

[1] Konfiguration der Speicherpartitionierung: »Name des Speichers: `onchip`«

`scat.txt` dafür sorgen, dass der Linker die Objektdateien mit der Adresse der entsprechenden Speicherpartition versieht.

Die Konfigurationsdatei `scat.txt`:

```
FLASH 0x00400000 0x00400000 {
  FLASH 0x00400000 {
    regioninit.o (+RO)
    startup.o    (+RO)
    ...
  }
  offchip 0x00500000 {
    * (+RO)  * (+RW)  * (+ZI)
    ...
    mem.0.o    (+RO,+RW,+ZI)
   }
   mem64.1  0x00700000 { mem.1.o  (+RO,+RW,+ZI) }
   mem64.2  0x00704000 { mem.2.o  (+RO,+RW,+ZI) }
   mem64.3  0x00708000 { mem.3.o  (+RO,+RW,+ZI) }
   mem64.4  0x0070C000 { mem.4.o  (+RO,+RW,+ZI) }
   ...
   mem32k.1  0x00800000 { mem.37.o (+RO,+RW,+ZI) }
   mem32k.2  0x00810000 { mem.38.o (+RO,+RW,+ZI) }
   mem32k.3  0x00820000 { mem.39.o (+RO,+RW,+ZI) }
   mem32k.4  0x00830000 { mem.40.o (+RO,+RW,+ZI) }
}
```

6.2 Auswahl geeigneter Modellparameter

Ein oft unterschätztes Problem der Simulationsvorbereitung ist die Auswahl geeigneter Simulationsparameter, da diese maßgeblich den Erfolg der Simulation und die Güte der Simulationsergebnisse beeinflussen. Aus diesem Grunde sind der Auswahl geeigneter Parameter die folgenden Kapitel gewidmet.

6.2.1 Geeignete Programme

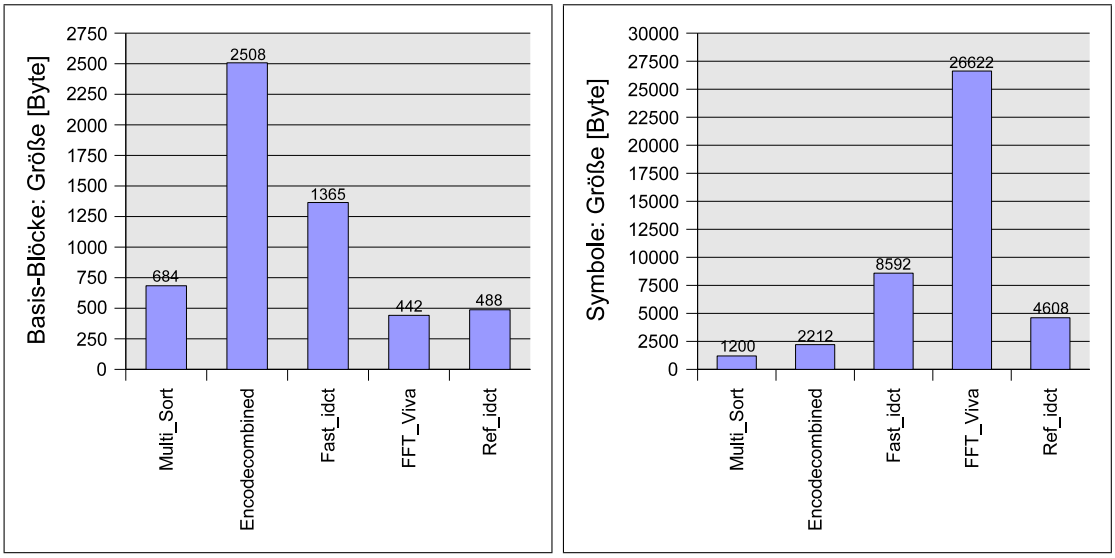
Für die Durchführung der Simulation werden geeignete Testprogramme benötigt, die zu exemplarischen und gegebenenfalls übertragbaren Ergebnissen führen, wobei die Wahl auf die in der Tabelle 6.1 aufgeführten Programme fiel.

Programm	Beschreibung
Multi_Sort.c	Künstliches Testprogramm (mehrere Sortieralgorithmen)
Encodecombined.c	GSM
FFT_Viva.c	Fast Fourier Transformation
Fast_idct.c	Inverse Diskrete Cosinus Transformation
Ref_idct.c	Inverse Diskrete Cosinus Transformation (Referenzimpl.)

Tabelle 6.1: Ausgewählte Programme für die Simulation

Diese haben in den vorangegangenen Arbeiten bereits mehrfach Verwendung gefunden und ermöglichen somit die Vergleichbarkeit der Ergebnisse. Darüber hinaus lassen diese Programme Aussagen über das Energiesparpotential in eingebetteten Systemen zu, da diese typische Anwendungsfälle implementieren. Um die Ergebnisse der Versuche besser interpretieren zu können, sind jedoch detaillierte Informationen über den Aufbau der Programme und deren Eigenschaften vonnöten.

Als erste interessante Eigenschaft kann die Programmgröße betrachtet werden, da sich anhand dieser die Möglichkeiten einer sinnvollen Speicherpartitionierung ableiten lassen. Das heißt, dass zum Beispiel bei sehr kleinen Programmen eine oder mehrere sehr große Speicherpartitionen wahrscheinlich nur geringe Energievorteile bewirken würden. Die Gesamtgröße der Programmobjekte in den ausgewählten Programmen wird durch die Abbildung 6.1 veranschaulicht.



(a) Basis-Blöcke und Funktionen (b) Symbole

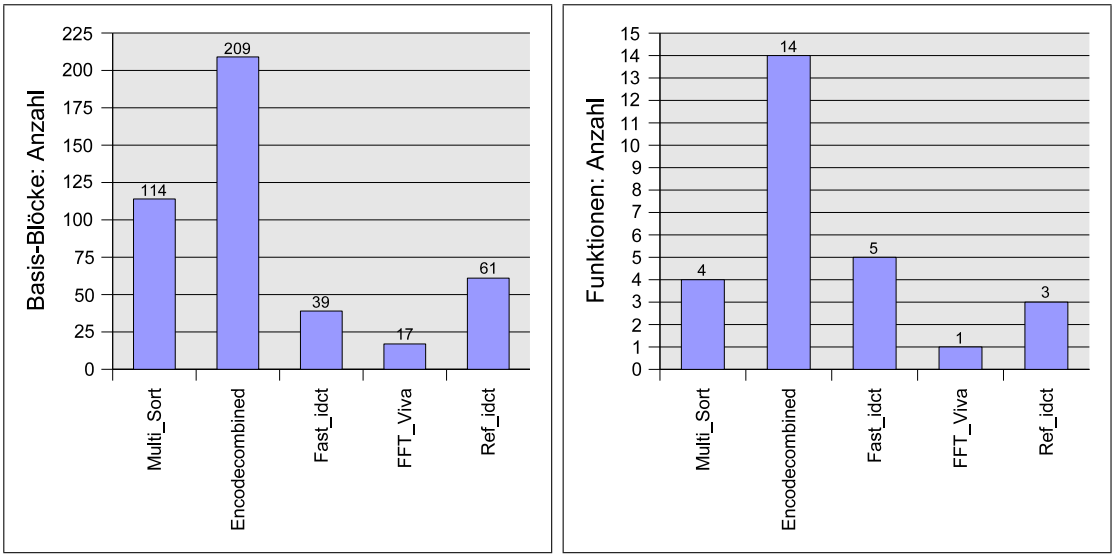
Abbildung 6.1: Die Gesamtgröße der Programmobjekte in ausgewählten Programmen

Eine weitere wichtige Eigenschaft ist die Anzahl der Programmobjekte (Funktionen, Basis-Blöcke, Variablen), aus denen ein Programm aufgebaut ist, da diese Eigenschaft direkten Einfluss auf die Modellkomplexität hat und somit die Laufzeit des *ILP-Solvers*, der für die Lösung der Modelle eingesetzt wird, direkt beeinflusst. Die jeweilige Anzahl der Programmobjekte in den ausgewählten Programmen wird durch Abbildung 6.2 dokumentiert.

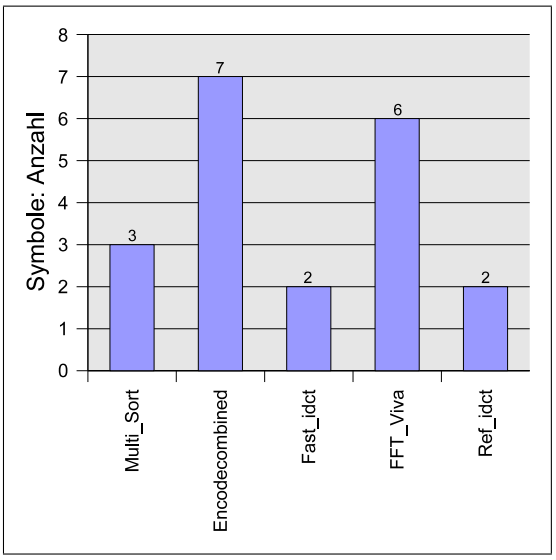
Interessant ist auch die maximale Programmobjektgröße (vgl. Abbildung 6.3), denn das Modell kann kleine Speicherpartitionen nur dann ausnutzen, wenn es möglichst viele Programmobjekte in die zur Verfügung stehenden Speicherpartitionen verschieben kann. Sollten die Programmobjekte jedoch zu groß werden, kann das Modell die kleinen Speicherpartitionen nicht mehr benutzen. Der Energiebedarf würde in diesem Fall wieder steigen.

6.2.2 Speicherpartitionen und sinnvolle Kombinationen

Nachdem nun die Testprogramme ausgewählt wurden und deren Eigenschaften dargestellt worden sind, müssen nun, nicht zuletzt wegen der sehr großen Anzahl an Möglichkeiten, geeignete Kombinationen von Speicherpartitionierungen

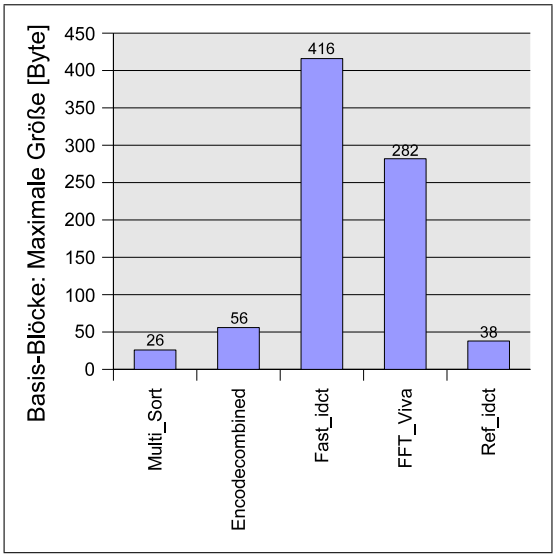


(a) Basis-Blöcke (b) Funktionen

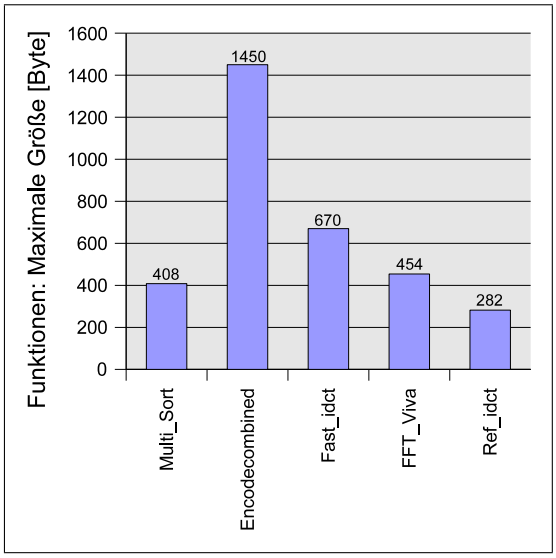


(c) Symbole

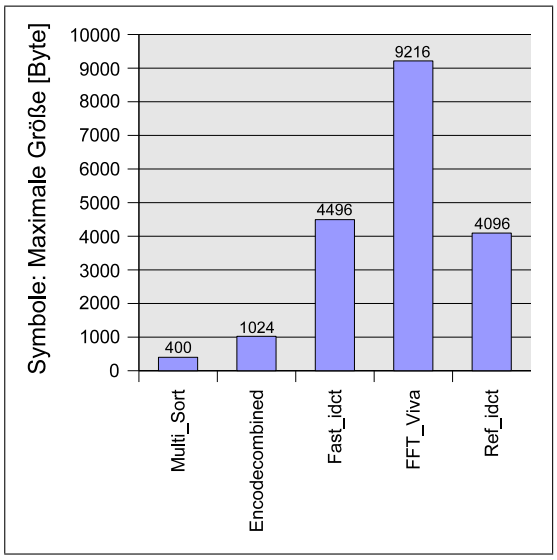
Abbildung 6.2: Die Anzahl der Programmobjekte in ausgewählten Programmen



(a) Basis-Blöcke



(b) Funktionen



(c) Symbole

Abbildung 6.3: Die maximalen Objektgrößen in ausgewählten Programmen

gefunden werden, wobei die Komplexität der Ergebnisgenerierung ebenfalls berücksichtigt werden muss.

Der erste wichtige Parameter ist die Gesamtspeichergröße, da jedes Programm einen Mindestbedarf an zur Verfügung stehendem Speicher hat. Eine naheliegende Lösung würde demnach einen sehr großen Speicher annehmen, so dass ausreichend Speicher für jedes ausgewählte Programm zur Verfügung stünde. Da ungenutzte Ressourcen jedoch in der Praxis auch Energie benötigen, ist es zur Erreichung des Ziels dieser Arbeit, den Energiebedarf von *ES* zu senken, unumgänglich, die zur Verfügung stehende Speichergröße zu begrenzen, wobei die Grenzen aus Gründen der Speicherorganisation durch Zweierpotenzen beschrieben werden sollten. Aus diesen Gründen werden Speichergrößen von 64 Byte (2^6) bis 32 Kilobyte (2^{15}) für die Simulation verwendet.

Nachdem die Gesamtspeichergröße feststeht, muss der Speicher nun in kleinere Speicher zerlegt werden, wobei diese Speicher aus Gründen der Speicherorganisation ebenfalls Größen, die durch Zweierpotenzen ausgedrückt werden können, haben sollten.

Erste Überlegungen ergaben zwei Möglichkeiten, wie eine sinnvolle Speicherpartitionierung aussehen könnte. Die erste Möglichkeit partitioniert, indem sie eine gegebene Speichergröße durch Teilung in jeweils gleichgroße Speicherpartitionen vornimmt (siehe Tabelle 6.2(a)). Die zweite Möglichkeit hingegen zerlegt eine gegebene Speichergröße in zwei gleich große Partitionen. Im nächsten Schritt wird eine der Partitionen wiederum in zwei gleich große Partitionen geteilt (siehe Tabelle 6.2(b)), wobei dieser Prozess solange fortgesetzt wird, bis die kleinste Partition eine Größe von 64 Byte erreicht.

Eine Zusammenfassung und Systematisierung beider Ansätze und die Erweiterung der Darstellung auf die Speichergrößen bis 1024 Byte kann der Tabelle A.1 im Anhang dieser Arbeit entnommen werden. Da die Anzahl der dargestellten Kombinationsmöglichkeiten jedoch mit zunehmender Speichergröße sehr schnell anwächst, müssen diese für die Simulationsdurchführung weiter reduziert werden.

Untersuchungen haben ergeben, dass viele gleich große Partitionen die Zeit, die der *ILP*-Solver benötigt, um eine Modellösung zu errechnen, sehr stark ansteigen lassen. Aus diesem Grunde erscheint es zur Reduktion der Komplexität sinnvoll, die Anzahl gleich großer Partitionen einzuschränken. Des weiteren besteht die

Anzahl Partitionen der Größe:				
1024	512	256	128	64
1	0	0	0	0
0	2	0	0	0
0	0	4	0	0
0	0	0	8	0
0	0	0	0	16

(a) Anzahl ausgewählter Speicher bei der Partitionierung in gleich große Partitionen

Anzahl Partitionen der Größe:				
1024	512	256	128	64
1	0	0	0	0
0	2	0	0	0
0	1	2	0	0
0	1	1	2	0
0	1	1	1	2

(b) Anzahl ausgewählter Speicher bei der Partitionierung der jeweils kleinsten Partition

Tabelle 6.2: Zwei Möglichkeiten der Speicherpartitionierung

Möglichkeit, nur eine maximale Anzahl an Partitionen, aus der eine gegebene Speichergröße zusammengesetzt wird, zuzulassen.

In der Tabelle 6.3 sind die für die Simulation ausgewählten Speicherpartitionierungen dargestellt, wobei die maximale Partitionsanzahl auf acht Partitionen beschränkt wurde. Des weiteren wurden lediglich zwei gleichartige Speicherpartitionen zugelassen.

Gesamtgröße [Byte]	Anzahl der Partitionen	Anzahl Partitionen der Größe:									
		32K	16K	8K	4K	2K	1K	512	256	128	64
64	1	0	0	0	0	0	0	0	0	0	1
128	2	0	0	0	0	0	0	0	0	0	2
	1	0	0	0	0	0	0	0	0	1	0
256	3	0	0	0	0	0	0	0	0	1	2
	2	0	0	0	0	0	0	0	0	2	0
	1	0	0	0	0	0	0	0	1	0	0
512	4	0	0	0	0	0	0	0	1	1	2
	3	0	0	0	0	0	0	0	1	2	0
	2	0	0	0	0	0	0	0	2	0	0
	1	0	0	0	0	0	0	1	0	0	0
1024	5	0	0	0	0	0	0	1	1	1	2
	4	0	0	0	0	0	0	1	1	2	0
	3	0	0	0	0	0	0	1	2	0	0
	2	0	0	0	0	0	0	2	0	0	0
	1	0	0	0	0	0	1	0	0	0	0
2048	6	0	0	0	0	0	1	1	1	1	2
	5	0	0	0	0	0	1	1	1	2	0
	4	0	0	0	0	0	1	1	2	0	0
	3	0	0	0	0	0	1	2	0	0	0
	2	0	0	0	0	0	2	0	0	0	0
	1	0	0	0	0	1	0	0	0	0	0
4096	7	0	0	0	0	1	1	1	1	1	2
	6	0	0	0	0	1	1	1	1	2	0
	5	0	0	0	0	1	1	1	2	0	0
	4	0	0	0	0	1	1	2	0	0	0
	3	0	0	0	0	1	2	0	0	0	0
	2	0	0	0	0	2	0	0	0	0	0
	1	0	0	0	1	0	0	0	0	0	0
8192	8	0	0	0	1	1	1	1	1	1	2
	7	0	0	0	1	1	1	1	1	2	0
	6	0	0	0	1	1	1	1	2	0	0
	5	0	0	0	1	1	1	2	0	0	0
	4	0	0	0	1	1	2	0	0	0	0
	3	0	0	0	1	2	0	0	0	0	0
	2	0	0	0	2	0	0	0	0	0	0
	1	0	0	1	0	0	0	0	0	0	0

Gesamtgröße [Byte]	Anzahl der Partitionen	Anzahl Partitionen der Größe:									
		32K	16K	8K	4K	2K	1K	512	256	128	64
16384	8	0	0	1	1	1	1	1	1	2	0
	7	0	0	1	1	1	1	1	2	0	0
	6	0	0	1	1	1	1	2	0	0	0
	5	0	0	1	1	1	2	0	0	0	0
	4	0	0	1	1	2	0	0	0	0	0
	3	0	0	1	2	0	0	0	0	0	0
	2	0	0	2	0	0	0	0	0	0	0
	1	0	1	0	0	0	0	0	0	0	0
32768	8	0	1	1	1	1	1	1	2	0	0
	7	0	1	1	1	1	1	2	0	0	0
	6	0	1	1	1	1	2	0	0	0	0
	5	0	1	1	1	2	0	0	0	0	0
	4	0	1	1	2	0	0	0	0	0	0
	3	0	1	2	0	0	0	0	0	0	0
	2	0	2	0	0	0	0	0	0	0	0
	1	1	0	0	0	0	0	0	0	0	0

Tabelle 6.3: Ausgewählte Speicherpartitionierungen
(Anzahl der ausgewählten Speicherpartitionen)

7 Ergebnisse

Auf der Basis der bisherigen theoretischen Grundlagen und Überlegungen soll das folgende Kapitel der Validierung der Modelle und der Darstellung des möglichen Energiesparpotentials dienen, wobei die grundsätzliche Vorgehensweise der Ergebnisgenerierung bereits im Kapitel 6 besprochen wurde. Ziel ist es somit, die fünf ausgewählten Programme mit den Speicherpartitionierungen, die in der Tabelle 6.3 dargestellt sind, zu untersuchen und das Energiesparpotential darzustellen. Des weiteren sollen die Vor- und Nachteile der statischen Häufigkeitsanalyse mit denen des dynamischen Profilings (vgl. Kapitel 4.3.2) verglichen werden und dessen Auswirkungen auf das Energiesparpotential dargestellt werden. Alle Ergebnisse wurden mit dem »Bottom-Up Modell« erstellt, da das »Top-Down Modell« keine besseren Ergebnisse erzeugen könnte und zudem nicht so flexibel an neue Gegebenheiten angepasst werden kann. Des weiteren wurde untersucht, wie sich die Möglichkeit Basis-Blöcke auf eine maximale Größe zu beschränken, auf das Energiesparpotential auswirkt (vgl. Kapitel 4.3.1).

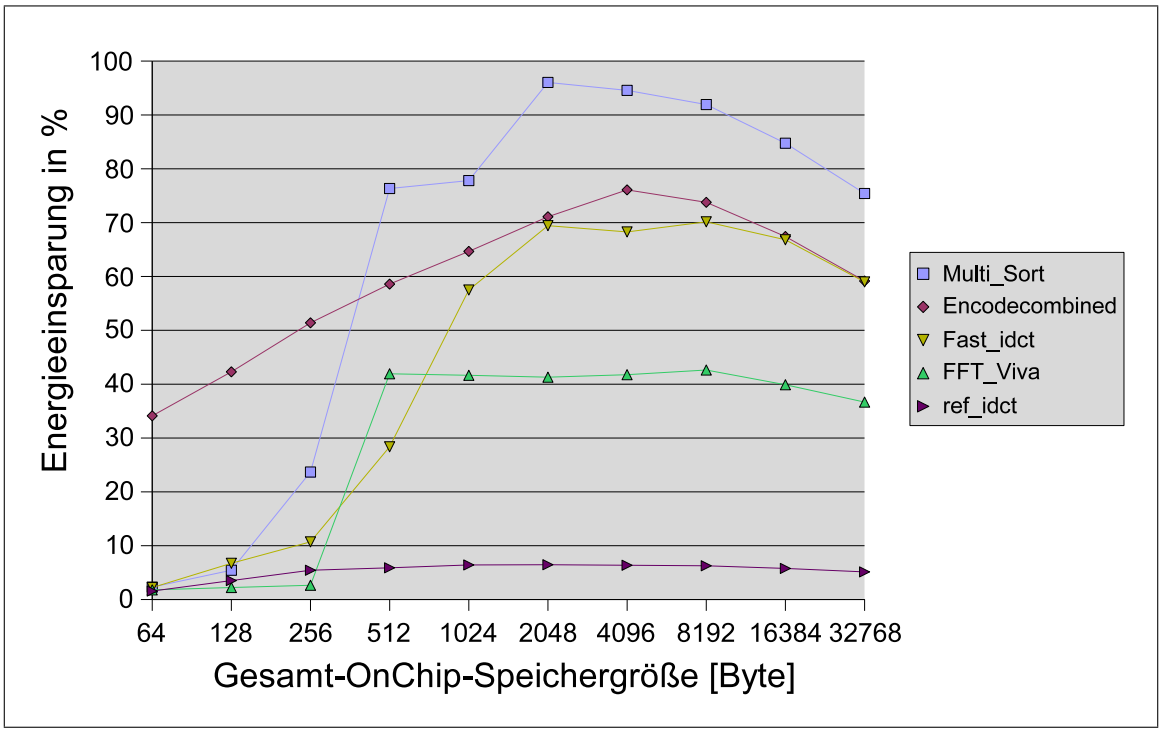
Um die Ergebnisse der Compiler-Optimierungsmethode interpretieren und einschätzen zu können, ist in der Tabelle 7.1 der jeweilige Energieverbrauch der ausgewählten Programme aufgeführt, da alle Ergebnisdarstellungen die erzielten Energieeinsparungen in Prozent der benötigen Speicherenergie angeben. In den Abbildungen 7.1(a)- 7.1(c) sind die möglichen Einsparungen dargestellt, die sich durch den Einsatz eines einzelnen Scratchpad-Speichers erreichen lassen. Die genauen Werte können der Tabelle A.9 im Anhang entnommen werden. Die Abbildungen zeigen die möglichen Energieeinsparungen der fünf ausgewählten Programme in Abhängigkeit der ausgewählten Scratchpad-Speichergröße. Wie aus den Abbildungen hervorgeht, steigt die mögliche Einsparung mit der zur Verfügung stehenden Scratchpad-Speichergröße an. Sobald das gesamte Programm in den Scratchpad-Speicher passt, sinkt die erzielte Energieeinsparung durch den zu

groß dimensionierten Scratchpad-Speicher wieder ab. Des weiteren kann man bei einem Vergleich der Abbildungen Unregelmäßigkeiten feststellen, die zum einen durch ungenaue Häufigkeitsanalysen verursacht werden. Es ist deutlich zu erkennen, dass die statische Analyse im Vergleich zum dynamischen Profiling vor allem bei dem Programm „Multi_sort“ schlechtere Ergebnisse erzeugt. Zum anderen können, wie aus der Abbildung 7.1(c) im Vergleich zu den beiden anderen Abbildungen hervorgeht, durch die Beschränkung der Basis-Blöcke auf eine Maximalgröße tendenziell bessere Ergebnisse erzielt werden. Diese Phänomen tritt vor allem bei kleinen Partitionsgrößen auf und ist sehr deutlich bei dem Programm „FFT_Viva“ (Speichergrößen bis 256 Byte) zu erkennen.

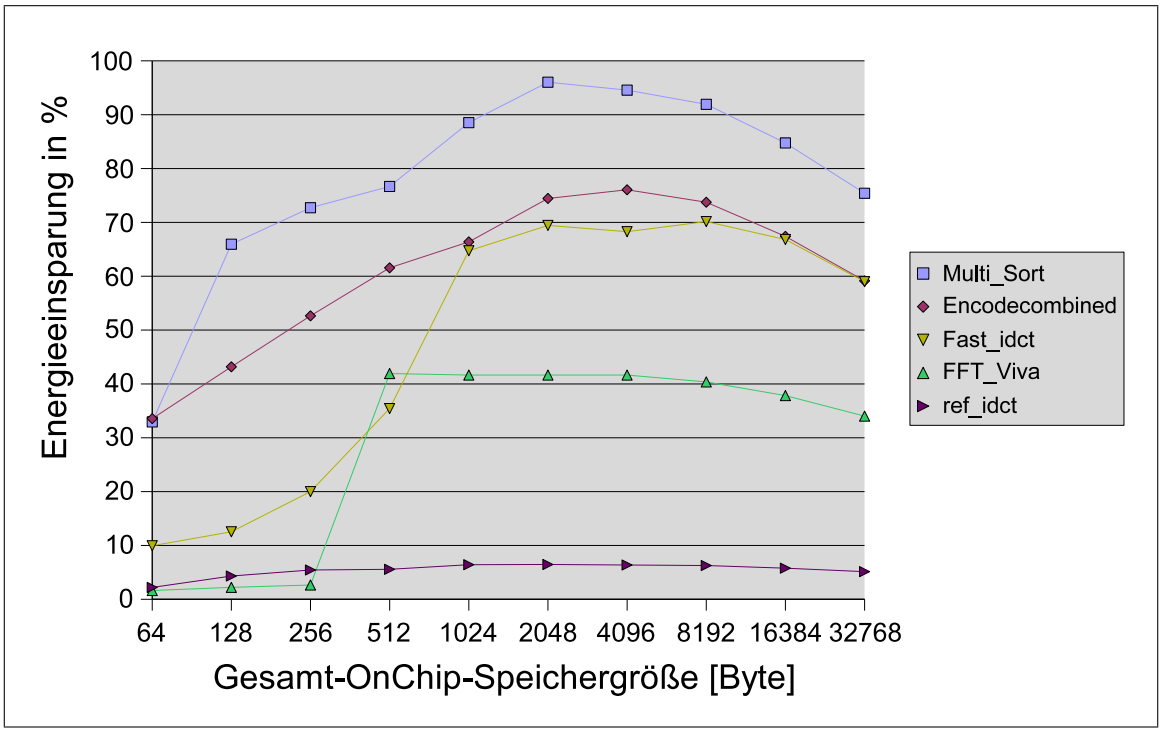
Programm	CPU	Speicher	Summe
Multi_Sort	2743	6035	8778
Encodedcombined	9114	20704	29818
Fast_idct	2166	4999	7165
FFT_Viva	9122	19990	29112
ref_idct	181367	470694	652061
mpeg2dec	25163	54892	80055

Tabelle 7.1: Der Energieverbrauch der ausgewählten Programme [μJ]

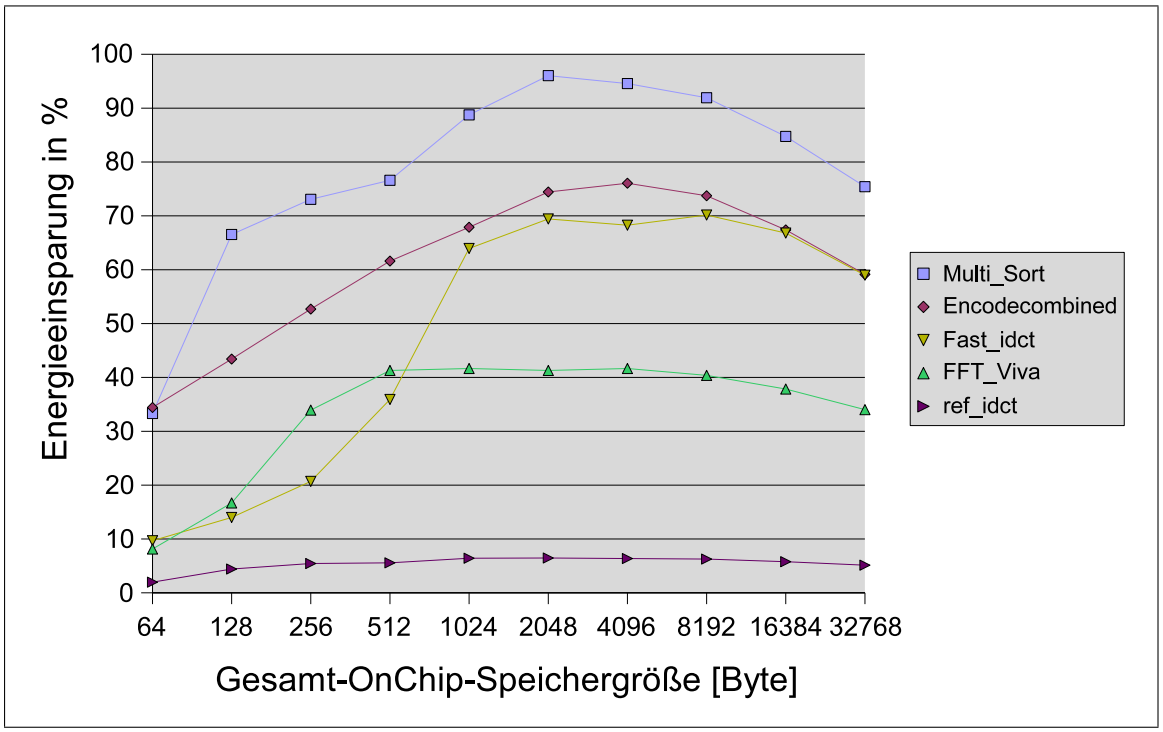
Wie aus den Abbildungen 7.1(a)- 7.1(c) ebenfalls hervorgeht, ist die während dieser Arbeit entwickelte Compiler-Optimierungsmethode in der Lage, ähnlich gute Energieeinsparungen wie die Compiler-Optimierungsmethode der Diplomarbeit [Zob01] zu erzielen. Das Ziele der neuen Compiler-Optimierungsmethode muss es demnach sein, weitere Energieeinsparungen durch eine Speicherpartitionierung zu erreichen. Im folgenden sollen die so erzielten Ergebnisse erläutert werden. Für die Interpretation der Ergebnisgrafiken ist allerdings die Tabelle 6.3 notwendig, da in den Grafiken lediglich die Anzahl der Partitionen angegeben ist. Die genaue Aufteilung, wie sich eine solche Partitionierung zusammensetzt, ist der Tabelle zu entnehmen.



(a) Statische Analyse



(b) Dynamisches Profiling



(c) Dynamisches Profiling mit kleinen Basis-Blöcken

Abbildung 7.1: Die Energieeinsparung in % (bei einem einzelnen Scratchpad-Speicher)

Die Ergebnisse des Programms „Encodecombined“ dienen der exemplarischen Vorstellung der Ergebnisse. Aus Übersichtsgründen werden in diesem Kapitel jedoch nur die aussagekräftigsten Daten in Diagrammform dargestellt. Die Tabellen mit den Simulationsergebnissen der fünf ausgewählten Programme sind im Anhang dargestellt. Hierzu noch einige Anmerkungen zum besseren Verständnis:

Jedes Programm wurde mit allen möglichen Partitionierungen, die in der Tabelle 6.3 dargestellt sind, simuliert. Des weiteren wurden diese Simulationen drei mal durchgeführt; zum einen mit der statischen Häufigkeitsanalyse und dem dynamischen Profiling. Als weitere Möglichkeit wurde das dynamische Profiling mit der Erweiterung, die die Basis-Blöcke in kleine Basis-Blöcke mit einer maximalen Größe von 6 Byte zerlegt, simuliert. Das heißt, dass im Anhang pro ausgewertetem Programm drei Ergebnistabellen, die die Energieeinsparung des Programms darstellen, existieren. Diese Ergebnisse sind anschließend in zwei Diagrammen pro Tabelle dargestellt. Das Erste stellt die Ergebnisse, die in den Tabellen dargestellt

sind, ohne Veränderung grafisch dar. Das zweite Diagramm hingegen stellt die relative Energieeinsparung bezogen auf den einfachen Scratchpad-Fall der jeweiligen Gesamtspeichergröße dar. Es fungiert als Interpretationshilfe der erzielten Ergebnisse.

In der Abbildung 7.2 sind die Simulationsergebnisse des Programms „Encodecombined“ dargestellt. Wie diesem zu entnehmen ist, steigt die Energieeinsparung für eine Speicherpartition bis zu einer Scratchpadspeichergröße von 4096 Byte auf 76,06% an. Diese Energieeinsparung kann bei größeren Scratchpad-Speichergrößen jedoch nicht aufrecht erhalten werden. Bei einem 32 KByte Scratchpad-Speicher beträgt die Energieeinsparung nur noch 59,14%. Durch die Speicherpartitionierung kann jedoch ab einer Speichergröße von 1024 Byte eine zusätzliche Energieeinsparung erreicht werden, die auch durch größere zur Verfügung stehende Onchip-Speicherkapazitäten nicht wieder aufgezehrt wird.

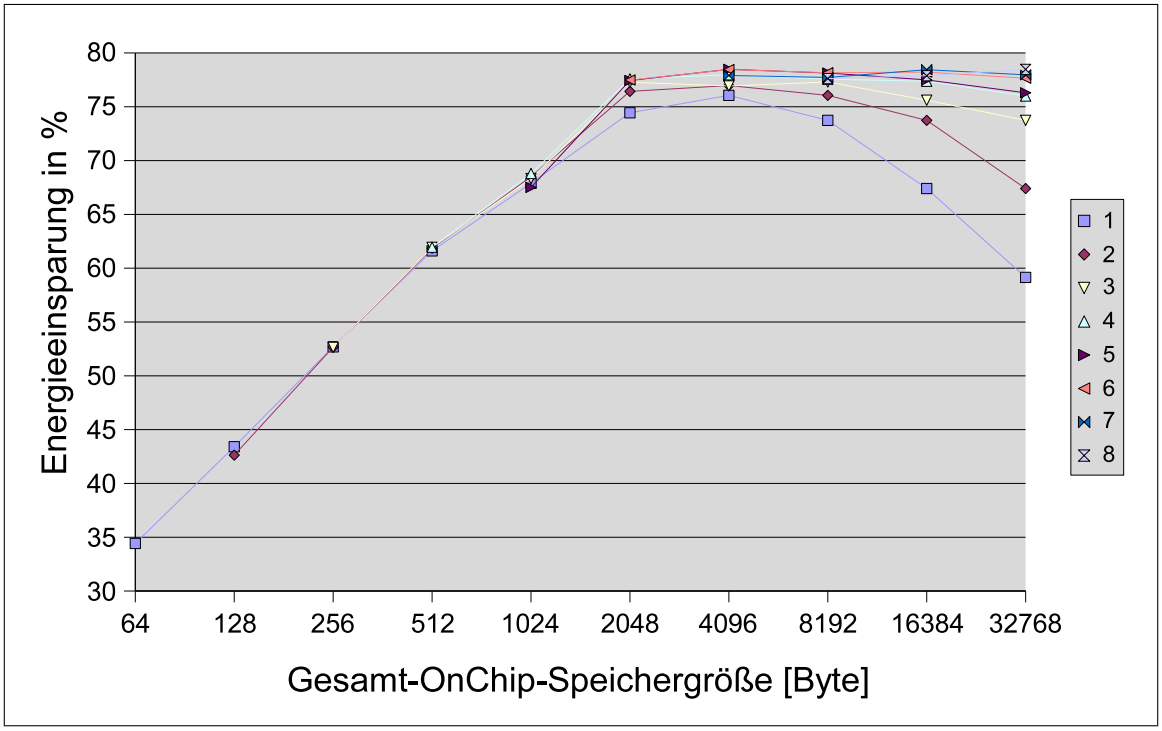


Abbildung 7.2: Die Energieeinsparung bei dem Programm „Encodecombined“ in % (Dynamisches Profiling mit kleinen Basis-Blöcken)

Bei einem Vergleich der im Anhang dargestellten Ergebnisse fällt auf, dass bei allen Programmen tendenziell ähnliche Ergebnisse erzielt wurden. Je nach Programm

und verwendeter Häufigkeitsanalyse können einzelne Ergebnisse jedoch erheblich nach unten abweichen. Dies lässt sich darauf zurückführen, dass große Basis-Blöcke oder Variablen nicht in kleine Speicher verschoben werden können, da sie größer sind als die zur Verfügung stehende Speicherkapazität. Diese Ausreißer können durch die Erweiterung, die große Basis-Blöcke in mehrere kleinere Basis-Blöcke zerlegt, weitgehend eliminiert werden.

Für einen Vergleich der neuen Ergebnisse mit denen, die sich durch den Einsatz eines einzelnen Scratchpad-Speichers erreichen lassen, ist in den Abbildungen 7.3-7.5 die maximale Energieeinsparung, die während der Simulationen erreicht worden ist, dargestellt. Die jeweils erste Abbildung zeigt die möglichen Energieeinsparungen der fünf ausgewählten Programme in Abhängigkeit der ausgewählten Onchip-Speichergröße. Die zugrundeliegenden Werte können der Tabelle A.10 im Anhang entnommen werden. Wie den Abbildungen zu entnehmen ist, können alle ausgewählten Programme von der entwickelten Compiler-Optimierungsmethode profitieren, wobei bei dem Programm „Multi_Sort“ die höchste Energieeinsparung von 97,7% erreicht wurde. Im Vergleich zu der Compiler-Optimierungsmethode der Diplomarbeit [Zob01] konnten durch die Speicherpartitionierung weitere Verbesserungen von bis zu 22,04% erreicht werden. Diese relativen Verbesserungen sind ebenfalls in den Abbildungen 7.3-7.5 dargestellt, wobei die zugrundeliegenden Werte der Tabelle A.11 im Anhang entnommen werden können.

Bei einer anderen Betrachtungsweise ist die mögliche Verbesserung gegenüber der ursprünglichen Optimierungsmethode der Diplomarbeit [Zob01] jedoch wesentlich geringer. Diese Betrachtungsweise vergleicht die beste Optimierung mit nur einem Scratchpad-Speicher mit der besten Optimierung, die mit mehreren Partitionen bei gleicher Speicherkapazität erreicht worden ist. Die Tabelle 7.2 stellt die in dieser Betrachtungsweise erzielten Verbesserungen dar. Wie der Tabelle zu entnehmen ist, kann vor allem das Programm „Fast_idct“ mit einer weiteren, sehr positiv zu bewertenden Energieeinsparung von 7% von der Speicherpartitionierung profitieren.

Für die Praxis bedeutet dies, dass die neue Optimierungmethode die gleichen sehr guten Ergebnisse wie die ursprüngliche Optimierungmethode [Zob01] erzielt. Diese Ergebnisse konnten sogar noch weiter verbessert werden. Da die neue Optimierungsmethode jedoch keine Nachteile gegenüber der ursprünglichen Optimierungs-

Programm	S	D	D6	Durchschnitt
Multi_Sort	1,51	1,66	1,59	1,59
Encodedcombined	2,13	2,46	2,42	2,34
Fast_idct	6,84	7,02	6,58	6,81
FFT_Viva	4,64	1,99	2,27	2,97
ref_idct	0,18	0,17	0,2	0,18

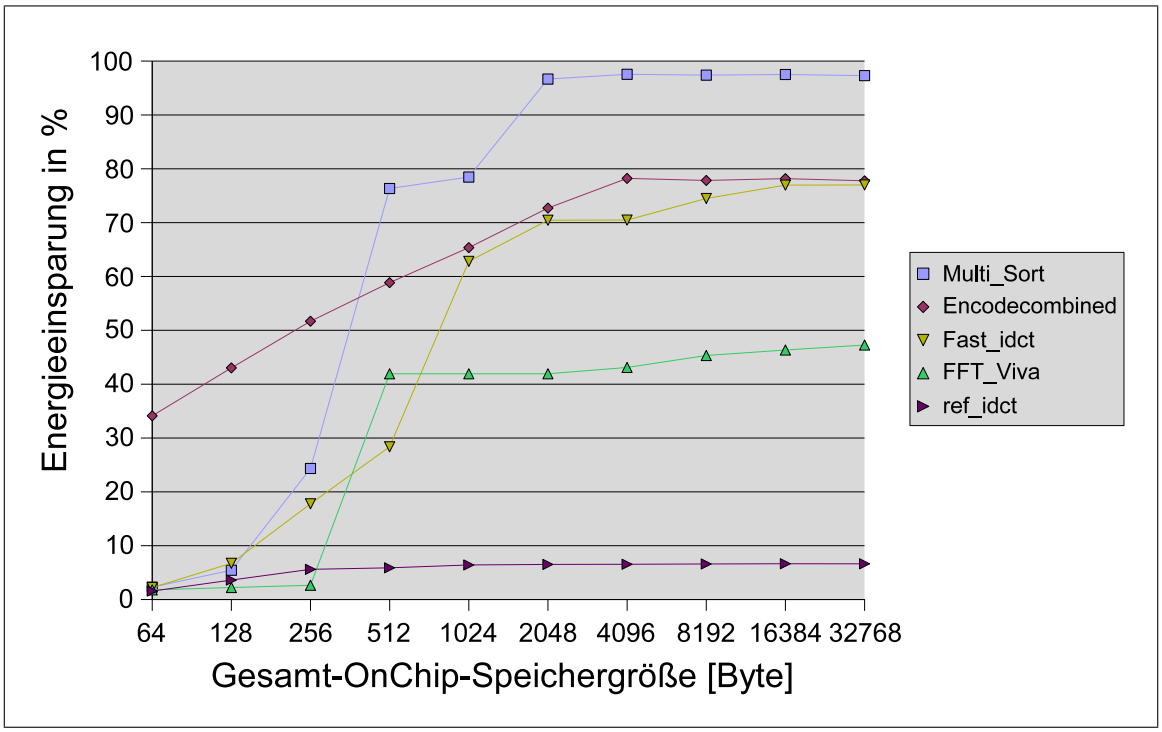
S = Statische Häufigkeitsanalyse; D = Dynamisches Profiling;
D6 = D mit kleinen Basis-Blöcken

Tabelle 7.2: Die maximale Energieeinsparung durch mehrere Partitionen in %

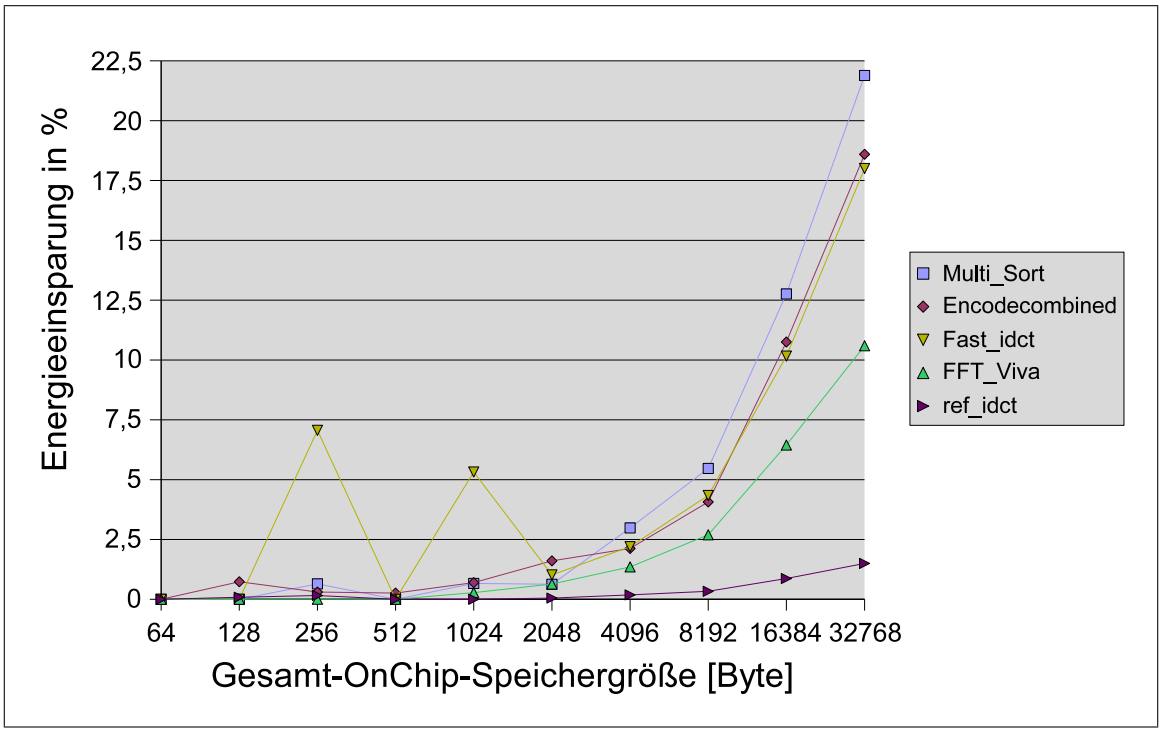
methode aufweist, ist sie in der Lage, diese vollständig abzulösen. Des weiteren kann die neue Optimierungmethode in der Praxis besser eingesetzt werden, da bei allgemeinen Prozessoren die für ein Programm optimale Scratchpad-Speichergröße normalerweise nicht zur Verfügung steht. Allgemeine Prozessoren könnten jedoch flexibel einsetzbare partitionierte Scratchpad-Speicher zur Verfügung stellen, die die im Zuge dieser Arbeit entwickelte Optimierungsmethode nutzen könnte. Dieses Vorgehen hätte den Vorteil, dass nahezu jedes Programm von einer Speicherpartitionierung profitieren könnte.

Um die bisherigen Ergebnisse weiter zu untermauern, wurde im Rahmen der Evaluierung der neu entwickelten Compiler-Optimierungsmethode ein weiteres Programm auf das mögliche Energiesparpotential hin untersucht. Das Programm „mpeg2dec“, dessen Programmeigenschaften in der Tabelle A.8 im Anhang dargestellt sind, implementiert ebenfalls einen typischen Anwendungsfall im Bereich der *ES*. Im Vergleich zu den anderen fünf ausgewählten Programmen benötigt das Programme „mpeg2dec“ jedoch mehr Speicher. Des weiteren besteht es aus wesentlich mehr Programmobjekten, die für die Verteilung auf unterschiedliche Partitionen in Frage kommen. Da es sich bei dem Compiler *enCC* jedoch um einen Forschungscompiler handelt, konnten leider nicht alle gewünschten Simulationsläufe abgeschlossen werden. Die erzielten Teilergebnisse sind in der Tabelle A.7 sowie in der Abbildung A.16 im Anhang dieser Arbeit dargestellt. Wie den Darstellungen zu entnehmen ist, zeigen die erzielten Energieeinsparungen die selben Tendenzen, die auch bei den anderen Programmen beobachtet werden konnten. Die Anzahl der zu beobachtenden Ausreißer ist bei Programm „mpeg2dec“ jedoch größer. Dies Phänomen hängt sehr wahrscheinlich mit der Größe der Program-

mobjekte zusammen und kann demnach mit der im Kapitel 4.3.1 beschriebenen Methode beseitigt werden.

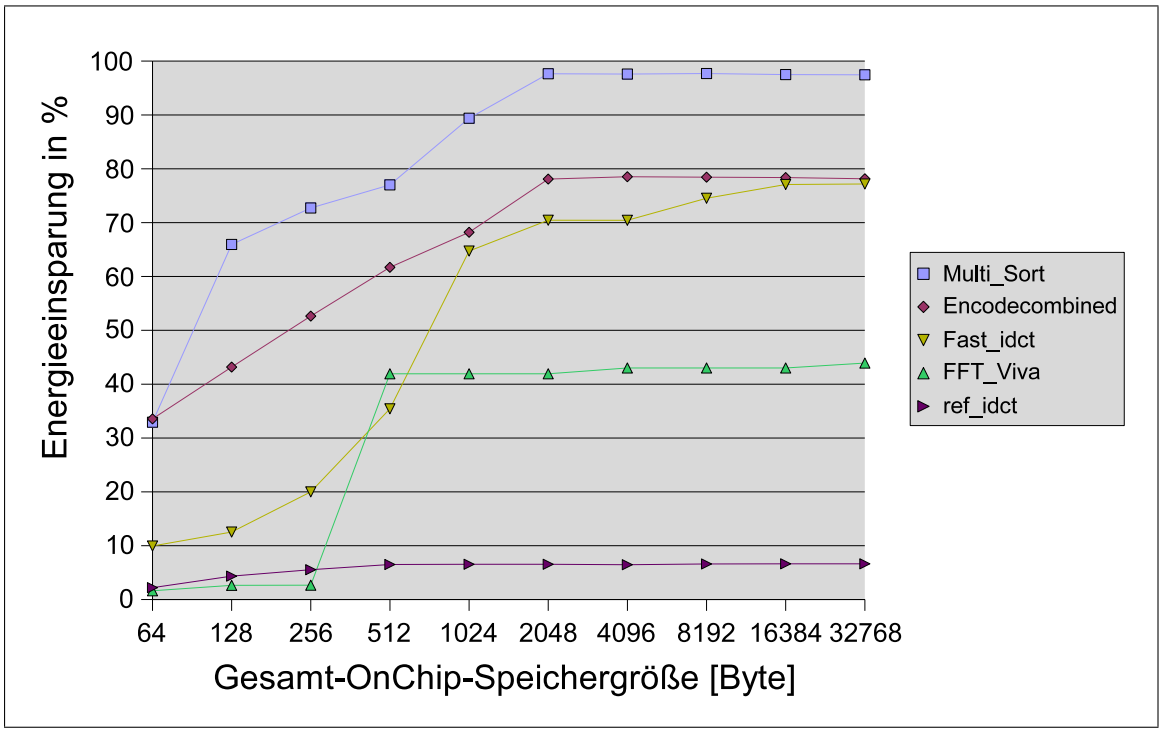


(a) Die maximal erreichte Energieeinsparung in %

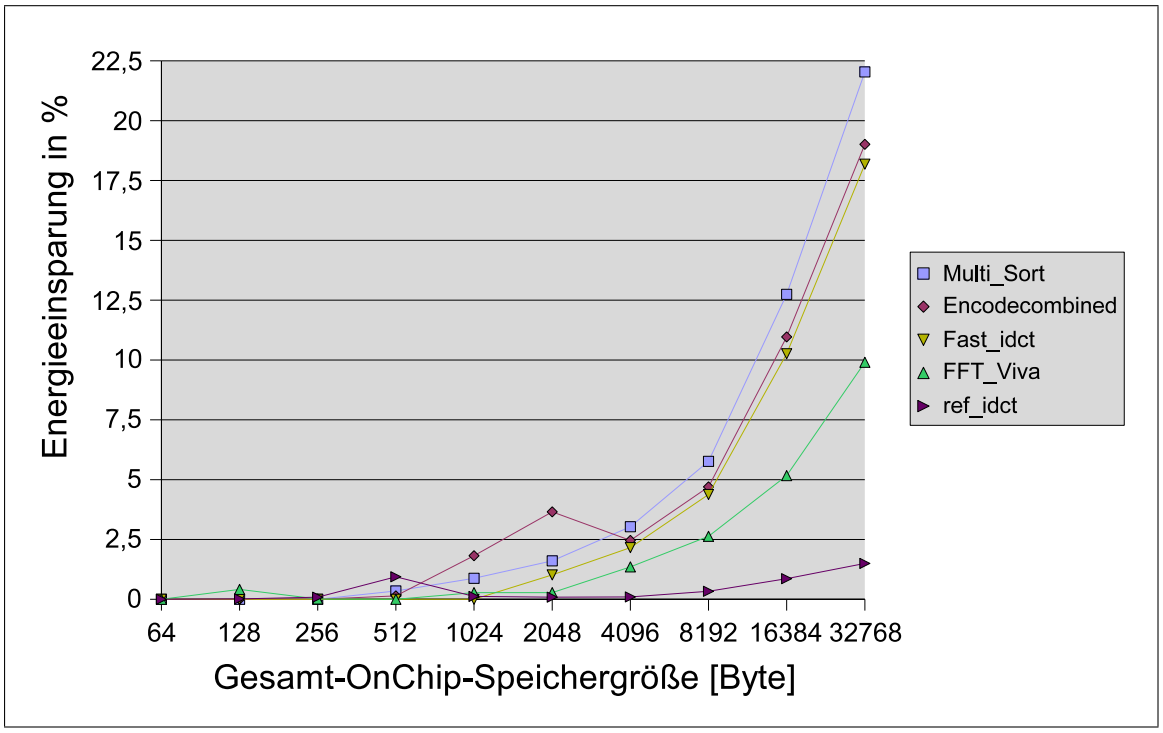


(b) Die relative Energieeinsparung in %

Abbildung 7.3: Gesamtergebnis (Statische Analyse)

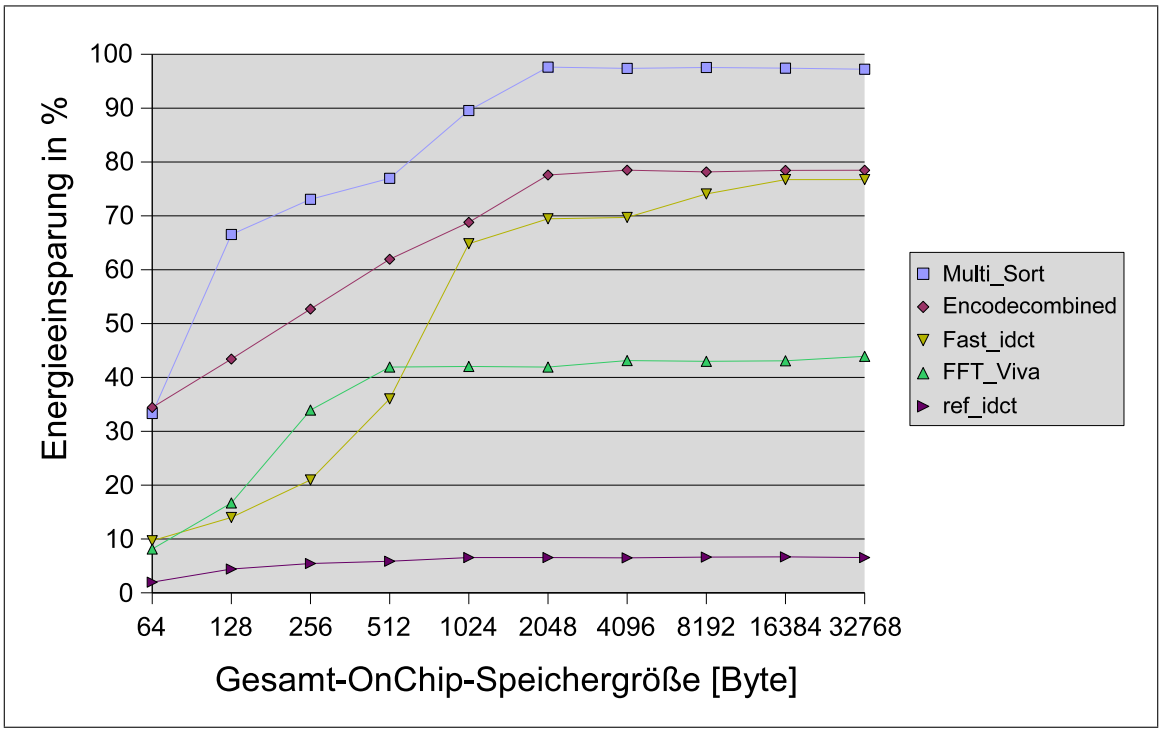


(a) Die maximal erreichte Energieeinsparung in %

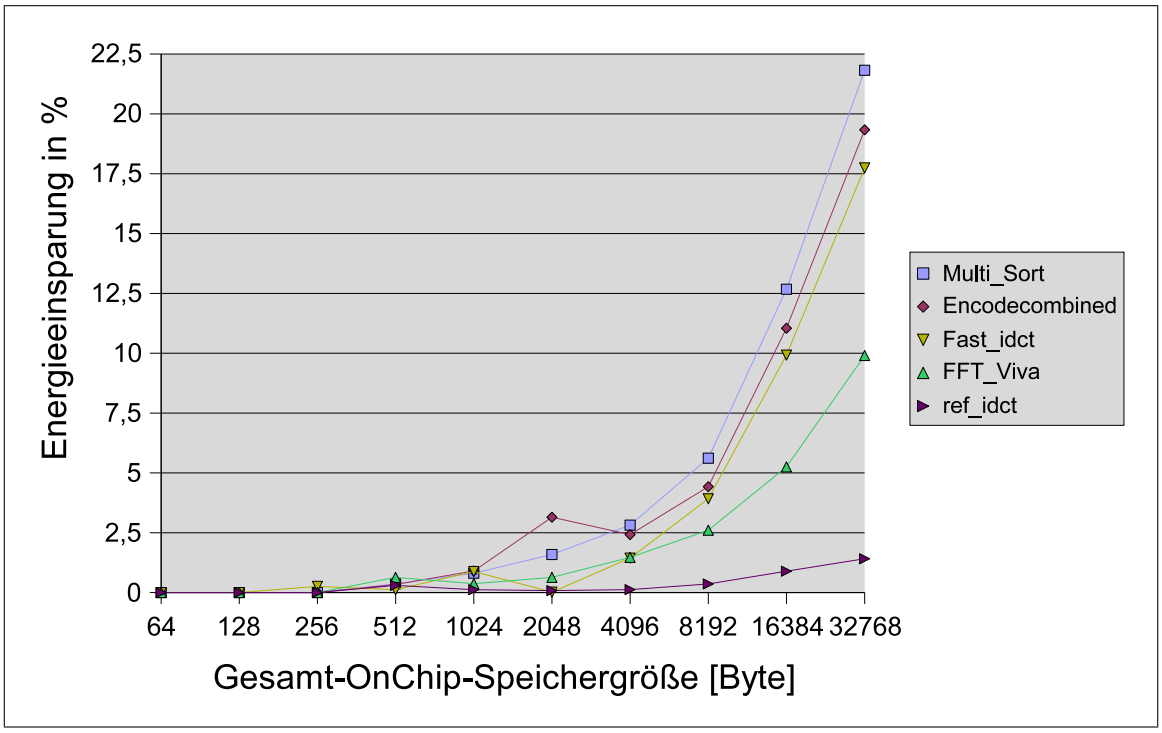


(b) Die relative Energieeinsparung in %

Abbildung 7.4: Gesamtergebnis (Dynamisches Profiling)



(a) Die maximal erreichte Energieeinsparung in %



(b) Die relative Energieeinsparung in %

Abbildung 7.5: Gesamtergebnis (Dynamisches Profiling mit kleinen Basis-Blöcken)

8 Fazit

Diese Arbeit hat gezeigt, dass die »Compilergestützte Optimierung von Zugriffen auf partitionierte Speicher« die erhofften Ergebnisse liefert. Es konnte gezeigt werden, dass die neue Optimierungsmethode Verbesserungen des Speicherenergieverbrauchs von bis zu 97% erreicht. Im Vergleich zu der Compiler-Optimierungsmethode der Diplomarbeit [Zob01] konnten durch die Speicherpartitionierung weitere Verbesserungen von bis zu 22,04% erreicht werden. Darüber hinaus hat die neue Compiler-Optimierungsmethode gegenüber der ursprünglichen Verfahrensweise keine Nachteile, da sie ebenfalls die Möglichkeit hat, einen einzelnen Scratchpad-Speicher (eine Partition) zu verwenden. Durch die Partitionierung ist sie jedoch in der Lage, die vorhandene Onchip-Speicherkapazität sehr gut auszunutzen.

Während der Ergebnisgenerierung hat sich gezeigt, dass die neue Optimierungsmethode in hohem Maße von guten Häufigkeitsanalysen (siehe Kapitel 4.3.2) abhängig ist. Diesbezüglich lassen sich die erzielten Ergebnisse noch weiter verbessern, da die Häufigkeitsanalysen des Forschungscompilers *enCC* keine idealen Ergebnisse liefern. Des weiteren konnten durch die Teilung der Basis-Blöcke in kleinere Einheiten (siehe Kapitel 4.3.1) deutlich bessere Ergebnisse erzielt werden, da die neu entwickelte Compiler-Optimierungsmethode durch dieses Verfahren auch kleinere Speicher besser nutzen kann.

Eine naheliegende Möglichkeit die vorliegenden Ergebnisse weiter zu verbessern, besteht darin, die großen Variablen (z.B. Array und Strings) ebenfalls in kleinere Einheiten zu teilen. Auf diesem Gebiet hat die Arbeit [VM03] bereits erste Ansätze geliefert. Für praxisnahe Anwendungen bedeutet dies, dass der Schlüssel für weitere Energieeinsparungen in der Kombination der bereits vorhandenen Methoden liegt. Hierin besteht noch weitreichendes und vielversprechendes Forschungspotential, dessen Ergebnisse von der Wirtschaft dringend benötigt werden.

A Tabellen und Diagramme

Gesamtgröße (Byte)	Anzahl ausgewählter Partitionen der Größe:				
	1024	512	256	128	64
64	0	0	0	0	1
128	0	0	0	0	2
	0	0	0	1	0
256	0	0	0	0	4
	0	0	0	1	2
	0	0	0	2	0
	0	0	1	0	0
512	0	0	0	0	8
	0	0	0	1	6
	0	0	0	2	4
	0	0	0	3	2
	0	0	0	4	0
	0	0	1	0	4
	0	0	1	1	2
	0	0	1	2	0
	0	0	2	0	0
	0	1	0	0	0
1024	0	0	0	0	16
	0	0	0	1	14
	0	0	0	2	12
	0	0	0	3	10
	0	0	0	4	8
	0	0	0	5	6
	0	0	0	6	4
	0	0	0	7	2
	0	0	0	8	0
	0	0	1	0	12

Gesamtgröße (Byte)	Anzahl ausgewählter Partitionen der Größe:				
	1024	512	256	128	64
1024	0	0	1	1	10
	0	0	1	2	8
	0	0	1	3	6
	0	0	1	4	4
	0	0	1	5	2
	0	0	1	6	0
	0	0	2	0	8
	0	0	2	1	6
	0	0	2	2	4
	0	0	2	3	2
	0	0	2	4	0
	0	0	3	0	4
	0	0	3	1	2
	0	0	3	2	0
	0	0	4	0	0
	0	1	0	0	8
	0	1	0	1	6
	0	1	0	2	4
	0	1	0	3	2
	0	1	0	4	0
	0	1	1	0	4
	0	1	1	1	2
	0	1	1	2	0
	0	1	2	0	0
	0	2	0	0	0
	1	0	0	0	0

Tabelle A.1: Alle Kombinationsmöglichkeiten der Speicherpartitionierung im Zweierpotenzbereich von 64 Byte bis 1 KByte

Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	2,32							
128	5,45	4,24						
256	23,7	15,18	24,34					
512	76,34	76,22	76,3	75,77				
1024	77,81	78,31	78,48	78,44	78,36			
2048	96,04	96,67	85,75	86,03	93,65	85,97		
4096	94,57	96,04	96,82	97,08	97,4	97,55	97,38	
8192	91,93	94,57	95,96	96,82	97,15	97,4	97,4	97,27
16384	84,76	91,93	94,57	96,04	96,8	97,13	97,51	97,18
32768	75,41	84,76	91,93	94,57	95,86	96,8	97,2	97,3

(a) Statische Analyse

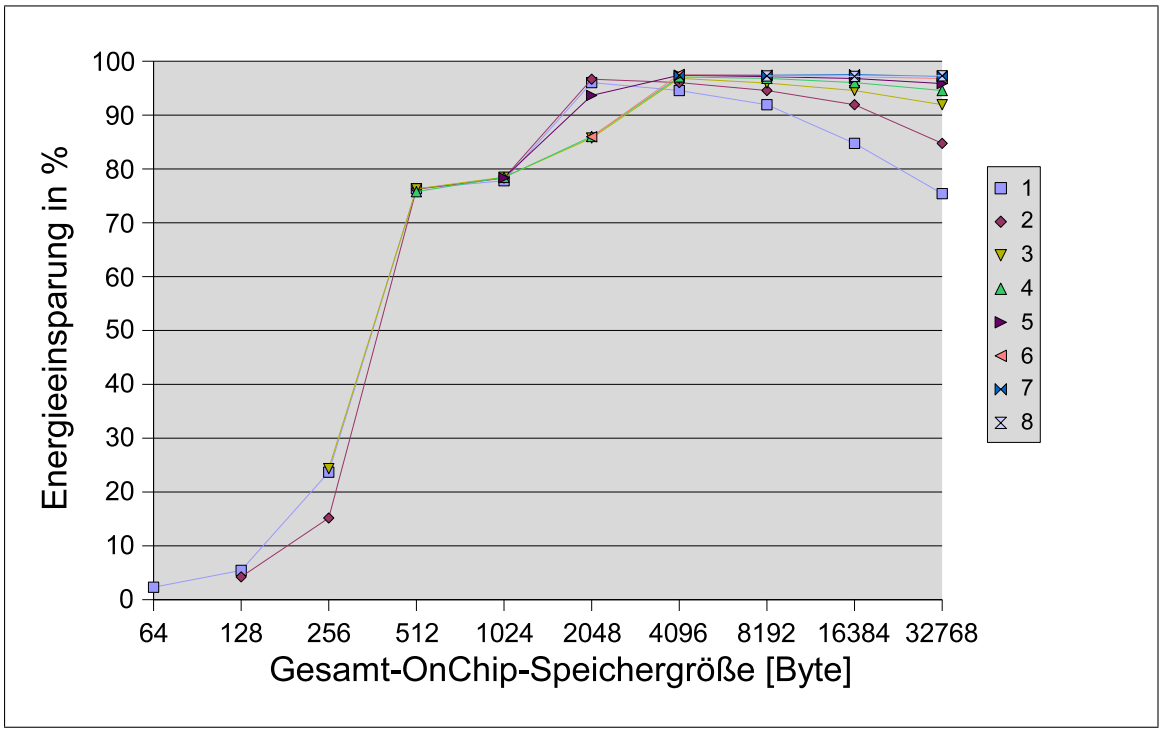
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	32,94							
128	65,95	64,42						
256	72,73	72,59	70,87					
512	76,67	77,02	76,8	74,58				
1024	88,52	89,23	89,4	87,8	87,13			
2048	96,04	96,74	97,2	97,63	97,65	95,92		
4096	94,57	95,96	96,65	97,32	97,4	97,6	95,74	
8192	91,93	94,57	95,96	96,69	97,2	97,05	97,51	97,7
16384	84,76	91,93	94,57	96,04	96,65	97,2	97,27	97,5
32768	75,41	84,76	91,93	94,57	95,96	96,82	97,2	97,45

(b) Dynamisches Profiling

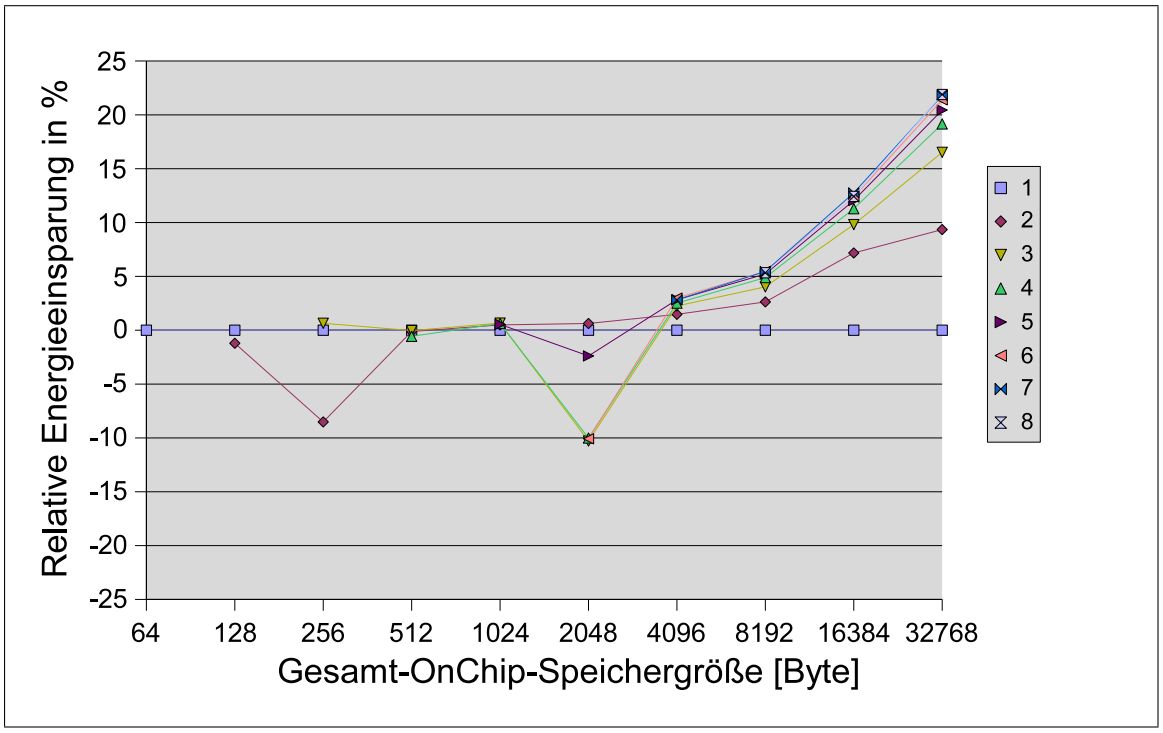
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	33,31							
128	66,55	64,74						
256	73,07	73,04	71,35					
512	76,6	76,97	76,87	74,83				
1024	88,75	89,25	89,56	88,98	87,56			
2048	96,04	96,57	97,35	97,46	97,63	95,82		
4096	94,57	96,01	96,65	97,28	97,38	97,33	95,72	
8192	91,93	94,57	95,82	96,62	97,3	97,18	97,55	95,97
16384	84,76	91,93	94,57	96,02	96,79	97,23	97,43	97,28
32768	75,41	84,76	91,93	94,57	96,02	96,67	97,23	97,1

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.2: Die Energieeinsparung bei dem Programm „Multi_Sort“ in %

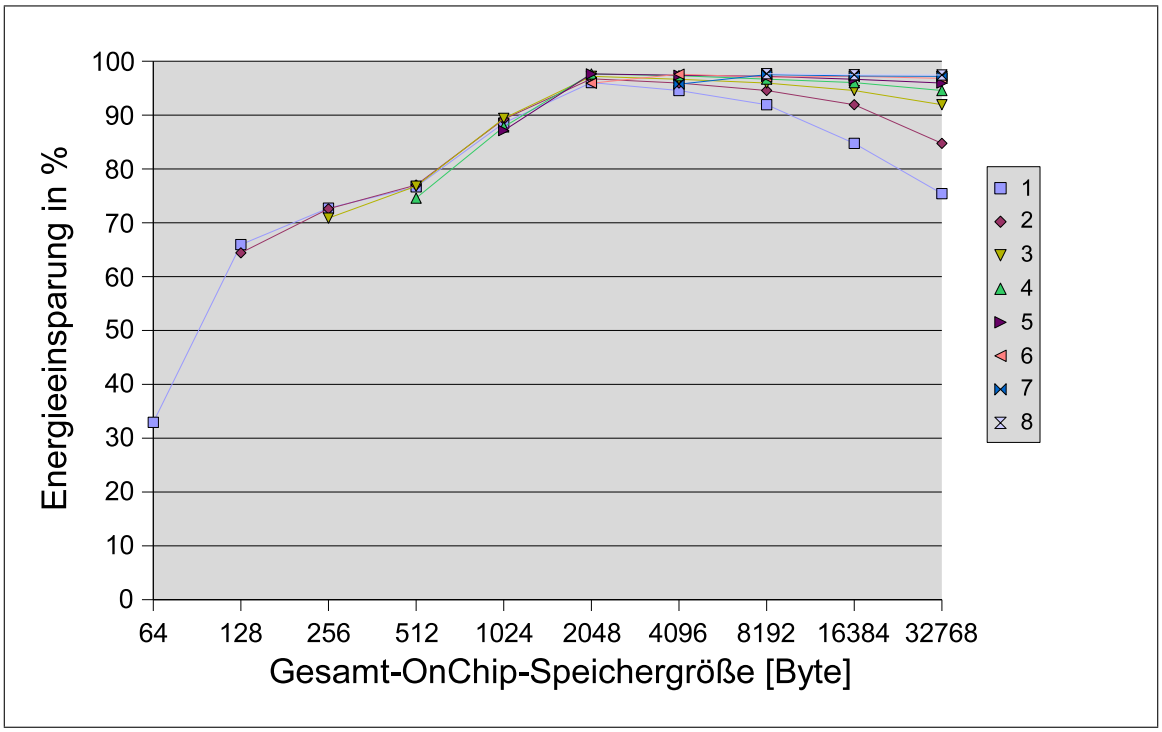


(a) Die Energieeinsparung in %

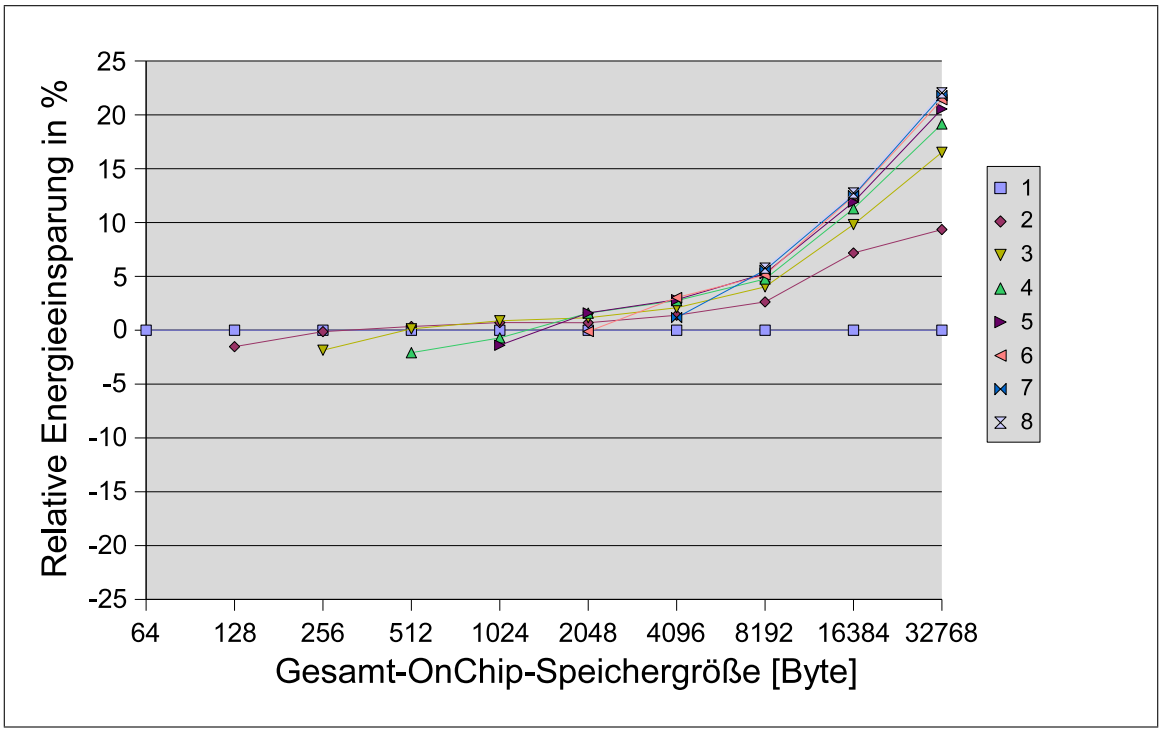


(b) Die relative Energieeinsparung in %

Abbildung A.1: Multi_Sort (Statische Analyse)

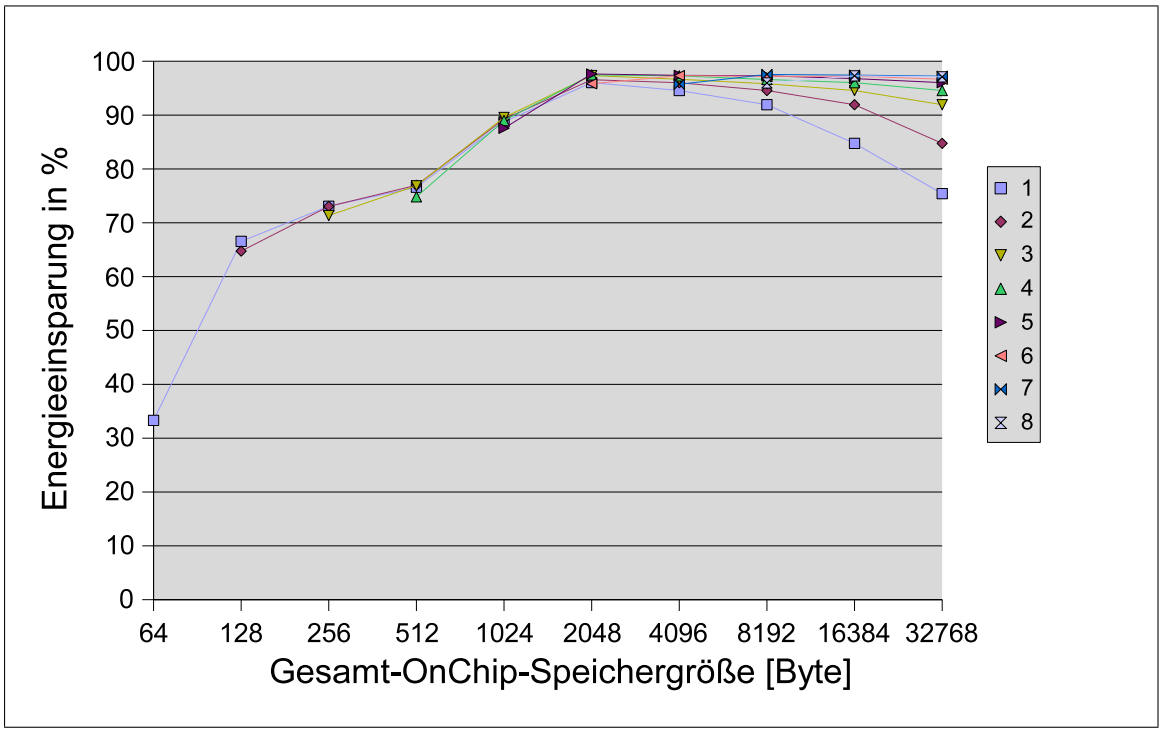


(a) Die Energieeinsparung in %

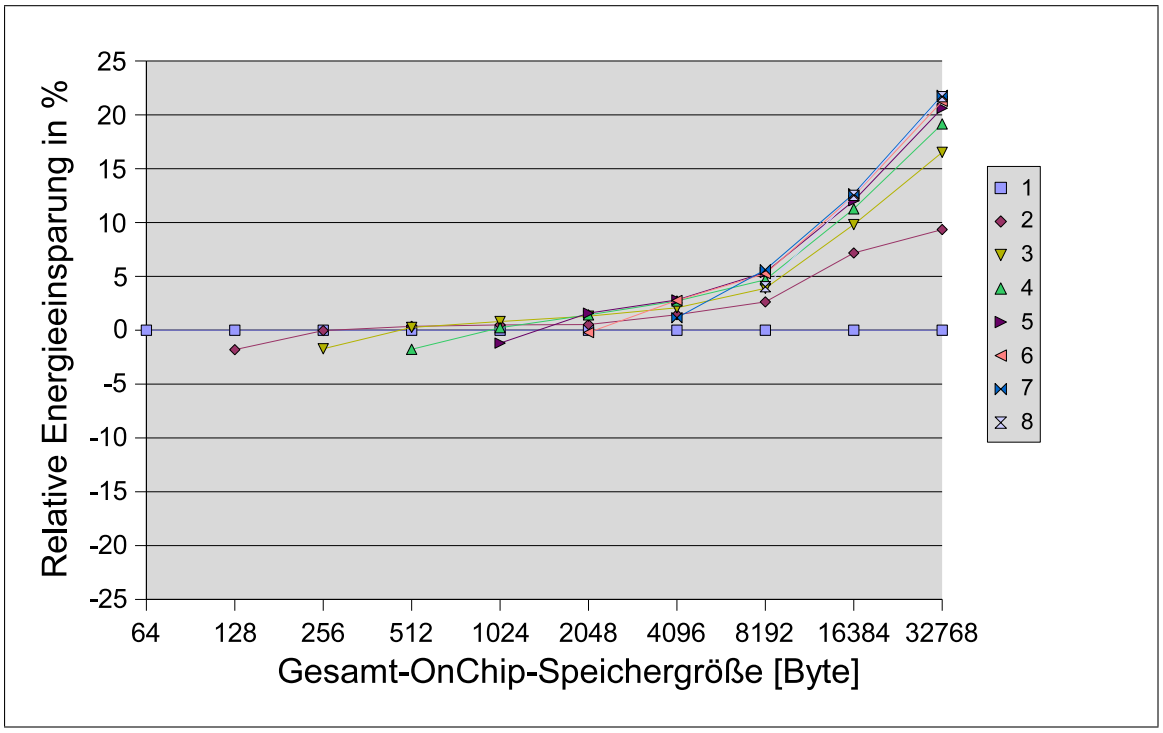


(b) Die relative Energieeinsparung in %

Abbildung A.2: Multi_Sort (Dynamisches Profiling)



(a) Die Energieeinsparung in %



(b) Die relative Energieeinsparung in %

Abbildung A.3: Multi_Sort (Dynamisches Profiling mit kleinen Basis-Blöcken)

Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	34,12							
128	42,3	43,03						
256	51,39	51,7	50,83					
512	58,59	58,85	57,93	58,23				
1024	64,66	64,89	65,36	64,34	64,71			
2048	71,09	71,32	72,02	72,7	71,53	70,49		
4096	76,09	77,37	77,48	74,35	77,94	77,83	78,21	
8192	73,77	76,09	77,36	77,84	77,75	77,48	77,41	77,72
16384	67,43	73,77	76,08	77,41	77,35	72,85	77,72	78,18
32768	59,17	67,43	73,77	76,1	77,37	77,54	77,76	77,4

(a) Statische Analyse

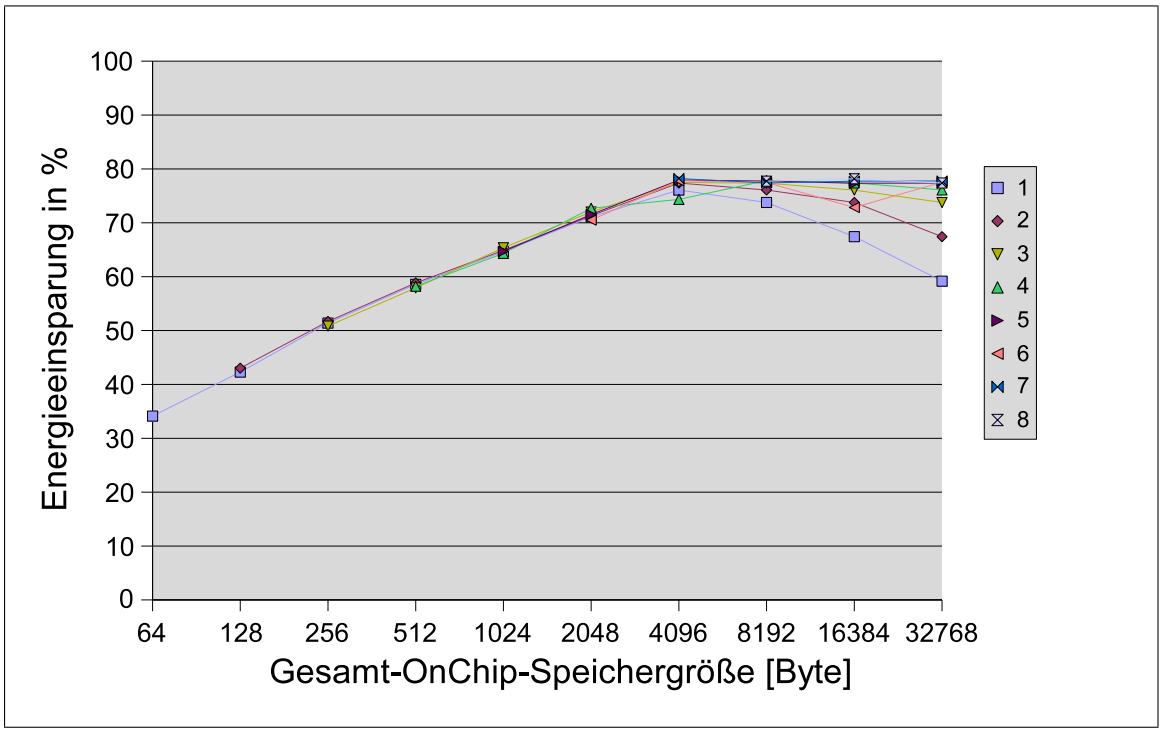
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	33,56							
128	43,18	40						
256	52,64	52,33	52,22					
512	61,56	61,57	61,7	61,35				
1024	66,38	68,04	67,95	68,2	68,18			
2048	74,44	75,66	77,69	78,1	77,13	77,82		
4096	76,06	77,3	77,82	78,33	78,52	77,84	77,54	
8192	73,74	76,06	77,32	77,89	78,01	78,21	78,44	77,84
16384	67,4	73,74	76,07	77,38	77,52	78,37	78,2	78,26
32768	59,14	67,4	73,74	76,07	77,37	77,3	77,9	78,15

(b) Dynamisches Profiling

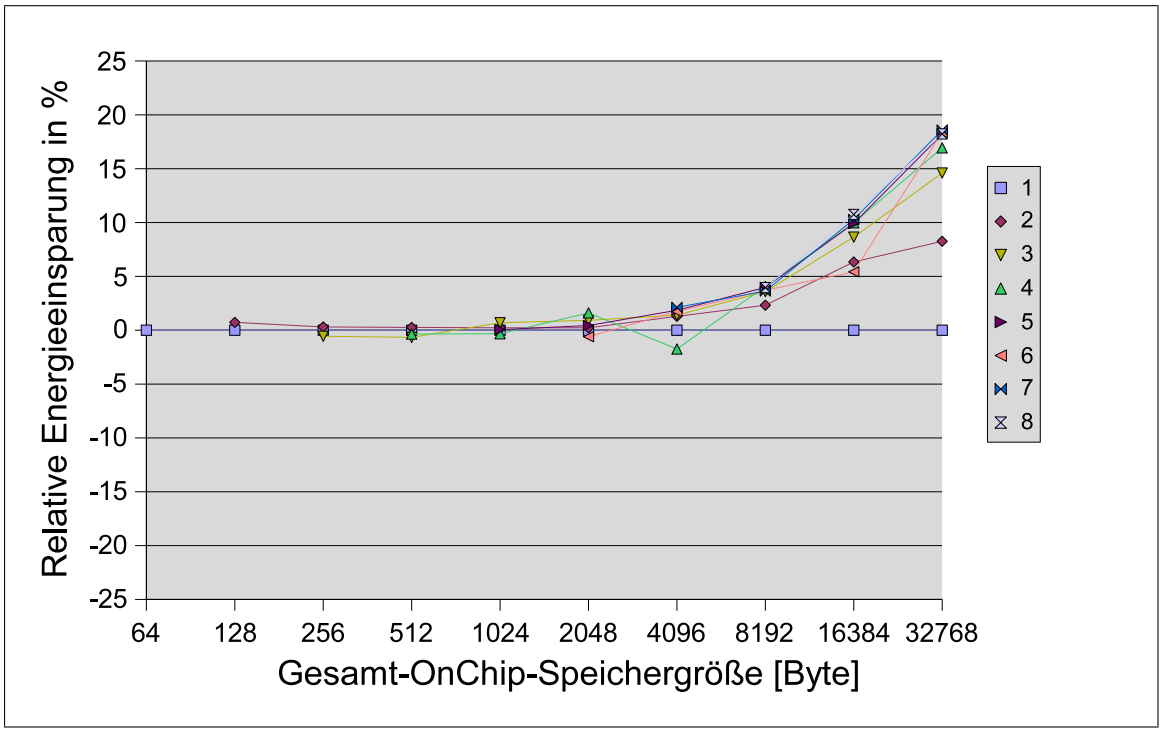
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	34,42							
128	43,41	42,63						
256	52,69	52,67	52,64					
512	61,61	61,86	61,94	61,95				
1024	67,9	68,5	68,33	68,8	67,51			
2048	74,44	76,43	77,28	77,59	77,44	77,49		
4096	76,06	76,98	76,94	77,98	78,48	78,46	77,91	
8192	73,74	76,06	77,3	77,56	78,14	78,16	77,72	77,62
16384	67,4	73,74	75,61	77,37	77,5	78,22	78,45	77,88
32768	59,14	67,4	73,74	75,99	76,3	77,65	77,97	78,48

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.3: Die Energieeinsparung bei dem Programm „Encodecombined“ in %

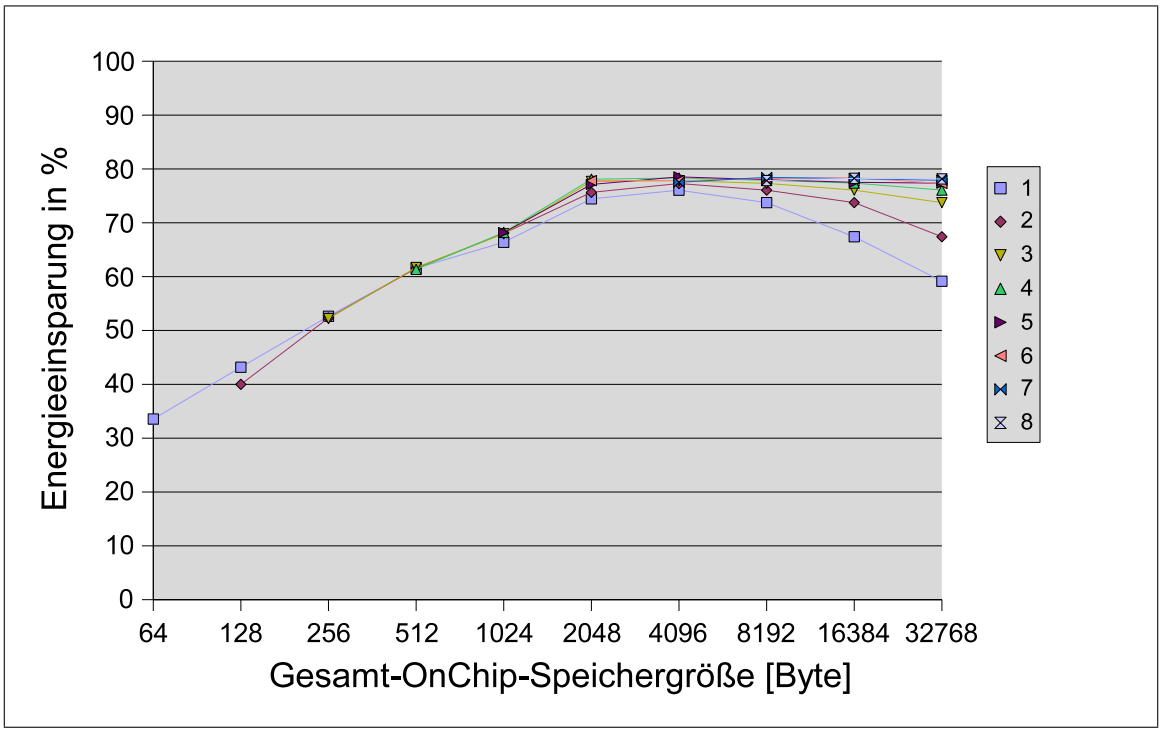


(a) Die Energieeinsparung in %

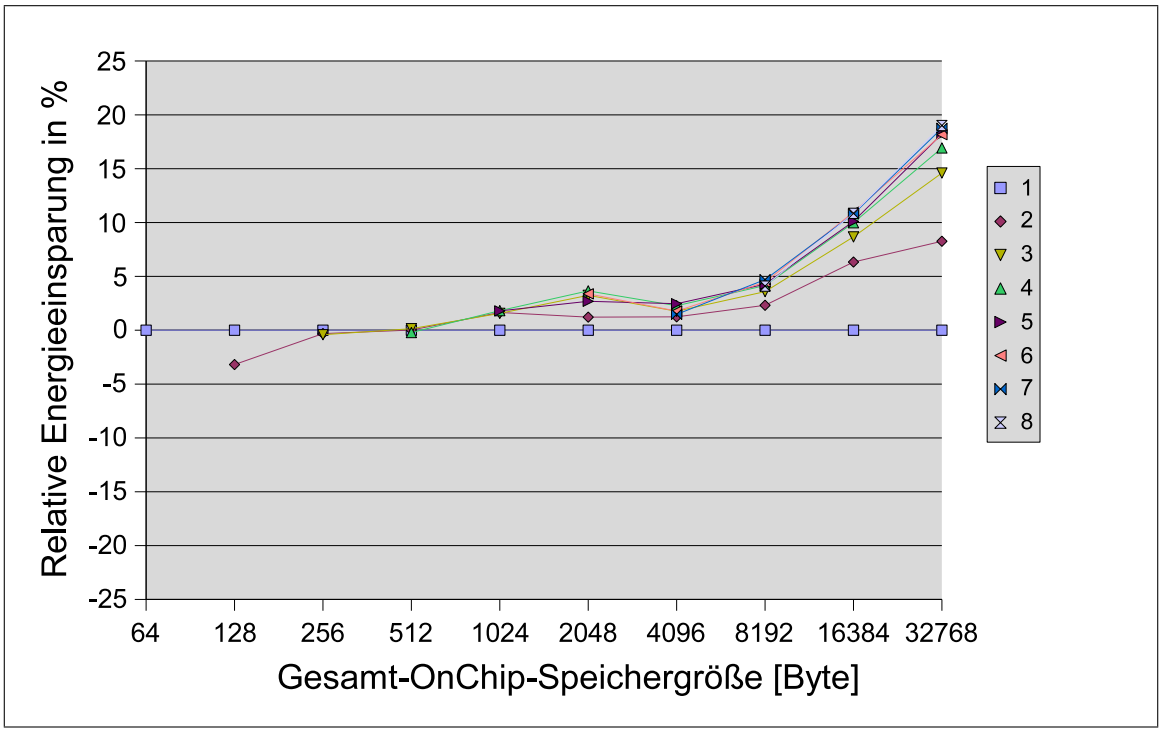


(b) Die relative Energieeinsparung in %

Abbildung A.4: Encodecombined (Statische Analyse)

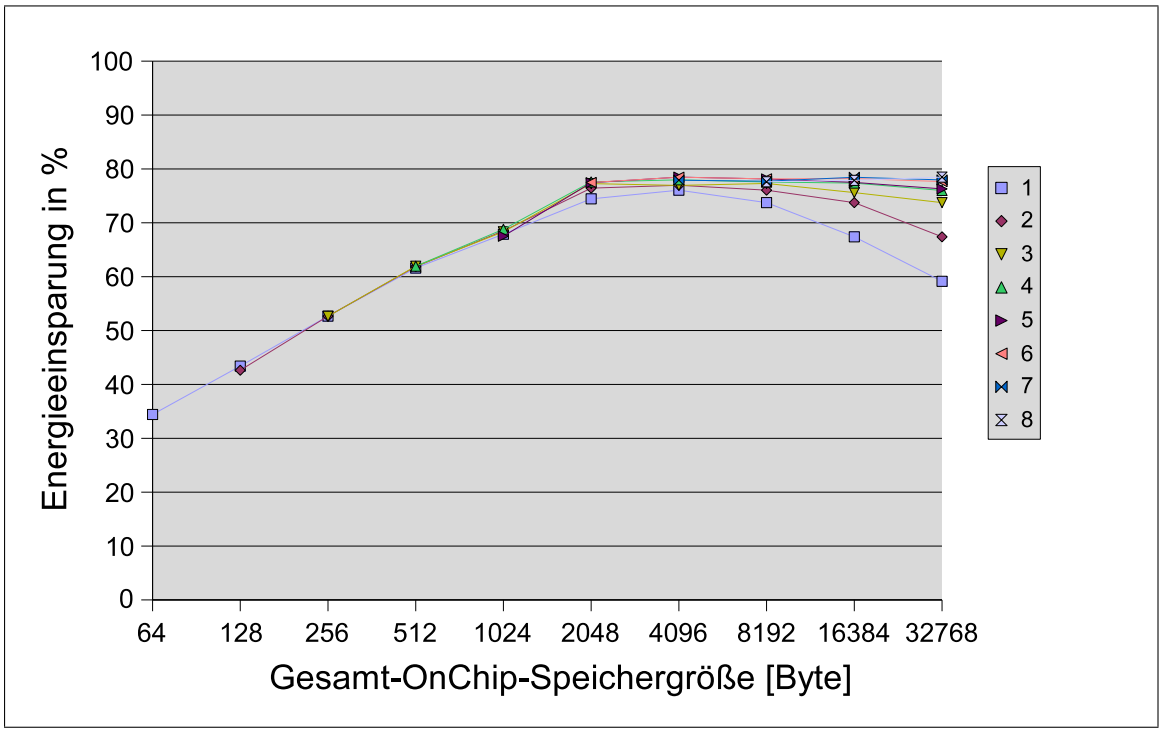


(a) Die Energieeinsparung in %

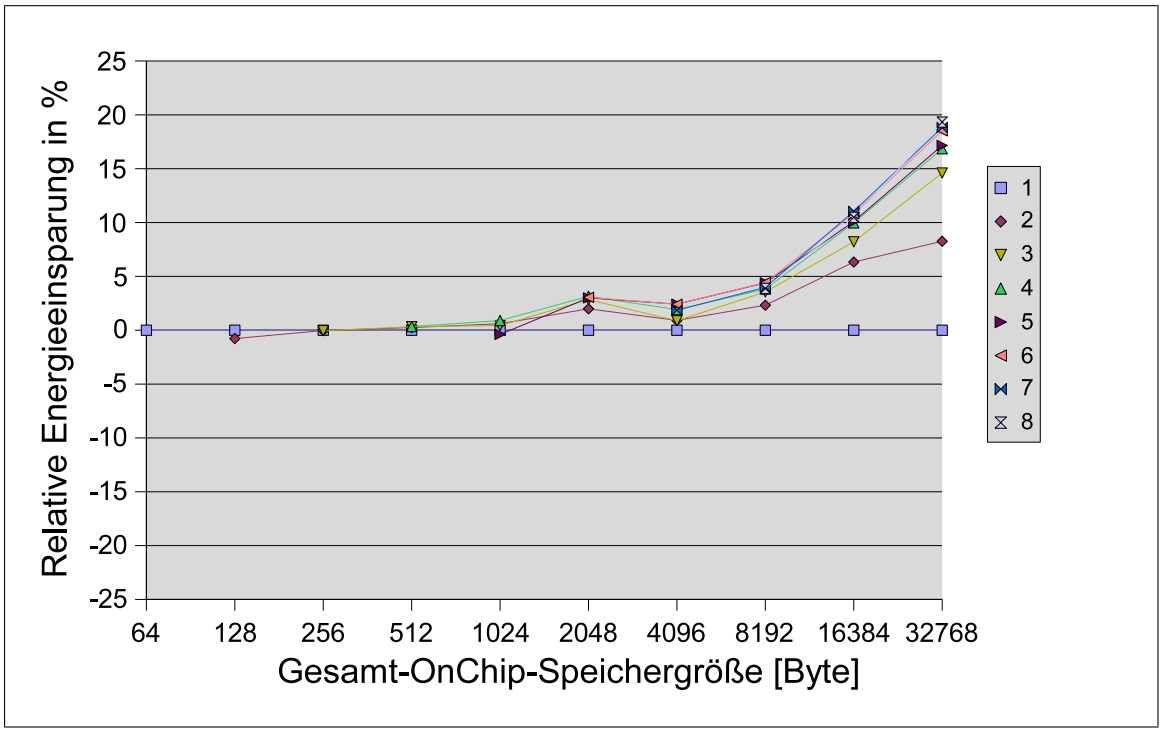


(b) Die relative Energieeinsparung in %

Abbildung A.5: Encodecombined (Dynamisches Profiling)



(a) Die Energieeinsparung in %



(b) Die relative Energieeinsparung in %

Abbildung A.6: Encodecombined (Dynamisches Profiling mit kleinen Basis-Blöcken)

Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	2,22							
128	6,74	3						
256	10,72	17,78	10,4					
512	28,37	19,52	17,66	17,34				
1024	57,49	62,81	52,03	43,05	52,19			
2048	69,43	70,03	68,15	70,29	69,77	70,45		
4096	68,27	69,43	70,03	68,87	70,45	70,23	70,47	
8192	70,15	72,31	73,47	74,03	74,23	73,91	74,49	73,99
16384	66,81	72,81	74,99	76,16	76,76	76,98	74,11	75,98
32768	58,99	66,81	72,81	74,51	76,16	76,76	77	76,94

(a) Statische Analyse

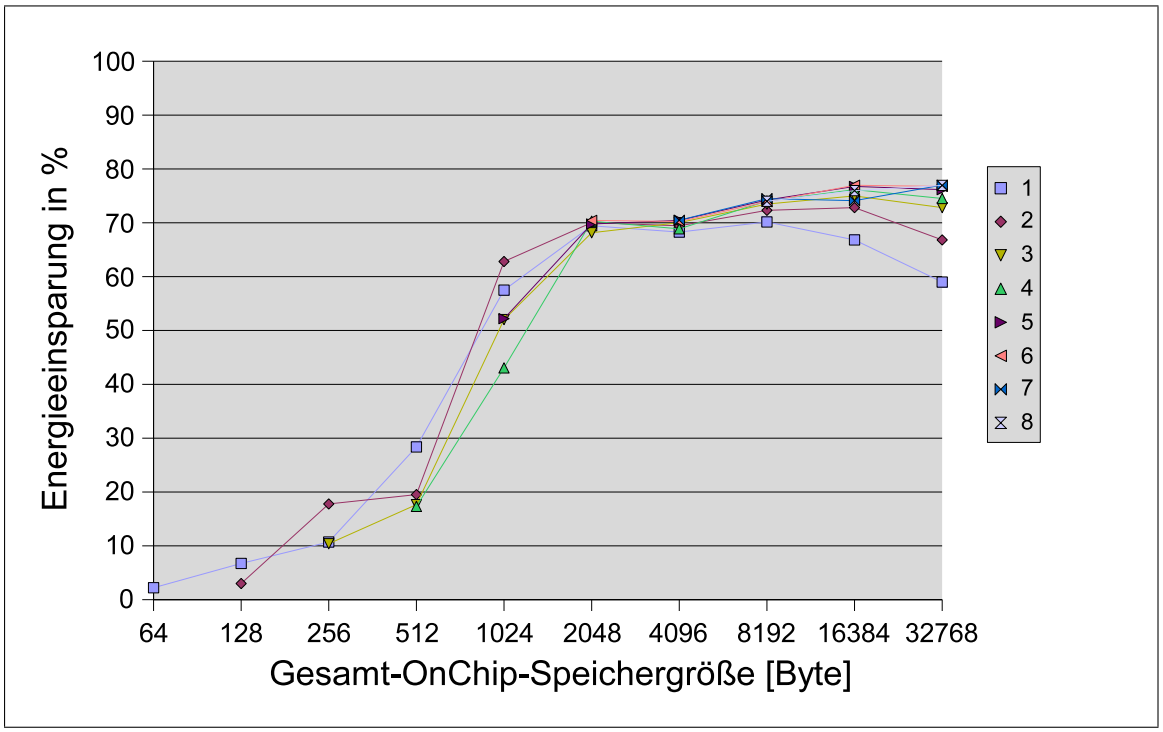
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	9,94							
128	12,54	12,02						
256	20	19,52	19,34					
512	35,43	27,15	27,17	26,97				
1024	64,73	62,57	52,15	52,05	52,09			
2048	69,43	70,03	70,31	70,37	70,37	70,45		
4096	68,27	69,43	70,03	70,43	70,33	70,29	69,05	
8192	70,15	72,31	73,47	72,41	74,35	74,15	74,39	74,53
16384	66,81	72,81	74,99	76,16	75,2	77,08	76,94	77,08
32768	58,99	66,81	72,81	74,99	76,16	76,76	74,41	77,18

(b) Dynamisches Profiling

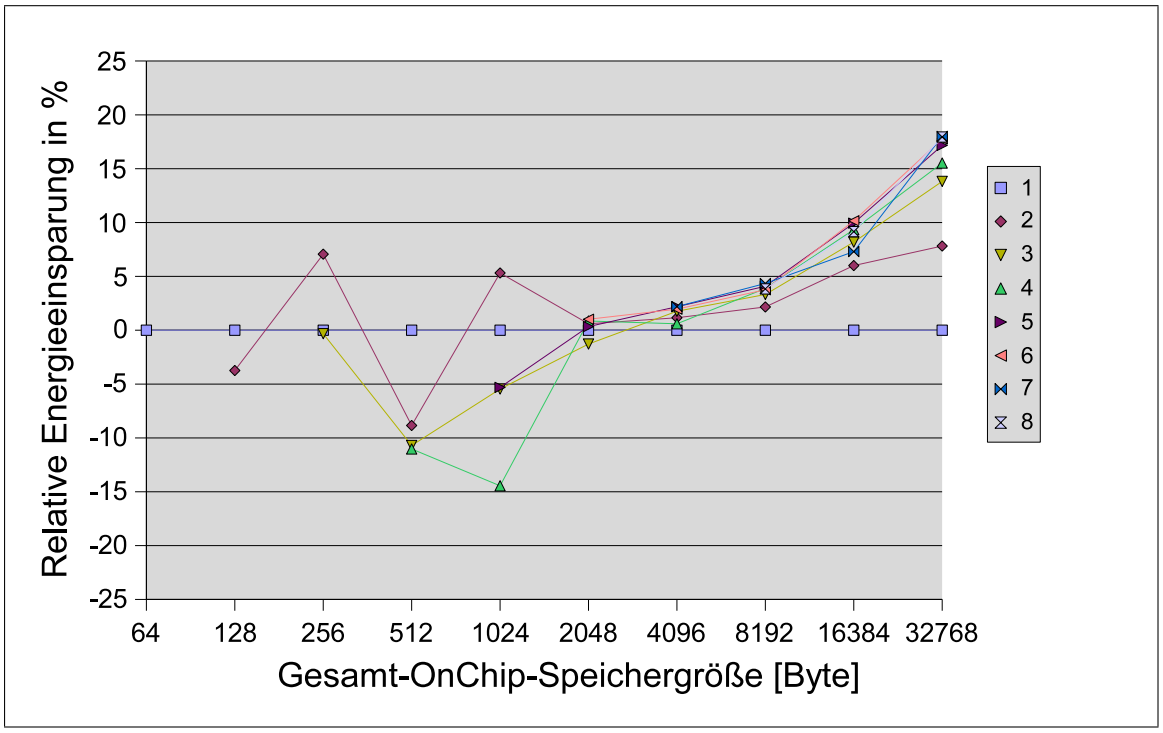
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	9,68							
128	14	13,64						
256	20,7	20,96	20,72					
512	35,91	36,03	35,47	35,05				
1024	63,95	64,83	64,83	64,63	63,39			
2048	69,43	67,97	68,51	69,45	68,95	69,23		
4096	68,27	69,43	68,25	69,71	69,63	68,67	68,45	
8192	70,15	72,31	73,41	74,07	72,79	73,13	72,45	72,65
16384	66,81	72,77	74,99	76,08	76,74	75,24	75,16	75,66
32768	58,99	66,81	72,73	74,65	76,14	76,74	75,82	75,88

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.4: Die Energieeinsparung bei dem Programm „Fast_idct“ in %

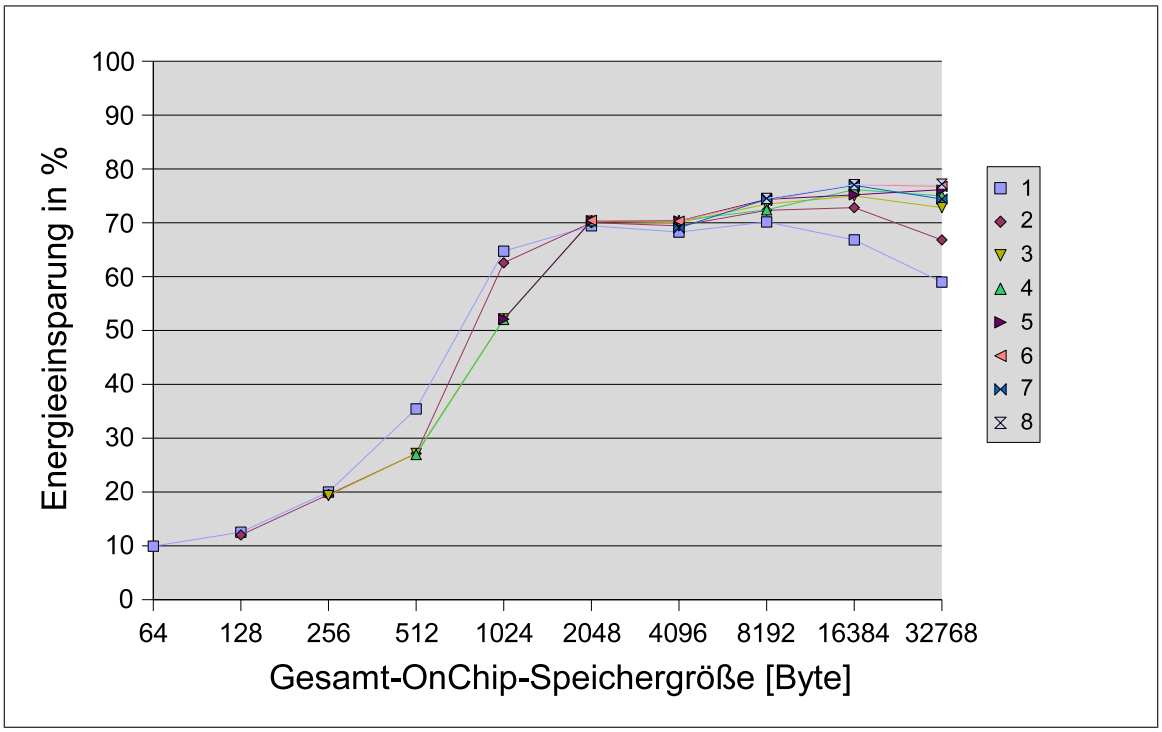


(a) Die Energieeinsparung in %

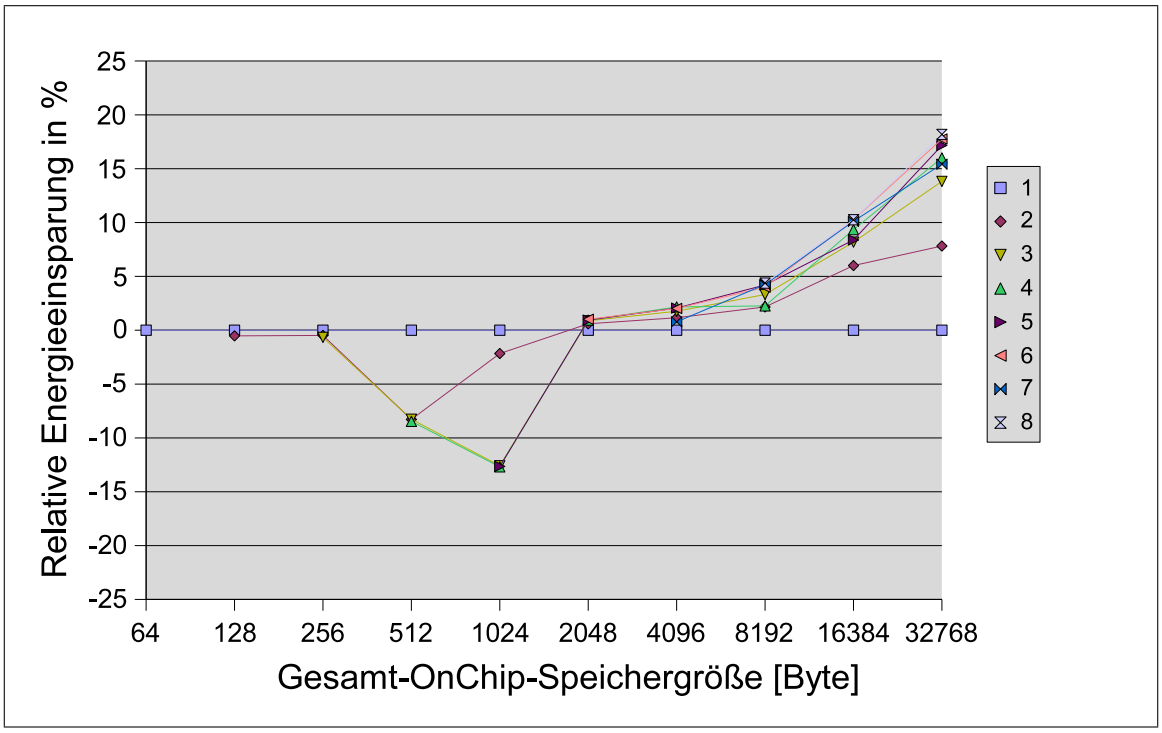


(b) Die relative Energieeinsparung in %

Abbildung A.7: Fast_idct (Statische Analyse)

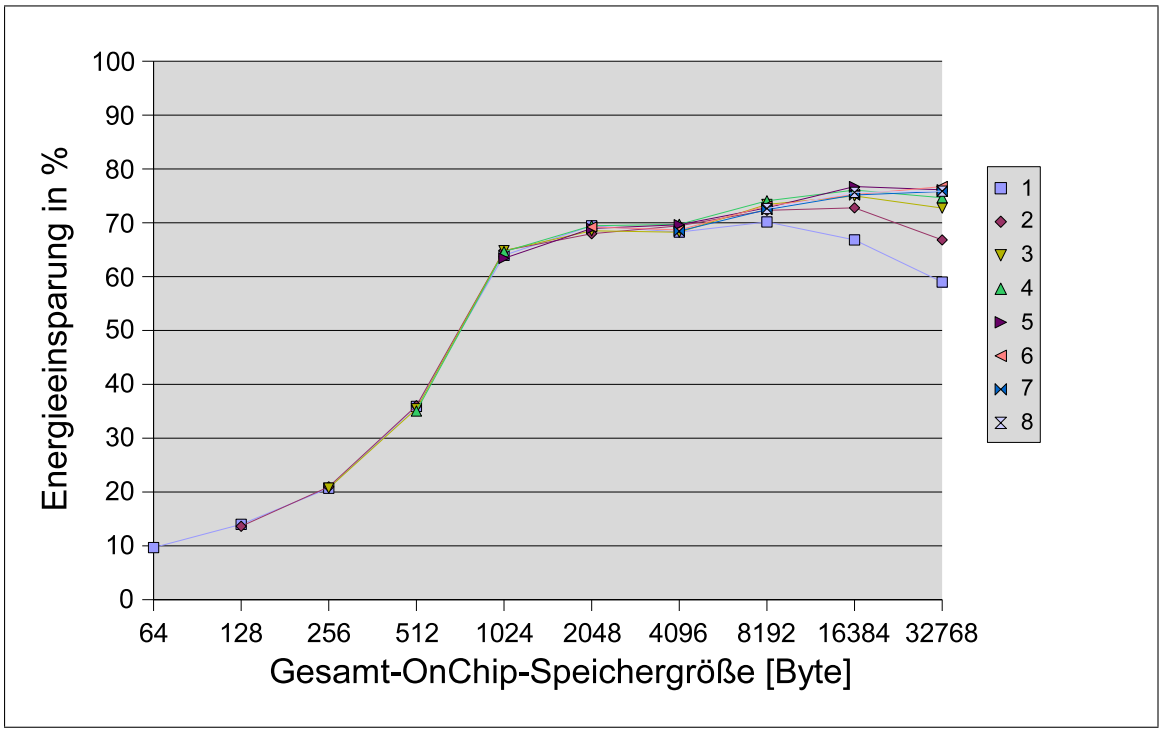


(a) Die Energieeinsparung in %

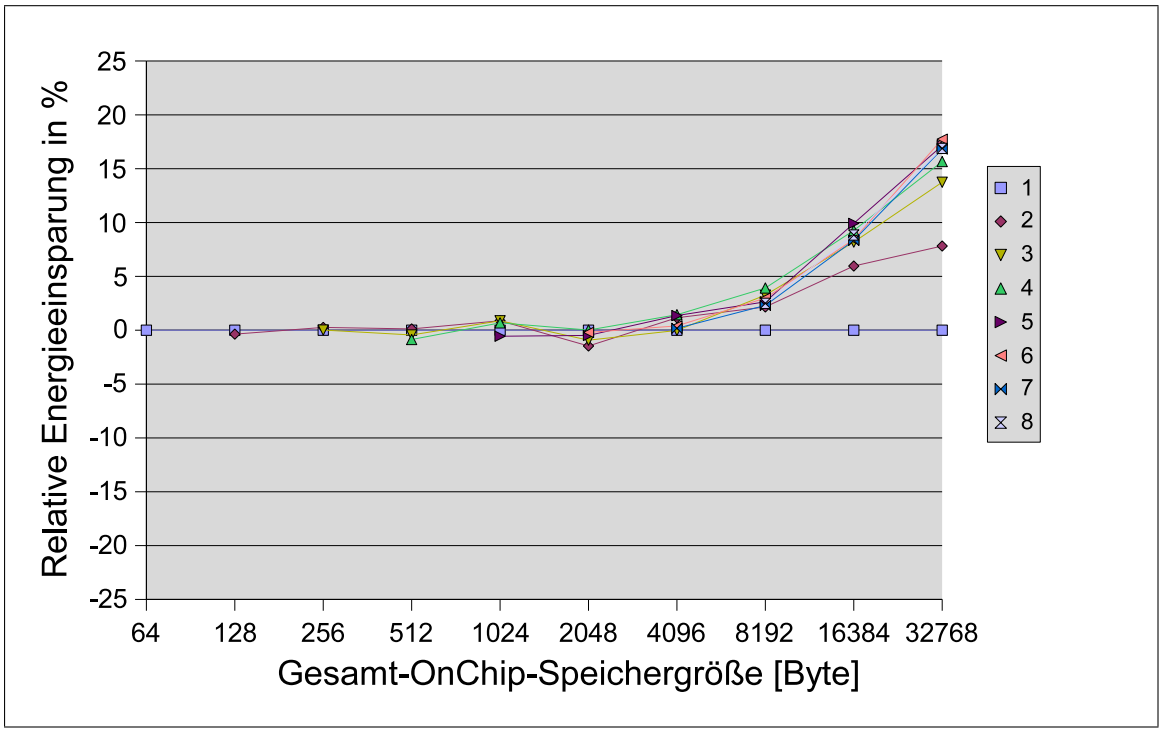


(b) Die relative Energieeinsparung in %

Abbildung A.8: Fast_idct (Dynamisches Profiling)



(a) Die Energieeinsparung in %



(b) Die relative Energieeinsparung in %

Abbildung A.9: Fast_idct (Dynamisches Profiling mit kleinen Basis-Blöcken)

Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	1,8							
128	2,23	1,82						
256	2,65	2,66	2,6					
512	41,93	2,65	2,66	2,66				
1024	41,66	41,93	41,93	41,93	41,94			
2048	41,3	41,66	41,93	41,93	41,93	41,94		
4096	41,76	42,47	42,83	43,11	43,1	43,11	43,11	
8192	42,63	43,97	44,69	45,05	45,32	45,32	45,32	45,27
16384	39,9	43,65	45	45,71	46,08	46,35	46,34	46,35
32768	36,67	41,77	44,57	45,92	46,63	46,99	47,27	47,26

(a) Statische Analyse

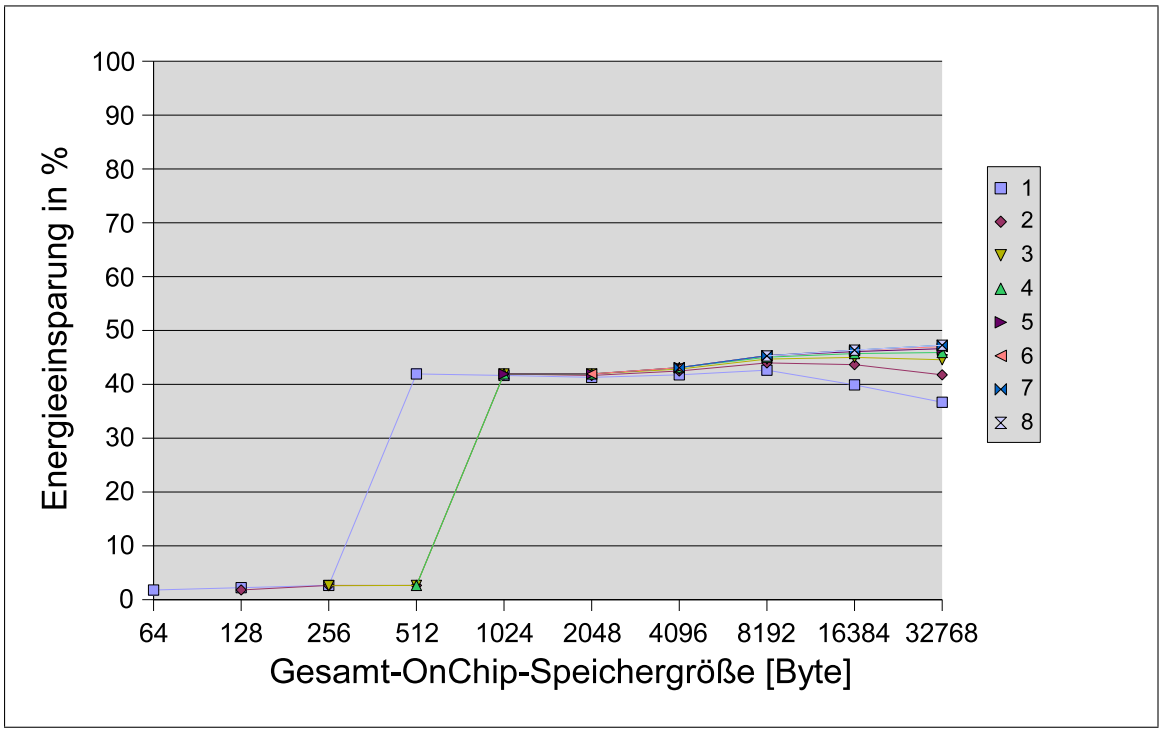
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	1,63							
128	2,23	2,64						
256	2,65	2,66	2,67					
512	41,93	2,65	2,66	2,61				
1024	41,66	41,93	41,93	41,93	41,94			
2048	41,66	41,93	41,93	41,93	41,94	41,94		
4096	41,66	42,37	42,73	43,01	43	43,01	43,01	
8192	40,38	41,66	42,37	42,73	43,01	43	43,01	43,01
16384	37,84	40,38	41,66	42,37	42,73	43,01	43	43,01
32768	34,02	38,76	41,3	42,58	43,29	43,65	43,92	43,92

(b) Dynamisches Profiling

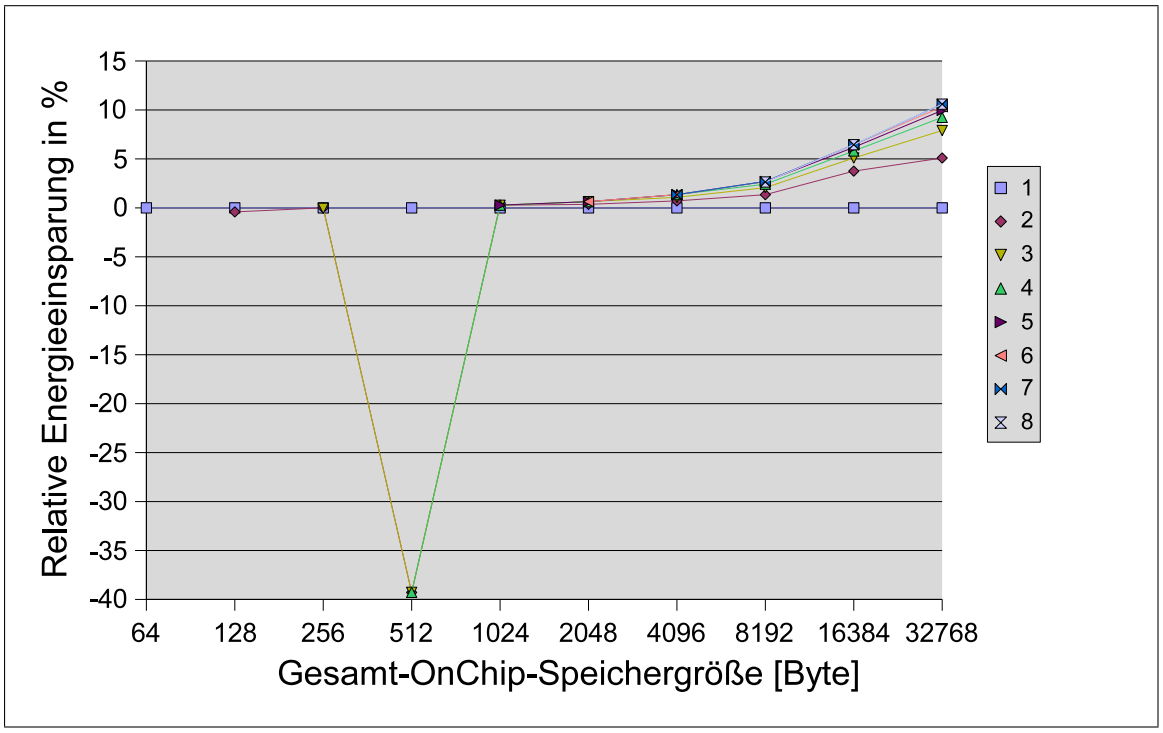
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	8,13							
128	16,69	15,65						
256	33,93	33,43	33,23					
512	41,3	41,66	41,93	41,51				
1024	41,66	41,93	40,94	42,03	41,82			
2048	41,3	41,66	41,93	41,51	41,47	40,67		
4096	41,66	42,37	42,73	42,99	42	43,13	42,36	
8192	40,38	41,66	42,37	42,71	42,99	42,54	42,58	42,33
16384	37,84	40,38	41,66	42,32	42,71	42,99	41,97	43,09
32768	34,02	38,64	41,2	42,58	43,26	43,65	43,92	43,41

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.5: Die Energieeinsparung bei dem Programm „FFT_Viva“ in %

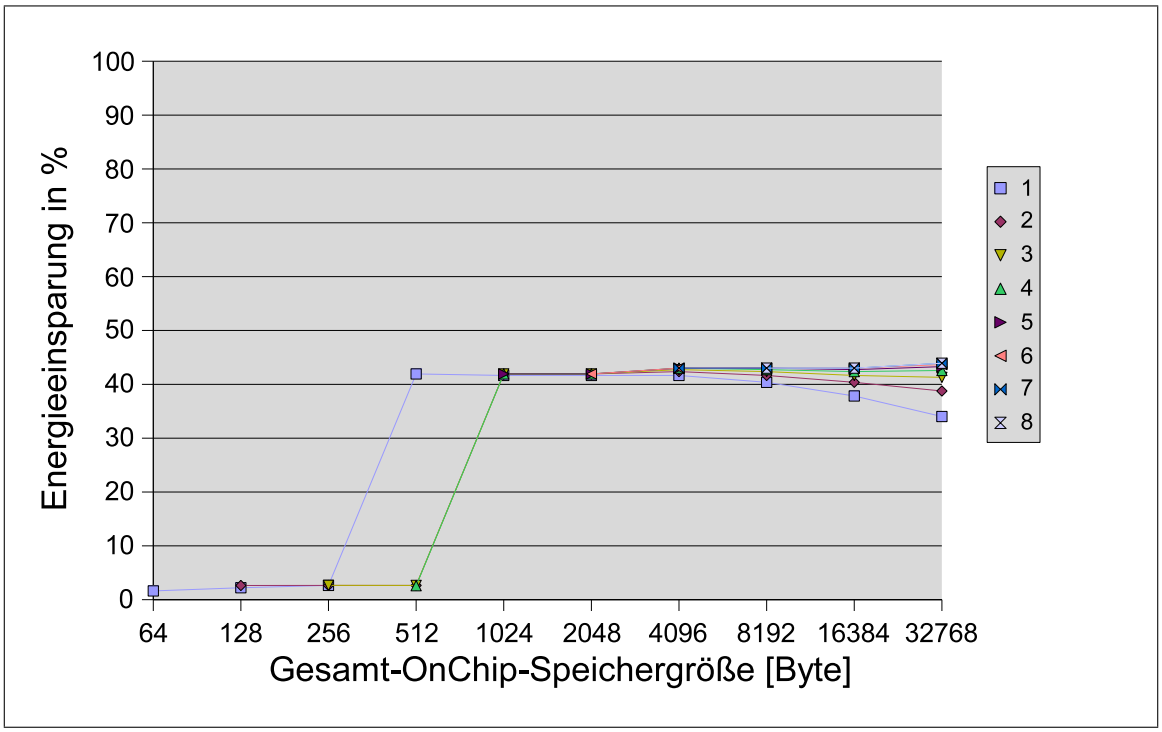


(a) Die Energieeinsparung in %

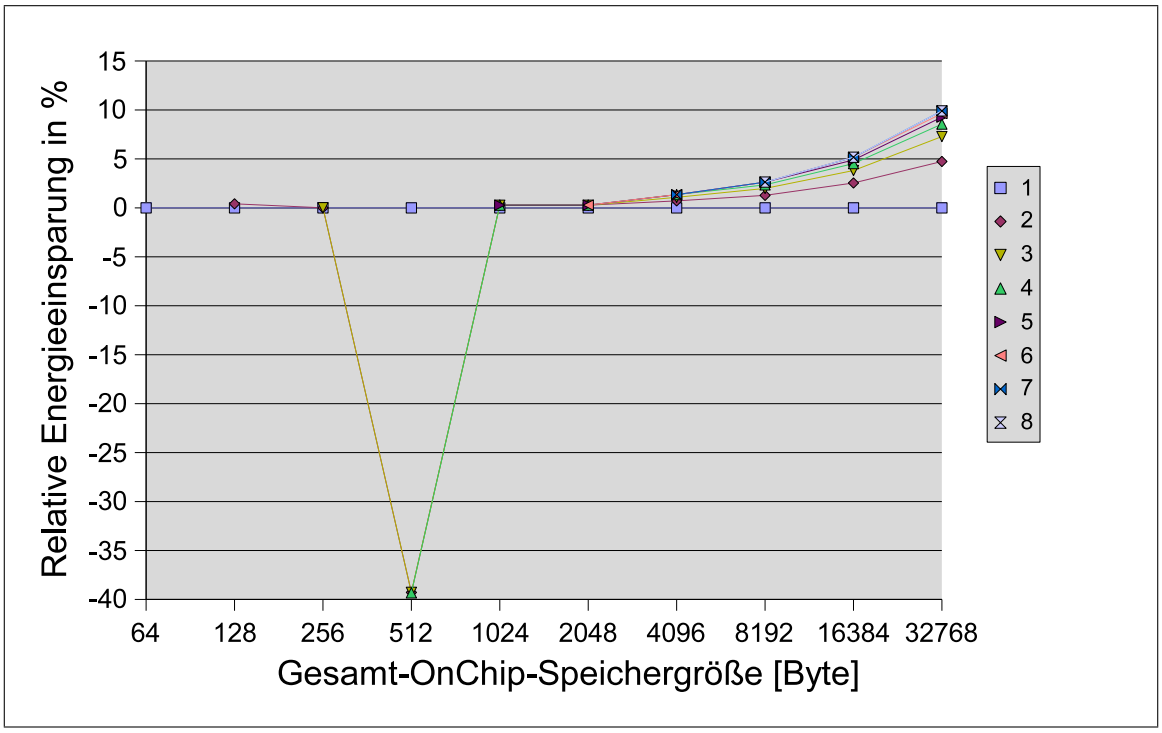


(b) Die relative Energieeinsparung in %

Abbildung A.10: FFT_Viva (Statische Analyse)

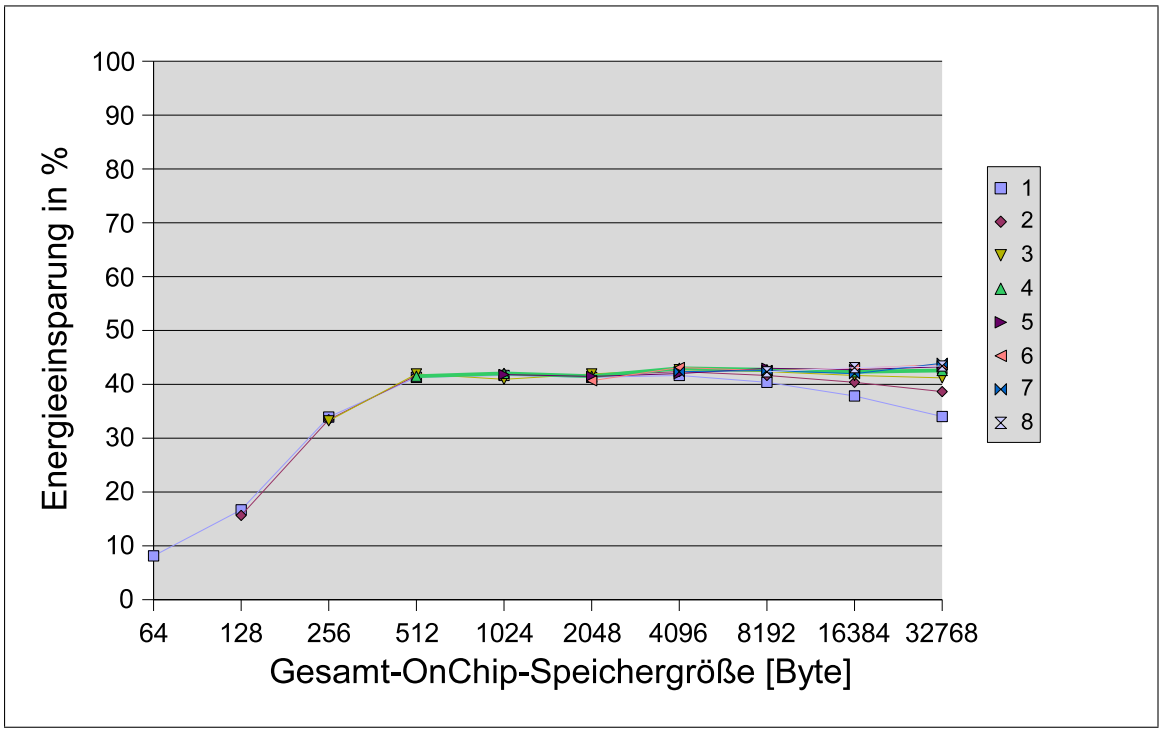


(a) Die Energieeinsparung in %

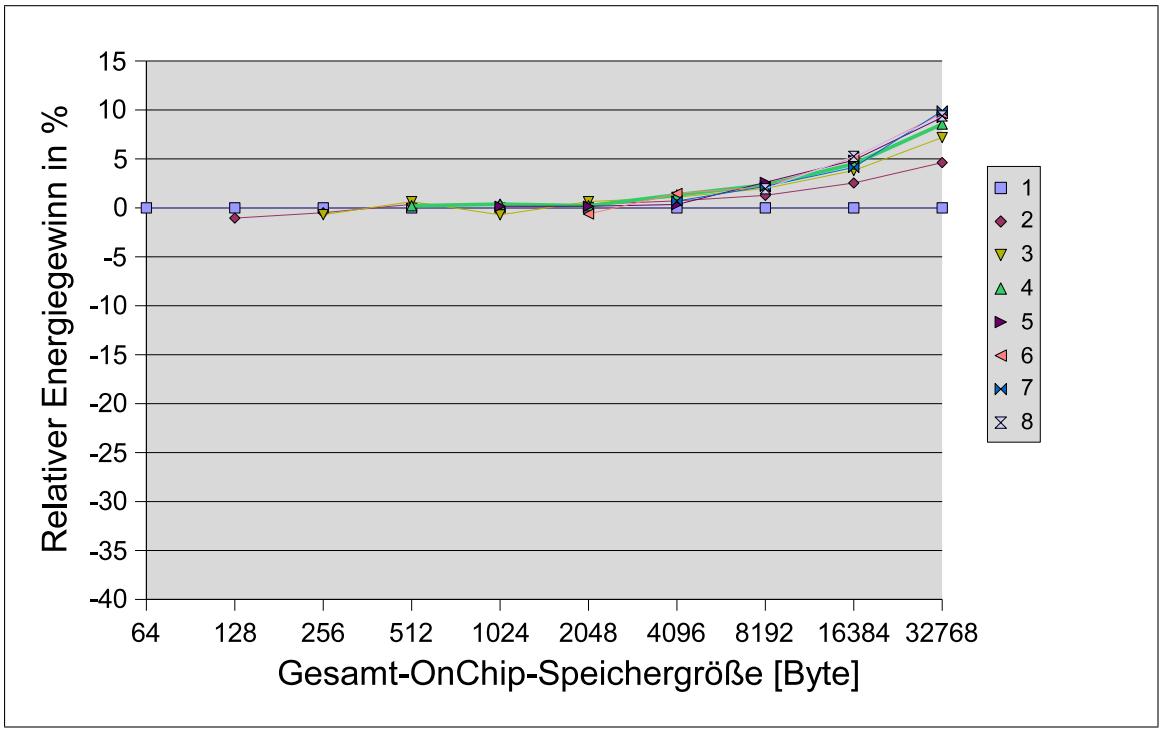


(b) Die relative Energieeinsparung in %

Abbildung A.11: FFT_Viva (Dynamisches Profiling)



(a) Die Energieeinsparung in %



(b) Die relative Energieeinsparung in %

Abbildung A.12: FFT_Viva (Dynamisches Profiling mit kleinen Basis-Blöcken)

Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	1,58							
128	3,53	3,62						
256	5,45	5,45	5,61					
512	5,9	5,74	5,48	5,73				
1024	6,43	5,89	6,04	5,92	5,59			
2048	6,47	6,47	6,23	6,31	6,52	6,4		
4096	6,37	6,47	6,52	6,56	6,15	6,42	6,44	
8192	6,28	6,37	6,56	6,61	6,31	6,27	6,51	6,61
16384	5,78	6,28	6,46	6,56	6,61	6,61	6,65	6,49
32768	5,14	5,78	6,28	6,46	6,55	6,61	6,64	6,29

(a) Statische Analyse

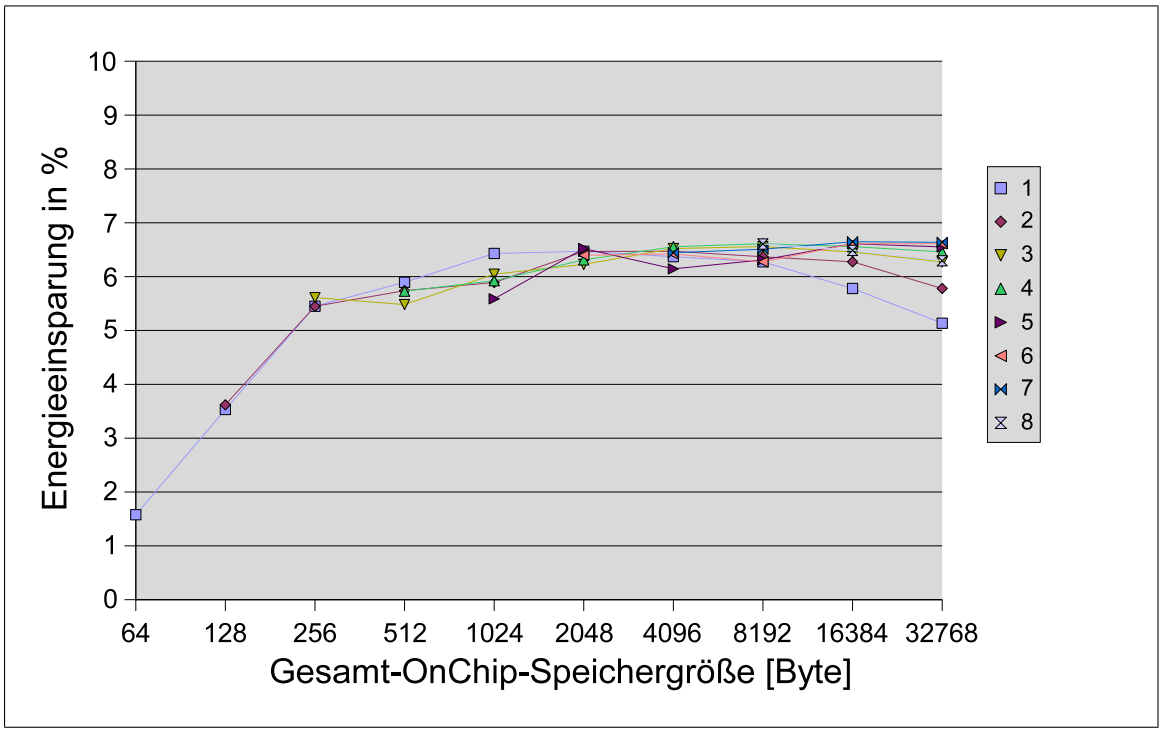
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	2,22							
128	4,33	4,35						
256	5,45	5,45	5,54					
512	5,57	5,61	5,72	6,51				
1024	6,43	6,55	6,52	6,25	6,35			
2048	6,47	6,47	6,56	6,39	6,23	6,48		
4096	6,37	6,47	6,14	6,23	6,16	6,23	6,16	
8192	6,28	6,46	6,55	6,61	6,31	6,29	6,32	6,61
16384	5,78	6,28	6,46	6,56	6,61	6,64	6,26	6,32
32768	5,14	5,78	6,28	6,46	6,56	6,61	6,64	6,28

(b) Dynamisches Profiling

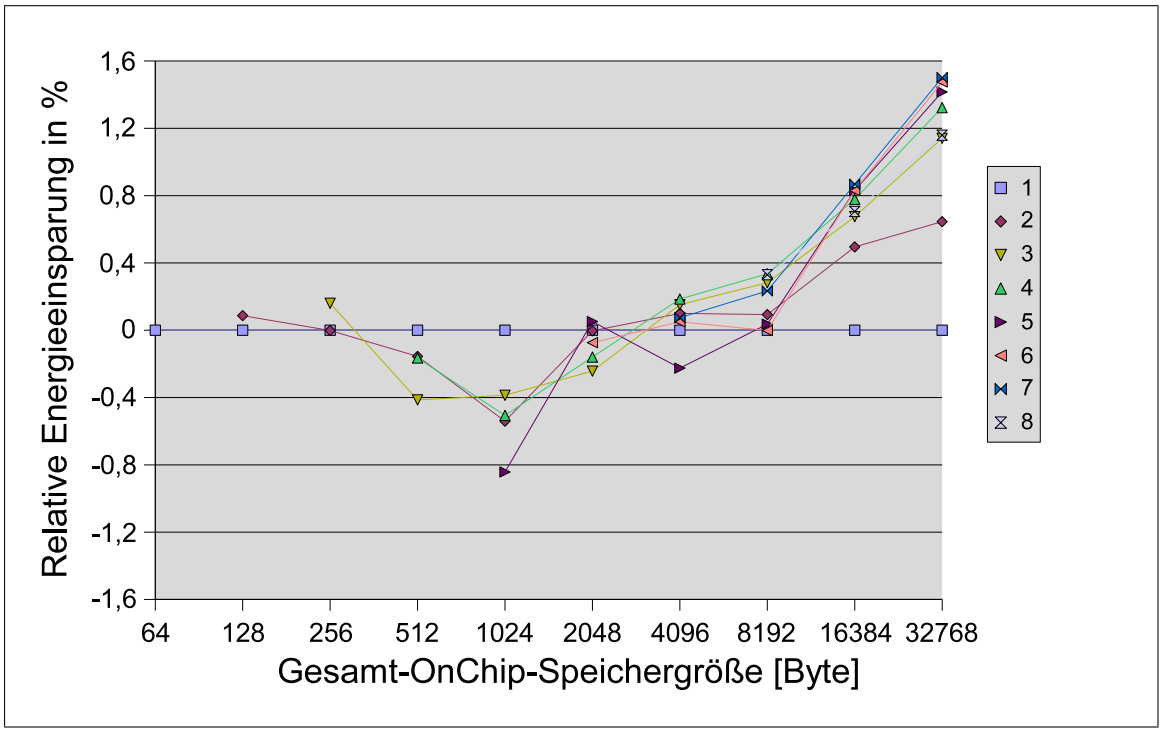
Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	1,98							
128	4,42	4,35						
256	5,45	5,45	5,35					
512	5,57	5,48	5,68	5,87				
1024	6,43	6,55	6,06	6,48	6,32			
2048	6,47	6,51	6,56	6,23	6,19	6,19		
4096	6,37	6,47	6,48	6,22	6,3	6,49	6,41	
8192	6,28	6,46	6,55	6,57	6,64	6,23	6,24	6,52
16384	5,78	6,28	6,44	6,55	6,57	6,65	6,29	6,67
32768	5,14	5,78	6,28	6,45	6,55	6,53	6,31	6,28

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.6: Die Energieeinsparung bei dem Programm „Ref_idct“ in %

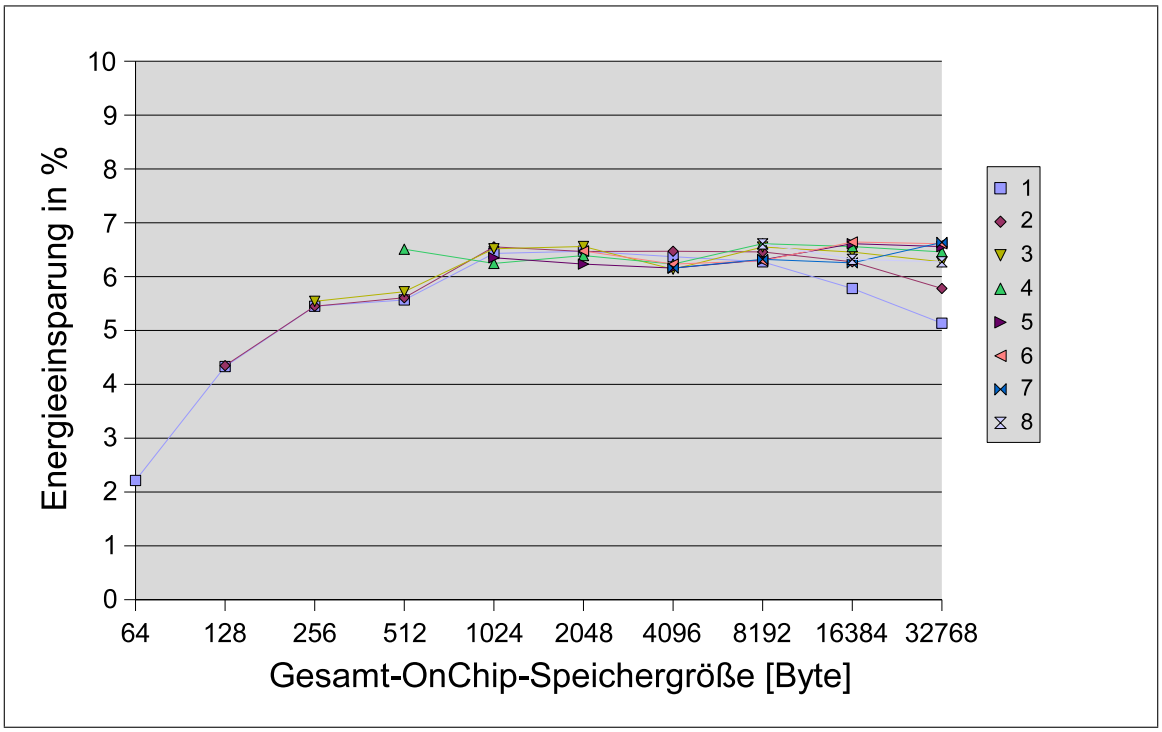


(a) Die Energieeinsparung in %

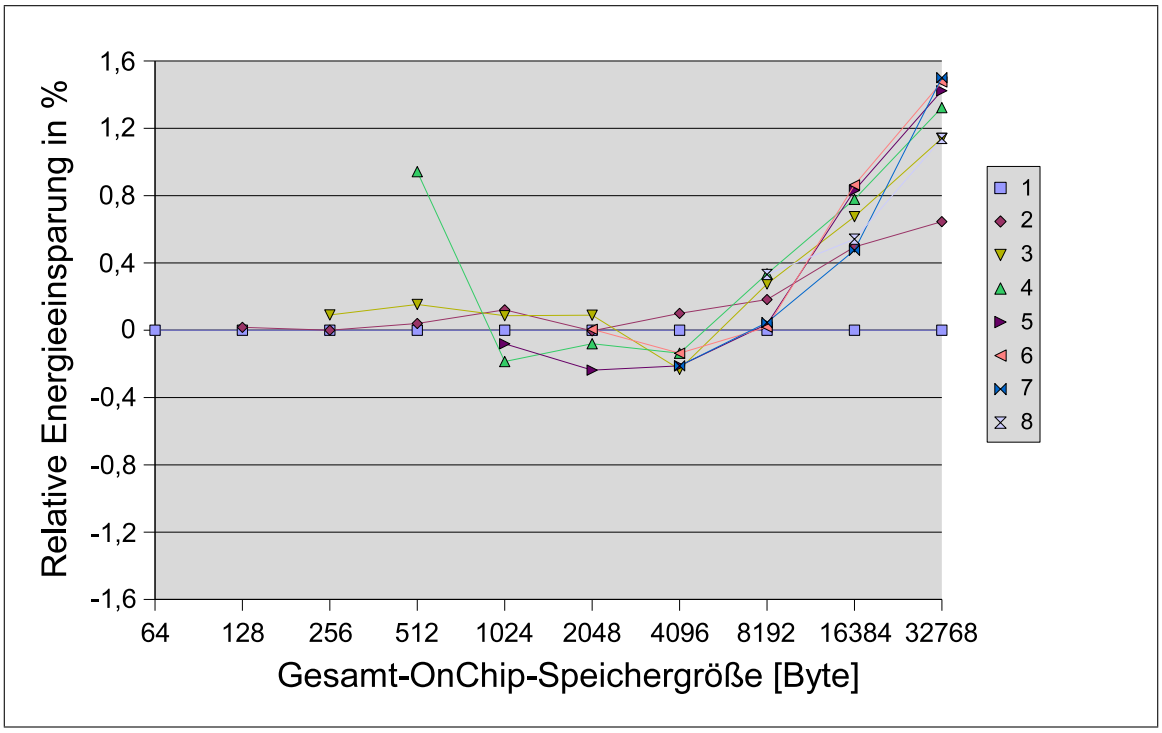


(b) Die relative Energieeinsparung in %

Abbildung A.13: Ref_idct (Statische Analyse)

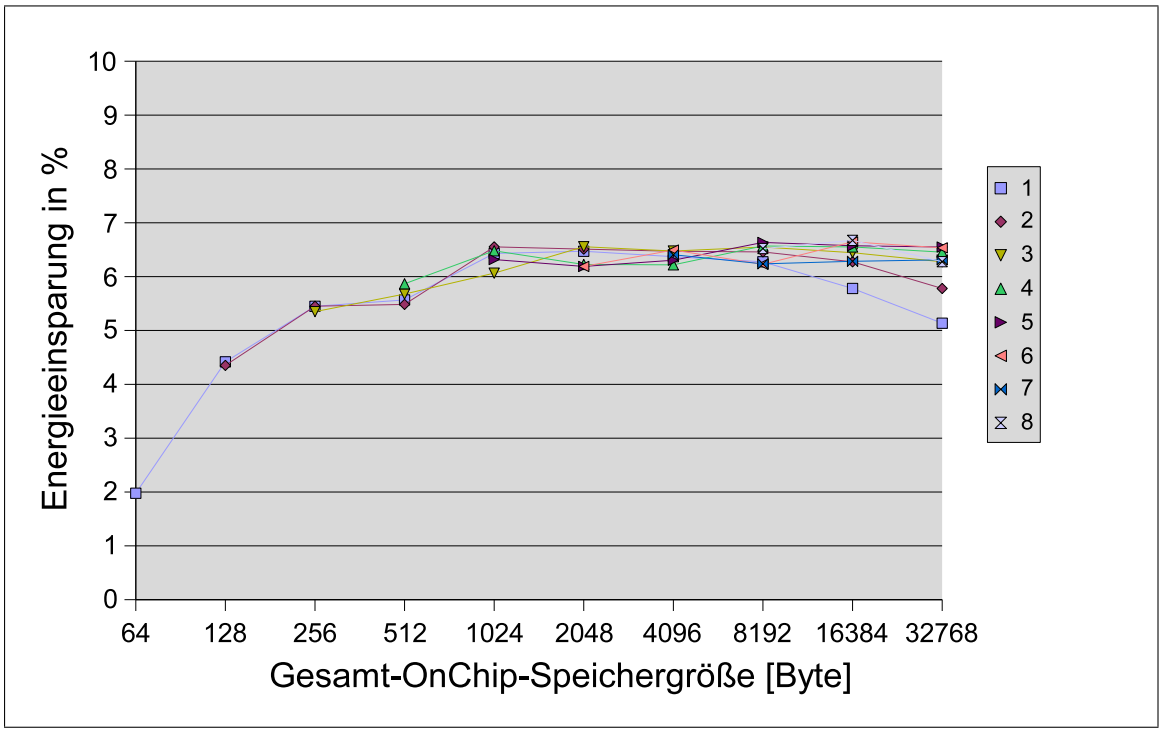


(a) Die Energieeinsparung in %

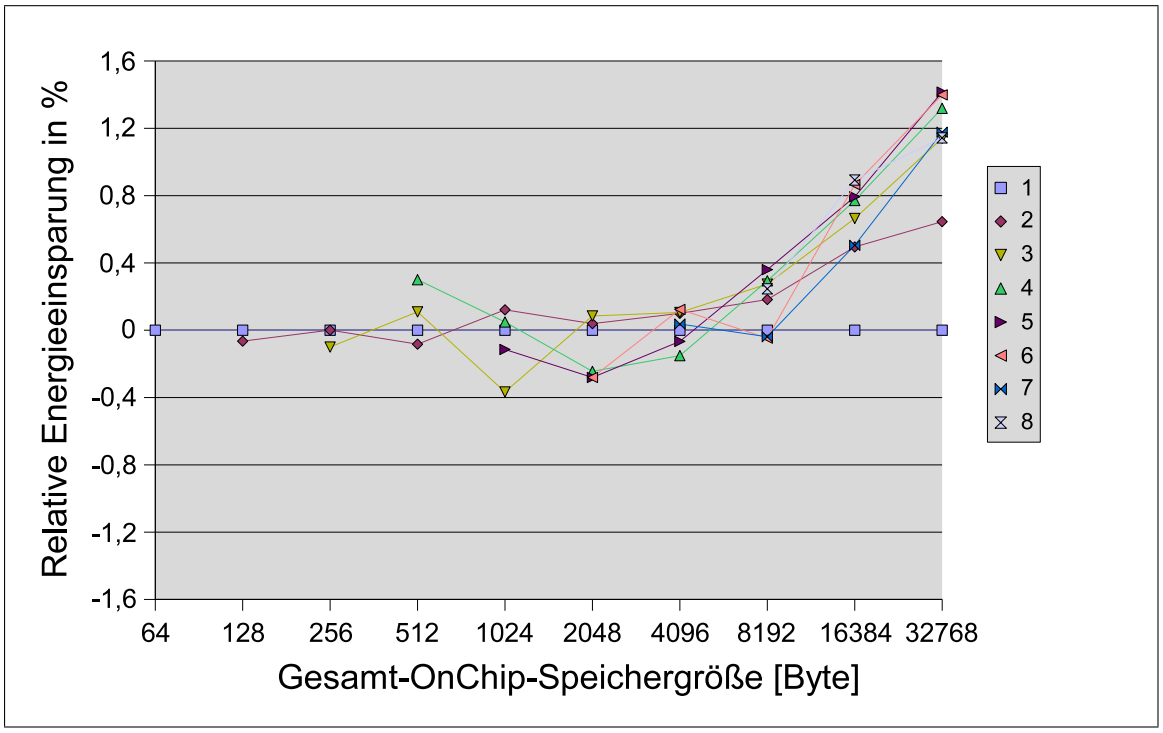


(b) Die relative Energieeinsparung in %

Abbildung A.14: Ref_idct (Dynamisches Profiling)



(a) Die Energieeinsparung in %



(b) Die relative Energieeinsparung in %

Abbildung A.15: Ref_idct (Dynamisches Profiling mit kleinen Basis-Blöcken)

Gesamtgröße (Byte)	Anzahl der Speicherpartitionierungen							
	1	2	3	4	5	6	7	8
64	4,59							
128	12,23	12,8						
256	44,38	44,44	48,12					
512	36,89	37,56	37,88	31,27				
1024	44,38	44,44	48,12	46,28				
2048	52,61	56,85	54,97	47,54	55,64	49,85		
4096	58,99	59,33	58,23	47,4	56,9	47,4	56,9	
8192	66,28	68,13	64,83	59,63		63,02	61,27	
16384	61,63	67,17	69,18	59,96	59,11	63,16	65,22	65,65
32768	54,84	62,29	67,85	69,8	68,71	60,43	62,55	66,17

(a) Dynamisches Profiling

Tabelle A.7: Die Energieeinsparung bei dem Programm „mpeg2dec“ in %

Typ	Größe [Byte]
Basis-Blöcke und Funktionen:	14025
Symbole:	22568

(a) Die Gesamtgröße der Programmobjekte

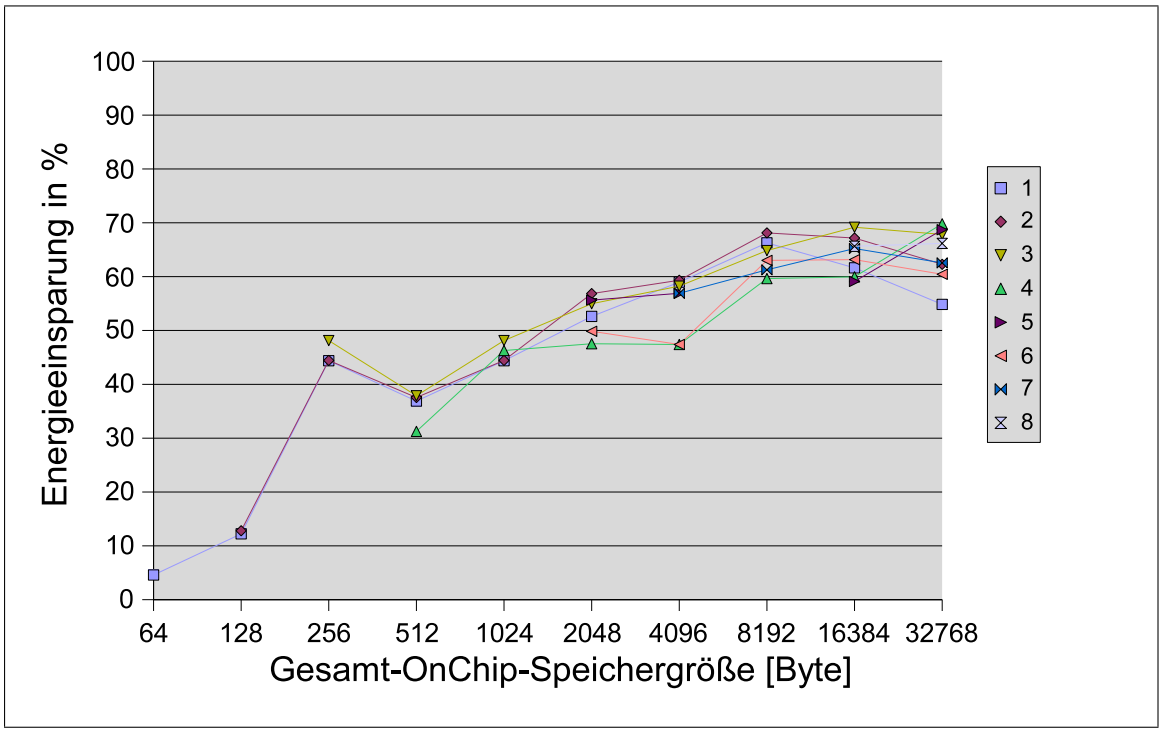
Typ	Anzahl
Basis-Blöcke:	1475
Funktionen:	75
Symbole:	4096

(b) Die Anzahl der Programmobjekte

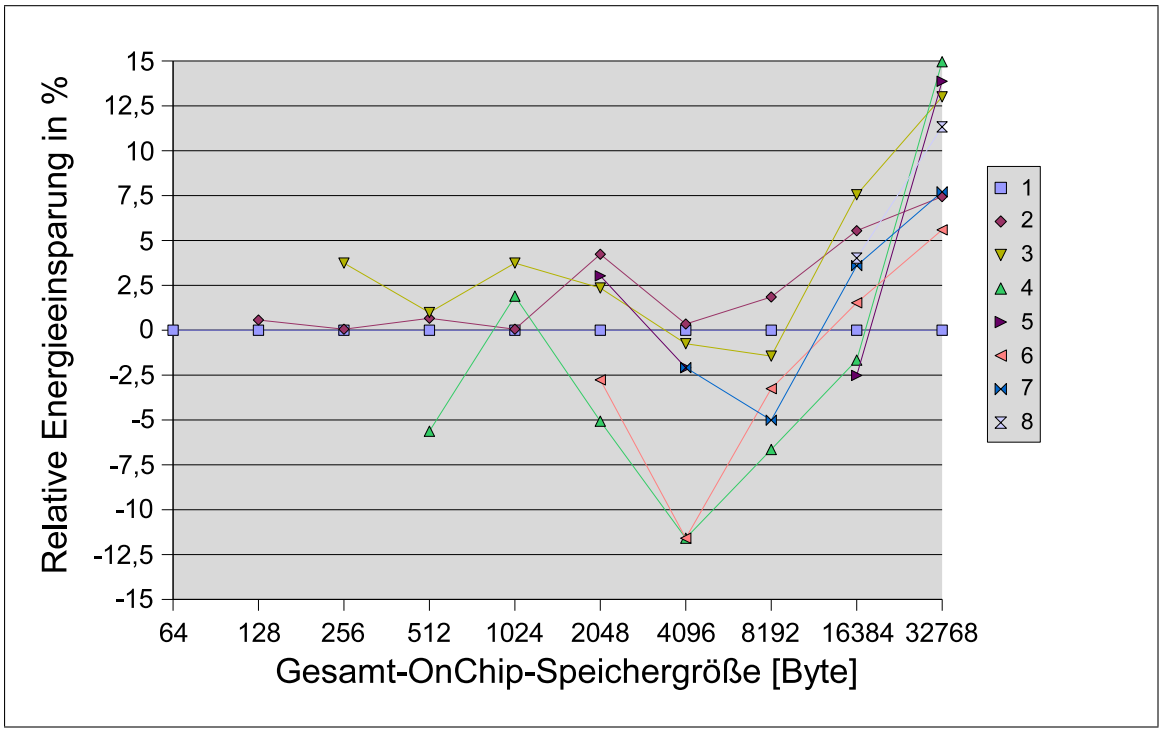
Typ	Größe [Byte]
Basis-Blöcke:	416
Funktionen:	1094
Symbole:	4096

(c) Die maximalen Objektgrößen

Tabelle A.8: Die Programmeigenschaften des Programms „mpeg2dec“



(a) Die Energieeinsparung in %



(b) Die relative Energieeinsparung in %

Abbildung A.16: mpeg2dec (Dynamisches Profiling)

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	2,32	34,12	2,22	1,8	1,58
128	5,45	42,3	6,74	2,23	3,53
256	23,7	51,39	10,72	2,65	5,45
512	76,34	58,59	28,37	41,93	5,9
1024	77,81	64,66	57,49	41,66	6,43
2048	96,04	71,09	69,43	41,3	6,47
4096	94,57	76,09	68,27	41,76	6,37
8192	91,93	73,77	70,15	42,63	6,28
16384	84,76	67,43	66,81	39,9	5,78
32768	75,41	59,17	58,99	36,67	5,14

(a) Statische Analyse

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	32,94	33,56	9,94	1,63	2,22
128	65,95	43,18	12,54	2,23	4,33
256	72,73	52,64	20	2,65	5,45
512	76,67	61,56	35,43	41,93	5,57
1024	88,52	66,38	64,73	41,66	6,43
2048	96,04	74,44	69,43	41,66	6,47
4096	94,57	76,06	68,27	41,66	6,37
8192	91,93	73,74	70,15	40,38	6,28
16384	84,76	67,4	66,81	37,84	5,78
32768	75,41	59,14	58,99	34,02	5,14

(b) Dynamisches Profiling

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	33,31	34,42	9,68	8,13	1,98
128	66,55	43,41	14	16,69	4,42
256	73,07	52,69	20,7	33,93	5,45
512	76,6	61,61	35,91	41,3	5,57
1024	88,75	67,9	63,95	41,66	6,43
2048	96,04	74,44	69,43	41,3	6,47
4096	94,57	76,06	68,27	41,66	6,37
8192	91,93	73,74	70,15	40,38	6,28
16384	84,76	67,4	66,81	37,84	5,78
32768	75,41	59,14	58,99	34,02	5,14

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.9: Die Energieeinsparung in % (bei einem einzelnen Scratchpad-Speicher)

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	2,32	34,12	2,22	1,8	1,58
128	5,45	43,03	6,74	2,23	3,62
256	24,34	51,7	17,78	2,66	5,61
512	76,34	58,85	28,37	41,93	5,9
1024	78,48	65,36	62,81	41,94	6,43
2048	96,67	72,7	70,45	41,94	6,52
4096	97,55	78,21	70,47	43,11	6,56
8192	97,4	77,84	74,49	45,32	6,61
16384	97,51	78,18	76,98	46,35	6,65
32768	97,3	77,76	77	47,27	6,64

(a) Statische Analyse

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	32,94	33,56	9,94	1,63	2,22
128	65,95	43,18	12,54	2,64	4,35
256	72,73	52,64	20	2,67	5,54
512	77,02	61,7	35,43	41,93	6,51
1024	89,4	68,2	64,73	41,94	6,55
2048	97,65	78,1	70,45	41,94	6,56
4096	97,6	78,52	70,43	43,01	6,47
8192	97,7	78,44	74,53	43,01	6,61
16384	97,5	78,37	77,08	43,01	6,64
32768	97,45	78,15	77,18	43,92	6,64

(b) Dynamisches Profiling

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	33,31	34,42	9,68	8,13	1,98
128	66,55	43,41	14	16,69	4,42
256	73,07	52,69	20,96	33,93	5,45
512	76,97	61,95	36,03	41,93	5,87
1024	89,56	68,8	64,83	42,03	6,55
2048	97,63	77,59	69,45	41,93	6,56
4096	97,38	78,48	69,71	43,13	6,49
8192	97,55	78,16	74,07	42,99	6,64
16384	97,43	78,45	76,74	43,09	6,67
32768	97,23	78,48	76,74	43,92	6,55

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.10: Die maximal erreichte Energieeinsparung in %

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	0	0	0	0	0
128	0	0,73	0	0	0,09
256	0,65	0,3	7,06	0,01	0,16
512	0	0,26	0	0	0
1024	0,66	0,7	5,32	0,28	0
2048	0,63	1,61	1,02	0,64	0,05
4096	2,98	2,13	2,2	1,36	0,19
8192	5,47	4,07	4,34	2,7	0,34
16384	12,76	10,75	10,16	6,45	0,87
32768	21,89	18,6	18	10,6	1,5

(a) Statische Analyse

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	0	0	0	0	0
128	0	0	0	0,42	0,02
256	0	0	0	0,02	0,09
512	0,35	0,14	0	0	0,94
1024	0,88	1,82	0	0,28	0,12
2048	1,61	3,66	1,02	0,28	0,09
4096	3,03	2,46	2,16	1,36	0,1
8192	5,77	4,7	4,38	2,63	0,34
16384	12,74	10,96	10,26	5,17	0,86
32768	22,04	19,01	18,18	9,9	1,5

(b) Dynamisches Profiling

Gesamtgröße (Byte)	Programme				
	Multi_Sort	Encodecombined	Fast_idct	FFT_Viva	ref_idct
64	0	0	0	0	0
128	0	0	0	0	0
256	0	0	0,26	0	0
512	0,36	0,34	0,12	0,64	0,3
1024	0,81	0,9	0,88	0,38	0,12
2048	1,59	3,15	0,02	0,64	0,08
4096	2,82	2,42	1,44	1,47	0,12
8192	5,62	4,42	3,92	2,61	0,36
16384	12,68	11,05	9,92	5,25	0,89
32768	21,82	19,33	17,74	9,9	1,42

(c) Dynamisches Profiling mit kleinen Basis-Blöcken (max. 6 Byte)

Tabelle A.11: Die maximal erreichte relative Energieeinsparung in %

B Literaturverzeichnis

[AC03] Angiolini,Frederico, Benini,Luca und Caprara,Alberto: *Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning.* In: *Proceedings of the Workshop on Compiler and Architectural Support for Embedded Computer Systems CASES'03.* ACM, Ovtober 2003.

[ARM] ARM: Advanced RISC Machines Ltd., `http://www.arm.com`.

[Atm] Atmel, `http://www.atmel.de`.

[AU99] Aho, Alfred V., Sethi, Ravi und Ullman, Jeffrey D.: *Compilerbau (Teil 1 und 2).* Oldenbourg, 1999.

[CH98] Cooper, Keith D. und Harvey, Timothy J.: *Compiler-Controlled Memory.* In: *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, October 1998.

[Fre98] Fremerey, Frank: *D-Silizium: Chips made in Germany.* c't Magazin für Computertechnik `http://www.heise.de/ct`, (04), 1998.

[Gru02] Grunwald, Nils: *Energieminimierung eingebetteter Programme durch die dynamische Nutzung eines Scratchpad-Speichers.* Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl XII, April 2002.

[HP96] Hennessy, J.L. und Patterson, D.A.: *Computer Architecture : A Quantitative Approach; second edition.* Morgan Kaufmann, 1996.

[ILO] ILOG, `http://www.ilog.com`: *CPLEX optimizer.*

[Kna01] KNAUER, MARKUS: *Codierungsverfahren zur Reduktion des Energiebedarfs von Programmen.* Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl XII, Juli 2001.

[Lee01] LEE, BO-SIK: *Vergleich des Energieverbrauchs von Cache- und Scratch-Pad-Speichern für den ARM7-Prozessor.* Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl XII, Oktober 2001.

[Mar00] MARWEDEL, PETER: *Script zur Vorlesung "Prozessrechnertechnik/Eingebettete Systeme" (WS 2000/2001).* `http://ls12-www.cs.uni-dortmund.de`, 2000.

[Mar03] MARWEDEL, PETER: *Embedded System Design.* Kluwer Academic Publishers, Dordrecht, Boston, London, 2003.

[Muc97] MUCHNICK, STEVEN S.: *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[Opi03] OPITZ, RUDOLF: *Mobilfunk-Dämmerung: UMTS-Netz im offenen Test.* c't Magazin für Computertechnik `http://www.heise.de/ct`, (26), 2003.

[PM04] PETER MARWEDEL, MANISH VERMA, LARS WEHMEYER STEFAN STEINKE URS HELMIG: *Fast, Predictable, and Low-energy Memory References through Architecture-aware Compilation.* In: *Proceedings of the 2004 Asia-South-Pacific Design Automation Conference (ASPDAC)*, Seiten 4–11, January 2004.

[PN99] PANDA, P.R., DUTT, N.D. und NICOLAN, A.: *Memory Issues in Embedded Systems-On-Chip.* Kluwer Academic Publishers, 1999.

[Sch00a] SCHWARZ, RUEDIGER: *Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining.* Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl XII, September 2000.

[Sch00b] SCHWIEGELSHOHN, UWE: *Script zur Vorlesung "Technische Informatik I" (WS 2000/2001).* `http://www-ds.e-technik.uni-dortmund.de`, 2000.

[Sch01a] Schulte, Mike: *Advanced Computer Architecture (ECE401) - Lecture 13*. http://www.cse.lehigh.edu/~mschulte/ece401-01, 2001.

[Sch01b] Schwiegelshohn, Uwe: *Script zur Vorlesung "Parallele Rechnersysteme I" (WS 2000/2001)*. http://www-ds.e-technik.uni-dortmund.de, 2001.

[Sch01c] Schwiegelshohn, Uwe: *Script zur Vorlesung "Parallele Rechnersysteme II" (SS 2001)*. http://www-ds.e-technik.uni-dortmund.de, 2001.

[Ste02] Steinke, Stefan: *Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik*. Doktorarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl XII, 2002.

[VM03] Verma, Manish, Steinke, Stefan und Marwedel, Peter: *Data Partitioning for Maximal Scratchpad Usage*. In: *Proceedings of the 2003 Asia-South-Pacific Design Automation Conference (ASPDAC)*, Seiten 77–86, January 2003.

[Weg99] Wegener, Ingo: *Theoretische Informatik – eine algorithmenorientierte Einführung*. B.G. Teubner, 1999.

[WM94] Wulf, Wm. A. und McKee, Sally A.: *Hitting the Memory Wall: Implications of the Obvious*. Computer Science Report, (CS-94-48), 1994.

[Zob01] Zobiegala, Christoph: *Energieeinsparung durch compilergesteuerte Nutzung des On-Chip-Speichers*. Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl XII, September 2001.

Erklärung

Ich versichere, dass ich die Arbeit selbständig verfasst, keine anderen Hilfsmittel als die angegebenen benutzt und die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Das Gleiche gilt auch für die beigebenen Zeichnungen und Darstellungen.

_____

Datum, Unterschrift