Diplomarbeit

Energieminimierung eingebetteter Programme durch die dynamische Nutzung eines Scratchpad-Speichers

Nils Grunwald

Diplomarbeit
am Lehrstuhl 12
des Fachbereichs Informatik
der Universität Dortmund

16. April 2002

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

Inhaltsverzeichnis

1 Einleitung

Immer schnellere Computer, neue PDAs, MP3-Player und nicht zuletzt Handys haben ihren Siegeszug in der heutigen Gesellschaft schon vor längerem begonnen. Heute ist ein Leben ohne diese elektronischen Helfer schon fast nicht mehr denkbar.

In den letzten Jahrzehnten haben diese eingebetteten Systeme (EIS) eine rapide Entwicklung durchgemacht. Häufig handelt es sich bei den EIS heute um mobile Systeme, die die Mobilität des Menschen auf die Spitze treiben sollen. Alles soll an jedem Ort und zu jeder Zeit möglich sein. Da durch einen harten Konkurrenzkampf und stetig wachsende Ansprüche der Konsumenten mehr Funktionalitäten und mehr Performance für die mobilen Geräte gefordert werden, werden sie mit immer leistungsstärkeren Prozessoren sowie schnelleren und größeren Speicherbausteinen ausgestattet. Natürlich sollen dabei diese Leistungssteigerungen nicht auf Kosten der Betriebsdauer realisiert werden.

Diese Forderungen nach höherer Geschwindigkeit, mehr Funktionalität und längerer Betriebszeit führen in der Regel zu einem höheren Energiebedarf, der bei der Entwicklung von aktuellen Geräten immer mehr zu Problemen und einer Verzögerung der Marktreife führen kann. Ein Beispiel hierfür stellt die im Moment sehr stark forcierte Entwicklung neuer UMTS-Handys dar. Der Heise-Newsticker zitierte im Januar 2002 eine Studie der Financial Times Deutschland, die von großen Problemen mit der neuen Technologie berichtete:

„Die Akkus der Prototypen hielten kaum länger als eine Stunde, wenn Daten übertragen werden. Da von der Batterietechnik keine wesentliche Verbesserung zu erwarten ist, setzten die Hersteller auf sparsamere Chips. Diese sollen jedoch nicht vor 2005 verfügbar sein.“ [HE02].

Wie dieses Beispiel sehr eindrucksvoll veranschaulicht, macht es in Zukunft Sinn, sich nicht nur mit der Hardware zur Energieminimierung zu beschäftigen, da die Entwicklungszeiten bis zur Marktreife zum Teil sehr beträchtlich sind, sondern auch mit der Softwareseite. Ein neues Verfahren zur Energieminimierung eingebetteter Software soll also im Verlauf dieser Diplomarbeit erarbeitet werden.

Wie bisherige Forschungsarbeiten zeigten, wird bei RISC[1]-Prozessoren ein großer Teil der gesamt benötigten Energie im Speicher verbraucht. Dazu kommt, dass im Prozessor selbst wenig Energie eingespart werden kann. Für mögliche Energieminimierungsmethoden bietet sich aus diesen Gründen eine genauere Untersuchung der Speicher beziehungsweise der Speicherhierarchien an. Bei Betrachtung einer Speicherhierarchie ist es vorteilhaft, häufige Zugriffe auf kleine, schnelle und nah am Prozessor liegende Speicher zu verlegen. Ideal in dieser Hinsicht sind On-Chip- statt Off-Chip-Speicher.

[1] Reduced Instruction Set Computer

Heutzutage werden häufig Caches eingesetzt. Forschungsarbeiten zeigen allerdings, dass es auch lukrativ sein kann, Scratchpad-Speicher[2] einzusetzen [SWB02], da der Energiebedarf beim Zugriff auf einen solchen Speicher aufgrund der gegenüber einem Cache fehlenden Tag-Speicher und Komparatoren relativ gering ist. Auf der anderen Seite muss diese Logik des Caches, die dem Scratchpad fehlt, entweder bereits im Compiler oder im generierten Programm nachgebildet werden.

Ein Ansatz hier ist die Verlagerung von Programmteilen (Funktionen und Basicblocks) oder auch Variablen in den Scratchpad. Dabei kann selbst eine statische Verlagerung in den Scratchpad schon erhebliche Vorteile gegenüber einer Cache-Lösung bringen [LE01].

In dieser Diplomarbeit nun soll dieser statische Ansatz weiterentwickelt und die statische Nutzung des Scratchpads durch eine dynamische Nutzung erweitert werden, das heißt, dass der Scratchpad zu verschiedenen Zeitpunkten der Programmausführung mit unterschiedlichen Inhalten gefüllt und somit dynamisch wiederverwendet wird. Dazu müssen Memoryobjekte an verschiedenen Stellen im Programmcode in den Scratchpad kopiert und dort ausgeführt werden. Im Zuge dieser Diplomarbeit werden weitere Memoryobjekte wie Variablen und der Stack nicht berücksichtigt, könnten aber prinzipiell auch von einem solchen dynamischen Verfahren optimiert werden.

Da durch diese Kopierfunktionen gegenüber der Ausführung im Off-Chip-Speicher ein Mehraufwand entsteht, der durch die Ausführung im Scratchpad kompensiert werden muss, wird auch gesondertes Augenmerk auf die Kopierfunktion gelegt, da die Ausführungskosten dieser Funktion maßgeblich für die Entscheidung sind, ob es sich lohnt einen Block überhaupt auszulagern, oder nicht.

Um den Nutzen bzw. Aufwand des Kopierens und Ausführens im Scratchpad bestimmen zu können, muss eine genaue Analyse der Ausführungshäufigkeiten durchgeführt werden. Diese Analyse soll sich hier auf das Profiling stützen, das bereits fester Bestandteil des am Lehrstuhl XII der Informatik der Universität Dortmund entwickelten Compilers und enProfilers ist.

[2] Bei einem Scratchpad-Speicher handelt es sich um einen meist nur wenige Kilobyte großen Speicher, der zusammen mit dem eigentlichen Prozessorcore auf einem Chip realisiert werden kann und dabei beim Lesen und Schreiben wesentlich weniger Energie verbraucht als ein größerer externer Speicher, der nur über einen längeren Bus an den eigentlichen Prozessor angeschlossen ist.

1.1 Motivation

Mit der zunehmenden Mobilität des Menschen und zunehmender Anzahl mobiler Geräte, die diesen Trend begleiten, hat Energiesparsamkeit und niedriger Verbrauch einen neuen Stellenwert erlangt. Die Hauptgründe für diesen Umstand sind:

1. Begrenzte Energiezufuhr

   Der begrenzte Energievorrat mobiler Systeme führt zwangsläufig zu einer ebenfalls begrenzten Einsatzdauer. Diese könnte durch Reduzierung des Energieverbrauchs im Betrieb entsprechend verlängert werden und somit vielleicht auch marktentscheidende Vorteile bringen. Dabei braucht man nur an den aktuellen Handymarkt zu denken.

2. Kritische Wärmeentwicklung

   Ein hoher Energieverbrauch hat eine Erhöhung der Betriebstemperaturen zur Folge. Es entsteht zusätzlicher Kühlungsbedarf, wodurch der Platzbedarf und die Kosten für solche Systeme immer weiter steigen. Das Problem der Wärmeabfuhr wird dabei immer akuter, wenn man bedenkt, dass aktuelle PC-Prozessoren heute schon die 80 Watt Leistungsgrenze erreicht haben.

3. Hohe Zuverlässigkeit

   Die heutzutage zum Teil sehr starke Wärmeentwicklung bewirkt auch hohe Materialbeanspruchung, die sich negativ auf die Zuverlässigkeit und die Lebensdauer der Systeme auswirkt. Da die Anzahl eingebetteter Systeme auch zunehmend in kritischen Bereichen wie Autopiloten und Überwachungssystemen steigt, ist der Faktor der Zuverlässigkeit enorm wichtig geworden.

4. Energiekosten

   Auch wenn man bei mobilen Systemen mit geringem Energieverbrauch pro Einheit zu tun hat, so macht ihre ständig steigende Zahl in der Summe einen immer größeren Betrag aus.

Daraus ersichtlich wird, dass es einige gute Gründe für die Reduktion des Energieverbrauchs gibt. Sie umfassen nicht nur die Ökonomie sondern auch die Ökologie, die nicht zu vernachlässigen ist.

1.2 Verwandte Forschungsarbeiten

In einer Arbeit von Kandemir [KA00] wurden mögliche Auswirkungen von Standard-Compileroptimierungen (lineare Schleifentransformationen, Tiling, Unrolling, Fusion, Fission und skalarer Expansion) auf den Energieverbrauch von Software untersucht. Dabei wurde der Energieverbrauch im Speichersystem als auch im Prozessor selbst berücksichtigt. Das Ergebnis dieser Arbeit war, dass bei unoptimiertem Code, mehr Energie im Speichersystem als im Prozessorkern selbst verbraucht wird.

Für die Energieminimierung von Software erscheint der Einsatz von verschiedenen Speicherformen dadurch sehr sinnvoll.

Einige Forschungsarbeiten propagierten dabei den Einsatz von Caches. Wobei Caches in eingebetteten Systemen, die eine genaue Vorhersagbarkeit der Ausführungszeiten zur Einhaltung fester Zeitschranken benötigen, sehr kritisch sind, da eine Beurteilung der ungünstigsten Ausführungsgeschwindigkeit („worst case“) nur sehr schwer mit hoher Wahrscheinlichkeit möglich ist. Dennoch gab es z.B. von Tomiyama [TO96] schon Untersuchungen, um eine optimale Code-Platzierung zu erhalten, die in weniger Cache-Misses resultiert und somit den Energiebedarf senkt und die Ausführungsgeschwindigkeit erhöht. Die Analyse der Codestruktur ist dabei ähnlich dem hier zu entwickelnden Verfahren zur dynamischen Nutzung des Scratchpads. Auch die Formulierung des Problems mit Hilfe der linearen Integer-Programmierung wurde für diese Arbeit übernommen, auch wenn das zu lösende Problem wenige Gemeinsamkeiten aufweist. Auch Bellas [BH00] empfiehlt den Einsatz eines zusätzlichen L0-Cache, der die am häufigsten ausgeführten Programmteile enthalten soll.

In letzter Zeit wurde von der Forschung der Blick immer mehr weg von den Cache-Systemen hin zu anderen On-Chip-Speichern wie dem Scratchpad, auf der einen Seite zur Steigerung der Geschwindigkeit und auf der anderen Seiten zur Reduktion des Energiebedarfs von Programmen, gerichtet. Speicher wie der Scratchpad werden dabei hauptsächlich dadurch interessant, dass sie eine höhere Energieeffizienz bei geringerer Integrationsfläche[3] auf dem Chip als ein entsprechender Cache versprechen.

Da ein Scratchpad eine Speicherart darstellt, die direkt vom Prozessor adressiert wird, müssen für den effektiven Einsatz dieses Speichers auch entsprechende Compiler entwickelt werden, die diesen Speicher effizient unterstützen. Nur so kann der Einsatz eines Cache-Systems zugunsten eines Scratchpad-Speichers in den Hintergrund treten.

Heute eingesetzte eingebettete Prozessoren, speziell im Bereich Multimedia und Grafik, besitzen meist einen solchen On-Chip-Speicher.

[3] Die Integrationsfläche ist die auf dem Chip benötigte Fläche, um den Scratchpad-Speicher zu realisieren. Aufgrund der einfacheren Struktur, es werden ja keine Kontroll-Schaltungen benötigt, benötigt ein gleichgroßer Scratchpad weniger Fläche auf einem Chip als ein entsprechender Cache.

Für die Art der Nutzung des Scratchpad-Speichers gibt es verschiedene Ansätze. Aufgrund der heute meist eingesetzten von-Neumann-Architektur[4] kann der Speicher verschiedene Memory-Objekte aufnehmen. So können entweder nur Datenobjekte [PA99], nur Instruktionen [IS00], oder beide Arten von Speicherobjekten im Scratchpad seinen Platz finden [SWB02]. Auch Benini [BE00] propagiert häufig zugegriffene Speicherstellen in einen kleinen programmspezifischen Speicher zu mappen, fokussiert bei seiner Arbeit aber mehr auf die Synthese von Speicher für Low Power eingebettete Systeme.

All diesen Forschungsarbeiten ist gemein, dass der vorhandene On-Chip-Speicher dabei nur statisch genutzt wird. Einmal geladene Inhalte werden im Speicher nicht mehr durch andere Speicherobjekte verdrängt.

Einen ersten Ansatz für die dynamische Nutzung des Scratchpad bietet Kandemir [KA01]. Seine Forschung richtete sich darauf, wie Teile von großen Datenstrukturen partitioniert und in den Scratchpad geladen werden können, um die Ausführungsgeschwindigkeit von Programmen aus dem Bildverarbeitungs- und Videobereich zu erhöhen. Die einzelnen Partitionen verdrängen sich dabei gegenseitig aus dem Scratchpad-Speicher.

1.3 Ziele der Diplomarbeit

Wie in der Einleitung bereits kurz erwähnt, ist das Ziel dieser Diplomarbeit die Implementierung und Untersuchung eines Optimierungsverfahrens zur Senkung des Energiebedarfs von Programmen durch die dynamische Nutzung des Scratchpads, basierend auf dem bereits untersuchten statischen Verfahren.

Um dieses Projekt zu realisieren wird im Rahmen dieser Diplomarbeit die bestehende Infrastruktur am Lehrstuhl 12, Fachbereich Informatik der Universität Dortmund, genutzt. Diese Umgebung besteht dabei maßgeblich aus dem bereits implementierten enCC-Compiler und den dazugehörenden Tools wie z.B. dem enProfiler.

Gerade der enProfiler spielt dabei eine wichtige Rolle, da mit ihm die Profilingdaten gewonnen werden, aber, noch viel wichtiger, auch die am Programm vorgenommenen Änderungen auf ihre Auswirkungen auf den Energieverbrauch untersucht werden können.

[4] Die von-Neumann-Architektur sieht eine Aufteilung eines digitalen Rechners in die Bestandteile: Eingabe-, Ausgabe-, Rechen-, Steuerwerk und den Speicher vor. Wichtig im Hinblick auf den Speicher ist hier, dass bei dieser Architektur keine Trennung von Daten und Instruktionen in verschiedene Speicher besteht. Alle Speicherobjekte werden in ein und demselben Speicher gehalten.

1.4 Überblick

Der weitere Teil der Diplomarbeit beschäftigt sich zunächst im Kapitel 2 mit themenrelevanten Gebieten wie den elektrotechnischen Grundlagen und dem komplexen Thema der Compilerentwicklung sowie den dort möglichen energiereduzierenden Maßnahmen, wozu im besonderen die Standard- und die energieverbrauchssenkenden Optimierungen gehören. Ebenso wird auf die Gemeinsamkeiten und die Unterschiede zwischen alten Overlay-Techniken und dem hier zu entwickelnden Verfahren eingegangen.

Kapitel 3 beinhaltet eine genaue Problembeschreibung und eine Formalisierung des Problems. Anhand eines Beispiels wird der gewählte Lösungsansatz dargestellt.

In Kapitel 4 wird die Realisierung vorgestellt. Zunächst wird der zur Verfügung stehende ARM7TDMI-Prozessor, das verwendete Evaluationsboard und dessen Energieverbrauch betrachtet und anschließend wird auf die Softwareseite eingegangen, wobei das Zusammenspiel zwischen den einzelnen Komponenten, deren Funktionalitäten und Arbeitsweisen im Vordergrund stehen.

Kapitel 5 gibt die mit dem neuen Verfahren erzielten Ergebnisse wieder, die hier auch ausführlich diskutiert und im Vergleich zum statischen Verfahren und einem Cache-System untersucht werden.

Das Kapitel 6 schließlich schließt diese Arbeit mit einer Zusammenfassung ab und soll einen Ausblick auf künftig noch mögliche Untersuchungen und Verbesserungen geben.

2 Grundlagen

Die Frage, warum man sich heute überhaupt mit dem Energieverbrauch und der Energieminimierung eingebetteter Programme beschäftigt, ist relativ schnell und einfach zu beantworten.

Unterm Strich ergibt sich aus den aktuellen Entwicklungen der Mikroelektronik ein immer weiter gestiegener Energiebedarf. Dieser Trend schließt den Bereich der Embedded Systems natürlich nicht aus. Auch hier werden immer schnellere und leistungshungrigere Bauteile eingesetzt, die viele neue Anwendungen überhaupt erst ermöglichen.

Im Bereich der Embedded Systems gibt es allerdings zusätzlich noch das Problem, dass diese Systeme ihre Energie oft nur aus Akkumulatoren beziehen können, um einen mobilen Einsatz zu ermöglichen. Diese Akkumulatoren haben durch ihre verwendete Technik und die Unabhängigkeit von einem Stromnetz eine begrenzte Speichermöglichkeit, die auch in den letzten Jahren durch neue Entwicklungen nicht mit der Geschwindigkeit der Entwicklung der Mikroelektronik und dem damit gestiegenen Energiebedarf Schritt halten konnten. Von einem Einsatz von Solartechnik, der eine unbegrenzte Energieversorgung sicherstellen könnte, wird heute noch fast immer abgesehen, da die Solartechnik entweder zu teuer oder zu groß und zu schwer wäre.

Aus diesem Dilemma ergibt sich ein neues Forschungsziel. Es geht heute nicht mehr nur darum, die Ausführungsgeschwindigkeit zu erhöhen, sondern auch darum, den Energieverbrauch zu reduzieren. Die ersten Bemühungen in diesem Feld wurden fast ausschließlich auf der Hardwareseite unternommen. Doch heute hat man erkannt, dass man auch durch die Modifikation von Programmen, natürlich bei gleichbleibendem Verhalten, eine Energiereduktion erreichen kann.

Da heutige Software allerdings hauptsächlich in Hochsprachen geschrieben wird, ist der direkte Einfluss des Programmierers sehr begrenzt. Eigenarten der genutzten Zielhardware sind quasi nicht nutzbar, wenn man von der möglichen Nutzung eines Power-Managements absieht. Aus diesem Grund richtet sich das Augenmerk bei der Energieoptimierung für eingebettete Software zwangsläufig auf den Compiler als wichtige Instanz zwischen Hochsprache und Hardwareebene, da die Möglichkeiten im Assembler und Linker eher beschränkt sind.

In neuerer Zeit wird auch der Einsatz von Echtzeit-Betriebssystemen immer häufiger, auch wenn der Einsatz in energiekritischen Bereichen durch den doch zum Teil erheblichen Overhead noch selten ist. Auch ein solches Betriebssystem kann entsprechend mit dem Ziel eines geringen Energieverbrauchs optimiert werden.

2.1 Energieverbrauch in eingebetteten Systemen

Um die Energieeinsparungspotenziale eines Compilers voll ausschöpfen zu können, bedarf es zunächst einer genaueren Analyse der Hardware im Hinblick auf ihren Energieverbrauch. Zu untersuchende Komponenten können bei einem mobilen Standardcomputersystem (Abb. 2.1-1) zum Beispiel umfassen:

- Motherboard (Prozessor mit Prozessorcore und On-Chip SRAM, Off-Chip DRAM, I/O-Funktionen)
- Massenspeicher (Festplatte, Floppy-Laufwerk, Tape, etc.)
- Anzeige-/Ausgabegerät (LCD-, VGA-Bildschirm)
- Stromversorgung (Netzteil)

Der mögliche Einfluss eines Compilers beschränkt sich hier allerdings auf wenige dieser Komponenten. Hauptsächlich sind dies der Prozessor und der Speicher. Diese beiden Komponenten sollen nun im Detail betrachtet werden.

Beide, Prozessor und Speicher, sind in fast allen in der heutigen Praxis relevanten Fällen durch CMOS-Schaltungen realisiert. Hauptvorteil der CMOS-Technik gegenüber anderen Techniken wie NMOS oder der bipolaren Technik ist, dass sie viel stromsparender ist und fast keinen statischen Energieverbrauch besitzt.

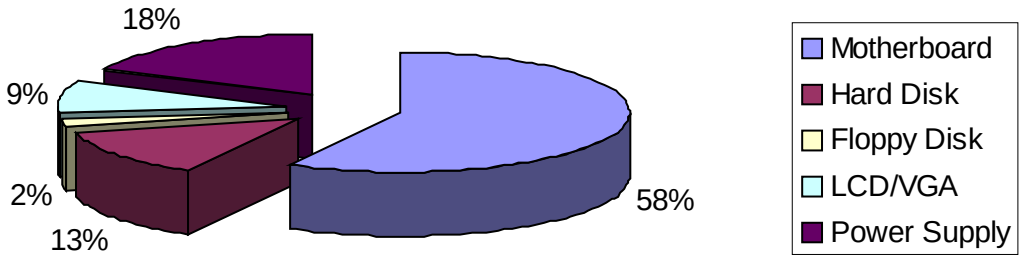

Abbildung 2.1-1 Energiebilanz eines mobilen Systems [BM01]

Die in Abbildung 2.1-1 aufgeführten Energieanteile wurden von einem handelsüblichen Laptop ermittelt. Auch wenn es sich bei einem Laptop nicht um ein „typisches“ eingebettetes System im eigentlichen Sinn handelt, so lassen sich doch durchaus die Ergebnisse auf diese Art von Systemen zum Grossteil übertragen.

2.1.1 Physikalische Grundlagen

Hier soll nun kurz auf die physikalischen Grundlagen, die für den Energieverbrauch in CMOS-Schaltungen relevant sind, und deren Definitionen eingegangen werden.

Die Leistung *P* ist definiert als Produkt aus der Spannung U und dem Strom I. Für den Fall von konstanter Spannung und konstantem Strom gilt:

Gleichung 2.1-1 Definition der Leistung

$$P = U \cdot I$$

Die Einheit der Leistung *P* ist 1 W (Watt) = 1 V (Volt) * 1 A (Ampere).

Gleichung 2.1-2 Definition der Energie – I

$$E = P \cdot t$$

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

Gleichung 2.1-3 Definition der Energie – II

$$E = U \cdot I \cdot t$$

2.1.2 Energieverbrauch und seine Ursachen

Um die Ursache für den Energieverbrauch zu verstehen, muss man bis auf die Ebene der CMOS-Schaltungen heruntergehen und sich deren Verhalten während der Ausführung von Instruktionen veranschaulichen.

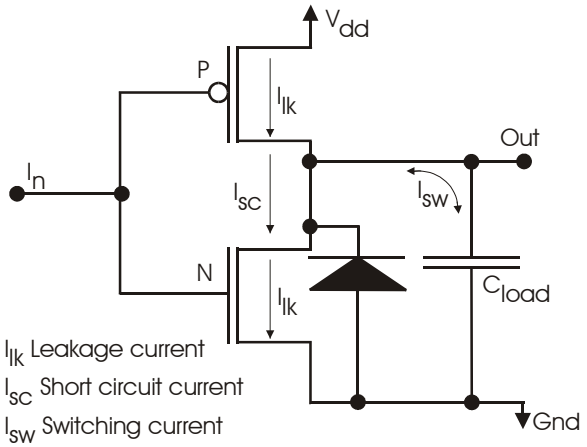

Abbildung 2.1-2 Inverter in CMOS-Technologie [SY96]

Bei diesen Schaltungen gibt es drei Ursachen, durch die Energie verbraucht wird.

- Switching Power

Der Schaltstrom ist der Drain-Strom, der beim Laden und Entladen von Lastkapazitäten am Ausgang fließt. Diese Komponente hat den größten Anteil mit 70% bis 90% des Energieverbrauchs bei aktiven Schaltungen. Der Energieverbrauch wird durch folgende Formel bestimmt:

Gleichung 2.1-4 Berechnung der Switching Power

$$P_{SW} = \frac{1}{2} V_{dd}^2 \sum_i \left(C_{Load_i} \cdot TR_i \right)$$

Dabei repräsentiert V_{dd} die Versorgungsspannung, $C_{load\ i}$ die Lastkapazität einer Zelle und TR_i die Wechselrate beim Laden und Entladen der Kapazität.

- Short Circuit Power

  Der Kurzschlussstrom fließt beim Schalten eines Transistors kurzzeitig. Je nach der Übergangszeit kann der Anteil am Gesamtenergieverbrauch bis zu 30% betragen.

  Gleichung 2.1-5 Berechnung der Short Circuit Power

$$P_{SC} = \sum_j f\left(C_{Load_j}, Tr_{input} \right) \cdot TR_j$$

  Tr_{input} entspricht dabei der Schaltzeit eines Transistors.

- Leakage Power

  Leckströme sind unerwünschte, sehr kleine Ströme, die bei angelegter Spannung durch isolierendes Material hindurchfließen. Diese Ströme können bei Halbleitern an verschiedenen Stellen auftreten. Mit 1% hat der Leckstrom einen geringen Anteil am Gesamtenergieverbrauch einer aktiven Schaltung. Allerdings wird dieser geringe Anteil mit abnehmender Versorgungsspannung größer, so dass ihm in Zukunft mehr Bedeutung zukommen wird.

  Gleichung 2.1-6 Berechnung der Leakage Power

$$P_{Leak} = \sum_i P_{Cell-Leakage_i}$$

  $P_{Cell\text{-}Leakage\ i}$ ist der Energieverbrauch einer Zelle *i* in einem inaktiven Zustand.

2.1.3 Einflussnahme auf den Energieverbrauch

Die Ebene der CMOS-Schaltungen ist eigentlich nur für Hardware-Entwickler von Interesse. Die Möglichkeiten der Software auf den Stromverbrauch Einfluss zu nehmen, sind auf dieser Ebene stark eingeschränkt.

Die Spannung ist eine der Größen, die zur Reduktion des Verbrauchs beitragen kann. Aus der Formel (2.1-3), wo Energie das Produkt der Spannung, des Stroms und der Zeit ist, bleiben noch Strom und Zeit als beeinflussbare Parameter.

Diese Parameter sind im entscheidenden Maße von der Software beeinflussbar. Dies geschieht hauptsächlich durch die Güte des Programms (Algorithmenwahl, Laufzeitkomplexität) aber auch durch den Compiler, auf den in Kapitel 2.3 eingegangen werden soll.

2.2 Energiemodell

Die im folgenden beschriebenen Energiemodelle für die On-Chip- und Off-Chip-Speicher werden benötigt, um die Auswirkungen des dynamischen Verschiebens von Programmteilen in den Scratchpad verstehen zu können und die Entscheidungen im Compiler überhaupt treffen zu können. Das Modell für den On-Chip Cache soll nur kurz dargestellt werden und wird bei der abschließenden Interpretation der Ergebnisse genutzt, um das hier entwickelte Verfahren mit einem Cache-System richtig vergleichen zu können.

Tiwari hat in seinem Arbeiten [TMW94] [TIW96] ein allgemeines Energiemodell für Prozessor-Instruktions-Kosten vorgestellt. Auf diesem Modell basierend wurde ein erweitertes Modell für verschiedene Speicherkosten (für On-Chip- und Off-Chip) erstellt [SKW01], das auch zusätzlich noch die Schaltaktivitäten auf dem Bus und die Verwendung verschiedener funktionaler Einheiten des Prozessor mit berücksichtigt. Dieses erweiterte Modell wird in dieser Form für den enProfiler verwendet.

2.2.1 Prozessorkosten

Die Prozessorkosten bestehen nach Tiwari [TIW96] aus:

- BasicCosts:

  Jede Befehlsausführung innerhalb eines Programms bewirkt eine Zustandsänderung am Prozessor. Diese Zustandsänderungen, die in einer erhöhten Schaltkreisaktivität resultieren, sind die Energiegrundkosten einer Instruktion.

  Gleichung 2.2-1 Prozessorbasiskosten

  $$E_{Base} = \sum_i \left(B_i \cdot N_i \right)$$

  Die Basisenergie E_{Base} besteht aus der Summe der Basiskosten B_i einer Instruktion multipliziert mit der Anzahl ihrer Ausführungen N_i.

- Kosten für Pipeline-Stalls und Cachemisses:

  Die meisten modernen Prozessoren haben Pipelining zur Erhöhung ihrer Leistung. Pipelining ist dabei eine Methode der parallelen Datenverarbeitung. Es gibt mehrere Phasen bei einer Ausführung einer Instruktion. Die Instruktionen werden nacheinander, sobald eine Phase der vorherigen Instruktion beendet ist, ausgeführt, so dass ein großer Teil ihrer

Bearbeitung parallel laufen kann. Allerdings können nicht immer alle Phasen parallel laufen. Bei einem Pipeline-Stall wird die Weiterverarbeitung der Befehle in der Pipeline durch eingefügte interne Wartebefehle vorerst angehalten, bis der Pipeline-Stall aufgehoben ist.

Bei einem Cachemiss wird nach einem misslungenen Cacheread/-write auf den Hauptspeicher zugegriffen. All das bedeutet zusätzliche Energiekosten, die berücksichtigt werden müssen. Die Hauptursachen für Pipeline-Stalls sind Hazards, wobei zwischen Resource-, Data- und Control-Hazards unterschieden wird.

Die Kosten für Pipelinestalls und Cachemisses werden wie folgt berechnet:

Gleichung 2.2-2 Kosten für Pipelinestalls und Cachemisses

$$E_{Pipelinestall/Cachemiss} = \sum_{k} \left(E_k \cdot N_k \right)$$

Die Energiekosten E_k werden bei jedem Auftreten N_k von Pipeline-Stalls und Cachemisses berechnet.

- Inter-Instruction Costs:

  Normalerweise gehört noch ein Kostenfaktor zu den Prozessorkosten, und zwar die „Inter-Instruction Costs“. Diese Kosten entstehen beim Wechsel von einer Instruktion zur nächsten. Ein Assemblercode besteht in der Regel aus aufeinanderfolgenden Befehlen und der Wechsel der Instruktionen führt zu unterschiedlichen Schaltkreiszuständen, da verschiedene Befehle auf unterschiedliche Prozessorressourcen zugreifen.

Die Inter-Instruction Costs fallen sowohl für den Prozessor als auch für den Speicher an. Aus praktikablen Gründen werden diese Kosten als Durchschnittswerte auf die Basiskosten aufgeschlagen. Dieser neue Wert aus Basiskosten und den durchschnittlichen Inter-Instruction Costs wird im folgenden mit E_{Base}^{*} bezeichnet.

Die Prozessorkosten E_{CPU} sind also die Summe der Basiskosten E_{Base}^{*} und der Pipelinestall- und Cachemiss-Kosten $E_{Pipelinestall/Cachemiss}$.

Gleichung 2.2-3 Prozessorkosten

$$E_{CPU} = E_{Base}^{*} + E_{Pipelinestall/Cachemiss}$$

2.2.2 On-Chip Speicherkosten (Cache)

Die Speicherkosten für den On-Chip für die Organisationsform Cache, als auch für den Scratchpad, wurden im Rahmen der Diplomarbeit von Lee ermittelt [LE01].

Da der Cache nur zu Vergleichszwecken hier angeführt wird, soll hier nur das mit dem „CACTI-Modell“ [WJ94] [RJ99] ermittelte Ergebnis dargestellt werden. Der Energieverbrauch des Caches pro Zugriff wird dabei wie folgt berechnet:

Gleichung 2.2-4 Energieverbrauch eines Caches pro Zugriff

$$E_{Cache,i,j} = \sum \begin{pmatrix} E_{Dataarray} + E_{Tagarray} + E_{Decoder} + E_{Wordline} + E_{Bitline} + \\ E_{Senseamp} + E_{Columnmuxes} + E_{Compare} + E_{Valid} + E_{Outputdriver} \end{pmatrix}$$

Dabei bedeutet *i* die Cache-Größe, *j* die Set-Assoziativität und die *E*-Werte die Energieverbrauchszahlen für das Data-Array, das Tag-Array, den Decoder (für Tag und Data), die Wordlines, die Bitlines, die Column Muxes (für Tag und Data), die Sense Amps (für Tag und Data), die Komparatoren, die Valid-Output-Drivers und schließlich die Output-Drivers.

Damit ergibt sich für den Energieverbrauch eines Caches die Summe der einzelnen Energieverbräuche pro Zugriff:

Gleichung 2.2-5 Energieverbrauch eines Caches

$$E_{Cache} = E_{Cache,i,j} \cdot N_k$$

N_k entspricht der Anzahl der Cachezugriffe und setzt sich aus der Anzahl der Lesezugriffe und der Anzahl der Schreibzugriffe zusammen:

Gleichung 2.2-6 Anzahl der Cachezugriffe

$$N_k = N_{\mathrm{Re}ad} + N_{Write}$$

2.2.3 On-Chip Speicherkosten (Scratchpad)

Wie schon erwähnt wurden die hier vorzustellenden Speicherkosten für den Scratchpad von Lee in seiner Diplomarbeit beschrieben [LE01].

Die Kosten für den Zugriff auf den Scratchpad-Speicher lassen sich einfach berechnen. Ein Zugriff auf den Scratchpad-Speicher kostet unabhängig von der Datenbreite einen Taktzyklus.

Die Kosten setzen sich einfach aus $E_{Scratchpad,size}$, dem Energieverbrauch pro Zugriff in Abhängigkeit von der Größe des Scratchpads, und N_l, der Anzahl der Zugriffe auf den Scratchpad, zusammen:

Gleichung 2.2-7 Speicherkosten: Scratchpad

$$E_{Scratchpad} = E_{Scratchpad,size} \cdot N_l$$

In der folgenden Tabelle werden einige Werte für $E_{Scratchpad,size}$ dargestellt, wobei *size* die unterschiedlichen Kapazitäten repräsentiert. Zum Vergleich werden auch noch die entsprechenden Werte beispielhaft für einen 4fach assoziativen Cache mit angegeben.

	Energieverbrauch pro Zugriff		
Kapazität	Scratchpad	Cache (4x assoziativ)	Faktor Cache/Scratchpad
64 Byte	0,49 nJ	2,71 nJ	5,5
128 Byte	0,53 nJ	3,05 nJ	5,8
256 Byte	0,61 nJ	3,32 nJ	5,4
512 Byte	0,69 nJ	3,48 nJ	5,0
1024 Byte	0,82 nJ	3,75 nJ	4,6
2048 Byte	1,07 nJ	4,04 nJ	3,8
4096 Byte	1,21 nJ	4,71 nJ	3,9
8192 Byte	2,07 nJ	5,39 nJ	2,6

Tabelle 2.2-1 Energieverbrauch eines Scratchpads pro Zugriff

Wichtiges Ergebnis der genannten Forschungsarbeit für diese Arbeit ist die Tatsache, dass ein Cache-Speicher zwischen 145% für einen „Direct-Mapped-Cache“ mit 64 Byte und 833% für einen „Fully Associative Cache“ mit 8192 Byte mehr Energie verbraucht als ein gleich großer Scratchpad.

2.2.4 Off-Chip Speicherkosten

Die Kosten der Zugriffe auf den Hauptspeicher wurden von Theokharidis im Laufe seiner Diplomarbeit [TH00] berechnet. Diese Werte wurden für die Simulation weiter benutzt. Da die Anzahl der Zugriffe auf den On-Chip-Speicher und auf den Off-Chip-Speicher sehr eng mit der Größe des On-Chip-Speichers verbunden sind, ist dieser Kostenfaktor des Off-Chip-Speichers nicht zu vernachlässigen. Es gibt zwei Arten der Off-Chip-Speicherkosten:

- Zugriffskosten beim Holen einer Instruktion:
  Diese Kosten entstehen beim Holen einer Instruktion aus dem Hauptspeicher.
- Schreib- und Lesekosten auf ein Datum:
  Wenn eine Instruktion auf ein Datum im Hauptspeicher zugreift, entstehen zusätzliche Kosten, weil das Datum beim Load in den Prozessor geschickt und beim Store in den Hauptspeicher zurückgeschrieben werden muss.

Da der enCC-Compiler, für den die dynamische Nutzung des Scratchpads untersucht werden soll, für den ARM7T-Prozessor entwickelt wurde, und dieser Prozessor für das Holen eines Befehls einen 16bit-Zugriff ausführt und bei der Simulation mit den eingesetzten Benchmarks für das Holen der Daten einen 8bit-/16bit- oder 32bit-Zugriff benutzt, sind die jeweils unterschiedlichen Wartezyklen zu berücksichtigen. Diese Zyklen sind in der nachfolgenden Tabelle wiedergegeben:

Zugriffsart:	Prozessorzyklen:
OnChip-Speicher 8-/16- und 32 bit	1 Zyklus
Hauptspeicher 8-/16 bit	1 Zyklus + 1 Wartezyklus
Hauptspeicher 32 bit	1 Zyklus + 3 Wartezyklen

Tabelle 2.2-2 Speicherzugriff: Wartezyklen ARM7T-Prozessor

Die Summe dieser beiden Kosten stellt die gesamten Off-Chip-Speicherkosten $E_{Mainmemory}$ dar.

Gleichung 2.2-8 Off-Chip-Speicherkosten

$$E_{Mainmemory} = (D_{instruction} \cdot N) + (Z_{data} \cdot M)$$

$D_{instruction}$ gibt die Kosten des Holens einer Instruktion aus dem Hauptspeicher (Instruction Fetch) an. Z_{data} ist der Energieverbrauch beim Lesen oder Schreiben eines Datums. N gibt die Anzahl der Zugriffe auf Instruktionen an. M gibt die Anzahl der Zugriffe auf Daten, lesend und schreibend, an.

Der Gesamtenergieverbrauch eines Programmablaufs lässt sich somit aus der Summe der Prozessorkosten, der On-Chip- und der Off-Chip-Speicherkosten berechnen.

Gleichung 2.2-9 Gesamtenergieverbrauch

$$E_{Gesamt} = E_{CPU} + E_{OnChip} + E_{Mainmemory}$$

E_{OnChip} entspricht dabei entweder dem Energieverbrauch im Scratchpad $E_{Scratchpad}$ oder aber dem des Caches E_{Cache}. Der Energieverbrauch des On-Chip- und der Off-Chip-Speichers wird in der Analyse des Profilers addiert und als Speicherkosten ausgegeben.

2.3 Aufbau eines C Compiler

Für die Entwicklung komplexer und vom Codeumfang großer Programme sind Hochsprachen wie C, C++ oder Java heute unumgänglich. Sie bieten einen hohen Abstraktionsgrad gegenüber Assemblerprogrammen und bieten damit die Möglichkeit, Probleme einfacher und gezielter darzustellen. Auf der anderen Seite

klafft eine semantische Lücke zwischen der Hoch- und der Zielsprache – den Maschineninstruktionen.

Ziel des Compilers, einem Analyse – Synthese Programm, ist es, diese Lücke zu schließen. Oft übertrifft der Compiler in der Komplexität sogar die zu übersetzenden Programme.

Die nachfolgende Beschreibung eines Compilers stützt sich hauptsächlich auf das Buch „Compilerbau“ von A. Aho et al.[ASU88], aber auch auf die Texte [WI95] und [GBJ00].

2.3.1 Genereller Aufbau eines Compilers

Heutige Compiler arbeiten mit verschiedenen Phasen. Diese Phasen umfassen alle Schritte, die nötig sind, ein Quellprogramm in Objektcode zu überführen.

Der generelle Ablauf soll durch die folgende Grafik verdeutlicht werden:

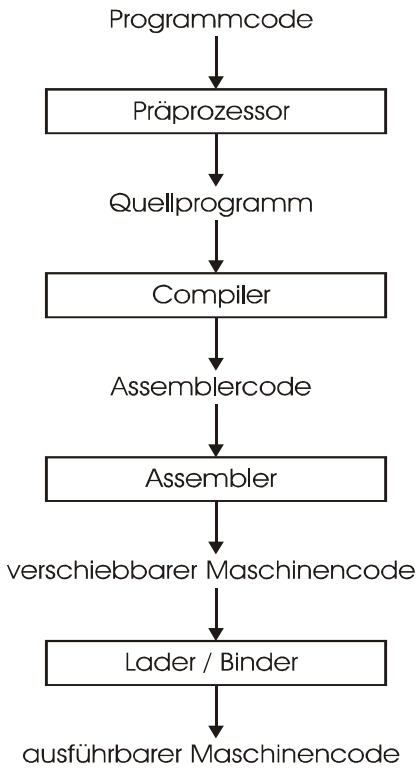

Abbildung 2.3-1 Compilerumgebung

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

Zu diesem ersten Teil zählen:

- Lexikalische und syntaktische Analyse
- Semantische Analyse
- Erzeugung von Zwischencode
- Erstellung der Symboltabelle

Diese Phasen werden oft in dem sogenannten vorderen Teil oder „Front-End“ zusammengefasst.

Der zweite Teil, das sogenannte „Back-End“, umfasst all die Phasen, die sich auf die Zielmaschine beziehen und den Zielcode generieren. Im Einzelnen sind dies:

- Befehlsauswahl
- Instruction scheduling
- Registervergabe

Darüber hinaus kann, was das eigentliche Thema dieser Diplomarbeit darstellt, der Zielcode um Kopierfunktionen erweitert werden, die in der Lage sind, während der Laufzeit des Programms, einzelne Teile des Programms in den On-Chip-Speicher zu laden und dort auszuführen, um eine Reduktion des Energieverbrauchs zu erreichen. Der Platz im On-Chip-Speicher kann dadurch flexibel immer wieder neu benutzt werden.

Die Zweiteilung des Compilers in Front- und Back-End mit der Schnittstelle Zwischencode hat sich heutzutage im Compilerbau durchgesetzt. Die Vorteile dieser Vorgehensweise liegen hauptsächlich darin begründet, dass man Compiler einfacher an neue Zielplattformen anpassen kann, indem man lediglich das Back-End anpasst. Des weiteren kann man bereits auf dem Zwischencode, der das Ausgabeformat des Front-Ends darstellt, einen maschinenunabhängigen Code-Optimierer anwenden.

2.3.2 Das Front-End

Der in einer Hochsprache verfasste Quellcode steht immer zu Anfang eines jeden Compilerlaufs. Der Quellcode stellt dabei zunächst nichts anderes als einen Zeichenstrom dar. Erst die erste Phase des Front-End, die lexikalische Analyse, transformiert den Quellcode in eine Form, die in den nachfolgenden Phasen verarbeitet werden kann.

Mit fortschreitender Übersetzung verändert sich dann die Repräsentation des Quellcodes im Front-End, bis sie schließlich die Zwischencodedarstellung erreicht hat. Der Zwischencode ist damit das Endergebnis des Front-End. Alle weiteren Verarbeitungsschritte geschehen dann im Back-End des Compilers.

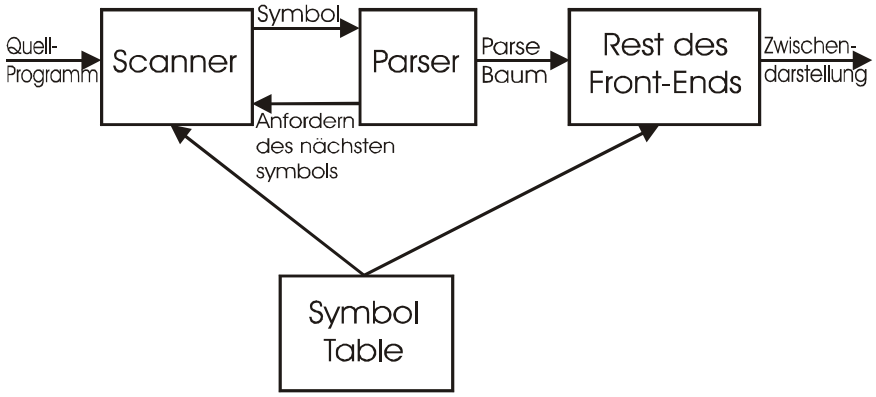

Abbildung 2.3-2 Front-End

Im Folgenden soll nun kurz auf entscheidende Punkte des Front-End eingegangen werden.

Lexikalische Analyse

Bei der lexikalischen Analyse wird jedes einzelne Zeichen des Quelltextes gelesen und in einen Strom von Symbolen umgewandelt. Dabei werden alle Kommentare, überflüssigen Leerzeichen und sonstige Formatierungen entfernt. Die Symbole stellen bereits kleine logische Einheiten als Folge von zusammengehörigen Zeichen dar und werden Lexeme genannt.

Syntaktische Analyse

In der Phase der syntaktischen Analyse wird nun der veränderte Quellcode auf seine Struktur hin untersucht. Die Hauptaufgabe hier ist es, die Symbole des Quellprogramms zu grammatikalischen Sätzen zusammenzufassen, die durch die Syntax der Hochsprache vorgegeben sind. Die Syntax lässt sich durch eine kontextfreie Grammatik beschreiben.

Semantische Analyse

Die semantische Analyse überprüft auf semantische Fehler und sammelt Typ-Informationen für die anschließende Phase der Zwischencode-Generierung. Hier wird die hierarchische Struktur, die während der syntaktischen Analyse erzeugt wird, genutzt, um die Operatoren und Operanden von Ausdrücken und Anweisungen zu unterscheiden.

Symboltabelle

Die Symboltabelle ist eine Tabelle, in der der Compiler für jeden Bezeichner einen Eintrag mit allen dazugehörigen Informationen erzeugt. Diese Informationen, Attribute genannt, sind der Speicherbedarf, der Typ und sein Gültigkeitsbereich.

Für Funktionen, die ebenfalls Bezeichner sind, werden darüber hinaus noch die Anzahl der Argumente, die Methode ihrer Übergabe und der Typ des Rückgabewertes eingetragen.

Erzeugung des Zwischencodes

Der Zwischencode schließlich ist das Endergebnis des Front-End. Dabei gibt es verschiedene Formen, die die Darstellung im Zwischencode annehmen kann. Die geläufigste hierbei ist allerdings der Drei-Adress-Code.

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

Zum Zeitpunkt, an dem der Compiler den Zwischencode erzeugt, muss er die Reihenfolge festlegen, in der die Operationen auszuführen sind. Deshalb muss der Rang der Operationen berücksichtigt werden.

Die Übersetzung der Ausdrücke ist allerdings nur ein Teil der Arbeit, die die Phase der Code-Erzeugung zu leisten hat. Sie muss darüber hinaus auch Zwischencode-Befehle generieren, die den Kontrollfluss und Unterprogrammaufrufe realisieren, also typische programmiersprachliche Konstrukte. Eine graphische Darstellung von Drei-Adress-Befehlen wird Kontrollflussgraph genannt. Er spiegelt den Kontrollfluss in einem Programm wider, wobei die Knoten, die Berechnungen und die Kanten den Fluss darstellen. Dieser Kontrollflussgraph ist als Hilfsmittel für weitere Phasen der Code-Erzeugung sehr wichtig. Viele Analyseverfahren, an die sich Optimierungen des Codes anschließen, nutzen diese Darstellung des Programms.

Neben dem Kontrollflussgraphen gibt es auch noch den sogenannten Datenflussgraphen. Dieser besteht aus Knoten, die Operatoren darstellen und gerichteten Kanten, die Daten repräsentieren.

2.3.3 Das Back-End

Im Back-End geschieht die eigentliche Maschinen- oder Assemblercode-Erzeugung. Hier wird die Zwischendarstellung des Front-End benutzt, um das ausführbare Zielprogramm zu erzeugen.

Im folgenden wird nun auf die einzelnen Phasen des Back-End wie Befehlsauswahl und Registervergabe kurz eingegangen.

Befehlsauswahl

Die Befehlsauswahl ist diejenige Phase, in der jeder Anweisung des Zwischencodes ein Befehl oder eine Folge von Befehlen des Zielprozessors zugeordnet wird. Dabei spielen Eindeutigkeit und Vollständigkeit des Befehlssatzes eine wichtige Rolle. Gute Befehlsauswahl zeichnet sich dadurch aus, dass sie die Ausführungszeiten und Inanspruchnahme der Ressourcen der einzelnen Befehle berücksichtigt und die effizientesten auswählt. Sehr wichtig in diesem Zusammenhang ist auch die Ausnutzung der Besonderheiten des Befehlssatzes.

Registervergabe

Die effiziente Nutzung der Register des Prozessors ist für den Energieverbrauch und die Ausführungsdauer des Zielcodes von besonderer Bedeutung. Sehr wichtig in diesem Zusammenhang ist die Vermeidung oder zumindest die Minimierung des sogenannten Spillings. Beim Spilling werden Register zwischengespeichert, damit

sie vorübergehend andere Daten aufnehmen können. Durch das Abspeichern der Registerinhalte und das spätere Zurückladen entsteht ein immenser Overhead. Durch geschickte Registervergabe kann dieser Effekt minimiert werden.

2.3.4 Optimierungen

Moderne Compiler nehmen in einigen Phasen Änderungen am Programmcode vor, ohne die Semantik zu verändern. Diese Veränderungen dienen in verschiedener Hinsicht der Verbesserung der Codequalität. Ziele hierbei sind unter anderem:

- ein kompakterer Code
- schnellere Ausführungszeiten
- ein geringerer Energieverbrauch

Die Compiler, die in die letztgenannte Kategorie fallen, werden *Power Aware Compiler* genannt.

Es folgt hier eine kleine Auswahl an Möglichkeiten, um den Energieverbrauch über Methoden der Optimierung zu senken. Diese Methoden werden im Rahmen des Compilerbaus eingesetzt. Die Grundidee ist hierbei, andere Befehlssequenzen zu verwenden, die schneller sind oder weniger Energie benötigen.

- Reduced Memory Access

  Externe Speicherzugriffe verursachen hohe Energiekosten. Compiler sollten den Zugriff auf den externen Speicher nach Möglichkeit vermeiden oder alternativ kostengünstigere Speicherhierarchien nutzen, wie zum Beispiel den Scratchpad-Speicher, falls dieser vorhanden ist. Ein Optimierungsverfahren zur Reduktion von Speicherzugriffen ist Registerpipelining [SS00].

- Strength Reduction

  Teure Befehle werden bei dieser Technik durch adäquate Kostengünstigere ersetzt. Ein typisches Beispiel für Strengh-Reduction ist das Ersetzen einer teuren mehrzyklischen Multiplikationsoperation durch eine billigere Shift-Left-Anweisung [SIT99].

- Instruction Reordering

  Befehle und Befehlssequenzen werden so umgruppiert, dass Schaltkreisaktivitäten minimiert werden. Die logische Abhängigkeit der Befehlssequenzen untereinander und die Datenabhängigkeit darf durch die Reorganisation nicht verletzt werden. Durch entsprechendes Instruction-Reordering werden teure externe Datenzugriffe bei Zugriffsfehlern auf Cache-Speichersysteme (Cache-Misses) und Wartezustände des Fließbands (Pipeline-Stalls) ebenfalls reduziert [SIT99].

- Data Regeneration

  Daten werden bei diesem Verfahren nicht über teure Speicherzugriffe erneut geladen, sondern kostengünstiger neu berechnet.

- Instruction Scheduling

Bei sehr hohem Registerbedarf, das heißt, es werden mehr Register als verfügbar benötigt, können durch geschickte Umordnung der Befehle Register frei werden. Dadurch kann das sonst praktizierte Spilling unter Umständen entfallen.

Neben diesen Optimierungen gibt es allerdings auch noch die sogenannten Standardoptimierungen, die von fast allen Compilern standardmäßig ausgeführt werden. Hier soll nur eine kleine Auswahl exemplarisch vorgestellt werden:

- Common Subexpression Elimination

  Wird ein Teilausdruck in verschiedenen Blöcken des Codes erkannt, so kann man auf die erneute Berechnung des Ausdrucks verzichten, indem man dem zweiten Ausdruck den Wert des zuerst errechneten zuweist, wenn sich keiner der Werte des Ausdrucks in der Zwischenzeit verändert hat.

- Dead Code Elimination

  Hierbei wird versucht, durch Codeanalysen herauszufinden, ob Codebereiche nie erreicht werden können. Solche Sequenzen können dann ersatzlos gestrichen werden.

- Schleifenoptimierungen

  Üblicherweise verbringen Programme viel Rechenzeit in Schleifendurchläufen. Aus diesem Grund erscheinen Optimierungen hier besonders sinnvoll und es sollen zwei mögliche Optimierungen genannt werden:

  - Loop Invariant Code Motion

    Hierbei wird Code falls möglich aus der Schleife herausgenommen und vor die Schleife verschoben. Dadurch erspart man sich *n-1* mal die Ausführung dieser Anweisungen, falls die Schleife *n* mal durchlaufen wird.

  - Elimination von Induktionsvariablen

    Oft wird in Schleifen die Induktionsvariable *i* für lineare Funktionen benötigt, die relative Adressen *t* einer Feldkomponente berechnet. In diesem Fall kann man, falls *i* als Abbruchbedingung genutzt wird, den Test auf *t* übertragen und *i* streichen.

3 Problembeschreibung und Formalisierung

3.1 Overlay-Techniken

Bei der Betrachtung des Problems, einen RAM-Speicher (nicht einen Cache) immer wieder neu mit unterschiedlichen Programmteilen zu laden und dann die geladenen Programme dort auszuführen, mag einige Leser an alte Overlay-Techniken erinnern. In der Tat scheint es auf den ersten Blick einige Ähnlichkeiten zu geben.

Die eigentlichen Overlay-Techniken stammen aus Zeiten, als Programme noch häufig größer waren als der real zur Verfügung stehende Arbeitsspeicher. Das bekannteste Beispiel hierfür war die Intel x86-Architektur in Verbindung mit MS-DOS, bei der man nur das erste MB an Hauptspeicher direkt adressieren konnte und dabei den Bereich über der 640Kb-Grenze sogar nur mit einigen Programmiertricks. Teile des Programms mussten also entweder auf externe Speicher wie Festplatten oder Floppys ausgelagert werden, oder wurden in einen Bereich des Speichers ausgelagert, der nicht direkt adressierbar war.

Bei den Mechanismen der Overlay-Technik wurde das Programm vom Programmierer in Teile aufgebrochen und immer nur einer oder mehrere dieser Teile befanden sich neben den Betriebssystemfunktionen und dem Programmrumpf (in der folgenden Grafik mit „0-0“ dargestellt) zu einem bestimmten Zeitpunkt im direkt adressierbaren Speicher, wo sie direkt ausgeführt werden konnte. Das Programm wurde dabei in einer baumartigen Datenstruktur, Overlay-Tree genannt, organisiert [ES01].

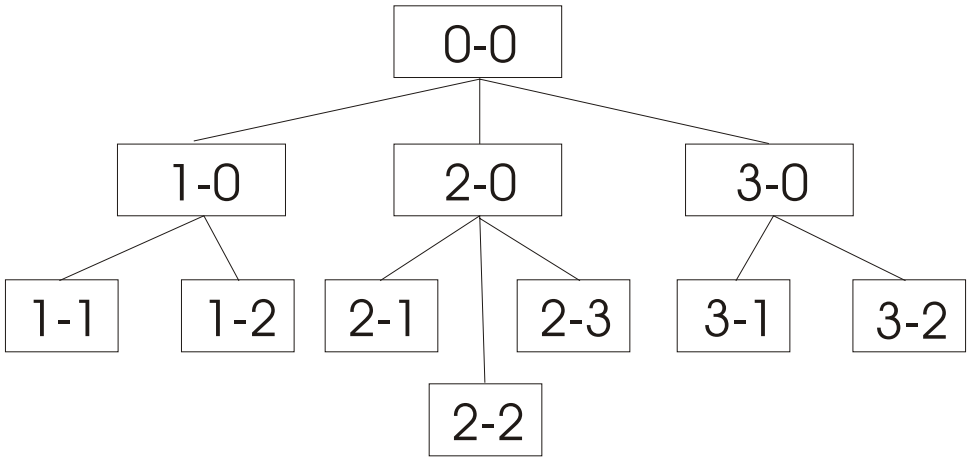

Abbildung 3.1-1 Overlay Tree

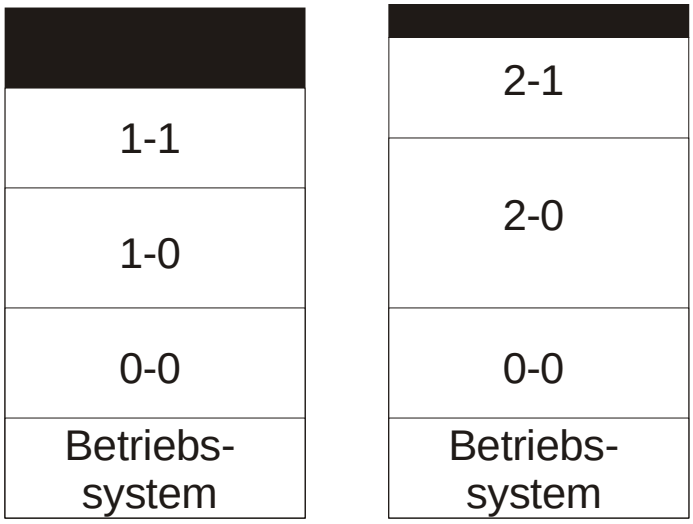

Abbildung 3.1-2 Speicheraufteilung zu zwei unterschiedlichen Zeitpunkten

Wie man an den Abbildungen 3.1-1 und 3.1-2 sieht, ist nur der Programmrumpf permanent im Speicher enthalten. Dieser Teil des Programms sorgt auch dafür, dass die anderen Teile des Programms bei Bedarf ebenfalls in den Speicher geholt werden. Im einfachsten Fall reicht es also aus, dass Programm nur in den Programmrumpf und in einzelne Teile aufzuteilen, die dann abwechselnd in den Speicher geholt werden. Damit aber nicht übermäßig oft größere Programmteile verschoben werden müssen, gab es auch damals schon die Möglichkeit, zusätzlichen Hierarchiestufen einzufügen. Z.B. so wie es in der Abbildung 3.1-1 dargestellt ist. So enthält der Block 1-0 beispielsweise Funktionen, die von den Programmteilen 1-1 und 1-2 benötigt werden. Da aber unter Umständen nicht alle drei Teile gemeinsam in den Speicher passen, kann es günstig sein, einmal die Teile 1-0 und 1-1 und zu einem anderen Zeitpunkt die Teile 1-0 und 1-2 im Speicher zu halten. Bei dieser Aufteilung könnte man sich also das zweifache Verschieben des Blocks 1-0 sparen.

MS-DOS bot zum Zweck des Einlagerns einzelner Overlays einen eigenen Softwareinterrupt (Interrupt 21, Funktion 4Bh, Unterfunktion 3), der es einem Programm ermöglichte, ein anderes Programm in den Speicher zu laden und dabei gleich den passenden Relokationsfaktor[5], um die Sprungziele korrekt zu ermitteln, zu übergeben [PC90]. Immer wenn ein anderer Teil benötigt wurde, musste der Inhalt des Speicher ersetzt werden und der vorhergehende Inhalt des Speichers wurde überschrieben. Hier besteht auch die Hauptähnlichkeit mit dem Vorgehen des hier vorgestellten Optimierungsverfahrens. Die Inhalte im Scratchpad können ebenso verdrängt werden, wie es auch in der Overlay-Technik praktiziert wird. Beiden Verfahren ist also gemein, dass der Speicherinhalt durch Software bestimmt wird und nicht durch Hardware wie im Fall eines Caches.

[5] Der Relokationsfaktor wurde benutzt, um die Sprünge in einem Programmteil richtig anzupassen, da in diesem Fall alle Sprunge relativ zu einem fixen Ausgangspunkt vorberechnet wurden. Wurde nun das Programm an eine andere Startadresse geladen, so mussten alles Sprünge um einen konstanten Offset, den Relokationsfaktor, angepasst werden.

Eine weitere Gemeinsamkeit zwischen dem dynML[6]-Verfahren und der Overlay-Technik ist die Art, wie die Programmteile angesprungen werden. In beiden Fällen werden FAR-Calls[7] benötigt (bzw. der Branch-Link-Befehl[8] im ARM-Assembler), um die Programmteile zu erreichen und wieder zurückzuspringen. Hierdurch entsteht ein Overhead, der durch die Ausführung im Scratchpad wieder wett gemacht werden muss. Im Falle einer ARM-Zielhardware erschwert sich diese Sprungberechnung noch dadurch, das der BL-Befehl nur einen relativen Sprung darstellt. Absolute Sprünge werden von dieser Hardware nicht unterstützt. Dadurch wird das später vorzustellende Programm enJumpCorrection überhaupt erst notwendig, um die richtigen Sprungziele zu errechnen.

Einer der Hauptunterschiede liegt allerdings darin begründet, dass bei der Overlay-Technik der Zwang zum Kopieren der Programmblöcke besteht. Die Programmteile können nur in einem Speicherort ausgeführt werden. Der normale Speicher wird also wie eine Art Cache genutzt. Bevor Programmteile ausgeführt werden können, müssen sie unbedingt in den entsprechenden Speicher geladen werden. Bei der Overlay-Technik entspricht dies entweder einem Kopieren von der Festplatte in den normalen Speicher oder von einem Speicherbereich, der nicht direkt adressiert werden kann in einen normal adressierbaren Speicher. Analog geschieht dies bei der Verwendung eines Caches, da dort alle Inhalte vor dem Ausführen aus dem normalen Speicher in den eigentlichen Cache geladen werden müssen. Auf die Ausnahme von non-cacheable-areas, Speicherbereichen die nicht in den Cache verschoben werden sollen, soll hier nicht weiter eingegangen werden.

Bei der Verwendung des Scratchpads unterliegt man dieser Einschränkung nicht. Der Scratchpad ist fest an eine Stelle im direkt erreichbaren Adressbereich gemappt und kann dann wie ein ganz normaler Speicher benutzt werden, wodurch die unbedingte Notwendigkeit für ein Verschieben vor dem eigentlichen Ausführen nicht mehr besteht.

Zusätzlich ist es auch schwer, Erkenntnisse aus der Zeit der Overlay-Techniken für das hier vorliegende Problem nutzbar zu machen. Damals unterlag das Management des Ein- und Auslagerns fast ausschließlich dem Programmierer, der sich selber darum kümmern musste, welche Blöcke gerade im Speicher benötigt wurden. Aufgrund der geringen Rechenleistung und der Komplexität des Problems gab es keine real eingesetzten Algorithmen, die dem Programmierer diesen Job abnehmen oder vereinfachen konnten. Ein Zitat aus der Microsoft-Dokumentation zu der von ihnen entwickelten Overlay-Bibliothek macht dies mehr als deutlich:

---

[6] Im Folgenden wird das im Laufe dieser Diplomarbeit erarbeitete Verfahren zur dynamischen Nutzung des Scratchpads kurz mit „dynML“ (=dynamischer memory locator) bezeichnet.

[7] Bei der x86-Architektur in Verbindung von MS-DOS wurde der Speicher in 64Kb große Blöcke organisiert. Sprünge über diese Blockgrenzen hinaus wurden als FAR-Calls bezeichnet. Der FAR-Call besitzt dabei eine große Ähnlichkeit zum BL-Befehl der ARM-Architektur.

[8] Im Folgenden kurz: BL-Befehl

„Producing a good overlay structure requires lengthy and tedious trial-and-error work. As new capabilities are added to your program, the structure quickly becomes obsolete. Programmers working on a large system that contains hundreds of source files and thousands of functions often spend as much time tuning the overlay structure as they do writing code.“ [MS92].

Erst sehr spät, als die Notwendigkeit für Overlays aufgrund von neu entwickelter Hard- und Software, die diesen Einschränkungen nicht mehr unterlag, schon fast nicht mehr bestand, wurden Verfahren und Algorithmen eingeführt, die die Aufgaben des Programmierers zumindest teilweise übernehmen konnten. Turbo Pascal z.B. unterstützte den Programmierer erst ab der Version 6.0, die 1990/91 erstmals vorgestellt wurde, mit dem sogenannten VROOMM (Virtual Realtime Object-Oriented Memory Manager) [BO02]. Selbst bei der Benutzung dieses Managers musste der Programmierer selber festlegen, welche Bereiche zu einem Overlay-Teil gehören sollten. Die Aufteilung unterlag also immer noch dem Programmierer. Lediglich um das eigentliche Ein- und Auslagern der Overlay-Blöcke musste man sich nicht mehr selber kümmern. Diese Aufgabe übernahm der Manager unter zu Hilfenahme einer LRU-Strategie (Least recently used). Dieses Verfahren wurde dabei damals als besonders fortschrittlich und „ausgefeilt“ beworben.

Das hier zu entwickelnde Verfahren zur dynamischen Nutzung des Scratchpads soll in seiner Funktionalität weitergehen als dies die damals üblichen Overlay-Tools je geleistet haben. Automatisiert sollen bei diesem neuen Verfahren die Entscheidungen getroffen werden, zu welchen Zeitpunkten einzelne Programmteile in den Scratchpad verschoben und welche Programmteile verdrängt werden sollen. Entwicklungen aus der Zeit der Overlay-Techniken sind dabei nicht anwendbar.

3.2 Bisherige Untersuchungen

Bei den bisherigen Untersuchungen, die die Grundlage für diese Diplomarbeit darstellen, handelt es sich hauptsächlich um die Diplomarbeit mit dem Titel „Energieeinsparung durch compilergesteuerte Nutzung eines On-Chip Speichers“ von Zobiegala [ZO01] und einem Konferenzbeitrag von Steinke et al. [SWB02].

In dieser Arbeit wurde durch die statische Nutzung des Scratchpads für Daten und Programmteile versucht, den Energiebedarf von Programmen zu senken. Mit statischer Nutzung ist hier gemeint, dass Daten und Programmteile beim Programmstart direkt in den Scratchpad geladen werden und dann dort an unveränderter Stelle bis zum Programmende verbleiben.

Das Problem, welche Programmteile verschoben werden sollen, ist dabei identisch mit dem bekannten Knapsackproblem. Dieses lässt sich mit einem Branch-and-Bound Algorithmus lösen, oder einfach in ein Problem der linearen Integer Programmierung, das in Kapitel 3.6 genauer beschrieben wird, umformen, das dann mit entsprechenden Programmen, sogenannten ILP-Solvern, sehr effizient gelöst werden kann.

Die Programmausführung vollzieht sich in diesem Fall, wie in der folgenden Abbildung dargestellt:

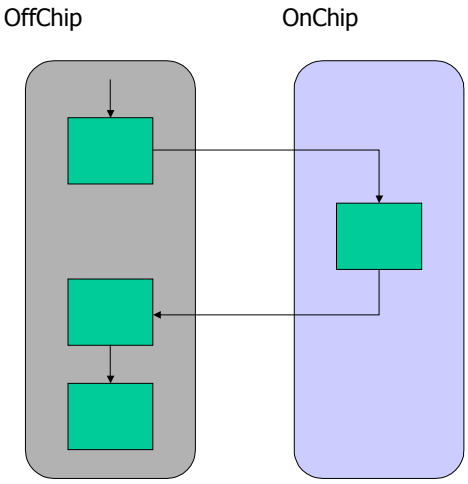

Abbildung 3.2-1 statische Nutzung des Scratchpads

Als Ergebnis der Arbeit von Zobiegala [ZO01] ist eine bereits erhebliche Energiereduktion von bis zu 80% im Vergleich zu einem System ohne Scratchpad oder Cache bei gleichzeitiger Performancesteigerung zu nennen. Nicht unerwähnt darf dabei bleiben, dass diese zugegebenermaßen sehr hohe Reduktion des Energiebedarfs darin resultiert, dass bereits das gesamte Programm mit allen Daten und einschließlich dem Stack in den Scratchpad passte. An dieser Stelle ist dann auch leicht einzusehen, dass ein dynamisches Verfahren, dass versucht den Scratchpad mit verschiedenen Inhalten zu unterschiedlichen Zeiten effizienter zu nutzen, in diesem Fall nicht besser abschneiden kann. Nur wenn das Programm und die Daten wesentlich größer sind als der zur Verfügung stehende Scratchpad-Speicher, kann ein solches dynamisches Verfahren bessere Werte erzielen.

Dieser für das statische Verfahren sehr günstige Fall, dass das gesamte Programm in den Speicher passt, ist aber bestimmt nicht der Allgemeinfall, handelt es sich bei dem Scratchpad doch um einen kleinen, prozessornahen Zwischenspeicher. In realen Anwendungen wird das Größenverhältnis zwischen Programm- und Datengröße und der Scratchpadgröße wesentlich ungünstiger ausfallen. Hier ist dann wohl eher ein Faktor von 10/1 oder sogar 100/1 anzunehmen.

In der folgenden Abbildung wird schematisch dargestellt wie ein dynamisches Verfahren an verschiedenen Stellen (hier als Kopierfunktion gekennzeichnet) die Inhalte im Scratchpad verändern kann. Die beiden Blöcke auf der rechten Seite der Grafik im On-Chip-Bereich sind so zu verstehen, dass sie den gesamten Scratchpad mit unterschiedlichen Inhalten zu unterschiedlichen Zeitpunkten darstellen.

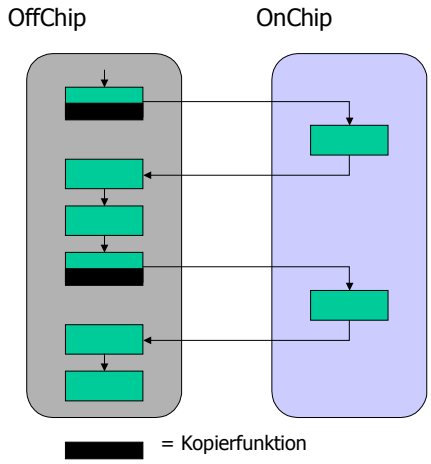

Abbildung 3.2-2 dynamische Nutzung des Scratchpads

Aus dieser Überlegung scheint die Untersuchung eines dynamischen Verfahrens sehr sinnvoll.

3.3 Vorüberlegungen

Wie im vorangegangenen Kapitel schon angesprochen wurde, soll der zur Verfügung stehende Scratchpad-Speicher dynamisch zur Programmlaufzeit mit immer neuen Inhalten genutzt werden.

Im folgenden sollen nun Überlegungen vorgenommen werden, welche Verfahren für den Austausch der Inhalte des Scratchpads am erfolgversprechendsten sind oder überhaupt erst in Frage kommen. Diese Vorüberlegungen sind notwendig, da es sich bei diesem Problem um ein Problem mit vielen Freiheitsgraden in der möglichen Realisierung handelt.

Auch wenn das hier zu lösenden Problem zunächst an „alte[9]“ Overlay-Techniken erinnert, so ist diese Tatsache hier nicht sehr hilfreich, wie aus verschiedenen Gründen schon in den Grundlagen in Kapitel 3.1 versucht wurde darzulegen.

Die folgende Grafik soll die dennoch vielfältigen Freiheitsgrade darstellen:

[9] „alt“ in dem Sinne, dass der Haupteinsatzbereich der Overlay-Techniken bereits in den frühen 80er Jahren lag. Damals waren die Rechnerleistungen noch nicht ausreichend, um rechenintensive Optimierungen durchführen zu können und aus diesem Grund musste meistens der Programmierer „von Hand“ die Entscheidungen für die Overlays treffen.

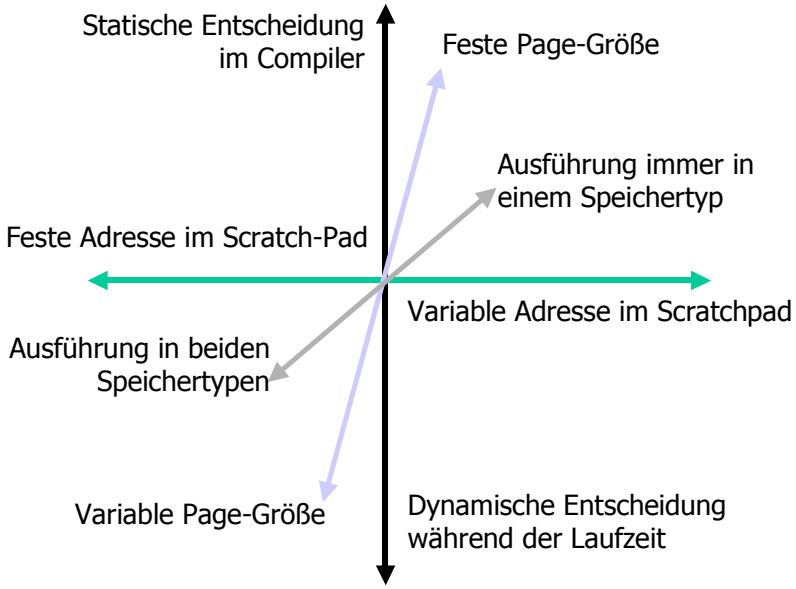

Abbildung 3.3-1 Freiheitgrade in der Realisierung

Die einzelnen Dimensionen sollen nun detailliert beleuchtet werden:

Entscheidungszeitpunkt

- Dynamische Entscheidung zur Laufzeit

  Bei einer dynamischen Entscheidung zur Laufzeit, müsste das zu compilierende Programm um die Funktionalität einer Speicherverwaltung eines Betriebssystems erweitert werden. Für die Entscheidung über die Auslagerung könnte man dabei eine Art von Branch-Prediction in Verbindung mit einer Control-Flow-Analyse verwenden.

  Gegen diese Vorgehensweise spricht allerdings der immense Overhead während der Laufzeit. Es müssten Analysen zur Laufzeit durchgeführt werden und auch die Sprunganpassungen müssten zur Laufzeit dynamisch angepasst werden.

  Dieser Overhead wird den möglichen Gewinn für relativ kleine Speicherblöcke mit Sicherheit überschreiten und selbst bei größeren Blöcken, bei denen ein größerer Energiegewinn erzielbar wäre, ist mit hoher Wahrscheinlichkeit kein positives Ergebnis mehr zu erzielen.

- Statische Entscheidung im Compiler

  Bei einer statischen Entscheidung im Compiler würde bereits während des Compilierens darüber entschieden, welche Teile des Programms zu welchen Zeitpunkten in den Scratchpad kopiert werden sollen. Die Kopierfunktionen könnten so statisch in das Programm integriert werden. Diese Vorgehensweise erzeugt einen erheblich geringeren Overhead im Vergleich zur Entscheidung während der Laufzeit.

  Der Nachteil dieser Vorgehensweise ist die Datenunabhängigkeit der Entscheidung.

Positionierung im Speicher

- Variable Adresse im Scratchpad

  Sollte dasselbe Speicherobjekt zu unterschiedlichen Zeiten an unterschiedlichen Stellen im Scratchpad positioniert werden, so würde dies eine Sprungtabelle notwendig machen, in die die jeweils gültige Speicheradresse eingetragen werden müsste. Dadurch entsteht ein zusätzlicher Overhead, da die Tabelle immer aktualisiert werden muss. Zudem muss bei jedem Zugriff auf ein Objekt in dieser Tabelle die gerade gültige Adresse ermittelt werden. Neben diesem Programmoverhead entsteht noch der Speicheroverhead durch die zwingend notwendige Sprungtabelle.

- Feste Adresse im Scratchpad

  Bei einer festen Adressierung eines Blockes entsteht ein wesentlich geringerer Overhead. Als Nachteil könnte allerdings die Ausnutzung des Scratchpads geringer ausfallen, falls es zu einer Fragmentierung des Scratchpads kommt.

Ausführungsort

- Ausführung in beiden Speichertypen

  Bei einer Ausführung in beiden Speichertypen könnte auf ein bestimmtes Memoryobjekt einmal im Scratchpad und zu einem anderen Zeitpunkt im Off-Chip-Speicher zugegriffen werden. Dadurch erreicht man eine höhere Flexibilität, die hier nicht durch zusätzlichen Overhead zur Programmlaufzeit erkauft werden muss.

  Diese erhöhte Flexibilität kann dabei durchaus in einem besseren Ergebnis im Hinblick auf den Energieverbrauch resultieren.

  Der Nachteil hier liegt lediglich in einem größeren Aufwand bei der Implementierung eines Algorithmus innerhalb des Compilers, der für ein und dasselbe Speicherobjekt zu unterschiedlichen Zeitpunkten unterschiedliche Entscheidungen über das Verlagern in den Scratchpad treffen muss.

- Ausführung nur in einem Speichertyp

  Der einzige Vorteil bei dieser Vorgehensweise wäre eine vereinfachte Implementierung eines Algorithmus. Overheadeinsparungen sind durch die Einschränkung, dass ein Speicherobjekt immer in einer bestimmten Speicherform ausgeführt werden muss, nicht möglich.

Speicherorganisation des Scratchpads

- Feste Page-Größe

  Eine feste Page-Größe für den Scratchpad, so wie es zum Beispiel bei der Speicherverwaltungsart des Pagings in manchen Realzeit-Betriebssystemen [RL94] eingesetzt wird, bringt für dieses Problem keine Vorteile. Ganz im Gegenteil würde durch das Einfügen eines Pagingsystem der Overhead unnötig vergrößert werden, da zusätzliche Tabellen mit der Beschreibung des Inhalts jeder Page eingefügt werden müssten. Zur Laufzeit ergibt sich somit ein zusätzlicher Speicheraufwand für die Tabelle,

aber auch ein zusätzlicher Programmaufwand, da die Tabelle ausgelesen und ausgewertet werden muss. Zusätzlich entsteht durch eine Partitionierung in feste Page-Größen das Problem der internen Fragmentierung[10].

- Variable Page-Größe

  Bei einer Variablen Page-Größe könnte als Problem die externe Fragmentierung[11] entstehen. Durch eine geschickte Wahl der Austauschzeitpunkte kann diese Fragmentierung hier allerdings vermieden werden. Ansonsten wird aber kein zusätzlicher Overhead durch die Verwendung von variablen Page-Größen eingefügt.

Als Ergebnis dieser Vorüberlegungen bleibt festzuhalten, dass, gerade wenn man sich den zum Teil erheblichen Overhead der einzelnen Methoden vor Augen hält, es nur eine sinnvolle Realisierung zu geben scheint. Für die einzelnen Entscheidungsachsen kommt man zu dem Ergebnis:

Eine dynamische Entscheidung zur Laufzeit scheidet aufgrund des zu hohen Overheads aus und eine **statische Entscheidung bereits im Compiler** ist zu bevorzugen. Ebenso ist eine Einführung von festen Page-Größen nicht sinnvoll. Ein **frei adressierbarer und nutzbarer Scratchpad** ist also zu bevorzugen. **Verschobene Speicherobjekte** sollten dabei immer an **die gleiche Stelle im Scratchpad** kopiert werden, damit nicht noch zusätzliche Sprungtabellen eingefügt werden müssen. Bei der Entscheidung über den **Ausführungsort**, ob ein **Speicherobjekt** nur in einer Speicherform, oder aber **in beiden erreichbar** sein soll, sollte die flexiblere Variante gewählt werden, auch wenn dies einen erhöhten Implementierungsaufwand bedeutet. Dadurch kann unter Umständen ein besseres Ergebnis im Hinblick auf den Energiebedarf des Programms erreicht werden.

[10] Bei festen Pagegrößen entsteht immer das Problem, dass die letzte Page, die von einem Programmteil benötigt wird, nur sehr selten zu 100% genutzt werden kann. Hat man z.B. eine feste Pagegöße von 32 Bytes und will darin einen Block der Größe 50 Bytes speichern, so benötigt man 2 Pages. Die erste Page wird dabei voll genutzt, die zweite Page allerdings nur zu einem kleineren Anteil, da ja nur noch 18 Bytes benötigt werden. 14 Bytes bleiben so ungenutzt. Dieser Umstand wird interne Fragmentierung genannt.

[11] Bei variablen Pagegrößen entsteht das Problem, dass einzelne Blöcke zwar auch nur immer den Platz im Speicher benötigen der ihrer Größe entspricht, allerdings durch Blöcke mit unterschiedlicher Größe ersetzt werden. Wenn Blöcke nicht mehr benötigt werden und wieder aus dem Speicher entfernt werden, entsteht eine Lücke in der entsprechenden Größe. Da jeder Block eine unterschiedliche Größe haben kann, kann es sein, dass ein neuer Block diese Lücke nicht vollständig ausfüllen kann. Dieser Umstand wird externe Fragmentierung genannt.

3.4 Definitionen

Definition Basicblock:

Ein Basicblock ist eine Menge von Instruktionen, die stets hintereinander ausgeführt werden. Sprünge von anderen Basicblöcken zu einer der Instruktionen sind nur zur ersten Instruktion eines Basicblockes zulässig. Wenn die erste Instruktion kein Sprungziel ist, muss die letzte Instruktion des vorherigen Basicblockes ein Sprungbefehl sein. Der Basicblock endet mit einem Sprungbefehl als letzter Instruktion oder ist dadurch begrenzt, dass die erste Instruktion, die diesem Basicblock nachfolgt, selbst ein Sprungziel ist.

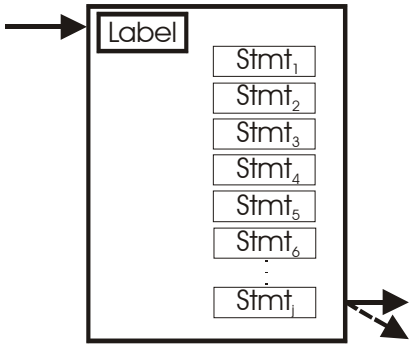

Abbildung 3.4-1 Schematische Darstellung eines Basicblocks

Definition Superblock:

Ein Superblock besteht aus einem oder mehreren Basicblöcken und hat die Eigenschaft, dass der Kontrollfluss von außerhalb stets an genau einer Stelle im Superblock beginnt. Beim Verlassen des Superblocks gibt es nur genau eine nachfolgende Instruktion.

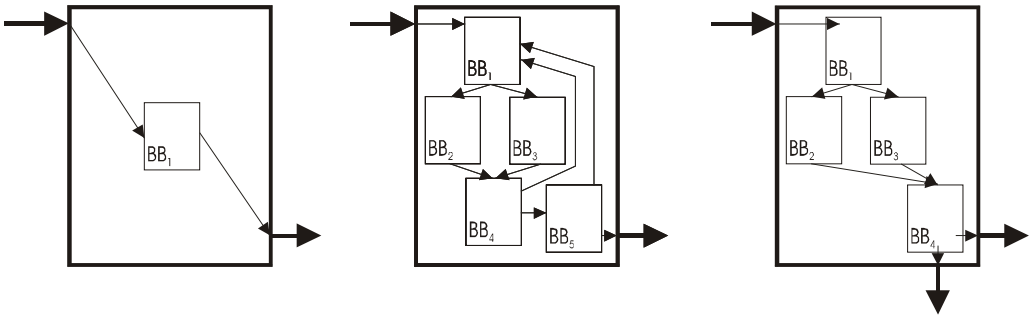

Abbildung 3.4-2 Links: minimaler Superblock, Mitte: komplexer Superblock, Rechts: kein Superblock

Definition Minimaler Superblock:

Bei einem minimalen Superblock handelt es sich um einen Superblock, der lediglich die minimale Anzahl von Basicblöcken enthält, die zusammen die Superblockbedingung erfüllen. In Abb. 3.4-2 (links) sieht man, dass auch ein einzelner Basicblock die Superblockbedingung schon erfüllen kann. Es kann sich aber ebenso um eine Kombination aus mehreren Basicblöcken handeln.

Definition Komplexer Superblock:

Bei einem komplexen Superblock handelt es sich um eine Kombination aus einem oder mehreren Superblöcken und einzelnen möglichen Basicblöcken. Die Bedingung, dass der Kontrollfluss von außerhalb stets an einer genauen Stelle im Superblock beginnt und es beim Verlassen nur genau eine nachfolgende Instruktion gibt, bleibt dabei erhalten. In Abb. 3.4-2 (mitte) sieht man ein Beispiel für einen komplexen Superblock. Dieser Superblock erfüllt nach außen also die Superblockbedingung. Intern besteht er mit BB_2 und BB_3 aus zwei minimalen Superblöcken und mit BB_1, BB_4 und BB_5 aus drei normalen Basicblöcken.

3.5 Lösungsansatz

Zum Einlagern von Programmteilen in den Scratchpad werden an verschiedenen Punkten im Programm sogenannte Kopierfunktionen eingefügt. Es gilt nun, die optimalen Positionen dieser Kopierfunktionen zu berechnen und die optimalen Programmteile, die von den jeweiligen Kopierfunktionen kopiert werden sollen, zu bestimmen.

3.5.1 Wahl der auszutauschenden Blöcke

Ein Einlagern von Basicblöcken ist nur sinnvoll, wenn die eingelagerten Programmteile danach mehrfach ausgeführt werden. Diese Forderung ist sehr leicht einzusehen, da ein Einlagern bedeutet, dass der Basicblock in einer Kopierfunktion geladen werden muss und dann anschließend im Scratchpad wieder gespeichert werden muss. Mit diesem Vorgehen ist ein Energieaufwand verbunden, der nicht mit der einmaligen Ausführung des Programmteils im Scratchpad aufgefangen werden kann, da eine einmalige Ausführung des Programmteils außerhalb des Scratchpads lediglich ein einmaliges Lesen in der Instructionfetch-Phase des Prozessors bedeutet. Der Overhead durch das Kopieren in den Scratchpad lohnt sich daher nur, wenn die verschobenen Programmteile häufiger aufgerufen werden als die dazugehörige Kopierfunktion.

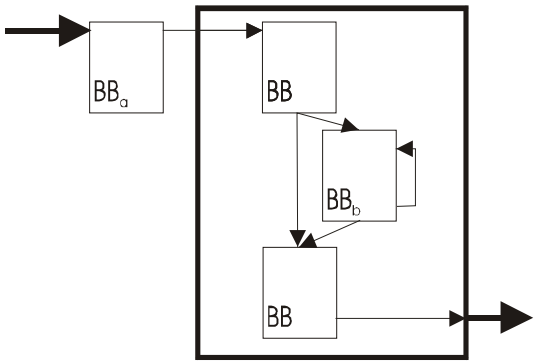

Abbildung 3.5-1 Kopierfunktion mit Superblock

Aus diesem Umstand ergibt sich, dass für eine optimale Lösung eine Kopierfunktion Basicblöcke aus nachfolgenden Schleifenkörpern verschieben muss. Wenn die Kopierfunktion in einem Basicblock *a* und ein kopierter Basicblock *b* betrachtet wird, muss die Ausführungshäufigkeit von *b* größer sein als die von *a*. Die Ausführungshäufigkeit eines dem Basicblock *a* nachfolgenden Blockes *b* ist nur dann höher, wenn der Kontrollfluss zusammengeführt wird. Da sichergestellt werden muss, dass die Kopierfunktion *a* vor der Ausführung von *b* durchlaufen werden muss, ist ein Zusammenfluss des Kontrollflusses z.B. nach einem if-Befehl nicht ausreichend. Es muss sich daher um eine Rückkante handeln, die z.B. durch Schleifen generiert wird. Die andere Ursache von Rücksprüngen sind GOTO-Befehle, die hier aus Gründen des Programmierstils nicht näher betrachtet werden sollen.

3.5.2 Position der Kopierfunktionen

Ohne Einschränkung der Optimalität genügt genau eine Kopierfunktion an jedem Schleifenanfang. Da die Kopierfunktion *a* vor dem Schleifenkörper steht und keinen Einfluss auf das ausgeführte Programm hat, kann sie auch als letzter Basicblock vor dem Schleifenkörper eingefügt werden.

Es wurde hiermit also gezeigt, dass es vollkommen ausreicht, bei einem Programm mit *n* Schleifen ebenfalls *n* Kopierfunktionen jeweils direkt vor den Schleifen einzufügen. Trotzdem kann es sein, dass ein Basicblock aus einer inneren Schleife in der Kopierfunktion einer äußeren Schleife in den Scratchpad hineinkopiert wird.

3.6 Programmanalyse

Der Energievorteil durch das Kopieren eines Basicblocks *b* in den Scratchpad durch die Kopierfunktion *a* errechnet sich wie folgt:

Gleichung 3.6-1 Energievorteil durch das dynamische Verschieben von Programmteilen

$$E_{adv}(a,b) = acc(b)\cdot instrcount(b)\cdot\left(fetch\cos t_{off} - fetch\cos t_{on}\right) - acc(a)\cdot copy\cos t\cdot instrcount(b) - l\cdot E_{jump}$$

Die einzelnen Faktoren der Gleichung im Detail:

- acc(x): Anzahl der Ausführungshäufigkeiten des Basicblocks *x*.
- instrcount(x): Anzahl der Instruktionen im Basicblock *x*.
- $fetchcost_{off}$: Energiekosten für den Instructionfetch einer Instruktion aus dem Off-Chip.
- $fetchcost_{on}$: Energiekosten für den Instructionfetch einer Instruktion aus dem On-Chip.

Die Fetchkosten für On-Chip und Off-Chip schließen auch den Energieeffekt durch unterschiedliche vom Prozessor benötige Zyklenanzahlen mit ein.

- copycost: Energiekosten für das Kopieren eines Befehls aus dem Off-Chip- in den On-Chip-Speicher.
- l: Anzahl der benötigten zusätzlichen Sprünge
- E_{jump}: Energiebedarf für einen Sprungbefehl

Für das zu optimierende Programm kann nun ein Kontrollflussgraph generiert werden. Für jede Schleife wird ein Superblock gebildet, der genau diese Schleife umfasst.

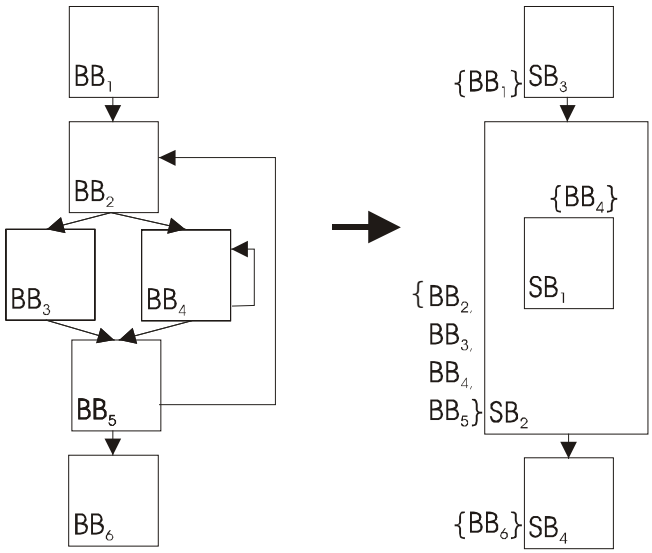

Abbildung 3.6-1 Generierung von Superblöcken

In der Abbildung 3.6-1 sieht man, wie die Basicblöcke der linken Seite zu Superblöcken auf der rechten Seite zusammengefasst werden. Die Basicblöcke BB_1, BB_3, BB_4 und BB_6 bilden minimale Superblöcke. Da BB_3 allerdings keine Schleife beinhaltet interessiert er nicht alleine für das weiter Vorgehen. Erst als komplexer Superblock in Verbindung mit den Basicblöcken BB_2 ,BB_4 und BB_5 muss er weiter analysiert werden.

In diesem Beispiel gibt es eine innere Schleife, die ausschließlich aus Basicblock 4 besteht. Diese Schleife wird zum minimalen Superblock 1. Die äußere Schleife, die aus den Basicblöcken 2 bis 5, besteht wird zu dem komplexen Superblock 2. Da auch einzelne Basicblöcke einen Superblock darstellen können, werden auch die Basicblöcke 1 und 6 jeweils zu einem Superblock überführt. Da aber keine Schleife enthalten ist, wird in diesem Fall auch keine Kopierfunktion benötigt, da es keine Blöcke geben kann, für die sich ein Verschieben in den Scratchpad lohnen könnte.

Schleifen, die nicht die Single-Entry Bedingung erfüllen, können in der Programmiersprache C nur durch GOTO-Befehle erzeugt werden. Diese werden hier, wie schon erwähnt, nicht betrachtet. Schleifen, die nicht die Single-Exit Bedingung erfüllen, z.B. durch einen break-Befehl, führen trotzdem zu genau einem nachfolgenden Befehl, der immer ausgeführt wird. Andere Kontrollstrukturen können nur durch GOTO-Befehle realisiert werden. Die Einschränkung auf Single-Entry und gemeinsamen nachfolgenden Basicblock ist daher keine wesentliche Einschränkung.

Für die Aufteilung des Scratchpads zur Nutzung durch unterschiedliche Basicblöcke muss nun die zeitliche Abfolge berücksichtigt werden. Kopierfunktionen von Superblöcken, die ineinander verschachtelt sind, müssen sich für die kopierten Programmteile auch den Scratchpad teilen.

Es muss nun ermittelt werden, welche Basicblöcke in den jeweiligen Kopierfunktionen der Superblöcke, zu denen sie gehören, kopiert werden.

Für die Auswahl innerhalb eines Superblocks, der ja genau eine Kopierfunktion hat, kann auf das Verfahren für die statische Auswahl von Basicblöcken zurückgegriffen werden.

Nachfolgend wird ein ILP-Modell vorgestellt, welches die optimale Lösung für das dynamische Einlagern von Basicblöcken findet.

3.7 Aufbau eines ILP-Modells

3.7.1 Das Modell

Für das Modell sollen zunächst die folgenden Symbole definiert werden:

- BB_x: Basicblock *x* des betrachteten Programms.
- sel_x: Ist genau dann „1", wenn das Memory-Objekt *x* in den Scratchpad kopiert werden soll (binäre Variable).
- $Copyfkt_y$: Kopierfunktion des Superblocks *y*.
- MO_z: Memory-Objekt mit einem oder mehreren Basicblöcken.
- $MemberOfCopyFkt_x(y)$: Genau dann „1", wenn in der Kopierfunktion $CopyFkt_x$ das Memory-Objekt *y* kopiert wird.
- $MemberOfMO_y(x)$: Genau dann „1", wenn der Basicblock *x* Bestandteil des Memory-Objekts *y* ist.
- size(x): Größe des Memory-Objekts *x* im Scratchpad.
- spsize: Größe des zur Verfügung stehenden Scratchpads.

Mit Hilfe dieser Symbole lässt sich nun das eigentliche Modell aufstellen.

Die zu maximierende Zielfunktion des ILP-Modells lautet nun:

Gleichung 3.7-1 Zielfunktion des ILP-Modells

$$\sum_a \sum_x MemberOfCopyFkt_a(x) \cdot E_{adv}(a,x)$$

Die Menge, der innerhalb eines Superblocks im Scratchpad liegenden Blöcke, darf die Scratchpad-Grösse *spsize* nicht überschreiten. Daher muss für jeden Superblock *y* der obersten Hierarchiestufe genau ein Constraint generiert werden. Superblöcke, die innere Schleifen repräsentieren, müssen sich den Scratchpad ebenfalls mit den äußeren Superblöcken teilen und zusammengenommen die Gesamtgröße des Scratchpads nicht überschreiten:

Gleichung 3.7-2 Constraints, um die maximale Scratchpadgröße nicht zu überschreiten

$$\sum_{x \subseteq y} \sum_z MemberOfCopyFkt_x(z) \cdot size(z) \leq spsize$$

Jeder Basicblock *x* darf nur in einem der ausgewählten Memory-Objekte enthalten sein:

Gleichung 3.7-3 Constraints um Blöcke nicht doppelt zu verschieben

$$\sum_y MemberOfMO_y(x) \leq 1$$

Hiermit wurde ein vollständiges ILP-Modell aufgestellt, welches die optimale Lösung zum dynamischen Einlagern von Programmteilen in den Scratchpad ermittelt.

3.7.2 CPLEX – ILP Solver

Bei dem Programm CPLEX von der Firma ILOG Inc. handelt es sich um einen ILP- bzw. LP Solver[12]. Dieses Programm soll auch hier eingesetzt werden, um die Entscheidungen über die Austauschzeitpunkte und die Austauschblöcke zu treffen.

Die Übergabe der Daten des ILP-Modells an das CPLEX-Programm erfordert ein geeignetes Eingabeformat und eine geeignete Schnittstelle zum enCC-Compiler. Dazu soll das LP-Dateiformat verwendet werden, welches 14 Regeln umfasst. Die wichtigsten Regeln hierbei, die zur Erzeugung der Eingabedatei benutzt werden sollen, werden hier kurz erläutert:

- Die Datei muss mit einem der folgendem Schlüsselworten beginnen:
  - „MINIMIZE“ oder „MINIMUM“
  - „MAXIMIZE“ oder „MAXIMUM“
  Mit Hilfe dieser Wörter wird die Zielfunktion angegeben.
- Variablen dürfen nicht mehr als 16 Buchstaben haben und mit einer Zahl beginnen.
  Durch diese Regel müssen die benötigten Variablen in ein geeignetes Format überführt werden, da ja einzelne Basicblöcke an verschiedenen Positionen ausgetauscht werden können und auch Bestandteil einer Kombination von Basicblöcken sein kann. Für eine genauere Beschreibung der Variablennamenerzeugung soll hier auf das Kapitel 4.5.1 verwiesen werden.
- Der Constraint-Bereich wird durch die Zeichensequenz „SUBJECT TO“ begonnen.
- Variablen, die binäres Format haben, müssen mit dem Schlüsselwort „BINARY“ gekennzeichnet werden.
- Das Ende der Datei schließlich wird mit der Zeichenfolge „END“ abgeschlossen.

[12] Programm zum Lösen von Problemen der linearen (Integer) Programmierung - (Integer) Linear Programing

Im folgenden nun ein Beispiel für eine vom Compiler erzeugte LP-Datei:

```
Maximize
      obj : 492946SB2_2 + 5946SB2_3 + 492921SB2_4 + 12507SB4_1 + ...
Subject To
      44SB2_2 + 12SB2_3 + 40SB2_4 <= 50
      …
      32SB8_1 + 128SB8_2 + 28SB8_4 + 100SB8_5 <= 50
      SB2_2 <= 1
      SB2_2 + SB2_4 <= 1
      …
      SB8_2 + SB8_5 <= 1
      SB8_4 <= 1

Binary
      SB2_2
      …
      SB8_2
      SB8_4
      SB8_5
END
```

Abbildung 3.7-1 (gekürztes) Beispiel einer LP-Datei

Die Einbindung des Programms CPLEX kann somit mit Hilfe dieser Schnittstelle einfach über einen Systemcall erreicht werden. Benötigte Befehlseingaben werden dabei aus einer dafür vorgesehen Datei durch Eingabeumleitung eingelesen.

Diese Datei sieht wie folgt aus:

```
read myCplex.lp              // Einlesen der ILP-Beschreibungs-Datei
set timelimit 900            // Übergabe eines (optionalen) Timeouts
mipopt                       // Starten des CPLEX Lösungsalgorithmus
display solution var -       // Ergebnisdarstellung
quit                         // CPLEX verlassen
```

Abbildung 3.7-2 Befehlsübergabe an CPLEX

Nun braucht man nur noch die Ausgaben, die CPLEX während des Laufs erzeugt, in eine Datei umzuleiten und der Compiler kann daraus die Lösung einfach herauslesen und geeignet darauf reagieren. Die Analyse der Ausgabe beschränkt sich allerdings hier darauf, diejenigen Variablen, die nicht von CPLEX bei der Lösung auf „0“ gesetzt wurden, zu erkennen. Diese Variablen entsprechen dann den Basicblöcken bzw. der Kombination aus mehreren Basicblöcken, die zu einem bestimmten Zeitpunkt in den Scratchpad verschoben werden sollen. Die Rückübersetzung der Variablennamen in die realen Basicblöcken und die realen Austauschzeitpunkte wird als Implementierungsdetail ebenfalls in Kapitel 4.5.1 kurz dargestellt.

Abschließend ein Beispiel für die Ausgabe von CPLEX:

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

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

CPLEX> Problem 'myCplex.lp' read.
Read time =    0.01 sec.
CPLEX> New value for time limit in seconds: 900
CPLEX> Tried aggregator 1 time.
MIP Presolve eliminated 9 rows and 4 columns.
MIP Presolve modified 8 coefficients.
Reduced MIP has 3 rows, 5 columns, and 7 nonzeros.
Presolve time =    0.00 sec.
Clique table members: 2
Root relaxation solution time =    0.00 sec.
Objective is integral.

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt
Gap

      0     0   556279.0000     1                 556279.0000        3
*               552340.0000     0   552340.0000      Cuts:  2        4

Clique cuts applied:  1
Cover cuts applied:  1

Integer optimal solution:  Objective =    5.5234000000e+05
Solution time =    0.00 sec.  Iterations = 4  Nodes = 0

CPLEX> Variable Name           Solution Value
SB2_2                         1.000000
SB4_1                         1.000000
SB6_1                         1.000000
SB8_1                         1.000000
All other variables in the range 1-9 are zero.
CPLEX>
```

Abbildung 3.7-3 CPLEX Ausgabe

In diesem Beispiel würden also sämtliche Basicblöcken, die durch die Variablen SB2-2, SB4_1, SB6_1 und SB8_1 repräsentiert werden, an den entsprechenden Stellen verschoben. Die einzelnen Variablen der CPLEX-Lösung stellen hier also die Lösung für beide Probleme, wo Blöcke ausgetauscht und welche Blöcke verschoben werden sollen, dar.

3.8 Vorstellung des Algorithmus

Im folgenden soll der Algorithmus detaillierter an einem Beispiel vorgestellt werden. Als Beispiel wurde das Programm Heap-Sort ausgewählt. Es handelt sich dabei um eine einfache Implementierung des bekannten Heap-Sort-Sortieralgorithmus und eignet sich besonders gut, um die einzelnen Schritte des dynML-Verfahrens zu beschreiben. Bei der folgenden Beispielbetrachtung wird allerdings aus Gründen der Übersichtlichkeit nur auf einen kleinen Teil des oben genannten Programms in abstrahierter und verallgemeinerter Weise eingegangen.

Das Vorgehen des Algorithmus, der die bestmögliche dynamische Ausnutzung eines Scratchpads erreicht, im zeitlichen Ablauf:

Wie in den vorangegangenen Kapiteln dargelegt wurde, lohnt sich ein Verschieben in den Scratchpad nur für Basicblöcke, die mehrfach ausgeführt werden. Dabei hängt es davon ab, wie häufig der Block verschoben werden muss, wie häufig er ausgeführt wird und wie groß dieser ist, um eine Entscheidung zu fällen, ob sich ein Verschieben überhaupt lohnt. Für diese Entscheidung lassen sich zwei Regeln festhalten:

1. Je größer ein Block ist, desto weniger häufig muss er ausgeführt werden, damit sich ein Verschieben in den Scratchpad lohnt.
2. Es lohnt sich immer weniger einen Block zu verschieben, je häufiger er verschoben werden muss.

Diese beiden Regeln lassen sich so interpretieren, dass durch das Verschieben in den Scratchpad zunächst Kosten entstehen, die dann durch die Ausführung im Scratchpad kompensiert werden müssen. Das Kopieren einzelner Basicblöcke beinhaltet dabei zwei Kostenkomponenten, auf der einen Seite konstante Kosten für alle zu verschiebenden Basicblöcke und auf der anderen Seite variable Kosten für jeden einzelnen Basicblock abhängig von seiner Größe. Die erste Regel muss dabei so interpretiert werden, dass bei kleinen Basicblöcken die konstanten Kosten einen größeren Teil der Gesamtkosten ausmachen und deshalb muss der Block auch häufiger ausgeführt werden, um im Scratchpad insgesamt gesehen einen Energiegewinn erzielen zu können. Größere Blöcke müssen dementsprechend nicht so häufig ausgeführt werden. Die zweite Regel ist einfach so zu verstehen, dass, wenn ein Block mehrmals verschoben wird, er auch mehrmals den Kopieraufwand wettmachen muss.

3.8.1 Analyse der Programmschleifen

Für den Algorithmus bedeutet dieser oben genannte Zusammenhang, dass es in einem ersten Schritt darum gehen muss, sämtliche Schleifen in dem zu compilierenden Programm ausfindig zu machen. Schleifen erfüllen dabei in der Hochsprache C die Bedingung eines Superblocks.

Da es allerdings auch Schleifen innerhalb von anderen Schleifen in einem Programm geben kann, muss der Algorithmus mit der innersten Schleife anfangen und den dortigen Superblock finden. Anschließend müssen die weiter außen liegenden Schleifen gefunden werden, die eine, oder mehrere, Schleifen

beinhalten können. Das Ergebnis dieses ersten Schritts ist eine Hierarchie von Superblöcken.

Wenn man sich nur die erste Hierarchiestufe anschaut, so ist das Ergebnis hier eine strikte Linearisierung des Programms. Es werden also Blöcke erkannt, die immer hintereinander ausgeführt werden. Die folgende Abbildung soll dies verdeutlichen:

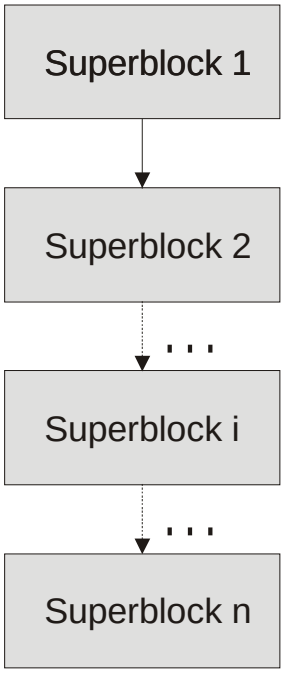

Abbildung 3.8-1 oberste Hierarchiestufe von Superblocks

Wenn das Programm also aus *n* Superblöcken der obersten Hierarchiestufe besteht, so kann man sich sicher sein, dass, nachdem ein Block *i* verlassen wurde, er nie wieder erreicht werden kann. Für die Nutzung des Scratchpads hat dies zur Folge, dass für jeden Superblock der ersten Hierarchiestufe der Scratchpad-Speicher mit der vollen Größe zur Verfügung steht. Würde ein Programm keine verschachtelten Schleifen enthalten, so könnte man mit dieser Linearisierung des Programms bereits die optimale Lösung erzielen.

Da es aber auch mehrere verschachtelte Schleifen geben kann, muss jeder dieser Superblöcke der ersten Hierarchiestufe noch einmal gesondert untersucht werden. Wie oben schon erwähnt, soll dabei ein Superblock, im Folgenden Superblock *i* genannt, aus dem Heap-Sort Benchmark hierzu verwendet werden:

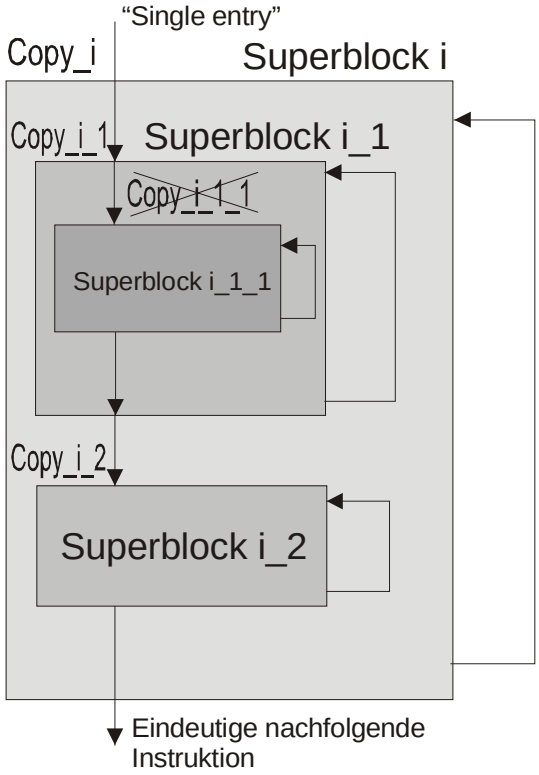

Abbildung 3.8-2 Superblock aus dem Heap-Sort Benchmark

Die Abbildung 3.7-2 soll diesen Superblock *i* darstellen. Dieser Superblock besitzt zwei innere Schleifen, die hier mit Superblock „i_1" und Superblock „i_2" gekennzeichnet wurden. Der Superblock „i_1" besitzt dabei noch eine weitere innere Schleife mit dem Superblock „i_1_1". Da dies allerdings die einzige innere Schleife darstellt, kann es sich unter keinen Umständen lohnen, eine Kopierfunktion auch für den Superblock „i_1_1" einzufügen. Nur wenn es mindestens zwei innere Schleifen der gleichen Hierarchiestufe in einem Superblock gibt (hier ist das mit Superblock i_1 und i_2 der Fall), kann es günstig sein, einzelne Teile aus den inneren Schleifen alternierend in den Scratchpad zu verschieben. Die inneren Schleifen konkurrieren also um den gleichen Platz im Scratchpad. In diesem Beispiel bedeutet das, dass alle Basicblöcke, die aus Superblock „i_1_1" ausgewählt wurden, schon in der Kopierfunktion „Copy_i_1" in den Scratchpad verschoben werden können, ohne dabei eine Einschränkung zu bedeuten.

Durch die besondere Eigenschaft des Superblocks kann man beim Eintritt in einen solchen sehr einfach eine Kopierfunktion einfügen. Für das Beispiel hier könnte es unter Umständen also lohnend sein, an drei Stellen Kopierfunktionen (copy_i, copy_i_1, copy_i_2) einzufügen. Copy_i_1_1 ist aus den oben genannten Gründen keine Alternative, da es keine konkurrierenden Inhalte der gleichen Hierarchiestufe (z.B. in einem Superblock „i_1_2") gibt.

Eine mögliche Scratchpadausnutzung könnte also wie folgt aussehen, wobei die mit copy_"x" gefüllten Flächen den Bereich im Scratchpad repräsentieren, der von dieser Kopierfunktion gefüllt wird:

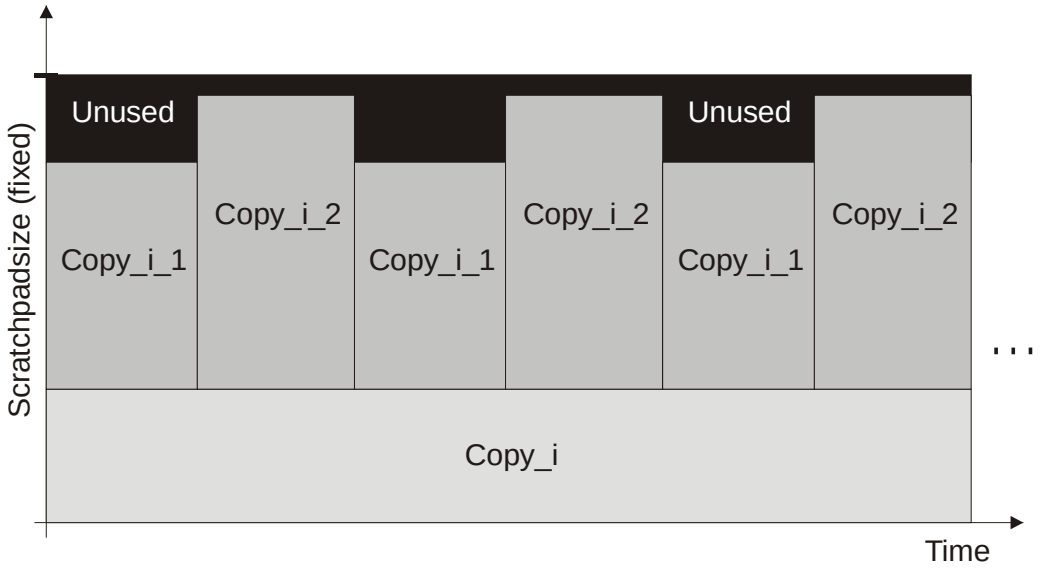

Abbildung 3.8-3 mögliche Scratchpadausnutzung

Mit diesem ersten Schritt des Algorithmus ist der erste Hauptteil bereits abgeschlossen. Es wurde eindeutig festgestellt, welche Blöcke zu welchen Austauschzeiten um einen Platz im Scratchpad konkurrieren.

3.8.2 Auswahl der Basicblöcke

Im zweiten Schritt muss es beim Algorithmus also nun darum gehen, die richtigen Basicblöcke aus jedem Superblock auszuwählen, die in den Scratchpad verschoben werden sollen. Gehen wir dabei als Beispiel von einer Zuordnung der Basicblöcke zu den Schleifen wie folgt aus:

Superblock:	**Enthaltene Basicblöcke:**
Copy_i	BB1, BB2, BB3, BB4, BB5, BB6
Copy_i_1	BB1, BB2, BB3
Copy_i_2	BB4, BB5, BB6

Tabelle 3.8-1 Beispiel: Basicblockkonstellation

Nun geht es darum, Kandidaten für jeden Superblock zu finden, und dessen möglichen Gewinn zu berechnen. Bei der Berechnung des Gewinns stößt man allerdings auf das Problem, dass es Wechselwirkungen zwischen benachbarten Basicblocks gibt. So ist der mögliche Energiegewinn unterschiedlich, wenn man zwei aufeinanderfolgende Basicblöcke zusammen betrachtet, oder jeden für sich. Betrachtet man mehrere Basicblöcke zusammen, so kann man Sprungbefehle sparen und somit einen größeren Energiegewinn erzielen, falls die Basicblöcke

keinen Funktionsaufruf, Funktionsrücksprung oder einen unbedingten Sprung enthalten. An diesen gesonderten Programmstellen, an denen kein linearer Programmablauf vorhanden ist, lohnt es sich also nicht, eine Kombination von Basicblöcken gesondert zu untersuchen. Ein Verschieben der beiden Basicblöcke BB1 und BB2 könnte aber einen größeren Gewinn bedeuten, als wenn man die Basicblöcke getrennt betrachtet.

Wenn man also davon ausgeht, dass die Basicblöcke in diesem Beispiel hintereinander von BB1 bis BB6 im Speicher liegen und keiner der oben genannten Sonderfälle zutrifft, so kommt man zu folgenden Kandidaten, für die ein möglicher Gewinn berechnet werden muss:

Basicblock:	**Kandidaten:**
Copy_i	BB1, BB2, BB3, BB4, BB5, BB6, {BB1,BB2}, {BB1-BB3}, … {BB1-BB6}, {BB2,BB3}, {BB2-BB4}, ... {BB2-BB6}, {BB3,BB4}, {BB3-BB5}, {BB3-BB6}, {BB4,BB5}, {BB4-BB6}, {BB5,BB6}
Copy_i_1	BB1, BB2, BB3, {BB1,BB2}, {BB1,BB2,BB3}, {BB2,BB3}
Copy_i_2	BB4, BB5, BB6, {BB4,BB5}, {BB4,BB5,BB6}, {BB5,BB6}

Tabelle 3.8-2 Beispiel: mögliche Kandidaten

An diesem einfachen Beispiel erkennt man schon, dass die Anzahl möglicher Kandidaten, die untersucht werden muss, wesentlich größer ist, als die Anzahl der im Superblock enthaltenen Basicblöcke.

Wenn man nun allerdings die Energiewerte für alle Kandidaten berechnet hat, kann man sehr einfach, wie in Kapitel 3.6 beschrieben, ein ILP-Problem formulieren und mit Hilfe von CPLEX lösen. Dabei müssen natürlich nur diejenigen Kandidaten im ILP berücksichtigt werden, für die ein positiver Effekt auf die vom Programm verbrauchte Energie ermittelt wurde.

3.8.3 Programmanpassungen

Im dritten und letzten Schritt muss der Algorithmus nur noch das ILP-Ergebnis auslesen und dann entsprechend dieser Daten die Kopierfunktionen in das Programm einfügen und die entsprechenden Sprunganpassungen vornehmen, damit das Programm richtig ausgeführt werden kann.

Durch die Einschränkung der eingesetzten Zielhardware, nur relative Sprünge ausführen zu können, müssen im Anschluss an den Compilerlauf die

Sprunganpassungen, die der ARM-Assembler ausführt, noch angepasst werden. Für einige Spezialfälle müssen ebenso Änderungen am Assemblerfile vorgenommen werden, um eine korrekte Ausführung im Scratchpad sicherzustellen. Da es sich bei diesen Änderungen auf der einen Seite nicht unbedingt um triviale Anpassungen handelt, auf der anderen Seite aber nicht das Hauptaugenmerk dieser Diplomarbeit darstellt, werden diese Anpassungen, die das Programm enJumpCorrection vornimmt, im Anhang A detailliert beschrieben.

Abschließend lässt sich also festhalten, dass mit diesem hier vorgestellten Algorithmus eine optimale Lösung des Problems, den vorhandenen Scratchpad-Speicher dynamisch für Programmteile nutzbar zu machen, realisiert ist.

4 Eingliederung in den Designflow

Für das neue, in dieser Arbeit vorzustellende Optimierungsverfahren „dynamische Memory Allocation“ (dynML = dynamic memory locator) wurde auf der Hardwareseite der ARM7TDMI-Prozessor und auf der Softwareseite der experimentelle „enCC“ Compiler eingesetzt.

4.1 Überblick: Hardware

Da sich dieses neue Verfahren auf bestimmte Architekturmerkmale der Hardwareumgebung abstützt, wird an dieser Stelle auf den Prozessor und das eingesetzte Evaluationsboard, auf dem sich der Prozessor befindet, eingegangen.

4.1.1 ARM7TDMI

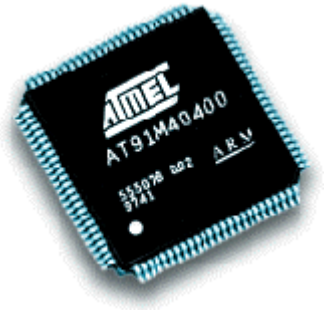

Abbildung 4.1-1 Atmel AT91M40400 mit ARM7TDMI-Core

Der ARM7TDMI gehört zur Familie der RISC-Prozessoren. Seine geringe Chipfläche und sein geringer Verbrauch pro Befehl (117 MIPS/Watt) machen ihn besonders für mobile Anwendungen interessant. Der Prozessor beherrscht zwei verschiedene Befehlssätze. Es gibt den 16 bit „Thumb“-Befehlssatz, der auf 36 Befehle reduziert wurde und den „normalen“ 32 bit Befehlssatz mit 80 Kernelbefehlen. Gerade dieser Thumb-Befehlssatz zeichnet sich durch eine sehr hohe Codedichte aus und wird bevorzugt für die Energieminimierung eingesetzt. Dieser Befehlssatz wird daher auch als einziger vom enCC-Compiler unterstützt.

Der 32 bit Befehlssatz zeichnet sich durch höhere Ausführungsgeschwindigkeiten aus, allerdings auch auf Kosten des Energieverbrauchs. Hier machen sich die 32 bit Wortbreite und die damit verbundenen Schaltaktivitäten auf dem Bus negativ bemerkbar. Grundsätzlich ist ein Umschalten zwischen den Befehlssätzen möglich. Im weiteren soll aber das Augenmerk nur noch auf den Thumb-Befehlssatz aus oben genannten Gründen gelegt werden.

Der ARM7TDMI weist folgende Charakteristika auf:

- 32-bit-RISC Architektur
- 31 x 32 Bit-Register plus 6 zusätzliche Statusregister

- Load-/Store-Architektur
- Dreistufige Instruction-Pipeline
- 32-bit ALU
- separater Barrelshifter
- 32-bit Multiplizierwerk
- 32-bit Adress- und Datenbus.

4.1.2 Evaluationsboard

Die Arbeitsumgebung des Prozessors ist ein Evaluationsboard der Firma ATMEL (Abbildung 4.1-1 Atmel AT91M40400 mit ARM7TDMI-Core). Der AT91M40400 Mikrocontroller besitzt auf dem Chip den ARM7TDMI-Prozessorcore, zusätzliche Peripherie und einen 4 Kb-RAM Speicher, den eigentlichen Scratchpad.

Darüber hinaus stehen extern auf dem Evaluationsboard ein 512 Kb-RAM Speicher und ein 128 Kb-ROM Speicher zur Verfügung. Weitere Eigenschaften des AT91EB01-Boards sind:

- 16-bit Datenbus, 24-bit Adressbus
- 2 serielle Schnittstellen
- JTAG ICE Debug Interface
- Angel Debug Monitor

Wie schon erwähnt, handelt es sich bei dem 4 Kb großen RAM-Speicher auf dem Controller um den eigentlichen Scratchpad-Speicher, der das in dieser Arbeit beschriebene Optimierungsverfahren überhaupt erst ermöglicht. Der Speicher kann genauso wie der 512 Kb-RAM große Speicher angesprochen werden, befindet sich aber mit auf dem Controller und arbeitet dadurch schneller und energieeffizienter als der große RAM-Speicher.

Damit kann man programmgesteuert sowohl Programm- als auch Datenteile in den Scratchpad laden und gegebenenfalls auch dort ausführen.

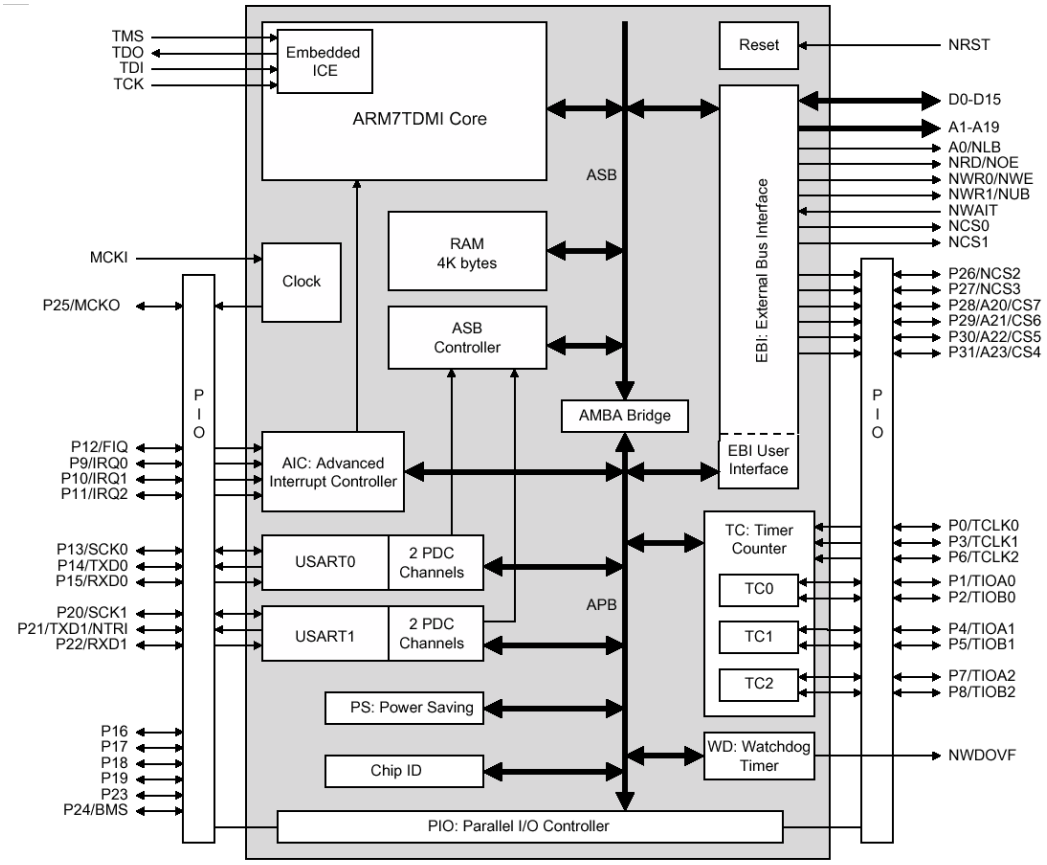

Abbildung 4.1-2 ATMEL AT91M40400

4.1.3 Energieverbrauch des AT91M40400

Die Motivation für die Optimierungsmethode dynML war das Vorhandensein des Scratchpad-Speichers, in dem für diese Arbeit zur Verfügung stehenden Mikrocontroller und die vorausgehende Arbeit [ZO01], in der, in einem ersten Schritt, dieser Speicher statisch für Daten und Programmteile genutzt wurde. Als Erweiterung wird in dieser Arbeit dieser Speicher dynamisch genutzt. Es befinden sich zu unterschiedlichen Laufzeiten unterschiedliche Daten und Programmteile im Scratchpad. Dadurch soll eine weitere Energiereduktion erreicht werden.

Aus einer weiteren zuvor erstellten Diplomarbeit [TH00] mit dem Thema „Energiemessung von ARM7TDMI Prozessor Instruktionen“ ging hervor, dass sich die Kosten von Instruktionen wie in der folgenden Grafik (4.3-1) dargestellt aufteilen. Diese für Instruktionen mit externen Speicherzugriffen ungünstige Energiebilanz hat drei Ursachen:

- Zyklenanzahl
- Stromverbrauch externer Speicherbausteine
- Stromverbrauch des Busses

Man sieht, dass sich sehr hohe Energieeinsparungen durch die Reduktion der Speicherzugriffe erzielen lassen.

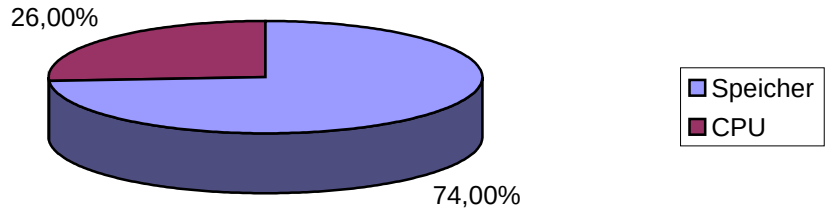

Abbildung 4.1-3 Energieverteilung zwischen Prozessor und Speicher

Im Vergleich zwischen dem Scratchpad und dem Off-Chip-RAM Speicher ist der Unterschied im Energieverbrauch im Zugriff enorm. Im Off-Chip-Speicher macht sich dabei auch die erhöhte Zyklenanzahl beim Zugriff auf die einzelnen Instruktionen negativ bemerkbar und somit ist eine Ausführung eines Befehls im Scratchpad wesentlich günstiger als im Off-Chip-Speicher. Wenn man den Energiebedarf der Ausführung eines einzelnen Befehls im Off-Chip-Speicher auf 100% normiert, so verbraucht ein entsprechender Befehl im Scratchpad gerade einmal 15% von dieser Energiemenge:

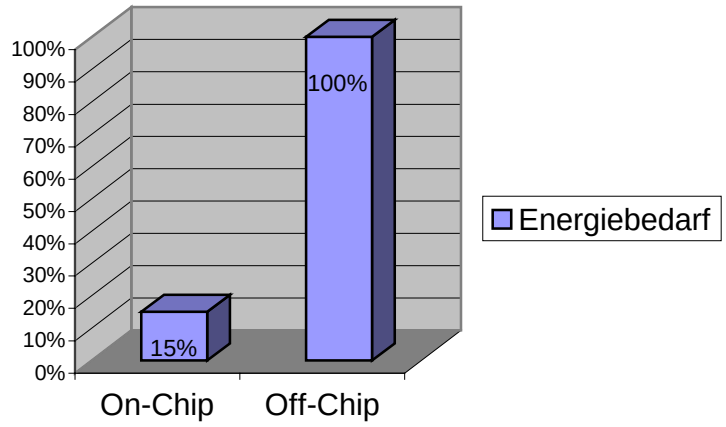

Abbildung 4.1-4 Energierelation zwischen On-Chip- und Off-Chip-Speicher

4.2 Überblick: experimenteller Workflow

Die Codeerzeugung eines Programms, die Profilingdaten nutzt, vollzieht sich in zwei Durchgängen (Abbildung 4.2-1 Ablaufdiagramm dynML). Im ersten Durchlauf wird der Compiler ohne die Unterstützung für den Scratchpad gestartet und das erzeugte Assember-Programm wird in gewohnter Weise assembliert, gelinkt und zuletzt simuliert. Im Profiler werden nun die Ausführungshäufigkeiten der einzelnen Funktionen und Basicblöcken ermittelt. Diese Information wird gespeichert und steht so dem Compiler für den zweiten Durchlauf zur Verfügung.

Im zweiten Durchlauf kann das Modul für die dynamische Nutzung des Scratchpads, das die zuvor ermittelten Profilingdaten benötigt, gestartet werden. Das Modul „dynML“ entscheidet an welchen Stellen welche Blöcke in den Scratchpad kopiert werden sollen. Ebenso werden die benötigten Kopierfunktionen in die entsprechenden Datenstrukturen, die der Compiler benötigt, um den Assemblercode zu erstellen, eingefügt.

Nun erzeugt der Compiler in gewohnter Weise die Quelltextdatei für den ARM-Assembler.

Durch die Ausführung von Codeblöcken auch im Scratchpad stellt sich allerdings noch das Problem der Sprungadressen, die angepasst werden müssen, da die Programmteile nach einem möglichen Kopieren ja an unterschiedlichen Stellen im Speicher liegen.

Die Sprungadressen können nicht schon im Compiler, sondern erst später angepasst werden, nachdem der Linker in einem ersten Durchlauf die Adressen für sämtliche Blöcke der Quelldatei erzeugt hat. Daher muss nach der ersten Assembler- und Linkerausführung und nach dem zweiten Durchlauf ein weiteres Programm, das „enJumpCorrection“-Programm, diese Anpassungen vornehmen. Erst nachdem die Sprünge neu berechnet worden sind und diese Anpassungen in der Assembler-Datei vorgenommen wurden, kann das Programm assembliert und gelinkt werden, mit dem *armsd*[13] simuliert und schließlich mit dem Profiler ausgewertet werden.

[13] armsd = ARM Symbolic Debugger

Dies wird noch einmal in der folgenden Grafik verdeutlicht:

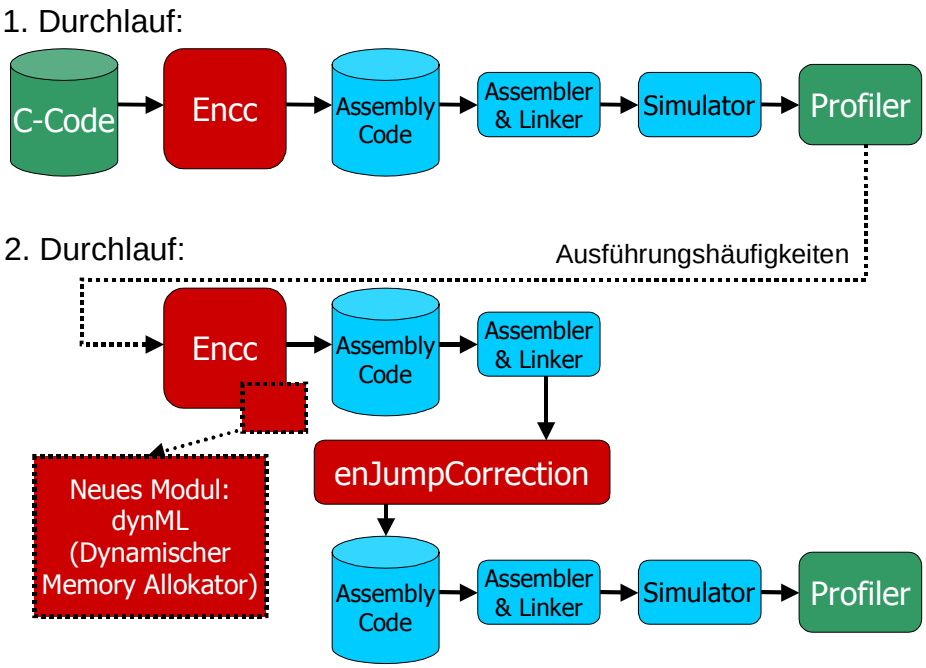

Abbildung 4.2-1 Ablaufdiagramm dynML

4.3 Kopierfunktion

Wie schon in der Einleitung erwähnt, besitzt die Kopierfunktion und deren Realisierung eine entscheidende Bedeutung für das gesamte Verfahren. Nur wenn es gelingt, die Kosten für das Verschieben von Basicblocks möglichst gering zu halten, kann ein gutes Ergebnis erzielt werden, da der Overhead an Energie erst durch die Ausführung im Scratchpad wieder wett gemacht werden muss. Würde das Kopieren zu viel Energie benötigen, könnte es am Ende nur für sehr wenige oder sogar gar keine Basicblöcke sinnvoll im Sinne eines verringerten Energiebedarfs bei der Ausführung sein, sie überhaupt zu verschieben.

4.3.1 Alternative Vorgehensweisen

Aus diesem Grund wurden vor der eigentlichen Realisierung des Algorithmus Versuche mit verschiedenen Kopierfunktionen unternommen, um eine möglichst effektive Kopierfunktion zu erhalten. Dabei gab es durchaus verschiedene Ansätze. Sogar ein Wechsel in den 32bit Befehlssatz des ARM wurde am Anfang in Erwägung gezogen. Davon wurde aber abgesehen, da der Wechsel der Befehlssätze, der innerhalb eines Programms durchaus möglich ist, einen erheblichen Energiebedarf verursacht. Auch wenn man sich allein auf den Thumb-Befehlssatz (16Bit) des ARM konzentriert, so gibt es immer noch verschiedene Lösungsansätze, um einen Programmblock zu verschieben.

Hier sollen nun exemplarisch zwei der Alternativen vorgestellt werden, zwischen denen für die endgültige Implementierung entschieden wurde. Alleine für das eigentliche Verschieben der Blöcke gibt es verschiedene Ansatzpunkte. Im Thumb-Befehlssatz bieten sich dabei vor allem zwei verschiedene Vorgehen an. Auf der einen Seite gibt es die Kombination von LDMIA- und STMIA-Befehlen, mit denen man auf einmal mehrere Words verschieben kann. Auf der anderen Seiten steht die Kombination aus LDRH- und STRH-Befehlen, mit denen man jeweils nur ein einzelnes Word verschieben kann. Beiden Befehlspaaren ist gemein, dass sie automatisch die Register mit den Schreib- und Lesepositionen im Speicher anpassen, so dass diese Änderungen nicht mehr durch zusätzliche Befehle erfolgen müssen.

Die erste Methode zeichnet sich durch seine Einfachheit aus. Es können mit jedem Befehlspaar viele Bytes in den Scratchpad verschoben werden. Eine Schleifenkonstruktion wird nicht nötigt, da selbst größere Basicblocks mit zwei bis drei Befehlspaaren verschoben werden können. Man benötigt zwar mehr Register, muss diese aber nur einmal auf dem Stack sichern und nur einmal wieder zurückladen. Bei der zweiten Methode bräuchte man im Vergleich zu dieser Methode die sechsfache Anzahl an Befehlen, falls man dort keine Schleifen einführt.

Der Vorteil der zweiten Methode ist die Tatsache, dass weniger Register benutzt werden. Es müssen also nicht so viele Register für die Kopierfunktion gesichert werden. Auf der anderen Seite müsste allerdings eine Schleifenkonstruktion eingeführt werden, um die Programmgröße nicht noch zusätzlich zu erhöhen, als dies durch das statML-Verfahren ohnehin schon geschieht. Durch diese Schleifenkonstruktion entstehen allerdings wieder zusätzliche Kosten, da Zählvariablen, Vergleiche und Sprünge eingeführt werden müssten.

Die nachfolgende Tabelle gibt eine Kostenabschätzung für die beiden Befehlspaare:

	Lesen (Werte in 10^{-7}J)		Schreiben (Werte in 10^{-7}J)	
	LDMIA	LDRH	STMIA	STRH
1 Word	1,12	0,81	0,95	0,70
2 Words	1,74	1,63	1,55	1,41
3 Words	2,32	2,45	2,13	2,12
4 Words	2,89	3,26	2,71	2,83
5 Words	3,46	4,08	3,29	3,54
6 Words	4,00	4,90	3,85	4,24
7 Words	4,56	5,71	4,42	4,95
8 Words	5,11	6,53	4,97	5,66

Tabelle 4.3-1 Kostenvergleich zwischen STMIA/LDMIA und STRH/LDRH

Die Werte der Tabelle wurde aus den Ergebnissen der Arbeit von Theokaridis [TH00] entnommen. Wenn man nur die reinen Kosten für das Kopieren allein betrachtet, so könnte man meinen, dass die Kombination aus LDRH-/STRH-Befehlen zumindest für kleine Basicblöcke vorteilhaft ist. Wenn man nun allerdings noch die zusätzlichen Kosten für die benötigte Schleifenfunktionalität mit berücksichtigt (Zählvariable verändern, Abbruchbedingung überprüfen, Sprungbefehl ausführen), dann kommt man zu dem Ergebnis, dass die Kombination aus LDMIA-/STMIA-Befehlen immer zu bevorzugen ist, solange man auf Schleifen verzichten kann. Ab einer Basicblockgröße von 3 Words schneidet diese Kombination immer besser ab, selbst wenn man in beiden Fällen auf Schleifen verzichtet.

Aus diesen Überlegungen wurde die Kombination aus LDMIA-/STMIA-Befehlen für die Kopierfunktion favorisiert.

4.3.2 Gewählte Realisierung

Hier soll nun die gewählte Realisierung vorgestellt werden, die sich in den durchgeführten Versuchen als die günstigste herauskristallisiert hat. Die Label _MAB_ 17 und _MAB_ 19 repräsentieren hier Konstanten, die die Startadressen im normalen Speicher enthalten, die Label _MAB_18 und _MAB_20 halten die Adressen im Scratchpad, an die die Blöcke verschoben werden sollen. Es werden hier also zwei Basicblöcke nacheinander verschoben. Das Label _MAB_5 enthält eine Konstante, die von einem verschobenen Programmteil benötigt wird und somit ebenfalls in den Scratchpad verschoben werden muss:

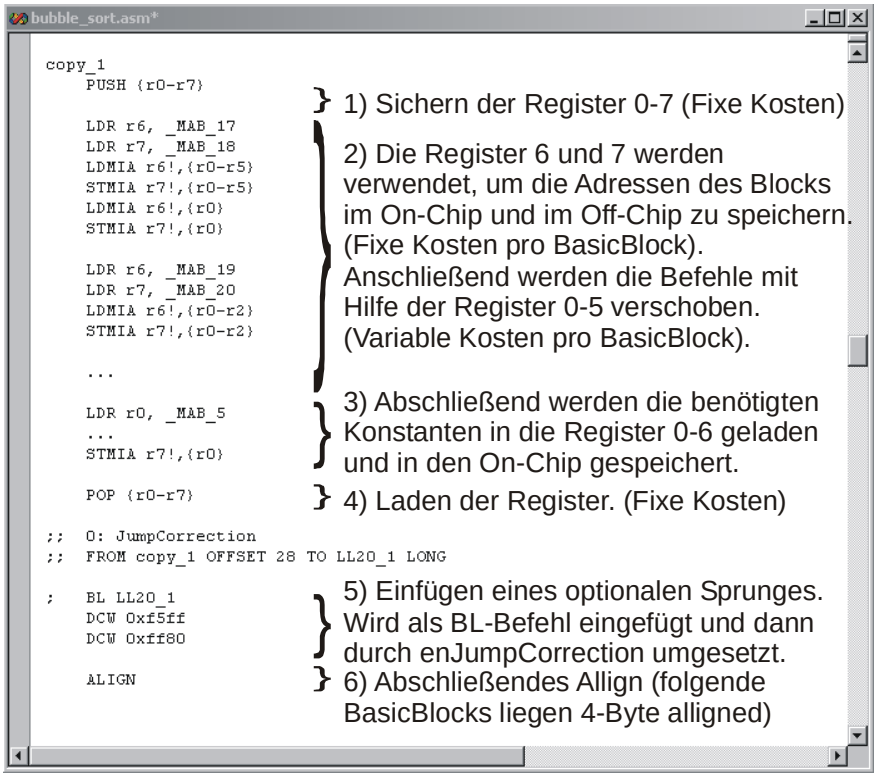

Abbildung 4.3-1 Beispiel einer Kopierfunktion

Anhand des Beispiels in Abbildung 4.5-1 kann man sich den typischen Aufbau einer solchen Kopierfunktion veranschaulichen. Zunächst einmal muss gesagt werden, dass die Kopierfunktionen an einer Stelle in die Datenstrukturen des Compilers eingefügt werden, an dem die eigentliche Registervergabe schon stattgefunden hat. Dies wäre aufgrund von diversen Abhängigkeiten nur sehr schwer zu umgehen. Daher müssen zunächst für jede Kopierfunktion einige Register auf den Stack verschoben werden, damit sie für die Kopierfunktion zur Verfügung stehen. Nach dem Ausführen der Kopierfunktion werden diese Register einfach wieder vom Stack zurückgeladen. Aus Abwägungen zwischen dem Energiebedarf der Push- und Pop-Befehle für verschiedene Registeranzahlen und dem daraus erreichbaren Gewinn, wurde entschieden, sämtliche Low-Register, die Register 0-7, für die Kopierfunktion zu benutzen, auch wenn der seltene Fall auftreten kann, dass nicht alle diese Register wirklich benutzt werden. Sobald allerdings ein einziger Basicblock mit mehr als 11 oder mehr Befehlen verschoben werden soll, rechtfertigt sich bereits dieses Vorgehen. Tests mit Programmen haben gezeigt, dass diese Basicblockgröße eine durchschnittliche Größe darstellt.

Das Befehlspaar aus Push- und Pop-Befehl stellt dabei für jede Kopierfunktion einen fixen Kostenbeitrag dar. Je mehr Basicblöcke mit einer Kopierfunktion verschoben werden, umso günstiger wird die Verwendung einer Kopierfunktion, da diese fixen Kosten auf sämtliche zu verschiebenden Basicblocks aufgeteilt werden können.

Anschließend an das Verschieben der Register auf den Stack kann schon mit dem eigentlichen Verschieben der Blöcke begonnen werden. Die Register 6 und 7 dienen dabei der Speicherung von Adressen. In Register 6 wird die Adresse im Off-Chip gespeichert, von der gelesen werden soll, und in Register 7 die Adresse im On-Chip, an die geschrieben werden soll. Das Laden der Adressen in die Register geschieht dabei für jeden Basicblock unabhängig von dessen Größe. Es entstehen hier also fixe Kosten pro Basicblock.

Sind die Adressen geladen, kann mit dem Kopieren begonnen werden. Für das eigentliche Kopieren wurden die LDMIA und STMIA Befehle ausgewählt, da sie mit großem Abstand den größten Speicherbereich in Relation zum Energiebedarf des Befehls bewegen können.

Außerdem aktualisieren diese Befehle automatisch die Adressen in den Registern 6 und 7, sodass dies nicht „von Hand“ geschehen muss. Der Nachteil, dass dadurch sämtliche Basicblocks 4-Byte aligned im Speicher liegen müssen und auch nur 4-Byte aligned in den Scratchpad kopiert werden können, wird dadurch bei weitem aufgewogen. Selbst wenn man bedenkt, dass dadurch unter Umständen NoOps (No-Operations) eingefügt werden müssen, da jeder Basicblock, der verschoben werden soll, nun eine Größe des Vielfachen von 4 Bytes besitzen muss und dadurch eventuell auch mehr Platz im Scratchpad benötigt.

Je nach Größe des zu verschiebenden Basicblocks wird das Instruktionspaar LDMIA-STMIA mehrmals benötigt. Pro Ausführung können dabei maximal 24 Bytes verschoben werden, was 12 Befehlen im 16-Bit Befehlssatz entspricht. Das

eigentliche Verschieben stellt somit den einzigen Teil der Kopierfunktion dar, an dem pro Basicblock variable Kosten entstehen.

Sind die eigentlichen Programmteile verschoben, so kann es sein, dass auch noch benötigte Konstanten in den On-Chip verschoben werden müssen, da sie nicht mehr direkt vom On-Chip Bereich aus relativ adressiert werden können. Dazu werden nun die Register 0-6 verwendet. In diese Register werden die Konstanten geladen und dann mit einem einzigen STMIA Befehl in den On-Chip kopiert. Sollten mehr als 7 Konstanten benötigt werden, so wird das Vorgehen einfach so lange wiederholt, bis alle Konstanten verschoben wurden.

Nun befinden sich alle benötigten Daten im Scratchpad und die Register, die hier verwendet wurden, können wieder vom Stack geholt werden.

Zum Abschluss kann es noch einen optionalen BL-Befehl geben, falls der im Speicher direkt auf die Kopierfunktion folgende Basicblock in den Scratchpad verschoben wurde. Somit wird hier ein BL eingefügt, der dann später vom Programm enJumpCorrection in ein passendes Bitmuster umgewandelt wird, damit der Assembler keine falsche Sprungberechnung durchführt. In diesem Fall muss die Kopierfunktion auch mit einem ALIGN-Befehl abschließen, damit der nächste Block 4-Byte aligned liegt und korrekt kopiert werden kann. Wird der an die Kopierfunktion anschließende Block nicht verschoben, so werden der Sprung und der ALIGN-Befehl an dieser Stelle nicht benötigt.

4.3.3 Anmerkungen und Einschränkungen

Anzumerken ist hier noch, das die Kopierfunktion selber nicht in den Scratchpad verschoben werden darf. Ein Fall, in dem sich ein solches Kopieren lohnen würde, ist dabei nicht möglich, selbst wenn mehrere Schleifen verschachtelt sind. Hier würde sich das Verschieben nur für die Kopierfunktion der inneren Schleife lohnen, wenn auch alle anderen Teile der Schleife bereits verschoben wurden. Wenn ein so großer Platz im Scratchpad vorhanden ist, dass die gesamte Schleife in den Scratchpad passt, ist die Kopierfunktion an dieser Stelle nicht mehr notwendig, da ein Verschieben der Programmteile in der übergeordneten Kopierfunktion einen günstigeren Austauschzeitpunkt bedeuten würde, da dadurch fixe Kosten eingespart werden könnten.

4.3.4 Kostenfunktion

Wie schon erwähnt, setzten sich die Kosten für die Kopierfunktion aus verschiedenen Einzelkosten zusammen:

Gleichung 4.3-1 Zusammensetzung der Kopierkosten

$$CopyCosts = Fix_{CopyFunction} + \sum_{i} \left(Fix_{BB} + Var_{BB_i} \right) + Var_{Consts}$$

Die einzelnen Faktoren der Gleichung im Detail:

CopyCosts: Gesamtenergiekosten der Kopierfunktion

$Fix_{CopyFunction}$: Fixe Kosten der Kopierfunktion. Diese setzen sich aus den Kosten für die Push- und Pop-Operationen zusammen.

Fix_{BB}: Fixe Kosten für einen Basicblock. Die fixen Kosten bestehen hier aus dem Laden der Position in die Register und sind für jeden Basicblock unabhängig von der Größe gleich.

Var_{BBi}: Variable Kosten jedes Basicbocks *i*. Diese Kosten sind variabel, da jeder Basicblock eine unterschiedliche Größe besitzen kann und damit unterschiedlich viel Energie für dessen Verschieben in den Scratchpad aufgewandt werden muss.

Var_{Consts}: Variable Kosten für das Verschieben von Konstanten, die zusätzlich in den Scratchpad verschoben werden müssen. Diese Kosten schwanken sehr stark.

Festhalten lässt sich hier also, dass das Kopieren von Basicblöcken mehrere Kostenkomponenten beinhaltet. Bei der Berechnung dieser Kosten und der damit verbundenen Einschätzung, ob es sich lohnt einen Basicblock in den Scratchpad zu verschieben oder nicht, müssen also die variablen Kosten jedes einzelnen Basicblocks, die fixen Kosten jedes Basicblocks und schließlich die fixen Kosten der eigentlichen Kopierfunktion berücksichtigt werden.

4.4 Integration in den enCC Compiler

Das Modul dynML wurde in den vorhandenen enCC-Compiler integriert. Der enCC ist ein C-Compiler, mit dem es möglich ist, den Energiebedarf des zu compilierenden Programms zu minimieren. Es handelt sich dabei um einen Forschungscompiler, der am Lehrstuhl 12 für Informatik an der Universität Dortmund entwickelt wurde.

Aktuell besteht der Compiler aus den Teilen:

- LANCE2 Frontend
- Ein Backend für den 16Bit Thumb-Befehlssatz des ARM7TDMI Prozessors
- Ein Backend für den 32Bit Leon Prozessor
- Diversen Anpassungen an das ARM 2.5 SDT
- Ein Energieprofiler, mit dem die Energiewerte eines Programms ermittelt werden

In diesen bestehenden Compiler, der ständig weiterentwickelt wird, soll das neue Verfahren zur dynamischen Nutzung des Scratchpads für die ARM7 Architektur integriert werden.

Dazu wird der Compiler um ein zusätzliches Modul, „dynML“ genannt, erweitert.

Das Modul dynML arbeitet in mehreren Schritten, um eine möglichst gute Auswahl an Basicblocks für den Scratchpad zu finden.

Die Arbeitsschritte des Moduls sind im Einzelnen:

1. Importieren der Knoten aus sämtlichen zum Programm gehörigen CFGs (Control-Flow-Graphen). Bei den Knoten handelt es sich um die Basicblöcke des Programms.
2. Importieren der zu den Knoten gehörigen Kanten. Die Kanten spiegeln dabei den Kontrollfluss zwischen ein einzelnen Basicblöcken wider.
3. Vereinfachen des/der Graphen und Erstellung der Superblöcke, um geeignete Austauschzeitpunkte zu erhalten.
4. Erstellung einer Liste mit möglichen Kandidaten für jeden Superblock, die einen Gewinn bei der Ausführung im Scratchpad versprechen. Hierbei müssen auch schon die Kosten für den Kopiervorgang berücksichtigt werden. Bei den Kandidaten muss auch berücksichtigt werden, dass verschiedene Basicblocks zusammengenommen einen besseren Energiegewinn erzielen können, als wenn man jeden Basicblock nur für sich allein betrachtet, da eventuell Sprünge eingespart werden können. Es werden also auch Multiblöcke, bestehend aus mehreren Basicblocks, als Kandidaten betrachtet.
5. Im nächsten Schritt werden aus den möglichen Kandidaten für jeden Superblock, abhängig von der zur Verfügung stehenden Scratchpadgröße, diejenigen Basicblocks ausgewählt, für die der Gesamtgewinn maximal wird.
   Dieser Schritt ist vergleichbar mit der im Modul statML gelösten Problematik. Auch hier wird zur Lösung dieses Problems ein ILP formuliert und mit Hilfe von CPLEX gelöst.
6. Abschließend werden die benötigten Kopierfunktionen für alle Superblöcke eingefügt, aus denen ein oder mehrere Basicblöcke in den Scratchpad kopiert werden sollen. Zusätzlich findet eine nötige Control-Flow-Anpassung statt, in der die benötigten Sprünge in das Programm eingefügt werden. Die Kopierfunktionen beinhalten den Code, um die eigentlichen Basicblocks zu verschieben, aber auch, um benötigte Konstanten, auf die mit relativem Offset zugegriffen wird, zu verschieben, da sonst der Offsetbereich durch das Verschieben verlassen wird und die Konstante nicht mehr adressiert werden kann.
7. Alle weiteren noch notwendigen Änderungen, um einem korrekten, ausführbaren Code zu erhalten, sind erst möglich, nachdem mit dem ARM-Linker die endgültigen Adressen der Basicblocks im Speicher ermittelt wurden.
   Aus diesem Grund mussten die weiteren Arbeitsschritte in ein externes Programm, „enJumpCorrection“ genannt, das im Anhang A detailliert beschrieben wird, ausgelagert werden.

Eine genaue Beschreibung der einzelnen Arbeitsschritte folgt nun:

4.4.1 Importieren der Knoten und Kanten

Beim Importieren der Knoten und Kanten werden sämtliche benötigten Daten aus dem Call Graph (Abbildung 4.4-1), der im Falle des enCC aus einem FCG (Function Call Graph) und einem, oder mehreren CFGs (Control Flow Graph), je nach der Anzahl der im Programm vorhandenen Funktionen, besteht, in eine eigene Datenstruktur, im folgenden SAG[14] (Structural Analysis Graph) genannt, übernommen. Dies ist notwendig, da diverse tiefgreifende Änderungen auf diesen Datenstrukturen für die notwendigen Analysen des dynML-Moduls durchgeführt werden, wie zum Beispiel das Erstellen der Superblocks aus zusammengefassten Basicblöcken.
Beim eigentlichen Kopiervorgang werden nun alle Knoten des FCG abgelaufen und auf möglicherweise vorhandene CFGs hin untersucht.
Dies geschieht so lange, bis die Main-Funktion des Programms gefunden wurde.
Von der Main-Funktion ausgehend wird nun eine Art „Function-Inlining“ durchgeführt, das bedeutet, dass rekursiv von der Main-Funktion ausgehend alle aufgerufenen Funktionen in den SA-Graphen übernommen werden. Dabei muss darauf geachtet werden, dass rekursive Funktionen an einer Stelle nur einmal importiert werden. Die Kanten werden auch entsprechend angepasst, so dass man am Ende des Importiervorgangs einen einzigen großen Graphen mit allen Funktionen aus dem ursprünglichen FCG mit mehreren CFGs gebildet hat. Die Wurzel bildet dabei also der erste Basicblock der Main-Funktion.
Nach dem Importieren repräsentiert jeder Knoten im SAG einen Basicblock. Jeder Knoten im SA-Graph kann aber nicht nur einen Basicblock sondern auch einen Superblock repräsentieren. Da dafür noch weitere Informationen benötigt werden, enthält ein Knoten im SAG mehr Informationen, als diejenigen, die im Knoten eines CFG enthalten sind. Durch diese Zusatzinformationen wird es im nächsten Schritt einfacher, die benötigten Superblocks zu bilden. Im einzelnen besteht ein SAG-Knoten aus einer Liste mit allen enthaltenen Basicblöcken, einer Liste mit möglichen Basicblöcken, für die sich das Verschieben in den Scratchpad lohnen würde (Multiblöcke), einer Liste mit den Blöcken, die dann auch später fürs Kopieren ausgewählt werden und verschiedenen weiteren Informationen, die für die Verarbeitung der Daten notwendig sind.

Jede dieser gerade vorgestellten Listen besteht aus drei Zeigern und der Funktionsnummer, die der Funktion, aus der der Basicblock stammt, zugeordnet ist. Die benötigten Zeiger, die hier gespeichert sind, sind die Zeiger auf den eigentlichen Basicblock (CFGraphNode), die Zeiger auf den übergeordneten Graphen (CFGraph) und schließlich auf die dazugehörige Symboltabelle (SymTable). All diese Informationen sind nötig, um die Funktionsweise von dynML sicher zu stellen.

[14] SAG = Structural Analysis Graph, dieser Begriff wurde gewählt, da diese Datenstruktur zur Ermittlung der im Programm enthaltenen Schleifen und Strukturen genutzt wird.

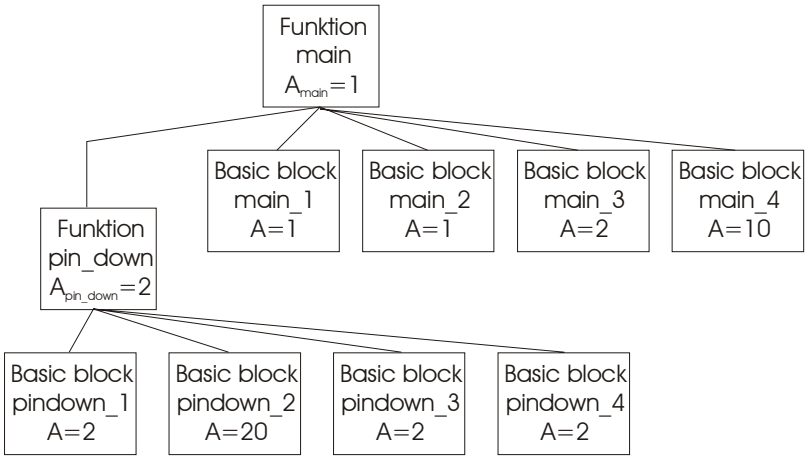

Abbildung 4.4-1 Aufbau eines Call-Graphen

4.4.2 Erstellung der Superblöcke

Nun, da wir sämtliche benötigten Daten importiert haben, können wir damit beginnen, die Datenstrukturen zu analysieren und unserem Ziel entsprechend zu verändern. Wie schon oben erwähnt, kann ein Knoten im SAG sowohl einen einzelnen Basicblock als auch einen komplexeren Superblock repräsentieren. Da nach dem Importieren der Kanten und Knoten aus den vorgegebenen Datenstrukturen des Compilers zunächst jeder Knoten im SAG auch nur einem einzelnen Basicblock entspricht, müssen hier nur einzelne Knoten, die auf mehreren Wegen erreicht werden können (siehe Definition Superblock), zusammengefasst werden und die Kanten zwischen diesen zusammengefassten Knoten richtig angepasst werden. Es werden diejenigen Basicblocks in einem Superblock zusammengefasst, die zu einer Schleife gehören.

Im Anschluss an die Beschreibung des Algorithmus folgt ein Beispiel, das das Vorgehen exemplarisch Schritt für Schritt noch einmal erläutert.

Bevor das im folgenden beschriebene rekursive Verfahren des Algorithmus allerdings ausgeführt werden kann, müssen zunächst alle Kanten aus dem Graphen entfernt werden, die einer Self-Loop, also einer Kante von einem Knoten zu demselben Knoten, entsprechen. Mit dieser Self-Loop hat man bereits die innersten Schleifen, die nur aus einem einzigen Basicblock bestehen, gefunden.

Dann wird zur Schleifendetektion eine rekursive Funktion verwendet, die den Graphen durchläuft und jeden besuchten Knoten markiert. Wird ein bereits markierter Knoten K_i vom Knoten K_j erneut erreicht, so werden die Knoten K_i und K_j zu einem Superknoten mit dem Namen K_i vereinigt und der Knoten K_j wird zum Löschen freigegeben. Der Knoten K_i enthält nun alle Informationen der beiden Knoten.
Einzige Ausnahme hierbei ist, falls es sich beim Knoten K_j um den Startknoten der Funktion oder des gesamten Programms handelt. In diesem Fall wird K_j mit K_i

vereinigt und K_i gelöscht, da der Startknoten für das weitere Vorgehen mit dem Startnamen erhalten bleiben muss.

Sobald dieser Punkt erreicht wird, wird die Rekursion gestoppt, die Knoten zusammengefasst, die Kanten entsprechend angepasst und schließlich der nun überflüssige Knoten gelöscht.

Nun muss kontrolliert werden, ob nach diesem Schritt wieder eine Self-Loop entstanden ist. Wird eine solche Self-Loop erkannt, so hat man eine Schleife gefunden, die allerdings Bestandteil einer größeren Schleife sein kann. Daher wird dieser Knoten nun in eine zusätzliche Datenstruktur zwischengespeichert und die Self-Loop gelöscht.

Dieses Verfahren wird nun so lange durchgeführt, bis keine Knoten mehr gefunden wurden, die zusammengefasst werden müssen. Als Ergebnis bildet sich aus dem anfänglichen SAG mit einfachen Basicblöcken ein SAG, der nun ausschließlich aus Superblöcken besteht. Allerdings enthalten diese Superblöcke nur die äußersten Schleifen. Alle inneren Schleifen, die während der Ausführung zwischengespeichert wurden, müssen nun ebenfalls zurückkopiert und wieder in den SAG eingeführt werden. Dieses, zugegebenermaßen etwas komplizierte Vorgehen wird dadurch notwendig, dass einzelne Basicblocks nun in mehreren Knoten des SAG enthalten sein können.

Die folgende Abbildung 4.4-2 soll den hier entwickelten Algorithmus verdeutlichen. Im ersten Schritt wird die Self-Loop von Basicblock 4 erkannt. Damit wird dieser Block in einen Superblock konvertiert und anschließend wird die Self-Loop gelöscht. Zusätzlich wird der neue Superblock in einer zusätzlichen Datenstruktur zwischengespeichert. Das Ergebnis gibt Schritt 2 wieder. In Schritt 3 werden die Knoten in einem Depth-First Vorgehen durchlaufen und markiert. Erreicht man einen bereits markierten Knoten, so werden diese beiden Knoten zu einem Block zusammengefasst. In diesem Beispiel Basicblock 4 und Superblock 1. Das Endergebnis des Zusammenfügens ist in Schritt 4 dargestellt. Die Schritte 3 und 4 werden jetzt so lange wiederholt, bis man eine Self-Loop erkennt. Diese Self-Loop entspricht dann einem Superblock. In Schritt 5 entspricht dies dem Basicblock 5, der durch das Zusammenfügen der Basicblöcke 2-5 entstanden ist. Dieser Block wird dann ebenfalls zum Superblock, der ebenfalls zwischengespeichert wird, in diesem Beispiel zum Superblock 2. Dieses Vorgehen der Schritte 3 bis 5 wird so lange wiederholt, bis keine Blöcke mehr zusammengefasst werden müssen. Abschließend in Schritt 6 müssen dann die zwischenzeitlich gespeicherten Superblöcke, die innere Blöcke darstellen, wieder zurückgeladen und in den Graph eingefügt werden. In diesem kleinen Beispiel ist es ausschließlich nötig den Superblock 1 wieder zurückzuladen, da dieser den einzigen inneren Block darstellt.

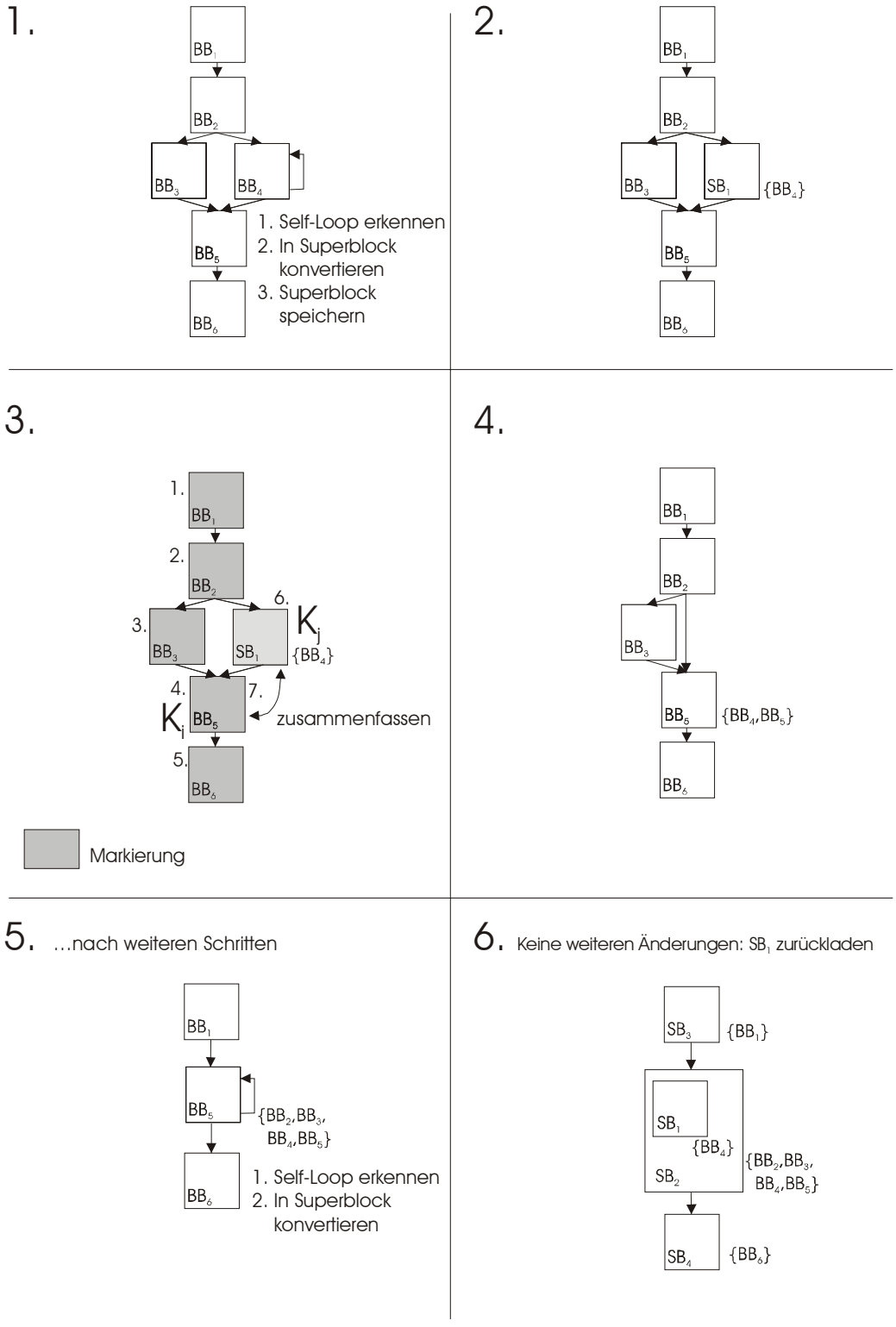

Abbildung 4.4-2 Schematische Darstellung des Algorithmus zur Generierung der Superblöcke

Wie man an der Abbildung 4.6-3, die das Ergebnis für den HeapSort-Benchmark wiedergibt, erkennen kann, wurden hier verschiedene innere Schleifen gefunden. Insgesamt besteht der SAG hier aus 7 Superblöcken (main, ArraySort, LL23, LL53, LL30, LL58 und _M_34). Dabei ist der Superblock mit der Bezeichnung „LL30“ eine innere Schleife im Superblock „LL53“. Die Superblöcke „LL53“ und „LL23“ wiederum sind Schleifen innerhalb des Superblocks mit dem Namen „ArraySort“. Man erkennt dies ganz einfach daran, dass sämtliche Basicblocks, die in einer inneren Schleife vorhanden sind, auch in der äußeren Schleife/Superblock zu finden sind.

Durch dieses Vorgehen wird dann der nächste Schritt, die Erzeugung der Kandidaten, vereinfacht.

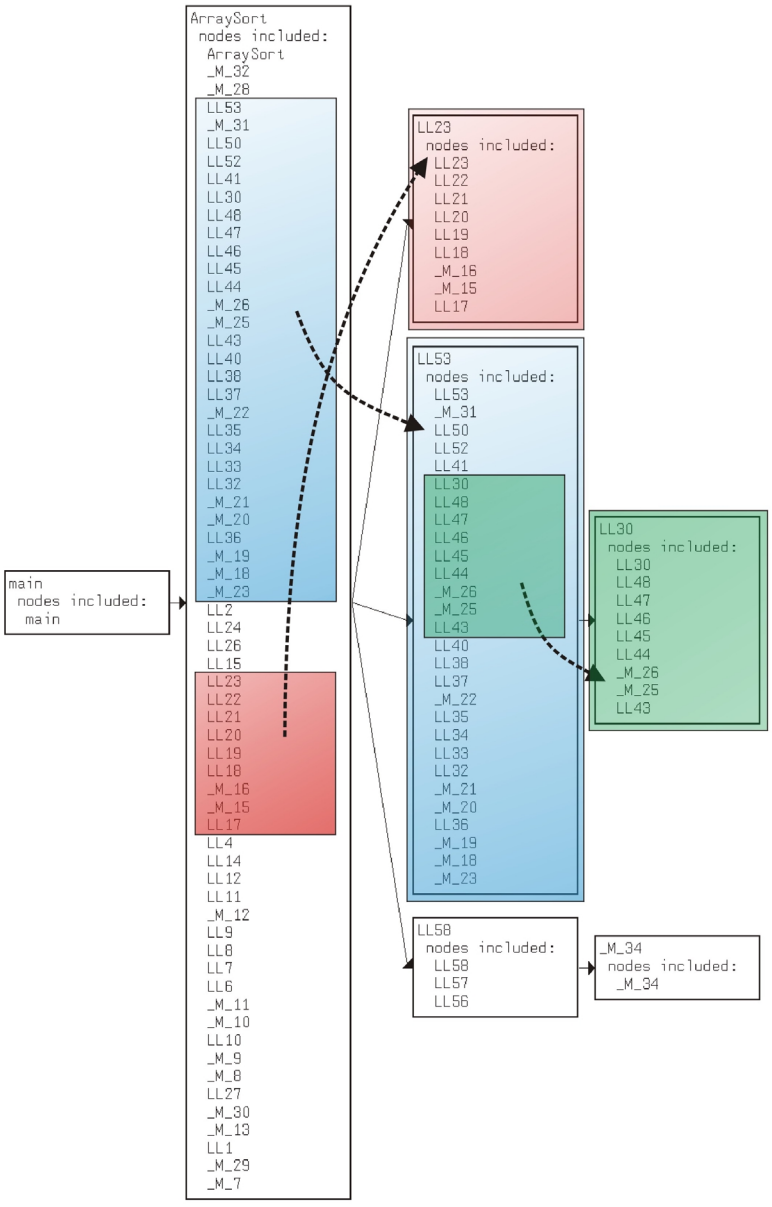

Abbildung 4.4-3 Beispiel eines SAG (Programm: HeapSort)

4.4.3 Erstellung möglicher Kandidaten

Nun, da wir einen Graphen mit Superblöcken gebildet haben, können wir uns daran machen, jeden Superblock für sich auf mögliche Austauschkandidaten, für die sich eine Ausführung im Scratchpad lohnen würde, hin zu untersuchen. Da wir allerdings die Programmblöcke selbst in den Scratchpad kopieren und dazu nicht den ARM Assembler und Linker verwenden, stößt man dabei auf das Problem, dass externe Funktionen wie z.B. „printf“ nur mit einem Trick verschoben werden können. Die Information an welcher Stelle im Speicher diese externe Funktion gelinkt wird, wird zwar vom Linker zur Verfügung gestellt, die Größe der Funktion allerdings nicht. Diese Information wäre allerdings notwendig, um die Funktion in den Speicher verschieben zu können. Indirekt könnte man versuchen, aus der Objekt-Datei die benötigte Größe der Funktion zu erhalten.

Theoretisch ergibt sich somit sogar die Möglichkeit, auch Library-Funktionen mit in den Scratchpad zu verschieben. Im Zuge dieser Diplomarbeit wurde ein Verschieben von Library-Funktionen allerdings nicht implementiert. Da die untersuchten Programme allerdings keinen Gebrauch von solchen Funktionen machen, bzw. dieser Gebrauch auf ein Minimum reduziert wurde, ergibt sich hierdurch keine wesentliche Einschränkung oder Verfälschung der erzielten Ergebnisse.

Da ein Verschieben von mehreren im Speicher hintereinander liegenden Basicblöcke günstiger sein kann, als wenn man jeden Basicblock allein für sich betrachtet, werden die Kandidaten so gebildet, wie es im Kapitel 3.7 beschrieben wird. Es wird nicht nur jeder Basicblock für sich betrachtet, sondern auch Kombinationen aus Blöcken, die hintereinander im Speicher liegen und keine absoluten Sprünge enthalten.

Jedem Kandidaten wird hier auch ein eindeutiger Name zugewiesen. Dies ist notwendig, da ja ein Basicblock in verschiedenen Kandidaten und sogar in verschiedenen Superblöcken auftauchen kann. Man kann also nicht einfach den Namen des Basicblocks verwenden.

Daher erhält jeder Superblock eine eindeutige Nummer und jeder Kandidat eines Superblocks ebenfalls. Da CPLEX die Einschränkung auf 16 Zeichen für seine Variablen besitzt, wird hier der Name für den Kandidaten also so kurz wie möglich zusammengesetzt. Der erste Kandidat des ersten Superblocks erhält einfach die Bezeichnung: „SB1_1“, der zweite: „SB1_2“ usw.

4.4.4 Auswahl des besten Sets

Die Auswahl geschieht, wie schon in Kapitel 3.7 in einem kleinen Beispiel beschrieben, mit Hilfe von CPLEX. Es wird also ein entsprechenden ILP formuliert und dann wird der ILP-Solver gestartet.

Beim Aufbau des ILPs muss man nur darauf achten, dass der zur Verfügung stehende Platz im Scratchpad nicht überschritten wird. Dabei kann man sich nicht einfach auf die Programmgröße vor dem Verschieben abstützen, da durch das Verschieben Sprunganpassungen und eventuell zusätzlich auch noch zu verschiebende Konstanten hinzukommen. Im ungünstigsten Fall sind das 8 Byte

für zwei Branch-Link Befehle für konditionale Sprünge plus der mögliche Speicherbedarf von 4 Byte pro verschobener Konstante.

Ebenfalls muss der mögliche Gewinn für jeden Kandidaten richtig berechnet werden. Dazu müssen die fixen und variablen Kosten der Kopierfunktion und die Kosten für notwendige Sprünge berücksichtigt werden.

Hat man die Größen der Blöcke und deren möglichen Gewinn berechnet, kann man ganz leicht das entsprechende ILP aufbauen.

4.4.5 Controlflow Korrektur und Kopierfunktion

Die Anpassung und Korrektur der Controlflows ist nun eine der kompliziertesten Aufgaben des dynML-Moduls. Schließlich muss sichergestellt werden, dass die semantische Korrektheit des Programms auch nach dem Verschieben einzelner Programmteile und dem Ausführen im Scratchpad erhalten bleibt. Zunächst muss für jeden Superblock, für den eine Kopierfunktion eingefügt wird, geprüft werden, ob im Anschluss an die Kopierfunktion bereits ein langer Sprung in den Scratchpad nötig wird, oder nicht. Falls dies der Fall ist, so muss beim Einfügen der Kopierfunktion in den CFG des EnCC auch der entsprechende Sprung mit eingefügt werden. Ebenso muss bei der Kopierfunktion darauf geachtet werden, dass neben dem eigentlichen Programmcode, der zu diesem Zeitpunkt schon durch notwendige Sprünge ergänzt wurde, auch eventuell benötigte Konstanten direkt mit in den Scratchpad kopiert werden.

Bevor die Kopierfunktionen allerdings eingefügt werden, werden die Basicblöcke des Superblocks einer Vielzahl von Untersuchungen und eventuellen Änderungen unterzogen:
Notwendige Änderungen können dabei bei allen Blöcken auftreten, die in den Scratchpad verschoben werden, aber auch bei sämtlichen Vorgängerknoten, egal, ob sie auch in den Scratchpad verschoben werden, oder nicht. Einer der Vorgänger kann dabei durch die Definition der Superblöcke nicht einmal im gleichen Superblock liegen. Dieser Block muss dann einen Sprung nicht auf den entsprechenden Basicblock, sondern auf die Kopierfunktion des Superblocks ausführen, damit sichergestellt werden kann, dass die Kopierfunktion auch richtig ausgeführt wird und die richtigen Inhalte im Scratchpad vorhanden sind. Abgesehen von diesem Spezialfall werden die Vorgänger, sofern sie nicht auch in den Scratchpad kopiert werden, entsprechend der Tabelle 4.4-1 bearbeitet. Falls der Vorgänger auch im Scratchpad ausgeführt werden soll, wird er hier nicht verändert, damit die Veränderungen nicht doppelt ausgeführt werden. Der Basicblock befindet sich immer im Off-Chip Speicher:

Position: Nachfolger 1 des Basicblocks	Position: Nachfolger 2 des Basicblocks	Aktion durchgeführte Änderung
Scratchpad	-	Hinzufügen eines BL-Befehls, oder Ändern eines B-Befehls in einen BL-Befehl
Scratchpad	Off-Chip	Invertieren des konditionalen Sprungs und Hinzufügen eines BL-Befehls
Scratchpad	Scratchpad	Hinzufügen eines zusätzlichen Labels und Hinzufügen von 2 BL-Befehlen
Off-Chip	Scratchpad	Hinzufügen eines BL-Befehls

Tabelle 4.4-1 Änderungen im Kontrollfluss: Vorgänger

In der folgenden Abbildung 4.4-4 sind die vier Fälle noch einmal dargestellt:

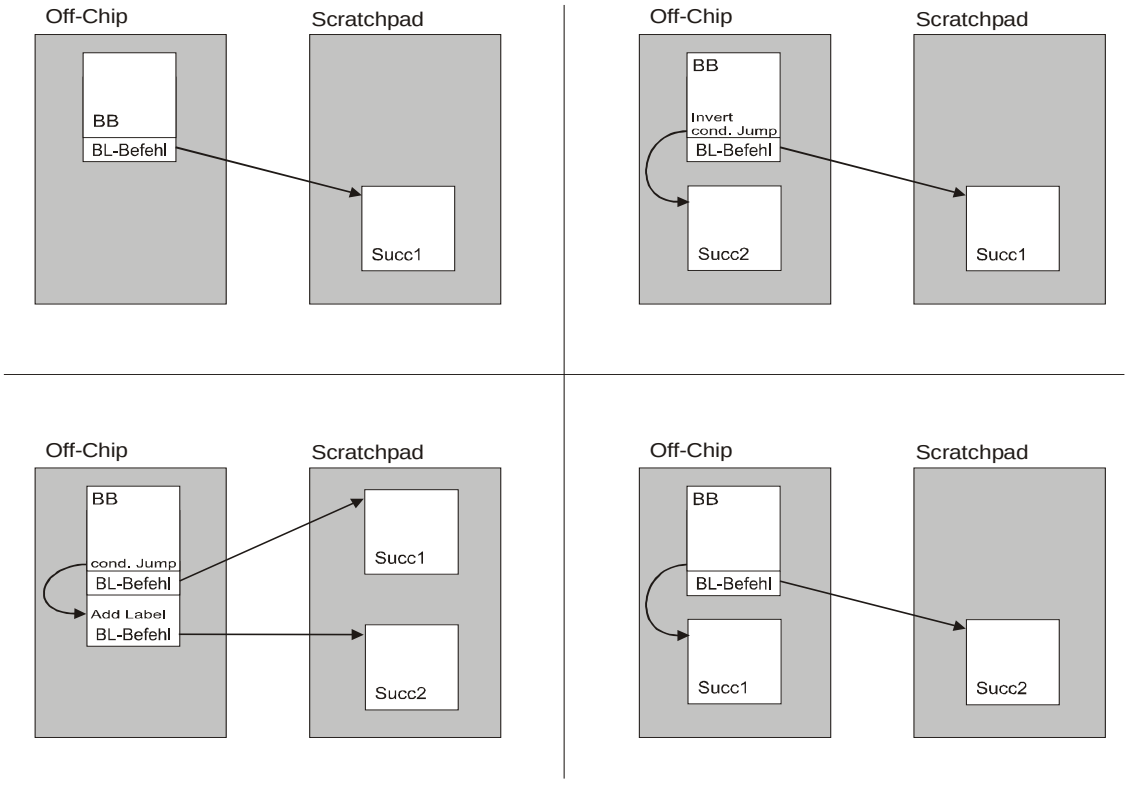

Abbildung 4.4-4 Controlflow-Korrektur von Basicblöcken im Off-Chip

Die Veränderungen an den Basicblöcken, die in den Scratchpad kopiert werden, werden wie folgt ausgeführt:

Position: Nachfolger 1 des Basicblocks	Position: Nachfolger 2 des Basicblocks	Aktion durchgeführte Änderung
Off-Chip (gleicher Superblock)	-	Hinzufügen eines BL-Befehls, oder Ändern eines B-Befehls in einen BL-Befehl
Off-Chip (nächster Superblock)	-	Hinzufügen eines BL-Befehls, der auf die Kopierfunktion des nächsten Superblocks zeigt
Scratchpad	-	Keine Anpassung nötig oder Anpassung eines kurzen Sprungs
Scratchpad	Scratchpad	Keine Anpassung nötig oder Anpassung eines kurzen Sprungs
Scratchpad	Off-Chip	Hinzufügen eines BL-Befehls
Off-Chip	Scratchpad	Invertieren des konditionalen Sprungs und Hinzufügen eines BL-Befehls
Off-Chip	Off-Chip	Hinzufügen einer zusätzlichen Labels und Hinzufügen von 2 BL-Befehlen

Tabelle 4.4-2 Änderungen im Kontrollfluss: verschobener Basicblock

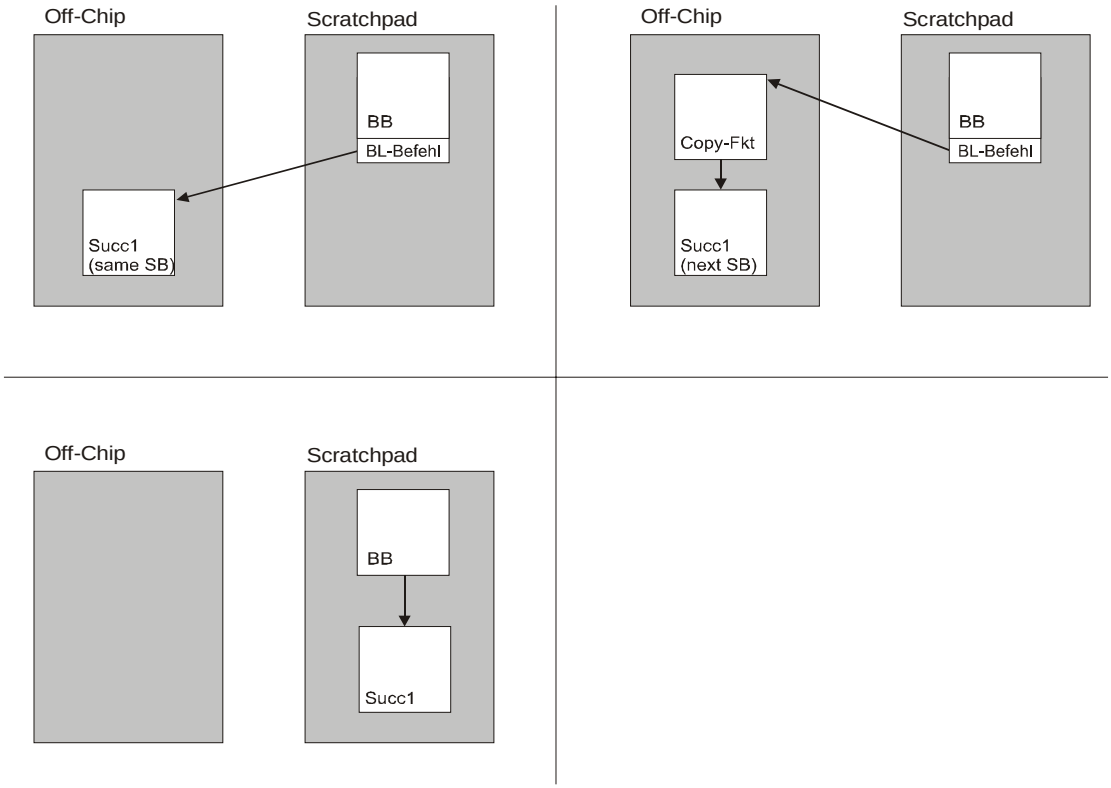

Abbildung 4.4-5 Controlflow-Korrektur von Basicblöcken im Scratchpad mit einem Nachfolger

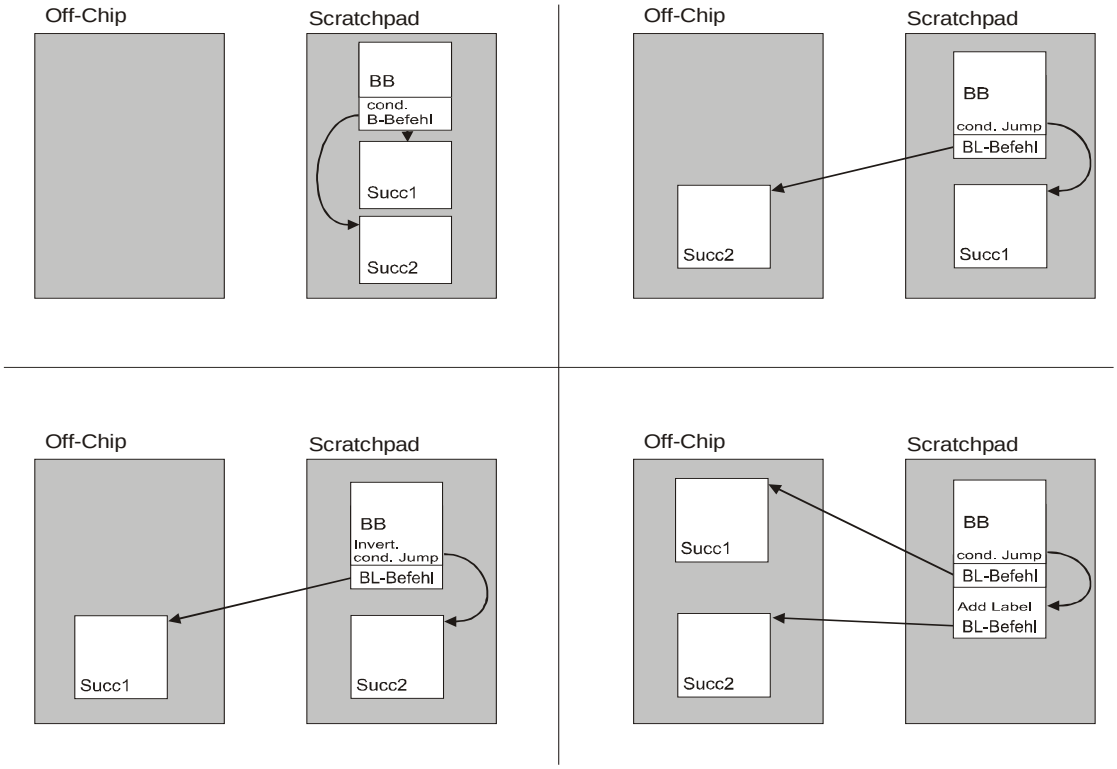

Abbildung 4.4-6 Controlflow-Korrektur von Basicblöcken im Scratchpad mit zwei Nachfolgern

Das in den Tabellen angesprochene Invertieren der konditionalen Sprünge kann unter Umständen zu einer Verringerung des benötigten Speicherbedarfs im Scratchpad führen. Ein Nebeneffekt dabei ist natürlich auch der geringere Energiebedarf.

Damit ist das Vorgehen des dynML-Moduls weitgehend beschrieben. Zusammenfassend werden also alle Daten aus Datenstrukturen des Compilers in eine eigene Datenstruktur kopiert. Dort werden dann Analysen durchgeführt, die sämtliche Schleifen des Programms finden, innere sowohl als auch äußere. Sind diese Schleifen gefunden, so können Kandidaten erzeugt werden, für die sich ein Verschieben lohnt. Mit diesen Kandidaten, ist deren Speicherplatzbedarf und deren möglicher Energiegewinn erst einmal bekannt, kann leicht ein ILP aufgebaut werden und mit Hilfe von CPLEX gelöst werden. Damit erhält man die Lösung der Probleme, welche Blöcke zu welchen Zeitpunkten verschoben werden sollen. Anschließend werden noch Anpassungen am Programm notwendig, um die semantisch korrekte Ausführung im Scratchpad sicherzustellen. Dieser eher technische Teil kann nur zur Hälfte zur Laufzeit des Compilers ausgeführt werden, da ebenso Informationen des Linkers benötigt werden. Somit können die

Programmanpassungen erst zu einem späteren Zeitpunkt, hier mit Hilfe des Programms „enJumpCorrection“, durchgeführt werden. Diese Vielzahl von Anpassungen sind also notwendig, um eine dynamische Nutzung des Scratchpads zu ermöglichen.

4.5 Profiling

Der enCC-Compiler des Lehrstuhls besitzt in der aktuellen Version die Möglichkeit, die Ausführungshäufigkeiten einzelner Funktionen und Basicblöcke sowohl durch eine interne Control- und Daten-Flow-Analyse, als auch durch Profiling einer Simulation zu bestimmen.

Beide Verfahren besitzen dabei ihre Vor- und Nachteile, die gerade für das in dieser Diplomarbeit entwickelte Modul dynML von großer Bedeutung sind.

Die genaue Bestimmung der Ausführungshäufigkeiten ist gerade in diesem Fall besonders wichtig, da nur mit dieser Information entschieden werden kann, ob es sich lohnt, einen Basicblock in den Scratchpad zu verschieben, oder nicht. Dabei kann es durch den notwendigen Overhead des Kopierens von Programmcode und der benötigten Konstanten zu einem negativen Effekt auf den Gesamtenergiebedarf kommen, falls die vorhergesagte Ausführungshäufigkeit größer ist als die tatsächliche. Je nach der Größe des zu verschiebenden Basicblocks muss dieser zwischen mindestens 2 und 10 mal ausgeführt werden, um einen positiven Effekt zu bewirken. Wird also z.B. eine Ausführungshäufigkeit falsch mit 10 vorhergesagt und in Wirklichkeit wird der Block nur 3 oder 4 mal ausgeführt, so kann es unter Umständen zu einer Verschlechterung des Energiebedarfs kommen.

Die Vor- und Nachteile der einzelnen Vorgehensweisen sollen hier kurz vorgestellt werden:

Die Control- und Daten-Flow-Analyse besitzt die Vorteile, dass sie zweifelsfrei datenunabhängig ist und direkt im Compiler beim ersten Durchlauf durchgeführt werden kann. Dabei stößt dieses Verfahren bei rekursiven Aufrufen und Schleifenabbruchbedingungen, die datenabhängig sind, schnell an seine Grenzen und die vorhergesagten Ausführungshäufigkeiten können zum Teil erheblich von den realen Häufigkeiten abweichen. Dazu kommt mitunter ein erheblicher Implementierungsaufwand, um eine genaue Vorhersage zu ermöglichen. Aus den oben angegebenen Gründen kann dies zu negativen Effekten im Modul dynML führen.

Auf der anderen Seite unterliegt das Profiling nicht diesen Einschränkungen. Zwar ist eine direkte Ermittlung der Häufigkeiten während des Compilerlaufs nicht möglich, jedoch ist der Implementierungsaufwand dadurch geringer, dass das fertige, ausführbare Programm einfach in einem ersten Durchlauf auf die entsprechenden Häufigkeiten hin untersucht wird. In einem zweiten Compilerlauf liegen die im ersten Lauf gefundenen Werte dann als Ausführungshäufigkeiten vor und können vom Compiler für Optimierungsentscheidungen genutzt werden. Dieses Verfahren entspricht also viel mehr einem einfachen Abzählen der

Häufigkeiten als einer tiefergreifenden Analyse. Dieses Vorgehen konnte sehr einfach in den Profiler implementiert werden, da dort sowieso der gesamte Programmtrace untersucht wird. Der Vorteil ist somit einwandfrei die Einfachheit und Zuverlässigkeit dieser Methode.

Der große Nachteil des Profilings liegt dabei allerdings immanent in der Vorgehensweise begründet. Durch das Zählen der Ausführungshäufigkeiten des fertigen Programms sind die ermittelten Ausführungshäufigkeiten auf jeden Fall datenabhängig. Die Werte sind also nur für einen bestimmten Fall der Eingabemenge gültig. Damit ist dieses Verfahren alleine ohne Änderungen nicht uneingeschränkt nutzbar. Der Effekt der Datenabhängigkeit lässt sich allerdings dadurch abschwächen, dass man in mehreren Durchläufen verschiedene typischen Eingabemengen untersucht und dann über die ermittelten Werte mittelt.

Erste Tests haben allerdings schon gezeigt, dass bereits bei kleineren Zahlen von Testdurchläufen (3-5) die ermittelten Ausführungshäufigkeiten eine bessere dynamische Nutzung des Scratchpads durch das dynML-Modul ermöglichen als durch Control- und Datenflow-Analyse.

Wenn man sich das Ziel dieser Diplomarbeit vor Augen führt, den Energiebedarf eingebetteter Software zu minimieren, wobei man von einer Minimierung für die Mehrzahl der Ausführungen ausgehen kann und nicht einer „worst case“-Minimierung, so ist das hier vorgestellte Verfahren des Profilings für die Ermittlung der Ausführungshäufigkeiten vorzuziehen. Im ungünstigsten Fall kann es also zu einer Verschlechterung des Energiebedarfs kommen, falls die realen Häufigkeiten kleiner als die ermittelten sind. Dies sollte jedoch nur in Ausnahmefällen der Fall sein. Für die Mehrzahl der Eingabemengen für ein entsprechenden Programm wird so ein positiver Energieeffekt auftreten.

4.6 Sprunganpassung

Das Programm enJumpCorrection wird bei dem dynML-Verfahren notwendig, da der ARM-Assembler alle Sprungbefehle, die bei der ARM-Architektur relativ und nicht absolut sind, aus den Adressen, die sich aus den Labeln im Assembler-Code ergeben, berechnet.

Dabei kann der Assembler natürlich nicht wissen an welche Stelle im Scratchpad ein Basicblock verschoben werden soll. Somit berechnet der ARM-Assembler einen Sprung, der im Assemblercode zum Beispiel mit „BL LL1“ angegeben ist, so um, dass der Sprung an die Position des entsprechenden Basicblocks im Off-Chip-Speicher geht und nicht an die Adresse im Scratchpad. Diese Anpassung muss dann das enJumpCorrection-Programm vornehmen.

Die Anpassung geschieht dabei in der Form, dass der Sprungbefehl in der korrekten Weite als Bitmuster codiert wird und dann in die Assemblerdatei geschrieben wird. Dieses Bitmuster wird dann vom Assembler unverändert in das Object-File übernommen.

Neben der Anpassung dieser Sprünge müssen aber noch weitere Anpassungen stattfinden. So liegen die endgültigen Adressen der Basicblock erst nach dem Linken fest. Diese Adressen werden allerdings benötigt, damit die Kopierfunktion

auch den richtigen Speicherbereich in den Scratchpad verschiebt. Zusätzlich müssen auch noch Konstanten, die von verschobenen Blöcken benötigt werden, mitverschoben werden, damit sie vom Programm noch erreicht werden können. Die Befehle, die dabei auf die Konstanten zugreifen, müssen dabei auch noch angepasst werden, damit sie auf die richtigen Speicherstellen zugreifen.

Man sieht also, dass der Aufwand auch auf der Assemblerebene noch durchaus beachtlich ist, um eine korrekte Ausführung mit dynamischen Inhalten im Scratchpad sicherzustellen. Das im Laufe dieser Diplomarbeit erstellte Programm muss also einzelne Aspekte eines Assemblers, wie die Sprungberechnung, unter Umgehung des eigentlichen ARM-Assemblers realisieren.

Da eine genaue Beschreibung des enJumpCorrection-Programms hier zu weit führen würde, und auch nicht das Hauptaugenmerk dieser Diplomarbeit ist, wird an dieser Stelle auf den Anhang A verwiesen. Dort wird die komplette Funktionalität dieses Programms beschrieben.

4.7 Simulation

Die Simulation der vom enCC-Compiler erstellten Programme geschieht mit Hilfe einer Familie von Programmen, die unter dem Namen ARMulator (ARM-Instruction-Level Simulator) zusammengefasst werden.

Der ARMulator ist dabei ebenso wie der Assembler und Linker Bestandteil des ARM SDT, dem „Software Development Toolkit“, das von ARM für diese Hardware zur Verfügung gestellt wird.

Nachdem also der vom enCC-Compiler erstellte Assembler-Code vom ARM Assembler und Linker zu einer ausführbaren Objektdatei konvertiert wurde, kann diese Datei mit Hilfe des ARMulators simuliert werden. Zentraler Bestandteil des ARMulators ist das Programm *armsd*, der ARM Symbolic Debugger.

4.7.1 ARMulator

Mit dem ARMulator wird der Befehlssatz der ARM-Prozessoren auf einer vom ARM unterschiedlichen Hardware simuliert. Es wird dabei also ein ARM-Hardwaresystem in Software nachgebildet. Dies schließt die Simulation des Prozessors, der ROM/RAM-Bänke, der Adressbereiche, der Wartezyklen, der Geschwindigkeit der Peripherie etc. mit ein.

Der ARMulator besteht aus 4 Elementen [ARMu99]:

1. Prozessor-Core Modell: Dieses Modell bedient die Kommunikation mit dem Debugger.
2. Das Memorysystem: Es handelt sich um den Datentransfer zwischen dem ARM-Modell und dem Speicher-Modell. Dieses Modell kann auch ein Cachemodell beinhalten. Der ARMulator gibt die Information über den Datentransfer entweder zwischen dem Prozessor und Cache oder dem Prozessor und dem Hauptspeicher wieder.

3. Coprocessor Interface: Für den Fall, dass die Koprozessorbefehle ausgeführt werden.
4. OS Interface: Schnittstelle für verschiedene Betriebssysteme (Unix, Linux, Windows, etc.)

Wie erwähnt ist der ARMulator ein flexibles Programm zur Emulation der ARM-basierenden Systeme. Hier wird dieses Tool eingesetzt, um die für den ARM erstellten Programme zu simulieren und um eine Trace-Datei des Programmlaufs zu erzeugen, die dann vom enProfiler analysiert wird, um die Energiewerte für die Ausführung zu erhalten.

4.8 enProfiler

Ebenfalls für diese Diplomarbeit wird ein Programm benutzt, das im Rahmen einer anderen Diplomarbeit [SC00] am Lehrstuhl entstanden ist. Es handelt sich dabei um den Trace-Analyzer, der ständig weiterentwickelt wird und heute enProfiler genannt wird. Dieses Programm ist in der Lage, eine Bewertung des Energieverbrauchs und anderer wichtiger Eigenschaften wie Laufzeit (Anzahl der Taktzyklen), Speichergrößen (Daten und Programm), Anzahl der Zugriffe (getrennt nach Off- und On-Chip), Anzahl der Befehle usw. anhand eines Tracefiles für den ARM-Prozessor zu erstellen. Bei dem Tracefile handelt es sich um die simulierte Ausführung des ARM7TDMI-Prozessors durch den ARMulator (armsd, ARM Symbolic Debugger).

Die Grundlage für die Bewertung der Energiekosten bildet das Modell von Steinke et al. [SKW01], das auf den einzelnen vom Prozessor ausgeführten Instruktionen basiert, aber auch die Switching-Activity auf den Bussen, die für einen Großteil des Energiebedarfs verantwortlich ist, und den Energiebedarf in den Speichern mit berücksichtigt. Zusätzlich wird noch die Verwendung von verschiedenen funktionalen Einheiten im Modell mit berücksichtigt.

Neben der Aufgabe der Energieermittlung wird der enProfiler auch für die Ermittlung der Profilingdaten/Ausführungshäufigkeiten eingesetzt.

5 Ergebnisse

Im Rahmen dieser Diplomarbeit wurde das in den vorigen Kapiteln beschriebene Verfahren zur dynamischen Nutzung des Scratchpads erarbeitet und realisiert.

In diesem Kapitel sollen nun die mit diesem Verfahren erzielten Ergebnisse dargestellt und diskutiert werden. Zur Ermittlung der Ergebnisse wurde eine Benchmark-Suite verwendet, die aus 9 verschiedenen Programmen besteht. Alle Programme wurden mit dem neu erstellten Modul des enCC-Compilers compiliert und anschließend simuliert.

Bei den Programmen handelt es sich um Programme aus drei verschiedenen Bereichen.

Zunächst wurden mit den Programmen: Bubble-Sort, Heap-Sort, Quick-Sort, Selection-Sort und Insertion-Sort, verschiedene bekannte Sortieralgorithmen untersucht.

Zum zweiten Bereich zählen mit den Programmen Biquad-N-Section und Lattice-Filter Benchmarks, die häufig im Bereich der digitalen Signalverarbeitung eingesetzt werden.

Schließlich zählen auch ein Programm zur Matrixmultiplikation (Matrix_Mult) und eine weitere Applikation, die künstlich aus drei Sortieralgorithmen zusammengesetzt wurde und diese Algorithmen vergleicht (Multi_Sort), zur Benchmark-Suite. Bei dem Multi_Sort Programm handelt es sich also um ein konstruiertes Beispiel. Es wurde generiert, um ein Programm mit mehreren Hot-Spots[15] zu erhalten. Mit diesem Programm lässt sich das volle Potential des neuen Verfahrens besser verdeutlichen als mit den sonst relativ kleinen Programmen der Benchmark-Suite.

Im nachfolgenden Abschnitt werden die erzielten Ergebnisse des neuen Verfahrens grafisch dargestellt und mit Werten des alten, statischen Verfahrens und Werten einer Beispiel-Cache-Konfiguration verglichen.

Um diese Werte richtig deuten zu können, muss an dieser Stelle zunächst kurz darauf eingegangen werden, wie diese Werte ermittelt wurden:

statML
Beim statischen Verfahren wurden nur Basicblöcke und Funktionen zum Verlagern in den Scratchpad ausgewählt. Die Option, Variablen und den Stack ebenfalls in den Scratchpad zu verschieben, wurde deaktiviert.

dynML

Beim dynamischen Verfahren wurden ausschließlich Basicblöcke für das Verschieben in den Scratchpad ausgewählt. Durch die Generierung von

[15] Hotspots sind Programmteile, in denen prozentual gesehen viel Rechenzeit verbraucht wird.

Multiblöcken können quasi auch ganze Funktionen verschoben werden. Neben eventuell benötigten Konstanten werden keine Daten verlagert.

Cache

Beim Cache-System wurde ein 4-fach assoziativer Cache ausgewählt. Cache-Werte für die Cachegrößen 16 und 32 Bytes wurden dabei nicht ermittelt, da diese Größen für Caches ungewöhnlich klein wären und damit keinen fairen Vergleich erlauben würden.

Während es sich also beim Cache um einen kombinierten Instruktions- und Daten-Cache handelt, realisieren die beiden anderen Verfahren lediglich einen reinen Instruktions-Cache. Versuche, das Cachesystem ebenfalls als reinen Instruktions-Cache zu konfigurieren, scheiterten. Somit schneidet der Cache gerade bei größeren OnChip-Speichergrößen besser ab, da er auch Daten zwischenspeichert.

Für die Generierung der Werte wurden die Standardeinstellungen des enCC-Compilers benutzt. Lediglich das Profiling wurde für statML und dynML zusätzlich eingeschaltet. Bei den ermittelten Werten für statML und dynML wurden die Standardkonfigurationsdateien des enCC-Compilers verwendet. Da allerdings keine Konfigurationsdateien für Scratchpadgrößen von 16 und 32 Bytes zur Verfügung standen, weichen diese Werte ein wenig von den realen Werten nach unten ab. Die Abweichung ist allerdings als sehr klein zu bezeichnen. Ansonsten wirkt sich der Effekt, dass ein größerer Scratchpad mehr Energie pro Zugriff benötigt, voll auf die hier wiedergegebenen Werte aus und ermöglicht somit einen besseren Vergleich mit den ermittelten Cachewerten.

Anschließend werden die erzielten Ergebnisse, getrennt nach Energie und nach Performance, diskutiert. Auch wenn der Hauptaugenmerk hier die Energie ist und die Fähigkeit dieses Verfahrens, die vom Programm benötigte Energie zu reduzieren, so muss man zumindest auch kurz auf die Auswirkungen bezüglich der Performance eingehen.

Als nächstes soll dann das hier entwickelte Verfahren dem statischen Verfahren und dem Cache-System getrennt gegenübergestellt werden, wobei gesonderter Augenmerk auf die Vor- und Nachteile der einzelnen Verfahren gelegt werden soll.

Abschließend wird auf das Beispielprogramm Multi_Sort genauer eingegangen.

5.1 Ergebnisdarstellung

In diesem Kapitel werden nun die erzielten Ergebnisse des neuen Verfahrens dargestellt.

Aus Gründen der Übersichtlichkeit werden hier nur die Ergebnisse grafisch dargestellt und dann in den folgenden Kapiteln diskutiert. Die genauen ermittelten Werte sind noch einmal in tabellarischer Form im Anhang B wiedergegeben.

Für die Energiewerte gilt, dass die mit statML gekennzeichneten Werte die normal ermittelten Werte für das statische Verfahren darstellen. Für eine bessere Vergleichbarkeit mit dem dynamischen Verfahren sind allerdings noch zusätzliche Werte dargestellt, die zusätzlich die Kopierkosten für das erste Füllen des Scratchpads mit berücksichtigen. Diese Werte sind in der folgenden Grafik mit statML (*) gekennzeichnet. Diese Anpassung wurde hier eingeführt, da das statische Verfahren die Kosten für das erste und in diesem Fall auch einzige Füllen des Scratchpads nicht mit berücksichtigt. Die um diesen Faktor bereinigten Werte statML (*) wurden so erzeugt, dass in Abhängigkeit von der jeweiligen Scratchpadausnutzung eine Kostenpauschale errechnet wurde, die einem Verschieben durch eine Kopierfunktion entspräche, und dann anschließend auf den mit dem enProfiler ermittelten Wert aufgeschlagen wurde. Auch wenn diese Werte keine exakten Werte darstellen, so spiegeln sie zumindest die Größenordnungen wider und es wird ein fairer Vergleich der beiden Verfahren ermöglicht. Wie man in Abbildung 5.1-1 sieht, erzielt das statische Verfahren, wenn man die damit ermittelten Energiewerte um diesen Faktor anpasst, fast die identischen Ergebnisse wie das dynamische Verfahren. Dieser Effekt ist auch in dieser Form zu erwarten. Lediglich bei den Programmen Biquad-N-Sections und Quick-Sort kommt es zu kleinen Abweichungen. Bei Biquad-N-Sections resultiert der Unterschied zwischen dem statischen und dem dynamischen Verfahren, der in absoluten Werten auch nur sehr gering ausfällt, aus den Ungenauigkeiten, mit denen der Anpassungsfaktor errechnet wurde. Bei Quick-Sort resultiert der Unterschied aus einer Einschränkung in der Implementierung des dynamischen Verfahrens. Bei rekursiven Funktionsaufrufen werden bestimme Basicblöcke von einer Einlagerung in den Scratchpad ausgeschlossen. Diese ausgeschlossenen Basicblöcke verursachen hier diesen Effekt. Es handelt sich also nicht um eine Einschränkung des Verfahrens, sondern lediglich um eine Einschränkung in der gewählten Realisierung.

Die in den folgenden Grafiken hinter den Programmnamen dargestellte Byteanzahl repräsentiert die Programmgröße des entsprechenden Programms, falls kein Scratchpad zum Einsatz kommt. Die Größe bezieht sich dabei ausschließlich auf den Instruktionsteil des Programms und beinhaltet nicht die vom Programm benötigten Daten.

Die Ergebnisse werden getrennt nach Energie und Performance dargestellt:

5.1.1 Energiewerte

Die folgenden Grafiken geben die ermittelten Energiewerte der Benchmark-Suite wider:

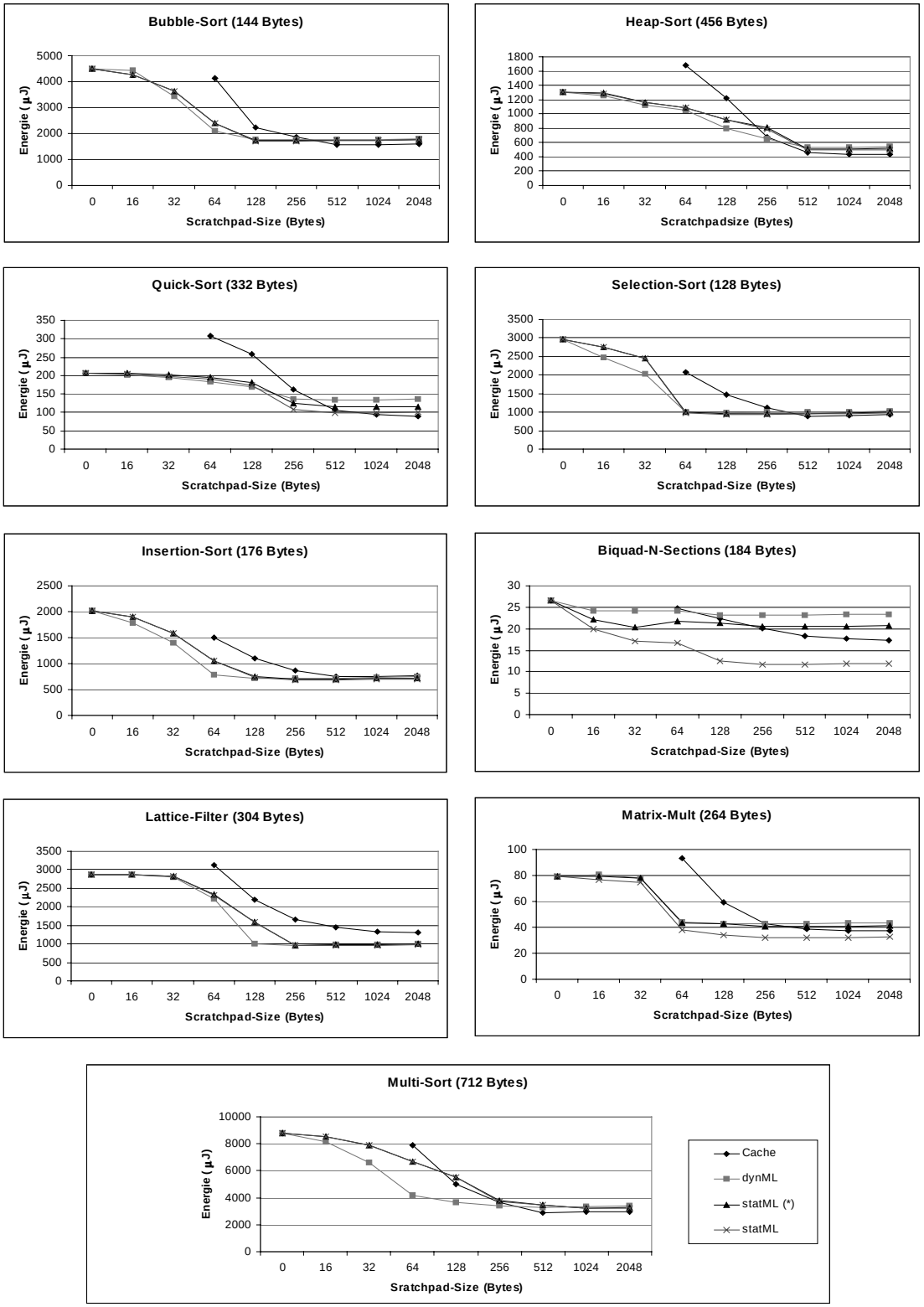

Abbildung 5.1-1 Energiewerte der Benchmark-Suite

5.1.2 Performancewerte

Die folgenden Grafiken geben die ermittelten Performancewerte wider:

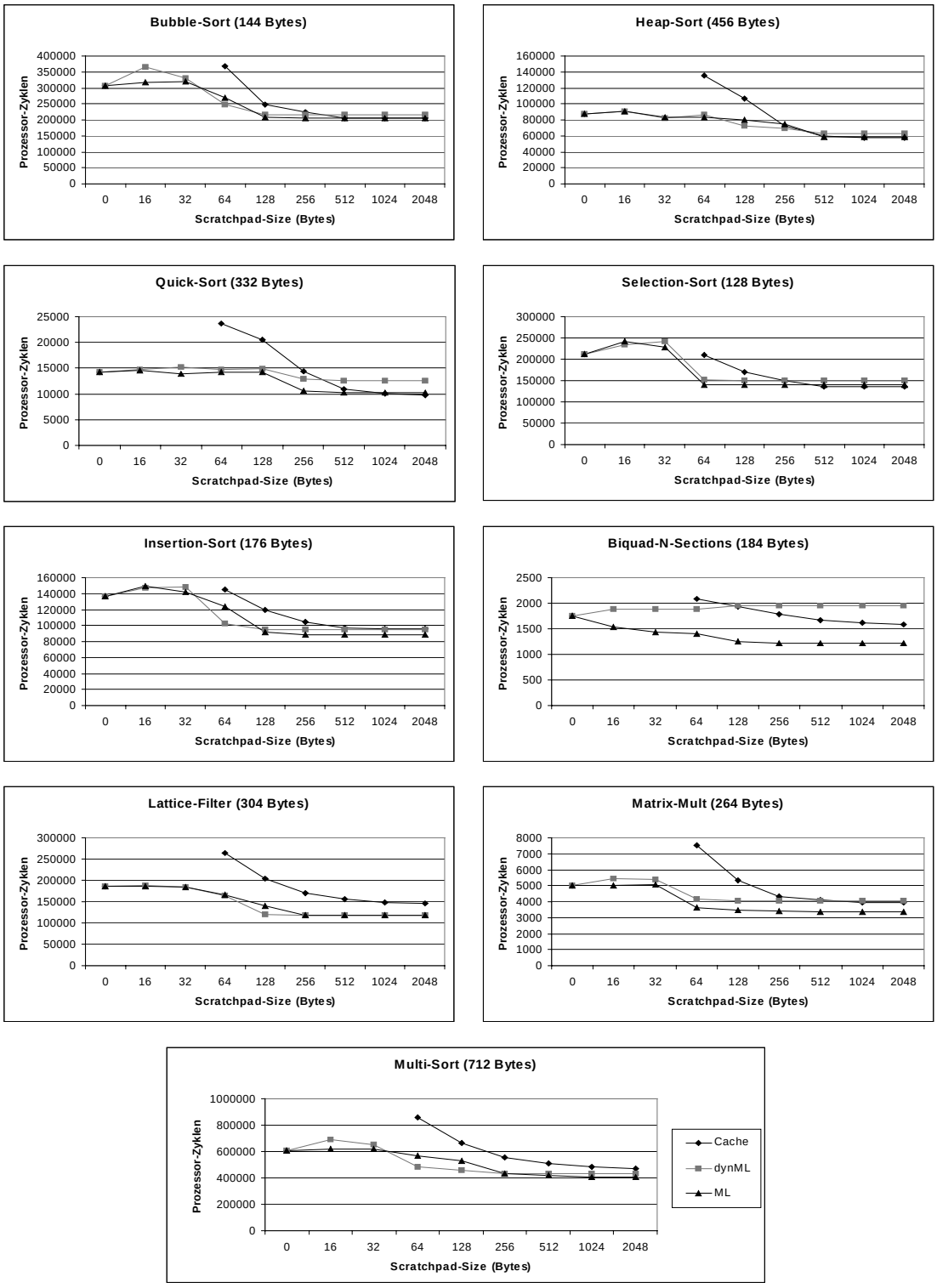

Abbildung 5.1-2 Performancewerte der Benchmark-Suite

5.1.3 Speicherplatzbedarf

An dieser Stelle soll eine Besonderheit des neuen Verfahrens nicht verschwiegen werden. Bei eingebetteten Systemen steht zwar eindeutig die verbrauchte Energie und die Performance im Vordergrund, doch kann mitunter auch die vom Programm benötigte Speicherplatzmenge von Interesse sein. Bei Systemen, die in sehr großen Stückzahlen produziert werden, kann eine Kosteneinsparung durch einen kleineren und dadurch günstigeren Speicherchip durchaus finanziell sehr interessant sein. Auf der anderen Seite werden viele Systeme heute schon so konzipiert, dass es auch nach der Produktion und sogar noch nach der Auslieferung eines Systems noch Änderungen an der Software geben kann. Aus diesem Grund werden die eingesetzten Speicher mit einer Platzreserve dimensioniert. Gründe hierfür sind, dass Hardware heute aus Standardkomponenten zusammengesetzt wird und es ganz einfach billiger, schneller und flexibler ist, die Hauptfunktionalitäten in Software zu realisieren. „Time-to-Market“ ist hier eines der Schlagworte. Ein weiterer Gesichtspunkt ist, dass man beim Einsatz eines Scratchpads im Gegensatz zu einem Cache Chipfläche einsparen kann und somit auch Kosten reduzieren kann. Es muss also untersucht werden, wie sich diese Kosten unterscheiden, wenn auf der einen Seite ein Cache eingesetzt wird und auf der anderen Seite ein Scratchpad, der weniger Chipfläche benötigt, aber dafür mehr Speicherplatz für das eigentliche Programm. Trotz des zusätzlich benötigten Speicherplatzes muss eine Scratchpad-Lösung also nicht wirtschaftlich ungünstiger als ein Cache-System sein.

Bei den hier untersuchten Programmen der Benchmark-Suite bedeutet dieser gesteigerte Speicherplatzbedarf im worst-case eine mehr als Vervierfachung (!) des benötigten Speichers im Fall des Quick-Sort Benchmarks. Im günstigen Fall der untersuchten Programme tritt ein eineinhalbfacher Speicherplatzbedarf auf.

Nicht unerwähnt in diesem Zusammenhang darf allerdings auch sein, dass der relative Speicherplatzzuwachs mit steigender Programmgröße geringer wird. So können die eingefügten Sprungbefehle, die Kopierfunktionen und die zusätzlich benötigten Konstanten bei kleinen Programmen einen größeren Anteil an der Gesamtgröße des Programms ausmachen als dies bei größeren Programmen in der Regel der Fall ist.

Wenn man zusätzlich bedenkt, dass die hier untersuchten Programme der Benchmark-Suite relativ klein sind, so relativiert dies auch die hier festgestellten Speicherbedarfszuwächse. Leider konnten aus Gründen der fehlenden Verfügbarkeit für den eingesetzten Compiler enCC keine wesentlich größeren Programme untersucht werden. Es ist jedoch davon auszugehen, dass bei realistischen Verhältnissen der Speicherplatzbedarf immer noch um den Faktor 1,5-2 ansteigen kann. Dabei ist der Anstieg allerdings sehr stark abhängig von der Struktur des Programms und nur in geringerem Maße von der Größe des Scratchpads oder von der Gesamtgröße des Programms.

Wie die folgende Grafik anzeigt, kann der Speicherplatzbedarfszuwachs beim Einsatz des neuen, dynamischen Verfahrens durchaus immens sein, während beim statischen Verfahren der Zuwachs an Platzbedarf wesentlich geringer ausfällt. Der Einsatz eines Cache-Systems ergibt keine Änderung am Speicherplatzbedarf eines Programms:

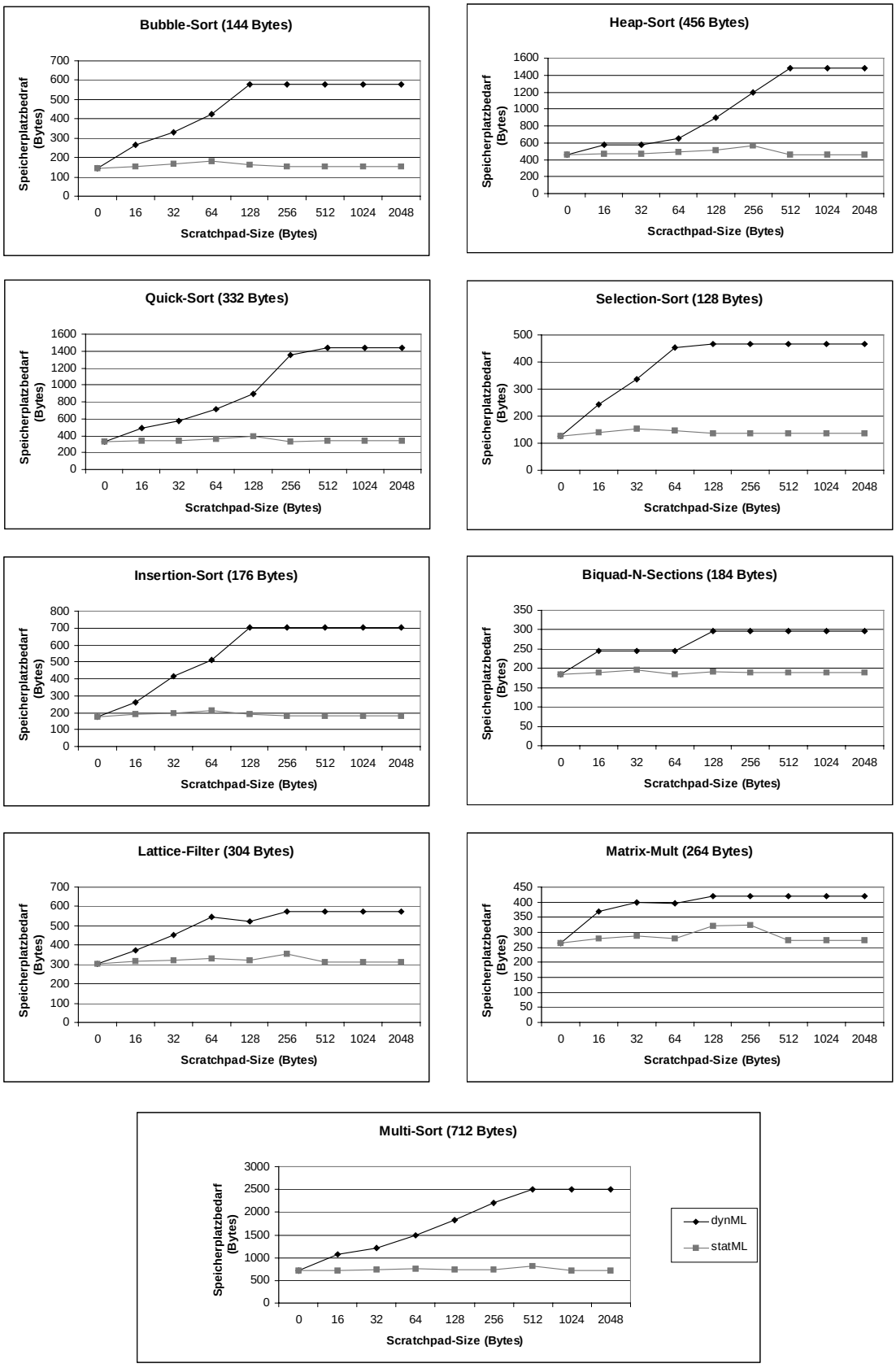

Abbildung 5.1-3 Speicherplatzbedarf der Benchmark-Suite

5.1.4 Anzahl ausgeführter Instruktionen

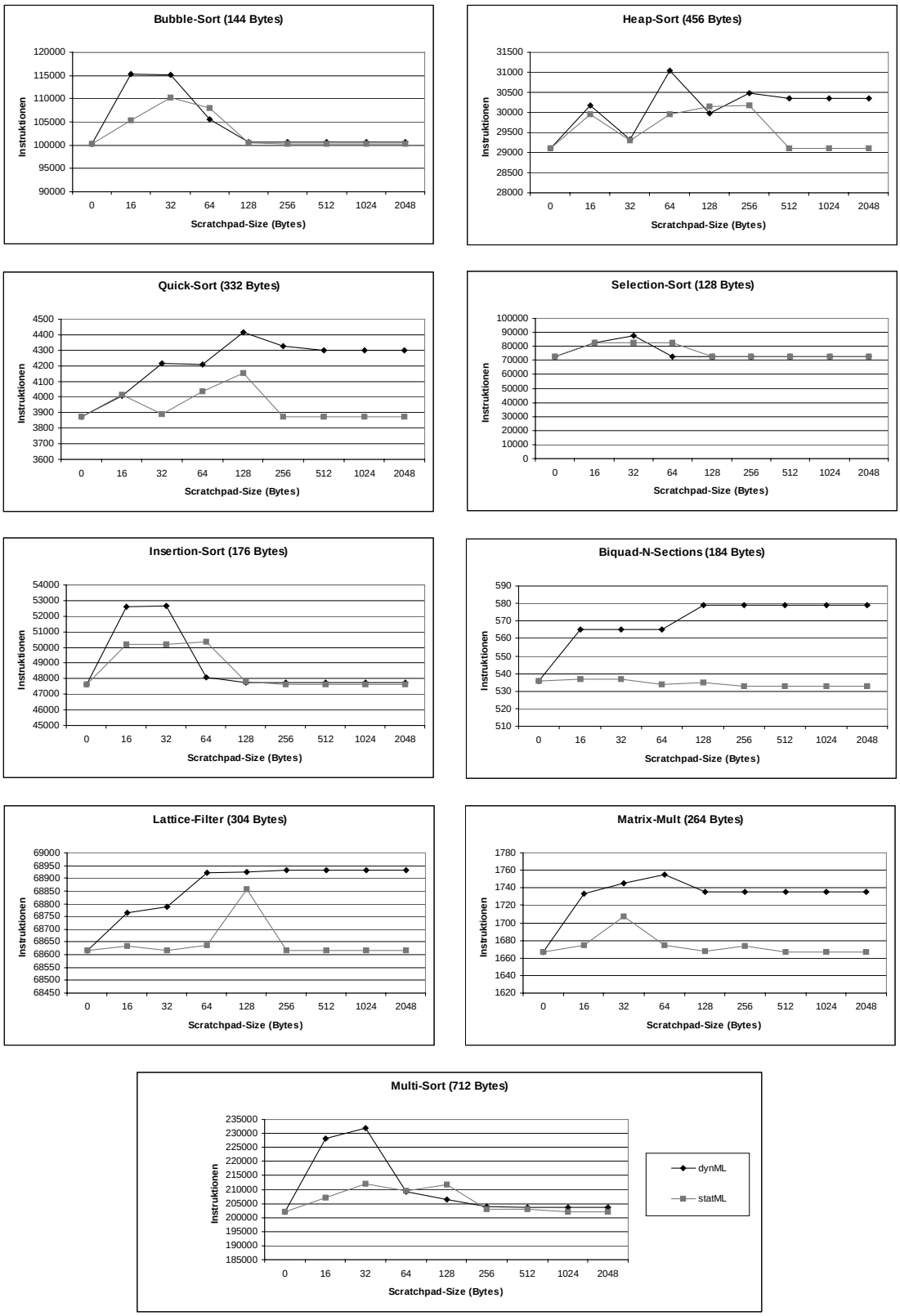

Abbildung 5.1-4 Ausgeführte Instruktionen der Benchmark-Suite

Die Statistik über ausgeführte Instruktionen wurde nur aus Gründen der Vollständigkeit mit aufgeführt. Die Ergebnisse repräsentieren dabei genau das, was man beim Einsatz der beiden Verfahren erwarten durfte.

Die Werte für das Cache-System wurden nicht mit in die Statistik aufgenommen, da ein Cache keinen Einfluss auf die Anzahl ausgeführter Instruktionen besitzt. Die Anzahl der Instruktionen ist im Fall eines Cache-Systems also immer gleich dem Fall, dass überhaupt kein Cache oder Scratchpad eingesetzt wird.

Für beide Verfahren gilt, dass die Anzahl der ausgeführten Instruktionen immer größer oder gleich der Anzahl der Instruktionen im Fall, dass kein Scratchpad genutzt wird, ist. Dieser Umstand ergibt sich aus der Tatsache, dass in beiden Fällen zusätzliche Instruktionen eingeführt werden müssen, um eine korrekte Ausführung des Programm sicherzustellen. Im Falle von statML werden lediglich Sprünge eingeführt. Im Falle von dynML allerdings auch noch die Kopierfunktionen selbst. Da beim statischen Verfahren mit wachsender Scratchpadgröße allerdings immer mehr Teile im Scratchpad landen, werden immer weniger Sprünge zwischen den verschiedenen Speicherformen notwendig. Liegt dann erst einmal das komplette Programm im Scratchpad, so werden überhaupt keine zusätzlichen Sprünge mehr benötigt und die Anzahl der Instruktionen ist gleich der Situation ohne Scratchpad. Die Zahl der ausgeführten Instruktionen steigt also zunächst einmal bis zu einer gewissen, programmabhängigen Größe an und sinkt anschließend wieder ab.

Auch im dynamischen Fall lässt sich eine ähnliche Beobachtung machen. Auch hier steigt die Anzahl der Instruktionen in den meisten Fälle zunächst stark an und fällt danach wieder ab. Da aber immer die Kopierfunktionen im Programm unabhängig von der Scratchpadgröße zusätzlich ausgeführt werden müssen, bleibt die Gesamtzahl der ausgeführten Instruktionen bis auf wenige Ausnahmen über der Instruktionszahl des statischen Verfahrens.

5.2 Energiebetrachtungen

Um eine bessere Einschätzung der Energieauswirkungen des neuen Verfahrens machen zu können, wurden die erzielten Energiewerte in der nachfolgenden Abbildung 5.2-1, die die Werte abhängig von der Scratchpad-Größe darstellt, auf den Energiewert ohne den Einsatz des Verfahrens normalisiert auf 100%.

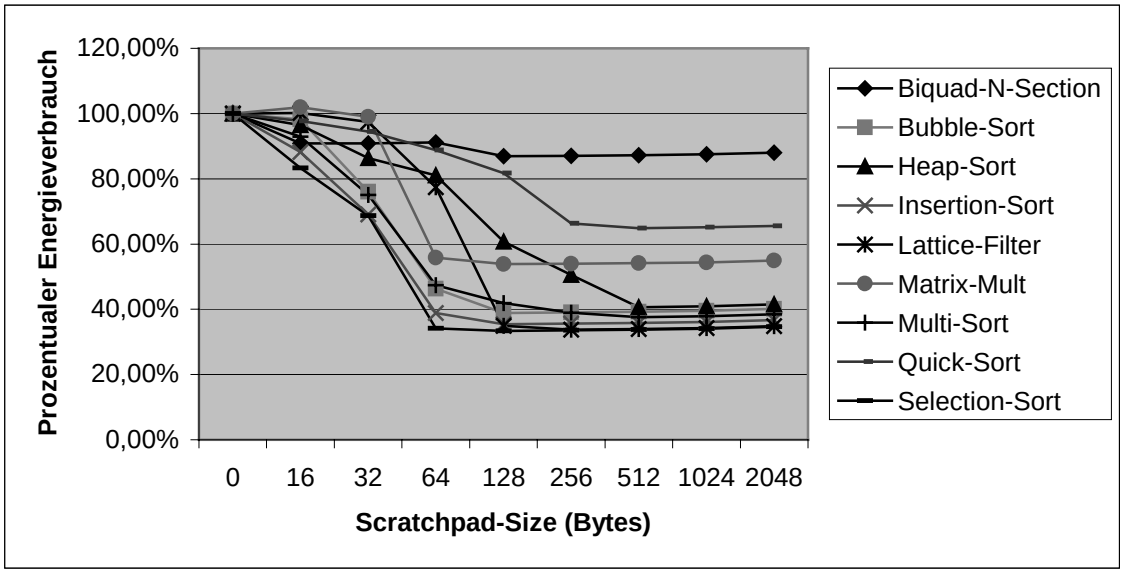

Abbildung 5.2-1 Prozentualer Energieverbrauch mit dem dynML-Verfahren

Wie man aus dieser Grafik sehr einfach ablesen kann, erzielt das neue Verfahren bei sämtlichen Programmen der Benchmark-Suite eine Energiereduktion, die je nach Programm zwischen ca. 12% und 70% liegt. Gemittelt über sämtliche Programme beträgt die Energiereduktion ungefähr 53%.

Zweidrittel der Programme erzielen dabei einen Gewinn, der weit über dem Durchschnitt von 50% liegt. Die Programme Biquad-N-Section, Quicksort und Matrix-Mult liegen dabei schlechter als der Durchschnitt. Die Begründung für diese Tatsache liegt dabei eindeutig darin, dass die Programme Biquad-N-Section und Matrix-Mult relativ klein sind und der Programmcode zu einem Grossteil aus Instruktionen besteht, die nicht wiederholt in einer Schleife ausgeführt werden. Insofern kann das dynML-Verfahren hier keinen Gewinn erzielen, da die Kopierkosten größer wären als die Energieeinsparung bei der Ausführung im Scratchpad. Das Programm Quicksort wiederum bildet eine Ausnahme, da es das einzige Programm mit rekursiven Funktionsaufrufen ist. Auch hier kann das neu entwickelte Verfahren nicht so effektiv den Scratchpad ausnutzen.

Dieser Umstand ist zum Teil allerdings auch durch implementierungstechnische Gründe entstanden. Aus Gründen der einfacheren Umsetzung werden bei rekursiven Funktionsaufrufen Teile des Programms nicht in den Scratchpad verschoben, um eine einfachere Handhabung der Rekursionsabbrüche und

Rücksprungadressen zu gewährleisten. Mit einem stark erhöhten Implementierungsaufwand könnte man diese Einschränkung beseitigen und somit den Energieverbrauch dieses Programms noch weiter verbessern.

Die Tatsache, dass rekursive Funktionen unter Umständen nicht sehr gut von dynML verarbeitet werden, stellt allerdings keine schwere Einschränkung dar, da sämtliche rekursive Funktionen in lineare Funktionen überführt werden können. Der Einfluss auf diese hier dargestellten Ergebnisse wirkt sich auch nur auf den Quicksort-Algorithmus aus.

Ebenfalls auffällig in der Grafik ist, dass ab einer bestimmten Scratchpadgröße keine Änderungen mehr bei den Energiewerten auftreten. Zu diesem Zeitpunkt sind dann alle Basicblöcke, für die ein Gewinn bei der Ausführung im Scratchpad erzielt werden kann, bereits in den Scratchpad verschoben worden. Wichtig hierbei ist, dass diese Speichergröße, an dem die Energiekurve gegen einen festen Wert konvergiert, bei allen Programmen wesentlich kleiner ist, als die Gesamtgröße des Programms. Das heißt, dass die optimale Lösung bereits bei einem in Relation zur Programmgröße relativ kleinen Scratchpad erreicht wird.

Als letzter Punkt soll hier noch erwähnt werden, dass bei sehr kleinen Scratchpadgrößen ein negativer Energieeffekt auftreten kann. Dieser entsteht durch den Overhead der Kopierfunktionen, der nur abgeschätzt und nicht genau berechnet werden kann. Dieser Effekt, der nur bei einer Scratchpadgröße von 16 Bytes und auch nur bei zwei Programmen der Benchmark-Suite auftritt, ist mit weniger als einem Prozent der Gesamtenergie allerdings als sehr gering zu bezeichnen.

Von dieser Besonderheit abgesehen erzielt das dynML-Verfahren, wie nicht anders zu erwarten, eine monoton fallende Energiekurve bei steigender Scratchpadgröße. Man kann also fest davon ausgehen, dass bei einem größeren Scratchpad die benötigte Energie geringer wird, oder aber im ungünstigsten Fall gleich bleibt, solange der zusätzliche Speicher von Programmteilen genutzt wird.

Wird der Scratchpad vergrößert, ohne das der neu hinzugekommene Speicher überhaupt genutzt wird, so wird sich der Energiebedarf des Programms erhöhen. Dieser Effekt resultiert aus der Tatsache, dass ein größerer Scratchpad auch mehr Energie pro Zugriff benötigt. In der Grafik 5.1-1 spiegelt sich das so wider, dass in allen Fällen bei großen Scratchpadgrößen der Energiebedarf wieder ein wenig ansteigt, da bereits alle zu verschiebenden Programmteile in einen kleineren Scratchpad passen würden.

5.3 Performancebetrachtungen

Bei der Diskussion der Performance soll von einer ähnlichen Darstellung, wie auch schon bei der Betrachtung der Energie, ausgegangen werden. Auch hier wurde aus Gründen der Übersichtlichkeit die Performance auf 100% der Ausführung ohne den Einsatz dieses Verfahrens normiert und in Abhängigkeit von verschiedenen Scratchpadgrößen dargestellt. So kommt man zu der Darstellung in Abbildung 5.3-1. Schon auf den ersten Blick ist dabei auffallend, dass die Performancefunktionen

der einzelnen Programme nicht monoton fallend sind, so wie dies bei der benötigten Energie der Fall ist.

Im ungünstigsten Fall des Programms Biquad-N-Sections kann es sogar bei maximaler Scratchpadgröße zu einer Verschlechterung der Performance kommen. Dieser Umstand ist um so bemerkenswerter, wenn man bedenkt, dass hier eine Performanceverschlechterung von über 10% mit einer Energiereduktion von über 10% einhergeht. Auch bei anderen Programmen der Benchmark-Suite kann dieser Effekt der reduzierten Performance bei kleineren Scratchpadgrößen auftreten.

Dieser Umstand einer auftretenden Performanceverschlechterung bei gleichzeitiger Energieeinsparung ist hierbei besonders bemerkenswert. Geht man doch im Normalfall davon aus, dass eine Performanceverbesserung auch eine Energiereduktion hervorruft und umgekehrt. So beschäftigen sich die meisten Forschungsarbeiten, die zum Teil auch in der Einleitung erwähnt werden, hauptsächlich mit Methoden der Performanceverbesserung, um eine Energiereduktion zu erzielen. Der Umkehrschluss, dass eine Energiebedarfsverbesserung auch eine Performanceverbesserung bewirkt, ist allerdings nicht immer richtig, wie dieses Verfahren hier eindeutig verdeutlicht, zumindest für kleinere Scratchpadgrößen.

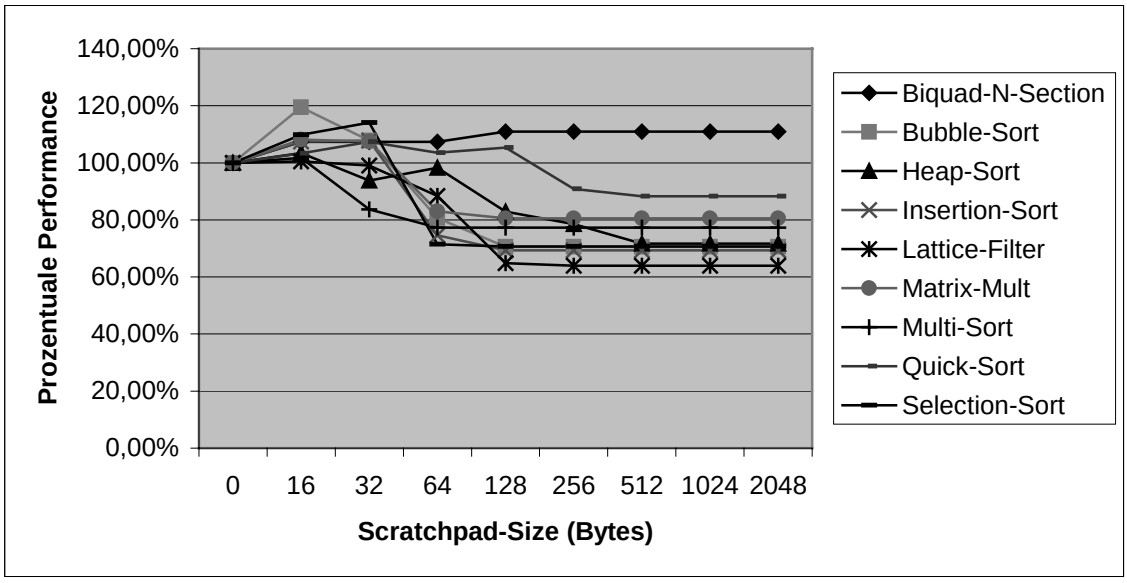

Abbildung 5.3-1 Prozentualer Performancegewinn mit dem dynML-Verfahren

Wenn man nun aber das Extrembeispiel Biquad-N-Section außer Acht lässt und nur die eingependelten Werte betrachtet, so erzielt dieses Verfahren aber doch auch einen Performancegewinn zwischen ungefähr 11% und 38%.

In Relation zum erzielten Energiegewinn von bis zu 70% fällt diese Performanceverbesserung allerdings eher gering aus.

Festzustellen bleibt hier also abschließend, dass die Performanceauswirkungen des neuen Verfahrens nicht in einer Weise voraussehbar sind, wie dies bei der Energie der Fall ist. Im ungünstigsten Fall kann es sogar zu einer Verschlechterung kommen, im Durchschnitt allerdings gibt es eine geringere Performance-verbesserung in Relation zur Energieeinsparung.

Zusätzlich erwähnt werden soll an dieser Stelle noch, dass der enCC auch die Option beinhaltet, nicht nur den Energieverbrauch, sondern auch die Performance eines Programms zu optimieren. Mit dem hier implementierten Verfahren wurde auch diese Option getestet. Die Ergebnisse fielen dabei allerdings bei 8 der 9 Programme genauso aus wie bei Verwendung der Energieoptimierung. Beim einzigen Programm, bei dem ein Unterschied auszumachen war, machte dieser auch nur einen Bruchteil eines Prozentpunktes an der Gesamtzyklenzahl der Programmausführung aus. Die Ursache für diesen geänderten Wert bei einem der Programme liegt allerdings nicht im dynML-Verfahren begründet, sondern liegt an einer veränderten Code-Auswahl des Compilers, die noch vor dem dynamischen Verfahren durchgeführt wird.

5.4 Statisches vs. Dynamisches Verfahren

Beim Vergleich des statischen Verfahrens statML mit dem hier neu entwickelten dynML-Verfahrens sollen zunächst die entsprechende Abbildung für die Energiewerte des statischen Verfahrens vorgestellt werden. Die Abbildung 5.4-1 entspricht dabei der Abbildung 5.2-1 beim dynamischen Verfahren.

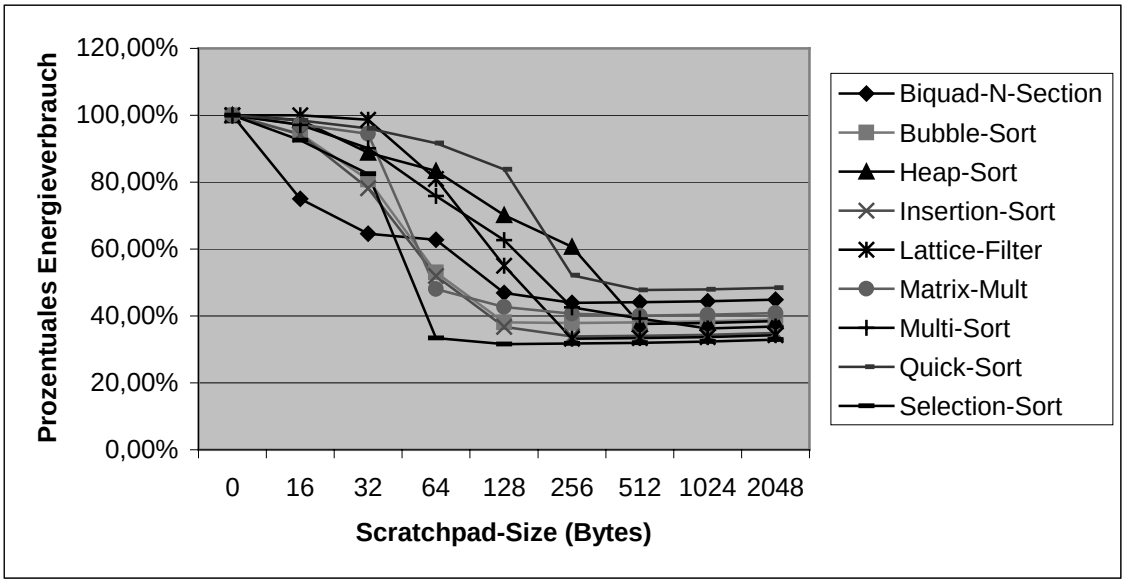

Abbildung 5.4-1 Prozentualer Energieverbrauch mit dem statML Verfahren

Wenn man nun also die Auswirkungen auf die Energie für beide Verfahren vergleicht, so fällt auf, dass zunächst einmal auch für das statML-Verfahren gilt,

dass die Energiekurve ebenfalls monoton fallend ist. Allerdings werden die Minimalwerte erst bei größeren Scratchpadgrößen erreicht. Dies ist darin begründet, dass beim statischen Verfahren keine Kosten für das Verschieben von Basicblöcken in den Scratchpad auftreten, da die Blöcke bereits vom Loader an die richtigen Positionen im Speicher geladen werden und dann nie die Position ändern. Beim statischen Verfahren können also alle Basicblöcke in den Scratchpad verschoben werden, während beim dynamischen Verfahren nur die Blöcke in den Scratchpad geladen werden, die mehrmals hintereinander ausgeführt werden. Der maximale Energiegewinn ergibt sich also beim statischen Verfahren erst dann, wenn das gesamte Programm in den Scratchpad verschoben wurde. Das dynamische Verfahren dagegen erzielt das beste Ergebnis bereits erheblich früher.

Dadurch, dass das statische Verfahren, wie gesagt, das gesamte Programm verschieben kann, falls der Scratchpad groß genug ist, ist natürlich auch der maximale Energiegewinn erheblich größer als beim dynamischen Verfahren. Diese Tatsache ist allerdings bei realen Anwendungen eher untypisch. Bei größeren Programmen wird der zur Verfügung stehende Scratchpad-Speicher im Vergleich zur Programmgröße um ein vielfaches kleiner sein.

Und gerade bei diesen Größenverhältnissen kann das dynamische Verfahren gegenüber dem statischen Verfahren Punkte sammeln. Durch den Austausch der Inhalte des Scratchpads kann das dynamische Verfahren mehr Blöcke verschieben als das statische Verfahren. Wenn dann der Gewinn der zusätzlich ausgetauschten Blöcken größer als der Aufwand ist, der durch das Verschieben entsteht, so kann das dynamische Verfahren ein besseres Ergebnis erzielen als das statische bei gleicher Scratchpadgröße.

Die folgende Abbildung 5.4-2 verdeutlicht diesen Sachverhalt. In dieser Abbildung wurde die Energiekurve gemittelt über alle untersuchten Programme für die beiden Verfahren dargestellt. Da das dynML-Verfahren bei 3 Programmen der Benchmark-Suite nur einen sehr geringen Gewinn erzielen kann, ist der angesprochene Effekt nicht sehr deutlich erkennbar. Dennoch erkennt man, dass das dynamischer Verfahren über alle Programme gemittelt bei einer Scratchpadgröße von 32 und 64 Bytes weniger Energie benötigt als das statische Verfahren. Bei einer Größe von 16 und 128 Bytes sind die Werte ungefähr gleich, bei allen anderen Speichergrößen allerdings zum Teil wesentlich schlechter. Man sieht, das dynML im Schnitt maximal nur etwas über 53% Energie einsparen kann, während statML im Schnitt immerhin etwa 61% einsparen kann.

Die durchschnittliche Programmgröße gemittelt über die Benchmark-Suite beträgt ca. 275 Bytes, wenn man nur den Instruktionsanteil der Programme betrachtet.

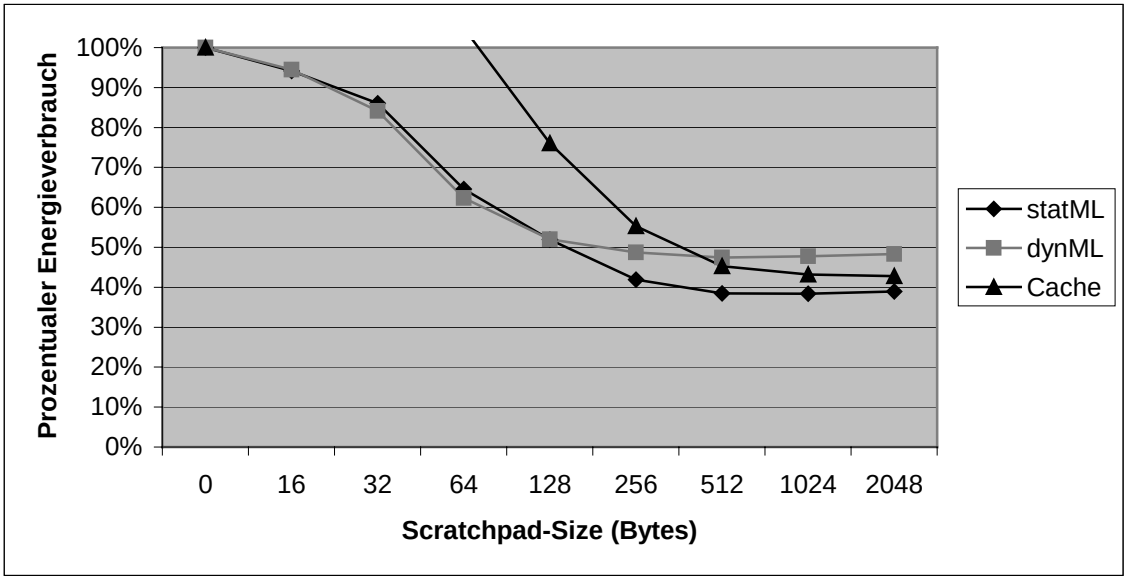

Abbildung 5.4-2 Gemittelter Energieverbrauch der gesamten Benchmark-Suite

Neben den Auswirkungen auf die Energie sollen hier auch kurz die Auswirkungen auf die Performance verglichen werden. Die folgende Abbildung 5.4-4 verdeutlicht die Auswirkungen auf die Performance. Auch beim statML-Verfahren ergibt sich eine schlechte Vorhersagbarkeit über die Performance, da auch hier keine monotonen Verläufe zu beobachten sind. Insgesamt sind die Auswirkungen aber sehr ähnlich den Beobachtungen beim dynamischen Verfahren. Lediglich die maximalen Performanceverbesserungen sind im Schnitt ungefähr 16% günstiger.

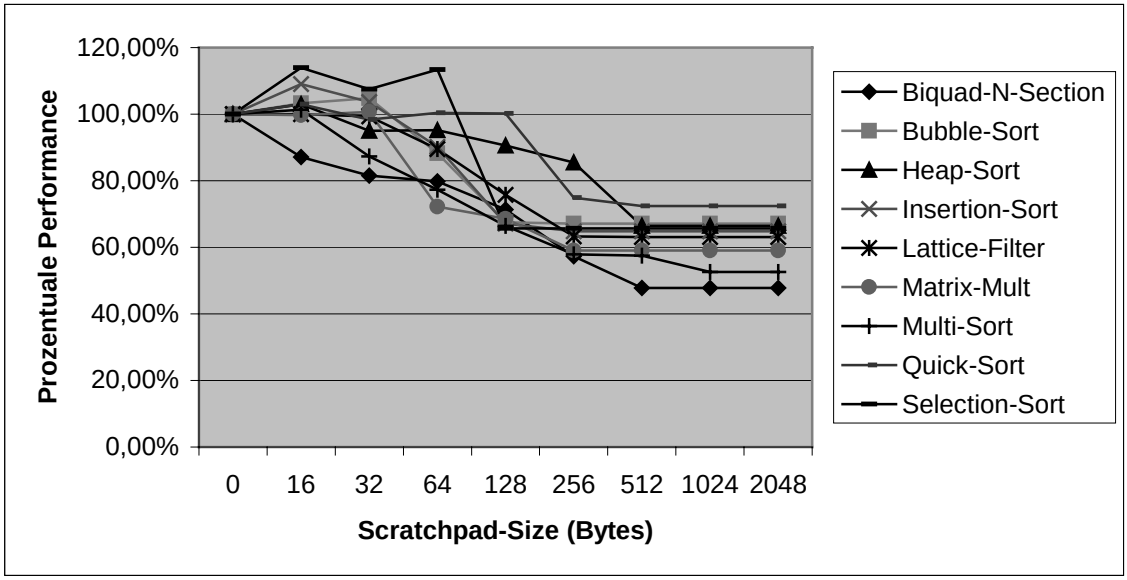

Abbildung 5.4-3 Prozentuale Performance des statML-Verfahrens

Wenn man nun auch hier die gemittelten Performancewerte aller Benchmarks für die beiden Verfahren gegenüberstellt, ergibt sich eine Grafik, die verblüffende Ähnlichkeit mit der entsprechenden Grafik der Energiewerte aufweist:

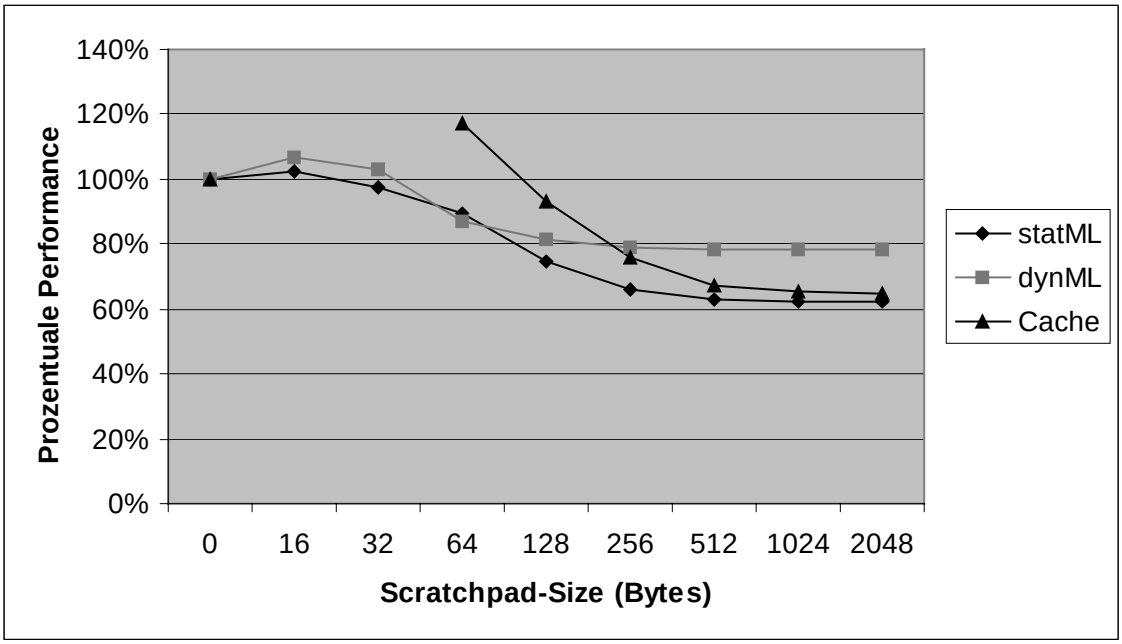

Abbildung 5.4-4 Gemittelter Performancegewinn der gesamten Benchmark-Suite

Es lässt sich also festhalten, dass die Auswirkungen, die das dynamische Verfahren im Vergleich zum statischen Verfahren auf die Energie hat, sehr ähnlich den Auswirkungen auf die Performance sind.

Die Schlussfolgerung hieraus ist also, dass in dem Bereich, in dem das dynamische Verfahren dem statischen Verfahren im Hinblick auf die vom Programm benötigte Energie zu bevorzugen ist, auch mit einer besseren Performance zu rechnen ist.

5.5 Cache vs. Dynamisches Verfahren

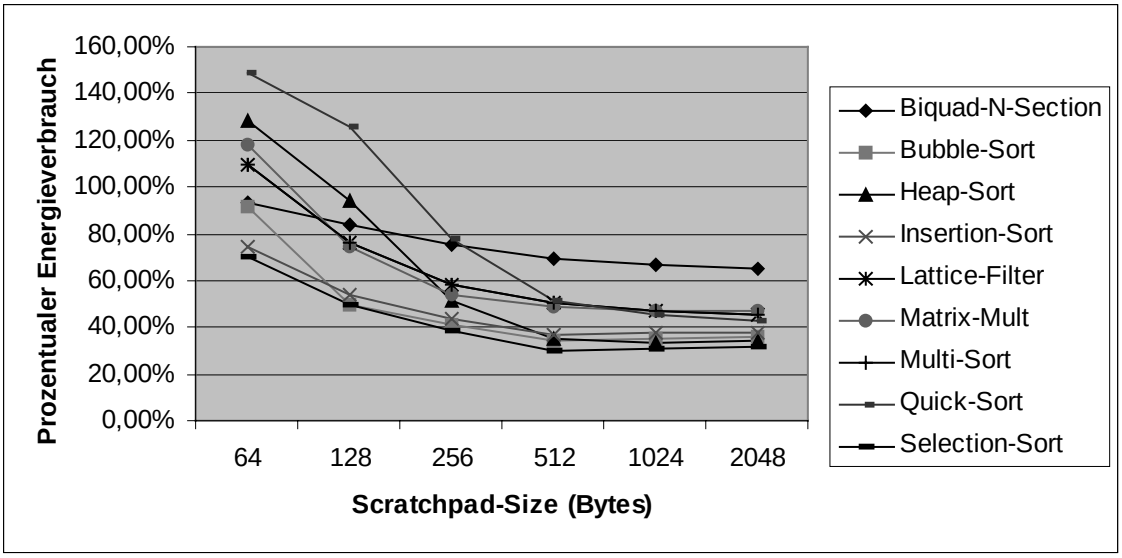

Abbildung 5.5-1 Prozentualer Energiegewinn mit einem Cache

Über die gesamte Benchmark-Suite gemittelt erreicht das Cache-System eine Energiereduktion von ca. 57%. Bei den Energiewerten schneidet das Cache-System allerdings erst ab einer Cache-Größe von 512 Bytes besser ab (Abb. 5.4-2) als das dynamische Verfahren. Nicht außer Acht lassen darf man in diesem Zusammenhang allerdings, dass es sich beim Cache um einen Daten- und Instruktions-Cache handelt. Die größeren Energieeinsparungen, gerade bei größeren Cache-Größen, können daher durchaus aus dem zusätzlichen Zwischenspeichern von Daten resultieren. Durch diesen Umstand fällt es hier schwer zu urteilen, ob der Cache oder das dynamische Verfahren einen höheren Energiegewinn bei großen OnChip-Speichergrößen erzielt. Bei den kleineren Speichergrößen unter 512 Bytes erzielt das dynamische Verfahren allerdings im Schnitt ein besseres Ergebnis.

Bei den Werten des Cache-Systems fällt ebenfalls auf, dass die Energiewerte aller Programme prozentual gesehen für alle Cache-Größen sehr ähnlich sind. Lediglich das Programm Biquad-N-Section, dass aufgrund seiner sehr geringen Größe auch bei den beiden anderen Verfahren schlechter abschnitt, fällt auch hier aus dem

Rahmen. Das Cache-System scheint hier also keine so großen Schwankungen im Ergebnis zu erzielen und dieser Umstand resultiert hier in einer besseren Vorhersagbarkeit der Ergebnisse, die durch einen Cacheeinsatz zu erzielen sind.

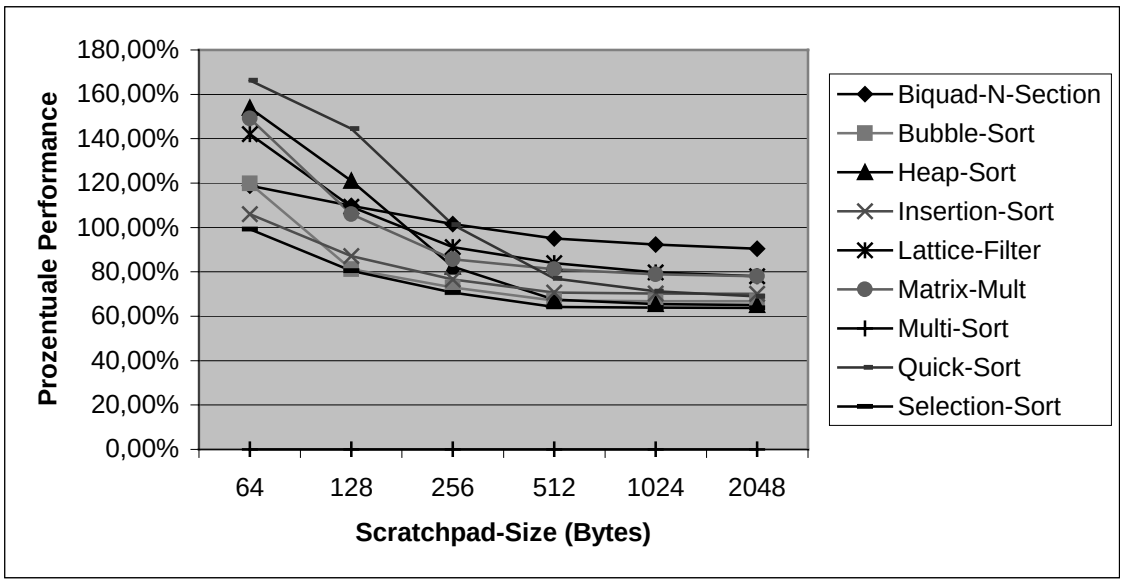

Abbildung 5.5-2 Prozentualer Performancegewinn mit einem Cache

Über die gesamte Benchmark-Suite gemittelt erreicht das Cache-System eine Performancesteigerung von ca. 37%. Das dynamische Verfahren erreicht dagegen lediglich eine Performanceverbesserung von ca. 22%. Allerdings schneidet der Cache auch hier erst ab einer Scratchpadgröße von 256 Bytes besser ab (Abb. 5.4-4).

Zusammenfassend lässt sich also feststellen, dass bei kleinen Scratchpadgrößen das dynamische Verfahren gegenüber dem hier untersuchten Cache-System Vorteile, sowohl was den Energiebedarf als auch die Performance angelangt, aufweist. Selbst wenn man nicht berücksichtigt, dass es sich beim Cache um ein System handelt, das Daten und Instruktionen zwischenspeichert und nicht nur Instruktionen, kommt man zu diesem Ergebnis.

5.6 Multi_Sort

Abschließend soll an dieser Stelle noch einmal gesondert auf das Programm Multi_Sort eingegangen werden. Dieses Programm, das zwar ein konstruiertes Beispiel darstellt, besitzt mehr Hot-Spots als die anderen Programme der Benchmark-Suite. Folglich besitzt das dynML-Verfahren mehr Möglichkeiten, einzelne Teile im Scratchpad vermehrt auszutauschen. Das Ergebnis stellt die folgende Grafik dar:

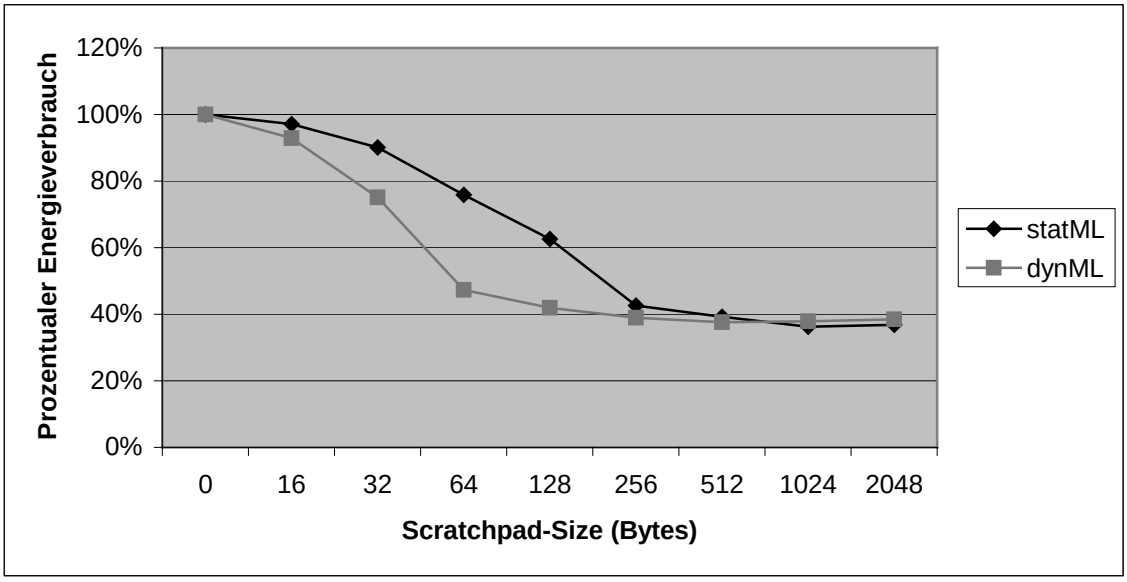

Abbildung 5.6-1 Energieverbrauch: Beispiel Multi-Sort

In dieser Abbildung sieht man nun sehr deutlich, in welchen Größenbereichen das dynamische Verfahren besser abschneiden kann als das statische. Wenn man nun weiterhin berücksichtigt, dass mit Multi_Sort versucht wurde die Komplexität realer Anwendungen nachzubilden, wird deutlich, wo das dynML-Verfahren durchaus eine Verbesserung des Energiebedarfs für solche Anwendungen darstellen kann.

In diesem Beispiel schafft das dynamische Verfahren im günstigsten Fall eine maximale Verbesserung des statischen Verfahrens um immerhin ca. 30% bei einer Scratchpadgröße von 64 Bytes. Im ungünstigsten Fall, dass das gesamte Programm in den Scratchpad passt, verliert das dynamische Verfahren gerade einmal ca. 2% auf das statische Verfahren. Diese 2% Prozent machen genau den Energieaufwand aus, der benötigt würde, um den Scratchpad beim statischen Verfahren einmal komplett zu laden. Unter solchen Rahmenbedingungen kann man also propagieren, dass das dynamische Verfahren in realistischen Scratchpadgrößen- und Programmverhältnissen eine Verbesserung des statischen Verfahrens darstellt.

6 Zusammenfassung und Ausblick

In den vergangenen Jahren war die Geschwindigkeit fast ausschließlich der Maßstab in der Computerindustrie. Durch die gesteigerte Ausführungsgeschwindigkeit wurden in letzter Zeit viele neue Einsatzbereiche von Computersystemen überhaupt erst realisierbar. Das wachsende Interesse der Konsumenten und der harte Konkurrenzkampf in dieser Sparte führten zu einer rapiden Entwicklung neuer Hardware mit immer mehr Funktionalitäten. Ein Ende dieser Entwicklung ist heute noch nicht abzusehen.

Allerdings geht immer schnellere Hardware mit mehr Funktionen auch meistens mit einem gesteigerten Energiebedarf einher. Gerade bei mobilen Geräten, die einen immer größeren Marktanteil erreichen, wird dieser Umstand immer problematischer, da eine autarke Energiequelle heute immer eine begrenzte Kapazität bedeutet, wenn man einmal von der nur sehr selten in diesem Bereich eingesetzten Solartechnik absieht. Die Hard- und Softwareindustrie arbeiten mit Hochdruck an einer Lösung dieses Problems. Auch die Batteriehersteller sind bemüht, die Kapazität ihrer Akkus immer weiter zu erhöhen. Auf der Hardwareseite ist dabei meist mit hohen Entwicklungszeiten zu rechnen.

Ein vielversprechender Lösungsansatz besteht im Einsatz von Cache- und Scratchpad-Speichern, wobei bei kritischen Echtzeitsystemen mit harten Zeitschranken der Einsatz eines Caches meist nicht möglich ist. Diese beiden On-Chip-Speicher bieten sich zur Energiereduktion an, da ein Zugriff auf den Hauptspeicher viel mehr Energie verbraucht als eben ein Zugriff auf einen dieser Speicherarten.

Der Cache ist aufgrund des Hardware-Controllers weitgehend anwendungsunabhängig einsetzbar. Ein Scratchpad hingegen wird durch Software gesteuert. Daher muss hier an die Stelle des Hardware-Controllers des Caches ein optimierender Compiler treten, der den Scratchpad richtig nutzen kann. Alternativ zum Compiler könnte der Programmierer höchstens sehr aufwendig von Hand seine Programme optimieren.

In Rahmen dieser Diplomarbeit wurde der Weg des optimierenden Compilers eingeschlagen. So wurde ein neues Verfahren zur dynamischen Nutzung des Scratchpads entwickelt und in den vorhandenen enCC-Compiler integriert.

Wie die Ergebnisse gezeigt haben, so kann das dynamische Verfahren, trotz des zum Teil im Verfahren begründeten immensen Overheads, besser abschneiden als die verglichenen Verfahren der statischen Nutzung des Scratchpads und des Caches.

Das dynamische Verfahren kann dort besser abschneiden, wo der Scratchpad so dimensioniert ist, dass er nicht alle häufig ausgeführten Programmteile auf einmal aufnehmen kann und gleichsam die Programmstruktur zu verschiedenen Laufzeiten konkurrierende Programmteile für die Verlagerung in den Scratchpad besitzt. Nur wenn der Scratchpad mehrfach zu einem relativ großen Prozentsatz

gefüllt werden kann, besitzt das dynamische Verfahren Vorteile gegenüber dem statischen Verfahren oder auch gegenüber einem Cache.

Für reale Anwendungen steigt dabei mit der Programmgröße die Wahrscheinlichkeit für ein Auftreten einer solchen Konkurrenz an. Je größer also ein Programm im Verhältnis zur vorhandenen Scratchpadgröße ist, desto günstiger kann das dynamische Verfahren arbeiten. Einzige Ausnahme für diese Regel ist natürlich, wenn der Scratchpad so klein ist, dass einzelne Basicblöcke erst gar nicht in den Scratchpad passen. Ab einer gewissen Mindestgröße des Scratchpads schneidet dynML dann allerdings mit wachsender Programmgröße immer besser ab. Dabei ist das Hauptaugenmerk hier auf die vom Programm verbrauchte Energie gelegt worden. Neben einer Energiereduktion ist auch eine Performancesteigerung zu beobachten, die allerdings nicht so stark ausgeprägt ausfällt.

Ein Problem beim Einsatz des dynamischen Verfahrens bleibt allerdings der immense Programmspeicherplatzbedarf, der je nach Programm bis zu einem Vielfachen der ursprünglichen Programmgröße ausmachen kann.

Weiterhin muss beachtet werden, dass die vom Algorithmus verwendeten Ausführungshäufigkeiten einzelner Programmteile mit Hilfe des Profilings ermittelt wurden und somit datenabhängig sind. Diese Optimierungsmethode findet somit zwar eine optimale Lösung, allerdings nur für eine vorgegebene Eingabedatenmenge. Durch Simulationen mit verschiedenen Eingabedaten-mengen kann dieser Effekt allerdings abgeschwächt werden.

DynML beschränkt sich dabei bei der Analyse und dem Einlagern von Speicherobjekten in den Scratchpad ausschließlich auf Basicblöcke. In weiteren Forschungsarbeiten müsste auch untersucht werden, in wieweit auch Datenobjekte und vielleicht auch der Stack zusätzlich in den Scratchpad verschoben werden können und wie sich dieses Verschieben auf den Energieverbrauch des Programms auswirken würde. Dabei könnten diese Speicherobjekte entweder permanent oder aber auch dynamisch verlagert werden. Bei einer dynamischen Verlagerung der globalen Variablen würde dabei allerdings im Unterschied zum dynamischen Verlagern von Programmteilen der Overhead noch weiter ansteigen, da im Scratchpad veränderte Datensätze auch wieder zurückgeschrieben werden müssten, falls sie zu einem späteren Zeitpunkt noch einmal im Scratchpad benötigt würden. Lokale Variablen, die unter Umständen überschattet werden, und Konstanten müssen natürlich nicht zurückgeschrieben werden.

Insbesondere der Partitionierung des Scratchpads zwischen Programm- und Datenteilen muss dabei besondere Aufmerksamkeit geschenkt werden. Auch für diesen Fall sind verschiedene Modelle denkbar, z.B. mit einer festen Aufteilung zwischen Stack, Programm- und Datenteilen oder aber einer variablen Einteilung, die sich über die Programmlaufzeit verändern kann.

Eine weitere Aufgabe für die Zukunft muss es sein, dieses Verfahren auch auf Library-Funktionen auszudehnen. Im Zuge dieser Diplomarbeit wurde erarbeitet, dass ein Verschieben solcher Funktionen ebenfalls möglich ist. Von einer Implementierung wurde allerdings abgesehen, da die Programme der untersuchten Benchmark-Suite solche Funktionen nur sehr wenig oder gar nicht benutzen. Für Programme, die einen Großteil der Rechenzeit in solchen Library-Funktionen

verbringen wird es in Zukunft allerdings notwendig werden, auch diese Programmteile verschiebbar zu machen.

Als Ergebnis dieser Diplomarbeit bleibt festzuhalten, dass das dynamische Verlagern von Programmteilen in den Scratchpad einen positiven Effekt auf den Energiebedarf von Software hat. Erkauft wird dieser Energiegewinn allerdings durch einen immensen Aufwand und einen wesentlich gesteigerten Speicherplatzbedarf.

Damit kann auch dieses hier entwickelte Verfahren dazu beitragen, das heutige Problem des immer weiter steigenden Energiebedarfs von Programmen, besonders in mobilen Geräten, erheblich abzuschwächen. Gerade wenn man bedenkt, dass die hier vorgenommene Implementierung sich auf Basicblöcke beschränkt. Wie bereits angedeutet stellt dies erst einen ersten Anfang dar und es gibt noch ein erhebliches Verbesserungspotential wenn das Verfahren auch auf Datenteile ausgeweitet wird.

Anhang A: Implementierung – enJumpCorrection

Wie schon mehrfach erwähnt, mussten einige notwendige Funktionen in ein externes Programm ausgelagert werden, um eine korrekte Ausführung des zu compilierenden Programms sicherzustellen. Dabei kann dieses externe Programm natürlich nicht mehr auf die Datenstrukturen des eigentlichen Compilers zugreifen, wie dies das Modul dynML vermag. Änderungen werden hier also in der vom Compiler erzeugten Assembler-Datei vollzogen.

Die Funktionen, die das „enJumpCorrection“ oder kurz *JC* ausführt, sind im Einzelnen:

- Anpassen der Sprünge: Der Assembler berechnet nur die relativen Sprünge für die Sprungabstände im Off-Chip Speicher. Nach einem Verschieben von Blöcken in den Scratchpad durch die Kopierfunktionen stimmen diese relativen Sprünge nicht mehr und müssen somit angepasst werden. Ebenfalls angepasst werden müssen die Funktionsaufrufe, falls die Kopierfunktion an erster Stelle einer Funktion steht. Dann muss der BL Befehl natürlich vom eigentlich Funktionsstart auf die Kopierfunktion umgebogen werden.
- Anpassung der Kopierfunktionen: Die Kopierfunktionen müssen die korrekten Adressen im Speicher kennen, um einen Basicblock korrekt in den Scratchpad verschieben zu können. Da diese Adresse erst nach dem ersten Linken verfügbar ist, muss diese Anpassung hier erfolgen.
- Einfügen von ALIGNs: Zusätzliche ALIGN-Befehle müssen in den ASM-Quellcode eingefügt werden, damit die hier verwendete Kopierfunktion, die DoubleWords aus dem Speicher liest und schreibt, richtig arbeiten kann. Alle Funktionen müssen also 4-Byte aligned sein. Bei einem 16bit Befehlssatz müssen hier also unter Umständen NoOp’s (no operations) eingefügt werden.
- Anpassung verwendeter Konstanten: Die Befehle, die mit direktem Offset auf Konstanten zugreifen, müssen geändert werden, um die ebenfalls in den Scratchpad kopierten Konstanten richtig zu adressieren. Im einzelnen sind dies die LDR und STR Befehle mit direktem 8-bit Offset.

Um die Aufgaben richtig erfüllen zu können, benötigt das JC-Programm verschiedene Eingabedateien, neben der eigentlichen Assembler-Datei. Diese Dateien sind wie folgt:

Vom ARM-Linker erzeugt:

- Symbol.txt: Diese Datei enthält die Adressen der einzelnen Basicblocks im Off-Chip Speicher, an die der Loader diese Blöcke laden wird.

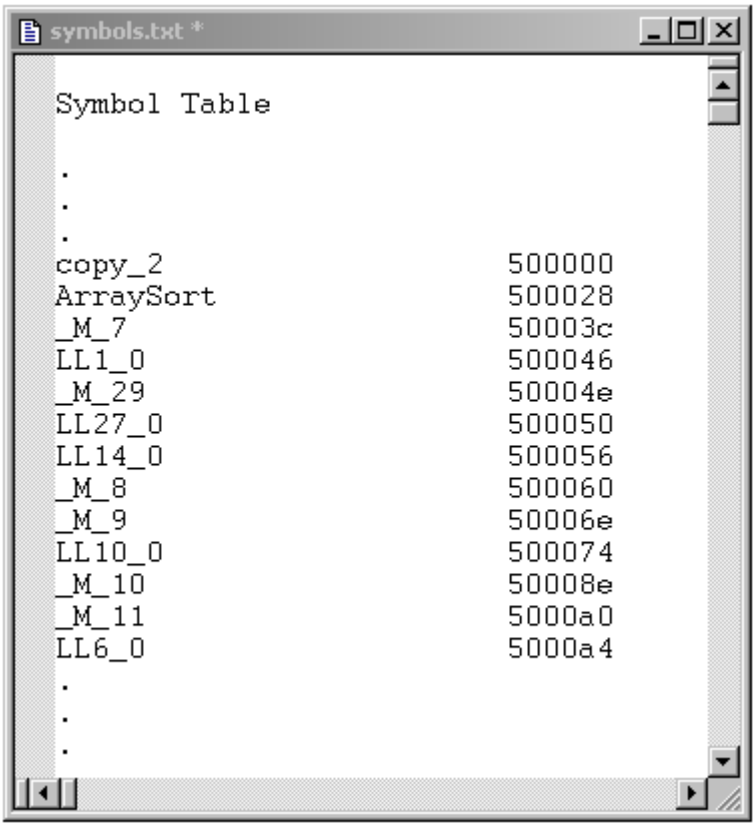

Abbildung A-1 Beispiel einer symbol.txt – Datei

Vom dynML-Modul erzeugt:

- Adjust1.txt: Die Datei enthält den Konstantennamen der Kopierfunktion und den damit verknüpften Basicblocknamen. Die entsprechende Konstante muss dann in der Assemblerdatei auf die Speicheradresse des Basicblocks gepatcht werden.

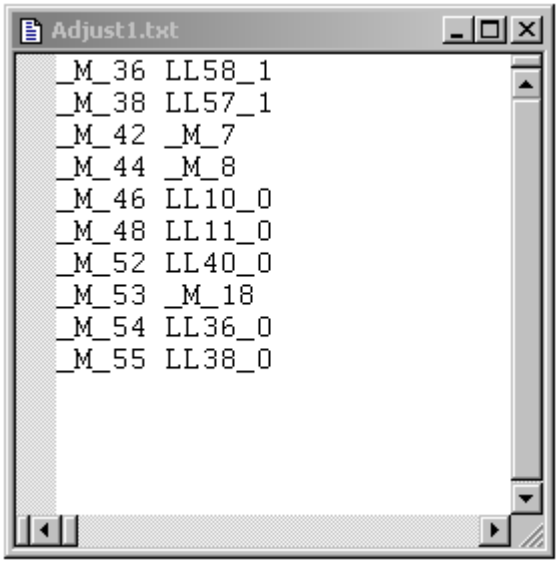

Abbildung A-2 Beispiel einer adjust1.txt – Datei

- Adjust2.txt: Hier wird der Name der Blöcke gespeichert, die im Scratchpad ausgeführt werden sollen und die dazugehörige Adresse, an die der Block im Scratchpad kopiert wird. Diese Angaben sind nötig, um die relativen Sprünge in und aus dem Scratchpad berechnen zu können.

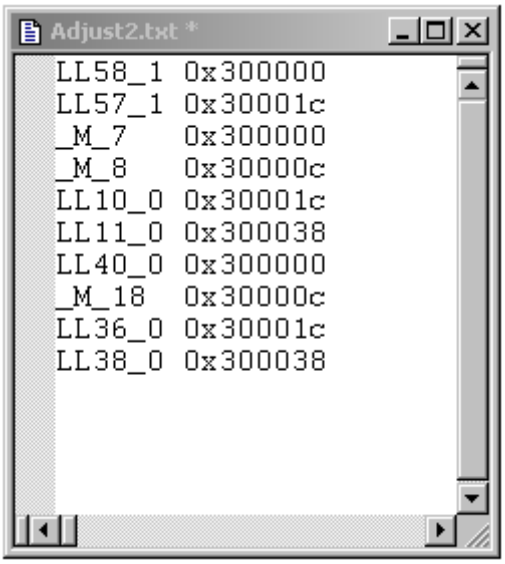

Abbildung A-3 Beispiel einer adjust2.txt – Datei

- Adjust3.txt: Diese Datei wird benutzt, falls die Kopierfunktion eines Superblocks an die ersten Stelle einer Funktion rutscht. Hier wird also der Funktionsname und der Name der Kopierfunktion gespeichert. Die Funktionsaufrufe für diese Funktion müssen also auf die Kopierfunktion umgebogen werden.

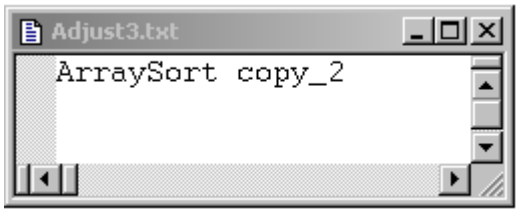

Abbildung A-4 Beispiel einer adjust3.txt – Datei

- Adjust4.txt: Hier schließlich werden die Konstantennamen und ihre Adressen im Scratchpad gespeichert, falls sie in den Scratchpad mitverschoben werden müssen. Diese Angaben sind nötig, um die LDR- und STR-Befehle richtig patchen zu können.

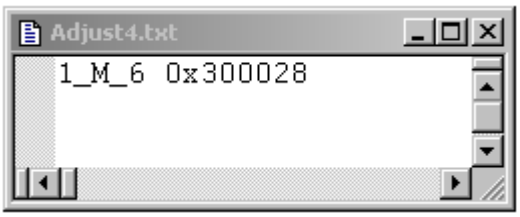

Abbildung A-5 Beispiel einer adjust4.txt – Datei

Die Vorgehensweise bei den einzelnen Funktionen im Detail:

- **Sprunganpassung:**
  Bei der Sprunganpassung muss zunächst zwischen zwei verschiedenen Arten von Sprüngen unterschieden werden, für die Anpassungen am Assemblercode notwendig werden.

  Zunächst sind das die Sprünge, die zwischen den Basicblöcken im Off-Chip Speicher und denen im Scratchpad-Speicher verbinden. Hierbei handelt es sich in jedem Fall um lange Sprünge, sogenannte BL – Branch Link Befehle.

  Diese Sprünge können erst jetzt berechnet werden, da die entgültigen Adressen im Off-Chip erst nach dem Linken feststehen und es sich beim BL-Befehl nur um einen relativen Sprung handelt.

  Bei dieser Art von Sprung wird also so vorgegangen: Im Compilerdurchlauf wird bereits ein Kommentar in die Assemblerdatei eingefügt, der den zu ändernden Sprungbefehl markiert und zusätzlich benötigte Informationen erhält. Die Informationen im Einzelnen sind, von welchem Basicblock zu welchen Basicblock gesprungen wird und welchen Offset der BL-Befehl relativ zum Beginn des Basicblocks besitzt. Das letzte Wort „LONG“ bedeutet dabei, dass es sich um einen langen Sprung handelt. Mit Hilfe der Informationen aus den Dateien Adjust2.txt und Symbols.txt lassen sich nun die endgültigen Adressen im Speicher ermitteln. Nun erfolgt die Berechnung des Sprungbefehls und dieser wird als Bitmuster als zwei DCW Instruktionen abgelegt. Dies ist notwendig, da sonst der Assembler erneut versuchen würde den Sprung relativ neu zu berechnen (aus einem BL-Befehl) und zu einem falschen Ergebnis kommen würde, da der Assembler ja nicht weiß, an welche Stelle im Scratchpad der Block verschoben werden soll. An dieser Stelle werden 2 DCW Befehle einem DCD Befehl bevorzugt, da sonst der Sprungbefehl 4-byte aligned sein müsste und somit weiteren Speicherplatz verschwendet werden würde. Die beiden 16 bit DCW Befehle werden nun als BL Befehl codiert. Die ersten 5 bit beider Instruktionen sind dabei für die Codierung des Befehls BL

zuständig. Somit stehen in jedem Word noch 11 bit für die Codierung der Sprunglänge zur Verfügung.

Da es sich um einem 16bit Befehlssatz handelt wird das letzte Bit nicht berücksichtigt, da nur Word-weite Sprünge zulässig sind. Somit ergibt sich ein ausreichender Platz für 23-bit (11+11+1) lange Sprünge, wobei die 23 bit eine Zahl im 2er Komplement darstellen um Sprünge vorwärts und rückwärts im Speicher zu ermöglichen.

Long Branch with link:

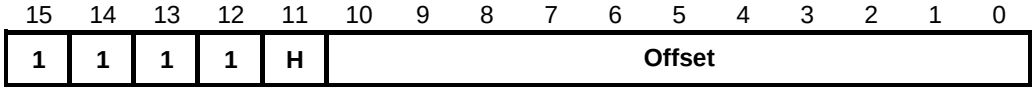

Abbildung A-6 Long Branch Link – Bitmuster

H	THUMB Assembler	Action
0	BL label	LR := PC + OffsetHigh<<12
1		temp := next instruction address PC := LR + OffsetLow <<1 LR := temp | 1

Tabelle A-1 Long Branch Link – Beschreibung

Um die richtige Sprunglänge zu errechnen, muss man also so vorgehen:

Gleichung A-1 Berechnung der Sprunglänge

(Adresse Zielblock – (Adresse Startblock + Offset des Branch Link Befehls + 4))

Die Anpassung mit +4 ist notwendig, da das PC Register durch den Operand Prefetch bereits 1 Word weiter steht. Des weiteren werden nur einfache Shiftoperationen und logische Verknüpfungen benötigt, um das richtige Format zu erhalten.

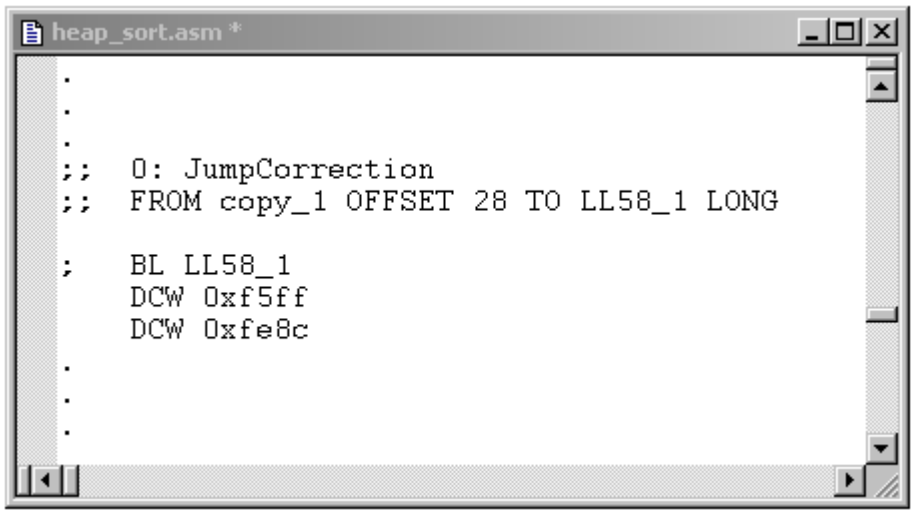

Abbildung A-7 Beispiel eines typischen langen Sprungs

Neben den langen Sprüngen kann es allerdings noch vorkommen, dass auch kurze Sprünge angepasst werden müssen. Dieser Fall tritt allerdings nur auf, falls es einen konditionalen oder kurzen Sprung zwischen zwei Blöcken im Scratchpad gibt, die im Scratchpad eine andere Entfernung zu einander haben als im Off-Chip Speicher. Dies kann dann auftreten, falls in einer Reihe von drei oder mehr Basicblöcken mehrere nicht aufeinanderfolgende (z.B. Block1 und Block3) Blöcke verschoben werden. Dann ändert sich natürlich der Abstand im Scratchpad und da wir nur relative Sprünge im Befehlssatz kennen, natürlich auch der benötigte Sprungbefehl.
Die kurzen Sprungbefehle wir der B-Befehl und alle konditionalen Sprungbefehle müssen somit auch wie die langen BL-Befehle durch DCW-Befehle ausgetauscht werden. Die Berechnung erfolgt dabei sehr ähnlich den langen Sprungbefehlen, nur das hier natürlich die Codierung der Sprungbefehle für jede Art von kurzem Sprung unterschiedlich ist.

Conditional Branch:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	**1**	**0**	**1**	**Cond**				**Soffset 8**							

Abbildung A-8 Conditional Branch - Bitmuster

Cond	THUMB Assembler	Action
0000	BEQ label	Branch if Z set (equal)
0001	BNE label	Branch if Z clear (not equal)
0010	BCS label	Branch if C set (unsigned higher or same)
0011	BCC label	Branch if C clear (unsigned lower)
0100	BMI label	Branch if N set (negative)
0101	BPL label	Branch if N clear (positive or zero)
0110	BVS label	Branch if V set (overflow)
0111	BVC label	Branch if V clear (no overflow)
1000	BHI label	Branch if C set and Z clear (unsigned higher)
1001	BLS label	Branch if C clear or Z set (unsigned lower or same)
1010	BGE label	Branch if N set and V set, or N clear and V clear (greater or equal)
1011	BLT label	Branch if N set and V clear, or N clear and V set (less than)
1100	BGT label	Branch if Z clear, and either N set and V set or N clear and V clear (greater than)
1101	BLE label	Branch if Z set, or N set and V clear, or N clear and V set (less than or equal)

Tabelle A-2 Conditional Branch - Beschreibung

Unconditional Branch:

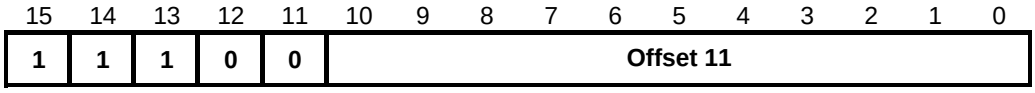

Abbildung A-9 Unconditional Branch – Bitmuster

THUMB Assembler	Action
B label	Branch PS relative +/- Offset11<<1, where label is PC +/- 2048 bytes.

Tabelle A-3 Unconditional Branch – Beschreibung

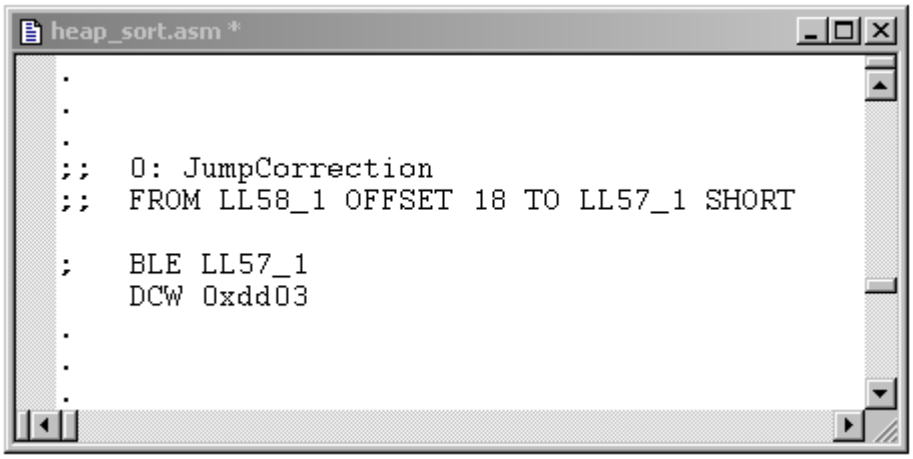

Abbildung A-10 Beispiel eines typischen kurzen Sprungs

- **Kopierfunktionanpassung:**
  Für die Kopierfunktion ist im wesentlichen nur ein kleine Anpassung notwenig.

  Wenn man sich das Beispiel für die Kopierfunktion ansieht, so erkennt man, dass zunächst alle unteren Register auf dem Stack gesichert werden und am Ende wieder zurückkopiert werden. Somit stehen der Kopierfunktion alles unteren Register zur Verfügung. In die Register R6 und R7 werden dabei die Adressen des zu kopierenden Blocks im Scratchpad und im normalen Speicher geladen. Da die Adresse im normalen Speicher aber erst nach dem Linken feststeht, wird diese Konstante erst hier auf den richtigen Wert gesetzt. Wie im unteren Beispiel zu sehen ist, handelt es sich dabei um eine einfache Änderung der Zeile, in der die Konstante definiert ist. Da allerdings die Konstanten an verschiedenen Stellen benutzt werden können und unter verschiedenen Namen auftauchen können (_M**_**), muss sichergestellt werden, dass alle entsprechenden Konstanten angepasst werden.

  Diese Mehrfachnamen für die Konstanten entsteht aus der Tatsache, dass die Konstanten auch relativ zum PC geladen werden und zwar mit einem relativ kleinen Offset. Falls die Konstante also an zwei weit entfernten Orten benutzt wird, so muss sie unter einem neuen Namen neu definiert werden.

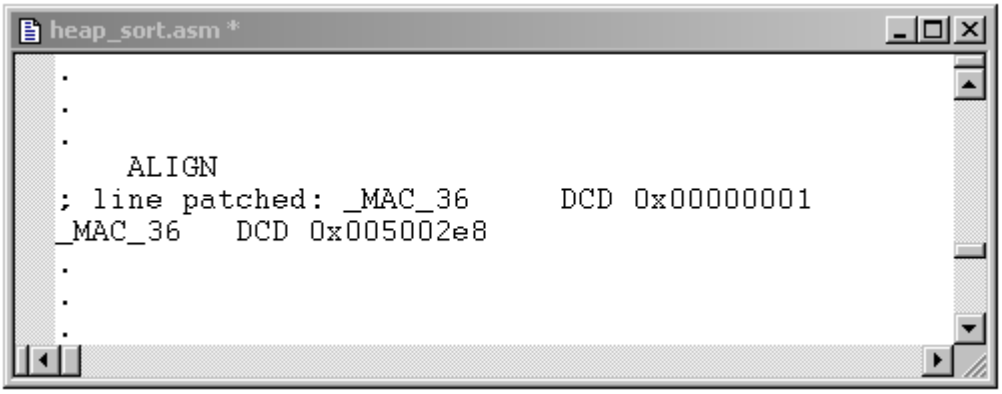

Abbildung A-11 Beispiel für das Anpassen der Konstanten

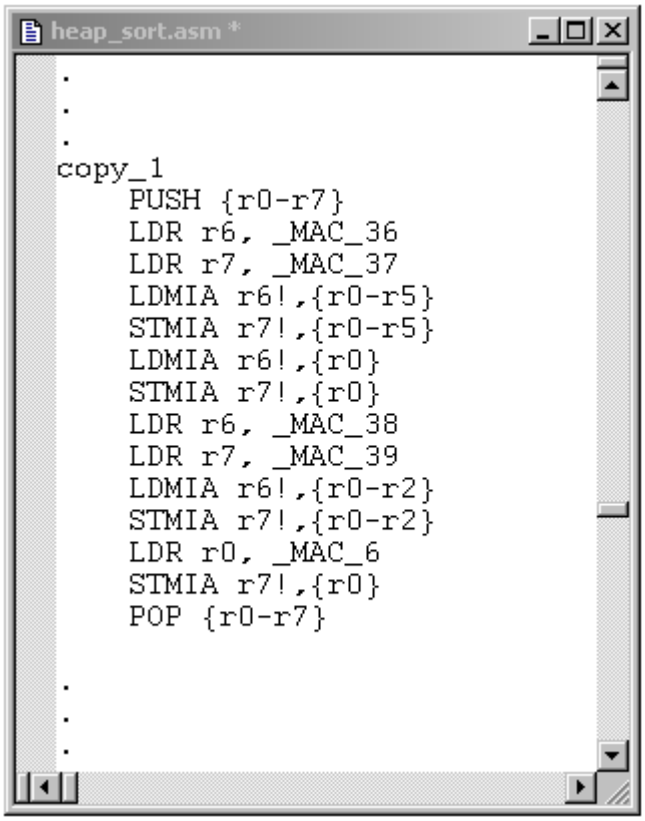

Abbildung A-12 Beispiel einer typischen Kopierfunktion

Ansonsten sieht man an diesem Beispiel sehr gut, wie zwei verschiedene Blöcke nacheinander in den Scratchpad kopiert werden und anschließend noch eine einzelne Konstante (_MAC_6) zusätzlich verschoben wird.

- **Einfügen von ALIGNs:**

  Des weiteren müssen an manchen Stellen im Assemblercode noch ALIGN-Befehle eingefügt werden. Diese ALIGNs sind notwendig vor allen Basicblöcken, die in den Scratchpad kopiert werden müssen. Dieser Befehl bewirkt, das der zu kopierende Block 4byte-aligned im Speicher liegt, so dass die Kopierfunktion mit den LDRMIA und STRMIA – Befehlen, die nur Doubleword-aligned arbeiten können, möglichst effektiv arbeiten kann. Durch den ALIGN-Befehl können NoOp's eingefügt werden, die im Normalfall allerdings durch die Sprünge in und aus dem Scratchpad nicht ausgeführt werden und somit auch keine weitere Energie verbrauchen.

- **Konstantenanpassung:**

  Bei der Konstantenanpassung müssen alle die in den Scratchpad zu kopierenden Basicblöcke auf Konstanten hin untersucht werden, auf die sie zugreifen. Diese Konstanten müssen dann durch die Kopierfunktion mit in den Scratchpad kopiert werden. Die entsprechenden LDR und STR-Befehle müssen im Anschluss natürlich, ähnlich den Sprungbefehlen, auch entsprechend angepasst werden. Auch hier werden die Befehle wieder

durch DCW-Befehle ersetzt, die die entsprechenden Bitmuster enthalten. Hier muss zusätzlich richtig kodiert werden, um welchen Befehl es sich handelt und in welches Register die Konstante geladen werden soll.

PC-relative load:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**1**	**0**	**0**	**1**	**Rd**			**Word 8**							

Abbildung A-13 PC-relative Load – Bitmuster

THUMB Assembler	Action
LDR Rd, [PC,#Imm]	Add unsigned offset (255 words, 1020 bytes) in Imm to the current value of the PC. Load the owrd from the resulting address into Rd.

Tabelle A-4 PC-relative Load - Beschreibung

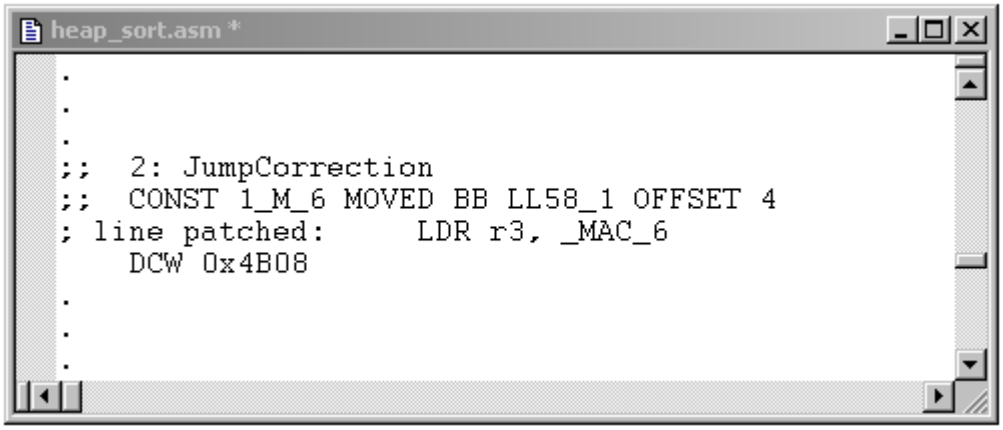

Abbildung A-14 Beispiel für das Anpassen von LDR und STR-Befehlen

Anhang B: Ergebnistabellen

Energiewerte (Werte in 10^{-6} Ws):

	Scratchpadsize									
selection_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	2964,330	(**)	(**)	2079,251	1460,941	1130,755	896,245	907,347	927,633	980,204
dynML	2964,330	2470,029	2036,530	1013,079	987,924	993,746	999,569	1009,030	1027,226	1037,415
statML (*)	2964,330	2746,751	2449,713	996,426	945,316	958,330	964,143	973,589	991,756	1001,929
statML	2964,330	2744,451	2446,513	991,326	936,516	942,330	948,143	957,589	975,756	985,929

bubble_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	4507,921	(**)	(**)	4130,489	2242,597	1852,485	1558,636	1576,970	1607,461	1684,615
dynML	4507,921	4427,772	3426,854	2085,991	1753,610	1761,665	1769,721	1782,811	1807,984	1822,082
statML (*)	4507,921	4258,524	3644,701	2393,771	1724,933	1724,883	1732,923	1745,987	1771,110	1785,179
statML	4507,921	4256,224	3641,501	2388,671	1716,133	1708,883	1716,923	1729,987	1755,110	1769,179

biquad_N_sections	0	16	32	64	128	256	512	1024	2048	4096
Cache	26,582	(**)	(**)	24,771	22,386	20,114	18,315	17,697	17,303	17,833
dynML	26,582	24,153	24,153	24,232	23,104	23,147	23,190	23,259	23,392	23,467
statML (*)	26,582	22,245	20,380	21,788	21,268	20,479	20,521	20,591	20,724	20,799
statML	26,582	19,945	17,180	16,688	12,468	11,679	11,721	11,791	11,924	11,999

insertion_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	2019,925	(**)	(**)	1501,108	1095,869	874,137	746,970	752,275	764,436	799,847
dynML	2019,925	1781,732	1395,641	784,688	716,154	719,981	723,808	730,027	741,986	748,683
statML (*)	2019,925	1907,099	1581,944	1054,253	750,219	700,016	703,841	710,057	722,012	728,706
statML	2019,925	1904,799	1578,744	1049,153	741,419	684,016	687,841	694,057	706,012	712,706

matrix_mult	0	16	32	64	128	256	512	1024	2048	4096
Cache	79,089	(**)	(**)	93,647	59,192	42,432	38,867	37,381	37,191	38,437
dynML	79,089	80,632	78,324	44,161	42,565	42,692	42,819	43,025	43,422	43,644
statML (*)	79,089	79,177	77,927	43,100	42,521	40,986	40,506	40,723	41,139	41,373
statML	79,089	76,877	74,727	38,000	33,721	32,186	31,706	31,923	32,339	32,573

lattice	0	16	32	64	128	256	512	1024	2048	4096
Cache	2863,208	(**)	(**)	3128,949	2187,619	1661,906	1454,186	1341,197	1308,575	1350,656
dynML	2863,208	2869,157	2790,821	2215,286	1000,625	965,724	971,235	980,191	997,414	1007,058
statML (*)	2863,208	2865,114	2829,154	2322,410	1585,470	967,438	970,942	979,863	997,017	1006,624
statML	2863,208	2862,814	2825,954	2317,310	1576,670	951,438	954,942	963,863	981,017	990,624

quick_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	206,451	(**)	(**)	307,308	259,439	161,335	105,211	92,825	88,745	91,012
dynML	206,451	201,774	195,081	183,376	168,788	136,889	133,960	134,445	135,378	135,900
statML (*)	206,451	205,597	201,490	194,327	181,907	123,733	114,518	115,022	115,991	116,533
statML	206,451	203,297	198,290	189,227	173,107	107,733	98,518	99,022	99,991	100,533

heap_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	1303,692	(**)	(**)	1676,975	1223,219	673,479	458,206	437,257	440,836	462,038
dynML	1303,692	1257,596	1125,720	1055,988	792,037	658,269	530,108	534,048	541,625	545,868
statML (*)	1303,692	1287,154	1161,255	1091,371	923,193	806,854	505,957	509,753	517,053	521,142
statML	1303,692	1284,854	1158,055	1086,271	914,393	790,854	489,957	493,753	501,053	505,142

multi_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	8775,446	(**)	(**)	7909,586	4982,795	3668,829	2905,811	2923,667	2977,805	3128,174
dynML	8775,446	8154,942	6588,593	4154,604	3675,295	3415,969	3298,943	3325,434	3376,380	3404,910
statML (*)	8775,446	8526,050	7912,226	6661,296	5505,305	3752,835	3462,298	3197,001	3247,590	3275,920
statML	8775,446	8523,750	7909,026	6656,196	5496,505	3736,835	3446,298	3181,001	3231,590	3259,920

(*) Die Werte des statischen Verfahrens wurden um die Kosten für das einmalige Kopieren in den Scratchpad korrigiert.

(**) Cache-Werte für die Cachegrößen von 32 und 64 Byte wurden nicht ermittelt, da diese Größen für Caches ungewöhnlich klein wären und keinen fairen Vergleich erlauben würden.

Performancewerte (Werte in Prozessorzyklen):

	Scratchpadsize									
selection_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	212420	(*)	(*)	210708	170997	150240	136222	135694	135556	135561
dynML	212420	233356	242270	151743	150100	150100	150100	150100	150100	150100
statML	212420	242120	228260	140723	139553	139553	139553	139553	139553	139553

bubble_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	306204	(*)	(*)	367049	248493	223384	205344	204678	204502	204486
dynML	306204	365804	330051	247048	216141	216141	216141	216141	216141	216141
statML	306204	316088	320919	270187	206807	205474	205474	205474	205474	205474

biquad_N_sections	0	16	32	64	128	256	512	1024	2048	4096
Cache	1755	(*)	(*)	2085	1926	1782	1669	1621	1587	1592
dynML	1755	1885	1885	1885	1947	1947	1947	1947	1947	1947
statML	1755	1529	1431	1400	1252	1214	1214	1214	1214	1214

insertion_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	137054	(*)	(*)	145354	119338	104952	96813	96306	96111	96089
dynML	137054	147270	147771	102217	95033	95033	95033	95033	95033	95033
statML	137054	149475	142077	123663	91928	88800	88800	88800	88800	88800

matrix_mult	0	16	32	64	128	256	512	1024	2048	4096
Cache	5035	(*)	(*)	7507	5337	4312	4087	3973	3927	3918
dynML	5035	5445	5412	4180	4057	4057	4057	4057	4057	4057
statML	5035	5022	5067	3635	3453	3416	3368	3368	3368	3368

lattice	0	16	32	64	128	256	512	1024	2048	4096
Cache	186087	(*)	(*)	264282	203516	169602	156298	148399	145191	144639
dynML	186087	187050	184431	164432	120594	118885	118885	118885	118885	118885
statML	186087	186141	184775	166673	140985	117742	117464	117464	117464	117464

quick_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	14202	(*)	(*)	23610	20517	14385	10932	10112	9787	9727
dynML	14202	14674	15246	14707	14958	12910	12542	12542	12542	12542
statML	14202	14628	13970	14253	14236	10632	10285	10285	10285	10285

heap_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	87785	(*)	(*)	135134	106287	72315	59228	57499	57145	57070
dynML	87785	90839	82284	86227	72794	68897	62939	62939	62939	62939
statML	87785	90365	83370	83598	79481	75001	58340	58340	58340	58340

multi_sort	0	16	32	64	128	256	512	1024	2048	4096
Cache	606375	(*)	(*)	861178	663168	552657	509305	483566	473113	47314
dynML	606375	689969	654565	484988	456835	435372	429150	429150	429150	429150
statML	606375	616259	621090	570358	526489	429290	419421	403417	403417	403417

(*) Cache-Werte für die Cachegrößen von 32 und 64 Byte wurden nicht ermittelt, da diese Größen für Caches ungewöhnlich klein wären und keinen fairen Vergleich erlauben würden.

Programmgröße (Werte in Byte):

	Scratchpadsize									
selection_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	128	244	336	452	468	468	468	468	468	468
statML	128	140	152	148	136	136	136	136	136	136

bubble_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	144	268	332	424	580	580	580	580	580	580
statML	144	152	168	184	164	152	152	152	152	152

biquad_N_sections	0	16	32	64	128	256	512	1024	2048	4096
dynML	184	244	244	244	296	296	296	296	296	296
statML	184	188	196	184	192	200	188	188	188	188

insertion_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	176	264	416	512	704	704	704	704	704	704
statML	176	192	200	216	192	184	184	184	184	184

matrix_mult	0	16	32	64	128	256	512	1024	2048	4096
dynML	264	368	400	396	420	420	420	420	420	420
statML	264	280	288	280	320	272	272	272	272	272

Lattice	0	16	32	64	128	256	512	1024	2048	4096
dynML	304	372	452	544	524	572	572	572	572	572
statML	304	316	324	332	324	356	312	312	312	312

quick_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	332	492	572	712	892	1352	1444	1444	1444	1444
statML	332	344	340	360	400	332	340	340	340	340

heap_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	456	576	572	652	896	1196	1488	1488	1488	1488
statML	456	468	468	488	516	568	464	464	464	464

multi_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	712	1072	1212	1488	1823	2208	2496	2496	2496	2496
statML	712	720	736	752	736	744	824	720	720	720

Instruktionsanzahl:

	Scratchpadsize									
selection_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	72566	82680	87448	72816	72723	72723	72723	72723	72723	72723
statML	72566	82466	82664	82466	72566	72566	72566	72566	72566	72566

bubble_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	100393	115322	115136	105485	100643	100643	100643	100643	100643	100643
statML	100393	105335	110203	107912	100591	100393	100393	100393	100393	100393

biquad_N_sections	0	16	32	64	128	256	512	1024	2048	4096
dynML	536	565	565	565	579	579	579	579	579	579
statML	536	537	537	534	535	539	533	533	533	533

insertion_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	47617	52577	52682	48049	47714	47714	47714	47714	47714	47714
statML	47617	50174	50174	50366	47815	47617	47617	47617	47617	47617

matrix_mult	0	16	32	64	128	256	512	1024	2048	4096
dynML	1667	1733	1745	1755	1735	1735	1735	1735	1735	1735
statML	1667	1675	1707	1675	1668	1699	1699	1699	1699	1699

lattice	0	16	32	64	128	256	512	1024	2048	4096
dynML	68617	68765	68787	68923	68924	68933	68933	68933	68933	68933
statML	68617	68635	68619	68637	68855	68619	68617	68617	68617	68617

quick_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	3873	4010	4213	4207	4416	4325	4299	4299	4299	4299
statML	3873	4015	3889	4034	4155	3873	3873	3873	3873	3873

heap_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	29102	30171	29320	31038	29977	30477	30355	30355	30355	30355
statML	29102	29962	29298	29947	30150	30180	29102	29102	29102	29102

multi_sort	0	16	32	64	128	256	512	1024	2048	4096
dynML	202055	228168	231894	209334	206542	203904	203716	203716	203716	203716
statML	202055	206997	211865	209574	211566	202898	203124	202055	202055	202055

Anhang C: Index - Gleichungen, Tabellen und Abbildungen

Literaturverzeichnis

- [ASU88] A.V. Aho, R. Sethi, J.D. Ullman: *Compilerbau*, Addison Wesley Verlag, 1988.
- [BE00] L. Benini, A. Macii, E. Macii, M. Poncino: *Synthesis of Application-Specific Memories for Power Optimization in Embedded Systems*, Universität Bologna, Italien, DAC, Los Angeles, CA, 2000.
- [BH00] N. Bellas, I. Hajj, C. Polychronopoulos: *Using dynamic cache management techniques to reduce energy in a high-performance processor*, Department of Electrical & Computer Engineering and the Coordinated Science Laboratory, Technical Report, University of Illinois, 2000.
- [BM01] P. Bose, M. Martonosi, D. Brooks: *Modeling and Analyzing CPU Power and Performance: Metrics, Methods and Abstractions*, Sigmetrics, Cambridge, Massachusetts, USA, 2001.
- [BO02] Online-Quelle: Borland Homepage: *Turbo Pascal 5 Fact Sheet*, URL: http://community.borland.com/article/0,1410,16329,00.html, online: 06.02.2002.
- [ES01] Online-Quelle: Eskicioglu & Marsland: *Memory Management*, *2001*, URL: http://www.cs.ualberta.ca/~tony/C379/Notes/PDF/07.4.pdf, Vorlesungsunterlagen, Universität Manitoba, Kanada, online: 08.02.2002.
- [GBJ00] D. Grune, H. Bal, C. Jacobs, K. Langendoen: *Modern Compiler Design*, John Wiley and Sons Ltd., 2000
- [HE02] Online-Quelle: Heise Newsticker: *Studie: UMTS verzögert sich*, URL: http://www.heise.de/newsticker/data/anw-15.01.02-003/, online: 15.01.2002.
- [IS00] T. Ishihara, H. Yasuura: *A power reduction technique with object code merging for application specific embedded processors*, DATE, March 2000.
- [KA00] M. Kandemir, N. Vijakrishnan, M.J. Irwin, W. Ye: *Influence of Compiler Optimizations on System Power*, DAC, Los Angeles, CA, 2000.
- [KA01] M. Kandemir, J. Ramanujam, M.J. Irwin, N. Vijaykrishnan, I. Kadayif, A. Parikh: *Dynamic Management of Scratch-Pad Memory Space*, DAC, Las Vegas, Nevada, USA, 2001.
- [LE01] B.-S. Lee: *Vergleich des Energieverbrauchs von Cache- und Scratch-Pad-Speichern für den ARM7-Prozessor*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Diplomarbeit 2001.
- [MS92] Online-Quelle: Microsoft Developer Network: *The Microsoft Overlay Virtual Environment (MOVE)*, URL: http://msdn.microsoft.com/archive/default.asp?url=/archive/en-

us/dnarvc/html/msdn_draft3.asp, Created March 20,1992, online: 10.01.2002.

- [PA99] P.R. Panda, N. Dutt, A. Nicolau: *Memory issues in embedded systems on-chip – Optimizations and exploration*, Kluwer Academic Publishers, 1999.
- [PC90] Tischer: *PC Intern 2.0*, 6. unveränderte Auflage, DATA Becker GmbH., 1990.
- [RJ99] G. Reinman, N. Jouppi: *An Integrated Cache Timing and Power Modell*, Report – COMPAQ Western Research Laboratory, Palo Alto, 1999.
- [RL94] U. Rembold, P. Levi: *Realzeitsysteme zur Prozessautomatisierung*, Seiten 547-635, Realzeitbetriebssysteme, Carl Hanser Verlag München, 1994.
- [SC00] R. Schwarz: *Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Diplomarbeit, 2000.
- [SIT99] G. Sinevriotis und T. Stouraitis: *Power Analysis of the ARM7 Embedded Microprocessor*, Proc. 9th Int. Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS), October 1999.
- [SKW01] S. Steinke, M. Knauer, L. Wehmeyer, P. Marwedel: *An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations,* Power and Timing Modeling, Optimization and Simulation (PATMOS), Yverdon (Switzerland), September 2001.
- [SS00] S. Steinke, R. Schwarz, L. Wehmeyer, P. Marwedel: *Low Power Code generation for a RISC Processor by Register Pipelining*, Universität Dortmund, Fakultät Informatik Lehrstuhl XII, Technical Report #754, August 2000.
- [SWB02] S. Steinke, L. Wehmeyer, B.-S. Lee, P. Marwedel: *Assigning Program and Data Objects to Scratchpad for Energy Reduction*, DATE, Paris/France, März 2002.
- [SY96] Synopsys, Inc.: *Power Product Reference Manual*, Version 3.5, 1996.
- [SZW01] S. Steinke, Ch. Zobiegala, L. Wehmeyer, P. Marwedel: *Moving Program Objects to Scratch-Pad Memory for Energy Reduction*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Technical Report #756, 2001.
- [TH00] M. Theokaridis: *Energiemessung von ARM7TDMI Prozessor-Instruktionen*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Diplomarbeit, 2000.
- [TIW96] V. Tiwari: *Logic and System Design for Low Power Consumption*, University of Princeton, Dissertation, Seite 163-203, November 1996.

- [TMW94] V. Tiwari, S. Malik, A. Wolfe: *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization*, IEEE Transaktion on VLSI System,1994.
- [TO96] H. Tomiyama, H. Yasuura: *Optimal Code Placement of Embedded Software for Instruction Caches*, Department of Information Systems, Kyushu University, Japan, ED&TC, 1996.
- [WI95] N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Oldenbourg, Mchn, 1995.
- [WJ94] S.J.L. Wilton, N.P. Jouppi: *An Enhanced Access and Cycle Time Model for On-Chip Caches*, WRL Research Report 93/5, 1994.
- [ZO01] Ch. Zobiegala: *Energieeinsparung durch compilergesteuerte Nutzung eines ON-Chip Speichers*, Universität Dortmund, Fakultät Informatik, Lehrstuhl XII, Diplomarbeit, 2001.