**UNIVERSITÄT DORTMUND**
FACHBEREICH INFORMATIK

Diplomarbeit

# Energieeffiziente Belegung von Scratchpad-Speichern mit Code und Arrays durch Loop-Tiling

**Thorsten Wilmer**
**10. November 2005**

INTERNE BERICHTE
INTERNAL REPORTS

Diplomarbeit am
Fachbereich Informatik
der Universität Dortmund

Betreuer:
Dr. Lars Wehmeyer,
Prof. Dr. Peter Marwedel

# Vorwort

Die Idee zu dieser Arbeit ist durch Diskussion aktueller Arbeiten und Probleme mit Dr. Lars Wehmeyer und Manish Verma entstanden. Für die hervorragende Betreuung durch Dr. Lars Wehmeyer möchte ich mich ganz besonders bedanken.

Die Diskussionen mit Prof. Dr. Peter Marwedel, Dr. Heiko Falk, Manish Verma und Dr. Lars Wehmeyer zu Beginn meiner Arbeit waren sehr hilfreich. Die Idee zum Highlevel-Ansatz stammt von Manish Verma.

Danken möchte ich auch den Mitarbeitern des Lehrstuhls Informatik XI: Dr. Thomas Bartz-Beielstein, Christian Lasarczyk, Mike Preuss, Boris Naujoks und Karlheinz Schmitt für die Bereitschaft, die Nutzung des genetischen Algorithmus in dieser Arbeit zu diskutieren. Das Gespräch hat die Analyse des Verhaltens angeregt.

Für konstruktive Kritik an der Arbeit und das Lesen der Arbeit danke ich ganz besonders Dr. Lars Wehmeyer, meiner Mutter Karin Wilmer und meiner Kommilitonin Andrea Schweer. Der Kommilitone Lei Yao war stets sehr hilfsbereit.

Den Administratoren des Lehrstuhls Informatik XII und der IRB danke ich für die hervorragende Infrastruktur und schnelle Unterstützung bei Problemen. Ein großer Dank gebührt auch den unzähligen Autoren freier Software, die ich während dieser Arbeit genutzt habe. Danken möchte ich auch den Autoren von Purify und ICD-C.

Allen, die einen Teil zum Erfolg dieser Arbeit beigetragen haben, herzlichen Dank!

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einführung

Die ersten Computer wurden erfunden, um mathematische Probleme zu lösen und um die Arbeit der Mathematiker zu vereinfachen. Diese Computer waren teuer, groß und wurden Tag und Nacht verwendet. Mit immer leistungsfähigeren Rechnern konnten immer kompliziertere Programme ausgeführt werden. Um die Rechner zu programmieren, haben Informatiker Programmiersprachen entwickelt, die einfach zu erlernen sind. Moderne Compiler können Programme automatisch optimieren, so dass sie auch auf parallelen Rechnern effizient ausgeführt werden können [Wol95]. Objektorientierte Sprachen wie C++, Java oder C# erlauben es, dass große Software-Produkte aus kleinen zusammengesetzt werden können und einzelne Komponenten für einen einzelnen Programmierer überschaubar bleiben. Durch die Nutzung von Bibliotheken und Betriebsystemen muss sich ein einzelner Programmierer nicht mehr mit Details alltäglicher Probleme abgeben; etwa mit der Frage, wie die Einstellungen gespeichert oder eine Graphik angezeigt werden kann. Die Größe der Software eines Rechners für einen einzelnen Nutzer nimmt ständig zu. Nur weil die Hardware ebenfalls weiterentwickelt wird, kann sich dieser Trend fortsetzen.

Die Miniaturisierung hat dazu geführt, dass Mikroprozessoren sehr günstig hergestellt werden können. Dadurch wurden die Super-Computer bei ähnlicher Größe leistungsfähiger. Universell einsetzbare Computer wurden zu persönlichen Computern, die sich viele Menschen leisten können, weil die Produktion der Komponenten günstiger geworden ist. Durch die Miniaturisierung und gesunkene Produktionskosten sind programmierbare Rechner in fast allen technischen Produkten zu finden.

Eine Kombination aus spezieller Software und Standard-Hardware ist einfacher zu entwerfen als eine spezielle Schaltung. Daher sind die meisten Prozessoren heute nicht in universell verwendbaren Computern verbaut, sondern in eingebetteten Systemen.

Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt eingebettet sind [Mar03]. Beispiele für Produkte sind Autos, Schiffe und Flugzeuge, Videorecorder, Fernseher, Telefone, Kameras, Kaffeemaschinen, Waschmaschinen, Kühlschränke, Nähmaschinen, Monitore, Tastaturen, Festplatten, Bank-Karten, Herzschrittmacher, Steuerungen für Häuser oder industrielle Anlagen und elektrisch betriebene Spielzeuge, wie Spielekonsolen, Spielzeugroboter oder Puppen. Eingebettete Systeme erfahren momentan große Absatzsteigerungen. Im 3. Quartal 2005 wurden 31,6 Millionen Mobiltelefone in den USA verkauft, dies sind 30% mehr gegenüber dem Vorjahr [Gon05]. Optimierungen des Energieverbrauchs lohnen sich besonders für mobile eingebettete Systeme wie Mobiltelefone oder Digitalkameras.

An eingebettete Systeme werden zwei wesentliche Anforderungen gestellt: Verlässlichkeit und Effizienz. Ein verlässliches System verrichtet die Arbeit zuverlässig und Fehler treten nur selten auf. Um ein System verlässlich zu konstruieren, ist es sinnvoll, sich auf die wesentlichen Aufgaben des Gesamtproduktes zu beschränken. Die Verlässlichkeit kann sich durch Sicherheit ausdrücken. Bei einigen Systemen kann sogar der Nutzer die Software des eingebetteten Systems aktualisieren. Eine Programmierung neuer Software durch andere als den Hersteller ist in der Regel bei eingebetteten Systemen nicht vorgesehen oder gewünscht.

Für den Hersteller des Produktes ist ein System nur dann effizient, wenn die Kosten möglichst gering sind. Dies bedeutet, dass möglichst langsame Prozessoren und kleine Speicher verbaut werden, da diese kostengünstig zu produzieren sind. Daher muss die Software den Prozessor und den Speicher möglichst effizient nutzen. Für den Nutzer ist zusätzlich der Energiebedarf und meistens auch das Gewicht ein Kaufkriterium, so dass der Hersteller dies auch optimieren sollte.

Die Minimierung des Energieverbrauchs ist ein besonders lohnendes Feld. Energie ist begrenzt, knapp und daher teuer. Mobile eingebettete Systeme werden durch Batterien oder Akkus versorgt. Gelingt es, den Energieverbrauch zu reduzieren, kann die Verfügbarkeit erhöht werden.

Prozessoren, die für den Einsatz in eingebetteten Systemen entwickelt werden, nutzen die verbrauchte Energie effizient. Mikrocontroller besitzen neben dem Prozessor auch Speicher und Peripherie, Baugruppen für Schnittstellen und Ein- und Ausgabeleitungen in einem integriertem Schaltkreis. Sie unterscheiden sich in Ausstattung, Geschwindigkeit, Busbreite und Befehlsarchitektur. Viele Mikrocontroller besitzen keinen Adressbus, um externen Speicher anzuschließen. Sehr viele Mikroprozessoren für eingebettete Systeme besitzen kleine Speicher, die auf dem Prozessor integriert sind, und bieten trotzdem noch einen externen Adressbus zur Anbindung großer Speicher. Kleine Speicher besitzen im Vergleich zu größeren Speichern, die in derselben Technologie gefertigt werden, einen geringeren Energieverbrauch. Durch intelligente Nutzung sparsamer Speicher ist es möglich, Energie zu sparen. Diese sparsamen Speicher werden im Folgenden Scratchpad-Speicher genannt.

Ein Programm besteht aus Code und Daten. Bei der Belegung von Scratchpad-Speichern können Code und Daten getrennt behandelt werden, indem der Speicher vor der Belegung in einen Code und einen Datenbereich fest partitioniert wird. Dann ist es möglich, getrennte Speicher für Code und Daten zu verwenden. Diese Architektur wird Harvard-Architektur genannt [HP02]. Ist die Aufteilung in Programm- und Daten-Bereich nicht durch die Architektur vorgegeben, und kann entschieden werden, welcher Teil des Scratchpad-Speichers für Daten oder Code verwendet wird, dann wird ein gemeinsamer Scratchpad-Speicher für Code und Daten verwendet. Der Gesamtenergieverbrauch des Systems kann als Optimierungsziel verwendet werden.

Ein Assembler optimiert den zu übersetztenden Code nicht. Eingebettete Systeme, die in Assembler programmiert sind, werden daher nicht automatisch optimiert. Ein Assembler-Programm hat keine sichtbare Struktur. Wenn Optimierungen auf Assembler-Ebene (Drei-Adress-Code) durchgeführt werden, müssen die Programmstrukturen wiedererkannt werden. Häufig werden nur Optimierungen angewendet, die ohne Kenntnis von z. B. Schleifen auskommen. In Hochsprachen können Transformationen der Programmstruktur einfach durchgeführt werden, um das Programm zu optimierten. Die Verwendung einer Hochsprache hat weiter den Vorteil, dass die Programme lesbar bleiben und daher besser wartbar sind. Die Verlässlichkeit des Systems kann durch strukturiertes Vorgehen während der Entwicklung und Nutzung einer Hochsprache verbessert werden. Viele eingebettete Systeme werden daher auch in Hochsprachen wie C oder C++ programmiert.

Wesentliche Konzepte von Hochsprachen sind, dass Datenstrukturen und Kontrollstrukturen angeboten werden. Arrays und for-Schleifen sind grundlegende Primitive, die in keiner imperativen Hochsprache fehlen. Arrays sind häufig große Datenstrukturen auf die in einem bestimmten Muster zugegriffen wird. Tiling ist eine Code-Transformation, die dieses Zugriffsmuster optimiert. Schon im Jahr 1969 wurde in [MEGC69] ein Verfahren beschrieben, wie das Zugriffsmuster der Matrix-Multiplikation für gekachelte Speicher (paged memory) optimiert werden kann. Tiling kann auch zur Optimierung der Lokalität für Caches [Wol95, LW91] und Scratchpad-Speicher [KRI+04] eingesetzt werden.

Anschaulich kann man sich Tiling so vorstellen, dass in einem getilten Programm in einem Programmabschnitt nicht mehr auf den gesamten Speicher zugegriffen wird, sondern nur auf einen kleineren Ausschnitt. Die Daten werden in kleine Fenster aufgeteilt und nur eine begrenzte Anzahl in einem Programmteil verwendet. Nachdem für die aktuelle Position der Fenster alle Berechnungen abgearbeitet sind, werden die Fenster verschoben. Damit das Programm trotzdem noch das gleiche Ergebnis liefert, müssen die Berechnungen im Programm transformiert werden.

## 1.1 Motivation

Eine Hochsprache wie C ist geeignet, eingebettete Systeme mit geringem Aufwand zu programmieren. Compiler für eingebettete Systeme können die speziellen Eigenschaften der Zielarchitektur ausnutzen. Der existierende energieoptimierende Compiler encc [SW05] kann für Code und Daten entscheiden, ob sie im Scratchpad-Speicher abgelegt werden sollen oder nicht. Der encc löst ein Packproblem. Die Arrays und Daten sind Objekte mit fester Größe. Die Basisblöcke haben in der Regel eine feste Größe und feste Kosten. Die Basisblöcke können durch ungeschickte Anordnung um einen Befehl größer werden. Diese Optimierung lässt sich als ganzzahliges lineares Optimierungsproblem beschreiben.

Mehrere große Arrays, die nicht in einem Stück in den Scratchpad-Speicher passen, können durch den encc nicht in den Scratchpad-Speicher verschoben werden. Tiling optimiert die Zugriffsmuster auf Arrays so, dass kleinere Teil-Arrays innerhalb von Schleifen verwendet werden. Durch den Einsatz von Tiling können auch kleine Scratchpad-Speicher effizienter genutzt werden, da nun große Arrays stückweise in den Scratchpad-Speicher passen.

Bei größeren Programmen entsteht durch den Einsatz von Tiling mehr Flexibilität, da mit Tiling auch kleinere Teile des Scratchpad-Speichers noch effizient genutzt werden können.

Tiling ist eine sehr spezielle Highlevel-Optimierung, die den Programmablauf und die Zugriffsmuster auf den Speicher grundlegend verändert. Durch Tiling verändert sich der Energieverbrauch des Programms. Wenn der Energieverbrauch eines getilten Programms vorhergesagt werden kann, ist es möglich, diesen zu optimieren. Das Optimierungsproblem ist nun nicht mehr ein einfaches Packproblem. Die Parameter, die für Tiling existieren, wirken sich auf die Größe der Daten und des Codes aus. Zusätzlich entstehen Kosten, wenn Array-Teile zwischen den Speichern kopiert werden. Die resultierende Code-Größe und die Kosten der Ausführung des getilten Programms müssen modelliert werden.

Es ist sinnvoll, Code und Daten in den Scratchpad-Speicher zu verschieben. Die Kosten einzelner Instruktionen hängen davon ab, von welchem Speicher sie geladen werden. Die Kosten für Zugriffe auf Daten hängen davon ab, in welchem Speicher die Daten liegen. Existierende Arbeiten, die die Kombination von Tiling und Scratchpad-Speichern zur Energieminimierung behandeln [KRI+04], betrachten die Optimierung der Partitionierung zwischen Code und Daten nicht.

## 1.2 Ziele der Diplomarbeit

Ziel dieser Arbeit ist es, Loop-Tiling und vorhandene Algorithmen zur Scratchpad-Belegung zu kombinieren. Durch die Kombination ist eine flexiblere Nutzung des Scratchpad-Speichers möglich und es kann mehr Energie gespart werden.

Die Auswirkungen von Tiling auf das zu übersetztende Programm werden modelliert. Im Modell wird die Energie optimiert und optimale Parameter für die Programmtransformation gefunden. Das Programm wird anschließend entsprechend den Parametern transformiert und übersetzt.

Dieses Verfahren soll so in eine Toolchain integriert sein, dass verschiedene Programme automatisch übersetzt und evaluiert werden können.

Mit den Ergebnissen dieser Arbeit ist es möglich, den Energieverbrauch von eingebetteten Systemen zu reduzieren. Die Algorithmen, die in dieser Arbeit optimiert werden, finden in Systemen der digitalen Bild-, Video- und Tonsignalverarbeitung Anwendung. Die Laufzeit von Digitalkameras, Camcordern, MP3-Playern und Handys kann durch die Anwendung des Verfahrens dieser Arbeit verbessert werden.

## 1.3 Aufbau der Diplomarbeit

Im Folgenden wird ein Überblick über die Arbeit gegeben und der Inhalt der einzelnen Kapitel kurz skizziert. Die Arbeit gliedert sich wie folgt:

- **Kapitel 2 – Grundlagen**

  In diesem Kapitel werden grundlegende Konzepte dargestellt, die das Verständnis der weiteren Arbeit erleichtern.

  Zunächst wird der betrachtete Prozessor vorgestellt. Es werden gebräuchliche Speicherhierarchien eingeführt. Das verwendete Energiemodell und weitere Energiemodelle werden gegenübergestellt. Anschließend wird betrachtet wie ein Compiler funktioniert. Auf Optimierungen und insbesondere Tiling wird besonders eingegangen. Abgeschlossen wird dieses Kapitel durch eine Aufzählung, mit welchen Konzepten das vorliegende Optimierungsproblem gelöst werden kann.

- **Kapitel 3 – Verwandte Arbeiten**

  Diese Arbeit berührt verschiedene aktuelle Forschungsgebiete wie z. B. die Nutzung von Scratchpad-Speichern, Hochsprachenoptimierungen und die Abschätzung der Auswirkung von Programmtransformationen. Ausgewählte aktuelle Arbeiten zu diesen Bereichen werden in diesem Kapitel zusammengefasst. Insbesondere werden die am Lehrstuhl Informatik XII an der Universität Dortmund entstandenen Forschungsarbeiten vorgestellt, die zum Teil die Grundlage für die vorliegende Arbeit bilden.

- **Kapitel 4 – Energieeffiziente Scratchpad-Belegung durch Loop-Tiling**

  Die in Kapitel 2 erläuterten Grundlagen werden angewendet, um ein Programm zu entwickeln, das Scratchpad-Speicher möglichst effizient mit Code und Arrays belegt. Zunächst wird die Aufgabe weiter eingegrenzt und die Klasse der analysierbaren Programme definiert. Es werden die Toolchain und die verwendeten Programme und Bibliotheken vorgestellt. Nach der Beschreibung des gewählten Arbeitsablaufs werden die Aufgaben des Programms näher definiert.

  Die Aufgaben Programmanalyse, Lösung des Optimierungsproblems und Programmtransformation werden in den folgenden Abschnitten ausführlich beschrieben. Das Optimierungsproblem wird formalisiert und es werden verschiedene Lösungsmöglichkeiten aufgezeigt.

- **Kapitel 5 – Ergebnisse**

  Die erzielten Ergebnisse der Arbeit werden graphisch dargestellt und ausgewertet. Dazu wurden einige Testprogramme übersetzt und hinsichtlich Energieverbrauch, Laufzeit, Anzahl ausgeführter Instruktionen und Programmgröße analysiert.

- **Kapitel 6 – Zusammenfassung und Ausblick**

  In diesem Kapitel werden die Inhalte der vorliegenden Arbeit rekapituliert und die Ergebnisse zusammengefasst. Ein Ausblick liefert Anregungen wie die gewonnenen Ergebnisse weiter verwendet werden können.

- **Anhang**

  Im Anhang sind die generierten Ergebnisse als Rohdaten wiedergegeben. Die verwendeten Testprogramme sind als Referenz abgedruckt.

# Kapitel 2

# Grundlagen

Die Grundlagen, die im Zusammenhang mit dem Thema dieser Arbeit stehen, werden in den folgenden Abschnitten besprochen. Kenntnisse über den betrachteten Prozessor sind wichtig, um die Ergebnisse einzuordnen. Im darauf folgenden Abschnitt werden Speicherhierarchien betrachtet und Caches und Scratchpad-Speicher gegenübergestellt. Energiemodelle geben Aufschluss über den zu erwartenden Energieverbrauch und ermöglichen auch die Betrachtung von Hardware, die für diese Arbeit nicht physikalisch verfügbar ist. Ohne ein Energiemodell ist eine gezielte Optimierung unmöglich. Der Compiler soll die Optimierung durchführen und wird daher im Aufbau betrachtet. Besonders werden Optimierungen behandelt. Unter den Optimierungen wird Tiling besonders hervorgehoben. Schließlich werden zwei Lösungsmöglichkeiten für Optimierungsprobleme vorgestellt.

## 2.1 Betrachteter Prozessor

In dieser Arbeit wird der Prozessor ARM7TDMI [Adv01b] betrachtet, der als Core von Advanced RISC Machines Ltd. (ARM) lizenziert werden kann. Wesentliche Eigenschaften dieses Prozessors sind:

- 32 Bit RISC Architektur
- von Neumann Load/Store Architektur
- 3-stufige Pipeline: Fetch, Decode und Execute
- 2 Instruktionssätze (ARM- und THUMB-Mode)
- 37 Register mit jeweils 32 Bit
- 4 GByte Adressraum
- 8 (Byte), 16 (Halbwort) und 32 Bit (Wort) Datentypen
- niedriger Energieverbrauch
- 3,3 Volt Versorgungsspannung

Es werden zwei Instruktionssätze angeboten: Der ARM-Mode bietet 32 Bit RISC-Befehle mit bedingter Ausführung. Die Instruktionen des THUMB-Mode sind 16 Bit breit und werden ohne Geschwindigkeitverlust intern übersetzt. Im THUMB-Mode können nur Sprünge bedingt ausgeführt werden. Die verfügbaren Register und Größe von Konstanten in Instruktionen sind eingeschränkt. Auch im THUMB-Mode existieren Load-Multiple-and-Increment und Store-Multiple-and-Increment Befehle. Mit Hilfe dieser Befehle

ist es möglich, eine Kopierfunktion im THUMB-Mode zu implementieren, die einen zusammenhängenden Speicherbereich beliebiger Länge kopieren kann und dabei bis zu fünf Werte auf einmal behandelt.

Zu diesem Prozessor wurde in vorangegangenen Arbeiten [SKWM01, The00, Kna01] ein präzises Energiemodell entworfen, das im Abschnitt 2.3 weiter beschrieben wird. Für den THUMB-Mode sind die genaue Ausführungszeit, der Speicherplatzbedarf und der Energieverbrauch einzelner Instruktionen bekannt. Da das Energiemodell nur für den THUMB-Mode verfügbar ist, wird in dieser Arbeit nur Code betrachtet, der im THUMB-Mode ausgeführt wird.

## 2.2 Speicherhierarchien

Es existieren verschiedene Speichertechnologien, die sich darin unterschieden wie ein einzelnes Bit gespeichert wird. Zunächst können flüchtige und nicht-flüchtige Speicher unterschieden werden. Ein Bit kann als statische Ladung im Floating-Gate dauerhaft gespeichert werden. Diese Technik wird von Flash-Speichern verwendet. Bei flüchtigen Speichern kann das Bit durch Ladung in einem Kondensator gespeichert werden, die regelmäßig erneuert werden muss. Diese Technik wird von dynamischen Speichern (DRAMs) verwendet. Schließlich kann die Information in einem Flip-Flop gespeichert werden; diese Technik wird von statischen Speichern (SRAMs) verwendet. Eine Übersicht über die verschiedenen Technologien auch unter Berücksichtigung des Energieverbrauchs liefert Benini [Ben03].

Einzelne Speicherbausteine bestehen aus Bit-Feldern. Eine Steuerungslogik steuert Lese- und Schreibvorgänge. Eine Zeile wird durch einen Zeilendecoder, der den einen Teil der Adresse verarbeitet, ausgewählt. Über den Rest der Adresse wird die gewünschte Spalte ausgewählt. Die Busbreite gibt an, wieviele Elemente parallel gelesen oder geschrieben werden.

Die Zugriffsgeschwindigkeit ist bei SRAMs höher als bei DRAMs, wenn sie in ähnlicher Strukturgröße gefertigt werden, da bei SRAMs die Daten-Leitungen direkt durch Transistoren getrieben werden. Bei DRAMs muss immer eine gesamte Speicherzeile gelesen und wieder zurückgeschrieben werden. Die sehr kleinen Ladungen müssen durch Leseverstärker aufbereitet werden. Die Lesegeschwindigkeit von Flash-Speichern ist mit langsamen DRAMs vergleichbar. In [Ker05] wird ein Energiemodell für Low-Power SDRAMs und Flash-Speicher entwickelt.

Ein weiterer Einfluss auf die Geschwindigkeit und Energieverbrauch eines Speichers ist die mit ihm verbundene elektrische Kapazität, die wesentlich durch die Strukturgröße und die Leitungslänge beeinflusst wird. Die größere Leitungslänge wird durch die größere notwendige Fläche bei großen Speichern verursacht. Kleinere Speicher besitzen weniger und nicht so tiefe Zeilendecoder, so dass auf sie mit geringerer Latenz zugegriffen werden kann.

Die Geschwindigkeit von Prozessoren wächst mit Faktoren zwischen 1,5 und 2 pro Jahr die Geschwindigkeit der Speicher nur um 7% pro Jahr [HP02]. Da schnelle Speicher nicht mit einer großen Kapazität hergestellt werden können, werden in einem System kleine schnelle und große langsame Speicher eingesetzt. Es wird eine Speicherpyramide verwendet, die in Abbildung 2.1 dargestellt ist. Sie besteht aus Registern, Caches, Hauptspeicher und Hintergrundspeicher. Die Geschwindigkeit der Speicher nimmt in der Pyramide von oben nach unten ab und die Kapazität nimmt von oben nach unten zu. Register werden explizit durch den Compiler oder Programmierer durch Load-/Store-Befehle belegt. Für die Belegung des Caches ist der Cache-Controller verantwortlich. Die Belegung des Hauptspeichers geschieht i. A. durch das Betriebsystem. Durch Auslagern auf den Hintergrundspeicher wie Festplatten ist das Betriebsystem in der Lage, Teile des Hauptspeichers freizugeben. Daten auf Festplattenspeicher können auf Wechseldatenträger wie Bänder oder optische Medien ausgelagert werden.

Eine Registerbank besteht aus SRAM-Speicherzellen. Register werden bei RISC-Architekturen direkt durch Bits im Befehlswort adressiert.

Caches werden vor langsamere Speicher geschaltet, um die Latenzzeit des eigentlichen Speichers zu verstecken. Am Cache können virtuelle oder reale Adressen anliegen. Liegen virtuelle Adressen an, kommuniziert die CPU direkt mit dem Cache. Liegen reale Adressen an, so kommuniziert die Memory-Mamagement-Unit (MMU) mit dem Cache.

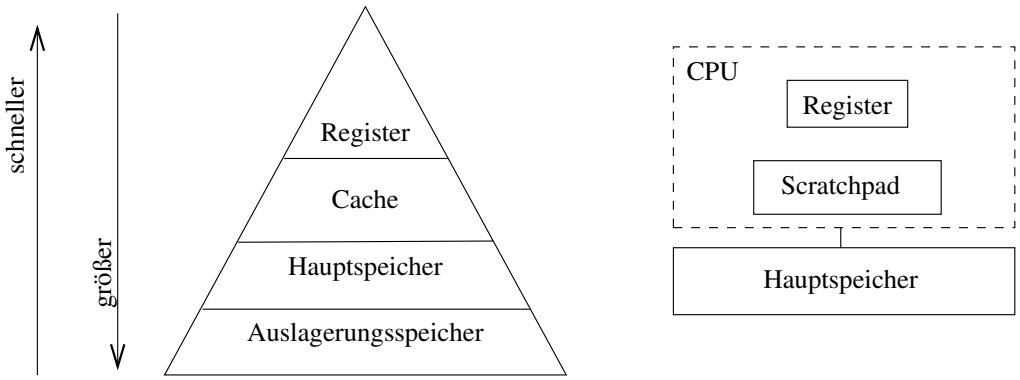


Abbildung 2.1: Die klassische Speicherpyramide und wie sie im ARM7TDMI realisiert ist

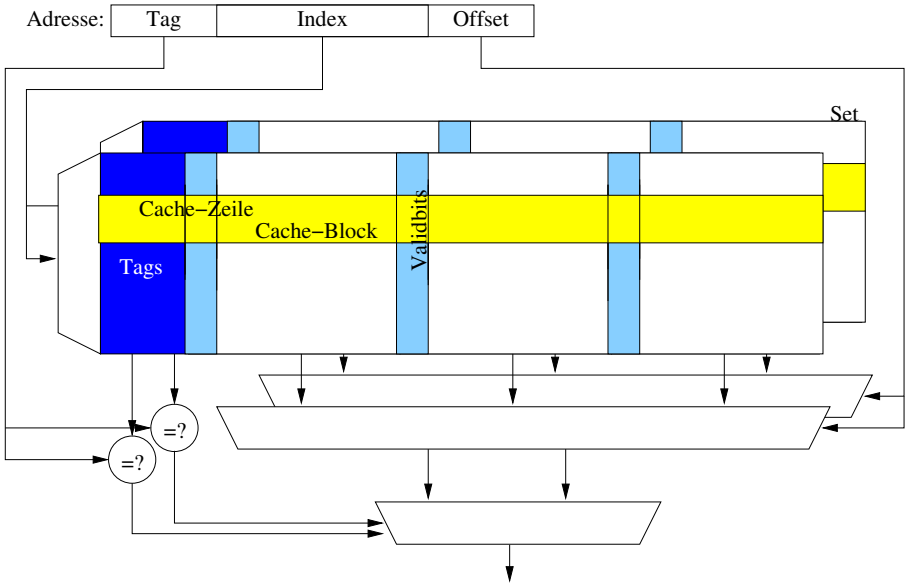


Abbildung 2.2: Aufbau eines Caches

Der Aufbau eines Caches ist in Abbildung 2.2 dargestellt. Die Anzahl der Sets ist die Assoziativität des Caches und gibt an, wie oft jede Cache-Zeile vorhanden ist. Eine Adresse, die am Cache anliegt, wird in drei Teile unterteilt: Einen Tag-, einen Index- und einen Offset-Teil. Mit Hilfe des Index wird eine Cache-Zeile je Set adressiert. Zu jeder Cache-Zeile existieren Tag-Bits, die angeben, welche Daten genau in einer Cache-Zeile gespeichert sind. Ein Datum ist nur dann in einer ausgewählten Cache-Zeile zu finden, wenn die Tag-Bits der zugegriffenen Adresse mit den Tag-Bits der Cache-Zeile übereinstimmen. Der Tag-Vergleich findet parallel in allen Sets der ausgewählten Cache-Zeile statt. Eine Cache-Zeile besteht aus einem oder mehreren Blöcken. Für jeden Block existieren Bits zur Verwaltung:

- Ein Dirty-Bit gibt bei einem Write-Back-Cache an, ob der Block vor einem Überschreiben in den unter dem Cache liegenden Speicher zurückgeschrieben werden muss.
- Ein Valid-Bit zeigt an, ob der Block der Zeile gültig ist oder erst geladen werden muss.

Mit Hilfe des Offsets der Adresse wird das Datum in der gerade aktuellen Zeile ausgewählt. In einem Cache werden immer ganze Blöcke vom Hauptspeicher ersetzt.

Besitzt ein Cache nur ein Set, dann handelt es sich um einen Direct-Mapped-Cache. Mit nur einem Set ist die zu verändernde Zeile durch den Index eindeutig bestimmt und eine Austauschstrategie existiert nicht. Ein voll assoziativer Cache besitzt nur eine Cache-Zeile. Diese Technik kommt aufgrund der Schaltungskomplexität nur selten vor. Häufig kommen Caches mit einer kleinen Assoziativität (2 bis 4) vor. Eine detaillierte Übersicht und Bewertung der verschiedenen Cache-Typen ist in [HP02] gegeben.

Caches können Daten oder Instruktionen speichern, sie werden dann Daten- oder Instruktions-Cache genannt. In einem unified Cache werden sowohl Daten als auch Instruktionen gespeichert.

Caches sind für den Nutzer sehr bequem, da der Austausch mit dem darunter liegenden Hauptspeicher automatisch erfolgt. Durch den Einsatz eines Caches kann die Ausführungsgeschwindigkeit erhöht werden. Wenn das Programm für den Cache optimiert wurde, kann der Gewinn sogar noch verbessert werden. Es ist aber auch möglich, dass ein Programm einen Cache ungünstig nutzt, wenn z. B. nur Speicheradressen verwendet werden, die nur auf wenige Cache-Zeilen abgebildet werden und die Cache-Blöcke ständig ersetzt werden müssen.

Durch den Cache-Controller und die notwendigen Bits zur Verwaltung wird zusätzlich Energie und Chipfläche verbraucht. Ein Scratchpad-Speicher benutzt wie ein Cache SRAM-Speicherzellen, besitzt aber keinen Cache-Controller und keine zusätzlichen Bits zur Verwaltung.

**Definition 2.2.1: Scratchpad-Speicher** *sind im Vergleich zum Hauptspeicher des Systems kleinere und schnellere Speicher, die in den Adressraum der CPU eingeblendet sind und durch den Compiler, den Programmierer oder das Betriebsystem belegt werden. In einem System können mehrere auch unterschiedliche Scratchpad-Speicher nebeneinander vorhanden sein.*

Scratchpad-Speicher werden häufig und aus SRAM-Zellen gefertigt in den Prozessor-Chip integriert, dadurch werden die Leitungslängen verkürzt und die Kapazität der Leitung verringert. So ist es möglich, die Speicher mit einer höheren Frequenz zu betreiben und es entstehen weniger Verluste. Bei SRAM-Zellen ist im Gegensatz zu DRAM kein Refresh notwendig, so dass weniger Energie verbraucht wird.

In den Adressraum der CPU können mehrere verschiedene Scratchpad-Speicher eingeblendet werden. Dies kann von Vorteil sein, da die Größe und der Energieverbrauch zueinander positiv korreliert sind. Durch den Einsatz von Scratchpad-Speichern unterschiedlicher Größe kann weitere Energie gespart werden [WHM04, Hel04].

Die Belegung der Scratchpad-Speicher erfolgt durch den Compiler oder den Programmierer durch direkte Belegung oder den Aufruf expliziter Kopieroperationen. Diese können in Software oder in Hardware durch eine DMA-Einheit realisiert sein.

Scratchpad-Speicher können Caches ersetzen, sie können aber auch gemeinsam mit diesen in einem System eingesetzt werden und eine komplexe Speicherhierarchie bilden.

Das Verhalten der in dieser Arbeit verwendeten Speicheranordnung wird durch Simulation nachgebildet. In der Simulation werden Scratchpad-Speicher mit unterschiedlichen Größen angeboten. Die Energiedaten für unterschiedliche Speicher wurden mit CACTI gewonnen. Die Energie-Daten für die Simulation der CPU basieren auf einem Energiemodell, dass im folgenden Abschnitt beschrieben wird. Die Messungen wurden an einem Evaluation-Board (AT91EB01) vorgenommen. Das Evaluation-Board besitzt den ARM7TDMI in Form des Microcontroller AT91M40400 [Atm99]. Der AT91M40400 besitzt keinen Cache aber 4 KByte internen Scratchpad-Speicher. Auf dem Board befinden sich 512 KByte 16 Bit Flash Speicher und 512 KByte 16 Bit SRAM. Ein Blockschaltbild [Atm99] des Evaluation-Boards ist in Abbildung 2.3 vereinfacht wiedergegeben.

## 2.3 Energiemodelle

Die momentane Leistung des Stroms ist definiert als das Produkt aus momentaner Spannung und momentanem Strom [Mös89]:

$$P(t) \quad = \quad U(t) \cdot I(t) \tag{2.1}$$

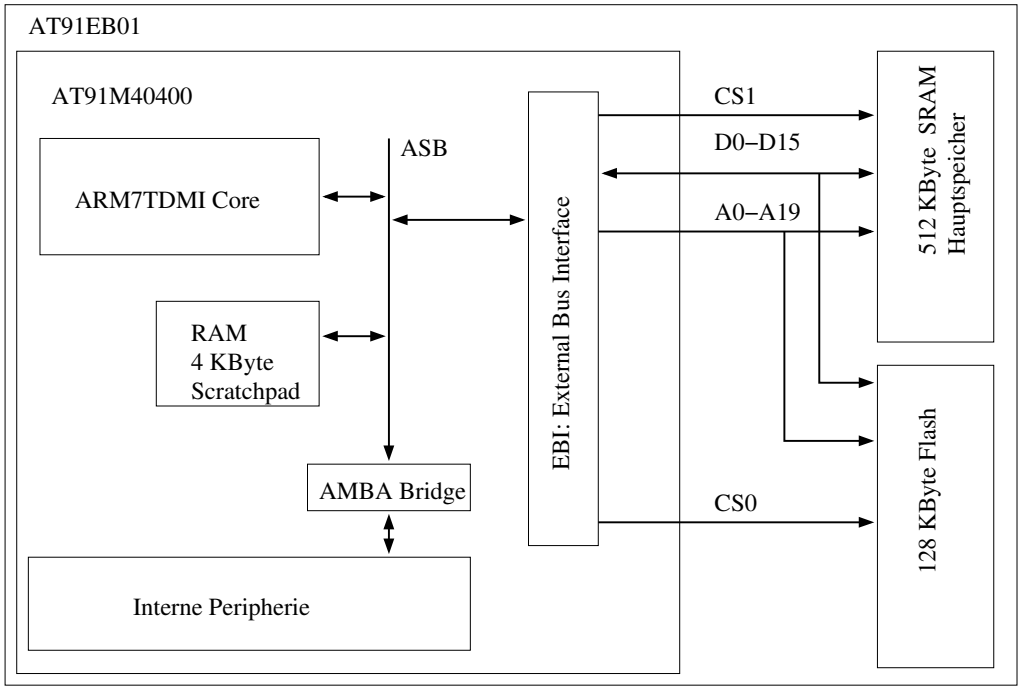


Abbildung 2.3: Vereinfachtes Blockschaltbild [Atm99] des Evaluation-Board AT91EB01

Die elektrische Energie ist das Integral der Leistung über einen betrachteten Zeitraum $[t_0, t_1]$, z. B. die Programmausführung. Die Energie, die durch die zu betrachtenden Verbraucher CPU und Speicher während der Ausführung umgewandelt wird, berechnet sich durch:

$$E = \int_{t_0}^{t_1} U(t) \cdot I(t) dt = \int_{t_0}^{t_1} P(t) dt \tag{2.2}$$

Werden Strom und Spannung über die Zeit gemittelt, dann kann die Beziehung wie folgt vereinfacht werden:

$$E = U \cdot I \cdot t = P \cdot t \tag{2.3}$$

Im statischen Zustand einer CMOS-Schaltung fließen nur Leckströme. Wenn ein Schaltvorgang stattfindet, fließt kurzzeitig ein Kurzschlussstrom, wenn die komplementären Transistoren gleichzeitig leiten. Durch häufige Signalwechsel entsteht eine oszillierende Spannung. Daher fließt durch die Kapazitäten in den Transistoren und die Kapazitäten der Leitungen ein zur Frequenz proportionaler Strom.

Die Verlustleistung einer CMOS-Schaltung ist proportional zu den vorhanden Kapazitäten, der Frequenz und dem Quadrat der Spannung [CSB92, Mar03]:

$$P = \alpha \cdot C \cdot U^2 \cdot f \tag{2.4}$$

Um Optimierungen des Energieverbrauchs durch einen Compiler vorzunehmen, muss der Energieverbrauch des generierten Programms bekannt sein. Eine immer wiederkehrende Messung des Energieverbrauchs einzelner Programmabschnitte durch den energieoptimierenden Compiler kommt nicht in Frage.

Da der konkrete Energieverbrauch eines Programms von seinem Zeitverhalten abhängt, könnte die Schaltung des Prozessors während der Ausführung simuliert werden, dies ist aber sehr zeitaufwendig und aufgrund der Schaltungskomplexität nicht praktikabel.

Energiemodelle sind ein integraler Bestandteil einer jeden Energieoptimierung. Energiemodelle sagen zu einzelnen Befehlen oder Befehlskombinationen den Energieverbrauch voraus. Effizient nutzbare Energiemodelle haben den Nachteil, dass sie häufig nicht sehr präzise sind. Eine Übersicht existierender Energiemodelle ist in [Mar03] gegeben. Hier wird nur auf eine kleine Auswahl der interessanteren Energiemodelle eingegangen.

Eine detaillierte Analyse der Effekte, die in Pipelines auftreten, berücksichtigt das Energiemodell von Lee [LEM01]. Um die einzelnen Parameter für das Energiemodell genau zu bestimmen, wird taktgenau der Strom des ARM7TDMI gemessen [CKL00]. Dazu wird die Schwankung der Versorgungsspannung über zwei Kondensatoren gemessen. Sie werden abwechselnd während eines Taktes geladen und im nächsten Takt entladen. Mit einem Oszilloskop werden die größten und kleinsten Spannungen festgehalten. Diese Anordnung ist nicht anfällig für dynamische Stromänderungen, da die Spannung über dem Haltekondensator in einem stabilen Zustand ist, während sie gemessen wird. Dieses Energiemodell hat einen durchschnittlichen Fehler von 2,5% und einen größten Fehler von 6,33%.

Eines der ersten Energiemodelle für Instruktionen wurde von Tiwari [TMW94] vorgeschlagen. Dabei werden keine detaillierten Informationen über den physikalischen Aufbau des Prozessors herangezogen. Es wird der Strom zur CPU und zum Speicher mit einem standard dual-slope digital Multimeter gemessen.

Um ein Modell zu gewinnen, werden mehrere gleiche Befehle in einer Schleife ausgeführt und der fließende Strom wird durch den Haltekondensator des Amperemeters gemittelt. Die Basiskosten für einzelne Instruktionen können so bestimmt werden. Für jede Befehlskombination kann ermittelt werden, welche Interinstruktionskosten durch einen Übergang zwischen dem Befehlspaar zusätzlich entstehen. Der Speicher wird in diesem Modell nicht explizit berücksichtigt.

Das am Lehrstuhl Informatik XII der Universität Dortmund entwickelte Energiemodell von Steinke et al. [SKWM01, The00, Kna01] lehnt sich an die Methodik von Tiwari an. Dieses Energiemodell beruht auf Messungen an vorhandener Hardware und hat eine Genauigkeit von 1,7%. Insbesondere betrachtet dieses Modell auch den Speicher, auf den während der Ausführung der Instruktionen zugegriffen wird. Die Hamming-Abstände zwischen den Daten auf dem Daten- und Adressbus werden ebenfalls berücksichtigt. Wie bei Tiwari werden Basiskosten und Interinstruktionskosten angenommen.

Es wird angenommen, dass die Kosten von der Anzahl der Einsen in einem und dem Hamming-Abstand zwischen zwei Buszuständen abhängt. Der Energieverbrauch von CPU und Speicher wird jeweils unterschieden in einen instruktionsabhängigen Teil und einen datenabhängigen Teil. Es wird weiter berücksichtigt, dass das Ein- und Ausschalten funktioneller Einheiten Energie verbraucht. Der Strom zur CPU und zum Speicher wird getrennt gemessen.

Die Genauigkeit des Modells wurde validiert, indem verschiedene Befehle in einer Schleife ausgeführt wurden und die gemessene mit der durch das Modell vorhergesagten Energie verglichen wurden.

Mit Hilfe von Energiemodellen für Speicher können die Energiemodelle für Instruktionen verfeinert oder erweitert werden. CACTI ist ein Beispiel für ein Energiemodell für Caches und Speicher. CACTI verfolgt einen analytischen Ansatz [WJ96]. Es wurde ein analytisches Modell entwickelt, das mit dem HSPICE-Modell der zu betrachtenden Schaltungen verglichen wurde. Dabei wurde eine Genauigkeit von 6% ermittelt. Das Programm [RJ00] wird mit den Parametern für Cachegröße, Blockgröße, Assoziativität und Technologiegröße aufgerufen und liefert die wesentlichen Parameter für ein Energiemodell, um die für Zugriffe auf den Cache notwendige Energie abzuschätzen.

Mit Hilfe von CACTI wurden Parameter für verschiedene Scratchpad-Speicher unterschiedlicher Größe generiert, die dazu dienen, im enProfiler oder encc unterschiedliche Scratchpad-Speicher zu betrachten.

Das in dieser Arbeit verwendete Energiemodell von Steinke et al. [SKWM01, The00, Kna01], das durch die Ergebnisse von CACTI erweitert wurde, verwendet eine Energiedatenbank. Die Energiedatenbank wird vom encc dem enProfiler und auch von dem in dieser Arbeit erstellten Programm verwendet. Der encc und der enProfiler werden in den Abschnitten 4.3.2 auf Seite 45 bzw. 4.3.3 auf Seite 48 beschrieben.

## 2.4 Compiler

Eingebette Systeme haben hohe Anforderungen an die Effizienz. Da viele Compiler schlecht optimieren, vertrauen viele Entwickler auf ihre eigenen Fähigkeiten und schreiben die Software für eingebette Systeme in Assembler.

Dieses Vorgehen hat aber wesentliche Nachteile. Durch die geringe Abstraktion ist das Programm nicht portierbar und spätere Erweiterungen fallen schwer. Optimierungen können effizient durch Compiler durchgeführt werden, wenn diese in der Lage sind, die Auswirkungen der Codeauswahl zu berechnen.

Compiler existieren für verschiedene Hochsprachen. Es werden imperative, funktionale und logische Sprachen unterschieden. Funktionale und logische Sprachen werden häufig nicht direkt übersetzt, sondern interpretiert. Vor allem logische Sprachen bieten das höchste Maß der Abstraktion. Es ist möglich, den Sortieralgorithmus Quick-Sort in der funktionalen Programmiersprache Haskell in zwei Zeilen zu implementieren:

```
qsort []   = []
qsort x:xs = qsort [y|y<-xs, y<=x] ++ x ++ qsort [y|y<-xs, y>x]
```

Es existieren zwei unterschiedliche Methoden der Übersetzung für imperative Sprachen: Die attributierte Übersetzung und die Übersetzung durch Programmtransformation [ASU86, Muc97].

Die meisten Zwischenrepräsentationen erlauben es, an die einzelnen Knoten der Datenstruktur Attribute anzuhängen, die Informationen für Code-Analyse, Optimierung und Übersetzung speichern. Über den Wurzelknoten des attributierten Syntaxbaums kann das Zielprogramm als Attribut abgelesen werden. Für jedes Attribut existieren Funktionen, die das Attribut aus den Attributen der Nachbarknoten rekursiv berechnen. Durch sukzessive Iteration über alle Funktionen und Knoten kann das zu generierende Programm berechnet werden. Die Anzahl der Pässe eines Compilers ist die Anzahl der Durchläufe durch den Ausdrucksbaum, die zur Erzeugung des Programms notwendig sind [ASU86].

Bei der Übersetzung durch Programmtransformation werden alle Teile des Originalprogramms durch semantisch äquivalente Konstrukte der Zielsprache übersetzt. Ausdrucksbäume werden z. B. mit einem Keller übersetzt. Schleifen werden durch mehrere Anweisungen ersetzt [ASU86].

Imperative Sprachen wie C lassen sich in Maschinenprogramme übersetzen. Diese Übersetzung wird von einem Compiler vorgenommen. Ein Compiler besteht aus einem oder mehreren Programmen. Die Übersetzung findet in der Regel in mehreren Phasen statt. Eine detaillierte Beschreibung der Phasen und Optimierungen ist in [Muc97] gegeben, die hier zusammengefasst wird:

**Präprozessor** Ein Präprozessor entfernt Kommentare und löst Präprozessor-Makros und Präprozessor-Anweisungen auf.

**Frontend** Das Frontend liest die Ausgabe des Präprozessor. In mehreren Schritten wird die Eingabe für das Backend erzeugt.

- Die lexikalische Analyse zerlegt die Datei in Token. Ein Token ist ein nicht zerlegbares Programmelement und wird durch eine reguläre Grammatik definiert. Beispiele für Token sind: ’+’, ’-’ und ’for’.
- Die syntaktische Analyse erzeugt einen abstrakten Syntaxbaum. Dies ist eine Datenstruktur, die einzelne Programmteile wie Schleifen, Statements oder Ausdrücke repräsentiert. Dieser Baum wird durch eine attributierte Grammatik generiert. Hier werden Fehler wie z. B. fehlende schließende Klammern erkannt.
- Bei der semantischen Analyse werden Fehler in der Typ-Kompatibilität entdeckt und wenn nötig implizite Typumwandlungen explizit eingefügt, z. B. Umwandlungen von int nach float.
- Es folgen Highlevel-Optimierungen auf der Datenstruktur des abstrakten Syntaxbaumes.
- Abschließend erfolgt die Umwandlung in eine maschinenabhängige Zwischendarstellung.

Der attributierte Syntaxbaum ist die zentrale Datenstruktur des Frontends, sie wird auch Highlevel Intermediate Representation (HIR) genannt. Die Abstraktionen von Programmiersprachen durch Funktionen und explizite Schleifen bleiben erhalten.

**Backend** Das Backend ist maschinenabhängig und erzeugt die zur Eingabedatei gehörende Assembler-Datei aus der Zwischendarstellung des Frontends. Es werden verschiedene Optimierungen durchgeführt. Ein wesentlicher Schritt ist die Registerallokation; die benutzten virtuellen Register werden auf die vorhandenen physikalischen Register abgebildet. Wenn zu wenig Register vorhanden sind, werden Register auf den Stack ausgelagert (gespillt).

Die verwendete Datenstruktur wird Lowlevel Intermediate Representation (LIR) genannt. Im Backend werden nur Sprünge, Label und Instruktionen behandelt.

**Assembler** Die generierte Assembler-Datei wird vom Assembler nach Objektcode übersetzt. Die Objekt-Datei enthält noch Platzhalter für Adressen, die menschenlesbaren Assembler-Befehle (Memonics) werden in Programmworte übersetzt.

**Linker** Der Linker verbindet schließlich eine oder mehrere Objekt-Dateien zu einem ausführbaren Programm, indem die Platzhalter für Adressen durch feste Adressen ersetzt werden. Die genaue Anordnung der verschiedenen Programmteile wird durch ein Linker-Skript definiert. In dieser Datei ist auch definiert wie die verschiedenen Programmbereiche zu initialisieren sind.

### 2.4.1 Optimierungen

Optimierungen können danach unterschiedenen werden, ob sie auf den Datenstrukturen des Frontends, des Backends oder beiden arbeiten. Compiler-Optimierungen zielen darauf ab, den Code nach einem Kriterium zu verbessern: Laufzeit, Codegröße oder Energieverbrauch. Nicht immer können die Folgen einer Optimierung exakt abgeschätzt werden, so dass unter ungünstigen Umständen keine Verbesserung oder sogar eine Verschlechterung eintritt.

Oberstes Gebot aller Optimierungen ist es, dass das Ergebnis des Programms auf der ausführenden CPU durch die Optimierung nicht verändert wird.

**Definition 2.4.1:** *Eine* **Zugriffsfunktion** *ist ein arithmetischer Ausdruck, mit dem auf ein Element eines Arrays zugegriffen wird, z. B.* `A[i+1][j]`*. Die Zugriffsfunktionen mehrdimensionaler Arrays können auf eine eindimensionale abgebildet werden,z. B.* `A[10*i+1+j]`*.*

Wesentliche Highlevel-Optimierungen sind:

**Skalare Ersetzung von Array-Zugriffen** Es müssen dazu die Wertebereiche der Zugriffsfunktionen der Arrays in allen Dimensionen berechnet werden. Damit ist bekannt, welche Vorkommen auf die gleichen Zellen zugreifen, die dann in einem Register gehalten werden können. Auch muss sichergestellt sein, dass über keine anderen Pointer auf Array-Elemente zugegriffen wird. Im Zweifel können Array-Zugriffe nur in einem kleinen Codebereich ersetzt werden, für den die notwendigen Informationen berechnet werden können.

**Schleifenoptimierungen** verändern das Ausführungsverhalten von Schleifen.

**Daten-Cache-Optimierungen** Ein Beispiel ist Tiling. Es kann der Datenspeicherzugriff so verändert werden, dass die Daten besser im Cache gehalten werden können. Diese Optimierung wird im Abschnitt 2.4.2 beschrieben.

Diese Highlevel-Optimierungen dürfen nicht angewendet werden, wenn zwei Programme über gemeinsamen Speicher Informationen austauschen oder wenn Memory-Mapped-I/O-Geräte gesteuert werden sollen. Für viele Optimierungen existieren Kommandozeilen-Schalter oder Annotationen (Pragmas), die die Wirkung von Optimierungen steuern.

Optimierungen, die in Schleifen angewendet werden, haben einen großen Effekt, da Befehle in einer Schleife häufiger ausgeführt werden und die Mehrzahl der ausgeführten Befehle in Schleifen ausgeführt wird.

Einen sehr großen Teil der Zeit muss ein Programm in einer oder mehreren Schleifen verbringen, wenn ein begrenzter Speicher für Instruktionen zur Verfügung steht. Der Microcontroller PIC12F675 hat eine Taktfrequenz von 4 MHz und es werden 4 Takte pro Befehl benötigt. Pro Sekunde werden 1 000 000 Befehle ausgeführt. Der Instruktions-Speicher kann aber nur 1024 Befehle speichern [Mic03]. Nach spätestens $\frac{1024}{1\,000\,000}\text{s} = 1,024\text{ms}$ wird mindestens ein Befehl wiederholt, also in einer Schleife ausgeführt.

Insbesondere for-Schleifen sind für die Optimierung interessant. Schleifen werden durch die Induktionsvariablen charakterisiert.

**Definition 2.4.2: Induktionsvariablen** *sind Variablen, deren Wertfolge einen arithmetischen Fortschritt über einen Teil des Programms bilden.* Abhängige Induktionsvariablen *sind skalare Variablen, die vor einer Schleife im Kontrollflussgraphen mit einem Ausdruck initialisiert werden. Sie werden in der Schleife nur einmal beschrieben. Der Rücksprung zum Beginn der Schleife hängt von einem Vergleich der Induktionsvariablen mit einem anderen Ausdruck ab.*

Fundamentale Induktionsvariablen *sind abhängige Induktionsvariablen, die in jeder Iteration um dieselbe Schrittweite verändert werden. [Muc97]*

Kontrollvariablen *sind fundamentale Induktionsvariablen, deren Schrittweite 1 oder -1 ist.*

Schleifenoptimierungen können auch als Lowlevel-Optimierungen ausgeführt werden, wenn die Induktionsvariablen wiedererkannt werden können. In Sprachen wie Pascal ist fest definiert, dass eine Kontrollvariable mit einem Ausdruck initialisiert wird und bei einem Endwert abgebrochen wird. Die Schrittweite ist entweder 1 oder -1 [Hei87]. Sprachen wie C haben keine Einschränkungen, welche Ausdrücke in den drei Feldern der for-Schleife erlaubt sind. For-Schleifen in C können weiter eingeschränkt werden.

**Definition 2.4.3: Artige Schleifen** *sind for-Schleifen, die in der Initialisierung eine fundamentale Induktionsvariable beschreiben. Die Wiederholung der Schleife hängt nur vom Vergleich derselben fundamentalen Induktionsvariablen mit einem Ausdruck ab, der in der Schleife eine Konstante ist. Die Schrittweite der fundamentalen Induktionsvariablen ist ein konstanter Ausdruck. [Muc97]*

**Definition 2.4.4: Normalisierte Schleifen** *sind artige Schleifen, deren Kontrollvariablen mit 0 initialisiert werden, deren Schrittweite 1 ist und die eine obere Schranke besitzen, die eine natürliche Konstante ist.*

**Definition 2.4.5:** *Ein* **affiner Ausdruck** *ist ein Ausdruck, der aus einer Summe von ganzzahlig gewichteten Induktionsvariablen und ganzzahligen Konstanten gebildet wird.*

Eine grundlegende Schleifentransformation ist die Schleifennormalisierung. Dabei werden artige Schleifen in normalisierte Schleifen umgewandelt. Die fundamentale Induktionsvariable wird durch eine Kontrollvariable ersetzt und alle Vorkommen der fundamentalen Induktionsvariablen durch einen passenden affinen Ausdruck ersetzt.

Die durch Normalisierung redundant eingeführten Operationen können durch nachfolgende Optimierungen, wie Redundant Code Elimination, wieder rückgängig gemacht werden.

**Beispiel 2.4.1: Normalisierung artiger Schleifen**

```
for (i=3; i<=17; i+=5)
  for (j=2*i+5; j<=2*i+11; j++)
    A[5*j+4][3*i+2] = 5;
```

In diesem Beispiel hat das Array A die zwei Zugriffsfunktionen $5 \cdot j + 4$ und $3 \cdot i + 2$.

Zuerst wird dafür gesorgt, dass jede Schleife von 0 bis zur oberen Grenze minus der unteren Grenze läuft. Es müssen alle inneren affinen Zugriffsfunktionen angepasst werden. Wenn $i$ der Koeffizient in der aktuellen Zugriffsfunktion ist, dann muss die untere Schranke mit $i$ multipliziert zu der Zugriffsfunktion addiert werden.

```
for (i=3-3; i<=17-3; i+=5)
  for (j=2*i+5+2*3; j<=2*i+11+2*3; j++)
     A[5*j+4][3*i+2+3*3] = 5;

for (i=0; i<=14; i+=5)
  for (j=2*i+11; j<=2*i+17; j++)
     A[5*j+4][3*i+11] = 5;

for (i=0; i<=14; i+=5)
  for (j=0; j<=6; j++)
     A[(5*j+4+5*(2*i+11))][3*i+11] = 5;

for (i=0; i<=14; i+=5)
  for (j=0; j<=6; j++)
     A[(5*j+10*i+59)][3*i+11] = 5;
```

Im Folgenden wird die obere Grenze durch die Schrittweite dividiert und stattdessen die Schrittweite 1 verwendet. In allen affinen Funktionen im Schleifenkörper wird der zu dieser Schleife gehörende Koeffizient mit der Schrittweite multipliziert.

```
for (i=0; i<=14/5; i++)
  for (j=0; j<=6; j++)
     A[(5*j+5*10*i+59)][5*3*i+11] = 5;

for (i=0; i<=2; i++)
  for (j=0; j<=6; j++)
     A[5*j+50*i+59][15*i+11] = 5;
```

Schleifenverschmelzung fügt Schleifen mit gleichen Schleifenköpfen zusammen, so dass die Verarbeitung der Induktionsvariablen weniger oft stattfindet und ein größerer Schleifenrumpf entsteht.

Die inverse Operation ist das Schleifentrennen; hier wird aus einer Schleife eine zweite weitere erzeugt, die nur einen Teil der Befehle der ursprünglichen enthält. Dabei kann das Ziel verfolgt werden, dass in einer der beiden Schleifen nun Optimierungen erlaubt sind, die vorher verboten waren.

Durch Schleifenvertauschen können Schleifen, die weniger oft ausgeführt werden, nach außen verschoben werden, so dass die Kontrollstrukturen der inneren Schleifen weniger oft ausgeführt werden.

Durch Schleifenumdrehen werden Schleifen rückwärts abgearbeitet.

Durch Strip-Mining wird eine Schleife nicht sofort komplett abgearbeitet, sondern immer nur eine gewisse Anzahl Iterationen. Strip-Mining wird im Zusammenhang mit Tiling im Abschnitt 2.4.2 näher erläutert.

**Definition 2.4.6:** *Ein* **perfektes Schleifennest** *ist ein Nest aus mehreren ineinander verschachtelten artigen Schleifen, bei der nur in der innersten Schleife Anweisungen enthalten sind.*

Perfektionierung eines Schleifennestes erzeugt ein perfektes Schleifennest, indem Anweisungen, die neben den eigentlichen Schleifen in den nicht innersten Schleifen existieren, in die innerste Schleife verschoben werden und durch geeignete Bedingungen nur zu den ursprünglichen Iterationspunkten ausgeführt werden.

Code Hoisting ist die inverse Operation und sorgt dafür, dass gemeinsame Ausdrücke möglichst selten ausgeführt werden. Advanced Code Hoisting [Fal04] berücksichtigt dabei Kontrollflussaspekte und nimmt zusätzlich in Kauf, dass Teile des Nestes dupliziert werden müssen.

Schleifenoptimierungen können mehrere Ziele verfolgen. Durch einen größeren Rumpf entstehen größere Basisblöcke im Inneren der Schleife, die mehr Möglichkeiten zur Optimierung bieten können. Durch Umformung der Schleifen können weitere Schleifenoptimierungen erlaubt werden. Durch Umformungen kann die Abhängigkeitsanalyse vereinfacht werden.

Optimierungen, die immer ausgeführt werden können, sind die Faltung von Konstanten und algebraische Vereinfachungen. Dabei werden komplexe Ausdrücke aus Konstanten durch ihr Ergebnis ersetzt. Es muss dabei darauf geachtet werden, dass das Ergebnis mit der Ausführung auf der Zielplattform übereinstimmt. Dies kann z. B. bei Fließkommazahlen oder Sättigungsarithmetiken eine Simulation des Zielrechners notwendig machen.

Arithmetische Vereinfachungen ersetzten einzelne Anweisungen durch einen einfacheren Befehl, so kann es effizienter sein, eine Multiplikation mit 2 durch eine Schiebeoperation zu ersetzen. In Schleifen können häufig Multiplikationen mittels Strength-Reduction durch billigere Additionen ersetzt werden.

Die folgenden wesentlichen Optimierungen können auf jeder Repräsentation des Programms durchgeführt werden:

**(Globale) Wertnummerierung** Hierbei wird erkannt, ob zwei Berechnungen dasselbe Ergebnis haben. Eine Berechnung kann weggelassen werden, indem das vorhergehende Ergebnis verwendet wird. Zu jeder Berechnung wird ein symbolischer Wert berechnet, ohne dass die Berechnungen durchgeführt werden, aber so dass gleiche Symbole immer die gleiche Berechnung repräsentieren. Für eine genaue Beschreibung sei auf [Muc97] verwiesen.

Wesentlich ist, dass die Anfangswerte von in die Berechnung eingehenden Variablen nicht bekannt sein müssen.

**Konstantenweiterleitung** Wenn einer Variablen eine Konstante zugewiesen wird, wird die Variable durch die Konstante ersetzt, solange auf die Variable nicht geschrieben wurde.

**Entfernung gemeinsamer Unterausdrücke** Hier werden Ausdrücke im Kontrollflussgraphen gesucht, die schon vorher berechnet wurden. Zwischenergebnisse werden dabei explizit gespeichert und wiederverwendet.

In [Muc97] wird gezeigt, dass diese drei Optimierungen unterschiedliche Optimierungsmöglichkeiten haben und nicht durch einander ersetzt werden können.

**Entfernung von totem Code** Nachdem eine Bedingung durch eine der vorhergehenden Methoden bekannt ist, kann die Bedingung zur Übersetzungszeit ausgewertet werden und nie ausgeführter Code entfernt werden.

**Partial Redundancy Elimination** kombiniert globale Entfernung gemeinsamer Unterausdrücke mit der Verschiebung von schleifenunabhängigen Code. Eine Variante ist Lazy Code Motion [KRS92], die gemeinsamen Ausdrücke werden dabei möglichst spät, aber trotzdem gewinnbringend, außerhalb der Schleife positioniert, um den Registerdruck zu senken.

Lowlevel-Optimierungen finden im Backend statt, dabei ist der Code schon in Basisblöcke aufgeteilt und liegt in Drei-Adress-Instruktionen vor.

Um eine lange Pipeline zu füllen, kann es effizient sein, Schleifen abzurollen, um einen größeren Basisblock zu erhalten.

Auch die Anordnung der Basisblöcke kann optimiert werden, indem das wahrscheinlichere Sprungziel eines bedingten Sprungs als nächster Basisblock angefügt wird.

Für bedingte Sprünge kann jetzt entschieden werden, ob es von Vorteil ist, die bedingte Ausführung von Instruktionen zu verwenden, damit die Pipeline des Prozessors gefüllt bleibt. Dies ist dann von Vorteil, wenn nur wenige Befehle übersprungen werden müssen.

Beim Laden von Konstanten müssen Entscheidungen getroffen werden wie die Konstante geladen wird. Konstanten können

- als Immediate im Instruktionswort abgelegt werden,
- programmzählerrelativ beim Code gespeichert und dann geladen werden (Literalpool) oder
- durch arithmetische Operationen aus kleinen Konstanten gebildet werden.

Welche dieser Möglichkeiten verwendet wird, hängt davon ab, welche Möglichkeiten der Prozessor unterstützt und wieviel die einzelnen Möglichkeiten kosten. Dabei ist auch zu berücksichtigen, ob die Kosten auf mehrere Konstanten verteilt werden können.

Die Registerallokation ist die zentrale Aufgabe des Backends, die nicht im Frontend erledigt werden kann, weil erst jetzt alle verwendeten Befehle bekannt sind und Einschränkungen in der Registernutzung berücksichtigt werden können.

Im Folgenden wird das Konzept der Registerallokation durch Graphfärbung skizziert. Diese und andere Möglichkeiten der Registerallokation sind in [Muc97] näher beschrieben.

- In der vorherigen Repräsentation wurden alle Objekte in virtuellen Registern gehalten. Zunächst muss entschieden werden, für welche Objekte Register notwendig sind. Kleine Konstanten können beispielsweise direkt im Befehlswort gespeichert werden.
- Es wird ein Interferenz-Graph angelegt.
- Der Interferenz-Graph wird $R$-gefärbt, wobei $R$ die Anzahl der physikalischen Register ist.
- Anhand der Farbnummern werden Register zugewiesen.

Zunächst muss entschieden werden, für welche Objekte überhaupt Register notwendig sind. Kleine Konstanten und bisher genutzte Variablen sind in der Regel nicht geeignet. Variablen können in sehr unterschiedlichen Kontexten gebraucht werden, da sie häufig wiederverwendet werden, z. B. `i` als häufig verwendete Kontrollvariable. Es werden stattdessen Webs eingeführt. Ein Web ist die maximale Vereinigung von überlappenden Def-Use-Chains. Eine Def-Use-Chain beginnt mit der Definition, einem schreibenden Zugriff einer Variablen und besitzt eine oder mehrere lesende Zugriffe der Variablen, die durch einen Pfad im Kontrollflussgraphen erreicht werden können. Zwei Def-Use-Chains überlappen, wenn sie einen gemeinsamen Lesezugriff besitzen.

Für jedes Web wird ein Knoten im Interferenz-Graphen angelegt. Zwischen den Knoten im Interferenz-Graphen wird eine Kante eingefügt, wenn zum Definitionszeitpunkt des einen Web das andere Web lebendig ist.

Ein weiterer wesentlicher Schritt beim Anlegen des Graphen ist es, zu erkennen, welche Webs miteinander durch eine Kopierinstruktion verknüpft sind. Diese Webs werden im Schritt der Subsumierung zusammengefasst. Webs, die schon vorher aufgrund von Aufrufkonventionen an Register gebunden wurden, dürfen nicht subsumiert werden.

Schon die 3-Färbung eines Graphen ist NP-vollständig. Die verwendete Heuristik kann also nicht immer die optimale Lösung finden. Die Heuristik nutzt folgenden Satz aus:

Gegeben sei ein Graph, der einen Knoten mit dem Grad kleiner als *R* enthält. Der Graph ist *R*-färbbar genau dann, wenn der Graph ohne diesen Knoten *R*-färbbar ist [Muc97]. Für jeden Knoten mit weniger als *R* Nachbarn kann eine Farbe ausgewählt werden, die die Nachbarn noch nicht haben.

Wenn *R* Register verfügbar sind, soll eine R-Färbung gefunden werden. Es kann folgender Algorithmus verwendet werden [Cha82]:

1. Knoten mit weniger als *R* Nachbarn werden zusammen mit ihren Kanten entfernt und auf einen Stack gelegt. Dabei wird der Grad der Nachbar-Knoten verringert.
2. Ist der Graph jetzt noch nicht leer, d. h. es existieren nur noch Knoten mit mehr als *R* Nachbarn, werden die Knoten mit den meisten Kanten betrachtet und unter diesen der Knoten mit den kleinsten Spillingkosten gespillt und aus dem Graphen entfernt. Wenn wieder Knoten weniger als *R* Nachbarn haben, dann wird mit Schritt 1 fortgefahren, sonst wird dieser Schritt wiederholt.
3. Der Interferenz-Graph ist nun leer. Der Stack wird elementweise geleert und jeder Knoten mit der kleinsten Farbe gefärbt, die seine Nachbarn noch nicht haben

Spilling bedeutet, dass das Register zu einem bestimmten Zeitpunkt auf den Stack ausgelagert wird und erst bei seiner späteren Verwendung wieder eingeladen wird. Die Spillingpositionen können optimiert werden, so dass sie möglichst selten auf den Stack zugreifen. Wird der Spillcode vor und nach einer Schleife eingefügt, ist das Register in der Schleife nicht mehr lebendig und der Spillcode wird nur sehr selten ausgeführt.

### 2.4.2 Highlevel-Optimierung: Tiling

Tiling ist eine Kombination von Strip-Mining und dem Vertauschen von Schleifen. Zunächst wird Strip-Mining erläutert und anschließend mit dem Schleifenvertauschen kombiniert. Schließlich wird auf die mathematische Auffassung eingegangen und zusammengefasst, wie die Legalität von Tiling und anderen Highlevel-Optimierungen geprüft werden kann.

Strip-Mining ist eine einfache Schleifentransformation, die alleine angewendet immer möglich ist, aber auch immer zusätzlichen Overhead erzeugt. Strip-Mining bedeutet, dass eine Schleife nicht sofort komplett abgearbeitet wird, sondern immer nur ein Teil. Aus einer for-Schleife werden zwei for-Schleifen. Die innere Schleife wird Elementschleife genannt, die äußere Kontrollschleife.

Die untere und obere Grenze der Kontrollschleife ist die untere bzw. obere Grenze der Originalschleife. Die Schrittweite der Kontrollschleife hat das gleiche Vorzeichen wie die Originalschleife und kann im Bereich zwischen 1 und der Ausführungshäufigkeit der Originalschleife frei gewählt werden. Die Schrittweite darf nicht 0 sein und sollte größer als 1 sein.

Die Elementschleife hat die gleiche Schrittweite wie die Originalschleife; sie beginnt mit dem Wert der fundamentalen Induktionsvariablen der Kontrollschleife und bricht bei einer „passenden“ oberen Schranke ab. Die obere Schranke der Kontrollschleife ist passend, wenn im Originalschleifennest und im Schleifennest aus Kontrollschleife und Elementschleife immer die gleichen Indizes erreicht werden.

Tiling kombiniert Strip-Mining mit dem Vertauschen von Schleifen. Dabei werden alle Kontrollschleifen nach außen verschoben und Tiling-Schleifen genannt. Die Schrittweite der Tiling-Schleifen ist der Tiling-Faktor. Die fundamentalen Induktionsvariablen der Tiling-Schleife sind die Tiling-Indizes.

Die Namen der Induktionsvariablen der Schleifen werden in dieser Arbeit wie folgt gewählt: Die Elementschleife erhält als Induktionsvariablenname den der Originalschleife. Die Tiling-Schleife benutzt denselben Namen, wobei ein ‚t‘ (für Tiling-Schleife) angehängt wird. Für den Namen des Tiling-Faktors wird stattdessen ein ‚T‘ angehängt.

Da in dieser Arbeit nur perfekte Schleifennester aus artigen Schleifen betrachtet werden, soll hier der Zusammenhang zwischen oberer und unterer Schranke auch nur für diesen Fall beschrieben werden.

Für eine Schleife, die eine fundamentale Induktionsvariable mit dem Namen *i* besitzt, deren untere Schranke *u*, obere Schranke *o* und deren Schrittweite $s > 0$ ist, kann ein Tiling-Faktor *iT* gewählt werden. Die untere Schranke der Elementschleife ist *it*, die Induktionsvariable der Tiling-Schleife. Die obere Schranke der Elementschleife ist dann $it + min(o, it + s)$. Wenn der Tiling-Faktor ein Teiler von $o - u$ und kleiner als $o - u$ ist, kann die obere Schranke der Elementschleife vereinfacht werden zu $it + s$. Das folgende Beispiel veranschaulicht das Tiling an einer Schleife.

**Beispiel 2.4.2: Tiling einer einzelnen Schleife**

Sei folgende einzelne Schleife gegeben. Für die Konstanten gelte: $o, u, iT, s \in \mathbb{N}$ $u \geq 0$, $o > u$, $s > 0$.

```
for (i=u; i<o; i+=s)
```

Nach dem Tiling dieser einzelnen Schleife ist folgendes Nest gegeben:

```
for (it=u; it<o; it+=iT)
  for (i=it; i<min(o,it+iT); i+=s)
```

Wenn gilt $o - u = iT \cdot b$ und $iT < o - u$, dann kann das Nest vereinfacht werden zu:

```
for (it=u; it<o; it+=iT)
   for (i=it; i<it+iT; i+=s)
```

Es ist egal, ob die obere Schranke durch < oder <= getestet wird. Im zweiten Fall kann einfach 1 von der oberen Schranke abgezogen werden.

Damit Tiling legal ist, muss das Vertauschen von Schleifen im Nest erlaubt sein. Abhängigkeitsvektoren und die Darstellung der Transformation als unimodale Matrix können benutzt werden, um dies zu testen. Umfangreiche Einführungen in diese Methoden sind in [Muc97, AK02] gegeben, die hier auszugsweise zusammengefasst werden.

Eine Belegung von Induktionsvariablen eines perfekten Schleifennestes mit *ld* Schleifen entspricht einem Punkt in einem *ld*-dimensionalen Raum $\mathbb{Z}^{ld}$ . Dieser Punkt wird Iterationspunkt genannt. Durch die oberen und unteren Schranken der Schleifen wird ein Teilraum definiert, der alle Iterationspunkte des Nestes enthält.

Affine Funktionen erzeugen Hyperebenen. Eine Hyperebene im *n*-dimensionalen Raum ist ein $n-1$-dimensionaler Unterraum. Eine Gerade ist im zweidimensionalen Raum eine Hyperebene. Hyperebenen zerteilen den Raum in zwei Halbräume.

Um ein normalisierbares Schleifennest zu erhalten, wird gefordert, dass die oberen und unteren Grenzen des Schleifennestes affine Funktionen der umgebenden Induktionsvariablen sind. Durch die affinen Grenzen sind Hyperebenen definiert, die jeweils den Raum in zwei Halbräume teilen, von denen der eine Halbraum eine Obermenge des Iterationsraum ist. Der Iterationsraum ist eine Schnittmenge von Halbräumen und bildet ein (konvexes) Polytop. Der Iterationsraum eines einfachen Schleifennestes ist in Abbildung 2.4 dargestellt. Ein Polytop ist ein Polyhedron, da es kein unendliches Volumen besitzt. Da das Programm terminieren soll, ist also insbesondere die Menge der Iterationspunkte beschränkt [Fal04, Wil93].

Jeder Punkt ist ein Tupel aus *ld* Elementen, der Index-Vektor genannt wird. Die Elemente des Vektors sind die Werte der Induktionsvariablen der Schleifen von außen nach innen. Für die Schrittweiten der Schleifen soll angenommen werden, dass sie positiv sind. Dann entspricht die Reihenfolge der besuchten Iterationspunkte der lexikographisch positiven Sortierung der Index-Vektoren.

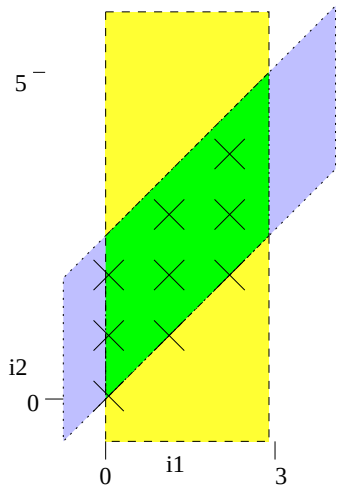


Abbildung 2.4: Index-Bereich des Schleifennestes zum Beispiel 2.4.3

**Beispiel 2.4.3: Ausführung eines Schleifennestes und die lexikographische Ordnung der Index-Vektoren**

Es sei das folgende perfekte Schleifennest gegeben:

```
printf("besuchte Index-Vektoren:");
for (i1=0; i1<3; i1++)
  for (i2=i1; i2<i1+3; i2++)
    printf("(%d,%d) ",i1,i2);
}
```

Das Programm hat folgende Ausgabe:

```
besuchte Index-Vektoren:(0,0) (0,1) (0,2) (1,1) (1,2) (1,3)
    (2,2) (2,3) (2,4)
```

Die Index-Vektoren sind lexikographisch positiv sortiert. Der Index-Bereich des Schleifennestes ist in Abbildung 2.4 graphisch dargestellt. Der von i1 überdeckte Bereich beginnt bei 0 in horizontaler Richtung und endet bei 2. Dies wird durch den senkrechten Balken dargestellt (gelb, $--$). Der Index-Bereich von i2 beginnt bei der Identität und wird nach oben durch die Gerade $x+3=y$ beschränkt, der diagonale Balken veranschaulicht diesen Index-Bereich (blau, $\cdot\cdot$). Der Index-Bereich der beiden Schleifen ist die Vereinigung der Index-Bereiche (grün, $-\cdot$). Jeder ganzzahlige Punkt ist ein Index-Vektor des Schleifenrumpfes und wird durch ein Kreuz dargestellt.

Ein Paar Index-Vektoren $\langle i_1,\ldots,i_{ld}\rangle_1$ , $\langle j_1,\ldots,j_{ld}\rangle$ ist lexikographisch sortiert, $\langle i_1,\ldots,i_{ld}\rangle \prec \langle j_1,\ldots,j_{ld}\rangle$, genau dann wenn

$$\exists k, 1 \leq k \leq ld : i_1 = j_1,\ldots,i_{k-1} = j_{k-1} \text{ und } i_k < j_k.$$

Die Relationen $\succ$, $\preceq$ und $\succeq$ werden analog mit Hilfe von $>$, $\leq$ und $\geq$ gebildet. Ein Index-Vektor $\vec{i}$ ist lexikographisch positiv, wenn gilt $\vec{i} \succeq 0$.

Alle Zugriffsfunktionen von Arrays, die im Schleifennest vorkommen, sollen affine Funktionen der Index-Variablen sein. Zwischen lesenden und schreibenden Zugriffen auf ein Array innerhalb einer Iteration und zwischen verschiedenen Iterationen können Datenabhängigkeiten existieren.

Da die Reihenfolge von Lesezugriffen für dir Korrektheit des Programms unerheblich ist, sind nur Paare von Zugriffen $I_1$, $I_2$ auf dieselbe Speicherzelle relevant, bei denen mindestens ein Schreibvorgang beteiligt ist. Es existieren drei mögliche Abhängigkeiten, die jeweils eine Relation bilden:

**Echte Abhängigkeit:** Die Speicherposition wird zuerst geschrieben und dann gelesen.
$\delta^f$ (**f**low-dependency)

**Antidatenabhängigkeit:** Die Speicherposition wird zuerst gelesen und dann geschrieben.
$\delta^a$ (**a**nti-dependency)

**Ausgabeabhängigkeit:** Die Speicherposition wird hintereinander geschrieben.
$\delta^o$ (**o**utput-dependency)

Wenn eine der Operationen in umgekehrter Reihenfolge auf einer Speicherzelle ausgeführt wird, muss das Ergebnis nicht mit der ursprünglichen Reihenfolge übereinstimmen.

Da die Abhängigkeiten nur auf dieselbe Speicherzelle Bezug nehmen, ist es egal, wann andere Zugriffe auf andere Speicherzellen erfolgen, sofern die für die anderen Speicherzellen geltenden Abhängigkeiten eingehalten werden.

**Definition 2.4.7: Datenabhängigkeit in einem Schleifennest** *Eine Datenabhängigkeit existiert in einer Schleife zwischen zwei Statements $S_1$ und $S_2$ in einem gemeinsamen Schleifennest genau dann, wenn es im Nest zwei Index-Vektoren $\vec{i}$ und $\vec{j}$ mit folgenden Eigenschaften gibt [AK02]:*

1. $\vec{i} \preceq \vec{j}$.
2. *Es existiert im Schleifenrumpf ein Pfad von $S_1$ nach $S_2$.*
3. *$S_1$ greift auf eine Speicherzelle M in der Iteration $\vec{i}$ und $S_2$ greift auf eine Speicherzelle M in der Iteration $\vec{j}$ zu und*
4. *einer der beiden Zugriffe aus (3) ist eine Schreiboperation.*

Eine Transformation wie Tiling ist dann legal, wenn alle Datenabhängigkeiten eingehalten werden.

Für die theoretische Betrachtung kann angenommen werden, dass Arrays immer eindimensional sind, da die affinen Zugriffsfunktionen durch den Compiler auf affine eindimensionale Zugriffsfunktionen geeignet abgebildet werden.

Durch Distanzvektoren lassen sich die Abhängigkeiten in einem Nest beschreiben.

**Definition 2.4.8:** *Existiert eine Datenabhängigkeit zwischen Statement $S_1$ in Iteration $\vec{i}$ nach $S_2$ in Iteration $\vec{j}$ einer Schleife, dann ist der* **Distanzvektor** *$\vec{d}(i,j)$ definiert als der Vektor $\vec{d}(i,j) = \vec{j} - \vec{i}$ [AK02].*

Ein Distanzvektor ist lexikographisch positiv, wenn das erste von Null verschiedene Element positiv ist.

Eine unimodale Matrix ist eine ganzzahlige quadratische Matrix, deren Determinante $-1$ oder 1 ist. Viele Schleifentransformationen lassen sich als unimodale Matrix repräsentieren. Durch eine unimodale Schleifentransformation wird der Iterationsraum der Schleife mit der korrespondierenden unimodalen Matrix multipliziert.

Das Vertauschen von zwei Schleifen kann durch die Matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ dargestellt werden. Für Punkte im Iterationsraum gilt:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} p_2 \\ p_1 \end{pmatrix}$$

Wolf [Wol92] hat folgenden Satz bewiesen: Ist eine unimodale Schleifentransformation durch die unimodale Matrix $U$ gegeben und ist die Menge der Distanzvektoren des Schleifennestes $D$, dann ist die Transformation genau dann legal, wenn gilt

$$\forall \vec{d} \in D : U\vec{d} \succeq 0.$$

Mit anderen Worten, die unimodale Transformation darf keine negativen Distanzvektoren erzeugen, also die Abhängigkeiten im Schleifennest nicht vertauschen.

## 2.5 Lösung von Optimierungsproblemen

Bei einem Optimierungsproblem wird ein globales Extremum einer Zielfunktion gesucht. Die Parameter für das Optimierungsproblem in dieser Arbeit sind jeweils entweder binär oder ganzzahlig. Die Parameter sind außerdem miteinander verknüpft, so dass die Parameter nicht einzeln optimiert werden können und eine gerichtete Suche nicht aussichtsreich scheint. Dies wird dadurch unterstützt, dass schon Teilprobleme, wie die Belegung von Scratchpad-Speichern mit Basisblöcken, NP-vollständig sind.

Zur Lösung des Optimierungsproblems wurden daher zwei Lösungsmöglichkeiten in Betracht gezogen. Durch die Formulierung als lineares ganzzahliges Optimierungsproblem ist eine Lösung mit einem ILP-Solver möglich. Nichtlineare Probleme können mit einem genetischen Algorithmus gelöst werden.

### 2.5.1 Ganzzahlige lineare Optimierung

Ein ganzzahliges lineares Optimierungsproblem besteht aus einer Zielfunktion und einer Reihe von Nebenbedingungen an die Variablen des Optimierungsproblems.

Die lineare Zielfunktion $f$ hat folgende Form:

$$\begin{aligned} a_0 + a_1 \cdot x_1 + a_2 \cdot x_2 + \cdots + a_n \cdot x_n &= f(x) \\ i \in 0 \ldots n : a_i &\in \mathbb{Q} \\ i \in 1 \ldots n : x_i &\in \mathbb{Z} \end{aligned}$$

Die lineare Zielfunktion kann maximiert oder minimiert werden.

Die Nebenbedingungen haben eine ähnliche Form wie die Zielfunktion, nur handelt es sich um Ungleichungen:

$$a_1 \cdot x_1 + a_2 \cdot x_2 + \cdots + a_n \cdot x_n \leq a_0$$

Die Lösung des Gleichungssystems eines ganzzahligen linearen Optimierungsproblems ist NP-vollständig. Es existieren mehrere Lösungsmöglichkeiten. Häufig werden Branch-and-Bound, Dynamische Programmierung oder das Simplex-Verfahren angewendet. Das Simplex-Verfahren ist eine Lösungsmöglichkeit, die für viele praktische Probleme ausreichend schnell eine optimale Lösung findet.

Jede Belegung von $x_i$ ist ein Punkt im $\mathbb{Z}^n$. Durch die Nebenbedingen wird ein Polyhedron $P \subseteq \mathbb{R}^n$ definert, da die Nebenbedingungen affine Funktionen sind. Nach dem Eckensatz liegt das Minimum der linearen Zielfunktion $f$ auf einer Ecke des Polyhedron. Das Simplex-Verfahren wandert über Abstiegskanten von Ecke zu Ecke und findet das Minimum, falls das Problem lösbar ist. Dann ist das Polyhedron nicht leer. Die nächste Ecke wird in der Richtung gewählt, in der die Zielfunktion das größte Gefälle besitzt. Dies geschieht, indem die Umgebung der Ecke in ein neues Koordinatensystem transformiert wird und die transformierte Funktion abgeleitet wird [Dan51, Sch86].

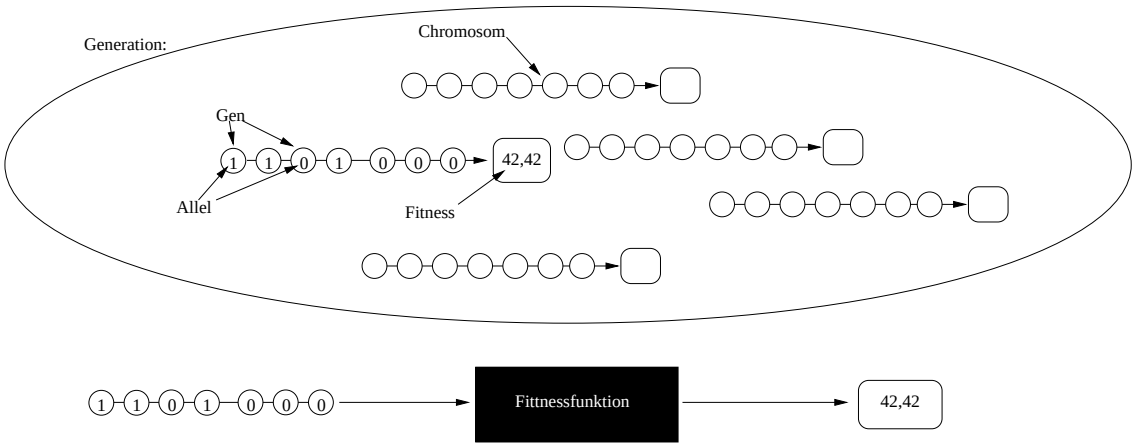


Abbildung 2.5: Die Komponenten eines genetischen Optimierungsproblems

### 2.5.2 Genetische Algorithmen

Genetische Algorithmen bilden die Evolution der Natur nach. Dabei sind die eingesetzten Konzepte der genetischen Rekombination von Bakterien am ähnlichsten.

Bakterien sind Prokaryoten und besitzen keinen echten Zellkern. Es findet keine Meiose oder Befruchtung statt. Bakterien vermehren sich durch einfache Zellteilung. Dabei wird die DNA des Bakterienchromosoms kopiert. Es entstehen zwei identische neue Individuen. Während des Kopiervorgangs des Chromosoms können Fehler auftreten. Ist die Kopie nicht exakt, dann hat eine *Mutation* stattgefunden. 1956 wurde von Lederberg und Tantum nachgewiesen, dass sich Eigenschaften von einem auf den anderen Stamm übertragen konnten. Durch eine Plasmabrücke wird der geöffnete DNA-Einzelstrang einer Hfr-Zelle (High frequency of recombination) in eine Empfängerzelle ($F^-$) übertragen. Teile des Chromosoms der Spenderzelle werden durch Stückaustausch in die Empfängerzelle übetragen; es hat genetische Rekombination stattgefunden. [HH95]

„Survival of the fittest“ ist die Auslese der Natur, die die Bakterien als eine der ältesten Lebensformen der Erde überlebt haben.

Ein genetischer Algorithmus ist eine probabilistische Methode, um Funktionen, deren Eigenschaften sich nicht mit anderen Mitteln der Optimierung ausnutzen lassen, trotzdem zu optimieren [Bäc96].

Eine Generation besteht aus einer Anzahl Chromosomen. Jedes Chromosom besteht aus Genen. Der Wert eines Gens ist ein Allel. Allele können als Bits repräsentiert werden, es können aber auch andere Kodierungen gewählt werden. Zu jedem Chromosom kann die Fitnessfunktion einen reellwertigen Fitnesswert berechnen, indem die Allele des Chromosoms geeignet interpretiert werden. Abbildung 2.5 stellt die Elemente graphisch dar.

Ein genetischer Algorithmus besteht zusätzlich aus einer Vorschrift, wie eine neue Generation gebildet werden kann. In der Regel ist die Anzahl der Chromosomen konstant. Sie werden nach ihrem Fitnesswert sortiert und ein gewisser Teil wird in die nächste Generation kopiert. Der restliche Teil wird durch Mutation oder Rekombination der Individuen aus dem ersten Teil aufgefüllt. Bei der Mutation werden einzelne Allele zufällig negiert. Bei der Rekombination oder Crossover werden Teile der Allele aus dem einen oder anderen Eltern-Chromosom zufällig kopiert.

Es existieren drei verbreitete Möglichkeiten, Crossover zu implementieren:

**Uniformes Crossover** Hier wird für jedes Gen zufällig entschieden, ob es aus dem einen oder anderen Elter übernommen werden soll.

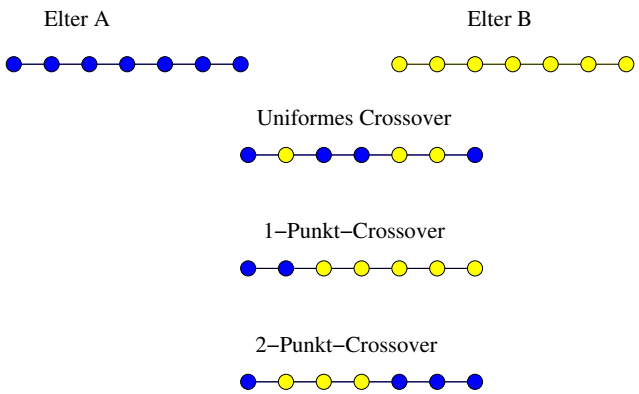


Abbildung 2.6: Crossover auf Chromosomen

**1-Punkt-Crossover** Es wird ein zufälliger Schnittpunkt gewählt, bis dahin werden alle Gene des ersten Elter übernommen, die restlichen Gene werden vom anderen Elter übernommen.

**2-Punkt-Crossover** Es werden zwei zufällige Schnittpunkte gewählt. Bis zum ersten Schnittpunkt werden alle Gene vom ersten Elter übernommen. Bis zum zweiten Schnittpunkt werden alle Gene vom zweiten Elter übernommen. Danach stammen alle Gene wieder vom ersten Elter.

In Abbildung 2.6 sind die drei Varianten des Crossover graphisch dargestellt.

Wie genau die Rekombination definiert ist, kann Gegenstand einer Optimierung des Algorithmus sein. Wenn bekannt ist, dass einzelne Allele als binär kodierte Zahlen interpretiert werden, kann es sinnvoll sein, keinen Schnittpunkt in diesen Bereich zu legen.

Es kann keine Garantie abgegeben werden, dass ein genetischer Algorithmus immer die optimale Lösung findet.

# Kapitel 3

# Verwandte Arbeiten

Zu dieser Arbeit existieren eine Reihe verwandter Arbeiten. Nutzungsmöglichkeiten von Scratchpads sind vielfältig und auch Tiling wurde mit verschiedenen Zielen eingesetzt.

## 3.1 Tiling

Tiling kann die Lokalität der Zugriffe auf Daten in einem Schleifennest optimieren. Schon 1969 wurde von McKaller und Coffman beschrieben, wie Tiling eingesetzt werden kann, um Matrix-Operationen auf Systemen mit Seitenadressierung zu optimieren [MEGC69]. Durch die Anordnung und Auswahl von Matrix-Operationen wird die Anzahl der Seitenfehler (page fault) reduziert, wenn nicht die gesamte Matrix in den zur Verfügung stehenden Speicher passt. Es werden die Operationen von ganzen Matrizen auf Teilmatrizen zurückgeführt. Die Programmtransformation ist nicht automatisiert. Die Erkenntnisse richten sich an Programmierer und Nutzer von Bibliotheken, die ihre Programme selbst entsprechend den Vorschlägen transformieren sollen. Auf die vorliegende Arbeit übertragen, entsprechen die Seitenfehler der Anzahl Kopiervorgänge, die während der Ausführung notwendig sind. Die Matrix-Multiplikation wird in fast allen Arbeiten betrachtet, die sich mit Tiling beschäftigen. Der Algorithmus ist einfach und die Schleifen lassen sich beliebig vertauschen.

Mit dem verbreiteten Einsatz von Caches wird Tiling zur Performance-Steigerung im Supercomputing eingesetzt. Wolf und Lam haben die Repräsentation von Schleifentransformationen als Multiplikation unimodaler Matrizen [LW91, Wol92] geprägt. Mathematisch wird die Lokalität und Wiederverwendung von Array-Zugriffen repräsentiert. Die Theorie der unimodalen Transformation fasst verschiedene Schleifentransformationen, wie Schleifenvertauschen, Schleifenumdrehen und Schleifenverschränken zusammen. Insbesondere wird auch darauf eingegangen, wie die Parameter zur optimalen Nutzung von Caches gewählt werden sollten. Es wird ein Algorithmus vorgestellt, der eine profitable Schleifenanordnung durch Suche der legalen Lösungen findet. Dabei wird ein Teilnest erzeugt, dass vollständig permutierbar ist, so dass Tiling angewendet werden kann. Optimierungsziel ist es, die Wiederverwendung innerhalb des Nestes zu erhöhen. Die Kosten oder Nebenwirkungen der Transformation werden nicht betrachtet.

Ein breites Forschungsfeld ist dabei die Analyse der Abhängigkeiten [Ban88, LW91, PW92] und das Erlauben von nicht perfekten Schleifennestern [LLL01]. Affine Partitionierung ist der universellste Ansatz, um die Abhängigkeiten im Programm einzuhalten, und wird daher näher erläutert.

Mit Hilfe von affiner Partitionierung [LLL01] kann auch ein beliebig geschachteltes Schleifennest von Tiling profitieren.

*Bei affiner Paritionierung wird eine Iterationsinstanz durch den Wert der Schleifenindizes der umgebenden Schleifen identifiziert. Eine affine Paritionierung besteht aus einer Abbildung zwischen den Indexvariablen im Original und den Werten der Indexvariablen im transformierten Programm. Anweisungen, die*

*gemeinsame äußere Schleifen besitzen, teilen sich die gleichen Indexvariablen.* Eine Kette von abhängigen Instruktionen bildet einen Thread. Zentrale Bedingung für die Korrektheit der Umordnung ist, dass innerhalb eines Threads die paarweise Reihenfolge der Befehle nicht vertauscht wird. Eine Erkenntnis ist, dass die Befehle zweier verschiedener Threads ineinander verschränkt werden können. Bestehen nun die Threads aus Schleifen, dann können auch die Schleifen verschränkt ausgeführt werden.

Es existieren zwei Sorten der Partitionierung. *n* unabhängige Lösungen der Partitionierung im Raum entsprechen *n* parallelen äußeren Schleifen. *n* unabhängige Lösungen der Partitionierung in der Zeit entsprechen *n* vollständig vertauschbaren äußeren Schleifen. Diese Schleifen können getiled werden. In [LL97, LCL99] wird gezeigt, wie die Partitionierung in Zeit und Raum formalisiert wird und anschließend linearisiert werden kann.

Dieser Ansatz kann zwar beliebige Programme transformieren, die Komplexität des generierten Codes geht jedoch nicht in die Optimierung ein. Außerdem haben die bisherigen Arbeiten nur die Ausführungszeit als Kriterium der Optimierung verwendet.

Zur effizienteren Nutzung von Scratchpad-Speichern wurde Tiling schon von [KRI+04] eingesetzt, auf diese Arbeit wird gegen Ende dieses Kapitels genauer eingegangen.

## 3.2 Scratchpad-Speicher

Die Optimierungsmöglichkeiten bei der Nutzung von Scratchpad-Speichern sind vielfältig. Einige Dimensionen des Designraumes sind in Abbildung 3.1 dargestellt. Die wesentlichen Optimierungsmöglichkeiten sind:

**Speichertechnologie** Es existieren viele unterschiedliche Speichertechnologien, wie SRAM, DRAM oder FLASH. In dieser Dimension werden unterschiedliche Architekturen betrachtet; auch Caches oder DMA-Einheiten können in die Betrachtungen zur Energieoptimierung einbezogen werden.

**Programm/Daten** Der Hauptspeicher wird mit Programmen und Daten belegt. Scratchpad-Speicher werden in manchen Arbeiten nur mit Code, nur mit Daten oder mit beidem belegt. Die Betrachtung von Daten und Programmen bietet die Sicherheit, eine umfassendere Analyse zu betreiben, während die exklusive Betrachtung von Daten oder Programmen das Verfahren wesentlich vereinfachen kann.

**Dynamische/Statische Allokation** Die Belegung des Scratchpad-Speichers kann statisch oder dynamisch erfolgen. Bei der statischen Belegung wird eine feste Belegung während der Ausführung beibehalten. Bei der dynamischen Belegung verändert sich die Belegung und Aufteilung während der Ausführung des Programms. Der Vorteil ist, dass dadurch auf lokale Besonderheiten eingegangen werden kann, wenn z. B. mehrere Schleifennester um den Speicher konkurrieren. Bei der dynamischen Allokation wird der Scratchpad-Speicher zur Laufzeit des Programms belegt.

**Programmoptimierungen** Bei der Energie optimierenden Übersetzung können zusätzliche Programmoptimierungen betrachtet werden, die einen Einfluss auf den Energieverbrauch haben.

### 3.2.1 Statische Belegung

Im Folgenden werden entlang der historischen Entwicklung der Nutzung von Scratchpad-Speichern wesentliche Arbeiten vorgestellt. Zuerst wurde der Scratchpad-Speicher nur mit Daten, insbesondere mit kompletten Arrays belegt [PDN97].

Die Arbeit von Panda [PDN97] präsentiert ein effizientes Verfahren, das Daten und Arrays vom DRAM-Hauptspeicher in den SRAM-Scratchpad-Speicher verschiebt, um die Gesamtlaufzeit zu verringern. Es wird angenommen, dass auf den DRAM durch einen Cache zugegriffen wird. Cache-Konflikte sollen verringert werden. Das Verfahren selektiert die Arrays und skalaren Variablen für den Scratchpad-Speicher,

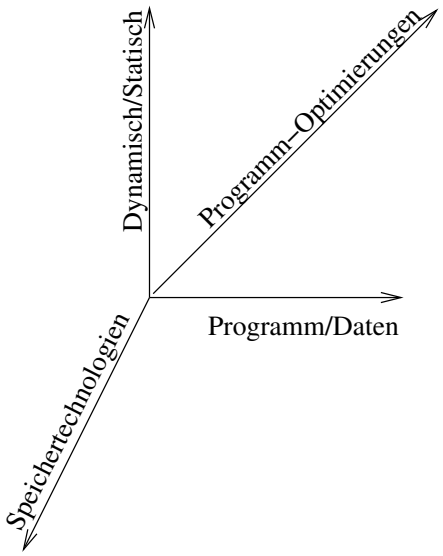


Abbildung 3.1: Möglichkeiten der Nutzung von Scratchpad-Speichern

die für die meisten Cache-Konflikte verantwortlich gemacht werden. Skalare Variablen werden in den Scratchpad-Speicher verschoben. Arrays, die zu groß sind, werden im Hauptspeicher belassen; nur die restlichen Arrays werden betrachtet. Diejenigen Arrays, die die meisten Konflikte verursachen, werden als erstes in den Scratchpad-Speicher verschoben.

Durch das Energiemodell von Steinke [SKWM01], das im Abschnitt 2.3 eingeordnet wird, ist es möglich, durch optimierte Übersetzung Energie zu sparen. Ein Beispiel für die Anwendung des Modells ist eine Energie optimierte Register-Pipeline [SSWM01], die aber neben der Verwendung des Energiemodells wenig mit der vorliegenden Arbeit gemeinsam hat.

In [SZWM01] wird der Scratchpad-Speicher statisch mit Basis-Blöcken, Funktionen und globalen Objekten belegt. Das Rucksackproblem wird mittels Branch-and-Bound gelöst. Die Information über die Ausführungshäufigkeit wird mit statischer Analyse gewonnen.

Die Formalisierung wurde in [SWLM02] wieder verwendet und ein IP Solver zur Lösung verwendet, der auch in vielen weiteren Arbeiten genutzt wird. Der Vergleich mit einer Cache-Architektur zeigt bei gleicher Speichergröße Einsparungen im Energieverbrauch zwischen 12% und 43%.

Eine Verallgemeinerung des Ansatzes auf mehrere Scratchpad-Speicher wurde in [WHM04, Hel04] vorgenommen. Im ganzzahligen linearen Optimierungsproblem wird der Energieverbrauch der Basisblöcke und globaler Variablen modelliert. Die Objekte werden auf mehrere Scratchpad-Speicher verteilt. Die wesentliche Idee ist, die Kanten des Kontrollflussgraphen zu verwenden, um die zusätzlichen Kosten bei notwendigen langen Sprüngen zwischen Basisblöcken zu modellieren. Es werden nicht alle möglichen Belegungen aufgezählt, so dass die Laufzeit des ILP-Solver akzeptabel bleibt.

Durch den Einsatz mehrerer kleiner Scratchpad-Speicher kann auch bei großen insgesamt verfügbarem Scratchpad-Speicher die Verbesserung des Energieverbrauchs auf einem hohem Niveau gehalten werden.

Ein Problem ist die Verwendung großer Arrays, die nur am Stück in den Scratchpad-Speicher verlegt werden können. Falk und Verma [FV04] ziehen auch einen Teil eines Arrays zu Verlagerung in den Scratchpad-Speicher in Betracht. Es wird schrittweise vorgegangen. Zuerst wird der Scratchpad-Speicher wie in den vorherigen Arbeiten gefüllt. Unter den verbleibenden Arrays wird ein Kandidat ausgewählt, der den restlichen Scratchpad-Speicher belegen soll. Dieses Array wird in zwei Teile zerlegt, so dass ein Teil in den Rest des Scratchpad-Speichers passt, der andere Teil verbleibt im Hauptspeicher. Für alle Zugriffe muss

nun eine Fallunterscheidung eingefügt werden, auf welchen Teil zugegriffen wird. Die Fallunterscheidung, ob ein Element des Arrays im Scratchpad-Speicher ist oder nicht, wird durch Verwendung von Loop Nest Splitting optimiert. Es wird geprüft, ob trotz des Overheads und der Optimierung keine Energie eingespart wird, dann wird das ausgewählte Array nicht geteilt und mit dem Nächstbesten derselbe Schritt wiederholt. Schließlich wird der Code entsprechend den Entscheidungen transformiert.

Das Ergebnis ist ähnlich einem getilten Schleifennest, bei dem nur zwei Iterationen in der Tiling-Schleife durchgeführt werden. Tiling bietet bei der Wahl der Größe der Arrays wesentlich mehr Flexibilität. Die Heuristik, das Scratchpad mit dem bekannten Verfahren [SWLM02] zuerst zu füllen und den Rest möglichst effizient zu nutzen, erleichtert die Umsetzung und Analyse. Die Ergebnisse dieser Arbeit zeigen aber, dass nicht pauschal Code bevorzugt werden muss. Tabelle 5.7 auf Seite 107 zeigt, dass bei kleinen Scratchpad-Speichern ein wesentlicher Code-Teil im Hauptspeicher verbleibt. Wenn noch ein relevanter Teil des Scratchpad-Speichers leer bleibt, ist der gesamte Code schon im Scratchpad-Speicher und wird bevorzugt. Die Analyse der Bedingungen des Loop Nest Splitting ist ähnlich mächtig wie affine Partitionierung. Auch dieses Verfahren belegt den Scratchpad-Speicher nur statisch.

Loop Nest Splitting ist eine eigenständige vielseitige Optimierung, die auch in anderen Benchmarks angewendet werden kann. Es wird ein regulärer Kontrollfluss in inneren Schleifen von Datenfluss dominierten Applikationen erzeugt, indem die Ausführungshäufigkeiten von Bedingungen minimiert werden. Eine weitere Anwendung ist beispielsweise in der MPEG-Codierung gegeben [Fal04].

### 3.2.2 Dynamische Belegung

Die dynamische Belegung von Scratchpad-Speichern erfordert, dass der Compiler Instruktionen generieren muss, die zur Laufzeit den Scratchpad-Speicher mit anderen Instruktionen oder Daten füllen und modifizierte aber noch benötigte Daten in den Hauptspeicher zurückkopieren.

In der Arbeit von Grunwald, Steinke et al. [Gru02, SGW+02] wird betrachtet, wie Basisblöcke dynamisch in den Scratchpad-Speicher verlagert werden können. Mögliche Kopierpositionen liegen vor den Schleifen des Programms. Es wird nicht der absolute Energieverbrauch modelliert, sondern der erzielbare Gewinn. Dadurch ist dieses Verfahren nicht sofort auf eine Architektur mit mehr als zwei Speichern erweiterbar. Das Optimierungsproblem wird ebenfalls mit einem ILP-Solver gelöst. Es wird dynamisches Profiling zur Analyse des Programms verwendet.

Mit Hilfe des dynamischen Kopierens von Basisblöcken kann bis zu 30% Energie gespart werden. Gegenüber dem statischen Ansatz kann besonders bei Programmen mit mehreren Schleifen eine Verbesserung des Energieverbrauchs bis zu 38% beobachtet werden. Die Geschwindigkeit wird mit 25% etwas weniger stark als die Energie verbessert.

Mit einem anderen Verfahren, das erweiterte Def-Use-Chains nutzt, ist Verma [VWM04b] in der Lage, Speicherobjekte dynamisch in den Scratchpad-Speicher zu verschieben. Speicherobjekte sind skalare und nicht skalare globale Variablen, nicht skalare lokale Variablen und Code-Segmente, sog. Traces. Das Verfahren zum dynamischen Belegen des Scratchpad-Speichers ist eine Erweiterung der globalen Registerbelegung. An jedes Speicherobjekt der Kante des Kontrollflussgraphen werden vier Attribute (DEF, MOD, USE, CONT) annotiert, mit deren Hilfe die Lebensdauer und die Konflikte im Grafen verfolgt werden können. Zusätzliche Attribute Load und Store modellieren das Kopieren der Speicherobjekte. Aus dem Graphen werden Gleichungen und Bedingungen gewonnen. Durch die Lösung des ganzzahligen Optimierungsproblems wird errechnet, welche Objekte wann zwischen den Speichern kopiert werden und den Scratchpad-Speicher belegen.

Ein weiteres Problem ist die Berechnung der Anordnung der Objekte im Scratchpad-Speicher. Dieses Problem ist NP-vollständig und wird ebenfalls durch ein ganzzahliges lineares Optimierungsproblem sowie durch eine Heuristik gelöst.

Dieses Verfahren erzielt eine Verringerung des Energieverbrauchs zwischen 34% und 38%. Die Größe des Programms nimmt nur leicht zu. Nachteile dieses Verfahrens liegen darin, dass im Ablauf drei Übersetzungen notwendig sind und dynamisches Profiling verwendet wird. Auch dieses Verfahren ist nicht in der

Lage, Teile von Arrays in den Scratchpad-Speicher zu verschieben. Es kann aber mit dem in dieser Arbeit beschriebenen Verfahren nach Anpassungen kombiniert werden Dies wird im Ausblick Abschnitt 6.2 auf Seite 126 weiter ausgeführt.

### 3.2.3 Andere Optimierungen

Nachdem die historische Entwicklung der Scratchpad-Speichernutzung nachvollzogen wurde, werden die anderen Design-Dimensionen betrachtet. Eine andere Dimension ist die Berücksichtigung unterschiedlicher Hardware. Verma [VWM04a] stellt ein Verfahren zur statischen Belegung von Scratchpad-Speichern vor, das Cache-Konflikte berücksichtigt. Der Energieverbrauch bei einem Zugriff auf den Scratchpad-Speicher und den Hauptspeicher wird modelliert. Hauptspeicherzugriffe unterscheiden sich, je nachdem ob ein Hit oder Miss vorliegt.

In den Scratchpad-Speicher können skalare Daten, Arrays und sog. Traces verschoben werden. Ein Trace ist eine Abfolge von Befehlen, die mit einem unbedingten langen Sprung endet. Wird innerhalb eines Traces verzweigt, wird in ein Feld mit langen Sprüngen im Anschluss des Traces gesprungen. Da ein Trace immer nur mit einem langen Sprung verlassen wird, können Traces beliebig auf Hauptspeicher und Scratchpad-Speicher verteilt werden. Der entstehende Overhead wird in Kauf genommen, da die Optimierung so wesentlich vereinfacht wird.

Das Verfahren arbeitet mit drei Übersetzungen und zwei Anwendungen von dynamischem Profiling. Im ersten Schritt wird das Originalprogramm ausgeführt und beobachtet, welche Teile des Codes zu Traces zusammengefasst werden können. Im zweiten Schritt wird das Programm, das nun nur noch aus Traces besteht, auf das Cacheverhalten dynamischem analysiert. Die Traces wurden dabei mit NOPs so aufgefüllt und angeordnet, dass sie auf einer Cache-Line beginnen. Mit Hilfe der gewonnenen Informationen über das Cacheverhalten wird dann das Optimierungsproblem formuliert. Das Problem wird mit einem ILP-Solver gelöst und das Programm anschließend zum dritten und letzten Mal übersetzt. Traces im Scratchpad-Speicher werden nicht mit NOPs aufgefüllt, da durch die Nutzung des Scratchpad-Speichers das Cacheverhalten nicht geändert wird und die NOPs dort nicht mehr gebraucht werden. Mit diesem Verfahren können 8-29% Verbesserung erzielt werden.

Eine weitere Möglichkeit der Nutzung anderer Hardware ist die Nutzung von DMA-Einheiten, um Speicherblöcke dynamisch zu kopieren. Poletti [FMA+04] zeigt, wie durch die Nutzung von DMA-Einheiten und der damit einhergehenden Hardwareveränderung Scratchpad-Speicher dynamisch verwendet werden können. Es werden Funktionen eingeführt, die eine explizite Speicherverwaltung ermöglichen, so dass dynamisch Speicher im Scratchpad belegt werden kann. Es treten dabei die typischen Probleme der Speicherfragmentierung auf, je nachdem wie der Speicher angefordert wird. Mit Funktionen zur Steuerung der DMA-Einheit sollen Teile von größeren Datenstrukturen zur Laufzeit kopiert werden. Die Matrix-Multiplikation wird erwähnt. Die Auswahl, welche Objekte in den Scratchpad-Speicher verschoben werden sollen, wird greedy anhand des größten Nutzens entschieden.

Wie dieser Nutzen berechnet wird, geht nicht aus der Arbeit hervor. Die Größe der verwendeten Datenstrukturen soll erst zur Laufzeit festgelegt werden, so dass sich der relative Nutzen der Objekte zur Laufzeit ändern kann. Eine Automatisierung des Austausches der Daten wird nicht vorgeschlagen. Beim Tilen der Matrix-Multiplikation wird nicht deutlich, wie die Tiling-Faktoren gewählt werden.

Ebenfalls ein dynamischer Ansatz wird in [UB03] verwendet, um globale Daten und den Stack in den Scratchpad-Speicher zu verschieben. Kopierpunkte werden vor Schleifen in Betracht gezogen, die Auswahl geschieht über ein Kostenmodell. Der Stack von rekursiven Funktionen wird nicht in den Scratchpad-Speicher verschoben. Zentrale Datenstruktur ist ein Data Program Relationship Graph. Dies ist ein Funktionsaufrufgraph, der zusätzlich die Schleifen von Funktionen als Knoten enthält. Entlang der Breitensuche wird für jeden Knoten berechnet, welche Variablen zwischen den Speichern kopiert werden sollten. Entsprechend einer Heuristik, die auf der Entfernung und der Zugriffshäufigkeit beruht, werden die Variablen zum Austausch in Betracht gezogen. Kopiert wird aber nur dann, wenn dem Modell nach der Nutzen die Kosten übersteigt.

Für Matrix-Multiplikation mit 273 KByte Daten wird eine Laufzeitverbesserung um 40% angegeben. Der Scratchpad-Speicher ist 68 KByte groß. Code wird durch das beschriebene Verfahren nicht in den Scratchpad-Speicher verlegt.

### 3.2.4 Kopieren von Teil-Arrays und Speicherhierarchien

Bis auf die Arbeit von Verma [FV04] betrachten alle bisher erwähnten Arbeiten Arrays immer als ganze Objekte. Wenn Arrays nicht als Ganzes in den Scratchpad-Speicher verlegt und dynamisch kopiert werden müssen, bietet dies mehr Flexibilität.

Die Arbeiten von van Achteren [ADCL02] und Issenin [IBMD04] nutzen Arrays mit komplexen Zugriffsmustern aus. Die Kopierkosten werden optimiert. Ein Array, auf das nicht immer gleichmäßig zugegriffen wird, muss nicht komplett kopiert werden. Die Position des Kopierpunktes beeinflusst, welcher Teil kopiert wird und somit die Kopierkosten. Das Konzept der Copy Candidates wird eingeführt. Ein Copy Candidate bestimmt, wann und wieviel von einem Array kopiert wird.

Bei genauerer Betrachtung der Eingabeprogramme fällt auf, dass diese Arbeiten schon getilten Code als Eingabe verwenden und die Kopierposition und die zu kopierenden Bereiche im Nachhinein errechnet werden. Dies bietet mehr Flexibilität, wenn nicht auf alle Objekte eines Arrays gleichmäßig zugegriffen wird. Die Möglichkeit, die Position der Kopierfunktion durch die getiled verwendeten Index-Variablen zu bestimmen, ist durch diese Arbeiten inspiriert. Da die Eingabeprogramme für diese Arbeiten schon getiled sind, wird die Größe der zu kopierenden Arrays nur in großen Schritten gewählt und kann nicht weiter angepasst werden.

In [BMCC03] wird das Konzept der Copy Candidates ebenfalls angewendet, um eine Speicherhierachie aus mehreren Ebenen effizient zu belegen. Die Heuristik verwendet das Verhältnis aus Größe und Häufigkeit der Zugriffe, um zu entscheiden, welche Arrays sich lohnend in kostengünstige Speicher verschoben werden können. Mehre Speicherebenen werden auch in den folgenden Arbeiten betrachtet.

Die Arbeiten von Kandemir [KRI+04, KC02] beschreiben ein Verfahren, das Tiling und Scratchpad-Speicher einsetzt, um Energie zu sparen. In der ersten Arbeit [KC02] wird anhand der Reusevektoren ermittelt, in welchen Schleifen Array-Elemente wieder verwendet werden. Anhand der gefunden Schleife wird ermittelt, wie groß der zu verwendende Speicher sein sollte. Die maximale Anzahl von Null verschiedener Elemente über alle Reusevektoren liefert die vorgeschlagene Anzahl Speicherebenen. Tiling wird verwendet, um bei gegebener Größe des Scratchpad-Speichers die Größe der Arbeitsbereiche anzupassen. In der jüngeren Arbeit [KRI+04] wird dieses Verfahren weiter verfeinert, indem auch die Schleifenanordnung systematisch gewählt wird.

Das Kostenmodell ist sehr einfach. Es werden nur Kopierkosten betrachtet, die aus einem konstanten Initialisierungsteil und einem zur Länge des zu kopierenden Arrays proportionalen Teil bestehen. Dieses Kostenmodell wird verwendet, um die Heuristik zu begründen. Das Kostenmodell wird bei der Abwägung der Programmtransformation selbst nicht eingesetzt.

Die betrachtete Architektur verwendet außerdem einen Cache, so dass die Kopier- und Zugriffskosten von örtlicher Lokalität profitieren. Die Möglichkeit, Code in den Scratchpad-Speicher zu verschieben, wird nicht betrachtet. Bei einer zweistufigen Hierachie werden Einsparungen zwischen 9% und 35% im Energieverbrauch ausgewiesen [KC02]. In dieser Veröffentlichung wird die Matrix-Multiplikation nicht evaluiert. In [KRI+04] werden die Zugriffskosten und Anzahl der Datentransfers betrachtet, so dass ein Vergleich mit anderen Arbeiten, die eine Reduktion des Energieverbrauchs betrachtet haben, nicht angebracht ist.

Diese Arbeiten lieferten die Idee, Tiling zur Belegung des Scratchpad-Speichers zu verwenden. Auf weitere Optimierung der Schleifenanordnung wurde verzichtet, da die Matrix-Multiplikation keine signifikanten Veränderungen bei einer Änderung der Schleifenanordnung zeigt. Dies gilt nicht für eine Architektur mit Cache oder einem Compiler, der schleifeninvariante Ausdrücke optimiert.

Der encc ist ebenfalls in der Lage, durch die Arbeit von [WHM04, Hel04] mehrere partitionierte Speicher statisch zu verwenden. Die Schlüsselidee aus dieser Arbeit wurde übernommen, nämlich die gesamte Energie zu modellieren und nicht nur die Verbesserung.

# Kapitel 4

# Energieeffiziente Scratchpad-Belegung durch Loop-Tiling

Dieser Teil der Arbeit beschäftigt sich damit, wie Scratchpad-Speicher durch Loop-Tiling energieeffizient mit Code und Arrays belegt werden können. Zuerst wird näher erläutert, wie Tiling für Scratchpad-Speicher funktioniert und welche Auswirkungen auf den Code existieren. Die Aufgabe und die zu analysierenden Programme werden anschließend weiter eingegrenzt, so dass Programmanalyse, Vorhersage des Overheads, die Programmtransformation und die tatsächlichen Veränderungen zusammenpassen.

Passend zur Aufgabe werden die eingesetzten Programme vorgestellt und auf Besonderheiten eingegangen. Die vorhandenen Programme ermöglichen verschiedene Vorgehensweisen; deren Vor- und Nachteile werden gegeneinander abgewogen. Der eingeschlagene Weg wird genauer beschrieben. Im Rest des Kapitels werden die drei Phasen der Problemlösung beschrieben. Das zu übersetzende Programm muss analysiert werden. Aus der Analyse wird das Optimierungsproblem erzeugt. Das Optimierungsproblem wird zunächst nur formal eingeführt. Die Lösungsmöglichkeiten werden dargestellt. Ist das Optimierungsproblem gelöst, kann schließlich das Programm transformiert werden.

Im nächsten Kapitel wird das Verfahren mit der vorhandenen statischen Belegung des Scratchpad-Speichers verglichen.

## 4.1 Tiling für Scratchpad-Speicher

Tiling wird eingesetzt, um die Lokalität von Array-Zugriffen in Schleifen zu erhöhen. Es wird örtliche und zeitliche Lokalität unterschieden. Eine Lokalität ist zeitlich, wenn auf dasselbe Element in einer späteren Iteration erneut zugegriffen wird. Eine Lokalität ist örtlich, wenn auf ein benachbartes Element in einer Iteration zugegriffen wird.

Um die Auswirkungen von Tiling genauer zu betrachten, wird zunächst angenommen, dass für ein System mit einem Cache optimiert wird. Auf die speziellen Eigenheiten der Nutzung von Scratchpad-Speichern wird im Folgenden eingegangen.

Tiling ist die Kombination aus Schleifenvertauschen und Strip-Mining. Das Vertauschen von Schleifen erhöht die Lokalität und Strip-Mining sorgt dafür, dass der begrenzte Platz ausreicht.

### 4.1.1 Schleifenvertauschen und Lokalität

Zunächst soll das Schleifenvertauschen näher betrachtet werden. Bei der Nutzung eines Caches wird ein Block aus dem Hauptspeicher beim ersten Zugriff in den Cache kopiert. Ein Block enthält meist mehr

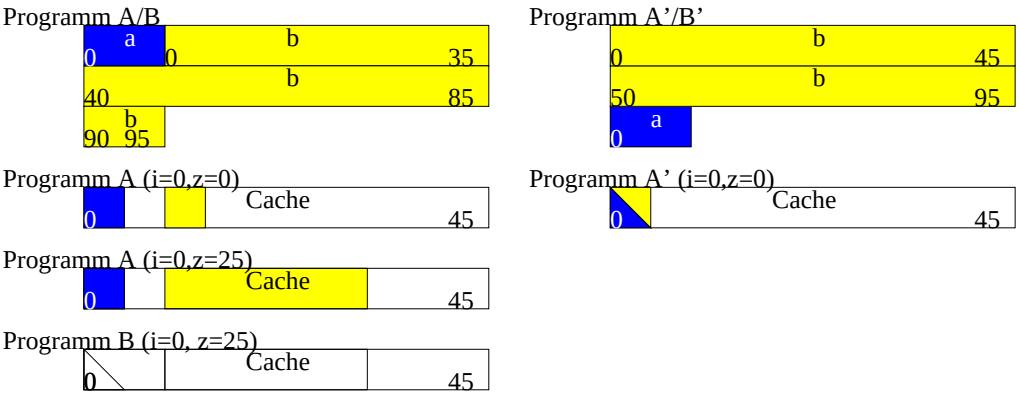


Abbildung 4.1: Speicheranordnung und Cache-Belegung

als ein Wort. Wenn der nächste Zugriff auf den gleichen Block stattfindet, liegt örtliche Lokalität vor, die vom Cache ausgenutzt wird, da der Block schon im Cache ist. Die Schleifen sollten so sortiert sein, dass möglichst selten Cache-Zeilen ausgetauscht werden müssen.

**Beispiel 4.1.1: Gewinn durch Schleifenvertauschen**

Ein direct-mapped Cache habe eine Größe von 50 Elementen und eine Blockgröße und Cache-Linegröße von 5. Das Array `b` ist 100 Elemente groß und liegt direkt hinter dem 10 Elemente großen Array `a`.

Das Programm A behandelt in der äußeren Schleife die Iteration über das `a`-Array:

```
for (i=0; i<10; i++)
  for (z=0; z<100; z++)
    a[i] = a[i] + b[z];
```

Das Programm B behandelt in der äußeren Schleife die Iteration über das `b`-Array:

```
for (z=0; z<100; z++)
  for (i=0; i<10; i++)
    a[i] = a[i] + b[z];
```

Beide Programme berechnen dasselbe Ergebnis, die Vertauschung der beiden Schleifen ist also legal.

Mit einem Cache-Simulator kann man einfach die Anzahl der Hits und Misses bestimmen.

Einen weiteren Einfluss auf die Miss- und Hit-Rate hat die Anordnung der Elemente im Speicher. In den Programmen A' und B' wurde die Speicheranordnung der beiden Arrays vertauscht. Die Ergebnisse sind in Tabelle 4.1 tabellarisch dargestellt.

Das Vertauschen der Schleifen hat die Miss-Rate im Programm B im Vergleich zu A um 78% gesenkt und den Schleifen-Overhead um 9% erhöht. Die Speicheranordnung hat nur einen geringen Einfluss, das Programm A' hat im Vergleich zu A 9% mehr Misses. In Abbildung 4.1 ist die Speicherbelegung mit den beiden Anordnungsvarianten links und rechts oben dargestellt. Es ist besser, zuerst das Array `a` (links) im Speicher abzulegen

Programm Version B hat eine bessere Cache-Performance, weil das `a`-Array in der innersten Schleife im Cache gehalten wird, während es in der Version A häufiger verdrängt wird. Der Schleifen-Overhead ($lt$) verhält sich genau entgegen gesetzt. In Programmvariante A wird die

| | A | B | A' | B' |
|---|---|---|---|---|
| Misses | 1730 | 383 | 1890 | 384 |
| Hits | 1270 | 2617 | 1110 | 2616 |
| Looptests | 1010 | 1100 | 1010 | 1100 |

Tabelle 4.1: Einfluss von Schleifenvertauschen auf Hits, Misses und die Anzahl der Schleifentests der Programme A und B sowie der Speicheranordnung bei den Programmen A' und B'.

äußere Schleife zehnmal und die innere Schleife $10 \cdot 100 = 1000$ Mal ausgeführt. Für den Schleifen-Overhead der Schleifen gilt:

$$\begin{aligned} lt_A &= 10 + 10 \cdot 100 = 1010 \\ lt_B &= 100 + 10 \cdot 100 = 1100 \end{aligned}$$

Das Beispiel zeigt, dass die Schleifenanordnung und die Speicheranordnung wesentliche Einflussfaktoren sind. Es wird auch deutlich, dass eine Optimierung Auswirkungen auf mehrere Messgrößen hat: Miss-/Hit-Rate und Schleifen-Overhead. Es muss in jedem Fall abgewogen werden, welcher Gewinn überwiegt, um eine globale Verbesserung zu erhalten.

### 4.1.2 Einfluss von Tiling auf Cache-Misses

Durch Tiling können die Cache-Misses weiter reduziert werden. Dazu wurde das Beispielprogramm aus Beispiel 4.1.1 in allen Varianten getiled und wieder die Anzahl der Hits und Misses sowie der Häufigkeit der Schleifentests bestimmt. Es existieren vier Möglichkeiten, das Nest zu tilen. Die Reihenfolge der Tiling-Schleifen ist gleich der Reihenfolge der Originalschleifen. Die Varianten A und B stammen aus den Möglichkeiten der Anordnung der Schleifen im Originalnest. Die Varianten A" und B" haben die Elementschleifen vertauscht. Es wird zuerst das kleinere Array im Speicher abgelegt, da dies die bessere Anordnung ist. Knapper Sieger mit 302 Misses ist die Anordnung, die direkt aus Programm B hervorgegangen ist, die oben schon die beste Anordnung der Schleifen bot:

```
for (zt=0; zt<100; zt+=50) {
  for (it=0; it<10; it+=5) {
    for (z=zt; z<zt+zT; z++) {
      for (i=it; i<min(10,it+iT); i++) {
        A[i] = A[i] + B[z];
      }
    }
  }
}
```

Alle Varianten können mit mindestens einer Wahl der Tiling-Faktoren 303 Misses erreichen. Eine Übersicht über die sieben besten Kandidaten ist in Tabelle 4.2 gegeben. Alle Varianten sind in der Lage, die Missrate zu senken. Die Häufigkeit des Schleifentests liegt in einer Schwankungsbreite von 5% und wird durch die Schleifenanordnung und zum größten Teil durch den Tiling-Faktor bestimmt. Als Tiling-Faktoren wurden nur Zahlen gewählt, die 5 als Teiler haben. Dies ist die gewählte Blockgröße und ist auch ein Teiler der Ausführungshäufigkeit der Originalschleifen. Für die obere Schranke kann im Experiment aber nicht auf die Minimumbildung verzichtet werden, da auch Tiling-Faktoren gewählt werden sollen, die größer gleich der Ausführungshäufigkeit der Originalschleifen sind. So kann annähernd der Fall betrachtet werden, in dem eine Schleife nicht getiled wird.

| Platz | | zT | iT | Miss | Hit | Looptest |
|---|---|---|---|---|---|---|
| 1 | Variante B | 50 | 5 | 302 | 2698 | 1206 |
| 2 | Variante B | 45 | 5 | 303 | 2697 | 1209 |
| 3 | Variante A | 5 | 10 | 303 | 2697 | 1221 |
| 4 | Variante B | 15 | 5 | 303 | 2697 | 1221 |
| 5 | Variante B” | 5 | 10 | 303 | 2697 | 1240 |
| 6 | Variante B | 5 | 5 | 303 | 2697 | 1260 |
| 7 | Variante B” | 5 | 5 | 303 | 2697 | 1260 |
| 8 | Variante B | 35 | 5 | 304 | 2696 | 1209 |

Tabelle 4.2: Die sieben besten Möglichkeiten, das Schleifennest zu tilen

### 4.1.3 Nutzung von Scratchpad-Speichern statt Caches

Caches sind für den Nutzer sehr komfortabel, da sie automatisch Blöcke zwischen dem Cache und dem Hauptspeicher austauschen. Wenn mehrmals auf die gleiche Cache-Zeile zugegriffen wird und die Daten schon vorhanden sind, kann die örtliche und zeitliche Lokalität ausgenutzt werden. Der Cache hat die Daten schon in die Cache-Zeile kopiert und es findet pro Miss nur ein Zugriff auf den Hauptspeicher statt. Ist der Arbeitsbereich einer Schleife so klein, dass alle Daten in den Cache passen, finden während der Ausführung dieser Schleife höchstens beim ersten Zugriff Cache-Misses statt.

Scratchpad-Speicher besitzen keine Cache-Logik und müssen daher von der CPU explizit beschrieben werden. Der Compiler kann Arbeitsbereiche in den Scratchpad-Speicher verschieben und innerhalb einer Schleife wird auf den Scratchpad-Speicher zugegriffen. Dazu müssen die Speicherzugriffe modifiziert und Kopieroperationen eingefügt werden. Eine Kopieroperation muss vor einer Schleife mit Zugriffen stattfinden, wenn der Speicher gelesen wird und nach der Ausführung der Schleife muss der Bereich zurückkopiert werden.

Tiling für Scratchpad-Speicher besteht aus folgenden vier Komponenten.

- Tilen des Schleifennestes,
- Einfügen kleiner Teil-Arrays,
- Modifizieren des Index-Bereichs der Elementschleifen,
- Einfügen von Kopieraufrufen.

Diese vier Schritte sollen am Beispiel der Matrix-Multiplikation nachvollzogen und erläutert werden. Das Originalprogramm hat folgende Form:

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

`N` ist die Anzahl der Elemente in jeder Dimension und eine Konstante. Der erste Schritt ist das Tilen des Nestes, wie es im Abschnitt 2.4.2 beschrieben wird. Die Anwendung von Tiling ist für dieses Programm legal, da alle Distanzvektoren $\vec{0}$ sind. Jede unimodale Matrix $U$ bildet den Nullvektor auf den Nullvektor ab $U\vec{0} = \vec{0}$. Es bleiben alle Distanzvektoren positiv. Für jede Schleife muss ein Tiling-Faktor gewählt werden, hier sind es die Tiling-Faktoren `iT`, `jT`, `kT`.

```
//Tiling-Schleifen
for (it=0; it<N; it+=iT) {
  im=min(it+iT,N);
  for (jt=0; jt<N; jt+=jT) {
     jm=min(jt+jT,N);
     for (kt=0; kt<N; kt+=kT) {
       km=min(kt+kT,N);
       //Elementschleifen:
       for (i=it; i<im; i++) {
         for (j=jt; j<jm; j++) {
           for (k=kt; k<km; k++) {
             C[i][j] = C[i][j] + A[i][k] * B[k][j];
           }
         }
       }
    }
  }
}
```

Für Scratchpad-Speicher ist dieses Programm noch nicht vorteilhafter, da der Scratchpad-Speicher nicht verwendet wird. Damit der Scratchpad-Speicher verwendet werden kann, wird zunächst die Größe der Arbeitsbereiche berechnet.

Wie groß der Arbeitsbereich ist, auf den in den einzelnen Arrays zugegriffen wird, hängt von den gewählten Tiling-Faktoren ab. Würde $iT = jT = kT = 5$ gewählt und ist 5 ein Teiler von $N$, dann wird in den Elementschleifen auf zweidimensionale Arrays zugegriffen, die in jeder Dimension fünf Elemente haben. Dies soll als Arbeitsbereich der ersten Elementschleife aufgefasst werden. Die Arbeitsbereiche der Arrays werden als kleinere globale Arrays deklariert. Die Arrays müssen so groß sein, dass jeder mögliche Arbeitsbereich während der Ausführung in das Array hineinpasst.

Im Beispiel werden folgende Arrays global definiert:

```
//Teil-Arrays:
As[iT][kT];
Bs[kT][jT];
Cs[iT][jT];
```

Exemplarisch soll betrachtet werden, warum `jT` die Größe der ersten Dimension von `As` sein soll. Die Zugriffsfunktion der ersten Dimension von `A` ist `j`. Es muss also der Arbeitsbereich von `j` in der ersten Elementschleife betrachtet werden.

Die größten und kleinsten möglichen Werte der Kontrollvariablen der Elementschleife müssen bekannt sein:

$$\begin{aligned} j_{min} &= jt \\ j_{max} &= jm - 1 \\ &= min(jt + jT, N) - 1 \\ &= jt + jT - 1, \text{ mit } jT \leq N \end{aligned}$$

In Abschnitt 4.6.4.3 wird beschrieben, wie der Arbeitsbereich eines Arrays aus der Zugriffsfunktion und dem kleinsten und größten Wert der vorkommenden Kontrollvariablen berechnet werden kann. Am Bei-

spiel ergibt sich Folgendes:

$$
\begin{aligned}
ws('j') &= 1 + j_{max} - j_{min} && (4.1)\\
&= 1 + j_{max} - j_{min} && (4.2)\\
&= 1 + jt + jT - 1 - jt && (4.3)\\
&= jT && (4.4)
\end{aligned}
$$

Im nächsten Schritt sollen die Index-Bereiche der Elementschleifen angepasst werden. Aus der Berechnung des Arbeitsbereiches wird in Zeile 4.3 deutlich, dass die untere Grenze der Schleife nichts Wesentliches beiträgt. Es sollen daher alle Elementschleifen so transformiert werden, dass sie bei 0 beginnen. Die Elementschleifen werden normalisiert, indem von der unteren und der oberen Schranke einer jeden Elementschleife die untere Schranke der Elementschleife abgezogen werden. Dies kann während der Bildung des Minimums geschehen, dann braucht nur noch die untere Schranke einer Elementschleife bei 0 zu beginnen.

Das Programm sieht dann wie folgt aus:

```
//Teil-Arrays:
As[iT][kT];
Bs[kT][jT];
Cs[iT][jT];
//Tiling-Schleifen
for (it=0; it<N; it+=iT) {
  im=min(it+iT,N)-it;
  for (jt=0; jt<N; jt+=jT) {
     jm=min(jt+jT,N)-jt;
     for (kt=0; kt<N; kt+=kT) {
       km=min(kt+kT,N)-kt;
       //Elementschleifen:
       for (i=0; i<im; i++) {
         for (j=0; j<jm; j++) {
           for (k=0; k<km; k++) {
             C[it+i][jt+j] = C[it+i][jt+j] +
                             A[it+i][kt+k] * B[kt+k][jt+j];
           }
         }
       }
    }
  }
}
```

Bei der Index-Bereichstransformation der Elementvariablen müssen die Zugriffe auf die Original-Arrays angepasst werden, indem die Index-Bereichstransformation rückgängig gemacht wird. Es wird jedes Vorkommen der Index-Variablen einer Elementschleife durch die Summe aus den Index-Variablen der Tiling-Schleifen und der Elementschleifen substituiert.

Es kann für jedes Vorkommen einer Index-Variablen gewählt werden, ob sie im Original oder getiled verwendet werden soll. In diesem Beispiel sollen aber alle Index-Variablen am Ende getiled verwendet werden.

Im folgenden Schritt werden die kleinen Arrays auch benutzt. Die Vorkommen der Original-Arrays werden durch die korrespondierenden kleinen Arrays ersetzt. Die eingefügten Kopieraufrufe verbinden die kleinen Arrays mit den großen Arrays.

Die Position, ab der im Original-Array kopiert wird, bestimmen die korrespondierenden Index-Variablen der Tiling-Schleifen. Wie viel in jeder Dimension kopiert wird, hängt vom aktuellen Arbeitsbereich in der Iteration ab. Das Programm hat dann folgende Gestalt:

```
//Teil-Arrays:
As[iT][kT];
Bs[kT][jT];
Cs[iT][jT];

//Tiling-Schleifen
for (it=0; it<N; it+=iT) {
  im=min(it+iT,N)-it;
  for (jt=0; jt<N; jt+=jT) {
     jm=min(jt+jT,N)-jm;
     for (kt=0; kt<N; kt+=kT) {
       km=min(kt+kT,N)-km;
       //Elementschleifen:
       copy(&As[0][0],&A[it][kt],im,km,kT,N);
       copy(&Bs[0][0],&B[kt][jt],km,jm,jT,N);
       copy(&Cs[0][0],&C[it][jt],im,jm,jT,N);
       for (i=0; i<im; i++) {
         for (j=0; j<jm; j++) {
           for (k=0; k<km; k++) {
             Cs[i][j] = Cs[i][j] + As[i][k] * Bs[k][j];
           }
         }
       }
       copy(&C[it][jt],&Cs[0][0],im,jm,jT,N);
    }
  }
}
```

Die Kopierfunktion hängt von der Dimension der zu kopierenden Arrays ab. Zuerst wird die Adresse des Ziel- und Quell-Arrays übergeben. Für die erste Dimension muss nur angeben werden, wieviele Elemente kopiert werden sollen, da die Elemente direkt hintereinander im Speicher abgelegt sind. Für jede weitere Dimension muss auch angegeben werden, wieviele Elemente (hier Zeilen) kopiert werden sollen. Zusätzlich wird für das Quell- und Ziel-Array eine Information benötigt, wieviele Elemente in dieser Dimension übersprungen werden müssen, damit auf das nächste Element zugegriffen werden kann.

Die Position zum Kopieren kann weiter optimiert werden, indem möglichst weit außen kopiert wird. Das Kopieren des Arrays `C` kann schon nach der zweiten Tiling-Schleife stattfinden, weil `iT` oder `im` nicht im Aufruf vorkommen.

Lesende Zugriffe auf ein Array machen eine Kopieroperation vor den Elementschleifen notwendig, schreibende Zugriffe machen ein Kopieren nach Ausführung der Elementschleifen notwendig. Die Kopierposition innerhalb der Tiling-Schleifen wird durch die verwendeten Indizes in der Zugriffsfunktion bestimmt.

Wenn ein Array während der Ausführung einer Schleife in den Scratchpad-Speicher verschoben wird, dann soll es vor der Ausführung der Elementschleifen kopiert werden. Ein Kopieren innerhalb der Elementschleifen hätte zur Folge, dass ein evtl. kleinerer Bereich kopiert werden müsste, in der innersten Elementschleife ist dies z. B. in der Regel ein einzelnes Element. Das Kopieren in einer Elementschleife hätte aber auch zur Folge, dass die Kopieroperation wesentlich häufiger ausgeführt werden müsste. Die Größe des zu kopierenden Arbeitsbereichs kann durch die Tiling-Faktoren ausreichend genau bestimmt werden. Durch die Möglichkeit, dass ein Index im Original verwendet werden soll, kann ein größerer Bereich zum Kopieren ausgewählt werden.

Die Position, wo genau kopiert werden soll, hängt davon ab, welche Tiling-Index-Variablen in der Zugriffsfunktion getiled verwendet werden. Wird ein Index getiled verwendet, dann kann das Array nur innerhalb der Tiling-Schleife kopiert werden. Der letzte getiled verwendete Index in den Zugriffsfunktionen eines Array bestimmt die Kopierposition. Es bestimmt immer nur eine Entscheidung über die Verwendung des

Original-Indexes die Kopierposition. Wird kein Index getiled verwendet, dann wird das Array vor dem Schleifennest kopiert, sofern es in den Scratchpad-Speicher verschoben werden soll.

Jede Entscheidung, ob eine Index-Variable getiled oder im Original verwendet wird, beeinflusst immer die Größe des zu kopierenden Teil-Arrays. Je weiter außen kopiert wird, desto größer ist der Arbeitsbereich. Werden alle Indizes ungetiled verwendet, kann auch das gesamte Array in den Scratchpad-Speicher verschoben werden. Die folgenden Schleifen können dasselbe Array verwenden, wenn sie ebenfalls alle Indizes ungetiled verwenden.

### 4.1.4 Overhead des Tiling für Scratchpad-Speicher

In dieser Arbeit wird der Energiebrauch minimiert. Da Tiling Overhead in das Programm einfügt, muss der Overhead bekannt sein. Im Programm kann folgender Overhead auftreten:

**Programmgröße:** Tiling verdoppelt die Anzahl der Schleifen. Kopierfunktionen und Aufrufe müssen kodiert werden. Die Größe des Programms nimmt zu. Der Energieverbrauch eines Programms hängt auch davon ab, welche Programmteile in den Scratchpad-Speicher verlegt werden können. Wenn das Programm größer wird, ist dies eher ein kleinerer Teil als im Originalprogramm.

**Kopieroperationen:** Der Scratchpad-Speicher muss explizit durch die CPU belegt werden, dazu werden zusätzliche Befehle ausgeführt.

**Ausführungshäufigkeiten:** Die Ausführungshäufigkeit von Code-Teilen verändert sich. Im Originalprogramm wird die Initialisierung der äußersten Elementschleife nur einmal ausgeführt. Im getilten Programm wird sie so oft ausgeführt, wie der Rumpf der innersten Tiling-Schleife.

**Zusätzliche Schleife:** Die Tiling-Schleifen sind zusätzliche Schleifen, die Ausführungszeit und Speicherplatz benötigen.

**Minimums-Bildung:** Je nach gewähltem Tiling-Faktor kann die Bildung des Minimums notwendig sein.

**Registerdruck:** Der Registerdruck kann durch Tiling erheblich zunehmen. Jede Tiling-Schleife hat eine Induktionsvariable, die während der Ausführung der Elementschleifen lebendig ist. Wenn eine Bildung des Minimums notwendig ist, ist auch das Ergebnis während der Ausführung der Elementschleifen lebendig.

**Zugriffsfunktionen:** Wenn ein Original-Index verwendet werden soll oder muss, wird im getilten Nest eine Addition durchgeführt, um diesen zu berechnen.

**Code-Generierung:** Die Code-Generierung kann durch Tiling verändert werden. Je nachdem wie Konstanten geladen werden können, wird unter Umständen anderer Code generiert.

Die Auswirkungen der Modifizierung sollen im Folgenden an einem realen Beispiel betrachtet werden. Um möglichst viele Effekte zu erkennen, wurde für den Benchmark Matrix-Multiplikation mit 15 Elementen der Tiling-Faktor 6 für alle Schleifen gewählt und das Programm mit dem encc übersetzt. Das modifizierte Programm besteht aus 213 Befehlen, das Originalprogramm dagegen nur aus 33 Befehlen. Die Anzahl der Befehle hat sich durch die Code-Modifikation mehr als versechsfacht. Die Veränderung der Code-Größe sollte schon aus diesem Grund möglichst genau nachvollzogen werden. Es bietet sich an, die Anzahl der zusätzlichen Befehle genauer aufzuschlüsseln.

Die Kopierkosten teilen sich in drei Gruppen. Die Initialisierung (COPY_INIT) schließt den Aufruf im Programm mit ein und umfasst im Beispiel 22 Befehle. Die Kopierschleifen selbst werden mit COPY_LOOP und COPY_BODY repräsentiert. Die Anzahl der Befehle, die bei den anderen Code-Modifikationen eingefügt wurden, sind in Tabelle 4.3 dargestellt. Jede Klasse wird in der Tabelle nur einmal mit ihrer Größe aufgeführt, unabhängig davon, wie oft sie im Programm auftritt. LOOPINIT ist die Schleifeninitialisierung. Die Befehle aus LOOPCONT bilden die Schleifenkontrolle. Bei der Bildung des Minimums existieren zwei

| Code-Fragment | # Befehle |
|---|---|
| COPY_INIT | 22 |
| COPY_LOOP | 12 |
| COPY_BODY | 7 |
| LOOPINIT | 1 |
| LOOPCONT | 3 |
| MIN | 6 |
| SPILL | 1 |

Tabelle 4.3: Übersicht über die Größe betrachteter Code-Fragmente

Code-Pfade mit einer unterschiedlichen Anzahl Befehlen. MIN repräsentiert hier den häufigeren Weg, der zwei Befehle weniger enthält. Instruktionen, die auf Spilling zurückzuführen sind, werden getrennt gezählt.

Im Originalprogramm musste kein Register gespillt werden. Im modifizierten Programm hingegen wurden 6 Register, die Tiling-Indizes und die oberen Grenzen der Elementschleifen gespillt. Insgesamt wurden 43 Instruktionen zum Spillen von Registern eingefügt.

Der Fall, dass ein Original-Index verwendet wird, wurde in diesem Beispiel nicht betrachtet, es wird aber nur eine zusätzliche Instruktion benötigt, wenn sowohl die Element-Indexvariable als auch die Tiling-Indexvariable nicht gespillt sind.

Veränderungen in der Code-Generierung, die durch eine andere Code-Selektion entstehen, werden nicht berücksichtigt.

Durch Auszählen kann berechnet werden, wie oft die verschiedenen Code-Fragmente ausgeführt werden und welches Gewicht im Energieverbrauch zu erwarten ist. Dazu wurde das Programm nicht ausgeführt. Anhand der Kostenklassen wird statisch die Höhe der anfallenden Kosten berechnet. In diesem Abschnitt werden nur Befehle betrachtet und von den tatsächlichen Kosten abstrahiert. Im implementierten Modell und der Formalisierung werden die Kosten für einzelne Befehle der Klassen nach dem Energiemodell berechnet. Die Kostenklassen des Modells werden in Abschnitt 4.6.4 eingeführt. Das Ergebnis ist in Abbildungen 4.2 und 4.3 dargestellt. Abbildung 4.2 zeigt, in welchem Code-Teil wieviele zusätzliche Befehle ausgeführt werden. Durch das Kopieren werden die meisten Befehle ausgeführt (18504$\hat{=}$70%). In der innersten Schleife ist ausschließlich Spilling für den Overhead verantwortlich. Das Spilling in der innersten Schleife hat aber noch 3375 (12%) Befehlsausführungen verursacht. Die nicht innersten Elementschleifen werden häufiger ausgeführt, dies hat bei der mittleren Elementschleife dazu geführt, dass 3051 zusätzliche Befehle ausgeführt werden. Spilling ist hier auch aufgetreten.

Die durch die Tiling-Schleifen selbst ausgeführten Befehle bilden nur einen geringen Overhead, die häufigere Ausführung der Elementschleifen ist aber signifikant, dies wird auch im folgenden Beispiel deutlich.

**Beispiel 4.1.2: Ausführungshäufigkeit von Schleifen**

> In einem getilten Schleifennest werden nicht innerste Elementschleifen häufiger ausgeführt als im Originalnest. In Tabelle 4.4 wird die Ausführungshäufigkeit in Abhängigkeit verschiedener Tiling-Faktoren dargestellt. Es wird das Nest der Matrix-Multiplikation mit 50 Elementen betrachtet, die Reihenfolge der Indizes ist `j`, `k`, `i`. Die Ausführungshäufigkeit der innersten Elementschleife ist bei allen Tiling-Faktoren konstant. Die Ausführungshäufigkeiten der nicht innersten Elementschleifen hängen von den gewählten Tiling-Faktoren ab. Der funktionale Zusammenhang wird in Abschnitt 4.6.4.2 dargestellt.

Der Energieverbrauch der Kopierfunktion und das Spilling sollten möglichst genau modelliert werden, da dies einen großen Anteil des Overheads ausmacht. Das Spilling in Elementschleifen kann oft durch geschickte Wahl der Tiling-Faktoren vermieden werden.

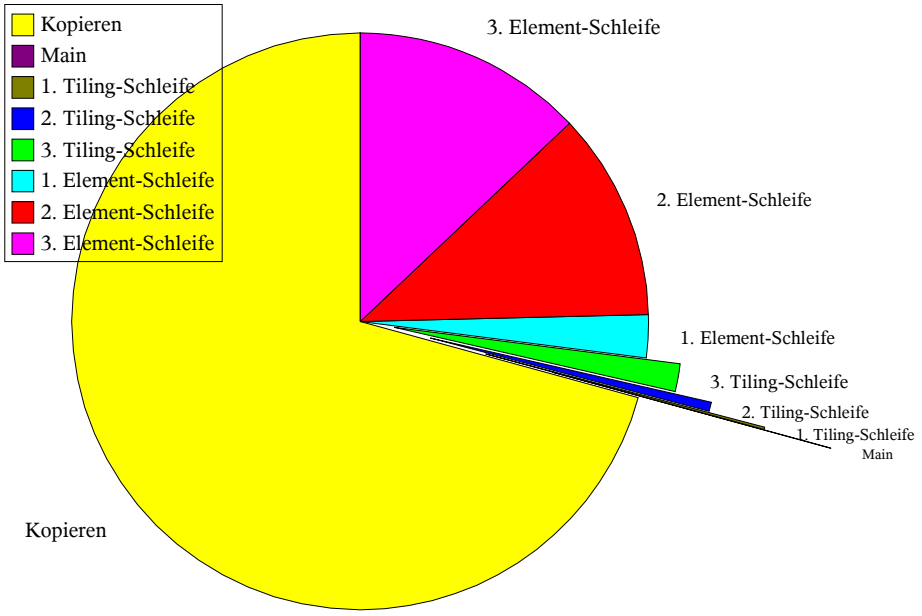


Abbildung 4.2: Zusätzlich ausgeführte Befehle in Code-Abschnitten

Da sich die Daten und das Programm den Scratchpad-Speicher teilen, kann der Speicher-Overhead durch die Tiling-Schleifen nicht vernachlässigt werden, weil dieser Code auch Platz in Anspruch nehmen kann. Der unterschiedliche Energieverbrauch von Befehlen, die aus dem Scratchpad-Speicher ausgeführt werden, sollte auch berücksichtigt werden.

Die Aufteilung nach Programmpositionen wie in Abbildung 4.2 ist für die Modellierung wenig hilfreich, daher wurden die Kostenklassen eingeführt. Abbildung 4.3 zeigt, wie sich die zusätzlich ausgeführten Befehle auf Kosten auf die verschiedenen Klassen verteilen. Dies gibt darüber Aufschluss, welchen Einfluss eine Klasse auf die Gesamtkosten besitzt. Dazu wurden die Anzahl der Betrachtungen einer Kostenklasse mit der Anzahl der Befehle dieser Klasse gewichtet. Dies zeigt, welchen relativen Einfluss eine Klasse auf die Gesamtkosten besitzt.

Die Kopierkosten dominieren auch hier, da sie auch insgesamt einen beträchtlichen Teil der Kosten ausmachen. Die Schleifenkosten bilden zusammen 10% der Information für die Overhead-Berechnung. Die Minimumskosten umfassen auch nach der Gewichtung nur einen Anteil von 0,008%.

Durch den Zugriff auf Scratchpad-Speicher kann Energie eingespart werden. Wenn ein Array durch Tiling in den Scratchpad-Speicher verschoben werden kann, entsteht dadurch ein Gewinn, der größer sein sollte als der Overhead, der durch das Tiling verursacht wird. Nur wenn dies der Fall ist, kann durch Tiling und die Nutzung von Scratchpad-Speichern Energie gespart werden.

| jT | kT | iT | ex(j) | ex(k) | ex(i) |
|---|---|---|---|---|---|
| 50 | 50 | 50 | 50 | 2500 | 125000 |
| 50 | 50 | 25 | 100 | 5000 | 125000 |
| 50 | 50 | 10 | 250 | 12500 | 125000 |
| 50 | 25 | 50 | 100 | 2500 | 125000 |
| 50 | 25 | 25 | 200 | 5000 | 125000 |
| 50 | 25 | 10 | 500 | 12500 | 125000 |
| 50 | 10 | 50 | 250 | 2500 | 125000 |
| 50 | 10 | 25 | 500 | 5000 | 125000 |
| 50 | 10 | 10 | 1250 | 12500 | 125000 |
| 25 | 50 | 50 | 50 | 2500 | 125000 |
| 25 | 50 | 25 | 100 | 5000 | 125000 |
| 25 | 50 | 10 | 250 | 12500 | 125000 |
| 25 | 25 | 50 | 100 | 2500 | 125000 |
| 25 | 25 | 25 | 200 | 5000 | 125000 |
| 25 | 25 | 10 | 500 | 12500 | 125000 |
| 25 | 10 | 50 | 250 | 2500 | 125000 |
| 25 | 10 | 25 | 500 | 5000 | 125000 |
| 25 | 10 | 10 | 1250 | 12500 | 125000 |
| 10 | 50 | 50 | 50 | 2500 | 125000 |
| 10 | 50 | 25 | 100 | 5000 | 125000 |
| 10 | 50 | 10 | 250 | 12500 | 125000 |
| 10 | 25 | 50 | 100 | 2500 | 125000 |
| 10 | 25 | 25 | 200 | 5000 | 125000 |
| 10 | 25 | 10 | 500 | 12500 | 125000 |
| 10 | 10 | 50 | 250 | 2500 | 125000 |
| 10 | 10 | 25 | 500 | 5000 | 125000 |
| 10 | 10 | 10 | 1250 | 12500 | 125000 |

Tabelle 4.4: Ausführungshäufigkeiten der Elementschleifen der getilten Matrix-Multiplikation bei verschiedenen Tiling-Faktoren

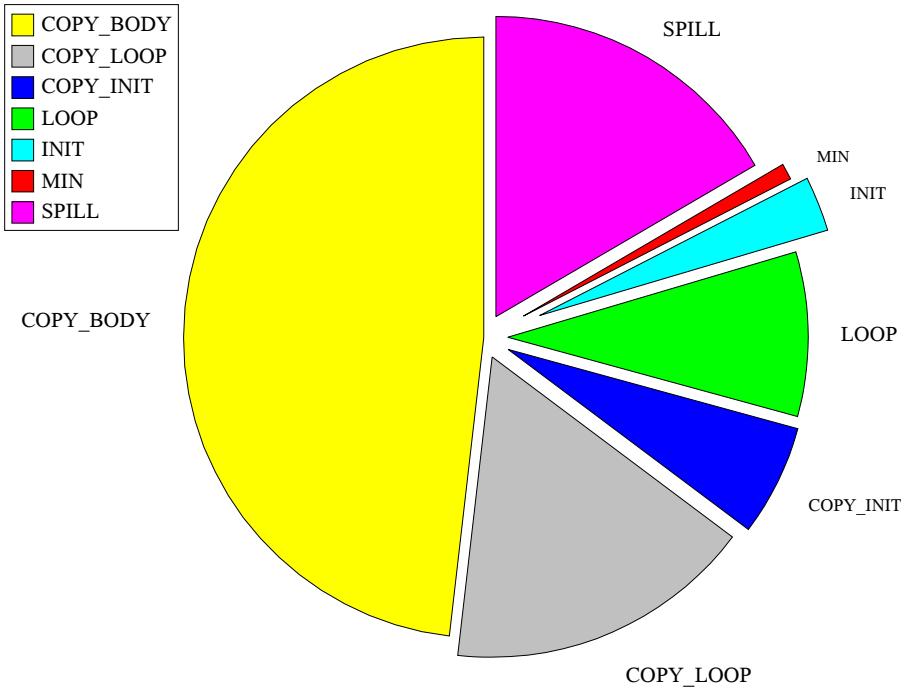


Abbildung 4.3: Gewicht der Kostenklassen an den Gesamtkosten

## 4.2 Eingrenzung der Aufgabe

Das in dieser Arbeit erstellte Programm ist in der Lage, Code und Arrays energiebewusst in den Scratchpad-Speicher zu verlegen. Tilebare Schleifennester werden fürs Tiling in Betracht gezogen und es wird ein passender Tiling-Faktor gewählt. Dazu wird der durch das Tiling und andere Codemodifikationen veränderte Energieverbrauch des Programms modelliert. Die Optimierung verwendet das Modell zur Bewertung der Kosten für die Transformation. Das Programm modifiziert den Code entsprechend dem Ergebnis der Optimierung. Die Codemodifikationen schließen das Tiling der Schleifen, das Modifizieren der Array-Zugriffe und das Einfügen der Kopierfunktionen ein.

Um die Menge zu betrachtender Programme einzuschränken, wird angenommen, dass es sich um einfach geschachtelte Schleifennester handelt. Die oberen und unteren Grenzen der Schleifen sind affine Funktionen der umgebenden Schleifen. Die Differenz zwischen oberer und unterer Grenze einer Schleife ist eine Konstante. Die Schrittweite ist 1. Für Anweisungen, die in nicht innersten Schleifen zusätzlich vorkommen, muss es legal sein, dass diese nach dem Tiling nur zu den ursprünglichen Iterationspunkten ausgeführt werden. Die zu betrachtenden Array-Zugriffe besitzen als Zugriffsfunktion affine Funktionen der Kontrollvariablen. Die Differenz zwischen den Zugriffsfunktionen der Arrays ist eine Konstante. Das zu analysierende Programm enthält in den Schleifen keine Bedingungen neben den Abbruchbedingungen.

Ist ein Benchmark nicht in dieser Form, so ist es meistens möglich, den Benchmark so zu verändern, dass er dennoch optimiert werden kann. Benchmarks, die nicht in diese Form gebracht werden können oder die Schleifennester enthalten, die durch dieses Vorgehen nicht tilebar sind, werden im weiteren Verlauf der Arbeit nicht betrachtet.

Eine Konsequenz, die sich schon aus diesen Einschränkungen ergibt, ist, dass z. B. Implementierungen der schnellen Fourietransformation nicht analysiert werden können. Das verwendete Teile-und-Herrsche-Verfahren führt zu einem dreieckigen Ausführungraum. Wenn die Differenz zwischen oberer und unterer Schranke einer Schleife keine Konstante ist, dann ist das Volumen des Ausführungsraums, also die Ausführungshäufigkeit, nicht mehr einfach zu berechnen.

Soll nicht gefordert werden, dass die Differenz zwischen oberer und unterer Schranke eine Konstante ist, kann eine Bibliothek [WL05] eingesetzt werden, die die Berechnung der Ausführungshäufigkeit auch für diesen Fall erlaubt. Bei der Lösung durch ein ganzzahliges Optimierungsproblem kann die Bibliothek nicht eingesetzt werden.

Auf den Einsatz dieser Bibliothek wurde aus zwei Gründen verzichtet. Die analysierbaren Programme im Kapitel 5 reichen aus, um die Möglichkeiten von Tiling zur Energieoptimierung abzuschätzen. Während der Formalisierung bestand der Wunsch, einen ILP-Solver verwenden zu können. Als klar wurde, dass dies nicht möglich ist, wurde die Formalisierung nicht verworfen, sondern zur Implementierung der Fitnessfunktion verwendet.

Die Optimierung der Schleifenanordnung wird zur Vereinfachung der Optimierung und Transformation nicht betrachtet. Die Arbeiten [Buc04, KRI+04] legen nahe, dass die Schleifenanordnung Einfluss auf das Ergebnis haben. Es wird daher angenommen, dass das Originalprogramm schon in der optimalen Anordnung vorliegt. Für Matrix-Multiplikation mit 14 Elementen hat eine Umordnung der Schleifen des Originalprogramms im getilten Programm keinen Effekt.

Zur Übersetzung des Programms soll der encc verwendet werden, der in Abschnitt 4.3.2 vorgestellt wird. Eine Designentscheidung ist, welches Speicher-Allokationsverfahren der neuen Optimierung zugrunde gelegt werden soll. Aus der Entscheidung für eine Optimierung ergibt sich auch, welche Programme sinnvoll übersetzt werden können.

**Cache Aware Scratchpad Allocation** Dieses Verfahren [VWM04a] ist sehr interessant, weil Caches berücksichtigt werden, die in modernen, auch eingebetteten CPUs vorkommen. Die Auswirkungen von Tiling sollen möglichst genau vorhergesagt werden, die Analyse des Cache-Verhalten ist aber schon ohne Optimierung schwierig.

**Dynamic Scratchpad-Overlay** Dieses Verfahren [VWM04b] ist für große Programme mit vielen Schleifen sehr interessant, die häufig ausgeführt werden. Die Nachteile sind aber, dass das Programm dreimal übersetzt und zweimal einem dynamischen Profiling unterzogen wird.

**Statische Scratchpad-Belegung** Für Programme mit nur einer häufig ausgeführten Schleife liefern Dynamic Scratchpad-Overlay und statische Scratchpad-Belegung das gleiche Ergebnis. Für Programme mit mehreren häufig ausgeführten Schleifen ist dieses Verfahren zwar auch anwendbar, aber nicht so gut geeignet. Es kann die Verwendung von statischer Analyse und dynamischem Profiling unterschieden werden. Die statische Analyse hat den Vorteil, dass sie schneller ist als dynamisches Profiling. Der Nachteil ist, dass die Informationen über das Programm nicht so genau sind.

Um einen Eindruck zu gewinnen, welche Möglichkeiten Tiling bietet, reicht eine Integration mit einer statischen Scratchpad-Belegung aus. Die Verwendung der statischen Scratchpad-Belegung impliziert, dass mehrere häufig ausgeführte Schleifennester nicht unbedingt optimal übersetzt werden können. Eine Integration mit dynamischer Belegung des Scratchpad-Speichers ist in dieser Arbeit nicht vorgesehen.

Um den Aufwand der Integration und der Implementierung gering zu halten, soll in dieser Arbeit nur die statische Belegung des Scratchpad-Speichers betrachtet werden. Da die statische Analyse eine schnellere Übersetzung ermöglicht, soll speziell die statische Analyse bevorzugt verwendet werden.

Im Folgenden wird beschrieben, wie ein Benchmark mit dem in dieser Arbeit entstandenen Programm übersetzt werden kann. Zunächst werden die verwendeten Programme und Bibliotheken kurz vorgestellt und auf einige Besonderheiten eingegangen. Anschließend werden Konzepte verschiedener Arbeitsabläufe vorgestellt und der gewählte Arbeitsablauf näher beschrieben.

## 4.3 Verwendete Programme und Bibliotheken

Die in dieser Arbeit eingesetzten Standard-Programme sind auf einem typischen modern eingerichteten Unix-System zu finden. Das in dieser Arbeit erstellte Programm ist in C++ geschrieben und läßt sich z. B. mit dem gcc 3.3.4 übersetzen. Eine Reihe weiterer Programme, die nicht allgemein verbreitet sind, werden in dieser Arbeit ebenfalls verwendet und im Folgenden vorgestellt.

### 4.3.1 ICD-C

ICD-C [ICD05b, ICD05a] ist eine C++ Bibliothek, die C99- oder C98-Dateien einlesen kann. Sie wurde am Informatik Centrum Dortmund entwickelt und wird kommerziell vertrieben. Da gegen diese Bibliothek statisch gelinkt wird, kann das in dieser Arbeit erstellte Programm nicht unter einer üblichen Open-Source Lizenz wie der GPL [FSF91] veröffentlicht werden. Auf den Quelltext der Bibliothek konnte zur Erstellung dieser Arbeit nicht zugegriffen werden.

ICD-C baut eine wohlstrukturierte objektorientierte Repräsentation der C-Datei auf, auf die durch geeignete Methoden zugegriffen werden kann. Es handelt sich dabei um den Syntaxbaum der C-Datei und um Symboltabellen. Es existieren Methoden zur Code-Analyse, die durch Datenflussgraphen, Kontrollflussgraphen und Def-Use-Chains unterstützt werden.

Es wird das Konzept der Übersetzung durch Code-Transformation und Baumüberdeckung propagiert. Attribute können an Knoten in Form von Pragma-Listen annotiert werden.

Ein besonderes Merkmal von ICD-C ist, dass keine Strukturinformation mutwillig entfernt wird. Es findet kein sog. „lowering“ statt. Wurde die interne Repräsentation nicht verändert, so kann das eingelesene Programm in derselben Struktur wieder herausgeschrieben werden, so dass sich ICD-C insbesondere zur Modifikation von Quelltexten eignet. Der generierte Code kann nach C99 oder C89 generiert werden, dies betrifft insbesondere die Ausgabe von for-Schleifen, Compounds, Initialisierungslisten und die Deklaration von Variablen.

Diese Bibliothek wurde im Vergleich zu anderen wie SUIF [Gro05] vorgezogen, weil die Objektorientierung vorbildlich umgesetzt wurde und die STL [Bre96] verwendet wird. Außerdem ist das Benutzerinterface sehr übersichtlich und leicht verständlich.

Es existieren aber auch Nachteile. SUIF hätte die Programmanalyse und Transformation wesentlich vereinfacht. In SUIF ist Tiling schon als Compiler-Pass vorhanden. Aufgrund dieser Tatsache wurde auf eine genaue Abhängigkeitsanalyse verzichtet und stattdessen ein Pragma („NOTILE“) definiert, das angibt, dass ein Nest nicht tilebar ist.

ICD bietet nur die Möglichkeit, Attribute in Form von Pragma-Strings an die Datenstruktur zu heften. Um die für diese Arbeit wesentlichen Attribute vorzuhalten und für einen effizienten Zugriff bereitzuhalten, wird eine eigene Datenstruktur verwendet, die Verweise auf bestimmte Objekte der IR-Struktur besitzt und die Attribute speichert. Diese Struktur stellt Methoden bereit, die repräsentierten Teile der IR-Struktur zu modifizieren. Dies wird in Abschnitt 4.5 näher beschrieben.

### 4.3.2 encc

Der Energy Aware C Compiler (encc) wurde am Lehrstuhl Informatik XII der Universität Dortmund entwickelt [SW05]. Er besteht aus folgenden Komponenten:

- einem LANCE2 Frontend [Leu04]
- einem Backend für den 16 Bit THUMB-Mode des ARM7TDMI
- einem Backend für den LEON 32 Bit Prozessor
- einigen Anpassungen für das ARM SDT 2.50 Kit [Adv98a, Adv98b] und
- einem Energieprofiler (enProfiler).

Der Ablauf einer Übersetzung einer C-Datei durch den encc zu einer Assembler-Datei ist in Abbildung 4.4 dargestellt.

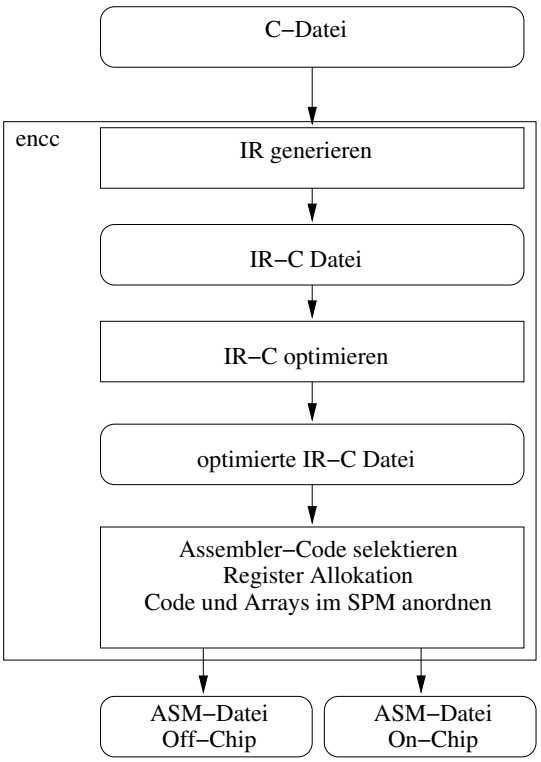


Abbildung 4.4: Übersetzung mit dem encc

Lance ist ein Frontend für den C89 Standard und verwendet als Zwischendarstellung Drei-Adress-Code, der als C-Datei gespeichert wird. Durch „lowering“ werden komplexe Programmkonstrukte wie Schleifen durch Label und Gotos ersetzt. Dies bedeutet, dass im encc die Information über Schleifen in einem ersten Schritt, dem Generieren der Zwischendarstellung, entfernt wird. Daher ist eine Code-Transformation im encc nicht so einfach wie mit dem ICD-C.

Der encc verwendet Baumüberdeckung zur Code-Erzeugung [Leu97], wobei die Überdeckung des Baums mit den geringsten Kosten ausgewählt wird. Um die Energie zu minimieren, existiert eine Datenbank, die zu jedem Befehlstyp angibt, welche Kosten entstehen. Diese Datenbank basiert auf dem Energiemodell von Steinke, das in Abschnitt 2.3 vorgestellt wurde.

Die Baumüberdeckung erschwert die Vorhersage, welcher Code generiert wird. Konstanten können durch Ausführung mehrerer Befehle generiert werden, anstatt aus dem Speicher geladen zu werden. Das Energiemodell des encc ermöglicht es, die günstigere Möglichkeit zu wählen.

Es wird eine globale Register-Allokation durch Färben des Interferenz-Graphen durchgeführt, d. h. die Register werden in jeder Methode möglichst optimal zugewiesen. Dabei verwendet der encc folgendes Vorgehen: Wenn ein Web durch Spilling aus dem Interferenz-Graph entfernt wurde, werden alle Zugriffe durch Speicherzugriffe ersetzt. Dabei entfällt die Berechnung der Spillingpositionen. Programme mit einer komplizierten Schleifenanordnung müssen daher nicht optimal übersetzt werden, sollen aber auch nicht in dieser Arbeit betrachtet werden.

Um die Spillingkosten abzuschätzen, wird für jede Schleife im Kontrollflussgraphen angenommen, dass sie konstant oft ausgeführt wird, so dass Variablen, die sehr weit außen verwendet werden, zuerst gespillt werden.

Die am Lehrstuhl Informatik XII der Universität Dortmund erforschten Optimierungen zur Scratchpad-Speicher-Nutzung sind im encc implementiert und können ausgewählt werden. Für diese Arbeit soll die statische Scratchpad-Speicher Allokation von Steinke [SWLM02] verwendet werden. Dabei werden Basisblöcke und Arrays in den Scratchpad-Speicher verlegt, wenn dies Gewinn bringt. Arrays und Basisblöcke werden immer als Ganzes verschoben.

Im Abschnitt 4.4 werden verschiedene Ansätze vorgestellt, wie die in Abschnitt 4.2 eingegrenzte Aufgabe umgesetzt werden kann. In dieser Arbeit wird der Algorithmus zu Belegung des Scratchpad-Speichers des encc nicht verändert.

Um die Entscheidung zu treffen, welche Objekte in den Scratchpad-Speicher verschoben werden können, müssen die Kosten und die Größe der einzelnen Objekte bekannt sein. Die Kosten hängen dabei nicht nur von den speziellen Befehlen und ihrer Abfolge ab, sondern auch davon, wie oft die Befehle ausgeführt werden.

Um die Information zu erhalten, wie oft ein Befehl ausgeführt wird, besitzt der encc zwei Möglichkeiten die statische Analyse und das dynamische Profiling. Bei der statischen Analyse wird das Programm selbst analysiert und angenommen, dass jede Schleife zehnmal ausgeführt wird. Bei Bedingungen wird angenommen, dass jeder Ausgang gleich wahrscheinlich ist. Aus diesen Informationen kann für jeden Basisblock geschätzt werden, wie oft er ausgeführt wird.

Für das dynamische Profiling wird das Programm vor der Belegung des Scratchpads vollständig generiert, übersetzt und im Simulator ausgeführt. Nach der Ausführung des Programms ist bekannt, welche Basisblöcke und Speicherzugriffe während der Ausführung stattgefunden haben. Die Kosten sind exakt berechenbar. Der Preis der Genauigkeit ist, dass das zu übersetzende Programm terminieren muss. Die Laufzeit der Übersetzung ist nicht nach oben beschränkt und hat mindestens die gleiche Komplexität wie das zu übersetzende Programm.

Für einfache Programme liefert die Heuristik der statischen Analyse gute Ergebnisse. Dynamisches Profiling hingegen kann das Ergebnis für getilte Programme, die mit dem in dieser Arbeit erzeugt werden, wesentlich verbessern. Tabelle 5.6 auf Seite 105, die im Abschnitt 5.2.4.2 auf Seite 104 beschrieben wird, zeigt den direkten Vergleich zwischen dynamischem Profiling und statischer Analyse. Für das optimierte Programm wird statt einer Verschlechterung um 6% mit dynamischen Profiling eine Verbesserung um 28% erreicht.

### 4.3.3 enProfiler

Der enProfiler ist ein Programm, das die Ausführung eines Programms verfolgt und anhand eines Energiemodells [SKWM01] den Energieverbrauch einer Ausführung aufschlüsselt.

Der enProfiler ist mit dem encc eng verbunden. Beide Programme verwenden die gleiche Datenbank mit Informationen über den Energieverbrauch einzelner Instruktionen. Der enProfiler berücksichtigt aber auch die Kosten, die bei einem Befehlsübergang entstehen. Sowohl für den enProfiler als auch den encc kann ausgewählt werden, wie groß der zur Verfügung stehende Scratchpad-Speicher sein soll und welche Konfiguration ein Cache haben soll.

Der enProfiler existiert in zwei Versionen. Beide Versionen liefern bei identischer Konfiguration die gleichen Ergebnisse. Während der Diplomarbeiten von Theokharidis [The00] und Knauer [Kna01] ist die ältere Version entstanden. Diese Version wurde in anderen Diplomarbeiten erweitert, um z. B. Flash und Low-Power SDRAM zu unterstützen [Ker05]. Es zeigte sich, dass die Codebasis zunehmend nicht mehr wartbar wurde. Eine neue Version nimmt die Erkenntnisse aus den vorherigen Versionen auf und ist besser erweiterbar.

In dieser Arbeit wurde die neue Version des enProfiler verwendet, um die Ergebnisse zu gewinnen, bei denen statische Analyse verwendet wird. Beim dynamischen Profiling wird auf den alten enProfiler zurückgegriffen, da dieser in die vorhandene Toolchain integriert ist. Der Arbeitsablauf ist in Abbildung 4.5 graphisch dargestellt. Der encc übersetzt das C Programm und generiert zwei Assembler-Dateien, eine für den Hauptspeicher und eine für den Scratchpad-Speicher.

Der Assembler übersetzt beide Dateien zu je einer Objekt-Datei. Der Linker linkt diese und einige Dateien der Laufzeitumgebung zu einem Binary zusammen. Der Armulator interpretiert das Binary und simuliert den Prozessor zyklengenau. Es existiert eine Schnittstelle, mit der sog. Trace-Files erzeugt werden. Ein Trace-File enthält alle Bustransaktionen der CPU und gibt Aufschluss darüber, welche Befehle ausgeführt wurden.

Das Trace-File wird vom enProfiler parallel analysiert und dem Energiemodell entsprechend der Energiebedarf der einzelnen Instruktionen bestimmt.

Im Rest dieses Abschnitts wird beschrieben, wie die Initialisierung des Systems funktioniert, welche Folgen sich daraus ergeben und wie die Analyse dies betrachten soll.

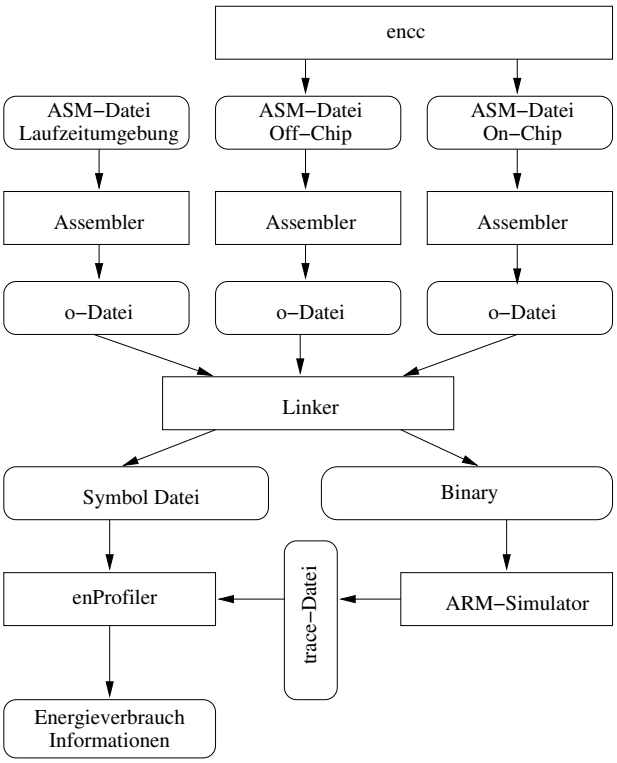


Abbildung 4.5: Profiling der Energie mit dem enProfiler

Wird ein Programm ausgeführt, so wird vor Beginn der main-Funktion das System initialisiert. Während dieser Initialisierung wird der Speicher konfiguriert und mit Daten aus dem Flash initialisiert. Sowohl der Hauptspeicher als auch der Scratchpad-Speicher wird durch die Initialisierung beschrieben. Je mehr Speicher belegt wird, desto länger dauert dieser Vorgang. Die Initialisierung des Speichers hängt nicht davon ab, wie der Speicher initialisiert werden soll. Auch wenn das Array nur deklariert wird, fällt der Aufwand für die Initialisierung an.

Die Funktion zur Initialisierung ist im ARM-Mode geschrieben. Da das Energiemodell nur für den THUMB-Mode vorliegt, kann der enProfiler diese Befehle nicht exakt analysieren. Statt des Werts, der für die ARM-Instruktion angenommen werden müsste, werden immer dieselben Kosten für eine Addition betrachtet. Das Modell für den Energieverbrauch des Speichers arbeitet im THUMB-Mode als auch im ARM-Mode gleich und ist daher vergleichbar.

Für die Analyse des Programms stellt sich die Frage, ob die Initialisierung, wenn auch ungenau, mitbetrachtet werden soll oder ob die Initialisierung nicht betrachtet werden soll.

Für die Betrachtung der Initialisierung spricht, dass

- die Kosten tatsächlich anfallen,

- Tiling mehr Scratchpad-Speicher und mehr Hauptspeicher belegt und somit Tiling mehr Kosten während der Initialisierung verursacht.

Es ist also fair, dass die Initialisierung beim Vergleich betrachtet wird. Wenn Code und Daten dynamisch in den Scratchpad-Speicher verlegt werden, ist eine Berücksichtigung der Initialisierung zum Vergleich erforderlich.

Dagegen spricht, dass

- die Kosten der Initialisierung nur eine Schätzung sind und die Genauigkeit der Ergebnisse beeinflussen können.
- Tiling bestraft wird, wenn dies berücksichtigt wird. In der Regel wird Tiling Kopierfunktionen einfügen und die Arrays nur deklarieren. Aber auch nur deklarierte Arrays werden nutzlos initialisiert. Eine Anpassung der Standardumgebung ist unerwünscht.
- der encc auch die Chance hat, den Scratchpad-Speicher kostenlos zu füllen. Dies kann im Modell einfach berücksichtigt werden.
- der encc bei statischer Belegung des Scratchpad-Speichers die Kosten in seinem Modell ebenfalls nicht berücksichtigt.

Die betrachteten Benchmarks sind nur Code-Ausschnitte und Teil eines größeren Programms, das die Code-Teile häufiger ausführt. Die Kosten zur Initialisierung fallen nur einmal an. In Abwägung des Für und Wider betrachtet diese Arbeit die Initialisierung des Speichers bei der Messung der Gesamtkosten nicht. Das Modell berücksichtigt die Möglichkeit der kostenlosen Initialisierung des Speichers.

### 4.3.4 PGAPack

PGAPack [Lev00] ist eine Bibliothek, die (parallel) genetische Algorithmen ausführt. Die Bibliothek ist unabhängig von den Datenstrukturen und kann von C aufgerufen werden. Die Nutzung von C++ ist auch möglich.

Über Funktionen werden die Parameter des genetischen Algorithmus gesetzt. Über `PGARun` wird ein Algorithmus entsprechend der gesetzten Parameter gestartet. Es wird eine Menge von Chromosomen erzeugt und für jedes Objekt eine Fitnessfunktion aufgerufen. Die Fitnessfunktion ist als Callback realisiert. Daher können nur statische Methoden von Klassen oder globale Methoden als Fitnessfunktion verwendet werden, da diese keinem Name-Mangling in C++ unterliegen.

Die Fitnessfunktion kann dann über Methoden auf einzelne Allele des Chromosoms zugreifen und eine Fließkommazahl als Fitnesswert zurückliefern.

Die Bibliothek kann nur maximieren, daher werden Minimierungsprobleme auf Maximierungsprobleme abgebildet. Es kann aus zwei Methoden gewählt werden.

Die Standardeinstellung sieht vor, dass die Fitness mit dem schlechtesten Wert mit 1.01 multipliziert wird. Von diesem Ergebnis werden dann jeweils alle Fitnesswerte abgezogen und stattdessen diese verwendet. Dadurch wird aus dem Minimierungsproblem ein Maximierungsproblem. Der schlechteste Wert wird dabei nicht 0 sondern bleibt etwas größer.

Ein Problem tritt auf, wenn vor der Transformation sehr große ($10^{99}$) und relativ kleine ($10^8$) Fitnesswerte vorkommen und minimiert werden sollen. Dann reicht die Auflösung der Stellen nicht aus und gute Fitnesswerte im Bereich $10^8$ werden durch sehr ähnliche Zahlen dargestellt. Die Sortierung dieser Zahlen ist zufällig, so dass die Fitness des besten Individuums nicht mehr monoton sein muss.

Es existiert eine andere Möglichkeit, die gewählt werden kann. Es wird die reziproke Fitnessfunktion verwendet. Dabei wird weniger Information verschenkt. Es wird der Effekt ausgenutzt, dass Fließkommazahlen im Bereich zwischen $[0, 1]$ beinahe gleich viele Zahlen darstellen können wie zwischen $[1, \infty]$. Hingegen

wird bei der Subtraktion zwischen großen und kleinen Zahlen immer eine Exponentenanpassung durchgeführt, die dafür sorgen kann, dass die Information der kleinen Zahl verloren geht, wenn die Exponenten nicht gleich sind. Ein Nachteil dieser Methode ist, dass die Verhältnisse nicht erhalten bleiben, daher kann der Nutzer zwischen beiden Möglichkeiten auswählen.

Die Bibliothek liegt im Quelltext vor und steht unter einer freien Lizenz, darf also kostenlos kopiert, modifiziert und weitergegeben werden, solange die Copyright-Hinweise wiedergegeben werden und kein Geld für die Software verlangt wird.

## 4.4 Lösungsmöglichkeiten

Es existieren verschiedene Möglichkeiten, wie die verschiedenen Schritte, die zur Lösung notwendig sind, angeordnet und in den vorhandenen Arbeitsablauf des encc eingesetzt werden können. Dies wird im folgenden Abschnitt beschrieben. Zunächst wird erläutert, warum der encc nicht erweitert wird und statt dessen ein Frontend geschrieben werden soll. Anschließend wird der Highlevel-Ansatz beschrieben, der den Scratchpad-Speicher dynamisch belegen kann, und erläutert, warum er nicht verwendet wird. Es wird auch diskutiert, warum eine dynamische Belegung des Scratchpad-Speichers mit Code und Arrays in dieser Arbeit nicht umgesetzt wird. Im letzten Teil dieses Abschnittes wird diskutiert, mit welchen Mitteln die Teil-Arrays im Scratchpad-Speicher deklariert werden sollen.

Von der Möglichkeit Gebrauch zu machen, den encc selbst zu erweitern, wurde Abstand genommen, da die Highlevel-Informationen über Schleifen während der Generierung der Zwischenrepräsentation entfernt werden. Ein weiteres Problem ist die Code-Transformation. Selbst wenn die Schleifen und Kontrollvariablen erkannt werden, so müsste auf Assembler-Ebene der Code transformiert werden. Die zu ändernden Teile müssen erkannt werden, auch wenn sie durch Entfernung gemeinsamer Ausdrücke nicht mehr einmalig vorhanden sind. Die Modifikation des Assembler-Code hat nur dann Vorteile, wenn der endgültige Code feststeht. Dann sind aber schon Optimierungen durchgeführt worden und Transformationen von gemeinsamen Ausdrücken müssten vermieden werden. Letztlich kann sich auch bei der Modifikation des Code im Backend dieser genauso verändern, wie er sich verändert, wenn das C-Programm modifiziert wird. Die Abschätzung der Kosten wird nicht vereinfacht. Eine Modifikation auf C-Ebene scheint wesentlich einfacher als eine Programmtransformation wie Tiling im Backend zu implementieren. Es wird daher ein Frontend für den encc präsentiert, das das C-Programm modifiziert. Dieses modifizierte Programm soll dann vom encc übersetzt werden.

Auch mit dem Vorsatz, ein Frontend zu schreiben, existieren noch eine Reihe verschiedener Möglichkeiten. Zur Verschiebung der Arbeitsbereiche von Arrays werden verschiedene Varianten in Betracht gezogen. Die verschiedenen Ansätze werden im Folgenden vorgestellt.

Der Highlevel-Ansatz verwendet die dynamische Scratchpad-Speicherbelegung von Verma [VWM04b]. Das C-Programm wird entsprechend einer Heuristik getiled, so dass alle Arrays in den Scratchpad-Speicher passen. Die Tiling-Schleifen werden abgerollt und die einzelnen Array-Tiles so umbenannt, dass sie als eigenständige kleine Arrays deklariert sind. Das modifizierte Programm wird dann vom encc übersetzt, der den Scratchpad-Speicher unter Verwendung des in Abschnitt 3.2.2 erwähnten Allokationsalgorithmus [VWM04b] dynamisch belegt. Die Kopierpositionen werden vom encc bestimmt.

Die Heuristik könnte an bestimmten Code-Teilen feststellen, dass Tiling in einem Nest von Vorteil ist und im anderen nicht. Wenn aber auf das gleiche Array zugegriffen wird, muss auch auf den gleichen Speicher zugegriffen werden. Es muss sowohl auf das komplette Array als auch auf die Teil-Arrays zugegriffen werden können. Die Speicheranordnung von mehrdimensionalen Array-Tiles und dem ursprünglichen Array ist nicht kompatibel. Es soll daher davon ausgegangen werden, dass auch das Original-Array als eine Struktur aus Array-Tiles abgespeichert wird. Um dies zu deklarieren, bietet sich ein Konstrukt aus verschachtelten Unions und Structs an, so dass auf das komplette Array oder das getilte Array wahlweise zugegriffen werden kann.

In C kann ein zweidimensionales Array mit ganzzahligen Elementen durch

```
int A[50][50];
```

deklariert werden. Damit getiled und ungetiled auf ein zweidimensionales Array zugegriffen werden kann, wird stattdessen die Datenstruktur Array2D wie folgt deklariert:

```
struct discontinuous{
        int  a0[25];
        int  a1[25];
};
union Array{
     int  cont[50];
     struct discontinuous split;
};
struct discontinuous2D{
        union Array  a0[25];
        union Array  a1[25];
};
union Array2D {
     union Array  cont[50];
     struct discontinuous2D split;
};
```

Zuerst wird für den eindimensionalen Fall mit dem Union `Array` eine Möglichkeit geschaffen, kontinuierlich (`cont`) oder in Teilen (`split`) auf den Speicher zuzugreifen. `split` ist ein Struct und überdeckt ebenfalls 50 Elemente. Über `a1` kann auf die ersten 25 Elemente zugegriffen werden, über `a2` auf die restlichen 25 Elemente.

Für den zweidimensionalen Fall wird dieses Schema fortgesetzt und auf die eindimensionale Struktur `Array` zugegriffen. Für weitere Dimensionen und andere Aufteilung ließe sich dieses Verfahren leicht fortsetzen.

Der Benchmark Matrix-Multiplikation mit 50 Elementen in jeder Dimension wurde nach diesem Ansatz getiled. Das Programm ist im Anhang A.2 abgedruckt.

Ein Zugriff auf das Matrix-Element `A[10][30]=5;` hat dann jeweils folgende Form:

**kontinuierlicher Zugriff:** Hier wird das Feld `cont` verwendet:

```
A.cont[10].cont[30]=5;
```

**getilter Zugriff:** Hier wird jeweils das Feld `split` verwendet und es wird ausgewählt, ob das erste oder zweite Feld verwendet wird:

```
A.split.a1[10].split.a2[5]=5;
```

Dieses Verfahren scheitert jedoch am experimentellen Charakter des encc. Durch eine Union kann auch durch einen anderen Namen auf den selben Speicher zugegriffen werden, was die 1-zu-1-Beziehung zwischen Namen und betrachteten Speicherzellen aufbricht. Daher kann der encc keine unions verarbeiten und das modifizierte Programm kann nicht übersetzt werden.

Unter der Annahme, dass alle Zugriffe getiled vorkommen, ist die Nutzung des Unions nicht notwendig und das Programm kann mit dem encc übersetzt werden. In Tabelle 4.5 ist für die Matrix-Multiplikation mit 14 Elementen der Energieverbrauch für verschiedene Übersetzungen angegeben. Es wird angenommen, dass 1024 Byte Scratchpad-Speicher zur Verfügung stehen. Für den Highlevel-Ansatz ist nicht bekannt, welche Tiling-Faktoren optimal sind. Daher wurde für alle Schleifen der Tiling-Faktor 7 gewählt. Das Programm wurde im Original übersetzt, dies dient als Referenz. Dann wurde das Programm mit den Tiling-Faktoren 7-7-7 getiled und mit dynamischen Profiling übersetzt. Dieses Programm hat eine Verschlechterung von 6%.

| | Orginal | Getiled | Highlevel | Interleaved-static | Interleaved-dyn |
|---|---|---|---|---|---|
| Tiling-Faktoren | - | 7-7-7 | 7-7-7 | 7-5-14 | 7-5-14 |
| Energie [$\mu J$] | 978 | 1039 | 1544 | 921 | 887 |
| Verbessserung [%] | - | −6 | −58 | 6 | 9 |

Tabelle 4.5: Energieverbrauch Matrix-Multiplikation mit 14 Elementen, verschiedene Übersetzungen, 1024 Byte Scratchpad-Speicher

Wird das Programm nach dem Highlevel-Ansatz mit den Tiling Faktoren 7 getiled, die Tiling-Schleifen abgerollt und mit dem dynamischen Verfahren der Scratchpad-Speicher belegt, wird etwa 1,5-mal soviel Energie verbraucht.

Im Vorgriff auf das Ergebnis ist in der Tabelle zum Vergleich auch der Energieverbrauch angegeben, der bei einer Übersetzung mit statischer Analyse oder dynamischen Profiling ensteht. Mit dynamischem Profiling ist eine Verbesserung um fast 10% gegenüber dem Original zu erzielen. Da statische Analyse eine wesentlich schnellere Übersetzungszeit erlaubt, ist statische Analyse das in dieser Arbeit bevorzugte Verfahren.

Ein weiterer Nachteil des Highlevel-Ansatzes ist die exponentielle Code-Explosion, die durch das Abrollen der Tiling-Schleifen entsteht. Der Code kann zwar dynamisch ausgetauscht werden, dies erzeugt aber zusätzlichen Overhead. Der Highlevel-Ansatz wurde daher nicht weiter verfolgt.

Die dynamische Belegung des Scratchpad-Speichers hat für größere Programme mit mehreren Schleifen Vorteile. Es würde sich daher anbieten, die dynamische Belegung des Scratchpad-Speichers mit Code, Daten und Arrays in Betracht zu ziehen. Damit ein Frontend diese Effekte im encc nachvollziehen kann, muss aber das Modell des Frontends dieselben Schritte durchführen und zusätzlich die Auswirkungen der Codemodifikation verfolgen. Es steht aber noch gar nicht fest, dass man in einem Frontend die Auswirkungen der Code-Modifikation genau genug vorhersagen kann. Der Aufwand einer Implementierung der dynamischen Belegung von Scratchpad-Speicher mit Code und Arrays ist nicht abzuschätzen und übersteigt den Aufwand, der in dieser Arbeit geleistet werden kann. Die historische Entwicklung der Scratchpad-Speicher-Belegung in Abschnitt 3.2.1 zeigt, dass zunächst eine statische Belegung betrachtet wird und diese immer weiter verfeinert wurde, bis schließlich ein Verfahren zur dynamischen Belegung gefunden wurde. Tiling erhöht die Komplexität der Modellierung, so dass es sinnvoll erscheint, zunächst mit einer statischen Code-Belegung zu beginnen.

Eine weitere Dimension der Lösungsmöglichkeiten ist, wie die Tile-Arrays im Scratchpad-Speicher abgelegt werden sollen. Zwei Möglichkeiten werden im Folgenden betrachtet. Die Implementierung verwendet für jedes Array ein korrespondierendes kleines Array, das in den Scratchpad-Speicher verschoben werden kann. Eine andere Möglichkeit ist, nur ein neues Array zu deklarieren.

Zuerst wird betrachtet ein neues Array zu deklarieren. Der Speicherbereich im neuen Array wird je nach Bedarf in mehrere logische Teile unterteilt, die dann als ein Array behandelt werden. Dies entspricht dem Vorgehen, wenn ein Compiler auf die verschiedenen globalen Arrays im Speicher zugreift und die Adressen für einzelne Array-Elemente berechnet werden. Mehrere Arbeitsbereiche unterschiedlicher Schleifennester können sich denselben Speicherbereich teilen. Für den encc existiert ein zusätzliches Array, das in den Scratchpad-Speicher verschoben werden kann. Es muss ein Verfahren implementiert werden, dass eine optimale Anordnung der Teil-Arrays in dem zusätzlichen Array für die Ausführungszeit findet. Ein mögliches Verfahren zur Lösung des „Onchip Address Assignment Problem“ ist in [VWM04b] beschrieben. Es würde sich anbieten, dieses Verfahren auch hier zu verwenden, indem für jeden Arbeitsbereich ein kleines Array deklariert wird und sich die Arrays den Platz im Scratchpad-Speicher teilen. Dadurch, dass nur ein Array verwendet wird, ändert sich der Code drastisch, so dass eine Abschätzung der entstehenden Kosten im Verhältnis zum Originalprogramm sehr schwierig erscheint.

In dieser Arbeit wird für jedes Array, das während der Ausführung einer Schleife in den Scratchpad-Speicher verschoben werden kann, ein Array angelegt. Der Code bleibt so zum Original noch sehr ähnlich

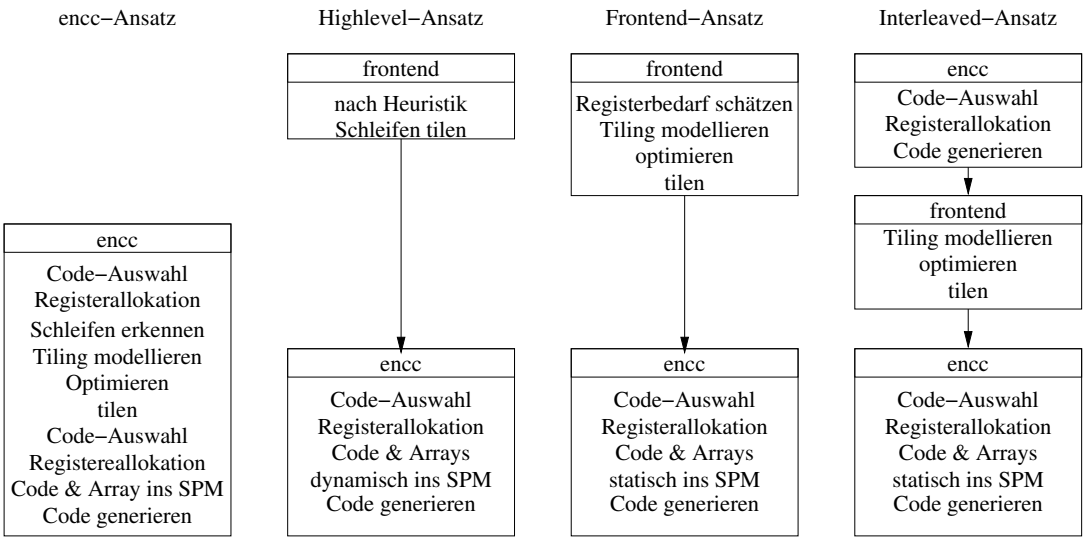


Abbildung 4.6: Mögliche Arbeitsabläufe

und leicht verständlich. Die verwendete statische Belegung des Scratchpad-Speichers verschiebt die deklarierten kleineren Arrays in den Scratchpad-Speicher. Auf eine Untersuchung von Benchmarks mit mehreren Schleifennestern wird verzichtet. Die Reimplementierung des „Onchip Address Assignment Problem“ im Frontend bleibt einer möglichen Erweiterung vorbehalten, steht aber dem Ziel entgegen, so viel wie möglich wieder zu verwenden.

Im folgenden Abschnitt wird der gewählte Arbeitsablauf beschrieben und mit anderen denkbaren Abläufen verglichen.

### 4.4.1 Gewählter Arbeitsablauf

In Abbildung 4.6 sind die verschiedenen Arbeitsabläufe gegenübergestellt. Die Möglichkeit, den encc zu erweitern, wurde nicht näher in Betracht gezogen. Informationen zu den Schleifen werden durch lowering im ersten Schritt der Generierung der IR entfernt. Die Informationen müssten erst zurückgewonnen werden. Diese Möglichkeit bildet den encc-Ansatz.

Der Highlevel-Ansatz wird nicht verfolgt, da durch Abrollen der Schleifen zuviel Code erzeugt wird.

Der Frontend-Ansatz verfolgt die Lösung durch ein Frontend. Der C-Code wird analysiert. Es wird ein Optimierungsproblem aufgestellt, dass die Ergebnisse der Analyse aufnimmt und als Ergebnis Informationen liefert, wie der Code zu verändern ist. Der Code wird durch das Frontend modifiziert und durch den encc übersetzt. Zur Abschätzung des Overheads sollen auch die Spillingkosten berücksichtigt werden. In Abschnitt 4.1.4 ist zu erkennen, dass Spilling wesentliche Kosten verursacht. Dazu ist es notwendig, dass abgeschätzt wird, wieviele Register schon belegt sind. Die Abschätzung, wieviele Register belegt sind, ist aber nur sehr ungenau mit vertretbarem Aufwand möglich. Einfach kann nur gezählt werden, wieviele Variablen lebendig sind.

Der Interleaved-Ansatz ist eng mit dem Frontend-Ansatz verwandt. Es wird abwechselnd der encc und das Frontend ausgeführt. Die Steuerung des Ablaufs übernimmt das istf-Frontend. Die Buchstaben stehen für **I**CD-C, **S**cratchpad-Speicher, **T**iling und **F**rontend. Der wesentliche Unterschied zum Frontend-Ansatz besteht darin, dass genauere Informationen über den zu erwartenden Code und den zu erwartenden Registerdruck vorhanden sind. Der Interleaved-Ansatz wird in dieser Arbeit umgesetzt.

Abbildung 4.6 zeigt rechts einen Überblick des Ablaufes. In Abbildung 4.7 wird detaillierter der Ablauf im Frontend dargestellt. Für den Interleaved-Ansatz wird das Originalprogramm zunächst vom encc übersetzt,

ohne dass der Scratchpad-Speicher benutzt wird. Der encc kann für jeden Basisblock ausgeben, wieviele Register verbraucht wurden. Diese Information kann zur Abschätzung der Spillingkosten herangezogen werden und wird in einer rpi-Datei (Register Pressure Information) gespeichert. Die generierte Assembler-Datei bietet Informationen darüber, welcher Code für einzelne Programmteile erzeugt wird.

Diese Informationen werden genutzt, um den Energieverbrauch des modifizieren Programms möglichst genau zu modellieren. Die Modifizierung des Programms besitzt verschiedene Freiheitsgrade, die Einfluss auf das Programm haben. Wenn die Modellierung des Tilings möglichst gut mit dem Energieverbrauch des tatsächlichen Programms korreliert, kann das Modell verwendet werden, um eine gewinnbringende Parameterkombination zu finden.

In das Modell fließen Informationen über den Registerdruck, Größe und Kosten der Basisblöcke des Originalprogramms, die Kopierkosten, Overhead durch zusätzliche Ausführungen nicht innerster Schleifen und Kosten für die Berechnung von Zugriffsfunktionen ein.

Entsprechend dem Ergebnis der Optimierung im Modell kann das Programm getiled werden. Dazu lesen die Klassen des Frontends das Optimierungsproblem aus und verändern die ICD-C IR-Repräsentation des Programms wie es das Ergebnis vorschreibt.

Die Größe der Basisblöcke wird zusammen mit der Information über die Größe der modifizierten Programmteile verwendet, um zu bewerten, welche Teile des Scratchpad-Speichers für Code oder Daten energieeffizient verwendet werden können.

Die Informationen über die analysierte Assembler-Datei und die Informationen aus der C-Datei werden in Objekten des istf gespeichert. Aus diesen Informationen wird das Optimierungsproblem gefüllt und gelöst. Die Objekte nehmen die Ergebnisse der Optimierung auf und transformieren die ICD-C IR-Repräsentation des Programms entsprechend. Das modifizierte Programm wird in eine .opt.c-Datei geschrieben. Das Frontend kann eine Übersetzung und ein anschließendes Profiling des Programms veranlassen.

Für die Arbeitsbereiche der Arrays in den Elementschleifen werden kleine Arrays deklariert, wenn der Arbeitsbereich in den Scratchpad-Speicher verschoben werden kann. Für jedes Array, das verschoben werden soll, werden vom Frontend Aufrufe zum Kopieren des Arbeitsbereichs eingefügt. Während der Ausführung des Programms werden die Arbeitsbereiche dynamisch kopiert. Ein lesender Zugriff erfordert einen Kopieraufruf vor dem Beginn des zugreifenden Schleifennestes. Ein schreibender Zugriff führt zu einem Kopieraufruf nach dem zugreifenden Schleifennest. Tiling-Schleifen werden eingefügt.

Das Frontend und der encc verwenden die gleiche Datenbank zur Abschätzung der Kosten von Instruktionen. Wenn das Modell zutreffend ist, wird vom encc der vorhergesagte Code generiert. Der encc verwendet ein einfacheres Modell zur Vorhersage der Ausführungshäufigkeit von Schleifen. Dies kommt für einfach geschachtelte Schleifen zu ähnlichen Verhältnissen, so dass die Modellierung des Energieverbrauchs und die Code-Größe der einzelnen Basisblöcke im Frontend und encc übereinstimmen.

Der encc benutzt den Algorithmus zur statischen Belegung des Scratchpad-Speichers von Steinke et al. [SWLM02]. Basisblöcke und Daten werden in den Scratchpad-Speicher statisch verschoben, wenn dies profitabel ist. Dazu wird ein modifiziertes Rucksackproblem als ganzzahliges lineares Optimierungsproblem aufgestellt und gelöst.

Auf die eingeführten kleinen Arrays wird so häufig zugegriffen, dass es sich lohnt, diese statisch in den Scratchpad-Speicher zu verlegen. Der Inhalt dieser Arrays wird aber letztendlich dynamisch durch die eingeführten C-Kopierfunktionen des Frontends ausgetauscht.

Sollte das Modell des encc nicht mit dem Frontend übereinstimmen, so kann zu wenig Platz im Scratchpad-Speicher sein. Code-Teile werden nicht so günstig ausgeführt wie angenommen. Dies betrifft eher Code-Teile, die selten ausgeführt werden, so dass der Effekt klein ist. Schwerwiegender ist es, wenn der encc zu der Entscheidung kommt, ein kleines Array nicht in den Scratchpad-Speicher zu verlegen, denn dann existiert der vom Frontend angenommene Gewinn nicht. Der Overhead zum Kopieren ist, wie in Abbildung 4.2 zu sehen, sehr groß, so dass keine Verbesserung des Energieverbrauchs erwartet werden kann. In beiden Fällen sollte betrachtet werden, wie das Modell des Frontends verfeinert werden kann. Alternativ kann

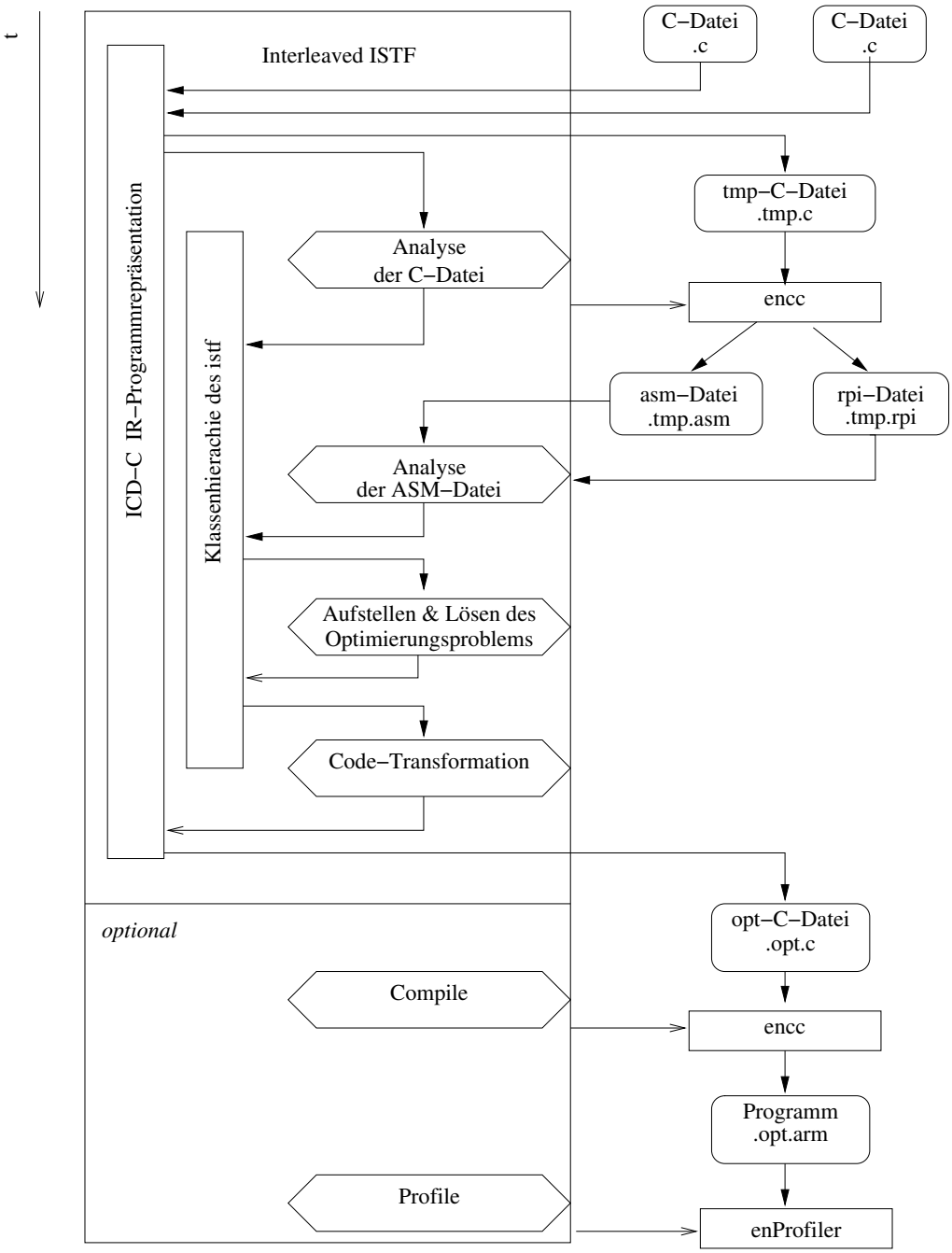


Abbildung 4.7: Übersicht über den Interleaved-Ansatz

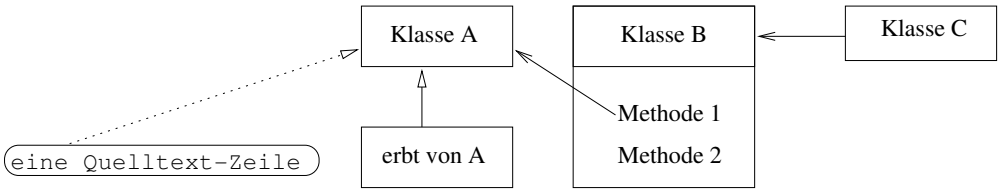


Abbildung 4.8: Übersicht über die verwendeten UML-Notationen

in Betracht gezogen werden, dem encc vorzugeben, dass bestimmte Arrays in den Scratchpad-Speicher verschoben werden müssen.

Im Folgenden wird der gewählte Ansatz umgesetzt. Die Struktur orientiert sich am Ablauf im Frontend: Programmanalyse – Optimierung – Programmtransformation.

## 4.5 Programmanalyse

Im Folgenden wird die Programmanalyse näher beschrieben. Dazu werden Ausschnitte aus Klassendiagrammen verwendet. Zunächst wird auf das Vorgehen beim Entwurf und die verwendete Notation eingegangen.

Hier und in den anderen Klassen wird das Konzept verfolgt, dass eine Klasse für eine abstrakte Struktur verantwortlich ist und Methoden bereit stellt, Informationen aufzunehmen und geeignet weiter zu propagieren. Die Klassen besitzen Methoden, die den Ablauf steuern und Datenstrukturen füllen. Einige Klassen entsprechen ICD-C IR Objekten, wesentliche Attribute werden zwischengespeichert. Durch Methoden dieser Klassen kann später auch das Programm transformiert werden.

In Abbildung 4.8 ist eine Übersicht über die verwendete UML-Notation gegeben. Eine ausführlichere Referenz bietet [HKKR05]. Die Klasse „erbt von A“ erbt von Klasse A, dies wird durch einen geschlossenen Pfeil gekennzeichnet. Methoden werden in abgetrennten Kästen dargestellt. Wenn eine Methode eine Klasse verwendet, wird dies mit einem offenen Pfeil zwischen der Methode und der Klasse dargestellt. Existiert zwischen den Klassen C und B eine Assoziation, so wird dies durch einen offenen Pfeil zwischen den Klassen dargestellt. Klasse C besitzt dabei einen Zeiger auf B. Existieren mehrere Zeiger, so wird ein *n* am Pfeil annotiert. Wenn eine Quelltextzeile durch eine Klasse repräsentiert wird, dann wird dies durch einen geschlossenen Pfeil mit einer gestrichelten Linie kenntlich gemacht. Wird nur ein Teil repräsentiert, so kann dies durch einen runden Rahmen um den Teil kenntlich gemacht werden.

Die Programmanalyse und der gesamte Ablauf im Frontend wird durch die Klasse File gesteuert. Bei der Analyse der C-Datei wird eine Klassenhierarchie verwendet, deren Wurzel ein Function-Objekt ist. Zur Analyse der Assembler-Datei wird eine Klassenhierarchie verwendet, deren Wurzel ein AsmFile-Objekt ist. Dies wird in Abbildung 4.9 graphisch dargestellt.

Die Verknüpfung der Informationen aus der Assembler-Datei mit den Informationen aus der C-Datei wird durch das File-Objekt gesteuert. Nach der Analyse wird das Optimierungsproblem durch die Klassen gefüllt und es wird gelöst.

Klassen aus der Function-Hierarchie werden benutzt, um das Programm zu modifizieren, die Ergebnisse des Optimierungsproblems werden dabei verwendet. Optional ist die Übersetzung (compile) des optimierten Programms durch den encc und das Profiling (profile) durch den enProfiler.

### 4.5.1 Analyse des C-Progamms mit ICD-C

Bei der Analyse des C-Programms wird auf die ICD-C Bibliothek zurückgegriffen. Wesentlich für die Analyse sind die Schleifen und Array-Zugriffe. Jeder Funktion wird ein Function-Objekt zugeordnet. An-

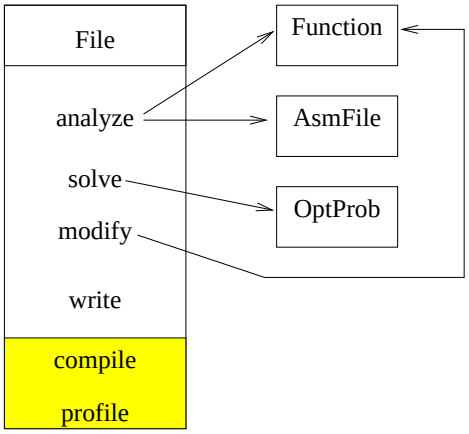


Abbildung 4.9: Ausschnitt aus dem UML Diagramm zur Klasse File

weisungen, die nicht Teil einer Schleife sind, werden mit dem Function-Objekt assoziiert. Schleifen werden gesondert behandelt. Zu jeder Schleife des Programms gehört ein LoopNest-Objekt. Es wird die obere und untere Schranke der Schleife gespeichert. Jedes LoopNest-Objekt kann weitere LoopNest-Objekte und Stmts behandeln. Im Rumpf einer jeden Schleife wird nach Array-Referenzen gesucht und zu jeder Referenz ein ArrayRef-Objekt erzeugt. Je Dimension existiert eine Zugriffsfunktion. Die Koeffizienten der affinen Zugriffsfunktion werden in einem Accessvektor-Objekt gespeichert.

Die Function-, LoopNest- und Arrayref-Objekte werden entsprechend Abschnitt 4.6.1 nummeriert. Die in Abbildung 4.10 genannten Klassen besitzen jeweils auch Referenzen auf die korrespondierenden ICD-C IR-Objekte, die bei der Code-Modifikation verwendet werden.

### 4.5.2 Analyse der Assembler-Datei

Nachdem alle C-Dateien eingelesen worden sind, wird die Zwischendarstellung ohne Modifikationen in eine Datei geschrieben, die die Endung .tmp.c besitzt. Diese Datei wird vom encc ohne Nutzung des Scratchpad-Speichers übersetzt und das übersetze Programm in einer Assembler-Datei mit der Endung .tmp.asm gespeichert.

Die Klasse AsmFile behandelt alle Belange dieser Assembler-Datei. Nach der Generierung wird die Assembler-Datei eingelesen und analysiert. Dabei wird das Format der generierten Assembler-Datei ausgenutzt. Es wird eine Baumstruktur erzeugt, die die wesentlichen Teile der Assembler-Datei repräsentiert. Wie bei der Analyse der C-Datei speichern die korrespondierenden Klassen die für die Analyse wesentlichen Attribute und können die Parameter des Optimierungsproblems setzen. Da die Assembler-Datei nicht verändert wird, entfallen Methoden zur Code-Modifikation.

Eine vom encc generierte Assembler-Datei besteht zunächst aus Kommentaren und Deklarationen, diese werden ignoriert. Nach dem Beginn des Code-Segments beginnt die Analyse einer jeden Zeile, bis das Ende des Programms erreicht ist. Eine Zeile wird ignoriert, wenn sie nur einen Kommentar enthält. Sonst wird ein Objekt erzeugt. Abbildung 4.11 stellt die Korrespondenzen und den Auszug aus dem UML-Diagramm dar.

**Definition 4.5.1:** *Ein* **Basisblock** *ist eine Folge von Assembler-Anweisungen, von denen nur der letzte Befehl ein (bedingter) Sprung sein kann. Ein Basisblock besteht aus mindestens einer Anweisung. Basisblöcke beginnen mit einem Label, das angesprungen werden kann. Spätestens nach jedem (bedingten) Sprung beginnt ein neuer Basisblock.*

Jeder durch den encc generierte Basisblock beginnt mit einem Label, auch wenn dies nicht angesprungen wird. Für jeden Basisblock wird ein Basisblock-Objekt angelegt; dieses Objekt verwaltet Listen von

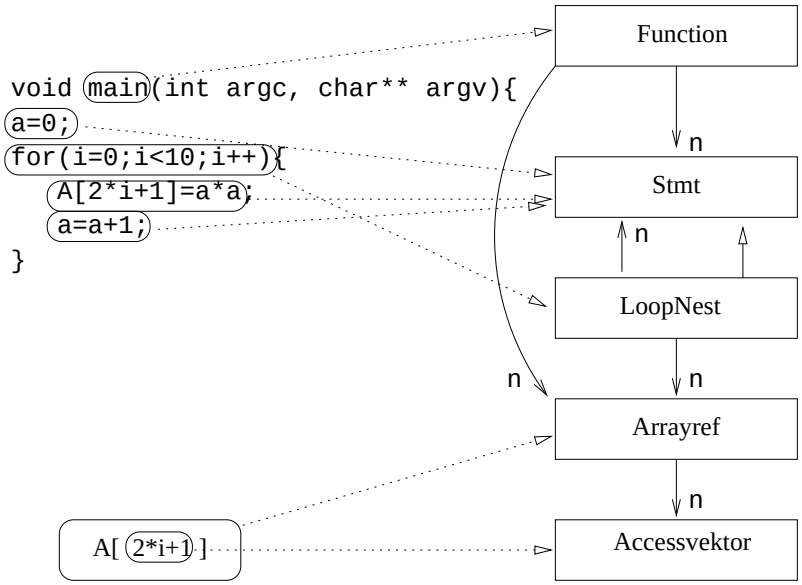


Abbildung 4.10: Ausschnitt aus dem Klassendiagramm zur Analyse der C-Datei und von diesen Klassen repräsentierte Programmstrukturen

AsmStmt-Objekten und LineSet-Objekten. In einem Basisblock können mehrere Kommentare vorkommen, die angeben, aus welcher Zeile der C-Datei die folgenden Assembler-Befehle generiert wurden. Ein Basisblock besteht aus Assembler-Zeilen; zu jeder Assembler-Zeile wird ein AsmStmt-Objekt erzeugt.

Aus der Assembler-Datei ist nicht direkt ersichtlich, wie oft jeder Basisblock ausgeführt wird. Die Ausführungshäufigkeit einzelner Teile kann sich durch Tiling ändern, daher muss berechnet werden, zu welchem Teil des Programms ein bestimmter Basisblock gehört. Dies wird im folgenden Abschnitt näher beschrieben.

Damit zu jedem Basisblock dieselben Kosten berechnet werden wie beim Modell des encc, muss dasselbe Energiemodell verwendet werden. Zu jeder Assembler-Zeile, die als Assembler-Befehl aufgefasst wird, überprüft das Objekt, ob es sich um ein Memonic (Kürzel) handelt. Das Energiemodell wird von der Klasse enccconnection gekapselt. Über eine Methode dieser Klasse kann das Kostenlabel für eine spezielle Code-Zeile berechnet werden.

Da auf die Datenbank nur über die numerischen Kostenlabel und Bitmuster der ausgeführten Befehle zugegriffen werden kann, musste ein Verfahren implementiert werden, wie zu einer gegebenen Assembler-Zeile das richtige Kostenlabel gefunden wird.

Dazu wird zu der Assembler-Zeile ein abstrakter Schlüssel generiert, der aus dem Memonic und einer abstrakten Kodierung der Parameter besteht. Die Kodierung abstrahiert von den konkret verwendeten Registern und numerischen und symbolischen Parametern. Das folgende Beispiel zeigt zu einer Assembler-Zeile den abstrakten Parameter:

```
POP {r1,r2}
POP{2}
```

Die Datenbank speichert die Prozessorkosten. Für jeden Befehl ist ein Eintrag vorhanden. Das Energiemodell berücksichtigt die Zugriffe auf Speicher separat. Die Kosten für Instruction Fetch und Datenzugriffe hängen vom verwendeten Speicher ab und werden separat erfasst. Die folgenden Zeilen zeigen einen Ausschnitt aus der Datenbank:

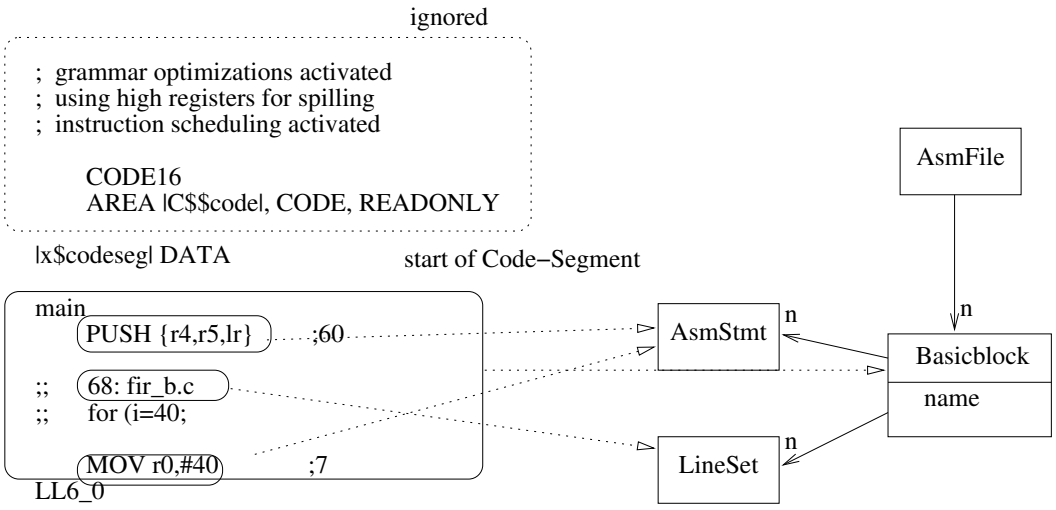


Abbildung 4.11: Ausschnitt aus dem Klassendiagramm zur Analyse der Assembler-Datei und von diesen Klassen repräsentierte Programmstrukturen

```
89,1011110000000101,4,2,2,8,48.8,"POP {Rlist}","CO_POP_2"
137,0001110xxxxxxxxx,1,2,0,0,44.4,"MOV Rd,Rs","CO_MOV_2_RR","Move"
```

Über Kommentare, die in der Datenbank enthalten sind, wird zu jeder Zeile der Datenbank ebenfalls ein passender Schlüssel generiert. Über die abstrakten Schlüssel eines Befehls kann eine Zeile in der Datenbank gefunden werden. Für jeden Basisblock werden die Kosten der Ausführung berechnet, wenn der Code vom Scratchpad-Speicher oder Hauptspeicher ausgeführt wird.

### 4.5.3 Verbindung des C-Programms mit den Informationen aus der Assembler-Datei

Der wesentliche Vorteil des Interleaved-Ansatzes ist, dass genaue Informationen über das Originalprogramm vorliegen. Die generierte Assembler-Datei wurde eingelesen und bietet Informationen, welche Kosten die einzelnen Basisblöcke je Ausführung verursachen.

Die Ausführunghäufigkeit einzelner Programmteile ist jedoch nur für das C-Programm bekannt. Es muss daher eine Zuordnung getroffen werden, zu welchem C-Programmteil ein Basisblock gehört. Die C-Programmteile werden auf CodeBlock-Objekte abgebildet. Ein CodeBlock-Objekt kapselt die Information, wo ein betreffendes Codefragment im Programm liegt, dazu wird die Funktionsnummer, Nestnummer und Schleifennummer gespeichert. Mit Hilfe dieser Information kann auch nach dem Tiling berechnet werden, wie oft ein Code-Fragment ausgeführt wird. Wie die Nummerierung genau erfolgt, wird in Abschnitt 4.6.1 erläutert. Ein CodeBlock ist die Bezeichnung einer Menge von Stmts, die gleich oft ausgeführt werden und zusammengehören, d. h. vom gleichen Block umklammert werden.

In Abbildung 4.12 sind die Objektinstanzen auszugsweise dargestellt, die nach dem Einlesen der C-Datei und der Assembler-Datei existieren. Das File-Objekt kann auf BasisBlock-Objekte und auf Stmt-Objekte zugreifen, die auch LoopNest-Objekte sein können. Stmts haben über korrespondierende IR_Stmts Zugriff auf ein IR_FileContext-Objekt, das die Datei und die Position repräsentiert, wo das Statement auftritt. Die BasicBlock-Objekte haben Zugriff auf mehrere LineSet-Objekte, die ebenfalls diese Information bereitstellen.

Eine Methode des File-Objektes iteriert über alle LineSet-Objekte und sucht unter den Stmt-Objekten eines mit der passenden Datei und Position. Dem Stmt-Objekt wird das LineSet-Objekt übergeben. Das Stmt-Objekt setzt die Referenz auf das „passende“ CodeBlock-Objekt.

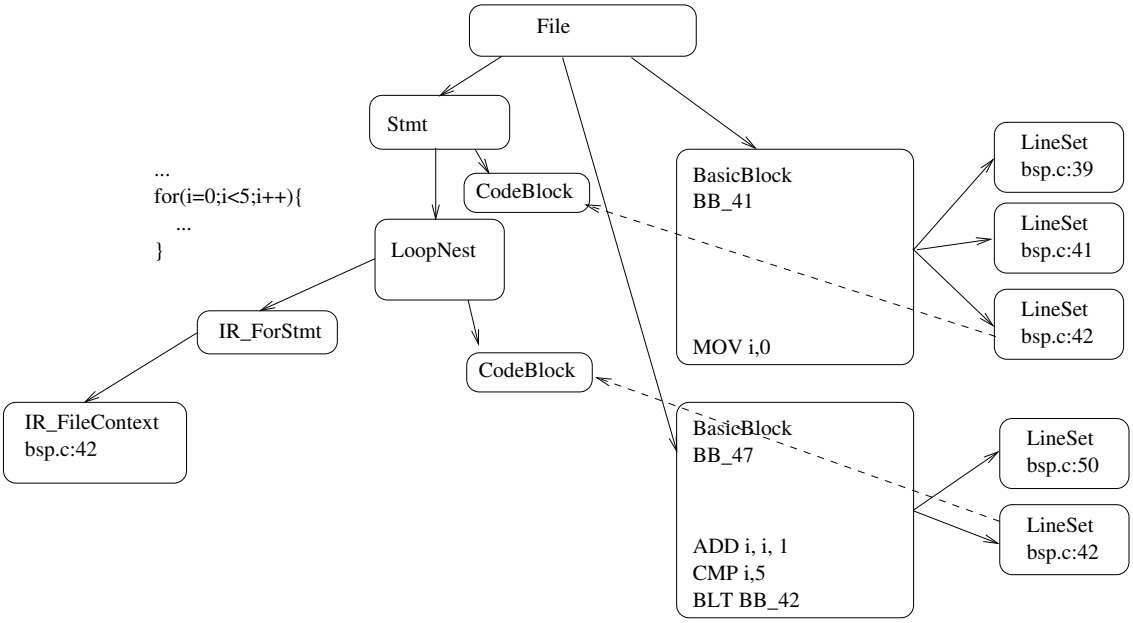


Abbildung 4.12: Zusammenführen von Assembler-Datei und C-Programm

Für einfache Stmt-Objekte ist dies das CodeBlock-Objekt, das zu diesem Stmt gehört.

Die Betrachtung von Schleifen ist besonders interessant, da zu einer Schleife ein Header erzeugt wird (im Beispiel in BB_41) und am Ende des Rumpfes werden die restlichen Instruktionen abgelegt, die Teil der Schleife sind (im Beispiel BB_47). Die Zeileninformation zur Schleife kommt doppelt im Assembler-File vor. Der Schleifenkopf gehört zum CodeBlock außerhalb der Schleife und die Schleifenkontrolle zum CodeBlock, der dem Schleifenrumpf zugeordnet wird.

Die Code-Generierung des encc ist so vorhersagbar, dass davon ausgegangen werden kann, dass im Assemblerfile zuerst der Schleifenkopf und später die Instruktionen zur Schleifenkontrolle generiert werden. Bei der ersten Übergabe eines LineSet-Objektes an ein LoopNest wird der CodeBlock außerhalb der Schleife im LineSet gespeichert und beim zweiten Aufruf der innere.

Ein ähnliches Vorgehen ist auch für komplexe C-Konstrukte, wie Switch/Case-Anweisungen notwendig, diese werden vom istf aber nicht unterstützt.

## 4.6 Das Optimierungsproblem

Die wesentliche Idee der in dieser Arbeit beschriebenen Optimierung ist die Tatsache, dass die durch Entscheidungen veränderten Ausführungshäufigkeiten den Energieverbrauch beeinflussen. Die Veränderung der Ausführungshäufigkeiten der nicht innersten Elementschleifen wird im Beispiel 4.1.2 auf Seite 39 deutlich. Wenn der Energieverbrauch möglichst exakt nachgebildet wird, kann eine bezüglich des Energieverbrauchs im Modell optimale Entscheidung getroffen werden.

Im folgenden Abschnitt wird erläutert, wie die Bezeichner von Variablen des Optimierungsproblems gewählt und nummeriert werden. Die Eingaben oder Parameter des Optimierungsproblems werden definiert. Anschließend werden die Ausgaben und Entscheidungsvariablen definiert. Die Entscheidungsvariablen und Parameter werden verwendet, um formal und anhand von Beispielen die einzelnen Teilformeln des Optimierungsproblems herzuleiten.

### 4.6.1 Nummerierungen

Für die formale Beschreibung des Optimierungsproblems werden Variablen für Entscheidungen, Größen oder Parameter eingeführt. Die Namen der Variablen bestehen aus Buchstaben. Großbuchstaben werden für Entscheidungen verwendet. Kleinbuchstabige Bezeichner repräsentieren Parameter, die aus dem zu optimierenden Programm abgeleitet werden, oder Größen, die aus Zwischenrechnungen gewonnen werden und einen Namen erhalten sollen.

An Variablen werden Indizes angehängt, um sie zu unterscheiden. Der Index soll, wenn es sinnvoll ist, auch die Position im Programm widerspiegeln, an der die Variable logisch vorkommt. Wenn eine Position nicht sinnvoll angenommen werden kann, dann werden die Variablen von 0 beginnend durchnummeriert.

Variablen und ihre Indizes werden zur Implementierung auf Arrays mit demselben Namen abgebildet. Die Dimension des Arrays entspricht der Anzahl der Indizes der Variablen. Jede Variable besitzt immer die gleiche Anzahl Indizes. Wenn die Position im Kontext unerheblich oder eindeutig ist, können die Indizes in der Formalisierung auch weggelassen werden.

Bei der Einführung einer Variablen mit Indizes werden alle Indizes aufgeführt und durch Buchstaben abgekürzt, welche Bedeutung die Position im Index besitzt.

$f$ **Funktionsnummer** Alle Funktionen des Programms werden beginnend ab 1 durchnummeriert.

$n$ **Nestnummer** Alle Schleifennester in einer Funktion werden beginnend ab 1 in Programmreihenfolge durchnummeriert.

$l$ **Schleifennummer** Alle Schleifen eines Nestes werden beginnend bei 1 von außen nach innen durchnummeriert.

$c$ **Array-Nummer** Die Nummern werden fortlaufend für die Array-Zugriffe vergeben, die die gleiche Zugriffsfunktion besitzen, also dasselbe Element lesen oder schreiben.

Es existieren drei Mengen mit Array-Nummern. In die ersten zwei Mengen, $C_r$ und $C_w$, wird für Array-Zugriffe, die sich eine Kopierfunktion teilen können, ein Repräsentant eingefügt. Als Repräsentant wird jeweils der Array-Zugriff ausgewählt, der den größten konstanten Anteil in der affinen Zugriffsfunktion besitzt. Es wird über $a_0$ der Formel 4.11 auf Seite 73 maximiert. Dies bedeutet: Arrays, die einen gleichen Namen aber unterschiedliche Zugriffsfunktion haben, deren Differenz nicht konstant ist, werden nicht zugelassen. Dies wurde schon in Abschnitt 4.2 gefordert.

In der Menge $C_r$ sind Array-Nummern gespeichert, bei deren Zugriffen es sich um lesende Zugriffe handelt. In der Menge $C_w$ sind Array-Nummern gespeichert, bei deren Zugriffen es sich um schreibende Zugriffe handelt. Die Elemente der Mengen korrespondieren zu den einzufügenden Kopierfunktionen.

Eine dritte Menge mit Array-Nummern fasst Arrays zusammen, die sich auf den gleichen Arbeitsbereich im Scratchpad-Speicher beziehen. In der Menge $C_u$ wird für jedes deklarierte Array, auf das in einer Schleife zugegriffen wird, ein Repräsentant eingefügt.

$d$ **Dimension** Die $d$-te Dimension eines Arrays. Die Dimensionen werden beginnend bei 1 durchnummeriert.

$i$ **Koeffizient** Der $i$-te Koeffizient der affinen Zugriffsfunktion. Der 0-te Koeffizient ist der konstante Anteil der Funktion. Die Sortierung der Indizes entspricht der Sortierung der Schleifen, wobei die äußerste Schleife zuerst betrachtet wird, also der 1. Koeffizient zur Induktionsvariablen der äußersten Schleife gehört.

$b$ **Basisblocknummer** Basisblöcke werden beginnend bei 0 durchnummeriert.

$m$ **Speichernummer** 0 steht für den Hauptspeicher und 1 steht für den Scratchpad-Speicher. Wenn weitere unterschiedliche Speicher zur Verfügung stehen, können weitere Werte möglich sein. Weitere Programmmodifikationen wären dann aber erforderlich. Die Anzahl der verfügbaren Speicher ist eine Konstante, die in `mc` gespeichert wird.

Einige Koeffizienten werden ab 0 durchnummeriert und andere ab 1, dies ist in der Regel unerwünscht. Der Fall 0 wird an manchen Stellen als Spezialfall gebraucht und es hat den Vorteil, dass die Zuordnung zwischen einzelnen Koeffizienten vereinfacht wird. Dass Basisblocknummern und Speichernummern bei 0 beginnen, liegt daran, dass sie nicht im Zusammenhang mit Schleifennummern verwendet werden und hier die 0 nicht als Sonderfall benutzt wird.

### 4.6.2 Eingaben

Das Optimierungsproblem kann auf folgende Parameter zurückgreifen. Die Parameter bilden die Eingaben des Optimierungsproblems. Zuerst werden Informationen zu Basisblöcken, dann zu den Schleifen und schließlich Informationen zu den Arrays bereitgestellt.

$bbc \in \mathbb{N}$ ist die Anzahl der Basisblöcke des Originalprogramms. *bbc* steht für **B**aisc**B**lock **C**ount.

$bl_b \in \{(f \times n \times l)\}$ gibt an, dass der Code des Basisblocks $b$ des Originalprogramms genau so oft ausgeführt wird wie der Rumpf der $l$-ten Schleife im $n$-ten Schleifennest der $f$-ten Funktion. *bl* steht für **B**asicblock **L**ocation.

$bs_b \in \mathbb{N}$ ist die Größe des $b$-ten Basisblocks des Originalprogramms in Bytes. *bs* steht für **B**asicblock **S**ize.

$bj_b \in \mathbb{N}$ Jeder Basisblock, bis auf den letzten, hat genau einen Nachfolger im Programmtext. Ein Basisblock kann als letzten Befehl einen (bedingten) Sprung enthalten. Die Nummer des Basisblocks des Sprungziels wird in $bj$ gespeichert. Wenn der letzte Befehl kein (bedingter) Sprung ist, wird die Nummer des Basisblocks selbst gespeichert. Es gilt dann $bj_b = b$. $bj$ steht für **B**asisblock **J**ump.

$ru_b \in \mathbb{N}$ ist die Anzahl der Register, die im $b$-ten Basisblock des Originalprogramms schon verwendet worden sind. In der Implementierung wird das Array **R**egs**U**sed verwendet.

$bc_{m,b} \in \mathbb{F}$ gibt die Kosten einer Ausführung des $b$-ten Basisblocks des Originalprogramms im $m$-ten Speicher an. *bc* steht für **B**asicblock **C**ost. $\mathbb{F}$ steht für die Menge der Fließkommazahlen.

$ec_{f,n,l} \in \mathbb{N}$ ist die Differenz zwischen oberer und unterer Schranke der $l$-ten Schleife im $n$-ten Nest in der $f$-ten Funktion. *ec* steht für **E**xecution **C**ount.

$lt_{f,n,l} \in \mathbb{B}$ gibt an, ob eine Schleife getiled werden soll. *lt* steht für **L**oop should be **T**iled. Diese Eingabe muss nicht im Optimierungsproblem verarbeitet werden, wenn ausschließlich teilbare Schleifennester in das Optimierungsproblem eingefügt werden.

$li_{f,n,l} \in \mathbb{N}$ gibt an, wieviele Befehle in der $l$-ten Schleife im $n$-ten Nest der $f$-ten Funktion durch ein if vor dem Tiling geschützt werden müssen. *li* steht für **L**oop needs **I**f.

Die Implementierung zählt in nicht innersten Schleifen des Originalprogramms die Anweisungen, die nicht Schleifen sind. Es wird davon ausgegangen, dass diese Befehle durch das Tiling nicht häufiger ausgeführt werden dürfen.

$ibb_{f,n,l} \in \mathbb{N}$ ist die Nummer des Basisblocks, der die Initialisierung der $l$-ten Schleife im $n$-ten Nest der $f$-ten Funktion enthält. *ibb* steht für **I**nit **B**asic**B**lock.

$cbb_{f,n,l} \in \mathbb{N}$ ist die Nummer des Basisblocks, der die Schleifenkontrolle und Addition der $l$-ten Schleife im $n$-ten Nest der $f$-ten Funktion enthält. *cbb* steht für **C**ontinue **B**asic**B**lock.

$fc \in \mathbb{N}$ ist die Anzahl der Funktionen im Programm. $fc$ steht für **F**unction **C**ount.

$nc_f \in \mathbb{N}$ ist die Anzahl der Nester in der $f$-ten Funktion. *nc* steht für Loop**N**est **C**ount.

$ld_{f,n} \in \mathbb{N}$ ist die Anzahl der Schleifen im $n$-ten Nest der $f$-ten Funktion. *ld* steht für **L**oopnest**D**epth.

$ac \in \mathbb{N}$ ist die Anzahl der Array-Zugriffe mit unterschiedlicher Zugriffsfunktion. *ac* steht für **A**rray **C**ount.

$ak_{c,d,i} \in \mathbb{N}$ ist der Koeffizient der Zugriffsfunktion zum Array mit der Nummer $c$ in der $d$-ten Dimension. Ist $i = 0$, dann handelt es sich um den konstanten Anteil. Ansonsten ist es der Koeffizient der $i$-ten Schleife im Originalprogramm. *ak* steht für **A**rray **K**oeffizient.

$ad_c \in \mathbb{N}$ ist die Dimension des Arrays mit der Nummer c. *ad* steht für **A**rray **D**imension.

$accr_c \subset \{(f \times n \times l \times b)\}$ Gibt an, dass das Array $c$ in der $l$-ten Schleife des $n$-ten Nestes der $f$-ten Funktion gelesen wird. Dieser Zugriff findet im Basisblock $b$ statt. *accr* Steht für **A**rray A**cc**ess is **R**ead.

*accr* bildet eine Menge, die die Tupel $(f \times n \times l \times b)$ enthält. Beim Aufruf der set-Methode wird eine boolesche Variable $Su_{f,n,c}$ auf Wahr (1) gesetzt, die angibt, dass in dem $n$-ten Nest der $f$-ten Funktion auf das Array $c$ zugegriffen wird. *Su* steht für **S**cratchpad-Memory **U**sage possible.

$accw_c \subset \{(f \times n \times l \times b)\}$ bildet die Analogie zu *accr*, es handelt sich aber um einen Schreibzugriff.

$SPMAvail \in \mathbb{N}_0$ ist eine Konstante, die angibt, wieviel Scratchpad-Speicher zur Verfügung steht. Dies kann nicht aus dem Programm ermittelt werden.

Bis auf die Ausnahmen (*accr*, *accw*) speichern die set-Modethoden die Werte in Arrays, deren Dimension der Anzahl der Indizes entspricht. Bei *accr* und *accw* werden die vier Werte $f$, $n$, $l$ und $b$ gemeinsam als Struktur in einer Menge gespeichert. Für $bl_b$ ist ein Element des Arrays ein Tripel aus der Postion im Code.

### 4.6.3 Ausgaben

Optimiert werden soll der Gesamtenergieverbrauch einer Ausführung des zu übersetztenden Programms. Scratchpad-Speicher haben geringere Kosten pro Zugriff als der Hauptspeicher. Wenn der Scratchpad-Speicher nicht alle Daten gleichzeitig aufnehmen kann, muss eine Auswahl getroffen werden, welche Daten in den Scratchpad-Speicher verlegt werden.

Das Optimierungsproblem soll die Auswahl treffen, welche Speicherobjekte in den Scratchpad-Speicher verlegt werden. In dieser Arbeit sollen dies der Programm-Code und Arrays sein. Die Tiling-Faktoren der Schleifen müssen gewählt werden. Da Tiling den Arbeitsbereich, auf den in einer Schleife zugegriffen wird, verändern kann, soll der Arbeitsbereich dynamisch in den Scratchpad-Speicher verlegt werden. Die folgenden drei Ausgaben sollen vom Optimierungsproblem explizit festgelegt werden: Die

**Tiling-Faktoren** der Schleifen einer jeden Schleife sollen festgelegt werden. Der Tiling-Faktor kann Einfluss auf den Arbeitsbereich eines Arrays haben. Der Tiling-Faktor bestimmt auch, wie oft die Tiling-Schleife ausgeführt wird und damit wieviel Overhead durch das Tiling anfällt. Der Tiling-Faktor soll mit $T_{f,n,l} \in \mathbb{N}$ bezeichnet werden. $T$ steht für **T**iling-Factor.

**Speicher-Belegung** des Scratchpad-Speichers soll explizit bestimmt werden. Für jedes Array muss entschieden werden, ob es im Scratchpad-Speicher liegen soll. Die Entscheidung, ob ein Array in den Scratchpad-Speicher verlegt wird, soll jeweils in einer booleschen Variablen $S_{f,n,c}$ gespeichert werden, $S = 1 \Leftrightarrow$ „Verwendung des Scratchpad-Speichers für das Array“. $S$ steht für „use **S**cratchpad Memory“

**Index-Verwendung** gibt an, ob ein Index in einer speziellen Zugriffsfunktion im Original oder getiled verwendet werden soll. Diese Entscheidung und die Tiling-Faktoren bestimmen, wie groß der Arbeitsbereich eines Arrays während der Ausführung der Elementschleifen ist. Diese Entscheidung wird in einer booleschen Variablen $O_{c,d,i}$ gespeichert. $O = 1 \Leftrightarrow$ „Verwendung des Original-Indexes“, d.h die Summe aus Tiling-Index und Element-Index wird verwendet, wie es in Abschnitt 4.1.3 vorgestellt wird. $O$ steht für „use **O**riginal Index“.

Die Variablen zur Entscheidung der Speicher-Belegung (S) und der Index-Verwendung (O) können nicht immer beliebig belegt werden. Wenn ein Array nicht in den Scratchpad-Speichers verschoben wird, muss

der Original-Index verwendet werden, um immer die richtige Zugriffsfunktion zu erzeugen. Es soll die Beziehung: $S \vee O = 1$ gelten.

Diese Entscheidungsvariablen müssen mehrfach vorkommen können und werden daher mit Indizes unterschieden oder als Arrays implementiert. Die Nummerierung wurde in Abschnitt 4.6.1 erläutert.

Explizit entschieden wird natürlich auch, welcher Code in den Scratchpad-Speicher verlegt werden soll. Die Verlegung selbst übernimmt der encc, das Modell muss die Belegung aber trotzdem berücksichtigen, da der Scratchpad-Speicher endlich ist. Entschieden wird welche

**Basisblöcke** in den Scratchpad-Speicher verschoben werden. Dies wird durch die binäre Entscheidungsvariable $B_b$ bestimmt. $B = 1 \Leftrightarrow$ „der Basisblock liegt im Scratchpad-Speicher“. $B$ steht für „**B**asisblock is in Scratchpad Memory“.

**Code-Fragmente** in den Scratchpad-Speicher verschoben werden. Durch Tiling entsteht Overhead. Für Tiling-Schleifen werden neue Basisblöcke angelegt. Für jede Tiling-Schleife wird ein Block angenommen. Dazu wird eine binäre Entscheidungsvariable $L_{f,n,l}$ eingeführt. $L = 1 \Leftrightarrow$ „Der Code liegt im Scratchpad-Speicher“. $L$ steht für „**L**ocation of Code is Scratchpad Memory“.

Für die Kopierfunktionen muss auch entschieden werden, wo sie abgelegt werden. Dafür wird die Variable $L_d$, $d \in D$ verwendet. $D$ ist die Menge der Dimensionen der Arrays, die im Programm vorkommen.

Indirekt sind durch diese expliziten Entscheidungen aber auch implizite Entscheidungen getroffen worden. Die folgenden drei impliziten Entscheidungen sind von besonderer Bedeutung:

**Kopierposition** Durch die Entscheidung, welche Index-Variablen eines Array-Zugriffs original oder getiled verwendet werden sollen, wird auch festgelegt, welche Kopierposition verwendet werden soll.

**Größe des Arbeitsbereichs** Durch die Wahl der Tiling-Faktoren ($T$), ob ein Original-Index verwendet ($O$) und ob ein Array in den Scratchpad-Speicher verlegt ($S$) werden soll, wird der Arbeitsbereich der Arrays und damit der zu reservierende Platz im Scratchpad-Speicher bestimmt.

**Minimum einfügen** Wenn der Tiling-Faktor kein Teiler der Differenz zwischen oberer und unterer Schranke ist, dann muss ein Minimum gebildet werden.

### 4.6.4 Modellierung des Energieverbrauchs

Durch das Optimierungsproblem wird entschieden, wie der Code zu transformieren ist. Diese Entscheidung wird getroffen, indem der Energievebrauch des transformierten Programms in Abhängigkiet der Eingaben vorhergesagt wird.

Die genauen Kosten können nur dann ermittelt werden, wenn der Code transformiert und ausgeführt wird. Dies ist aber ein langwieriger Prozess, der durch die Vereinfachungen des Modells beschleunigt wird.

Die Grundkosten des Programms werden aus den Informationen der Basisblöcke des Originalprogramms gewonnen. Um die Energiekosten zu modellieren, die durch die Code-Veränderung auftreten, wird anhand von Beispielprogrammen untersucht, wie sich die Veränderungen auswirken. Die Veränderungen werden in Klassen unterteilt und repräsentieren meist eine Gruppe von mehreren Befehlen, die wahrscheinlich in dieser Kombination auftreten.

Nicht berücksichtigt werden die genaue Erzeugung der Konstanten und Optimierungen, die im Code möglich sind, dazu zählt die Entfernung gemeinsamer Unterausdrücke. Die Auswirkungen dieser notwendigen Vereinfachungen werden im Abschnitt 5.2.3 näher beschrieben. Wie in Abschnitt 4.1.4 erläutert und in Abbildung 4.2 zu erkennen, sollten Veränderungen des Codes in den innersten Schleifen besonderes berücksichtigt werden.

Die folgenden Code-Klassen werden zur Modellierung des Energieverbrauchs des modifizierten Programms verwendet. Die Kosten werden in zwei Versionen vorgehalten. Der Code kann im Scratchpad-Speicher oder Hauptspeicher liegen, die Speichernummer *m* wird zur Unterscheidung verwendet. Es kann in einer Erweiterung so auch mehr als ein Scratchpad-Speicher betrachtet werden. Für alle Code-Fragmente wird auch die Größe berechnet. Im Namen der Konstanten wird für Energie ein *e* und für die Größe (**S**ize) ein *s* verwendet. Für die Code-Größe ist die Position unerheblich, so dass für *s* der erste Index der Name der Code-Klasse ist und die Speichernummer weggelassen wird.

$e_{m,LOOPINIT\_ELEMENT} \in \mathbb{F}$ Die Initialisierung einer Elementschleife lädt in der getilten Version eine 0 in die Kontrollvariable.

$e_{m,LOOPCONT\_ELEMENT} \in \mathbb{F}$ Die Schleifenkontrolle der Elementschleife muss prüfen, ob das Tile schon abgearbeitet ist.

$e_{m,LOOPINIT\_TILE} \in \mathbb{F}$ Die Initialisierung der Tiling-Schleife ist dieselbe wie im Originalprogramm, der Code ist aber nicht bekannt. Es wird angenommen, dass eine Konstante geladen wird, die als Immediate gespeichert werden kann.

$e_{m,LOOPCONT\_TILE} \in \mathbb{F}$ Die Schleifenkontrolle der Tiling-Schleife hat eine Schrittweite, die wesentlich größer als 1 ist. Auch hier wird angenommen, dass es sich um eine Konstante handelt, die noch als Immediate gespeichert werden kann.

$e_{m,MINIMUM} \in \mathbb{F}$ Die Minimumsbildung besitzt zwei Ausführungswege mit unterschiedlichem Energieverbrauch. Es wird hier der häufigere angenommen. Wenn die Tiling-Schleife mehr als zweimal ausgeführt wird, wird der Code zum Laden des Restes nur einmal ausgeführt. Für die Berechnung der Code-Größe wird die Summe der beiden Möglichkeiten verwendet.

Für die Minimumsbildung und die Kosten der Tiling-Schleifen sind grobe Schätzungen ausreichend, weil diese Kosten nur mit einem kleinen Teil in das Endergebnis eingehen werden.

$e_{m,IF} \in \mathbb{F}$ Befehle in nicht innersten Schleifen sollen so oft wie im Originalprogramm ausgeführt werden. Der Test, ob dies gerade eine Iteration ist, die auch im Originalprogramm stattgefunden hätte, wird durch eine Bedingung abgefragt. Für jede Tiling-Schleife wird jeweils eine Bedingung eingefügt, dies wird von dieser Klasse repräsentiert.

$e_{m,ADD} \in \mathbb{F}$ bildet die Kosten einer Addition zwischen zwei Registern ab. Eine Addition tritt auf, wenn der Original-Index einer getilten Schleife verwendet werden soll.

$e_{m,COPY\_INIT,d} \in \mathbb{F}$ Zum Kopieren eines Teil-Arrays muss die Kopierfunktion aufgerufen werden. Für zwei verschiedene Dimensionen ist die Anzahl der notwendigen Parameter unterschiedlich.

$e_{m,COPY\_INIT\_FUNCTION,d} \in \mathbb{F}$ Die Kopierfunktion selbst muss für jede Dimension nur einmal existieren, daher werden die Befehle, die in der Kopierfunktion nur einmal ausgeführt werden, in dieser Kostenklasse zusammengefasst.

$e_{m,COPY\_CALL,d} \in \mathbb{F}$ Dies ist der mehrfach im Programm vorkommende Teil zum Aufruf der Kopierfunktion. Die Initialisierungskosten setzen sich aus einem festen Teil innerhalb der Funktion und einem beim Aufruf der Kopierfunktion zusammen:

$$e_{m,COPY\_INIT,d} = e_{m,COPY\_INIT\_FUNCTION,d} + e_{m,COPY\_CALL,d} \tag{4.5}$$

$e_{m,COPY\_LOOP} \in \mathbb{F}$ Innerhalb der Kopierfunktion existiert eine innerste Schleife, die das eigentliche Kopieren übernimmt. Alle Befehle, die innerhalb diese Schleife ausgeführt werden, sind hier erfasst. In diesem Fall handelt es sich um einen lesenden Zugriff im Hauptspeicher.

$e_{m,COPY\_LOOPW} \in \mathbb{F}$ Wenn es sich um einen schreibenden Zugriff auf den Hauptspeicher handelt, wird diese Konstante verwendet. Es wird Lesen und Schreiben unterschieden, weil das Kopieren einen großen Anteil hat; diese beiden Klassen bilden die innersten Schleifen der Kopierfunktion nach.

$e_{m,COPY\_CONST} \in \mathbb{F}$ Wird ein Array mit mehr als einer Dimension kopiert, so existieren weitere Schleifen, die dafür sorgen, dass Abschnitte in den anderen Dimensionen ebenfalls kopiert werden. Diese Schleifen sind für alle weiteren Dimensionen gleich und können daher in einer Kostenklasse zusammengefasst werden.

$e_{m,JUMP} \in \mathbb{F}$ Wenn zwei hintereinander liegende Basisblöcke nicht in denselben Speicher gelegt werden können, muss ein zusätzlicher Sprung eingefügt werden.

$e_{m,GAIN\_WRITE,m'} \in \mathbb{F}$ Die Kosten des Programms ohne die Modifikationen werden über die mit der Ausführungshäufigkeit gewichteten Summe der Basisblockkosten ($bc_{m,b}$) berechnet. In den Basiskosten sind alle Array-Zugriffe des Originalprogramms enthalten. Kommt in einem Basisblock ein schreibender Array-Zugriff vor, der auf Daten im Scratchpad-Speicher zugreift, dann stimmen die Kosten für diesen Basisblock nicht. Die Kosten können durch gewichtete Addition der Zugriffshäufigkeit des betroffenen Arrays dieser Kostenklasse korrigiert werden.

$m$ ist die Speichernummer des Programms und $m'$ ist die Speichernummer der Daten. Wenn $m' = 0$ ist, also die Daten im Hauptspeicher liegen, ist die Differenz der Kosten 0.

$e_{m,GAIN\_READ,m'} \in \mathbb{F}$ Auch lesende Array-Zugriffe im Scratchpad-Speicher müssen korrigiert werden.

$e_{m,SPILL\_OUT} \in \mathbb{F}$ Alle bisherigen Kostenklassen gehen davon aus, dass kein Spilling notwendig ist. Ist Spilling notwendig, so wird ein zusätzlicher Hauptspeicherzugriff notwendig, es kann aber kein Register oder anderer Befehl eingespart werden. $e_{m,SPILL\_OUT}$ berücksichtigt das Spillen in den Hauptspeicher. Es wurde also gerade ein gespilltes Register beschrieben.

$e_{m,SPILL\_IN} \in \mathbb{F}$ Wenn ein gespilltes Register gelesen wird, muss diese Kostenklasse berücksichtigt werden.

Bei einer solchen Liste stellt sich die Frage, ob sie vollständig ist und ob die Detaillierung fein genug ist. Die Vollständigkeit ist gegeben, weil mit diesen Klassen und den folgenden Formeln die durchgeführten Änderungen am Code abgedeckt werden. Die durchführbaren Code-Modifikationen sind durch das implementierte Frontend beschränkt. Der zusätzliche Overhead wurde in Abschnitt 4.1.4 schon betrachtet. Die Betrachtungen der Ergebnisse legen nahe, dass eine sinnvolle Auswahl und Abstraktion getroffen wurde.

Im folgenden Beispiel wird anhand des Minimums gezeigt, wie ein Beispiel Code-Fragment einer Kostenklasse aussehen kann und welche Kosten berücksichtigt werden.

**Beispiel 4.6.1: Kosten einer Minimumsbildung**

Ist der Tiling-Faktor kein Teiler der Ausführungshäufigkeit der Schleife im Originalprogramm, muss eine Minimumsbildung eingefügt werden. In der folgenden Anweisung wird geprüft, ob schon die letzte Iteration der Tiling-Schleife erreicht ist. Das Ergebnis der Zuweisung bestimmt, wie oft die Elementschleife ausgeführt wird. Wenn ja, dann wird nur noch der Rest gespeichert, sonst der Tiling-Faktor. In diesem Fall ist der Tiling-Faktor 67. Zu folgender Minimumsbildung

```
iM = 441<iT+67 ? 441-iT : 67;
```

wird folgender Assembler-Code generiert.

```
        MOV r5,#0           ;***
        ...
LL13_0
        MOV r0,#220
        ADD r0,r0,r0        ; **
        ADD r1,r0,#1        ; **
        MOV r0,r5
        ADD r0,#67
```

```
        CMP r1,r0
        BLT LL3_0
_M_6
        MOV r4,#67
        B LL4_0
LL3_0
        MOV r0,#220
        ADD r0,r0,r0        ; **
        ADD r0,r0,#1        ; **
        SUB r4,r0,r5
LL4_0
```

Der Tiling-Faktor ist 67. Die Ausführungshäufigkeit der Tiling-Schleife ist 6, der Basisblock `LL3_0` wird dabei nur einmal ausgeführt. Wird dieses Code-Fragment zur Bestimmung der Kosten für die MINIMUM Klasse herangezogen, dann werden vier Befehle (**) ignoriert, da sie auf die Bildung großer Konstanten zurückgeführt werden. Die erste Zeile (***) wird ebenfalls ignoriert und dient nur dem Verständnis. Es wird angenommen, dass die Initialisierung eines Registers mit 0 so häufig ist, dass eine erneute Initialisierung nicht erneut vorkommen muss.

Zur Code-Größe tragen 9 Befehle bei ($s_{m,MINIMUM}$). Für die Berechnung der Energiekosten werden 7 Befehle berücksichtigt. Der Basisblock LL3_0 wird für die Berechnung der Energie nicht berücksichtigt, da er seltener ausgeführt wird.

#### 4.6.4.1 Basisblöcke

In diesem und den folgenden Abschnitten werden die Zwischenergebnisse definiert und schließlich zur Zielfunktion in Abschnitt 4.6.4.5 zusammengesetzt. Es wird jeweils ein Teil des Energieverbrauchs und ein Teil der Belegung des Scratchpad-Speichers berechnet.

Ein wesentliches Merkmal der Betrachtungen ist die Ausführungshäufigkeit. Ausführungshäufigkeiten eines einzelnen Rumpfes im Originalnest, also die Differenzen zwischen oberer und unterer Schranke der Schleifen, sind bekannt und definiert durch $ec_{f,n,l}$. Analog dazu wird die Ausführungshäufigkeit im Originalnest $oex_{f,n,l}$ (**O**riginal **EX**ecution) definiert.

Ausführungshäufigkeiten lassen sich rekursiv durch die Ausführungshäufigkeit der Schleife selbst und ihrer Umgebung definieren.

$$oex_{f,n,0} = 1 \tag{4.6}$$

$$oex_{f,n,l} = oex_{f,n,l-1} \cdot ec_{f,n,l} \tag{4.7}$$

Die Initialisierung der äußersten Schleife wird einmal ausgeführt. Die Ausführungshäufigkeit des Schleifenrumpfes ist das Produkt aus Ausführungshäufigkeit ($ec_{f,n,l}$) der Schleife selbst mit der Ausführungshäufigkeit ihrer Initialisierung ($oex_{f,n,l-1}$), die sich in einer äußeren Ebene ($l-1$) befindet.

Die Teilformeln für Basisblöcke sind sehr übersichtlich, daher wird mit ihnen begonnen.

Die Kosten der Ausführung des Originalprogramms bilden die Grundkosten des Programms. Der Overhead und der Gewinn, der in den folgenden weiteren Teilformeln hergeleitet wird, kann zu diesen Grundkosten addiert werden und ergibt in der Summe die Kosten des gesamten modifizierten Programms.

Zur Berechnung der Summe der Grundkosten müssen die Basisblöcke entsprechend ihrer Ausführungshäufigkeit im Originalprogramm und der Speicherposition gewichtet aufaddiert werden. Zusätzliche Sprünge

sind zu berücksichtigen, wenn nachfolgende Basisblöcke nicht im selben Speicher liegen. Dieser Zusammenhang kann durch folgende Formel ausgedrückt werden:

$$E_{BB} = \sum_{b=0}^{bbc} oex_{bl_b} \cdot bc_{B_b,b} \tag{4.8}$$

$$+ \sum_{b=0}^{bbc-1} oex_{bl_b} \cdot e_{B_b,JUMP} \cdot (B_{b+1} \neq B_b) \tag{4.9}$$

$$+ \sum_{b=0}^{bbc} oex_{bl_b} \cdot e_{B_b,JUMP} \cdot (B_{bj_b} \neq B_b) \tag{4.10}$$

Die erste Summe berechnet die Grundkosten des Originalprogramms, indem der Energieverbrauch der Basisblöcke mit der Ausführungshäufigkeit im Originalprogramm multipliziert werden. Die zweite Summe berechnet die Kosten, die entstehen, wenn aufeinanderfolgende Basisblöcke des Originalprogramms nicht im selben Speicher abgelegt werden können. Die Kosten für einen Sprung müssen mit der Ausführungshäufigkeit des Basisblocks multipliziert werden. Wird ein Basisblock mit einem bedingten Sprung beendet, der nicht im gleichen Speicher ist, wird auch ein zusätzlicher Sprung eingefügt.

Für alle bedingten Sprünge im Programm wird angenommen, dass sie immer ausgeführt werden. Da keine Informationen gewonnen werden, wie wahrscheinlich einzelne Sprünge sind, ist dies eine Überschätzung der Kosten.

In dieser Arbeit werden hauptsächlich Schleifen betrachtet. Es ist also sehr wahrscheinlich, dass es sich bei den bedingten Sprüngen, die im Teil 4.10 betrachtet werden, um Rücksprünge an den Anfang der Schleife handelt. Diese Sprünge werden sehr häufig genommen (taken). Weil diese Sprünge aber genauso oft ausgeführt werden wie das Sprungziel, ist es wahrscheinlich, dass der encc die Basisblöcke, die die Befehle enthalten, in denselben Speicher ablegt. Die starke Überschätzung der zusätzlichen Kosten für bedingte Sprünge scheint ungerechtfertigt und wurde daher optional implementiert.

Die Ergebnisse im Abschnitt 5 wurden ohne die Teilformel 4.10 gewonnen. Zum Vergleich sind in Abbildung A.4 auf Seite 143 die Verbesserungen beim dynamischen Profiling mit der Teilformel angegeben. Die Berechnung zusätzlicher Sprünge ist im Modell sehr ungenau, da der Schleifen-Overhead in einem Basisblock zusammengefasst wird. Daher hat eine genauere Betrachtung der zusätzlichen Sprünge zwischen den Basisblöcken nicht immer einen positiven Einfluss.

Der Ausdruck $(B_{b+1} \neq B_b)$ stellt sicher, dass nur dann, wenn die Speicher der Basisblöcke $b-1$ und $b$ nicht gleich sind, eine Sprunginstruktion eingefügt und berücksichtigt wird. $\neq$ und $=$ sind zwei Funktionen, die infix verwendet werden. Es gilt:

$$a = b := \begin{cases} 1 & a - b = 0 \\ 0 & \text{sonst} \end{cases}$$

$$a \neq b := \begin{cases} 0 & a - b = 0 \\ 1 & \text{sonst} \end{cases}$$

In anderen Arbeiten werden an diesen Stellen Hilfsvariablen eingefügt, die entsprechend belegt werden. Während der Formalisierung wird auf diese Vorgehensweise verzichtet.

Für die Größe des Scratchpad-Speichers, der durch Basisblöcke belegt wird, gilt analog:

$$S_{1,BB} = \sum_{b=0}^{bbc} (B_b = 1) \cdot bs_b$$

$$+ \sum_{b=0}^{bbc-1} (B_b = 1) \cdot (B_{b+1} \neq B_b) \cdot s_{JUMP}$$

$$+ \sum_{b=0}^{bbc} (B_b = 1) \cdot (B_{bj_b} \neq B_b) \cdot s_{JUMP}$$

Es wird jeweils geprüft, ob der Basisblock in den Scratchpad-Speicher verschoben wurde, denn dann fallen Kosten für die Belegung des Speichers an. Für alle Basisblöcke wird geprüft, ob das (implizite) Sprungziel im gleichen Speicher ist.

#### 4.6.4.2 Schleifen

Ein wesentliches Merkmal bei der Betrachtung der Schleifen ist ihre Ausführungshäufigkeit.

Analog zur Ausführungshäufigkeit im Originalnest $oex_{f,n,l}$ wird die Ausführungshäufigkeit der Tiling-Schleifen $tex_{f,n,l}$ (**T**iling **EX**ecution) und der Elementschleifen $ex_{f,n,l}$ (element **EX**ecution) definiert.

Elementschleifen haben gegenüber den Schleifen im Originalprogramm nicht immer die gleiche Ausführungshäufigkeit. Wie in Tabelle 4.4 auf Seite 41 zu erkennen ist, wird in einem getilten Nest nur die innerste Schleife so oft wie im Originalprogramm ausgeführt. Die nicht innersten Elementschleifen werden häufiger ausgeführt. Zusätzlich werden Elementschleifen nicht in jedem Durchlauf gleich oft ausgeführt. Wenn der Tiling-Faktor die Ausführungshäufigkeit der Originalschleife nicht ganzzahlig teilt, fällt ein Rest an. Die jeweils letzte Iteration der Tiling-Schleife sorgt dafür, dass die korrespondierende Elementschleife entsprechend diesem Rest ausgeführt wird.

Der rekursive Aufbau der Formel zur Berechnung der Ausführungshäufigkeit der Tiling-Schleifen entspricht dem Aufbau zur Berechnung der Originalausführungshäufigkeit 4.7

$$\begin{aligned} tex_{f,n,0} &= 1 \\ tex_{f,n,l} &= tex_{f,n,l-1} \cdot \left( \left( ec_{f,n,l} \div T_{f,n,l} \right) + \left( \left( ec_{f,n,l} \mod T_{f,n,l} \right) \neq 0 \right) \right) \end{aligned}$$

In der Klammer des Produktes findet die Berechnung der Ausführungshäufigkeit einer Tiling-Schleife statt. Die Ausführungshäufigkeit der Originalschleifen wird durch den Tiling-Faktor ganzzahlig geteilt. Bleibt ein Rest, dann muss die Tiling-Schleife ein weiteres Mal ausgeführt werden.

**Beispiel 4.6.2: Ausführungshäufigkeit einer Tiling-Schleife**

Angenommen eine Schleife soll zwölfmal ausgeführt werden und der Tiling-Faktor soll 4 sein. Dann wird die Tiling-Schleife genau dreimal ausgeführt. Es gilt $12 \div 4 = 3$. Die angegeben Code-Teile verdeutlichen das betrachtete Programm.

```
...
for(it=0; it<12; it+=4){ //Wird jeweils dreimal ausgefuehrt
  ...
  for(i=0; i<4; i++)     //dreimal 4 Iterationen
    ...
}
...
```

Wird eine Schleife hingegen elfmal ausgeführt und soll der Tiling-Faktor 4 bleiben, dann wird die Tiling-Schleife ebenfalls dreimal ausgeführt. Nur die jeweils letzte Ausführung der Elementschleife ist um 1 verringert. Während der ersten beiden Iterationen der Tiling-Schleife wird die Elementschleife zweimal mit vier Durchläufen ausgeführt ($11 \div 4 = 2$). Da der Tiling-Faktor einen Rest lässt ($11 \mod 4 = 3$) muss die Tiling-Schleife ein weiteres Mal ausgeführt werden.

```
...
for(it=0; it<11; it+=4){ //Wird jeweils dreimal ausgefuehrt
  im= it<7 ?  4 : 3;
  ...
  for(i=0; i<im; i++)    //zweimal 4 und einmal 3 Iterationen
    ...
```

```
}
...
```

Die Berechnung der Ausführungshäufigkeit der Elementschleifen teilt sich in einen trivialen und einen anspruchsvollen Teil.

Die Ausführungshäufigkeit der Elementschleifen im getilten Programm ist für die äußerste Elementschleife die Ausführungshäufigkeit des Schleifenrumpfes der innersten Tiling-Schleife. Für die innerste Elementschleife ist sie gleich der Ausführungshäufigkeit im Originalprogramm:

$$\begin{aligned} ex_{f,n,0} &= tex_{f,n,ld_{f,n}} \\ ex_{f,n,ld_{f,n}} &= oex_{f,n,l} \end{aligned}$$

Anspruchsvoll ist die Berechnung der Ausführungshäufigkeit von inneren Elementschleifen. Die Elementschleifen bilden Paare mit ihren Tiling-Schleifen. Dies ergibt sich daraus, dass Tiling die Kombination von Strip-Mining und dem Vertauschen von Schleifen ist, wie in 2.4.2 beschrieben. Innerhalb des Paares, ohne dass Schleifen um das Paar herum getauscht werden, ist die Ausführungshäufigkeit wie im Originalprogramm. Die Ausführungshäufigkeiten von inneren Elementschleifen erhöht sich, wenn die Tiling-Schleifen von weiter innen liegenden Schleifen über die Elementschleife hinweg vertauscht werden. Tiling-Schleifen, die zwischen der korrespondierenden Tiling-Schleife und den Elementschleifen liegen, sorgen dafür, dass der Rumpf der Elementschleife häufiger als im Originalprogramm ausgeführt wird. Es ist die Ausführungshäufigkeit von diesen Tiling-Schleifen maßgeblich, die zwischen der Tiling-Schleife und der dazugehörigen Elementschleife liegen. Diese zusätzlichen Ausführungen werden mit *rtex* (**R**est **T**iling **EX**ecution) bezeichnet und sind analog zu *tex* wie folgt definiert:

$$\begin{aligned} rtex_{f,n,ld_{f,n}+1} &= 1 \\ rtex_{f,n,l} &= rtex_{f,n,l+1} \cdot \left( \left( ec_{f,n,l} \div T_{f,n,l} \right) + \left( \left( ec_{f,n,l} \mod T_{f,n,l} \right) \neq 0 \right) \right) \end{aligned}$$

Der wesentliche Unterschied ist, dass jetzt nicht die äußeren Schleifen betrachtet werden, sondern von einer Tiling-Schleife aus alle weiter innen liegenden Tiling-Schleifen. Daher ist auch die Richtung der Rekursion $+1$ und es wird nach Betrachtung der innersten Tiling-Schleife abgebrochen. Der Ausdruck in der Klammer der Produkts ist für *tex* und *rtex* identisch.

Mit diesem Zwischenergebnis lässt sich die Formel der Ausführungshäufigkeit in den Elementschleifen des getilten Nestes vervollständigen:

$$ex_{f,n,l} = rtex_{f,n,l+1} \cdot oex_{f,n,l}$$

Die Ausführungshäufigkeit einer inneren Elementschleife ist das Produkt aus der Originalausführungshäufigkeit und der Ausführungshäufigkeit der Tiling-Schleifen, die zwischen der Tiling-Schleife und den Elementschleifen liegen.

**Beispiel 4.6.3: Ausführungshäufigkeit von Elementschleifen**

Es ist ein Nest gegeben, das aus drei Schleifen besteht, die jeweils elfmal ausgeführt werden. Alle Schleifen werden mit dem Tiling-Faktor 4 getiled. Es ergeben sich folgende Ausführungs-

häufigkeiten, wobei nur die Nummern der Schleifen betrachtet werden:

$$\begin{aligned}
tex_0 &= 1 \\
tex_1 &= 1 \cdot ((11 \div 4) + ((11 mod 4) \neq 0)) = 3 \\
tex_2 &= 3 \cdot ((11 \div 4) + ((11 mod 4) \neq 0)) = 9 \\
tex_3 &= 9 \cdot ((11 \div 4) + ((11 mod 4) \neq 0)) = 27 \\
rtex_4 &= 1 \\
rtex_3 &= 1 \cdot ((11 \div 4) + ((11 mod 4) \neq 0)) = 3 \\
rtex_2 &= 3 \cdot ((11 \div 4) + ((11 mod 4) \neq 0)) = 9 \\
rtex_1 &= 9 \cdot ((11 \div 4) + ((11 mod 4) \neq 0)) = 99 \\
oex_1 &= 11 \\
oex_2 &= 121 \\
oex_3 &= 1331 \\
ex_1 &= 27 \\
ex_2 &= 3 \cdot 121 = 363 \\
ex_3 &= 1331
\end{aligned}$$

Für die nicht innersten Elementschleifen wird die Information benötigt, wie oft diese zusätzlich ausgeführt werden, dies ist die Differenz zwischen Ausführungshäufigkeit im Originalnest und dem modifizierten Nest:

$$dex_{f,n,l} = ex_{f,n,l} - oex_{f,n,l}$$

*dex* steht für **D**ifference **EX**ecution.

Die Basisblöcke werden mit der Ausführungshäufigkeit im Originalprogramm gewichtet, so dass die Kosten des Originalprogramms in der Summe schon enthalten sind. Nichtinnerste Elementschleifen können aber häufiger ausgeführt werden, dies muss zusätzlich berücksichtigt werden.

Modelliert wird dies mit Hilfe der Kostenklassen für die Initialisierung der Tiling- und Elementschleifen (LOOPINIT_TILE, LOOPINIT_ELEMENT) sowie dem Rumpf der Tiling und Elementschleifen (LOOPCONT_TILE, LOOPCONT_ELEMENT). Durch die Definition der Ausführungshäufigkeiten kann bei der Initialisierung immer der Ausdruck $l-1$ verwendet werden.

In der folgenden Formel wird für jede Schleife der anfallende Overhead berechnet. Dieser kann zu den Grundkosten addiert werden und ergibt dann die Kosten für die Ausführung des Programms. Es wird für jede Schleife nur eine Entscheidungsvariable verwendet, weil durch zusätzliche Minimumsbildungen, Bedingungen oder das Nicht-Tilen von Schleifen die Anzahl und die jeweilige direkte Nachbarschaft der daraus resultierenden Basisblöcke verändert werden. Die durch Tiling neu anfallenden Basisblöcke einzeln zu modellieren und ihre Anordnung im Speicher zu verfolgen, scheint für das Modell zu aufwändig und wird daher als Ungenauigkeit in Kauf genommen. Da die Basisblöcke außerhalb der Elementschleifen liegen, werden diese Blöcke selten im Vergleich zu den Elementschleifen ausgeführt.

$$\begin{aligned}
E_{Loop} = & \sum_{f=1}^{fc} \sum_{n=1}^{nc_f} \sum_{l=1}^{ld_{f,n}} \\
& (T_{f,n,l} \neq ec_{f,n,l}) \cdot (tex_{f,n,l-1} \cdot e_{L_{f,n,l},LOOPINIT_TILE} + tex_{f,n,l} \cdot e_{L_{f,n,l},LOOPCONT_TILE}) \\
& + (ec_{f,n,l} \mod T_{f,n,l} \neq 0) \cdot tex_{f,n,l} \cdot e_{L_{f,n,l},MINIMUM} \\
& + dex_{f,n,l-1} \cdot e_{B[ibb_{f,n,l}],LOOPINIT_ELEMENT} + dex_{f,n,l} \cdot e_{B[cbb_{f,n,l}],LOOPCONT_ELEMENT} \\
& + li_{f,n,l} \cdot ex_{f,n,l} \cdot (\sum_{l'=1}^{l} (ex_{f,n,l'} \neq T_{f,n,l'}) \cdot e_{B[cbb_{f,n,l}],IF})
\end{aligned}$$

Die erste Zeile der Formel sorgt dafür, dass alle Schleifen betrachtet werden. Die zweite Zeile betrachtet den Overhead, der durch die Tiling-Schleifen anfällt. Wenn eine Schleife getiled ist ($T_{f,n,l} \neq ec_{f,n,l}$), muss die Tiling-Schleife ausgeführt werden. Berücksichtigt wird der typische Energieverbrauch für die Initialisierung der Tiling-Schleife und die Schleifenkontrolle.

In der dritten Zeile wird berücksichtigt, dass, wenn der Tiling-Faktor die Ausführungshäufigkeit der Originalschleife nicht ganzzahlig teilt, ein Minimum gebildet werden muss.

In der vierten Zeile wird berücksichtigt, dass innere Elementschleifen durch Tiling häufiger ausgeführt werden. Da die Elementschleifen aber schon in den Grundkosten enthalten sind, muss nur die Differenz der Ausführungshäufigkeit zum Originalprogramm berücksichtigt werden. Es wird jeweils eine typische Initialisierung und Schleifenkontrolle einer Elementschleife betrachtet. Die Kosten im Originalprogramm können größer sein als die im getilten Programm. Die Elementschleifen beginnen immer bei 0, dies muss für das Originalprogramm nicht stimmen. Diese Überschätzung der Kosten wird in Kauf genommen.

Die letzte Zeile der Formel schließlich berücksichtigt die Annahme, dass Befehle der inneren Elementschleifen vor erhöhter Ausführung durch Tiling geschützt werden. Die zusätzlichen Bedingungen müssen berücksichtigt werden. Für jede getilte Schleife wird eine Bedingung benötigt. Es wird nicht geprüft, ob die Bedingung überhaupt notwendig ist. Dies kann dazu führen, dass der Energieverbrauch durch die Entfernung der Bedingung minimiert werden kann.

Der Code der Tiling-Schleifen wird vom encc in verschiedenen Basisblöcken des modifizierten Programms abgelegt. Vereinfacht wird angenommen, dass für jede Schleife ein Block generiert wird, in dem die Initialisierungen, die Minimumsbildung und die Schleifenkontrolle enthalten sind. Die Modellierung des Speicherplatzes im Scratchpad-Speicher ist durch folgende Formel gegeben:

$$
\begin{aligned}
s_{1,Loop} \quad = \quad & \sum_{f=1}^{fc} \sum_{n=1}^{nc_f} \sum_{l=1}^{ld_{f,n}} \\
& (ec_{f,n,l} \neq T_{f,n,l}) \cdot (L_{f,n,l} = 1) \cdot (s_{1,LOOPINIT\_TILE} + s_{1,LOOPCONT\_TILE}) \\
+ \quad & (ec_{f,n,l} \mod T_{f,n,l} \neq 0) \cdot s_{1,MINIMUM} \\
+ \quad & li_{f,n,l} \cdot (\sum_{l'=1}^{l} (ex_{f,n,l'} \neq T_{f,n,l'}))
\end{aligned}
$$

Es werden wieder alle Schleifen des Programms betrachtet. Nur wenn die Schleife getiled ist, muss zusätzlicher Speicherplatz für die Schleife berücksichtigt werden. Es muss die Entscheidung getroffen sein, dass der Overhead für diese Schleife im Scratchpad-Speicher liegt. Im letzten Teil dieses Produktes wird der Speicherplatz berechnet. Die Tiling-Schleife besitzt eine Initialisierung und eine Schleifenkontrolle. Wenn der Tiling-Faktor die Ausführungshäufigkeit nicht ganzzahlig teilt, muss die Minimumsbildung berücksichtigt werden. Schließlich muss auch Platz für Bedingungen eingefügt werden, wenn zusätzliche Befehle vorhanden sind.

Die Änderung der Code-Größe, die durch veränderte Initialisierung und Abbruchbedingungen der Elementschleifen entstehen, werden in dieser Formel vernachlässigt. Da für die Tiling-Schleifen nur eine einfache Initialisierung und Abbruchbedingung angenommen wird, sie aber in Wirklichkeit die Initialisierung und Abbruchbedingung des Originalprogramms verwenden, wird nur die Position des Codes nicht richtig modelliert. Es ist nicht möglich, dies genauer zu fassen, da nicht analysiert wird, welche Code-Fragmente in den Basisblöcken der Initialisierung und Schleifenkontrolle zur Schleife selbst und zum Rumpf gehören.

Die Ungenauigkeiten der modellierten Auswirkungen der Schleifentransformation vereinfachen die Analyse und das Modell. Da die Kosten der Schleifen einen kleinen Teil ausmachen, siehe Abbildung 4.2 auf Seite 40, ist dieser Fehler vertretbar.

#### 4.6.4.3 Kopieren und Array-Zugriffe

Die Modellierung der Kopierkosten und die Veränderungen, die durch die Nutzung des Scratchpad-Speichers während eines Arrays-Zugriffs geschehen, sollten möglichst exakt modelliert werden, da sie einen

sehr großen Anteil an den Gesamtkosten haben. Zunächst wird modelliert, wie die Array-Zugriffe gewertet werden müssen. Anschließend wird betrachtet, wie die Kopierkosten berechnet werden.

Die Energiekosten für Array-Zugriffe $E_{Array}$ können aufgeteilt werden in den Gewinn $E_{Array\_gain}$, die Zugriffskosten $E_{Array\_access}$ und die Kopierkosten $E_{Array\_copy}$.

$$E_{Array} = E_{Array\_gain} + E_{Array\_access} + E_{Array\_copy}$$

Der Speicherverbrauch kann zur Übersichtlichkeit auch aufgeteilt werden:

$$s_{m,Array} = s_{m,Array\_gain} + s_{m,Array\_access} + s_{m,Array\_copy} + s_{m,Array\_use}$$

Die Kosten eines Array-Zugriffs hängen davon ab, von welchem Speicher aus der Befehl ausgeführt wird und ob die Daten im Scratchpad-Speicher liegen. Der generierte Code hat, egal ob die Daten im Scratchpad-Speicher liegen oder nicht, immer dieselbe Struktur, so dass für die Optimierung des eigentlichen Zugriffs kein zusätzlicher Speicher benötigt wird.

$$s_{m,Array\_gain} = 0$$

Die Kosten für den Speicherzugriff auf den Hauptspeicher sind in den Kosten der Basisblöcke schon enthalten. Wenn die Daten aber in den Scratchpad-Speicher vorschoben wurden, entsteht ein Gewinn. In den Kostenklassen *GAIN_WRITE* und *GAIN_READ* ist der Gewinn als Diffferenz gespeichert. Die Zahlen sind negativ, so dass sie zu den Gesamtkosten addiert werden können und in der Summe die tatsächlichen Kosten berücksichtigt werden. Es kann weiter davon ausgegangen werden, dass die Ausführungshäufigkeit der Zugriffe denen im Originalprogramm entspricht, weil die Befehle in den inneren Elementschleifen vor erhöhter Ausführung geschützt werden.

$$\begin{aligned} E_{Array\_gain} &= \sum_{c=0}^{ac} \sum_{(f\times n\times l\times b)\in accr_c} oex_{f,n,l} \cdot e_{B_b,GAIN\_READ,S_{f,n,c}} + \\ &\quad \sum_{c=0}^{ac} \sum_{(f\times n\times l\times b)\in accw_c} oex_{f,n,l} \cdot e_{B_b,GAIN\_WRITE,S_{f,n,c}} \end{aligned}$$

Es werden jeweils alle Arrays betrachtet, in der ersten Zeile werden lesende Zugriffe, in der zweiten Zeile die schreibenden Zugriffe berücksichtigt. Die Menge der Lese- und Schreibzugriffe eines Arrays besteht aus Tupeln, die einzelnen Elemente $(f,n,l,b)$ des Zugriffs werden im restlichen Teil der Formel verwendet.

Wenn die Zugriffsfunktion den Original-Index verwenden soll, wird eine zusätzliche Addition benötigt, diese Kosten sollen berücksichtigt werden. Diese Kosten fallen nur dann an, wenn die Index-Variable auch getiled ist. Als Position kann der Basisblock $B_b$ angenommen werden, in dem der Zugriff stattfindet.

$$E_{Array\_access} = \sum_{c=0}^{ac} \sum_{\substack{(f\times n\times l\times b)\in \\ accr_c\cup accw_c}} \sum_{d=1}^{ad_c} \sum_{i=1}^{ld_{f,n}} (O_{f,n,c,d,i} = 1) \cdot oex_{f,n,l} \cdot e_{B_b,ADD}$$

Der zusätzliche Speicherverbrauch kann in ähnlicher Weise berechnet werden:

$$S_{1,Array\_access} = \sum_{c=0}^{ac} \sum_{\substack{(f\times n\times l\times b)\in \\ accr_c\cup accw_c}} \sum_{d=1}^{ad_c} \sum_{i=1}^{ld_{f,n}} (B_b = 1) \cdot (O_{f,n,c,d,i} = 1) \cdot s_{ADD}$$

Bevor die Berechnung der Kopierkosten dargestellt werden kann, wird hergeleitet, wie sich der Arbeitsbereich eines Arrays berechnen lässt. Diese Betrachtung wird dimensionsweise durchgeführt und nicht mit der vom Compiler generierten eindimensionalen Zugriffsfunktion, da so geordnete, zusammenhängende Bereiche existieren. Der Arbeitsbereich ist der Teil des Arrays, der während der Ausführung der Elementschleifen sinnvoll in den Scratchpad-Speicher kopiert werden kann. Er soll die gleiche Dimension haben

wie das ursprüngliche Array. Auf die Optimierung der Anzahl der Dimensionen wird verzichtet, da so die Formeln zur Abschätzung der Kosten vereinfacht werden.

Der Arbeitsbereich (Workingset, ws) eines Arrays in einer Dimension hängt von der Zugriffsfunktion in dieser Dimension ab. Der Arbeitsbereich in verschiedenen Dimensionen kann unterschiedlich sein, so ist im folgenden Beispiel der Arbeitsbereich in der einen Dimension 1 und in der anderen Dimension `N`:

```
for (i=0; i<N; i++)
  A[1][i] = 5;
```

Der Arbeitsbereich in einer Dimension hängt davon ab, wieviele Indizes in dieser Dimension in dem betrachteten Teilnest überstrichen werden. Da die Koeffizienten der Zugriffsfunktion affine Funktionen sind, soll die Addition von Arbeitsbereichen und die Multiplikation mit Konstanten hergleitet werden. Eine affine Zugriffsfunktion hat folgende Form:

$$a_0 + \sum_k a_k \cdot i_k \tag{4.11}$$

$a_0$ wird der konstante Teil der Zugriffsfunktion genannt. Ein $a_k$ ist der Koeffizient der Index-Variablen $i_k$. $a_k$ soll eine ganze Zahl sein, da die Zugriffsfunktion zum Adressieren der Array-Elemente verwendet wird.

Eine Index-Variable `i` besitzt eine Menge aller möglichen Indizes, sie wird auch als Index-Raum bezeichnet: $I = \{i | i_{min} \leq i \leq i_{max}\}$. $i_{min} = min(I)$ ist der kleinste mögliche Index; $i_{max} = \max(I)$ ist der größte mögliche Index. Die Kardinalität dieser Menge ist $1 + i_{max} - i_{min} =: ws(\texttt{'i'})$ und entspricht dem Arbeitsbereich in einer Dimension. Die Zugriffsfunktion im Programm ist das Argument von *ws*. In diesem Fall ist es nur die Index-Variable `i`. Der Koeffizient von `i` ist 1 und alle anderen sind 0. Es wird davon ausgegangen, dass $i_{min} \geq 0$ und $i_{max} > 0$ ist. Die Schrittweite der Index-Variablen `i` soll 1 sein.

Arbeitsbereiche können addiert werden:

$$\begin{aligned} ws(\texttt{'i+j'}) &= |\{a | i_{min} + j_{min} \leq a \leq i_{max} + j_{max}\}| \\ &= 1 + i_{max} + j_{max} - i_{min} - j_{min} \\ &= 1 + i_{max} - i_{min} + 1 + j_{max} - j_{min} - 1 \\ &= |\{i | i_{min} \leq i \leq i_{max}\}| + |\{j | j_{min} \leq j \leq j_{max}\}| - 1 \\ &= ws(\texttt{'i'}) + ws(\texttt{'j'}) - 1 \end{aligned}$$

Dies entspricht einer Zugriffsfunktion, bei der die Koeffizienten der Index-Variablen `i` und `j` 1 und die anderen 0 sind.

Der Arbeitsbereich einer Index-Variablen `i` multipliziert mit einem positiven Koeffizienten `c` ist:

$$\begin{aligned} ws(\texttt{'c*i'}) &= |\{a | c * i_{min} \leq a \leq c * i_{max}\}| \\ &= 1 + c \cdot i_{max} - c \cdot i_{min} \\ &= 1 + c \cdot (-1 + 1 + (i_{max} - i_{min})) \\ &= 1 + c \cdot ws(\texttt{'i'}) - c \end{aligned}$$

Ist `c` negativ, gilt:

$$\begin{aligned} ws(\texttt{'c*i'}) &= |\{a | c * i_{max} \leq a \leq c * i_{min}\}| \\ &= |\{a | -1 * c * i_{min} \leq a \leq -1 * c * i_{max}\}| \\ &= ws(|c| \cdot i) \\ &= 1 + |c| \cdot i_{max} - |c| \cdot i_{min} \\ &= 1 + c \cdot (-1 + 1(i_{max} - i_{min})) \\ &= 1 + |c| \cdot ws(\texttt{'i'}) - |c| \end{aligned}$$

Ist $c = 0$, so ist $ws(\texttt{'c*i'}) = 0$. Für die Größe des Arbeitsbereichs ist das Vorzeichen der Koeffizienten unerheblich. Für die Größe des Arbeitsbereichs ist der konstante Teil der Zugriffsfunktion zunächst unerheblich.

Wenn zwei Arrays unterschiedliche Zugriffsfunktionen besitzen, dann ist die Differenz zwischen den Koeffizienten der Zugriffsfunktion entscheidend für die Optimierbarkeit dieses Array-Zugriffs durch die Nutzung von Scratchpad-Speicher. Ist bis auf den konstanten Teil die Differenz 0, so soll die Differenz zwischen den Zugriffsfunktionen als konstant bezeichnet werden. Ist die Differenz nicht konstant, existiert eine obere Schranke für die örtliche Lokalität der Zugriffe, die von den Parametern der beteiligten Schleifen abhängt. Es kann der Fall eintreten, dass sowohl Lokalität existiert, die durch den Scratchpad-Speicher genutzt werden kann, als auch der Fall, dass keine Lokalität existiert, so dass zwei Bereiche kopiert werden müssten. Da aber auf ein gemeinsames Array zugegriffen wird, muss bei der Code-Generierung eine Fallunterscheidung eingefügt werden oder ein zusammenhängender Bereich verwendet werden.

Im ersten Fall wird wesentlich mehr Code generiert und die Abschätzung der Kosten erschwert. Im anderen Fall wird ein sehr großes Array verwendet, das so groß wie das Original-Array sein kann. Dann ist es aber besser, direkt den Original-Index in der Zugriffsfunktion zu verwenden oder die beteiligten Schleifen nicht zu tilen.

Es soll daher nur der Fall modelliert werden, dass die Differenz der Zugriffsfunktionen in jeder Dimension konstant ist. Damit der generierte Code möglichst gut mit dem Original übereinstimmt, soll der konstante Teil der Zugriffsfunktion im getilten Programm nicht verändert werden. Dies hat zur Folge, dass der zur Zugriffsfunktion generierte Code in diesem Punkt im originalen und modifizierten Programm übereinstimmen. Zugriffe in der innersten Schleife haben einen großen Anteil an den Gesamtkosten, das Modell ist an dieser Stelle besonders genau.

Der Arbeitsbereich eines Arrays hängt davon ab, welcher Index-Bereich für die jeweilige Index-Variable überstrichen wird. $ei_{c,d,i} \in \mathbb{N}$ ist der **E**ffective **I**ndex, also entweder die Ausführungshäufigkeit der Schleife der Index-Variablen oder der Tiling-Faktor für die Index-Variable.

$$ei_{f,n,c,d,i} \quad = \quad (O_{f,n,c,d,i} = 1) \cdot ec_{f,n,i} + (O_{f,n,c,d,i} \neq 1) \cdot T_{f,n,i}$$

Für jede Index-Variable, die in der Zugriffsfunktion vorkommt, wird entschieden, ob der Original-Index oder der getilte Index verwendte werden soll.

Der zu kopierende Bereich in einer Dimension wird mit $ws_{f,n,c,d}$ abgekürzt. *ws* steht für **W**orking**S**et. Aus der obigen Herleitung der Rechenregeln wird in der folgenden Formel hergeleitet, wie in Abhängigkeit der gewählten Indizes und der Koeffizienten der zu kopierende Arbeitsbereich berechnet wird. Dass das Array an einer bestimmten Stelle im Programm vorkommt, wird während der Rechnung zunächst vernachlässigt.

$$\begin{aligned}
ws_{c,d,i} &= 1 + |ak_{c,d,i}| \cdot ei_{c,d,i} - |ak_{c,d,i}| \\
ws_{c,d} &= 1 + \sum_{i=1}^{ld} [ws_{c,d,i} - 1] \\
&= 1 + \sum_{i=1}^{ld} [1 + |ak_{c,d,i}| \cdot ei_{c,d,i} - |ak_{c,d,i}| - 1] \\
&= 1 + \sum_{i=1}^{ld} |ak_{c,d,i}| \cdot ei_{c,d,i} - |ak_{c,d,i}|
\end{aligned}$$

Für eine Index-Variable in einer Dimension, gilt die Gleichung in der ersten Zeile. Werden mehrere Summanden in einer Dimension zusammengesetzt, gilt die zweite Gleichung in der zweiten Zeile. Beide Gleichungen wurden oben hergeleitet. Durch Einsetzen der ersten Gleichung in die zweite und anschließendes Umformen kann die Gleichung weiter vereinfacht werden.

Unter der Berücksichtigung, dass der effektive Index einer Variablen von der Position der Schleife abhängt, ergibt sich folgende Formel für den Arbeitsbereich eines Arrays:

$$ws_{f,n,c,d} = 1 + \sum_{i=0}^{ld_{f,n}} |ak_{c,d,i}| \cdot ei_{f,n,c,d,i} - |ak_{c,d,i}|$$

Die Ausführungshäufigkeit der Kopierfunktion hängt davon ab, welche vorkommenden Index-Variablen getiled verwendet werden. $cp_{f,n,c}$ steht für **C**opy **P**oint und liefert die höchste Nummer der Schleife, deren zugehörige Index-Variable in der Zugriffsfunktion getiled verwendet wird. Genau in dieser Schleife sind alle Tiling-Indexvariablen aktuell und es kann der Arbeitsbereich des Arrays kopiert werden.

$$cp_{f,n,c} = \max(0 \cup \{i | 0 \leq i \leq ld_n, 0 \leq d \leq ad_c, ((ak_{c,d,i} \neq 0) \cdot (T_{f,n,i} \neq ec_{f,n,i}) \cdot (O_{f,n,c,d,i} = 0) = 1)\})$$

Anschaulich kann man sich das so vorstellen: Das Original-Array ist ein großes Feld, das in den Elementschleifen nur ausschnittsweise verwendet wird. Die Position des Ausschnittes wird durch die Tiling-Schleifen ausgewählt. Nur dann, wenn alle Tiling-Schleifen, die die Position des Ausschnittes bestimmen, aktuelle Tiling-Indizes besitzen, kann der Ausschnitt in den Scratchpad-Speicher kopiert werden.

Maximiert wird über die Menge der Schleifen-Indizes im Schleifennest. Alle Dimensionen des Arrays werden betrachtet. Es intessieren nur Index-Variablen, deren Koeffizienten nicht 0 sind und nur Tiling-Schleifen die tatsächlich getiled sind und auch getilted verwendet werden. Wird in der so gefundenen Tiling-Schleife kopiert, sind gerade alle verwendeten Tiling-Indizes aktuell und kein Index unnötig aktualisiert.

Mit Hilfe des Arbeitsbereiches und der Position der Kopierfunktion können die Kosten für das Kopieren eines Arrays in den Scratchpad-Speicher berechnet werden:

$$\begin{aligned}
E_{Array_copyR,f,n,c} &= tex_{f,n,cp_{f,n,c}} \cdot \left(e_{L_{f,n,cp_{f,n,c}},COPY_INIT,ad_c} + E_{Array_copyR,f,n,c,ad_c}\right) \\
E_{Array_copyR,f,n,c,1} &= ws(f,n,c,1) \cdot e_{L_{ad_c},COPY_LOOP} \\
E_{Array_copyR,f,n,c,d} &= ws(f,n,c,d) \cdot \left(e_{L_{ad_c},COPY_CONST} + E_{Array_copyR,f,n,c,d-1}\right)
\end{aligned}$$

Die Formel zur Berechnung der Kopierkosten für ein Array in einem Schleifennest bestehen aus drei Teilen. Nur wenn das Array im Scratchpad-Speicher liegen soll, muss kopiert werden. Die erste Zeile verwendet die Kopierposition ($cp$), um auf die Ausführungshäufigkeit der Tiling-Schleife zuzugreifen. Die Kosten werden mit dieser Ausführungshäufigkeit gewichtet. Die Kosten bestehen aus der Initialisierung und einem rekursiv berechneten Kostenteil. Die Position der Initialisierung hängt von der Position der Tiling-Schleife ab, in der die Initialisierung stattfindet.

Die Kopierfunktion besteht aus mehreren Schleifen. Je Dimension existiert eine Schleife. Die äußeren Schleifen sind gleich aufgebaut und sorgen dafür, dass jeweils das richtige innere zusammenhängende Feld kopiert wird. Die innerste Schleife führt die eigentliche Kopieroperation der ersten Dimension durch. Die Nummerierung ist so gewählt, dass sie zur Array-Organisation im encc und der Analyse passt. Die Kosten der innersten Schleife (in der Klasse COPY_LOOP) werden mit der Größe des Arbeitsbereichs dieser Dimension kopiert. Der Energieverbrauch hängt davon ab, wo die Kopierfunktion abgelegt wird. $L_d$ entscheidet für die $d$-dimensionale Kopierfunktion die Position. Die letzte Zeile ist die Rekursion über die Kosten der äußeren Schleifen. Da auch die inneren Kopierschleifen entsprechend des Arbeitsbereichs der weiteren Dimensionen ausgeführt werden, ist die Klammerung entsprechend.

Analog werden die Formeln definiert, wenn das Array geschrieben wird:

$$\begin{aligned}
E_{Array_copyW,f,n,c} &= tex_{f,n,cp_{f,n,c}} \cdot \left(e_{L_{f,n,cp_{f,n,c}},COPY_INIT,ad_c} + E_{Array_copyW,f,n,c,ad_c}\right) \\
E_{Array_copyW,f,n,c,1} &= ws(f,n,c,1) \cdot e_{L_{ad_c},COPY_LOOPW} \\
E_{Array_copyW,f,n,c,d} &= ws(f,n,c,d) \cdot \left(e_{L_{ad_c},COPY_CONST} + E_{Array_copyW,f,n,c,d-1}\right)
\end{aligned}$$

Jetzt ist es möglich, die gesamten Kopierkosten zu berechnen. Dazu muss zunächst betrachtet werden, in wie weit überhaupt kopiert werden muss. Wenn das Array außerhalb des Nestes kopiert wird, sollen im Modell keine Kopierkosten anfallen.

Mit der Notation lässt sich sehr einfach auf die Position der einzelnen Arrays zugreifen und es ist codiert, in welchen Basisblöcken und Schleifenabschnitten Zugriffe stattfinden. Es kommt aber vor, dass verschiedene Array-Zugriffe als eine Einheit betrachtet werden müssen. Wenn Arrays den gleichen Namen haben,

müssen sie auch auf den gleichen Speicherbereich zugreifen. Es muss für jedes Array auch nur eine Kopierfunktion eingeführt werden.

Es werden daher die Mengen $C_r$ und $C_w$ verwendet, in denen für jeden verwendeten Array-Namen eines Schleifennestes die Nummer eines Repräsentanten gespeichert ist. Für jeden Repräsentanten muss eine Kopierfunktion eingefügt werden.

$$\begin{aligned} E_{Array\_copy} &= \sum_{c' \in C_r} (S_{f,n,c} = 1) \cdot (cp_{f,n,c'} \neq 0) \sum_{(f \times n \times l \times b) \in accr'_c} E_{Array\_copyR,f,n,c'} \\ &+ \sum_{c' \in C_w} (S_{f,n,c} = 1) \cdot (cp_{f,n,c'} \neq 0) \sum_{(f \times n \times l \times b) \in accw'_c} E_{Array\_copyW,f,n,c'} \end{aligned}$$

In dieser Formel werden alle Arrays betrachtet. Nur wenn das Array im Scratchpad-Speicher liegt, muss kopiert werden. Nur dann, wenn die Kopierposition innerhalb der Tiling-Schleifen liegt, fallen weitere Kosten an.

In dieser Berechnung der Kopierkosten wird vernachlässigt, dass, wenn der Tiling-Faktor die Ausführungshäufigkeit nicht teilt, ein Rest entsteht und so die jeweils letzte Kopieroperation in einem Tiling-Durchlauf weniger Elemente im Speicher kopieren muss. Diese Überschätzung der Kosten wird in Kauf genommen, da es sich um einen i. d. R. kleinen Rest handelt. Die genaue Berücksichtigung dieses Umstandes würde eine exponentiell in der Dimension wachsende Komplexität der Formel verursachen, da Fallunterscheidungen notwendig sind.

Der für die Kopieroperationen anfallende Platzbedarf teilt sich in zwei Teile. Ein Teil, der bei jedem Kopieraufruf anfällt und ein anderer Teil, der je unterstützter Dimension anfällt.

$$\begin{aligned} S_{1,Array\_copy} &= S_{1,Array\_copyCall} + S_{1,Array\_copyFunction} \\ S_{1,Array\_copyCall} &= \sum_{c' \in C_r} \sum_{(f \times n \times l \times b) \in accr_{c'}} (L_{f,n,cp_{f,n,c}} = 1) \cdot (S_{f,n,c} = 1) \cdot s_{COPY\_CALL} \\ &+ \sum_{c' \in C_w} \sum_{(f \times n \times l \times b) \in accw_{c'}} (L_{f,n,cp_{f,n,c}} = 1) \cdot (S_{f,n,c} = 1) \cdot s_{COPY\_CALL} \\ S_{1,Array\_copyFunction} &= \sum_{d \in D} (L_d = 1) \cdot (s_{COPY\_INIT\_FUNCTION} + d \cdot s_{COPY\_CONST} + s_{COPY\_LOOP}) \end{aligned}$$

Die zweite Gleichung berechnet den zu reservierenden Platz für die Aufrufe der Kopierfunktionen. Wenn die Tiling-Schleife, in dem der Kopierpunkt liegt, im Scratchpad-Speicher liegt, können die Kosten anfallen. Nur wenn das Array auch im Scratchpad-Speicher liegen soll, muss kopiert werden. Es ist unerheblich für den Platzbedarf des Aufrufs der Kopierfunktion, in welche Richtung kopiert werden muss.

Ein besonderes Problem bei der Reservierung des Speichers stellen Literalpools dar. Literalpools müssen im selben Speicher liegen wie der Code, der auf sie zugreift. Einschränkungen im THUMB-Mode sorgen dafür, dass Literalpools nahe bei den Zugriffen stehen müssen, bei großen Programmen können sie auch mehrmals vorkommen.

Der encc berücksichtigt bei der Belegung des Scratchpad-Speichers Literalpools nicht ausreichend. Dies führt zu einer Überbelegung des Scratchpad-Speichers, wenn der Scratchpad-Speicher besonders voll belegt werden kann. Durch Tiling entsteht dieser Fall besonders oft. Da durch Tiling die Größe der Objekte variabel ist und sie feingranular an den freien Platz angepasst werden können. Wenn der Speicher fast bis zur Maximal-Kapazität gefüllt wird, kann dies in der Realität zu einem Fehler während der Programmausführung führen. Im enProfiler werden falsche Kosten für die Energie geliefert, da der Zugriff nicht definiert ist.

Es werden immer dann zwei Literalpools im Scratchpad-Speicher benötigt, wenn der Aufruf der Kopierfunktion im Scratchpad-Speicher liegt oder innerhalb der im Scratchpad-Speicher verschobenen Schleifenteile auf das Original-Array und das getilte Array zugegriffen wird. Eine Garantie über die Verteilung

kann nicht gegeben werden. Um eine Überbelegung sicher zu vermeiden, muss daher dem encc weniger Speicher zur Belegung zur Verfügung gestellt werden als eigentlich vorhanden wäre.

Literalpools werden im formalisierten Modell nicht berücksichtigt. In der Implementierung werden sie berücksichtigt, indem pro Array zwei Literalpools angenommen werden. Dieser Speicher wird vom zur Verfügung stehenden Speicher abgezogen und der restliche Speicher kann dann belegt werden.

Schließlich muss, wenn ein Array in den Scratchpad-Speicher verschoben wird, auch Speicher reserviert werden.

$$s_{m,Array\_use} = \sum_{c' \in C_u} (f \times n \times ł \times b \in (accr_{c'} \cup accw_{c'})) : (S_{f,n,c} = m) \cdot ws(f,n,c)$$

Es wird für jedes Array, das in einem Nest vorkommt, Platz in dem Speicher reserviert, in dem es abgelegt wird. Der Ausdruck zwischen Summenzeichen und : bedeutet, dass ein Zugriff aus der Menge der lesenden oder schreiben Zugriffe verwendet wird, um die Variablen $f$, $n$ und $c$ im Rest des Ausdrucks zu belegen.

#### 4.6.4.4 Spilling

Sofern Spilling nicht in den Elementschleifen auftritt, haben die zusätzlichen Zugriffe nur einen geringen Einfluss auf den Overhead. Da Spilling in den Elementschleifen aber nicht ausgeschlossen werden kann, werden die Kosten, die durch das Spilling anfallen, ebenfalls modelliert.

Der encc kann zu jedem Basisblock angeben, wieviele Register verwendet wurden. Die Anzahl verwendeter Register wird in der Variablen $ru_f$ gespeichert. Dies kann vor der Optimierung für jede Funktion berechnet werden:

$$ru_f \quad = \quad \max(r | r = ru_b, 0 \leq b \leq bbc, b \text{ gehört zur Funktion } f)$$

Die Betrachtungen zum Spilling werden je Funktion durchgeführt, da der encc eine globale Registerallokation, also funktionsweise Allokation, durchführt. Anhand der gewählten Tiling-Faktoren kann abgeschätzt werden, wieviele Register zusätzlich benötigt werden. Pro getilter Schleife wird ein Register benötigt. Wenn die Bildung des Minimums notwendig ist, wird ein weiteres Register benötigt. Die Anzahl zusätzlicher Register wird in *ar* gespeichert. *ar* steht für **A**dditional **R**egisters.

$$ar_f \quad = \quad \sum_{n=1}^{nc_f} \sum_{l=1}^{ld_{f,n}} (T_{f,n,l} \neq ec_{f,n,l}) + (T_{f,n,l} \mod ec_{f,n,l} \neq 0)$$

Die Anzahl schon verwendeter Register und die Anzahl zusätzlicher Register dürfen zusammen nicht größer als 8 sein, jedes weitere Register muss gespillt werden. *rs* ist die Anzahl zu spillender Register. *rs* steht für **R**egister **S**pilled.

$$rs_f \quad = \quad \max(0, ar_f + ru_f - 8)$$

Zur Berechnung, welche Register gespillt werden, kann ausgenutzt werden, dass der encc eine einfache Heuristik verwendet. Diese Heuristik wird weiter vereinfacht. Es wird angenommen, dass, wenn im Schleifennest Spilling durch Tiling verursacht wird, die Tiling-Indizes und Minimums-Variablen gespillt werden.

Diese Annahme stimmt z. B. nicht, wenn vorher nicht gespillt wird, eine Variable über das Schleifennest hinweg lebendig ist, aber nicht innerhalb des Nests verwendet wird. Dann wird eher diese Variable gespillt als Index- oder Minimums-Variablen. Für die betrachteten Benchmarks ist das implementierte Verfahren ausreichend.

Je nachdem, ob das Optimierungsproblem durch ganzzahlige lineare Optimierung oder durch einen genetischen Algorithmus gelöst werden soll, bieten sich unterschiedliche Vorgehensweisen an. Diese werden in den Abschnitten 4.7.3 bzw. 4.8.2 vorgestellt.

#### 4.6.4.5 Gesamtenergie und belegter Speicher

Nachdem nun alle Bestandteile des Modells beschrieben sind, können die Einzelergebnisse zusammengesetzt werden. Das Ergebnis des Modells ist eine Summe von Energieverbräuchen und Speicherbelegungen:

$$\begin{aligned} E = E_{mod} &= E_{BB} + E_{Loop} + E_{Array} + E_{Spill} \\ S_m &= S_{m,BB} + S_{m,Loop} + S_{m,Array} + S_{m,Spill} \end{aligned}$$

Die Summe des belegten Scratchpad-Speichers $S_1$ muss kleiner sein als der verfügbare Scratchpad-Speicher. $E$ ist der modellierte gesamte Energieverbrauch des Programms. Beide Größen hängen nun nur noch von den Eingaben des Optimierungsproblems und den gewünschten Ausgaben ab. $E$ kann nun kann unter Berücksichtigung der Speicherbelegung minimiert werden. Es gilt $E \in \mathbb{F}$, $S_m \in \mathbb{N}$.

## 4.7 Lösung des Optimierungsproblems durch ganzzahlige lineare Optimierung

Viele Arbeiten, die sich mit der optimierten Belegung von Scratchpad-Speichern auseinandergesetzt haben [SWLM02, The00, Gru02, VWM04b], verwenden eine Formulierung als ganzzahliges lineares Optimierungsproblem. Diese Vorgehensweise wurde zunächst auch für diese Arbeit in Betracht gezogen. Das bisher beschriebene Optimierungsproblem ist nicht in einer linearen Form. Es existieren Vergleiche und Produkte von Variablen, die umformuliert werden müssen.

Die Indizes der Variablen und Gleichungen werden vollständig aufgezählt, so dass für jede Variable mit unterschiedlichem Index eine Entscheidungsvariable eingeführt wird. Die Gleichungen können schrittweise weiter transformiert werden, dabei werden Variablen, Produkte und Tests durch neue Variablen ersetzt und weitere Bedingungen eingefügt. In den folgenden drei Abschnitten wird beschrieben, wie ein Teil der Übersetzung stattfinden kann und wie Spilling modelliert werden könnte. Schließlich wird im Abschnitt 4.7.4 dargestellt, warum es nicht sinnvoll ist, dieses Optimierungsproblem als ganzzahliges lineares Optimierungsproblem aufzufassen.

### 4.7.1 Linearisierung

Eine grundlegende Eigenschaft des vorliegenden Optimierungsproblems ist, dass Produkte von Entscheidungsvariablen vorkommen. Die Häufigkeit für die Ausführung eines Code-Fragmentes ist das Produkt zwischen der Ausführungshäufigkeit der äußeren Schleifen mit der Iterationsanzahl der aktuellen Schleife. Da die Ausführungshäufigkeiten der Tiling-Schleifen von den gewählten Tiling-Faktoren abhängen, werden alle Ausführungshäufigkeiten von den Entscheidungsvariablen im Produkt beeinflusst.

Zu jeder natürlichen Zahl $x$ gibt es eine Binärdarstellung $x = \sum_{i=0}^{n} x_i \cdot 2^i$. Das Produkt zweier Binärzahlen lässt sich mit Hilfe der Schulmethode [GKN81, Multiplikation III] berechnen, wie in Abbildung 4.13 dargestellt. Hier muss nur die Einbit-Multiplikation implementiert werden. Der Booth-Algorithmus [Boo51] erlaubt die Multiplikation von ganzen Zahlen. Für den Booth-Algorithmus muss die Multiplikation von einer Entscheidungsvariablen mit $0, 1, -1$ realisiert werden, daher wird die Schulmethode verwendet.

Es wird angenommen, dass bei der Generierung des ILP abgeschätzt werden kann, wie groß jede Variable $i$ maximal sein kann. In Binärdarstellung werden für die Variable $i$ $\lfloor \log_2(\max(i)) + 1 \rfloor = n_i$ Bits benötigt. Im Folgenden sind die Konstanten $n_x, n_y$ im Voraus berechnet worden. An die Stelle des Produktes $x \cdot y$ tritt eine neue Variable $z$, die durch folgendes Teilproblem berechnet wird. Da die Schulmethode ohne weitere Behandlung der Vorzeichen betrachtet wird, muss sicher gestellt sein, dass alle Entscheidungsvariablen des Produktes positiv sind.

Folgende Gleichungen sind in das ILP hinzuzufügen:

| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $\cdot$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|---|
| $z_{3,3}$ | $z_{2,3}$ | $z_{1,3}$ | $z_{0,3}$ | | | | | |
| | $z_{3,2}$ | $z_{2,2}$ | $z_{1,2}$ | $z_{0,2}$ | | | | |
| | | $z_{3,1}$ | $z_{2,1}$ | $z_{1,1}$ | $z_{0,1}$ | | | |
| | | | $z_{3,0}$ | $z_{2,0}$ | $z_{1,0}$ | $z_{0,0}$ | | |
| $z_6$ | $z_5$ | $z_4$ | $z_3$ | $z_2$ | $z_1$ | $z_0$ | | |

Abbildung 4.13: Multiplikation von x und y mit Hilfe der Schulmethode

$$
\begin{aligned}
x &= \sum_{i=0}^{n_x} x_i \cdot 2^i \\
y &= \sum_{i=0}^{n_y} y_i \cdot 2^i \\
z &\geq 0 \\
z &\leq 2^{n_x+n_y+1} \\
z &= \left(\sum_{i=0}^{n_x} x_i \cdot 2^i\right) \cdot \left(\sum_{j=0}^{n_y} y_j \cdot 2^j\right) \\
&= \sum_{j=0}^{n_y} 2^j \cdot y_j \cdot \sum_{i=0}^{n_x} x_i \cdot 2^i \\
&= \sum_{j=0}^{n_y} 2^j \cdot \sum_{i=0}^{n_x} x_i \cdot y_j \cdot 2^i \\
&= \sum_{j=0}^{n_y} 2^j \cdot \sum_{i=0}^{n_x} z_{i,j} \cdot 2^i \\
z_{i,j} &= x_i \cdot y_j
\end{aligned}
$$

Die letzte Gleichung kann so nicht als ILP implementiert werden und wird durch folgende Gleichungen ersetzt [Bea05]:

$$
\begin{aligned}
z_{i,j} &\leq \frac{1}{2}x_i + \frac{1}{2}y_j \\
z_{i,j} &\geq x_i + y_j - 1
\end{aligned}
$$

Wenn die maximale Größe der am Produkt beteiligten Zahlen bekannt ist, kann dieses Verfahren verwendet werden. Der Aufwand ist quadratisch in der Länge der Binärdarstellung der Zahlen.

### 4.7.2 Tests

An einigen Stellen muss geprüft werden, ob eine Entscheidungsvariable $x$ gleich Null $(x = 0)$ oder ungleich Null $(x \neq 0)$ ist. Dieser Ausdruck soll zur Linearisierung durch eine neue Entscheidungsvariable $y$ ersetzt werden. Außerdem werden zusätzliche Bedingungen eingefügt, damit $y$ den richtigen Wert hat. Es soll gelten:

$$
\begin{aligned}
(x = 0) &\Rightarrow y = 1 \\
(x \neq 0) &\Rightarrow y = 0
\end{aligned}
$$

Zwei Möglichkeiten wurden gefunden, um die Ersetzung durchzuführen: Eine einfachere Lösung, die nicht allgemein ist und eine allgmeine, die aber eine große Beschreibungskomplexität hat.

Durch die einfachere Lösung kann die Ersetzung von $(x = 0)$ durch $y$ nur in der Zielfunktion durchgeführt werden und nur dann, wenn $y = 0$ für das Ziel von Vorteil ist. Es ist von Vorteil für eine zu minimierende Zielfunktion, wenn ein Summand mit $y$ multipliziert wird.

Es wird folgende Bedingung eingefügt und die Ersetzung vorgenommen:

$$x + y > 0$$

Diese Bedingung sorgt dafür, dass $y$ immer dann 1 ist, wenn $x = 0$ ist und $y = 0$ von Vorteil ist. Falls $y = 0$ nicht von Vorteil ist, dann ist $y = 1$ von Vorteil und dieses Verfahren kann zur Ersetzung nicht angewendet werden.

Ein weiteres Problem ist, dass dieses Verfahren direkt nur für Vorkommen in der Zielfunktion funktioniert. Ein allgemeines Verfahren verwendet ebenfalls die Binärdarstellung und nutzt aus, dass logische Formeln mit einem ILP dargstellt werden können.

Wie bei der Linearisierung der Multiplikation kann die Variable, die auf 0 getestet werden soll, auch in ihre Binärdarstellung umgewandelt werden. Es muss dann nur jede Bitposition geprüft werden, wenn eine Stelle nicht 0 ist, dann ist die gesamte Zahl nicht 0.

In einem ILP lassen sich logische Formeln codieren. Eine Oder-Verknüpfung binärer Entscheidungsvariablen kann als Summe dargestellt werden, die Und-Verknüpfung mehrerer solcher Terme durch Hinzufügen dieser Terme in das Bedingungssystem.

Eine Oder-Verknüpfung $(x \vee y)$ lässt sich im ILP durch eine Bedingung $(x + y \geq 1)$ ausdrücken. Eine Negation einer binären Entscheidungsvariable $\neg x$ geschieht durch Substitution mit $1 - x$ in der Summe.

Die Zahl $x$ soll schon wie oben in die Binärdarstellung zerlegt und die entsprechende Gleichung eingefügt sein.

$$\begin{aligned}
&\bigvee_i x_i \Leftrightarrow \neg y \\
\leftrightarrow &\left(\bigvee_i x_i \Rightarrow \neg y\right) \wedge \left(\bigvee_i x_i \Leftarrow \neg y\right) \\
\leftrightarrow &\left(\neg\left(\bigvee_i x_i\right) \vee \neg y\right) \wedge \left(\bigvee_i x_i \vee y\right) \\
\leftrightarrow &\left(\left(\bigwedge_i \neg x_i\right) \vee \neg y\right) \wedge \left(\bigvee_i x_i \vee y\right) \\
\leftrightarrow &\left(\bigwedge_i \neg x_i \vee \neg y\right) \wedge \left(\bigvee_i x_i \vee y\right) \\
\leftrightarrow &\bigwedge_i (1 - x_i + 1 - y \geq 1) \wedge \left(\sum_i x_i + y \geq 1\right) \\
\leftrightarrow &\bigwedge_i (-x_i - y \geq -1) \wedge \left(\sum_i x_i + y \geq 1\right) \\
\leftrightarrow &\bigwedge_i (x_i + y \leq 1) \wedge \left(\sum_i x_i + y \geq 1\right)
\end{aligned}$$

Die erste Zeile drückt den gewünschten Zusammenhang aus. Genau dann, wenn eines der Bits in der Binärdarstellung von $x$ Eins ist, ist die Zahl nicht Null $(\neg y)$. In der zweiten und dritten Zeile wird die Äquivalenz

aufgelöst, so dass nur noch „oder“, „und“ und „nicht“ in der Formel enthalten sind. In der vierten und fünfte Zeile werden mit den de-Morganschen Regeln die Negationen zu den Entscheidungsvariablen geführt. Im Schritt zur 6. Zeile werden die Negationen und Oder-Verknüpfungen der Entscheidungsvariablen durch Summen und Ungleichungen ersetzt. Die letzten zwei Schritte dienen dazu zu erkennen, welche Gestalt die zusätzlichen Ungleichungen im Optimierungsproblem haben.

Die Anzahl der eingefügten Ungleichungen ist um eins größer als die Länge der geprüften Zahl. Sie bestehen bis auf eine aus zwei Summanden. Die letzte Gleichung hat für jede Stelle der binären Zahl einen Summanden.

### 4.7.3 Spilling im ILP

Um Spilling im ILP zu modellieren, werden für jede Schleife in einem Nest zwei Variablen eingeführt, die angeben, ob die Index-Variable oder die Minimums-Variable gespillt werden muss. $si_{f,n,l} \in \mathbb{B}$ steht für **S**pill **I**ndex; $sm_{f,n,l} \in \mathbb{B}$ steht für **S**pill **M**inimum.

Es müssen ausreichend viele Register gespillt werden, daher wird folgende Bedingung eingefügt:

$$\forall f, \forall n : \sum_{l=1}^{ld_{f,n}} si_{f,n,l} + sm_{f,n,l} \geq rs_f$$

Da durch jede mit 1 belegte Variable zusätzliche Kosten entstehen, werden genau die Register mit den geringsten Kosten gespillt. Es muss aber sichergestellt werden, dass nur dann eine der Variablen 1 ist, wenn die Variable auch wirklich gespillt werden kann, d. h. sie existiert im generierten Programm:

$$\begin{aligned} (T_{f,n,l} \neq ec_{f,n,l}) + (1 - si_{f,n,l}) &> 0 \\ (ec_{f,n,l} \mod T_{f,n,l} \neq 0) + (1 - sm_{f,n,l}) &> 0 \end{aligned}$$

Kosten für das Spilling entstehen bei jedem Zugriff auf die Variable. Exemplarisch wird dies für die Initialisierung der Index-Variablen gezeigt:

$$E_{Spill\_Init} = \sum_{f=1}^{fc} \sum_{n=1}^{nc_f} \sum_{l=1}^{ld_{f,n}} tex_{f,n,l-1} \cdot si_{f,n,l} \cdot e_{SPILLIN}$$

Die Index-Variable wird außerdem bei der Schleifenkontrolle, der Minimums-Bildung, den Kopieraufrufen und der Verwendung von Original-Indizes verwendet. Es ist jeweils auch der Ort des Zugriffs bekannt. Die zu diesen Stellen gehörenden Summen müssen auch für das Spilling des Minimums gebildet werden. Das Minimum wird bei seiner Berechnung und bei der Schleifenkontrolle der Elementschleifen gespillt.

### 4.7.4 Probleme der Linearisierung

Die Übersetzung der Teilformeln führt zu einer erheblichen Vergrößerung des Optimierungsproblems. Die Anzahl der Variablen und Bedingungen hängt nicht mehr, wie bei anderen Optimierungsproblemen, nur von der Struktur des zu optimierenden Programms ab, sondern auch von der Größe der darin vorkommenden Zahlen.

Einen Großteil zusätzlicher Terme und Produkte stellt die Berechnung des Energieverbrauchs. Zur Berechnung des Energieverbrauchs in der Kopierfunktion wird das Produkt zwischen Energieverbräuchen gebildet. Der Energievebrauch der betrachteten Programme ist schon in der Größenordnung von $4,5 \cdot 10^7$. Je länger die betrachteten Programme laufen, desto größer werden auch die betrachteten Zahlen. Es ist damit zu rechnen, dass die einzelne Produkte etwa 25 binäre Stellen haben können.

Da eine Schätzung der Wertebereiche keine kleinen Zahlen mehr liefert, müssen an sehr vielen Stellen Produkte von langen Zahlen gebildet werden. Unter Umständen wird statt dessen davon ausgegangen, dass die Zahlen eine feste Länge haben. Dann werden bei einem Produkt zwischen 25-stelligen Zahlen 75 zusätzliche Variablen und 325 zusätzliche Terme benötigt.

Ein weiteres ungelöstes Problem der Umcodierung ist die Formel zur Berechnung der Kopierposition. Die Bildung des Maximums kann nicht einfach in ein lineare Optimierungsproblem eingefügt werden.

Um die Berechnung der Kopierposition zu vereinfachen, kann auf die dynamische Berechnung verzichtet werden. Dann wird angenommen, dass immer vor den Elementschleifen kopiert wird.

Aufgrund dieser Einschränkungen und der hohen Beschreibungskomplexität, die eine Lösung in akzeptabler Zeit nicht erwarten lassen, wurde das Optimierungsproblem nicht in ein ganzzahliges lineares Optimierungsproblem umgeformt.

## 4.8 Lösung des Optimierungsproblems durch einen genetischen Algorithmus

Die Lösung des Optmierungsproblems durch einen genetischen Algorithmus ist leicht implementierbar, da das formalisierte Optimierungsproblem als Fitnessfunktion verwendet werden kann. Variablen mit Indizes werden durch Arrays entsprechender Dimension ersetzt, die Indizes bilden die Zugriffsfunktion. Bedingungen und die Selektion von Variablen können einfach realisiert werden. Die Fitnessfunktion kann effizient ausgewertet werden. Für 200 wiederholte Ausführungen des genetischen Algorithmus zur Optimierung der Matrix-Multiplikation werden 364s Sekunden benötigt. Von 500 Individuen werden 100 ausgetauscht. Insgesamt werden 22 956 400 Evaluationen durchgeführt. Auf eine Evaluation entfallen $15,85\mu$s, inklusive einem anteiligen Overhead des genetischen Algorithmus selbst. Der verwendete Rechner ist ein Pentium 4 mit 3 GHz CPU-Takt und 1 MByte Cache.

Im folgenden Abschnitt wird zunächst darauf eingegangen, wie die Parameter der Fitnessfunktion codiert werden. Anschließend wird beschrieben, wie Spilling betrachtet werden kann. Schließlich wird am Schluss dieses Kapitels betrachtet, wie der genetische Algorithmus zu optimieren ist, damit die Ergebnisse zufriedenstellend sind.

### 4.8.1 Codierung der Parameter der Fitnessfunktion

Zur Codierung eines Chromosoms kann ein Bitstring verwendet weren. Teile des Bitstrings werden als ganze Zahlen aufgefasst und als Tiling-Faktoren für die Schleifen des Programms verwendet, die restlichen Parameter sind binär. Hilfsvariablen, wie sie bei der Codierung als ILP eingeführt werden, müssen nicht durch Gene des Chromosoms repräsentiert werden, sondern können in der Funktion selbst berechnet werden. Die Anzahl notwendiger Bits wird vor dem Starten des genetischen Algorithmus berechnet.

Wenn möglich werden nicht zulässige Parameter wohlwollend korrigiert. Wenn der Tiling-Faktor zu groß ist, wird das bekannte Maximum angenommen. Immer wenn ein Array nicht im Scratchpad-Speicher liegen soll, wird angenommen, dass der Original-Index verwendet wird. Es können aber auch Parameterkombinationen existieren, die zu einer ungültigen Lösung führen, indem der Scratchpad-Speicher überbelegt wird. Diese sind nicht ohne Weiteres korrigierbar und es ist nicht klar, welche Änderung vorgenommen werden muss, um die Überbelegung zu vermeiden.

Eine Überbelegung des Scratchpad-Speichers findet statt, wenn mehr Scratchpad-Speicher belegt wird, als vorhanden ist. Diese Lösung ist auf jeden Fall nicht realisierbar und darf daher nicht als optimale Lösung weiter betrachtet werden. Es kann aber sein, dass ein Teil der Lösung zu einer guten Lösung in der Zukunft führen kann. Daher soll die Lösung nicht verworfen sondern bestraft werden, indem zum aktuellen Minimum, dem Optimum der Fitnessfunktionen in der Population, die aktuell berechneten Kosten, also die Fitness des illegalen Individuum, addiert werden. Dies sorgt auch bei einer Veränderung des besten

Individuums dafür, dass die Fitness illegaler Individuen mindestens doppelt so schlecht ist. Außerdem können auch schlechte Individuen noch unterschieden werden, so dass auch hier noch selektiert werden kann.

In Tabelle 4.6 wird anhand des Beispiels der Matrix-Multiplikation mit 15 Elementen je Dimension gezeigt, wie die Bits interpretiert werden. Der Bitstring belegt die Parameter des Optimierungsproblems, die in Abschnitt 4.6.3 eingeführt wurden.

### 4.8.2 Spilling für den genetischen Algorithmus

Die Modellierung des Spilling in der Fitnessfunktion kommt ohne zusätzliche Parameter aus. Für jede Funktion wird die Anzahl der zu spillenden Register ermittelt. In jedem Nest wird solange ein Register gespillt, bis genügend Register gespillt wurden. Zuerst werden die äußersten Schleifen betrachtet und jeweils zuerst die Minimum-Variablen. Für dieses Register werden an allen Vorkommen die zusätzlichen Zugriffskosten verfolgt. Für alle Zugriffe wird im Modell berücksichtigt, ob sie vom Scratchpad-Speicher oder Hauptspeicher ausgeführt werden.

### 4.8.3 Optimierung des genetischen Algorithmus

Im Folgenden werden einige Varianten verschiedener Eigenschaften des genetischen Algorithmus betrachtet, um mit dem genetischen Algorithmus ein möglichst gutes Ergebnis zu erzielen. Nach dem No-Free-Lunch-Theorem [Sch95] kann ein allgemeines Verfahren, wie ein genetischer Algorithmus, im Durchschnitt eine spezielle Optimierung nicht schlagen. Es bietet sich daher an zu betrachten, welchen Einfluss Spezialisierungen haben.

Als Beispieleingabe für den genetischen Algorithmus wird die Matrix-Multiplikation mit 15 Elementen je Dimension und 512 Byte zur Verfügung stehendem Scratchpad-Speicher verwendet. Um die Eigenschaften des genetischen Algorithmus zu beurteilen, ist es ausreichend, nur die Ergebnisse im Modell zu betrachten. Wesentliche Kriterien für die Bewertung sind:

- Wie schnell wurden die jeweils besten Individuen gefunden?
- Wie weit sind die Ergebnisse gestreut?

Die Laufzeit bestimmt im Wesentlichen die Güte des Ergebnisses. Der genetische Algorithmus soll aber so schnell sein, dass ein (interaktiver) Nutzer noch bereit ist, auf das Ergebnis zu warten.

Im Folgenden werden die Modifizierungen des genetischen Algorithmus mit Hilfe von Scatterplots und Laufzeitverteilungen dargestellt. Scatterplot und Laufzeitverteilung bestehen aus Punkten. Jede Punktsorte in einem Diagramm entspricht einer Einstellung des genetischen Algorithmus. Jeder Punkt korrespondiert zu einer Lösung des genetischen Algorithmus.

Die Koordinaten der Punkte im Scatterplot entsprechen der Generation, in der das beste Ergebnis des Laufs gefundenen wurde und der Fitness des Ergebnisses. Ein Scatterplot liefert eine Übersicht, wie die Fitness der Ergebnisse verteilt ist und wann die jeweils besten Ergebnisse gefunden wurden. Da die Änderungen durch den genetischen Algorithmus an den Individuen zufällig erfolgen, kann es passieren, dass kleine Änderungen, die zum Erreichen des Optimums notwendig sind, nicht unbedingt durchgeführt werden. Bilden sich bei einer ausreichenden Anzahl Versuche mehrere Linien heraus, dann können diese Linien als verschiedene lokale Optima aufgefasst werden.

Eine Laufzeit-Verteilung enthält ebenfalls zu jedem durchgeführten Lauf des genetischen Algorithmus einen Punkt. Die Fitness selbst geht nicht in die Bewertung ein. Die Ergebnisse werden entsprechend der Generation, in der sie gefunden wurden, sortiert und durchnummeriert. Die Nummerierung wird auf einer Skala von 0% bis 100% normiert. Es kann dann abgelesen werden, welcher Anteil Versuche nach einer gewissen Anzahl Generationen das lokale Optimum gefunden haben. Ist die sich herausbildende Kurve der

Chromosom: 01 0110 1000 | 100 100 100 | 0101 1 0101 0 0101 0 | 1

| Bit im Chromosom | formale Bedeutung | Interpretation |
|---|---|---|
| 0 | $B_0 = 0$ | Basisblock `main` nicht im Scratchpad |
| 1 | $B_1 = 1$ | Basisblock `LL9_0` im Scratchpad |
| 0 | $B_2 = 0$ | Basisblock `LL8_0` nicht im Scratchpad |
| 1 | $B_3 = 1$ | Basisblock `LL7_0` im Scratchpad |
| 1 | $B_4 = 1$ | Basisblock `_M_4` im Scratchpad |
| 0 | $B_5 = 0$ | Basisblock `_M_5` nicht im Scratchpad |
| 1 | $B_6 = 1$ | Basisblock `_M_6` im Scratchpad |
| 0 | $B_7 = 0$ | Basisblock `_MAA_3` nicht im Scratchpad |
| 0 | $B_8 = 0$ | Basisblock `_MAA_2` nicht im Scratchpad |
| 0 | $B_9 = 0$ | Basisblock `_MAA_1` nicht im Scratchpad |
| 1 | $S_{1,1,0} = 1$ | Array `C` im Scratchpad und: |
| 0 | $O_{1,1,0,1,2} = 0$ | `j` getiled verwenden |
| 0 | $O_{1,1,0,2,1} = 0$ | `i` getiled verwenden |
| 1 | $S_{1,1,1} = 1$ | Array `A` im Scratchpad und: |
| 0 | $O_{1,1,1,1,3} = 0$ | `k` getiled verwenden |
| 0 | $O_{1,1,1,2,1} = 0$ | `i` getiled verwenden |
| 1 | $S_{1,1,2} = 1$ | Array `B` im Scratchpad und: |
| 0 | $O_{1,1,2,1,2} = 0$ | `j` getiled verwenden |
| 0 | $O_{1,1,2,2,3} = 0$ | `k` getiled verwenden |
| 0 | $0 \cdot 8+$ | |
| 1 | $1 \cdot 4+$ | |
| 0 | $0 \cdot 2+$ | |
| 1 | $1 \cdot 1 = T_{1,1,1}$ | Tiling-Faktor i-Schleife=5 |
| 1 | $L_{1,1,1} = 1$ | Overhead der i-Schleife im Scratchpad |
| 0 | $0 \cdot 8+$ | |
| 1 | $1 \cdot 4+$ | |
| 0 | $0 \cdot 2+$ | |
| 1 | $1 \cdot 1 = T_{1,1,2}$ | Tiling-Faktor j-Schleife=5 |
| 0 | $L_{1,1,2} = 0$ | Overhead der j-Schleife nicht im Scratchpad |
| 0 | $0 \cdot 8+$ | |
| 1 | $1 \cdot 4+$ | |
| 0 | $0 \cdot 2+$ | |
| 1 | $1 \cdot 1 = T_{1,1,3}$ | Tiling-Faktor k-Schleife=5 |
| 0 | $L_{1,1,3} = 0$ | Overhead der k-Schleife nicht im Scratchpad |
| 1 | $L_2 = 1$ | Kopierfunktion für zwei Dimensionen im Scratchpad |

Tabelle 4.6: Chromosom für die Matrix-Multiplikation mit 15 Elementen bei 512 Byte Scratchpad und seine Interpretation

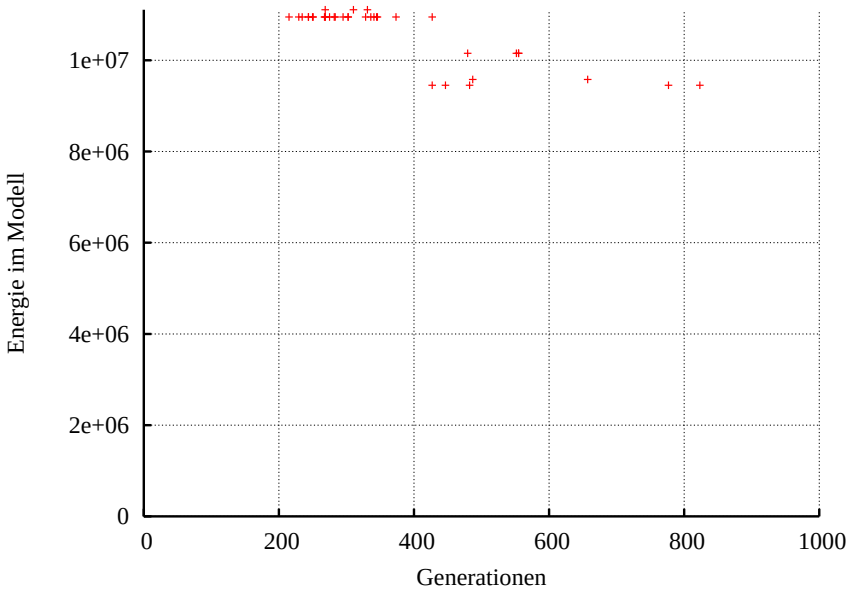


Abbildung 4.14: Beste gefundene Lösungen. Energie im Modell über der Generation, in der die Lösung gefunden wurde. S:3000 E:100 A:1000/100, Standard GA

Laufzeitverteilung flach, so lohnt es sich, in diesem Bereich nicht auf bessere Ergebnisse zu warten. Ist die Laufzeitverteilung sehr steil, so lohnt es sich, diesen Bereich immer abzuwarten.

Es wurden verschiedene Aspekte der Implementierung betrachtet, die Einfluss auf die Bewertung haben können:

- Populationsgröße
- Anteil der Ersetzungen
- Varianten der Mutationen
- Varianten für Rekombinationen

Für die Standardeinstellungen wurde das im Gesamtverlauf der Arbeit beste gefundene Ergebnis nicht gefunden. Da die Laufzeit für die Standardeinstellungen der Bibliothek extrem kurz ist, wurde die Populationsgröße von 100 auf 3000 Individuen erhöht und statt 10 Individuen 100 Individuen ausgetauscht. Die Anzahl erlaubter Generationen wurde bei 1000 belassen. Erst nach 100 Generationen ohne Veränderung wird abgebrochen.

Der Abstand zwischen der besten und schlechtesten Lösung in diesem Lauf ist 15%. In Abbildung 4.14 ist der Scatterplot angegeben, in dem diese Verbesserung deutlich wird. Das in Abbildung 4.15 dargestellte Laufzeitdiagramm zeigt, dass in 50% der Fälle die letzte Verbesserung schon nach 300 Generationen eingetreten ist. Ab 200 Generationen wurde das erste lokale Optimum gefunden. Die Kurve ist zunächst sehr steil und fast exponentiell, dies deutet darauf hin, dass ausreichend lange gewartet wird.

Um ein Bild zu erhalten, wie die Ergebnisse gefunden werden, ist ein Scatterplot nicht ausreichend. Unter unveränderten Versuchsbedingungen ist in Abbildung 4.16 der Verlauf des besten Individuums für verschiedene Läufe dargestellt. Nach einer gewissen Zeit stellt sich keine Verbesserung mehr ein. Es könnte

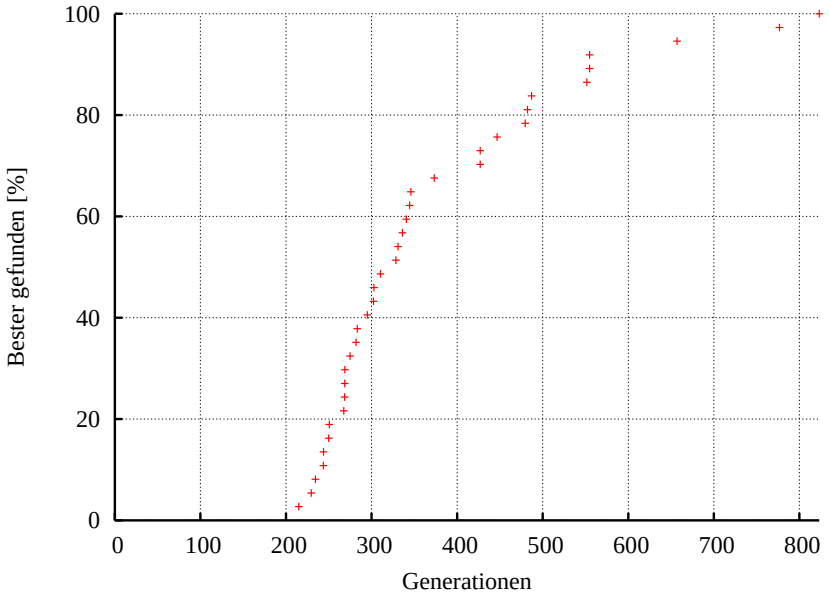


Abbildung 4.15: Laufzeitverteilung für S:3000 E:100 A:1000/100, Standard GA

also die Anzahl betrachteter Generationen verringert werden, wie es auch die Laufzeitverteilung in Abbildung 4.15 nahelegt. Bemerkenswert ist, dass für den Standard GA und eine Populationsgröße von 3000 in 37 Versuchen das Optimum oder eine sehr nahe Variante siebenmal gefunden wurde. Dies bedeutet, dass, wenn der genetische Algorithmus in dieser Konfiguration mehr als fünfmal neu gestartet wird, es sehr wahrscheinlich ist, dass dann das beste bekannte lokale Optimum darunter ist.

Es stellt sich die Frage, ob mit geeigneten Mutationen und Rekombinationen das Optimum bei einer kleineren Populationsgröße oder mit weniger Generationen gefunden werden kann. Dazu wurden eine Reihe verschiedener Mutations- und Rekombinationsmöglichkeiten ausprobiert und statt des binären Datentyps ein eigener definiert, auf dem auch andere Mutationen und Rekombinationen implementiert werden können.

Als mögliche Einzelmutationen für Tiling-Faktoren wurden, die folgenden in Betracht gezogen:

- 1 addieren oder subtrahieren,
- zufällige Zahl addieren oder subtrahieren,
- mit zufälliger Zahl (aus $[1,4]$) multiplizieren oder durch zufällige Zahl (aus $[1,4]$) dividieren und
- zufällig belegen.

Es wurden uniformes Crossover und 1-Punkt-Crossover evaluiert. Eine neue Rekombination ist das zyklische Vertauschen der Tiling-Faktoren. Außerdem wurde eine spezielle Rekombination eingeführt, bei der ein Tiling-Faktor auch auf die anderen Schleifen übetragen wird. Es kann gewählt werden, wieviele Einzelmutationen pro Mutationsschritt höchstens durchgeführt werden.

Es zeigt sich, dass eine zufällige Belegung während der Mutation und das Crossover, welches die Tiling-Faktoren anderer Schleifen übernimmt, sich positiv auf die gefundenen Ergebnisse auswirken. Letztendlich

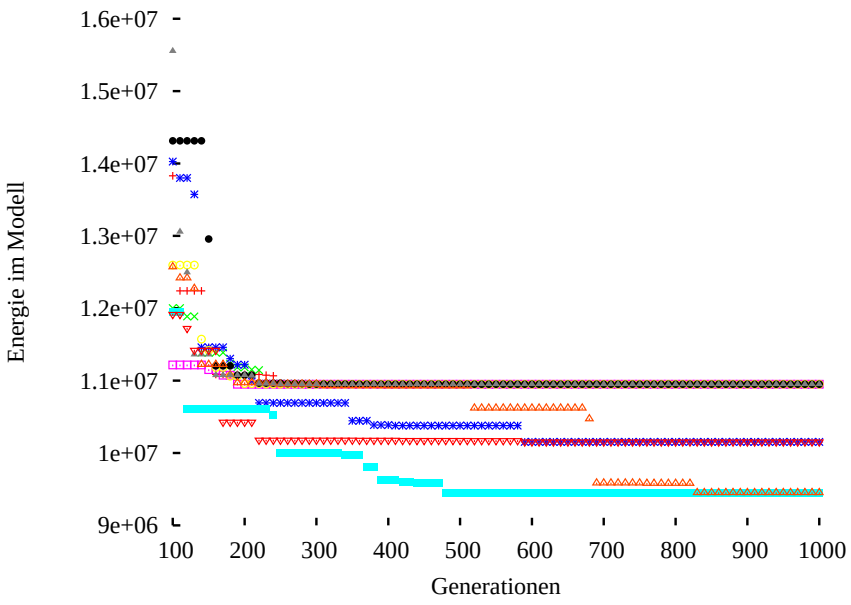


Abbildung 4.16: Entwicklung des besten Individuums über mehrere Generationen hinweg. S:3000 E:100 A:1000/100, Standard GA

werden alle Mutations-Operatoren kombiniert eingesetzt und nur auf das 1-Punkt-Crossover verzichtet. Als Wertebereich für die Addition und Multiplikation eignen sich ganze Zahlen aus dem Bereich $[1,4]$.

Um abzuschätzen, welche Aussicht auf Erfolg besteht, wird der genetische Algorithmus mit 500 Individuen, 100 zu ersetzenden Individuen und 2000 Generationen jeweils 201-mal ausgeführt. In der Kurve M5 werden höchstens 5 Mutationen pro Schritt zugelassen. In dieser Konfiguration wird das Optimum zweimal gefunden. Wenn allerdings die Anzahl der möglichen Mutationen in einem Schritt von 5 auf 10 erhöht wird (M10), wird das bekannte Optimum achtmal gefunden. Der Scatterplot ist in Abbildung 4.17 gegeben.

Mit einer zufälligen Initialisierung der Tiling-Faktoren innerhalb des Wertebereiches wird das beste bekannte Ergebnis in 18 von 200 Versuchen gefunden, unter Beibehaltung der möglichen 10 Einzelmutationen pro Mutationsschritt. Die Ergebnisse dieser Veränderung sind in der Kurve M10+rndinit dargestellt. Die dritte Kurve der Abbildung 4.17 zeigt, dass sehr viele gute Ergebnisse gefunden werden, wobei ein schlechteres lokales Optimum dominiert.

Die Laufzeitverteilung in Abbildung 4.18 wird durch die zufällige Initialisierung nicht wesentlich beeinflusst. Werden nur 5 einzelne Mutationen pro Mutationsschritt zugelassen ist die Kurve zunächst weniger steil, ist ab 150 Generationen aber steiler als die Kurven, die 10 Mutationen zulassen. Ab 250 Mutationen sind die Laufzeitverteilungen sehr ähnlich. In dem Bereich zwischen 70 und 150 Generationen finden die kleinschrittigen Mutationen kontinuierlich bessere Ergebnisse, so dass in diesem Bereich selten das Optimum gefunden wird.

Ein Verhältnis von 18 zu 200 liefert bei 20 Wiederholungen für diesen Benchmark eine gute Chance, das tatsächliche Optimum zu finden. Dies ist nur ein Beispiel, das nicht ohne Weiteres auf wesentlich komplexere Programme übertragen werden kann. Bei der Betrachtung der Laufzeitverteilung in Abbildung 4.18 fällt auf, dass nach 400 Generationen der genetische Algorithmus das lokale Optimum gefunden hat. Da insgesamt 2000 Generationen zugelassen werden, hat jeweils das Abbruchkriterium, das nach 200 Generationen ohne Änderung abbricht, das Ende der Suche bewirkt. Da sich die Kurve des Laufzeitdiagramms

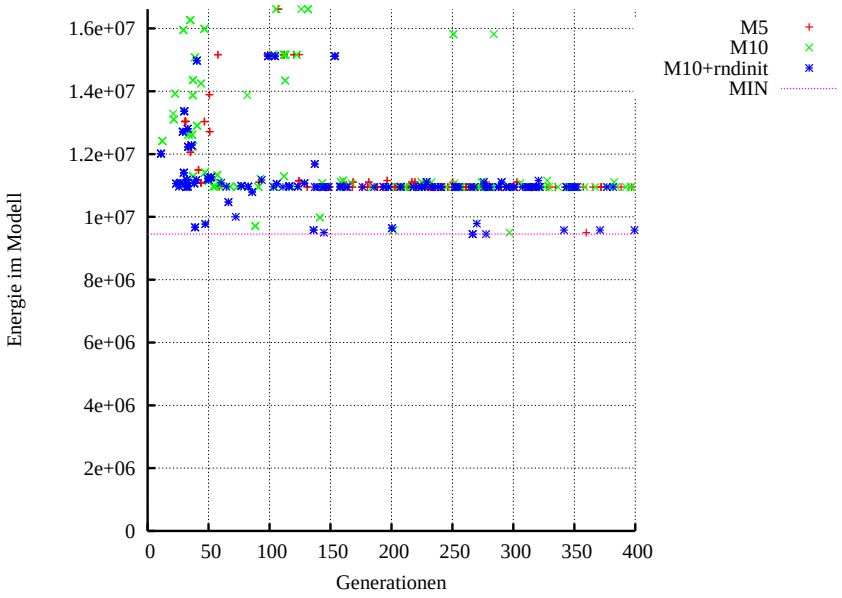


Abbildung 4.17: Scatterplot. Gegenübergestellt sind Varianten mit 5 Mutationen und 10 Mutationen. Es wird erlaubt, dass ein Tilingfaktor andere überschreibt. S:500 E:100 A:2000/100

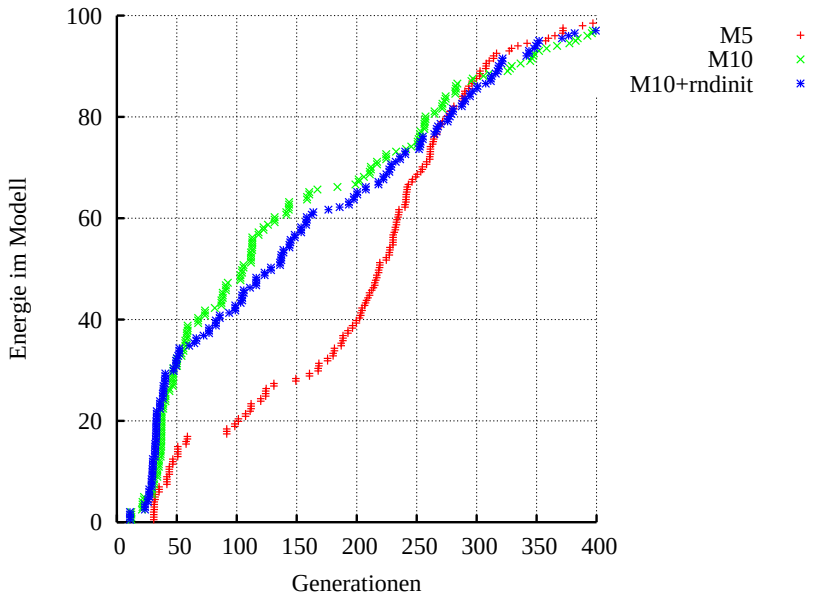


Abbildung 4.18: Laufzeitdiagramm. Gegenübergestellt sind Varianten mit bis zu 5 und 10 Einzel-Mutationen pro Mutation. Es wird erlaubt, dass ein Tilingfaktor andere überschreibt. S:500 E:100 A:2000/100

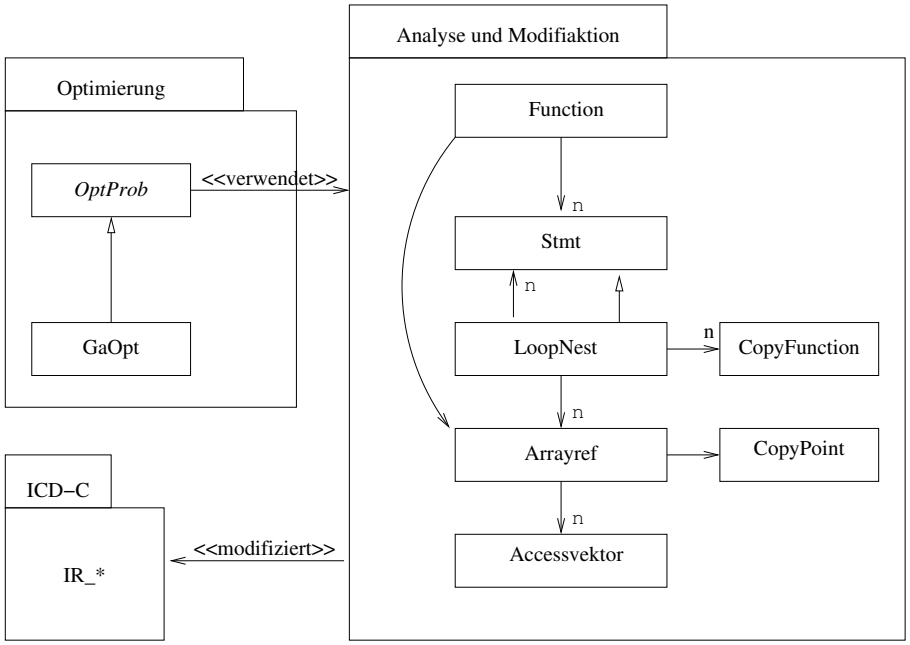


Abbildung 4.19: Klassendiagramm zur Code-Transformation

stark abflacht, scheint das Abbruchkriterium für diesen Benchmark geeignet.

Wenn hingegen die Laufzeit nur eine untergeordnete Rolle spielt, so kann auch Standardmutation und Crossover verwendet werden, wenn die Populationsgröße erhöht wird.

## 4.9 Programmtransformation mit ICD-C

Für die Programmtransformation sind dieselben Klassen aus Abbildung 4.10 verantwortlich, die das Optimierungsproblem mit den Informationen aus der ICD-C Datenstruktur gefüllt haben. Es kommen noch die Klassen CopyPoint und CopyFunction hinzu, es ergibt sich das in Abbildung 4.19 dargestellte Bild. Das Optimierungsproblem links hält die Belegung der Entscheidungsvariablen des besten gefundenen Wertes vor. Die Schnittstelle ist eine abstrakte Klasse. Der genetische Algorithmus wird in der Klasse GaOpt realisiert. Eine Instanz wird an die weiteren Klassen der Analyse und Modifikation zur Steuerung der Code-Transformation weiter geleitet. Die Klasse LoopNest rechts in der Mitte des Klassendiagramms transformiert die Schleifen selbst, indem Tiling-Schleifen eingefügt werden und die Schleifenköpfe angepasst werden. Durch ArrayRef rechts unten werden die Namen der Arrays, die in den Scratchpad-Speicher verschoben werden, umbenannt.

Die Klasse Accessvektor berechnetet die Modifikation der Zugriffsfunktionen entsprechend der Entscheidung, ob ein Index im Original verwendet wird. Wird ein Index im Original verwendet, muss die Index-Variable durch die Summe aus Index-Variable und Tiling-Variable ersetzt werden. Zur Berechnung der Parameter im Kopieraufruf werden auch Funktionen der Klasse Accessvektor verwendet, da sich einige Parameter aus den Koeffizienten der Zugriffsfunktion ableiten lassen.

Der Aufruf der Kopierfunktion wird durch die Klasse CopyPoint generiert. Es wird auch sichergestellt, dass jede verwendete Kopierfunktion genau einmal in den Code eingefügt wird. Der Aufruf der Kopierfunktion wird durch jedes ArrayRef-Objekt sichergestellt. Die Kopierfunktion selbst wird durch das File-Objekt in die IR-Datenstruktur eingefügt. Der Code der Kopierfunktion wird durch die Klasse CopyFunction generiert.

Die Parameter der Kopierfunktion sind die Quelladresse und die Zieladresse. Es folgt die Anzahl zu kopierender Elemente in der ersten Dimension. Für weitere Dimensionen muss die Anzahl der zu kopierenden Zeilen, Ebenen u. s. w. angegeben werden und wie groß das Array in der betroffenen Dimension ist.

Die Anzahl der zu kopierenden Objekte hängt davon ab, wie groß die Workingsets des Arrays sind. Hier wird die Formel aus Abschnitt 4.6.4.3 noch einmal angewendet. Jedes Array kann vor oder nach den Elementschleifen kopiert werden. Kommen Arrays mit gleichem Namen, aber mit unterschiedlicher Zugriffsfunktion vor, so fügt das Array mit dem größten konstanten Anteil der affinen Zugriffsfunktion die Kopierfunktionen ein, so dass auch ein Array mit 0 als konstantem Anteil korrekt auf das Array im Scratchpad-Speicher zugreifen kann.

Die Klassen des ICD-C unterstützen durch verschiedene Methoden die Code-Transformation. Da die Zeiger auf die entsprechenden Objekte während der Analyse der IR gespeichert werden, kann auf die betroffenen Objekte direkt zugegriffen werden. Die generierten Namen entsprechen der Benennung aus Abschnitt 2.4.2 und verwenden zusätzlich einen Zähler und Markierung, um zu verdeutlichen, dass es sich um generierte Symbole handelt.

Im Folgenden ist für die Belegung des Chromosoms aus Tabelle 4.6 auf Seite 84 das Ergebnis der Code-Transformation angegeben. Der Code wurde zur besseren Lesbarkeit vereinfacht.

```
void main(int arg, char *  argv[])
{

  int i;
  int j;
  int k;
  int iT;
  int jT;
  int kT;

  for (iT=0; iT<15; iT+=5)
    {
      0;
      for (jT=0; jT<15; jT+=5)
        {
          0;
          copy((&C[iT][jT]), &Cs[0][0], 5, 5, 15, 5);
          for (kT=0; kT<15; kT+=5)
            {
              0;
              copy(&B[kT][jT], &Bs[0][0], 5, 5, 15, 5);
              copy(&A[iT][kT], &As[0][0], 5, 5, 15, 5);
              for (i=0; i<5; i++)
                {
                  for (j=0; j<5; j++)
                    {
                      for (k=0; k<5; k++)
                        {
                          Cs[i][j] = Cs[i][j] + As[i][k] * Bs[k][j];
                        }
                    }
                }
            }

          copy(&Cs[0][0], &C[iT][jT], 5, 5, 5, 15);
        }
```

```
    }
  /* Remaining local symbols:
     int c;
     int iM;
     int jM;
     int kM;
  */

}

void copy(int * s, int * d, int c1, int c2, int sS2, int sD2)
{
  int i1;
  int i2;
  for (i1=c1;
       i1>0;
       i1-=1)
    {
      for (i2=c2;
           i2>0;
           i2-=1)
        {
          *d=*s;
          s++;
          d++;
        }
      s+=sS2-c2;
      d+=sD2-c2;
    }
}
```

# Kapitel 5

# Ergebnisse

In diesem Kapitel werden die Ergebnisse der Simulation präsentiert und analysiert. Die übersetzten Programme werden mit der statischen Belegung des Scratchpad-Speichers durch den encc verglichen. Das Verfahren wird nicht mit dem dynamischen Scratchpad Overlay von Verma [VWM04b] verglichen, da auch dieses Verfahren große Arrays nicht verlegen könnte. Da der implementierte Ansatz im Modell einen dynamischen Austausch verschiedener Arbeitsbereiche unterschiedlicher Arrays im Scratchpad-Speicher nicht unterstützt, ist auch eine Übersetzung mit Scratchpad Overlay nicht sinnvoll.

Das Verfahren zur statischen Belegung von Scratchpad-Speichern, das Caches [VWM04a] berücksichtigt, bietet sich nicht an, weil im Kostenmodell des Frontend Caches nicht berücksichtigt werden.

Zu Beginn der Arbeit war es das Ziel, ein Verfahren zu entwickeln, das ausschließlich statische Analyse verwendet. Es hat sich im Verlauf der Betrachtung der Ergebnisse gezeigt, dass getilte Programme eine Programmklasse bilden, für die statische Analyse wesentlich schlechter geeignet ist als dynamisches Profiling. Einige Probleme, die bei der Verwendung der statischen Analyse zu beobachten sind, treten auch noch bei der Verwendung von dynamischem Profiling auf. Die Probleme sind im verwendeten Ansatz und den Vereinfachungen zu suchen. Die Einflüsse auf die Ergebnisse werden im Abschnitt 5.2 anhand des Benchmarks Matrix-Multiplikation genauer betrachtet. An diesem Benchmark werden auch weitere Betrachtungen durchgeführt, die nicht Ziel der Optimierung sind.

Im Anschluss wird der Benchmark selbst verändert und die Optimierbarkeit von anderen Programmen betrachtet. In Tabelle 5.1 sind die untersuchten Benchmarks aufgeführt. Es werden auch Kernel aus der digitalen Signalverarbeitung betrachtet, da sie häufig in eingebetteten Systemen, wie z. B. Mobiltelefonen zur Bild- und Tonsignalverarbeitung eingesetzt werden. Die Kernel werden sehr häufig ausgeführt, so dass es sich lohnt, diese zu optimieren. Die Fast-Fourier-Transformation wäre unter diesem Gesichtpunkt auch ein interessanter Benchmark. Der Iterationsraum kann durch das verwendete Modell nicht betrachtet werden. Auf die Untersuchung eines großen Benchmarks wurde verzichtet, da eine dynamische Allozierung des Speichers nicht modelliert wird. Schließlich werden die Ergebnisse, die mit statischer Analyse gewonnen wurden, zusammen betrachtet. Anschließend werden die mit dynamischen Profiling erhaltenen Ergebnisse gezeigt.

Die Rohdaten werden mit dem enProfiler gewonnen. Der Energieverbrauch des Gesamtsystems, Speicher und CPU sowie die Anzahl ausgeführter Instruktionen und die benötigten Zyklen werden erfasst. Der durch das Modell errechnete Energieverbrauch und die Entscheidungen zum Tiling werden für jeden Lauf gespeichert. Zur späteren Analyse werden die Assembler-Datei und die C-Datei des optimierten Programms gespeichert.

Der Vergleich mit einem unoptimierten Programm ist zwar für die Gesamteinsparung interessant. Mit dem in dieser Arbeit präsentierten Verfahren werden insgesamt bis zu 80% gespart. Ein großer Anteil dieser Optimierung wird allerdings bereits durch den encc verursacht. Um darzustellen, welche zusätzliche Optimierung durch Tiling erreicht werden kann, muss als Referenz der Energieverbrauch des durch den encc

| Abkürzung | Beschreibung | Datengröße [Byte] |
|---|---|---|
| MM14 | Matrixmultiplikation mit 14 Elementen | 2352 |
| MM15 | Matrixmultiplikation mit 15 Elementen | 2700 |
| MM16 | Matrixmultiplikation mit 16 Elementen | 3072 |
| MM32 | Matrixmultiplikation mit 32 Elementen | 12288 |
| TP | Tiefpass, Bildsignalverbesserung | 964 |
| FIR | endliche Impuls-Antwort, Tonsignalverarbeitung | 3688 |

Tabelle 5.1: Verwendete Benchmarks und ihre Datengröße

übersetzten Programms bei gleicher Größe des Scratchpad-Speichers verwendet werden. In Abbildung 5.1 und den weiteren Abbildungen, die eine relative Verbesserung über die Größe des Scratchpad-Speichers auftragen, wird jeweils das mit dem encc bei gleicher Scratchpad-Speichergröße übersetzte Originalprogramm als Maßstab verwendet.

Positive Zahlen entsprechen einer Verbesserung durch das in dieser Arbeit dargestellte Verfahren gegenüber einer Übersetzung des Originalprogramms mit dem encc. Ist die relative Verbesserung Null, so wurde keine Verbesserung erzielt. Ein negativer Energieverbrauch tritt auf, wenn durch das modifizierte Programm mehr Energie verbraucht wird. Bei einer Verschlechterung ist insbesondere zu untersuchen, wodurch dieses schlechtere Ergebnis verursacht wird.

In Abbildung 5.1 sind verschiedene Möglichkeiten für Vergleiche durchgeführt worden. Der mal-Modell-Balken zeigt die auftretende Verbesserung, wenn das schlechteste durch den GA gefundene lokale Optimum verwendete wird. Für den opt-Modell-Balken wird das dem Modell nach beste Programm mit dem ungetilten Programm verglichen. Der Balken Erster zeigt das erste gefundene Optimum. Für den Minimum-Balken werden alle Verbesserungen betrachtetet und die kleinste aufgetretene Verbesserung dargestellt. Max zeigt entsprechend die maximale tatsächlich erreichte Verbesserung. Schließlich zeigt Mittel die aufgetretene mittlere Verbesserung.

In den folgenden Betrachtungen werden nur noch die aussagekräftigen max und opt-Modell-Balken aufgeführt, um die Übersichtlichkeit zu gewährleisten.

Für jeden Balken eines Diagramms werden mindestens elf Simulationen durchgeführt. Zuerst wird das untersuchte Programm durch den encc unter Verwendung der angegebenen Größe des Scratchpad-Speichers, abzüglich des reservierten Platzes für Literalpools, übersetzt. Wie am Ende des Abschnitts 4.6.4.3 eingeführt, werden 8 Byte pro Array reserviert. Das Programm wird zyklengenau simuliert und der Energieverbrauch $E_{ref}$ durch den enProfiler erfasst. Anschließend wird das in dieser Arbeit beschriebene Verfahren mindestens zehnmal angewendet. Der genetische Algorithmus sucht einen minimalen Energieverbrauch. Im $i$-ten Lauf wird die Energie des Modells mit $E_{mod,i}$ bezeichnet. Das Programm wird entsprechend der gefundenen Parameter transformiert und ebenfalls durch den encc unter Verwendung der Größe des Scratchpad-Speichers übersetzt. Der wie oben mit dem enProfiler ermittelte Energieverbrauch wird mit $E_{mes,i}$ bezeichnet. Für einen max-Balken ($B_{max}$) wird die maximale gefundene Verbesserung gesucht. Für einen opt-Model-Balken ($B_{opt}$) wird die im Modell gefundene minimale Energie gesucht und der zu diesem Lauf durch Simulation gewonnene Energieverbrauch als Verbesserung dargestellt. Folgende Formeln formalisieren die Rechnung:

$$B_{max} = 100 - 100 \cdot \frac{\min\left(\{0 \leq i \leq 9 | E_{mes,i}\}\right)}{E_{ref}}$$

$$\exists i \; E_{mod,i} = \min\left(\left\{0 \leq j \leq 9 |\; E_{mod,j}\right\}\right) \Rightarrow B_{opt} = 100 - 100 \cdot \frac{E_{mes,i}}{E_{ref}}$$

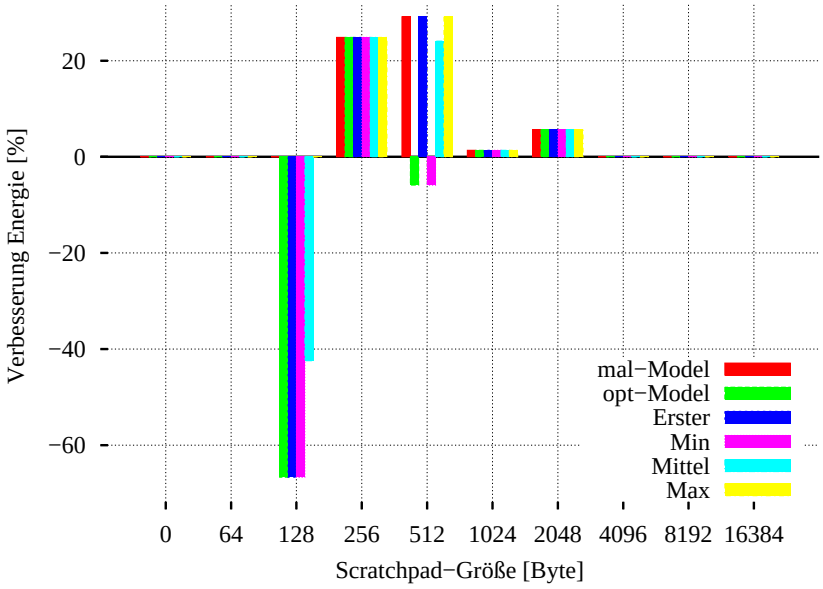


Abbildung 5.1: Verbesserung des Energieverbrauchs bezogen auf die gleiche Größe des Scratchpad-Speichers, Matrix-Multiplikation mit 15 Elementen, statische Analyse

## 5.1 Übersicht

Um nur die Effekte zu betrachten, die durch die Programmoptimierung verursacht werden, ist es sinnvoll, nur die Schleifen zu betrachten, die in zu optimierenden Programmen auch verändert werden können. Es werden daher nur Benchmarks betrachtet, die durch das in dieser Arbeit erstellte Programm getiled werden können und deren Ausführungsverhalten durch das Modell modelliert werden kann. Die Einschränkungen werden im Abschnitt 4.2 diskutiert.

Es wird darauf verzichtet, Programme zu betrachten, die einen großen nicht tilebaren Programmteil enthalten, da dann die Optimierungen, die durch den encc angestellt werden, dominieren.

Klassisch tilebar ist die Matrix-Multiplikation. Es handelt sich um ein Schleifennest aus drei Schleifen. Im Schleifennest werden die Elemente zweier Arrays miteinander multipliziert und aufakkumuliert. Das Ergebnis wird in einem dritten Array gespeichert.

In der Arbeit von Buchwald [Buc04] wird gezeigt, dass die folgende Schleifenanordnung für eine Architektur mit Cache optimal ist:

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

Diese Anordnung ijk wird im Folgenden zur Evaluierung des Verfahrens verwendet. Die Schleifenanordnung hat keinen signifikanten Einfluss auf das Ergebnis, da schleifeninvariante Ausdrücke durch den encc

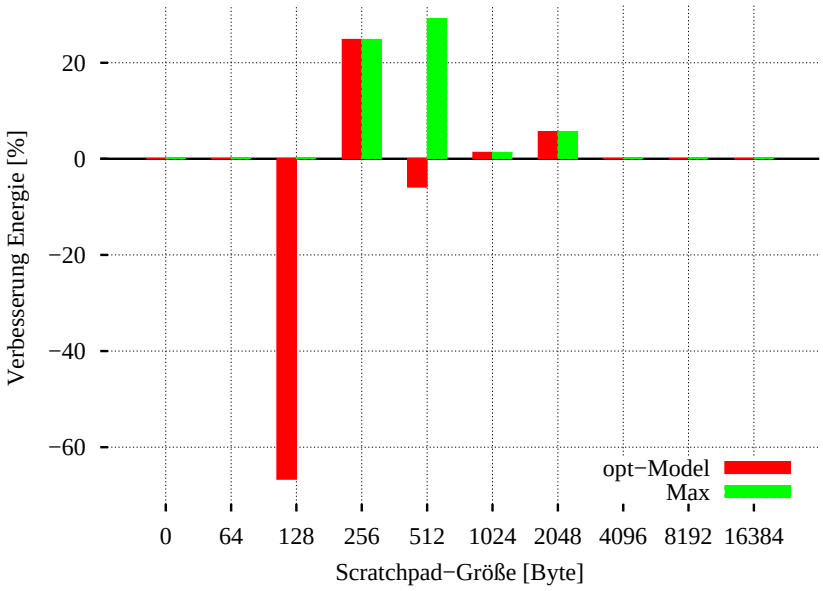


Abbildung 5.2: Verbesserung des Energieverbrauchs, Matrix-Multiplikation mit 15 Elementen, statische Analyse

nicht optimiert werden.

Den in Abbildung 5.2 dargestellten Ergebnissen liegt eine Matrixgröße von 15 Elementen in jeder Dimension zugrunde. Der maximale Gewinn mit 30% tritt bei einer Größe des Scratchpad-Speichers von 512 Byte auf. Mit nur 64 Byte ist keine Verbesserung durch dieses Verfahren möglich. Ab einer Größe des Scratchpad-Speichers von 4096 Byte ist ebenfalls keine Verbesserung mehr möglich. Die geringste Verbesserung von nur 1% ist bei 1024 Byte Scratchpad-Speicher aufgetreten. Bei Scratchpad-Speichergrößen von 128 und 512 Byte treten auch Verschlechterungen von bis zu 66% auf.

Bei diesem Benchmark stellen sich die Fragen:

- Warum ist bei manchen Scratchpad-Speichergrößen keine Veränderung aufgetreten?
- Warum ist bei einer Scratchpad-Speichergröße von 1024 nur ein kleiner Gewinn erzielt worden?
- Warum treten Verschlechterungen auf?
- Warum weichen das Optimum im Modell und das der Simulation voneinander ab?
- Welche Eigenschaften des Programms haben Einfluss auf das Ergebnis?

Diese Fragen werden im Folgenden beantwortet.

## 5.2 Einflüsse auf die Verbesserung

In den folgenden Abschnitten werden die Einflüsse auf die Verbesserung anhand des Beispiels der Matrix-Multiplikation erläutert. Die Größe des Scratchpad-Speichers ist ein grundlegende Eigenschaft und be-

stimmt, wie gut schon das Originalprogramm optimiert werden kann. Die Belegung des Scratchpad-Speichers durch den encc für das optimierte Programm bestimmt, ob die Entscheidungen aus dem Modell umgesetzt werden. Fehlentscheidungen bei der Belegung lassen sich auf falsche Annahmen im encc oder im Modell zurückführen. Ursache für Unstimmigkeiten sind z. B. zusätzliche Befehle, die im Modell nicht berücksichtigt werden. Geschlossen wird dieser Abschnitt mit einer Betrachtung der Genauigkeit des verwendeten Modells.

### 5.2.1 Größe des Scratchpad-Speichers

Ohne Scratchpad-Speicher kann keine Verbesserung auftreten. Tiling verursacht Overhead während des Kopierens und durch die zusätzliche Ausführung von Schleifen. Gewinn kann aber nur dann erzielt werden, wenn Speicherzugriffe im Scratchpad-Speicher statt im Hauptspeicher stattfinden. 64 Byte Scratchpad-Speicher reichen nicht aus, um Arrays zu verschieben. Die innerste Schleife des Programms besteht aus 22 Instruktionen, die 44 Byte belegen. Die innerste Schleife der Kopierfunktion belegt 14 Byte. Außerdem ist der Overhead für sehr kleine Tiling-Faktoren größer. Durch die Berücksichtigung von Literalpools, wie sie in Abschnitt 4.6.4.3 behandelt werden, stehen außerdem nur 40 Byte zur freien Verfügung.

Die Original-Arrays belegen 2700 Byte Speicher. In einem großen Scratchpad-Speicher mit 4096 Byte können die Arrays auch ohne Tiling verschoben werden, so dass dieses Verfahren keine Verbesserung bringen kann. Das Modell ist in der Lage zu erkennen, wann es sich nicht lohnt, das Programm zu tilen. Die trotzdem auftretenden, unerwarteten Verschlechterungen werden im Folgenden näher erläutert.

### 5.2.2 Speicherbelegung durch den encc

Stimmt die Speicherbelegung durch den encc nicht mit den Annahmen im Modell überein, findet meist keine oder eine kleinere Verbesserung als erwartet statt. Nicht-Übereinstimmungen bei der Speicherbelegung sind aber nur Symptome. Die Ursache kann beim encc oder im Modell gefunden werden.

Zunächst wird das Symptom näher beschrieben. In Abbildung 5.2 sind für das dem Modell nach optimale Programm bei 128 Byte und 512 Byte Scratchpad-Speicher Verschlechterungen laut Simulation zu erkennen. Die nähere Betrachtung der Speicherbelegungen zeigt, dass in beiden Fällen die vom Modell gewählte Speicherbelegung des Scratchpad-Speichers nicht umgesetzt wurde.

Um einen Eindruck zu gewinnen, wie der encc das Originalprogramm im Speicher verteilt, kann Tabelle 5.2 herangezogen werden. In Tabelle 5.3 sind repräsentative Fälle für die Speicherbelegung herausgegriffen, bei denen eine Verbesserung für bestimmte Scratchpad-Speichergrößen aufgetreten ist. In Tabelle 5.4 sind die Belegungen des Scratchpad-Speichers, bei denen es zu einer Verschlechterung gekommen ist, aufgeführt. In diesen Tabellen ist für jede relevante Scratchpad-Speichergröße ein Repräsentant ausgewählt und die Arrays angegeben, die in den Scratchpad-Speicher verschoben werden sollen. Ein Array ist eingeklammert, wenn das modifizierte Programm ein Array enthält, das dem Modell nach in den Scratchpad-Speicher verschoben werden soll, aber durch die Scratchpad-Speicherbelegung des encc nicht zur Verlagerung in den Scratchpad-Speicher ausgewählt wurde. In der letzten Zeile ist in Byte angegeben, wieviel Scratchpad-Speicher durch die Arrays belegt wird. Der restliche Speicher kann vom encc mit Code belegt werden.

Bei 1024 Byte Scratchpad-Speicher tritt laut Abbildung 5.2 nur ein sehr kleiner Gewinn auf. Bei der Betrachtung von Tabelle 5.2 fällt auf, dass im Vergleich zum 512 Byte Scratchpad-Speicher der encc nun das gesamte Array `C` auch ohne Tiling in den Scratchpad-Speicher verlegt. Dies bedeutet zwei von vier, also die Hälfte, der Array-Zugriffe sind durch den encc schon im Originalprogramm optimiert, ohne dass der Overhead des Tilings und des zusätzlichen Kopierens anfällt.

Bei 2048 Byte Scratchpad-Speicher kann der encc schon die Arrays `A` und `C` des Originalprogramms verschieben. Tiling liefert hier trotzdem noch Gewinn, weil große Tiling-Faktoren gewählt werden können. Das Array `C` wird, wie in der letzten Spalte von Tabelle 5.3 zu erkennen, im Original verwendet, so dass es ohne zusätzliche Kosten verschoben werden kann.

| Scratchpad-Größe: (Byte) | 1024 | 2048 |
|---|---|---|
| Arrays im Scratchpad | C[15][15] | C[15][15] A[15][15] |
| belegen: (Byte) | 900 | 1800 |

Tabelle 5.2: Belegung des Scratchpad-Speichers mit Arrays durch den encc bei der Übersetzung des unmodifizierten Programms

| Scratchpad-Größe: (Byte) | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|
| Arrays im Scratchpad | Cs[1][15] As[1][15] | Cs[2][15] As[2][15] | Cs[5][15] As[5][5] Bs[5][15] | C[15][15] As[15][5] Bs[5][15] |
| belegen: (Byte) | 120 | 240 | 700 | 1500 |

Tabelle 5.3: Belegung des Scratchpad-Speichers durch Arrays bei verschiedenen Scratchpad-Größen für den Benchmark Matrix-Multiplikation mit 15 Elementen. In den vorliegenden Fällen ist eine Verbesserung aufgetreten.

| Scratchpad-Größe: (Byte) | 128 | 256 | 512 |
|---|---|---|---|
| Arrays im Scratchpad | (Cs[3][3]) | Cs[8][1] Bs[15][1] | (Cs[5][5]) As[5][5] Bs[5][5] |
| belegen: (Byte) | 36 (0) | 120 | 300 (200) |

Tabelle 5.4: Belegung des Scratchpad-Speichers durch Arrays bei verschiedenen Scratchpad-Größen für den Benchmark Matrix-Multiplikation mit 15 Elementen. In den vorliegenden Fällen ist eine Verschlechterung aufgetreten. Die Arrays in Klammern wurden vom encc nicht in den Scratchpad-Speicher verschoben.

Im Vergleich der Tabellen 5.2 und 5.3 werden bei 1024 und 2048 Byte Scratchpad-Speicher mit Tiling weniger Daten in den Scratchpad-Speicher verlegt, weil der Code durch Tiling vergrößert wird.

In Tabelle 5.4 ist zu erkennen, dass bei 128 und 512 Byte Scratchpad-Speicher ein Array nicht in den Scratchpad-Speicher verschoben wurde. Wird ein Array zum Tiling vorgesehen, wird es dann aber nicht in den Scratchpad-Speicher verschoben, entsteht nicht der Gewinn, der die Kosten ausgleicht. Wenn ein Array nicht verschoben wird, kann es auch nicht den errechneten Gewinn erzielen. In diesen Fällen stimmt die prognostizierte Energie des Modells nicht mit dem Energieverbrauch des durch den encc erzeugten Programms überein.

Weil andere Entscheidungen getroffen werden, stimmen die Modelle des encc und des istf nicht überein. Fehlerquellen können im encc und im Modell gefunden werden.

#### 5.2.2.1 Abweichungen in der Bewertung des encc

In dieser Arbeit wird ein Verfahren präsentiert, das ohne dynamisches Profiling, nur durch Quelltextanalyse, zu einer Entscheidung kommt, wie das Programm zu tilen ist. Die Programmanalyse ist im Backend schwieriger als im Frontend. Der encc hat weniger Highlevel-Informationen über das zu übersetzende Programm. Dieses Defizit wird in Abschnitt 4.4 thematisiert.

Unterschiede ergeben sich in der Analyse der Kopierfunktion und der Ausführungshäufigkeiten insgesamt. Auch wenn encc und das Modell die gleiche Datenbank verwenden, kann es so zu unterschiedlichen Bewertungen kommen. Unterschiedliche Bewertungen können dazu führen, dass der Nutzen einzelner Objekte anders eingeschätzt wird und die Belegung des Scratchpad-Speichers unterschiedlich erfolgt.

Bei der Analyse der Kopierfunktion fehlt dem encc die Information, welche Daten in der Kopierfunktion verarbeitet werden. Es finden für den encc nur Pointerzugriffe statt. Dem encc ist nicht bekannt, dass die Kosten der Zugriffe in der Kopierfunktion von der Position der Arrays abhängen. Ein Teil des Gewinns der durch das Verschieben des Arrays in den Scratchpad-Speicher in der Kopierfunktion auftritt wird nicht berücksichtigt. Das Modell hingegen weiß sehr genau, welche Zugriffe in der Kopierfunktion stattfinden. Die in Abschnitt 4.6.4.3 beschriebene Teil-Formel unterscheidet sogar, ob gerade vom Hauptspeicher in den Scratchpad-Speicher kopiert wird oder umgekehrt.

Ein weiterer Punkt, an dem die Gewichtung der Nutzen zu einem unterschiedlichem Ergebnis kommt, ist die Abschätzung der Ausführungshäufigkeit. Der encc nimmt bei statischer Analyse an, dass jede Schleife zehnmal ausgeführt wird. Bei Bedingungen wird angenommen, dass jedes Sprungziel gleich wahrscheinlich ist.

Das Modell hingegen kennt die unterschiedlichen Ausführungshäufigkeiten einzelner Schleifen und kann so eine feinere Gewichtung vornehmen.

Alle diese Punkte können potentiell zu einer unterschiedlichen Bewertung durch den encc und das Frontend führen. Durch Verwendung des dynamischen Profiling können diese Fehlerquellen aber weitgehend eliminiert werden. Der encc besitzt nach der Ausführung des Programms die genauen Nutzen aller Objekte, die in den Scratchpad-Speicher verschoben werden können.

Unterschiede zwischen den Modellen im encc und dem Frontend können aber trotzdem noch auftreten. In diesen Fällen muss der Fehler im Modell des Frontends gesucht werden, da auch das Modell einzelne Aspekte vernachlässigt oder vereinfacht.

#### 5.2.2.2 Abweichungen in der Bewertung des Modells

Zentrale Fehlerquelle im Frontend ist das Fehlen von genauen Informationen über den Code des optimierten Programms. Eine weitere Fehlerquelle sind Vereinfachungen, die im Modell angewendet werden, um die Formeln zu vereinfachen.

Vereinfachungen im Modell sind in der Regel pessimistische Abschätzungen. So werden die Kosten der Kopierfunktion vereinfacht, wenn der Tiling-Faktor einen Rest lässt. Dies wird im Abschnitt 4.6.4.3 dargestellt. Eine pessimistische Abschätzung führt aber zu unerwarteten Gewinnen. Die verwendeten Kostenklassen des Modells, die in Abschnitt 4.6.4 eingeführt werden, bilden typische Größen der Programmtransformation ab. Sie sind möglichst wenig pessimistisch belegt.

Der größte Teil der Gesamtkosten wird aber durch die innerste Schleife des Programms erzeugt. Diese Kosten werden durch die Betrachtung der Basisblöcke abgedeckt. Veränderungen im Code werden in gewissen Grenzen modelliert, wie Kosten für Spilling oder wenn ein Original-Index verwendet wird.

Andere Änderungen in der innersten Schleife werden aber nicht berücksichtigt. Die in den folgenden Abschnitt 5.2.3 vorgestellten Ursachen sorgen dafür, dass die Anzahl der Befehle in der innersten Schleife von der erwarteten Anzahl abweicht.

Dies führt einerseits zu einer falschen Bewertung der Kosten für die innerste Schleife und andererseits zu einer falschen Abschätzung der Code-Größe. Durch eine falsche Code-Größe kann der Scratchpad-Speicher überbelegt sein.

Durch die unerwartete Veränderung der Größe und der Kosten verändert sich der Nutzen der betroffenen Objekte. Dies kann dazu führen, dass der encc auch bei dynamischen Profiling eine andere Speicherbelegung als im Modell als besser erachtet. Dann tritt der erwartete Gewinn nicht ein und auch die Kosten der anderen Objekte können sich verändern, wenn sie aufgrund der Überbelegung nicht in den Scratchpad-Speicher verschoben werden können. Im Fehlerfall kann auch ein anderes Programm zu einem besseren Ergebnis führen als das dem Modell nach optimale Programm.

Kleinere Unterschiede in der Code-Generierung, die im folgenden Abschnitt eingeführt werden, sollten in der Regel keine großen Auswirkungen auf das Ergebnis haben. Der encc betrachtet für Tiling-Schleifen mindestens zwei Basisblöcke, die im Modell als ein Block betrachtet werden. Die Kopierfunktion wird im Modell als ein Block gesehen, die für den encc aber aus mehreren Basisblöcken besteht. Wenn das Modell Tiling-Schleifen oder die Kopierfunktion in den Scratchpad-Speicher verschoben hat, kann der encc selten ausgeführte Teile der Kopierfunktion oder Teile der Tiling-Schleifen auch im Hauptspeicher belassen und unerwartet größere Code-Teile trotzdem im Scratchpad-Speicher ablegen. Diese Flexibilität ermöglicht es dem encc, eine gewinnbringende Belegung zu wählen, auch wenn das Modell den Speicherbedarf für den Code unterschätzt hat.

Da die Größe der deklarierten Arrays fest ist, beschränkt sich die Fehlerursache auf unerwartete zusätzliche Befehle. Auf die unterschiedlichen Aspekte dieser Abweichungen und insbesondere der zusätzlichen Befehle wird im Weiteren näher eingegangen.

### 5.2.3 Zusätzliche Befehle

Die Anzahl der Befehle in der innersten Schleife ergibt sich im Modell aus der Anzahl der Befehle des korrespondierenden Basisblocks des Originalprogramms, ob Index-Variablen im Original verwendet werden und ob Index-Variablen gespillt werden. Im Modell ist dies über mehrere Teilformeln verteilt und in den Abschnitten 4.6.4.1, 4.6.4.3 und 4.6.4.4 beschrieben. Die Differenzen zwischen erwarteter Anzahl Befehle und tatsächlicher Anzahl sind für die betrachteten Benchmarks in Tabelle 5.9 auf Seite 118 angegeben.

Durch die Veränderungen können auch Befehle gegenüber dem Originalprogramm eingespart werden. Ein Beispiel ist die Matrix-Multiplikation mit 32 Elementen. Können Befehle im Vergleich zum Modell eingespart werden, so wird das Ergebnis dadurch besser als angenommen. Da diese Möglichkeiten aber nicht modelliert werden, kann ihre Einsparung nicht gezielt ausgenutzt werden.

Zusätzliche Befehle haben dagegen einen negativen Einfluss auf das Ergebnis. Einerseits verbrauchen zusätzliche Befehle Platz, der unter Umständen gerade bei sehr kleinen Scratchpad-Speichern knapp ist. Durch die unterschiedliche Größe kann der Basisblock selbst oder ein anderes Objekt, das dadurch im Verhältnis einen schlechteren Nutzen aufweist, keinen Platz im Scratchpad-Speichern erhalten.

Findet ein Basisblock keinen Platz im Scratchpad-Speicher, so ist der Gewinn nicht so groß wie vom Modell vorhergesagt. Je häufiger der Basisblock ausgeführt wird, desto größer ist die Abweichung vom Modell. Der schlimmste Fall tritt ein, wenn ein Array, das für den Scratchpad-Speicher vorgesehen ist, keinen Platz mehr findet, dann fällt der vom Modell geplante Nutzen weg und der durch Tiling verursachte Overhead kann nicht aufgewogen werden.

Im Folgenden werden drei wesentliche Ursachen, die Quelle zusätzlicher Befehle sind, näher beschrieben.

#### 5.2.3.1 Sprungbefehle

Werden aufeinander folgende Basisblöcke nicht im selben Speicherbereich, also beide im Hauptspeicher oder beide im Scratchpad-Speicher abgelegt, so werden BL-Sprünge eingefügt. Zusätzliche Sprünge werden im Modell aber nur dann eingeplant, wenn das Modell nicht in der Lage ist, aufeinanderfolgende Basisblöcke des Originalprogramms im selben Speicher anzuordnen.

Das getilte Programm kann bis zu doppelt so viele Schleifen enthalten, kann also mehr als viermal so viele Basisblöcke besitzen. Immer wenn in den äußeren Schleifen zusätzliche BL-Befehle bei der Übersetzung des getilten Code eingefügt werden, konnten diese nicht durch das Modell berücksichtigt werden. Dabei werden die Kosten der Ausführung nicht berücksichtigt. Da es sich aber um äußere Schleifen handelt, ist ihr Anteil an den Gesamtkosten relativ klein, so dass die Kosten selbst dadurch nicht wesentlich beeinflusst werden. Für jeden dieser BL-Befehle werden 4 zusätzliche Byte belegt. Diese 4 Byte stehen im Scratchpad-Speicher nicht mehr zur Verfügung.

#### 5.2.3.2 Erzeugung von Konstanten

Für die Erzeugung von Konstanten existieren verschiedene Methoden. Sie können, wenn sie sehr klein sind, direkt im Befehl angegeben werden (immediates). Für größere Konstanten existieren zwei Möglichkeiten. Einerseits kann eine Befehlsfolge gefunden werden, die aus Konstanten im Befehl und arithmetischen Operationen den Wert erzeugt, andererseits kann die Konstante im Programm, in einem so genanntem Literalpool, gespeichert werden und durch eine Lade-Operation aus dem Speicher geholt werden. Die letzte Möglichkeit wird bei der Berechnung der Array-Elementadresse verwendet, die Basisadresse eines jeden Arrays wird in einem Literalpool gespeichert.

Bei Operationen auf zwei verschiedenen Registern kann die im Befehl verwendete Konstante zwischen -7 und 7 liegen. Wird dasselbe Register als Quelle und Ziel verwendet kann eine Konstante zwischen -255 und 255 im Befehl codiert werden [Adv01a]. Konstanten außerhalb dieser Bereiche können durch mehrere Operationen zusammengesetzt werden, soll die Konstante 1000 in das Register 0 geladen werden, kann dies auf folgende Weisen geschehen:

```
MOV  r0, #250        MOV r0,#250            LDR r0, _MAA_1
LSL  r0, r0,#2       ADD r0,r0,r0
                     ADD r0,r0,r0

                                            _MAA_1  DCD 0x000003E8
                                            _M_1    DCD 0x000003E8
```

Die zweite Möglichkeit ist ein Beispiel dafür, dass die Befehlsfolge nicht eindeutig sein muss. Das mittlere Beispiel wird vom encc nicht verwendet, weil eine LSL-Operation (Logisches Schieben nach Links) weniger kostet als zwei Additionen.

Der encc ist in der Lage, zwischen allen Möglichkeiten abzuwägen. Die genauen Kosten werden auch davon beeinflusst, ob der Code im Scratchpad-Speicher liegt oder nicht. Die Möglichkeit, von Zwischenergebnissen zu profitieren, hat auch Einfluss darauf, welcher Code generiert wird.

Konstanten werden durch die Programmtransformation in den Schleifenköpfen eingefügt oder verändert. Weil im Frontend nicht klar ist, welche Befehlsfolge gewählt wird, kann es zu Abweichungen kommen.

#### 5.2.3.3 Entfernung redundanter Ausdrücke

Der encc ist ein optimierender Compiler, der redundante Berechnungen innerhalb von Basisblöcken entfernen kann. Ein sehr großer Basisblock ist die innerste Schleife eines jeden Benchmarks. Häufig ergeben sich Optimierungen im Originalprogramm. Wenn z. B. die Array-Zugriffsfunktionen in einer Dimension übereinstimmen und die Arrays die gleiche Größe in der Dimension haben, kann ein Unterausdruck eingespart werden.

**Beispiel 5.2.1: Entfernung redundanter Ausdrücke**

In Abbildung 5.3 ist anhand eines Beispiels der Effekt dargestellt. Der generierte Code auf der linken Seite besitzt zwei Befehle mehr. Oben sind die relevanten deklarierten Arrays angegeben, darunter der übersetzte Ausdruck, der mit dem darunterliegenden Assembler-Programm korrespondiert. Die Unterschiede im Assembler-Programm sind fett hervorgehoben.

Auf der rechten Seite können zwei Befehle eingespart werden, da die Arrays `Cs` und `As` in der ersten Dimension die gleiche Zugriffsfunktion und die gleiche Dimension haben.

Zunächst wird in beiden Fällen der Offset für die zweite Dimension des Arrays `Cs` berechnet. Auf der rechten Seite wird dieses Ergebnis (`r3`) bei der Berechnung der Adresse von `As` ein weiteres Mal verwendet.

Im ersten Abschnitt von „access `Cs`“ wird in beiden Fällen die Adresse des Elements `Cs[j][k]` berechnet und in `r4` gespeichert.

Die zwei fett gedruckten zusätzlichen Befehle auf der linken Seite berechnen den Offset für das Array `As` in der ersten Dimension. Es müssen $j$-mal 3 Elemente zu je 4 Byte übersprungen werden. Dann ist schließlich auch auf der linken Seite in Register `r3` der Offset für das Array `As` in der zweiten Dimension.

Wenn die Entfernung redundanter Ausdrücke im getilten Programm nicht mehr möglich ist, kommen Befehle hinzu, die im Modell nicht erwartet wurden, da nur die Kosten des Originalprogramms berücksichtigt wurden. Da diese zusätzlichen Befehle meistens in der innersten Schleife auftreten, haben sie einen großen Einfluss auf den Gesamtenergieverbrauch und können das Ergebnis signifikant beeinflussen.

Der umgekehrte Fall ist ebenfalls möglich. Die Arrays können im Original unterschiedliche Dimensionen haben, dann ist eine Optimierung im Original nicht möglich. Wenn dann aber im getilten Programm die kleinen Arrays aufgrund gleicher Tiling-Faktoren und Zugriffsfunktionen die gleiche Dimension erhalten, ist eine Optimierung möglich, die nicht erwartet wurde. Der Energieverbrauch wird durch diesen Effekt weiter sinken.

Da diese Teile der Code-Generierung im Modell nicht berücksichtigt werden, kann das Programm nicht daraufhin optimiert werden. Das Auftreten dieser Effekte ist daher zufällig.

### 5.2.4 Genauigkeit des Modells

Ein gutes Modell beschreibt in möglichst guter Annäherung die Realität, bei vertretbarem Aufwand der Modellbildung.

Die wesentlichen Aspekte des Modells sind die Code-Größe und die Kosten. In den Abschnitten 5.2.4.2 und 5.2.4.3 wird die Speicherbelegung betrachtet und anschließend der modellierte Energieverbrauch mit dem durch Simulation erhaltenen Ergebnis verglichen. Auf die Genauigkeit des Gesamtsystems hat auch die Verwendung des genetischen Algorithmus einen Einfluss, der im nächsten Abschnitt beschrieben wird.

| Deklaration: | int As[2][3]<br>int Cs[2][15] | | int As[5][5]<br>int Cs[5][5] | |
|---|---|---|---|---|
| Ausdruck: | Cs[j][k]=Cs[j][k]+As[j][i] | | | |
| Code<br>zum<br>Ausdruck: | MOV r3,#60<br>MUL r3,r0 | r0:j | MOV r3,#20<br>MUL r3,r0 | r0:j |
| | | access Cs | | access Cs |
| | LDR r4, _MAA_4<br>ADD r4,r4,r3<br>LSL r5,r1,#2<br>ADD r4,r4,r5<br>**MOV r3,#12**<br>**MUL r3,r0** | r1:k | LDR r4, _MAA_4<br>ADD r4,r4,r3<br>LSL r5,r1,#2<br>ADD r4,r4,r5 | r1:k |
| | | access As | | access As |
| | LDR r6, _MAA_5<br>ADD r6,r6,r3<br>LSL r3,r2,#2<br>... | r2:i | LDR r6, _MAA_5<br>ADD r6,r6,r3<br>LSL r3,r2,#2<br>... | r2:i |

Abbildung 5.3: Gegenüberstellung des Assembler-Codes zu einem Ausdruck bei unterschiedlichen Array-Deklarationen

#### 5.2.4.1 Einfluss des genetischen Algorithmus auf die Genauigkeit

Der genetische Algorithmus findet lokale Optima. Da die Veränderungen zufällig durchgeführt werden, kann es passieren, dass der genetische Algorithmus knapp neben einem lokalen Optimum die Suche abbricht. Die Genauigkeit des gesamten Systems hängt auch davon ab, ob der genetische Algorithmus ein Optimum findet, das so auch realisiert wird. Diese Situation macht sich wie folgt bemerkbar:

Bei 256 Byte Scratchpad-Speichergröße wurden für zwei Läufe des Benchmarks Matrix-Multiplikation mit 14 Elementen je Dimension zwei unterschiedliche Ergebnisse gefunden. Die erste Kombination, laut Modell leicht schlechter (-0,37% Verbesserung bei der Betrachtung der modellierten Energie), erzielt in der Realität das wesentlich bessere Ergebnis (14% Verbesserung bei der Betrachtung der Simulation). Gewählt wurden die Tiling-Faktoren 1, 14 und 14. Es ist ebenso optimierbar wie das Originalprogramm und besitzt 22 Befehle in der innersten Schleife. Für das zweite Ergebnis wählt das Modell die Tiling-Faktoren 1, 7 und 14. Aufgrund der unterschiedlichen Dimensionen der Arrays werden vier zusätzliche Befehle in der innersten Schleife benötigt. In Tabelle 5.5 ist für die beiden Programme aufgeschlüsselt, welche Code-Teile zum Energieverbrauch beitragen. Für das zweite Programm ist die Verschlechterung durch die zusätzlichen Befehle in der innersten Schleife und durch den zusätzlichen Tiling-Overhead erklärbar.

Wie kann das Modell die kleineren Tiling-Faktoren besser bewerten? Die Antwort findet sich in der Belegung des Scratchpad-Speichers, der durch das Modell vorgenommen wurde. Diese Belegung ist nicht optimal. Statt die Kopierfunktion in den Scratchpad-Speicher zu verlegen, wurde die sehr große äußere Tiling-Schleife in den Scratchpad-Speicher verschoben.

Für die Tiling-Faktoren 1-14-14 hat auch das Modell in einem weiteren Durchlauf eine günstigere Speicherbelegung gefunden. Dann sind die Kosten im Modell 13% besser als die Tiling-Faktoren 1-7-14. Folglich kann das Modell, wenn die optimale Speicheranordnung gefunden wird, den Gewinn sehr gut vorhersagen.

Im Folgenden wird die Genauigkeit der Speicherbelegung und der Modellierung des Energieverbrauchs am Beispiel vorgeführt.

| | Energie [$\mu$J] | | Ausführungen | |
|---|---|---|---|---|
| Bereich / Tiling-F. | 1-14-14 | $1-7-14$ | 1-14-14 | 1-7-14 |
| copy | 36,45 | 66,01 | 42 | 70 |
| copy_const | 1,84 | 3,06 | 42 | 70 |
| copy_loop | 71,70 | 71,46 | 588 | 588 |
| i | 2,09 | 1,86 | 14 | 28 |
| iT | 28,95 | 17,97 | 14 | 14 |
| j | 28,04 | 13,91 | 196 | 196 |
| jT | 0,00 | 60,31 | - | 28 |
| k | 777,56 | 876,03 | 2744 | 2744 |
| main | 0,96 | 0,96 | 1 | 1 |

Tabelle 5.5: Energieverbrauch und Ausführungshäufigkeiten für Code-Teile des Benchmarks Matrix-Multiplikation mit 14 Elementen für zwei verschiedene Tiling-Faktoren, dynamisches Profiling

#### 5.2.4.2 Speicherbelegung

Anhand eines Beispiels, in dem unerwartet eine Verschlechterung aufgetreten ist, wird die Speicherbelegung betrachtet. Dabei wird gegenübergestellt, welche Code-Teile vom Modell im Scratchpad-Speicher angenommen werden und welche Code-Teile tatsächlich abgelegt wurden.

Bei einer Scratchpad-Speichergröße von 512 Byte wird bei Übersetzung des getilten Programms für das gefundene Optimum im Modell eine Verschlechterung erzielt. In Tabelle 5.4 ist zu erkennen, dass das Array `Cs` nicht in den Scratchpad-Speicher verschoben wurde.

Für das Programm wurden die Tiling-Faktoren 5 in jeder Schleife gewählt. Durch den enProfiler wird aber eine Verschlechterung um 5% ausgewiesen, wenn das Originalprogramm und das getilte Programm mit statischer Analyse durch den encc übersetzt werden. Alle Arrays sind von der gleichen Form, so dass der Code in der innersten Schleife die gleiche Struktur wie im Original besitzt. Er wird genauso wie im Original optimiert, wie es vom Modell erwartet wird. Der Verlust ergibt sich dadurch, dass der encc nicht in der Lage ist, das Array `C` in den Scratchpad-Speicher zu verschieben.

512 Byte Scratchpad-Speicher stehen insgesamt zur Verfügung. 488 Bytes können vom encc nach Reservierung des Platzes für Literalpools belegt werden. Nach der Optimierung hat der encc 476 Bytes belegt. 276 Byte hat der encc mit Code belegt und 200 Byte mit zwei Arrays. Es sind noch 12 Bytes frei. Ein 100 Byte Array hat der encc nicht in den Scratchpad-Speicher verschoben.

Nun zur Belegung des Modells. Für die Kopierfunktion wurden, wie im Modell angenommen, 54 Bytes verbraucht. Das Modell hat weitere 134 Byte mit Code belegt. Das Modell geht davon aus, dass 188 Byte mit Code belegt werden. Die restlichen 300 Byte werden mit den drei Arrays belegt.

Dem Moell nach hat der encc 88 Byte Code zu viel in den Scratchpad-Speicher verschoben. Die folgenden Unterschiede zwischen der Belegung des encc und dem Modell wurden vom Modell nicht erwartet:

- Der encc hat ein 100 Byte Array nicht in den Scratchpad-Speicher verschoben.
- Der encc hat 12 Byte frei gelassen.
- Das Modell ist davon ausgegangen, dass die zwei innersten Tiling-Schleifen nicht im Scratchpad-Speicher abgelegt werden. Sie enthalten die Aufrufe der Kopierfunktion und enthalten daher mehr Code als die anderen nicht innersten Schleifen. Der encc hat die beiden äußeren Tiling-Schleifen und Teile der Elementschleifen nicht im Scratchpad-Speicher abgelegt. Dieser Unterschied ist verantwortlich für 68 Bytes.

| | staisches Profiling | | dynamisches Profiling | |
|---|---|---|---|---|
| | Original | getiled: 5-5-5 | Original | getiled: 5-5-5 |
| Energie [$\mu J$] | 1595.48 | 1689.42 | 1595.47 | 1140.94 |
| Verbesserung [%] | | −5.89 | | 28.49 |

Tabelle 5.6: Gegenüberstellung des Energieverbrauchs für dynamisches Profiling und statische Analyse. Der Scratchpad-Speicher ist 512 Byte groß, Matrix-Multiplikation mit 15 Elementen

- Weiterer Platz wurde durch den encc für BL-Befehle verwendet, die im Modell nur zwischen Original-Basisblöcken berücksichtigt werden. Insgesamt belegen 5 BL-Befehle 20 Byte im Scratchpad-Speicher, die im Modell nicht berücksichtigt werden.

Durch die Unterschiede sind $12 + 68 + 20 = 100$ Byte erklärbar. Das Modell hat in seinen Grenzen die Größen der Belegung richtig abgeschätzt. In diesem Fall sind die 5 BL-Befehle durch das Modell nicht vorhergesehen worden. Die anderen Unterschiede sind Resultat einer ungenauen Bewertung durch die statische Analyse.

Um zu belegen, dass die Verschlechterung durch eine günstigere Belegung des Scratchpad-Speichers vermieden werden kann, wird dynamisches Profiling verwendet. Dynamisches Profiling liefert für alle Objekte, die in den Scratchpad-Speicher verschoben werden können, einen sehr genauen Nutzen. Für jeden Basisblock und Array ist die genaue Anzahl der Zugriffe bekannt. Durch Tabelle 5.6 ist belegt, dass die Verschlechterung, die in Abbildung 5.2 bei 512 Byte Scratchpad-Speicher für das Optimum des Modells zu erkennen ist, auf die falsche Gewichtung des Nutzen durch den encc zurückzuführen ist. Für das Originalprogramm macht es einen vernachlässigbar kleinen Unterschied, dynamisches Profiling zu verwenden. Für das optimierte Programm dagegen wird statt einer Verschlechterung um 6% eine Verbesserung um 28% mit dynamischen Profiling erreicht.

#### 5.2.4.3 Modellierung der Energie

Um festzustellen, wie gut die modellierten Kosten mit den realen Kosten für einen Benchmark übereinstimmen, wird folgendes Verfahren angewendet.

Für die verschiedenen Scratchpad-Speichergrößen wird das jeweils beste Ergebnis im Modell und der dazu korrespondierende simulierte Energieverbrauch in ein X-Y-Diagramm eingetragen. Je genauer das Modell den tatsächlichen Energieverbrauch abbildet, desto genauer liegen alle Punkte auf einer Geraden. Abbildung 5.4 zeigt dies exemplarisch für den Benchmark der Matrix-Multiplikation mit 15 Elementen.

In das Diagramm ist eine Ausgleichsgerade eingezeichnet, so dass die Punkte mit größerer Abweichung deutlich werden. Für Punkte oberhalb der Geraden hat das Modell ein besseres Ergebnis erwartet. Abweichungen existieren bei den Scratchpad-Speichergrößen 64 und 128 Byte.

Bei 64 Byte sind die Abweichungen dadurch zu erklären, dass BL-Befehle nicht richtig berücksichtigt werden und somit nicht die gleichen Basisblöcke, wie vom Modell angenommen, in den Scratchpad-Speicher verschoben werden können.

Bei 128 Byte Scratchpad-Speicher wurde das Array `Cs` trotz dynamischen Profiling nicht in den Scratchpad-Speicher verschoben. Der zur Verfügung stehende Speicher reicht nicht aus.

Das Modell ist exakt, wenn alle Punkte der Abbildung 5.4 auf einer Geraden liegen. Der Korrelationskoeffizient ist ein Maß der Regressionsanalyse, der genau diesen Sachverhalt beschreibt. Ein Korrelationskoeffizient nahe 1 bedeutet einen linaren Zusammenhang. Ein Korrelationskoeffizient von 0 bedeutet, dass kein Zusammenhang existiert [RF92, GKN81]. Für den Benchmark der Matrix-Multiplikation mit 15 Elementen je Dimension kann auf Grundlage der Daten für Abbildung 5.4 ein Korrelationskoeffizient von 0,98 bestimmt werden. Über alle Ergebnisse, die im Diagramm in Abbildung 5.16 verwendet wurden, ist der Korrelationskoeffizient ebenfalls 0,98.

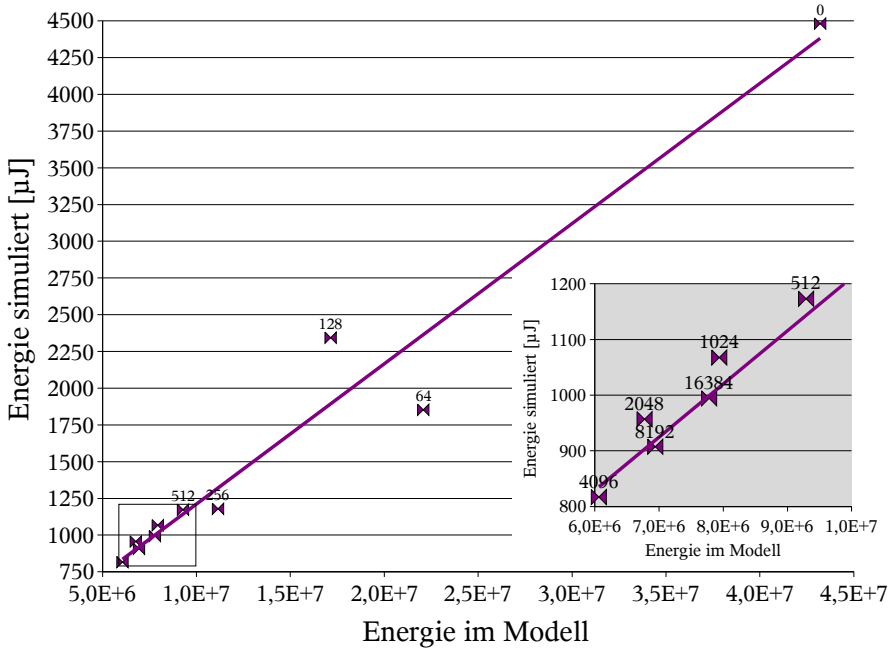


Abbildung 5.4: Zusammenhang zwischen dem modellierten Energieverbrauch und dem durch Simulation bestimmten Energieverbrauch, Matrix-Multiplikation mit 15 Elementen je Dimension

Die Analyse der Genauigkeit des Modells ist nun abgeschlossen. Es wird nun betrachtet, wie sich durch Tiling andere Größen verändern, die nicht Ziel der Optimierung sind.

## 5.3 Weitere Betrachtungen

Weitere wesentliche Eigenschaften eines übersetzen Programms sind die Code-Größe und die Laufzeit. Die Code-Größe ist bei eingebetteten Systemen besonders relevant, da die Hardware meistens speziell angefertigt wird. Ein kleiner Speicher führt zu geringeren Kosten.

Die Anzahl ausgeführter Instruktionen verbindet die Code-Größe und die Laufzeit. Wenn mehr Instruktionen ausgeführt werden, erhöht sich in der Regel die Laufzeit. Wenn der Code größer wird, ist es wahrscheinlich, dass auch mehr Befehle ausgeführt werden.

Die Laufzeit spielt dann eine Rolle, wenn der Nutzer auf das Resultat warten muss und wenn Echtzeitanforderungen aufgrund einer Verlängerung der Laufzeit nicht mehr eingehalten werden können.

Die drei Eigenschaften werden im Folgenden am Beispiel der Matrix-Multplikation mit 15 Elementen je Dimension betrachtet.

### 5.3.1 Code-Größe

Die Code-Größe wird in Byte gemessen und ist in absoluten Zahlen in Tabelle 5.7 angegeben. Die Code-Größe im Vergleich zum Originalprogramm ist 2,8 bis 4,5 Mal so groß wie ohne Tiling. Der größere Teil der Instruktionen wird im Scratchpad-Speicher gehalten. Bei dynamischem Profiling ändert sich die

| | Code-Größe [Byte] | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Orginal | | | getiltes Programm | | | | | | | |
| | statische Analyse | | | statische Analyse | | | | dynamisches Profiling | | | |
| SP -Größe[Byte] | HS | SP | Summe | HS | SP | AC [%] | Summe | HS | SP | AC [%] | Summe |
| 0 | 80 | | 80 | 80 | | | 80 | 80 | | | 80 |
| 64 | 72 | 32 | 104 | 72 | 32 | 50 | 104 | 68 | 32 | 50 | 100 |
| 128 | | 88 | 88 | 132 | 120 | 94 | 252 | 128 | 124 | 97 | 252 |
| 256 | | 88 | 88 | 164 | 120 | 47 | 284 | 136 | 128 | 50 | 264 |
| 512 | | 88 | 88 | 108 | 276 | 54 | 384 | 140 | 212 | 41 | 352 |
| 1024 | | 80 | 80 | 52 | 312 | 30 | 364 | | 304 | 30 | 304 |
| 2048 | | 80 | 80 | 4 | 228 | 11 | 232 | | 224 | 11 | 224 |
| 4096 | | 80 | 80 | | 80 | 2 | 80 | | 80 | 2 | 80 |
| 8192 | | 80 | 80 | | 80 | 1 | 80 | | 80 | 1 | 80 |
| 16384 | | 80 | 80 | | 80 | | 80 | | 80 | | 80 |

Tabelle 5.7: Programmgröße des Benchmarks Matrix-Multiplikation mit 15 Elementen, Scratchpad-Speicher (SP), Hauptspeicher (HS), Anteil des Codes im Scratchpad-Speicher in Prozent (AC)

Code-Größe im Wesentlichen nicht, da dasselbe Programm übersetzt wird. Änderungen können sich ausschließlich durch eine andere Anzahl BL-Befehle ergeben, die zum Sprung zwischen Hauptspeicher und Scratchpad-Speicher notwendig sind. Die Code-Selektion ist vor der Allokation des Scratchpad-Speichers abgeschlossen.

Bei 512 Byte Scratchpad-Speicher fällt auf, dass der encc mit dynamischem Profiling weniger Code im Scratchpad-Speicher abgelegt hat; dadurch hat er Platz für das Array gelassen, das mit Verwendung statischer Analyse im Hauptspeicher belassen wurde.

Der Anteil des Codes, der im Scratchpad-Speicher abgelegt wird, nimmt mit der Größe des Scratchpad-Speichers ab. Bei größeren Scratchpad-Speichern nimmt der Anteil des Code drastisch ab. Bei großen Scratchpad-Speichern wird mit Tiling weniger als 3% des Speichers mit Code belegt. Das Maximum mit 97% bei 128 Byte Scratchpad-Speichern muss etwas relativiert werden, da der encc auch beim dynamischen Profiling hier ein Array nicht in den Scratchpad-Speicher verschoben hat. Bei Ergebnissen mit Gewinn wird höchstens die Hälfte des Scratchpad-Speichers mit Code belegt.

### 5.3.2 Anzahl ausgeführter Instruktionen

In Abbildung 5.5 ist die (durchweg negative) Verbesserung der Anzahl ausgeführter Instruktionen für verschiedene Scratchpad-Speicher aufgetragen. Der durch Tiling eingefügte Overhead macht sich in der Anzahl ausgeführter Befehle deutlich bemerkbar. Anhand des Vergleichs der optimalen Ergebnisse im Modell (opt-Modell) in Abbildung 5.5 und Abbildung 5.2 ist zu erkennen, dass immer dann besonders wenig Gewinn erzeugt wird, wenn besonders viel Overhead vorliegt. Der geringe Overhead bei 256 Byte Scratchpad-Speicher ist durch die besondere Wahl der Tiling-Faktoren zu erklären. Es wurde nur die äußere Schleife mit dem Tiling-Faktor 1 getiled. Dies bewirkt, dass im resultieren Nest keine Schleife öfter als im Original ausgeführt wird. Es wurden keine Schleifen vertauscht. Der Overhead, der in der äußersten Schleife entsteht, wird nur 15-mal ausgeführt.

### 5.3.3 Laufzeit

In Abbildung 5.6 ist die Verbesserung der Anzahl ausgeführter Zyklen aufgetragen. Bei 256 Byte Scratchpad-Speicher ist eine Laufzeitverbesserung zu beobachten. Bei einer Variante für 512 Byte ist ebenfalls eine

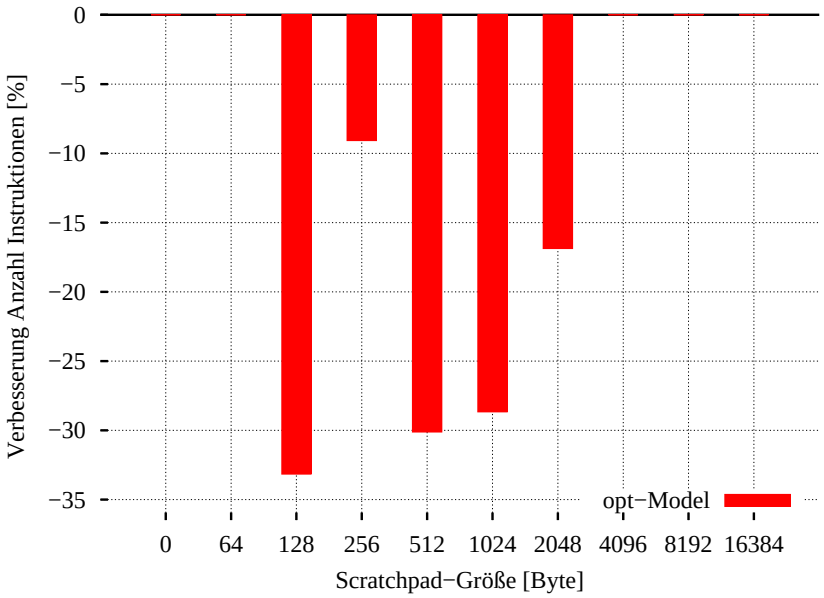


Abbildung 5.5: Verbesserung der Anzahl Instruktionen, Matrix-Multiplikation mit 15 Elementen, statische Analyse

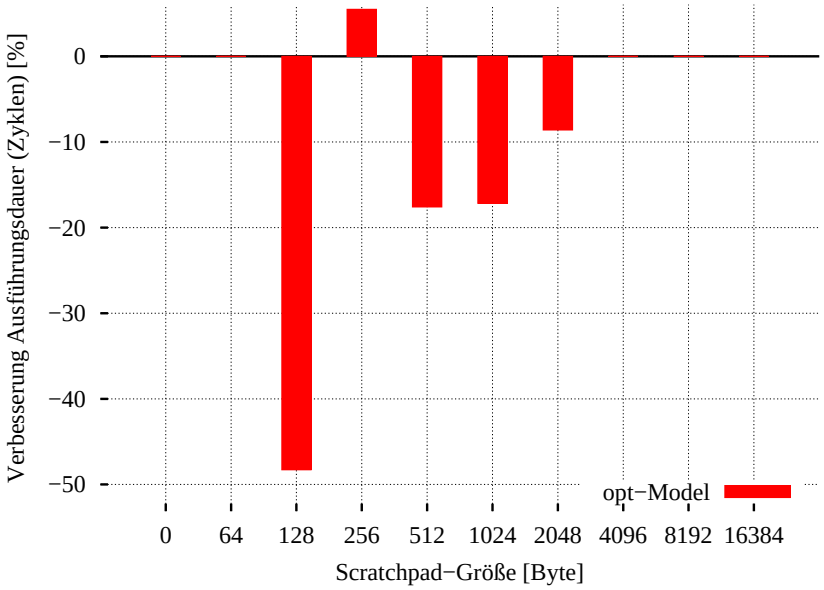


Abbildung 5.6: Verbesserung der Ausführungszyklen, Matrix-Multiplikation mit 15 Elementen, statische Analyse

Laufzeitverbesserung, hier um 6%, zu beobachten. Beide Fälle haben gemein, dass nur die äußerste Schleife getiled wurde: im einen Fall mit dem Tilingfaktor 1 und im anderen Fall mit 2. Ein Zusammenhang mit dem in Abbildung 5.5 zu erkennenden geringen Instruktions-Overhead ist naheliegend. Insgesamt sind die Verschlechterungen der Laufzeit im Vergleich zur Verschlechterung der Anzahl ausgeführter Instruktionen um etwa 10% kleiner. Auf Scratchpad-Speicher kann im Vergleich zum Hauptspeicher schneller zugegriffen werden, so dass das Programm schneller ausgeführt werden kann. Die Ausführung von zusätzlichen Instruktionen sorgt dafür, dass die Veränderung der Energie und Laufzeit nicht direkt zusammenhängen.

Die Ergebnisse für Scratchpad-Speicher mit einer Größe von 1024 und 2048 Byte zeigen, dass im Vergleich zum durch den encc übersetzten Originalprogramm eine Laufzeitzunahme stattgefunden hat. Der Gewinn im Energieverbrauch ist also nicht durch eine Laufzeitreduktion unterstützt worden. Der Gewinn ist ausschließlich durch die geringen Zugriffskosten auf den Speicher zurückzuführen. Der Anteil der Zugriffe auf den Scratchpad-Speicher ist, wie den Tabellen 5.2 und 5.3 zu entnehmen ist, höher, da alle Arrays und der Großteil des Code im Scratchpad-Speicher liegen.

Abbildung 5.6 ist eine Bestätigung dafür, die Energie selbst als Ziel für die Optimierung zu wählen und nicht die Laufzeit. Es kann sich lohnen, Zeit zu investieren, Daten an einen Ort zu verschieben, an dem auf sie effizient zugegriffen werden kann. Trotz längerer Laufzeit kann Energie eingespart werden. Der unterschiedliche Effekt auf Laufzeit und Energie führt aber auch dazu, dass, wenn ein Gesamtsystem auf Energieverbrauch optimiert werden soll, auch das gesamte System modelliert werden muss.

### 5.3.4 Energieverbrauch im Prozessor und Speicher

Das verwendete Modell ist in der Lage, die durch CPU und Speicher verbrauchte Energie getrennt aufzuschlüsseln. In Abbildung 5.7 sind die absoluten Energieverbräuche für Speicher (MEM) und Prozessor (CPU) für das durch das Modell optimierte Programm und das ausschließlich durch den encc optimierte Programm für verschiedene Scratchpad-Speicher dargestellt. Die Fälle für 0 und 64 Byte Scratchpad-Speicher wurden weggelassen, da die Balken sehr groß im Vergleich zu den anderen sind. Bei 128 Byte Scratchpad-Speicher wurde der größte Verlust für das vom Modell generierte Programm ausgewiesen. Sowohl der Energieverbrauch für Speicher als auch des Prozessors sind größer. Für 256 Byte Scratchpad-Speicher hat die Transformation auch Energie für die CPU gespart. Dies hängt mit der Verbesserung der Laufzeit zusammen. Die Einsparungen im Energieverbrauch für den Speicher sind hier mit 47% ebenfalls besonders hoch. In den anderen Fällen 512, 1024 und 2048 Byte Scratchpad-Speicher manifestiert sich der eingefügte Overhead auch in einem größeren Energieverbrauch der CPU. Bei 512 Byte Scratchpad-Speicher ist aber die Einsparung durch den Speicher zu klein, so dass kein Gewinn auftritt. Es wurde mehr Energie im Speicher verbraucht, weil ein Array nicht in den Scratchpad-Speicher verschoben wurde. Für größere Scratchpad-Speicher ab 4096 Byte passen die Original-Arrays und der Code in den Scratchpad-Speicher, der gestiegene Energieverbrauch ist auf die Größe des Scratchpad-Speichers zurückzuführen. Ein größerer Scratchpad-Speicher benötigt pro Zugriff mehr Energie.

## 5.4 Weitere Einflüsse auf die Resultate

Im Folgenden wird betrachtet, welchen Einfluss das Programm selbst auf die Verbesserung haben kann. Betrachtete Einflüsse sind die Größe der Arrays, die Schleifentiefe und die Dimension der Arrays.

### 5.4.1 Größe der benutzten Arrays

Eine um 1 kleinere Anzahl der Elemente in einer Dimension bewirkt, dass 13% weniger Daten verarbeitet werden. Dies ist der kleinste Schritt, in dem der Benchmark Matrix-Multiplikation verändert werden kann, ohne dass seine Semantik wesentlich geändert wird. Die Ergebnisse sind in Abbildung 5.8 zusammengefasst. Auffallend im Vergleich mit Abbildung 5.2 ist, dass bei einer Scratchpad-Größe von 1024

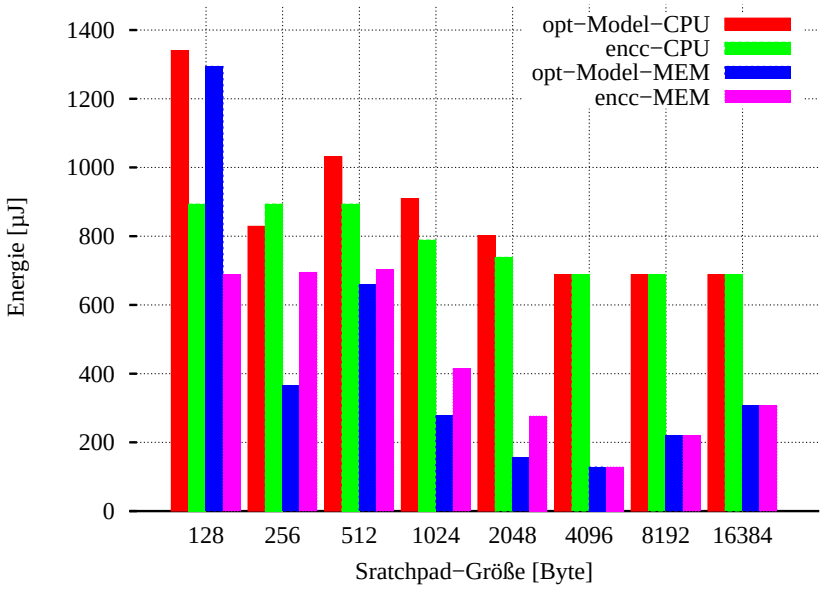


Abbildung 5.7: Energieverbrauch der CPU und des Speichers für das Optimum im Modell, Matrix-Multiplikation mit 15 Elementeni, statische Analyse

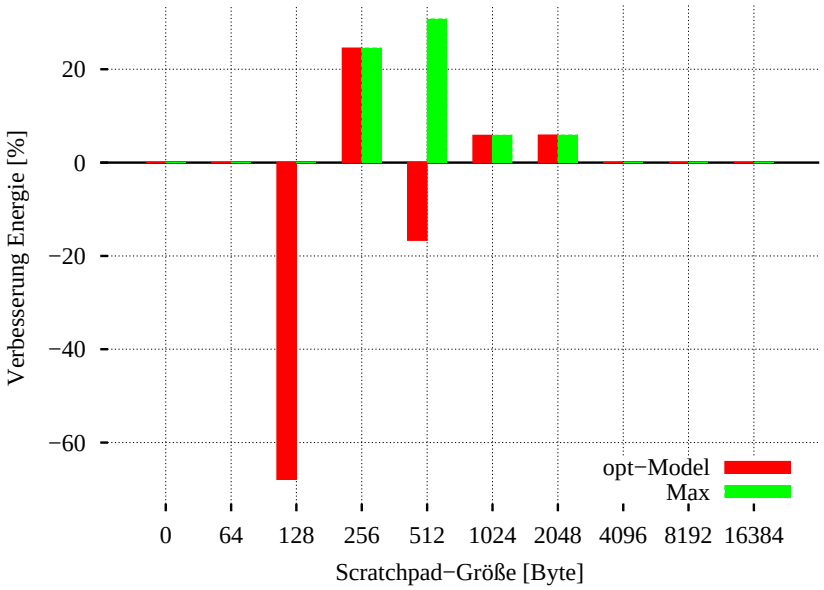


Abbildung 5.8: Verbesserung des Energieverbrauchs, Matrix-Multiplikation mit 14 Elementen, statische Analyse

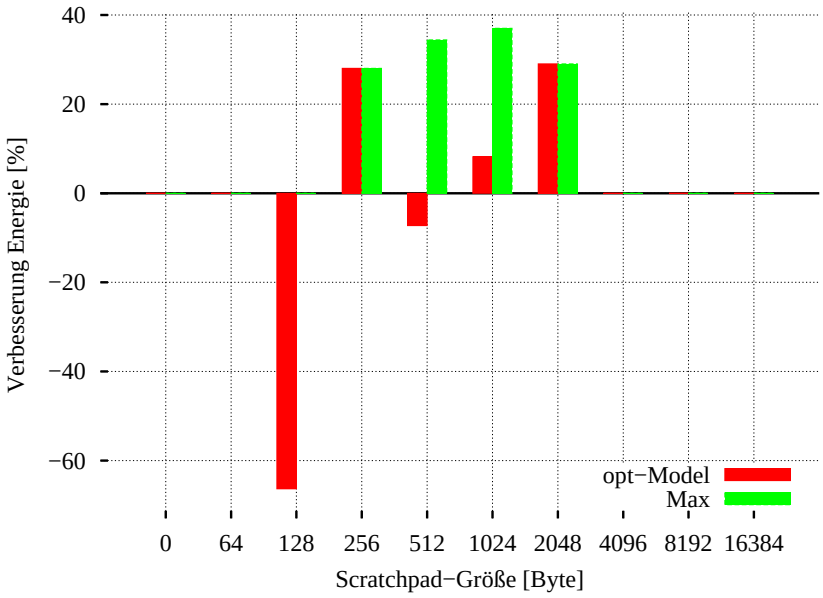


Abbildung 5.9: Verbesserung des Energieverbrauchs, Matrix-Multiplikation mit 16 Elementen, statische Analyse

Byte dieselbe Verbesserung auftritt wie bei 2048 Byte Scratchpad-Speicher beider Programmversionen. Bei 512 Byte Scratchpad-Speicher ist nur eine Parameterkombination optimal. Es wurden dieselben Verbesserungen erreicht wie bei 15 Elementen in jeder Dimension. Die maximale Verbesserung bei 256 Byte ist ebenfalls in der gleichen Größenordnung. Die vom Modell gefundene Verbesserung ist aber schlechter. Sowohl bei 256 als auch bei 512 Byte sind ausschließlich Verbesserungen aufgetreten. Bei 128 Byte ist ein sehr schlechtes Ergebnis durch das Modell ausgewählt worden.

Wie sind diese Unterschiede zu erklären? Naheliegend ist die Erkenntnis, dass es an der Teilbarkeit von 14 liegt. 14 hat die Teiler 2 und 7 im Gegensatz zu 15, welche die Teiler 3 und 5 enthält. 3 und 5 liegen nahe beieinander, so dass auch die Größen der getilten Arrays ähnlich sind. Da 2 kein Teiler von 15 ist, kann der Datenbereich nicht leicht in zwei Teile geteilt werden. Ein Tiling-Faktor, der den Datenbereich in zwei Teile teilt, verursacht den geringsten Overhead. Der Tiling-Faktor 7, der die Daten in zwei Teile teilt, kommt bei den Scratchpad-Größen 128, 512, 1024 und 2048 Byte in mindestens einem untersuchten Programm vor.

Bei 512 Byte ist der Tiling-Faktor der äußeren Schleife 2 und der der inneren 14. Die Arrays `C` und `A` haben die gleiche Größe und Zugriffsfunktion, so dass das modifizierte Programm wie das Originalprogramm optimiert werden kann. Die Größe der Arrays passt besonders gut zur Größe des Scratchpad-Speichers.

Große Arrays wirken sich positiv auf die Optimierbarkeit aus. Mit großen Arrays steigt auch die Wiederverwendbarkeit von Teil-Arrays, da die Schleifen mehr Iterationen ausführen. Wenn die Anzahl der Elemente je Dimension Zweierpotenzen sind, hat dies ebenfalls einen positiven Effekt auf den Energieverbrauch.

### 5.4.2 Schleifentiefe

Weitere wesentliche Eigenschaften der Programme sind die Tiefe des Schleifennestes und die Dimension der verwendeten Arrays. Zuerst wird die Tiefe des Schleifennestes variiert. Ein Tiefpass ist eine Anwen-

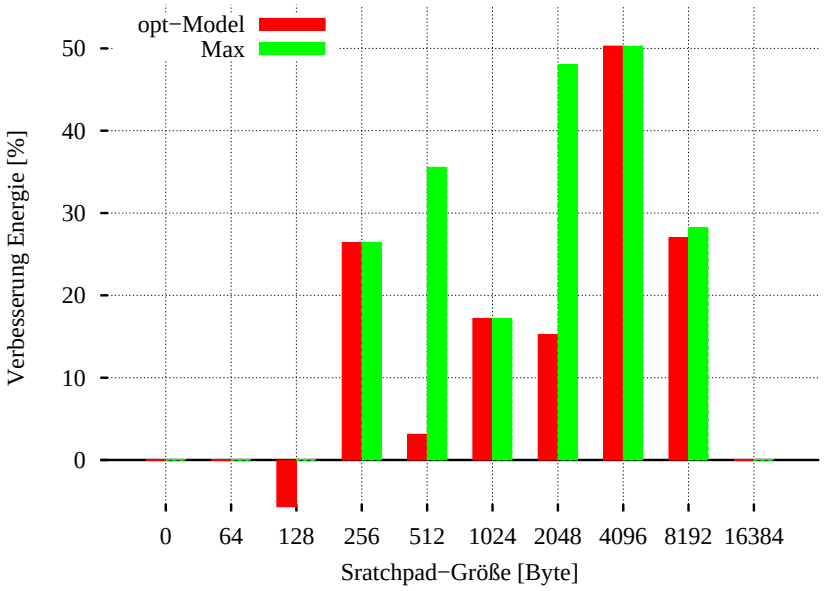


Abbildung 5.10: Verbesserung des Energieverbrauchs, Matrix-Multiplikation mit 32 Elementen, statische Analyse

dung, die häufig in der digitalen Signalverarbeitung eingesetzt wird. Zur Bildverbesserung kann er eingesetzt werden, um Bildrauschen zu unterdrücken oder einen Hochpass zu konstruieren, um das Bild zu schärfen [GW02]. Für kleine Masken ist es nicht sinnvoll, eine Fast-Fourier-Transformation zu verwenden, daher ist auch die Implementierung im Ortsbereich relevant. Folgendes Beispielprogramm kann den Kern eines Tiefpassfilters bilden:

```
for (x=0; x<N-D; x++) {
  for (y=0; y<N-D; y++) {
    for (bx=0; bx<D; bx++)
      for (by=0; by<D; by++)
        C[x][y] = C[x][y] + A[x+bx][y+by];
  }
}
```

Im Gegensatz zur Matrix-Multiplikation ist hier die Schleifentiefe 4. In Abbildung 5.11 sind die erzielten relativen Verbesserungen dargestellt. Für die Übersetzung wurde die Matrix-Größe $N = 15$ und eine Filtergröße von $D = 4$ gewählt. Da in diesem Beispiel nur zwei Matrizen vorkommen und auf die Eingabe-Matrix an den Rändern seltener zugegriffen wird, sind die maximalen Verbesserungen mit 16% eher klein. Bei 128 Byte Scratchpad-Speicher liefert eine Übersetzung mit dem encc und statischer Analyse eine Verschlechterung um 3%. Wird das generierte Programm hingegen mit dem dynamischen Profiling übersetzt, wird für 128 und 256 Byte Scratchpad-Speicher eine Verbesserung um 17 % erreicht. Die schlechten Ergebnisse bei statischer Analyse sind auf eine ungünstige Speicherbelegung zurückzuführen.

Mit der Schleifentiefe wurde auch die Zugriffshäufigkeiten aller Arrays verändert, daher ist ein direkter Vergleich mit der Matrix-Multiplikation nicht angebracht. Der Korrelationskoeffizient ist wie bei der Matrixmultiplikation mit 15 Elementen 0,98. Dies bedeutet, das Modell hat im Mittel dieselben Abweichungen. Im Vergleich zu den anderen Benchmarks werden in diesem Benchmark einzelne Arrays wesentlich

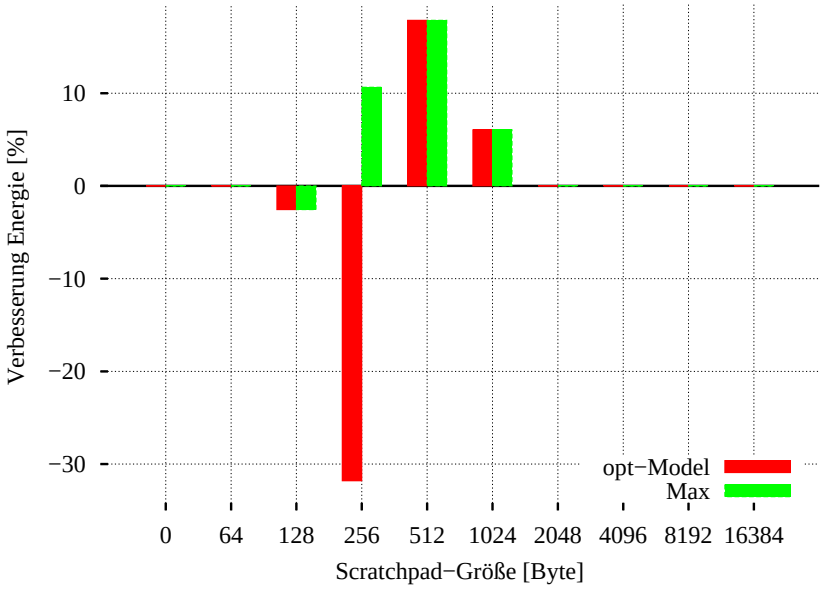


Abbildung 5.11: Verbesserung des Energieverbrauchs, Tiefpass von 14x14 Elementen mit Fenster 3x3, statische Analyse

seltener wieder verwendet, daher sind kleinere Verbesserungen zu erwarten. Die Schleifentiefe hat keinen direkten Einfluss auf die Optimierbarkeit.

Für eine Scratchpad-Größe von 512 Byte ist in der Abbildung 5.12 der Scatterplot angegeben, der mit Abbildung 4.17 auf Seite 88 vergleichbar ist. Es fällt auf, dass die lokalen Optima hier weiter verstreut sind. Durch eine größere Anzahl Schleifen sollte die Optimierung schwieriger werden, da mehr Freiheiten bestehen. Die Laufzeitverteilung in Abbildung 5.13 zeigt eine fast lineare Form, die zum Schluss abflacht. Es existiert also ein Bereich, in dem es sich lohnt, weiter auf ein gutes Ergebnis zu warten. Das Abflachen gegen Ende deutet darauf hin, dass insgesamt ausreichend lange gewartet wird.

### 5.4.3 Dimension der Arrays

Ein anderer Benchmark, der in dieser Form auch schon in anderen Arbeiten [LMD+04, VSM03, FV04] zur Evaluierung betrachtet wurde, ist der FIR-Benchmark. FIR steht für **F**inite **I**mpulse **R**esponse, deutsch endliche Impuls-Antwort. Die endliche Impuls-Antwort eines zeitinvarianten linearen Systems auf einen Dirac-Impuls kennzeichnet sein Signalverhalten. Es wird eine eindimensionale Faltung im Ortsbereich durchgeführt. Die Faltung ist ebenfalls eine Operation, die häufig in der digitalen Signalverarbeitung eingesetzt wird. Dieser Benchmark besitzt nur zwei Schleifen und eindimensionale Arrays. Die Arrays sind mit 401 und 40 Elementen sehr groß. Auf einzelne Elemente wird häufiger zugegriffen als in den anderen Programmen. Ein Element des Arrays b, das 40 Elemente groß ist, kann bis zu 401 mal wiederverwendet werden.

Der Benchmark besteht im Wesentlichen aus folgendem Schleifennest. Es ist notwendig, die Instruktionen der äußeren Schleifen mit einer Bedingung im getilten Programm vor erhöhter Ausführung zu schützen. Die Tiling-Faktoren, die für 256, 256, 1024 und 2048 Byte Scratchpad-Speicher gewählt wurden, hätten die Bedingung nicht benötigt.

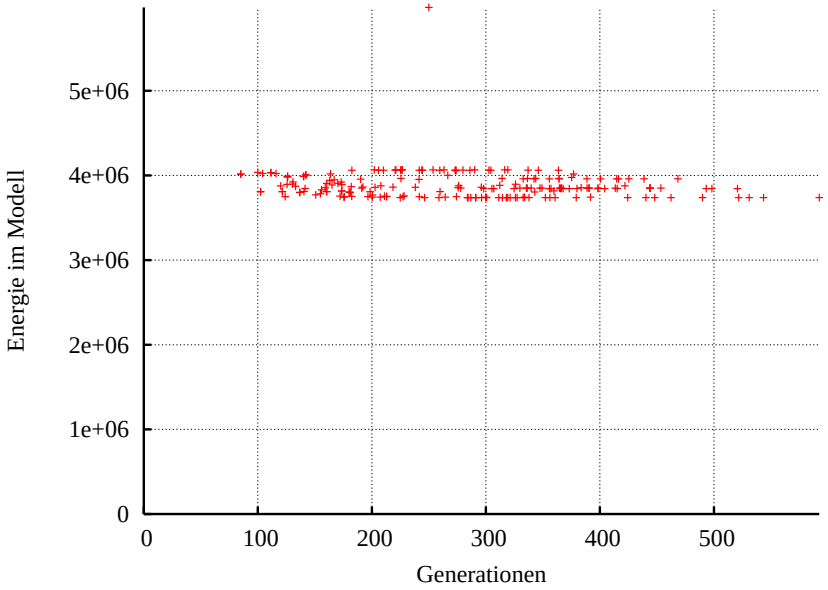


Abbildung 5.12: Scatterplot für den Tiefpass-Benchmark

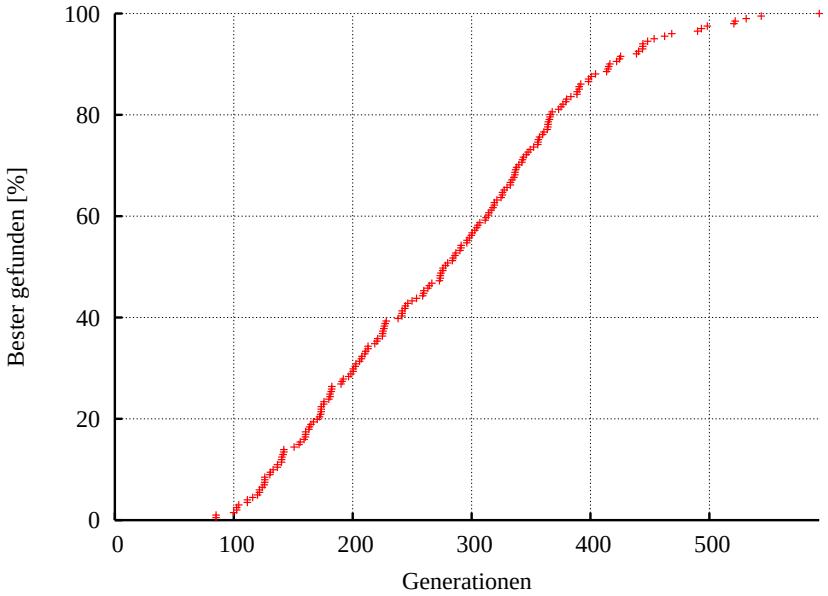


Abbildung 5.13: Laufzeitverteilung für den Tiefpass-Benchmark

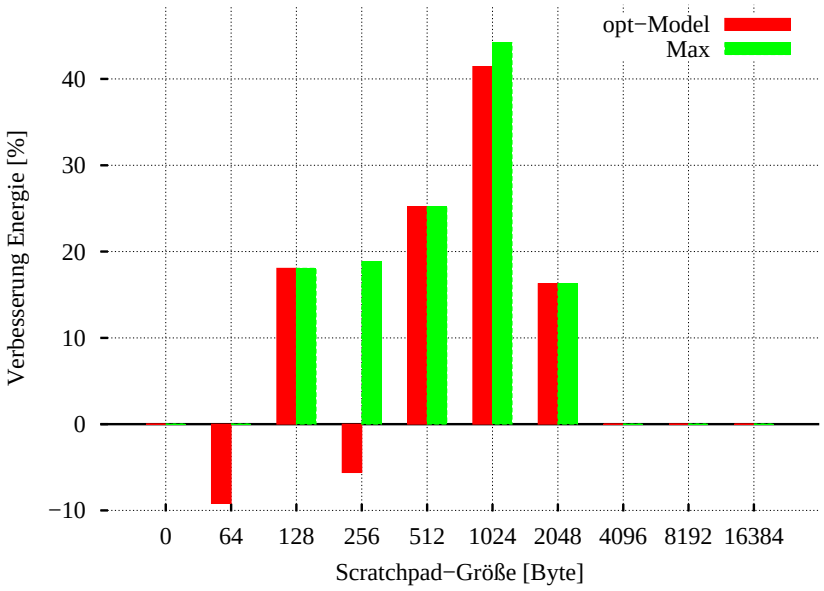


Abbildung 5.14: Verbesserung des Energieverbrauchs, Faltung von 401 Elementen mit 40 Elementen, statische Analyse

```
for (i=m; i<40+401; i++) {
  y[i] = 0;
  for (r=0; r<40; r++) {
    y[i] = y[i] + b[r] * x[i-r];
  }
  y[i] = y[i] >> sf;
}
```

Die Ergebnisse sind in Abbildung 5.14 dargestellt. Bei 256 Byte Scratchpad-Speicher ist der Grund für den Unterschied zwischen dem besten gefundenen Ergebnis und dem Optimum im Modell in der Belegung des Scratchpad-Speichers durch das Modell zu finden. Im Optimum des Modells werden alle Arrays verschoben. Für den anderen Fall hat der genetische Algorithmus ein Array nicht verschoben. Auch für diesen Benchmark sind die Verbesserungen erheblich besser, wenn dynamisches Profiling eingesetzt wird.

Auch bei eindimensionalen Arrays kann eine Verbesserung erzielt werden, die mit den anderen Ergebnissen vergleichbar ist. Dies liegt einerseits daran, dass auf das Array `b` sehr oft zugegriffen wird. Eindimensionale Arrays können beliebige Größen haben, so dass der Scratchpad-Speicher straff belegt werden kann. Aus diesem Grund hat das Modell für 64 Byte begonnen, das Array `ys` in den Scratchpad-Speicher zu schieben. Durch nicht vorhergesehene zusätzliche Instruktionen konnte die geplante Verbesserung nicht umgesetzt werden, weil zu wenig Speicher vorhanden ist. Das Modell hat weder die Kopierfunktion noch eine Tiling-Schleife in den Scratchpad-Speicher verschoben. Da das Modell den Scratchpad-Speicher besonders straff belegt und das Scratchpad komplett füllt, haben diese Ungenauigkeiten Auswirkungen auf das Ergebnis.

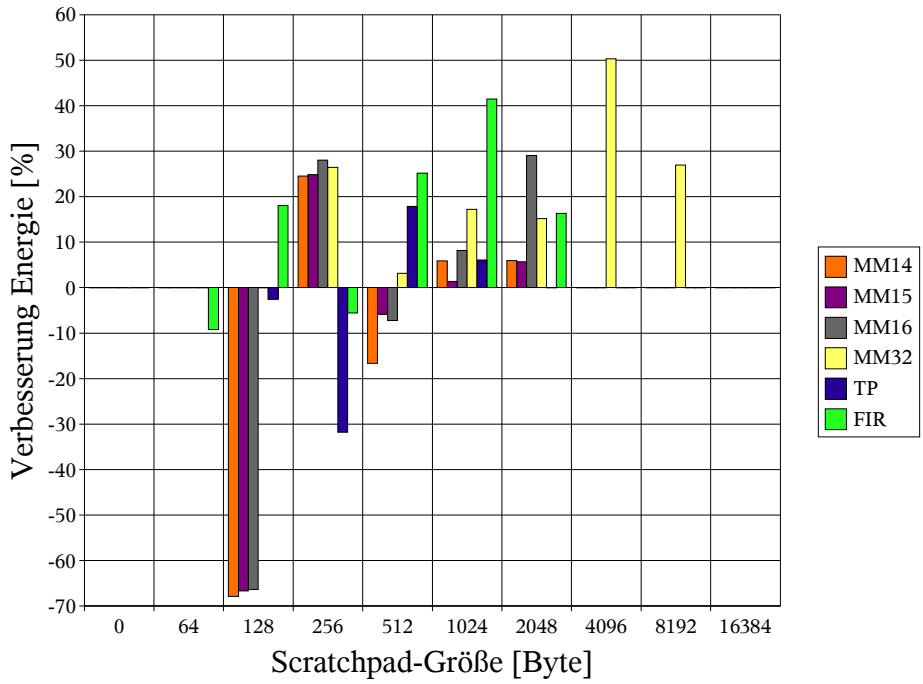


Abbildung 5.15: Verbesserung des Energieverbrauchs verschiedener Benchmarks bei Verwendung unterschiedlicher Scratchpad-Speicher. Die getilten Programme wurden mit statischer Analyse übersetzt.

## 5.5 Ergebnisse mit statischer Analyse

Die vorherigen Ergebnisse dienten der Erklärung, welche Effekte auftreten können. Der genetische Algorithmus wurde dazu mit einer Populationsgröße von 3000 Individuen und 3% zu ersetzenden Individuen ausgeführt. Hierbei wurde oft nicht das Optimum im Modell gefunden, sondern ein lokales Optimum, das sich in der Simulation als besser erweisen konnte.

Für die folgenden Ergebnisse wurde die Übersetzung und Simulation für jedes Programm nur zweimal ausgeführt, einmal zur Übersetzung des Originalprogramms und einmal zur Übersetzung des getilten Programms. Der genetische Algorithmus wurde mit 300 Individuen und 33% zu ersetzenden Individuen ausgeführt. Nach 1000 Generationen wird spätestens abgebrochen und der Algorithmus insgesamt 20-mal neu gestartet. Entsprechend den Ausführungen in Kapitel 4.8.3 ist es also wahrscheinlich, dass jeweils ein sehr gutes Optimum gefunden wird.

Die Verbesserungen gegenüber dem ebenfalls durch den encc mit statischer Analyse übersetzen Originalprogramm ist in Abbildung 5.15 dargestellt. Die maximale Verbesserung mit 50% gegenüber dem ungetilten Programm wird bei 4096 Byte Scratchpad-Speicher durch den MM32-Benchmark realisiert. Die größten Verschlechterungen (-67%) treten bei den Matrix-Multiplikationen mit 14, 15 und 16 Elementen je Dimension bei einem Scratchpad-Speicher von 128 Byte auf.

Um im Weiteren begründen zu können, warum Verschlechterungen aufgetreten sind, können die folgenden Informationen herangezogen werden:

- Tabelle 5.8 liefert eine Übersicht über die Anzahl der Befehle in der innersten Schleife. Fälle mit unerwarteten Verschlechterungen in der innersten Schleife sind fett und unerwartete Verbesserungen kursiv gedruckt.

- Tabelle 5.9 liefert eine Übersicht, welche Differenz zwischen der erwarteten und tatsächlichen Anzahl Befehle in der innersten Schleife existiert.
- Tabelle 5.10 liefert eine Übersicht, welche Arrays durch den encc nicht in den Scratchpad-Speicher verschoben wurden. Fälle mit unerwarteten Verschlechterungen in der innersten Schleife sind fett und unerwartete Verbesserungen kursiv gedruckt.
- Abbildung 5.16 stellt die Verbesserungen für die gleichen Programme dar. Im Unterschied zur Abbildung 5.15 wird dynamisches Profiling verwendet. Der Vergleich der Abbildungen liefert Aufschluss, welche Verschlechterungen durch eine ungenauere Modellierung im encc verursacht werden.

Entlang der Größe der Scratchpad-Speicher werden im Folgenden die Ergebnisse dargestellt und interpretiert.

64 Byte Scratchpad-Speicher ist sehr wenig. Der encc war für den FIR-Benchmark nicht in der Lage, ein 4 Byte großes Array in den Scratchpad-Speicher zu verschieben. Zusätzlicher Platz wurde durch einen nicht modellierten Befehl verbraucht. Da auch bei der Übersetzung mit dynamischem Profiling das Array nicht verschoben werden konnte, ist tatsächlich zu wenig Platz vorhanden.

Bei 128 Byte Scratchpad-Speicher wurde über Varianten aus Scratchpad-Speicher und Benchmark die größten Verluste erzielt. Die Matrix-Multiplikationen mit 14, 15 und 16 Elemente je Dimension haben hier die größte innerste Schleife. Zusätzlich lässt der encc bei Verwendung der statischen Analyse jeweils das für den Scratchpad-Speicher vorgesehene Array `Cs` im Hauptspeicher. Die Matrix-Multiplikation mit 32 Elementen wird auch für 128 Byte Scratchpad-Speicher noch nicht getiled. Da kleinere Tiling-Faktoren gewählt werden müssten, lohnt sich der zusätzliche Schleifen-Overhead nicht. Die Verschlechterung beim Tiefpass ist auf eine ungenaue Modellierungen des encc mit statischer Analyse zurückzuführen. Die korrespondierende Verbesserung beim dynamischen Profiling belegt dies. Der FIR-Benchmark hat im Vergleich zu den anderen Programmen eine kleinere innere Schleife und wesentlich mehr Speicherzugriffe, da auch in der äußeren Elementschleife Zugriffe stattfinden. Auch weil die Speicherbelegung fast dem Modell entsprechend umgesetzt wurde, konnten 18% Gewinn erzielt werden.

Für 256 Byte Speicher wurde für alle Matrix-Multiplikationen etwa 25% Gewinn erzielt. Die Matrix-Multiplikation mit 32 Elementen schneidet hier nicht besser als die anderen ab, weil ihre Tiling-Schleifen und Kopierfunktionen öfters durchlaufen werden und dadurch mehr Overhead entsteht. Weitere Kosten entstehen, indem ein zusätzlicher Befehl in der innersten Schleife nicht eingeplant wurde. Der Tiefpass hat bei 256 Byte sein schlechtestes Ergebnis, weil ein Array nicht in den Scratchpad-Speicher verlegt wurde. Für den FIR-Benchmark ist die Verschlechterung auf eine schlechte Auswahl des Codes für den Scratchpad-Speicher zurückzuführen.

Mit den Matrix-Multiplikationen mit 14, 15 und 16 Elementen je Dimension tritt bei 512 Byte Scratchpad-Speicher das letzte Mal ein Verlust auf. Der Grundverlust ist darauf zurückzuführen, dass jeweils ein Array nicht in den Scratchpad-Speicher verschoben wurde. Für den Benchmark Matrix-Multiplikation mit 14 Elementen ist ein zusätzlicher Befehl in der innersten Schleife nicht eingeplant und schneidet daher im Vergleich zu den anderen ebenfalls schlechter ab. Die Matrix-Multiplikation mit 32 Elementen je Dimension liefert ebenfalls ein enttäuschendes Ergebnis. Auch hier wurde ein Array nicht in den Scratchpad-Speicher verschoben. Es entsteht aber kein Verlust, da der Gewinn durch die anderen Arrays noch überwiegt. Dies wird besonders deutlich, wenn die korrespondierenden Ergebnisse mit dynamischem Profiling mitbetrachtet werden. Die Auswirkung, die eine andere Belegung des Scratchpad-Speichers hat, sind bedeutend. Für den Tiefpass und den FIR-Benchmark werden Verbesserungen erzielt, die aber durch eine bessere Speicheranordnung durch dynamisches Profiling noch verstärkt werden.

Für 1024 Byte Scratchpad-Speicher sind die Verbesserungen für die Matrix-Multiplikationen mit 14 und 15 Elementen nur sehr klein, weil zwei zusätzliche Befehle in der innersten Schleife existieren. Außerdem beginnt der encc mit der Belegung des Scratchpad-Speichers mit ungetilten Arrays im Originalprogramm. Für 16 Elemente wird auch schon bei Verwendung der statischen Analyse eine optimale Speicheranordnung gewählt, so dass statische Analyse und dynamisches Profiling ähnliche Ergebnisse liefern. Bei 32 Elementen wurde erneut ein Array nicht in den Scratchpad-Speicher verschoben. Die Verbesserung für den

| | MM14 | MM15 | MM16 | MM32 | TP | FIR |
|---|---|---|---|---|---|---|
| 0 | 22 | 22 | 20 | 22 | 20 | 17 |
| 64 | 22 | 22 | 20 | 22 | 20 | **18** |
| 128 | **28** | **28** | **25** | 22 | 21 | **18** |
| 256 | 22 | 22 | 20 | **25** | 20 | **18** |
| 512 | **23** | 22 | 21 | *21* | 20 | **18** |
| 1024 | **24** | **24** | 20 | *20* | 20 | **18** |
| 2048 | **24** | **24** | **21** | *21* | 20 | **18** |
| 4096 | 22 | 22 | 20 | *21* | 20 | 17 |
| 8192 | 22 | 22 | 20 | **23** | 20 | 17 |
| 16384 | 22 | 22 | 20 | 22 | 20 | 17 |

Tabelle 5.8: Anzahl Instruktionen der innersten Schleife für verschiedene Benchmarks und Scratchpad-Größen, nach der Optimierung

| | MM14 | MM15 | MM16 | MM32 | TP | FIR |
|---|---|---|---|---|---|---|
| 0 | | | | | | |
| 64 | | | | | | |
| 128 | 5 | 5 | 4 | | | 1 |
| 256 | | | | 2 | | 1 |
| 512 | 1 | | | −1 | | 1 |
| 1024 | 2 | 2 | | −2 | | 1 |
| 2048 | 2 | 2 | 1 | −1 | | 1 |
| 4096 | | | | −1 | | |
| 8192 | | | | 1 | | |
| 16384 | | | | | | |

Tabelle 5.9: Differenz der Instruktionen der innersten Schleife für verschiedene Benchmarks und Scratchpad-Größen, nach der Optimierung

Tiefpass fällt kleiner aus als erwartet, weil nur die innerste Schleife der Kopierfunktion in den Scratchpad-Speicher verschoben wurde. Die Speicherbelegung für den FIR-Benchmark ist fast optimal.

Für einen 2048 Byte Scratchpad-Speicher liefern die Matrix-Multiplikationen mit 14 und 15 Elementen im Vergleich zum encc nur noch eine sehr kleine Verbesserung, da der encc auch ohne Tiling schon Arrays in den Scratchpad-Speicher verschieben kann. Die Matrix-Multiplikation mit 16 Elementen liefert das beste Ergebnis, da die tatsächliche Speicherbelegung dem Modell sehr nahe kommt. Da ein zusätzliches Register verbraucht wird, ist der Gewinn nicht so groß, wie vom Modell erwartet. Bei der Matrix-Multiplikation für 32 Elemente wurde das Array `Cs` nicht verschoben. Daher ist die Verbesserung unter Verwendung der statischen Analyse kleiner als unter Verwendung des dynamischen Profilings. Die Verbesserung des FIR-Benchmarks fällt ebenfalls relativ klein aus, dies liegt hier aber daran, dass der encc den Scratchpad-Speicher auch ohne Tiling besser belegen kann.

Bei 4096 und 8192 Byte haben alle anderen Benchmarks außer der Matrix-Multiplikation mit 32 Elementen keine Verbesserung, dies liegt daran, dass der encc sämtliche Daten auch ohne Tiling in den Scratchpad-Speicher verschieben kann. Die Matrix-Multiplikation mit 32 Elementen wird nicht durch eine ungünstige Speicherbelegung benachteiligt. Daher ist der Gewinn genauso groß wie bei der Verwendung des dynamischen Profilings.

| | MM14 | MM15 | MM16 | MM32 | TP | FIR |
|---|---|---|---|---|---|---|
| 0 | | | | | | |
| 64 | | | | | | **ys[4]** |
| 128 | **Cs[1][14]** | **Cs[1][15]** | **Cs[1][16]** | | | |
| 256 | | | | | Cs[1][11] | |
| 512 | **Cs[7][7]** | Cs[5][5] | **Cs[4][8]** | *Cs[4][8]* | | |
| 1024 | | | Cs[8][8] | *Cs[8][8]* | | |
| 2048 | | | | *Cs[16][16]* | | |
| 4096 | | | | | | |
| 8192 | | | | | | |
| 16384 | | | | | | |

Tabelle 5.10: Für verschiedene Scratchpad-Größen und Benchmarks Arrays, die bei statischer Analyse nicht in den Scratchpad-Speicher verschoben wurden, obwohl sie verschoben werden sollten.

## 5.6 Ergebnisse mit dynamischem Profiling

Um die Ergebnisse für Abbildung 5.16 zu erzeugen, wurden die durch das Frontend erzeugten Programme nicht modifiziert. Die für bestimmte Scratchpad-Speicher getilten Programme und das Originalprogramm wurden für die jeweiligen Größen des Scratchpad-Speichers unter Verwendung des dynamischen Profilings durch den encc übersetzt.

Für die Betrachtung der Ergebnisse mit dynamischem Profiling kann die Beschreibung auf das Wesentliche reduziert werden. Die in Abbildung 5.16 dargestellten Ergebnisse zeigen klar die Verbesserungen, die Tiling bietet.

Für vier Ergebnisse ist eine Verschlechterung eingetreten. Ursächlich ist, dass in allen vier Fällen jeweils ein Array nicht in den Scratchpad-Speicher verschoben werden konnte. Gründe hierfür sind die zusätzlichen Instruktionen in den innersten Schleifen und einige BL-Befehle, die im Modell nicht ausreichend berücksichtigt wurden. Da der Scratchpad-Speicher straff belegt wird und die Kopierfunktion sowie die Tiling-Schleifen nicht für den Scratchpad-Speicher vorgesehen wurden, hat der encc keine Möglichkeit, für die zusätzlichen Befehle Platz zu schaffen.

Die erwarteten Verbesserungen im Modell werden aber auch unter Verwendung von dynamischem Profiling nicht immer erreicht, wenn zusätzliche nicht erwartete Befehle in der innersten Schleife eingefügt worden sind. Die Ergebnisse, für die das zutrifft, sind in Abbildung 5.16 mit einem Punkt im Balken für jeden unerwarteten Befehl markiert. Über alle Ergebnisse mit dynamischem Profiling und der Energie im Modell wurde ein Korrelationskoeffizient von 0,98 errechnet.

Alle Ergebnisse haben gemein, dass die Verbesserungen langsam zunehmen und zum Schluss einmal abnehmen und schließlich keine Verbesserung mehr erreicht wird. Für alle Phasen dieser Kurve existiert jeweils eine naheliegende Erklärung für ihre Form. Für zu kleine Scratchpad-Speicher ist zu wenig Platz vorhanden und zusätzliche Instruktionen können nicht entsprechend im Scratchpad-Speicher abgelegt werden. In der Phase der langsamen Zunahme werden immer mehr Daten in den Scratchpad-Speicher verschoben. Außerdem können immer größere Tiling-Faktoren gewählt werden, so dass weniger Overhead anfällt. Die Kurve erreicht schließlich ihr Maximum. Dann folgt eine oder mehrere kleinere Verbesserungen. In diesen Fällen kann der encc schon Daten ohne Tiling in den Scratchpad-Speicher verschieben, ohne dass Overhead anfällt. Dies kann am Beispiel der Matrix-Multiplikation mit 32 Elementen in Abbildung 5.17 verfolgt werden.

Im Anhang A.3 sind auch für das dynamische Profiling die jeweils dem Modell nach optimalen Ergebnisse und von den untersuchten Programmen die mit der jeweils besten Verbesserung aufgeführt. Dort ist auch

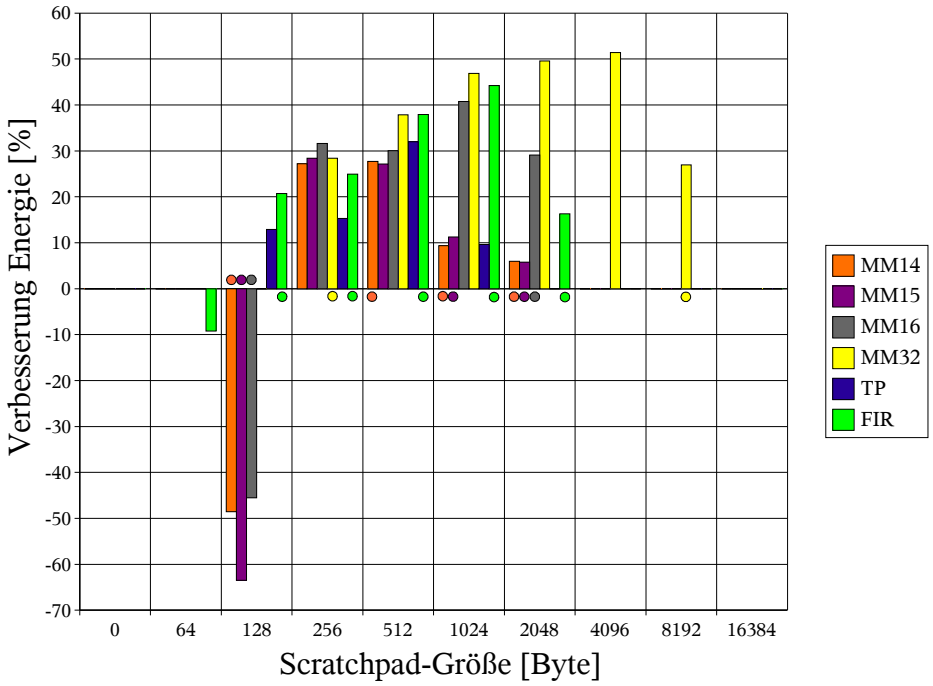


Abbildung 5.16: Verbesserung des Energieverbrauchs verschiedener Benchmarks bei Verwendung unterschiedlicher Scratchpad-Speicher. Die getilten Programme wurden mit dynamischem Profiling übersetzt. Balken mit Punkt haben mehr Befehle in der innersten Schleife als vom Modell erwartet.

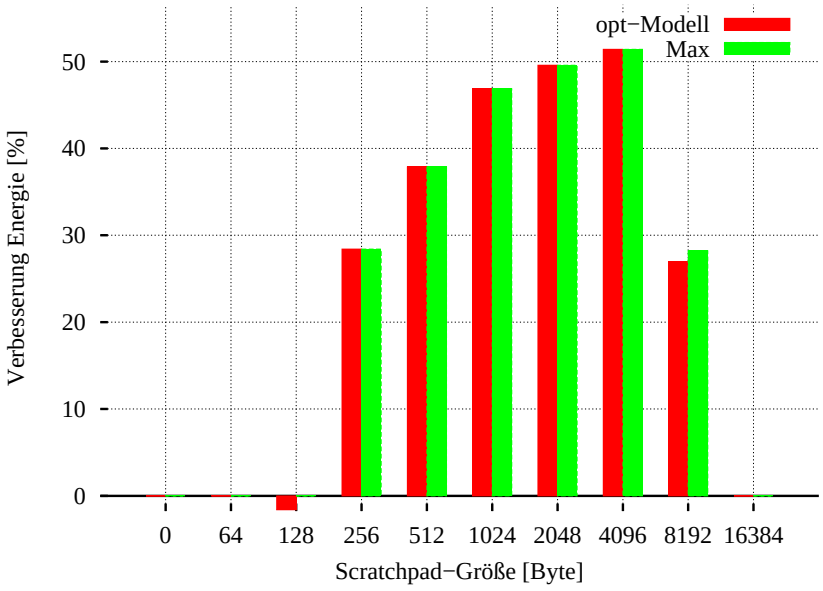


Abbildung 5.17: Verbesserung der Energie, Matrix-Multiplikation mit 32 Elementen je Dimension, dynamisches Profiling

die jeweils absolut verbrauchte Energie aufgeführt. Die Unterschiede zwischen opt-Modell und max lassen sich beim dynamischen Profiling auf Schwächen im Modell zurückführen, die in Abschnitt 5.2.3 eingeführt wurden.

Die Ergebnisse zur Matrix-Multiplikation mit 32 Elementen je Dimension zeigen einen weiteren Effekt und werden daher schon hier gezeigt. In Abbildung 5.18 ist der absolute Energieverbrauch für verschiedene Scratchpad-Speichergrößen dargestellt. Dabei ist zu beobachten, dass der Energieverbrauch über mehrere Scratchpad-Größen nur langsam sinkt, wie in den anderen Fällen auch. Bei diesem Benchmark ist das Minimum des absoluten Energieverbrauch über alle Scratchpad-Speicher mit Tiling erzielt worden, wie der absolute Energieverbrauch in Abbildung 5.18 bei 1024 bis 4096 Byte Scratchpad-Speicher zeigt. Bei anderen Benchmarks erreicht der encc das absolute Minimum über alle Scratchpad-Speicher, wenn ein großer Scratchpad-Speicher verwendet wird, in den schon alle Daten und der gesamte Code hineinpassen.

## 5.7 Erkenntnisse und Bewertung

Nach der Betrachtung der Ergebnisse lassen sich folgende wesentliche Erkenntnisse zusammenfassen:

- Der Overhead durch explizites Kopieren und die Code-Transformation ist wesentlich und erhöht den Energieverbrauch der CPU.
- Möglichkeiten der Optimierungen des Codes haben einen großen Einfluss auf das Ergebnis.
- Die Darstellbarkeit von Konstanten als Zweierpotenzen hat eine positive Auswirkung auf den generierten Code und Energieverbrauch.

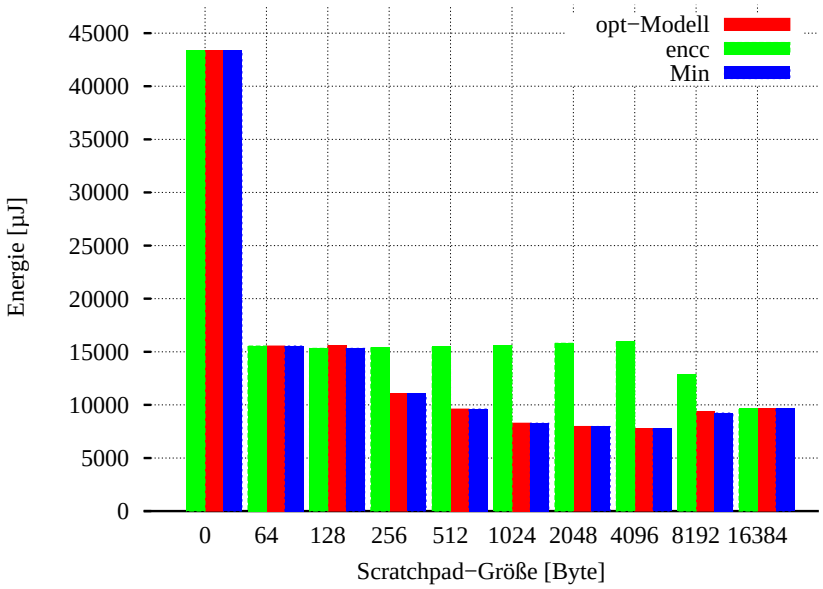


Abbildung 5.18: Absolute Energie, Matrix-Multiplikation mit 32 Elementen je Dimension, dynamisches Profiling

- Unterschiede in der Bewertung führen zu anderen Verteilungen des Codes auf den Speicher und können Verschlechterungen verursachen.
- Energie, Laufzeit und Anzahl ausgeführter Instruktionen sind unterschiedliche Größen, die sich unabhängig voneinander verändern können. Die Veränderungen können eine Verbesserung verstärken oder reduzieren.
- Der Anteil der Nutzung des Scratchpad-Speichers bestimmt, wieviel Gewinn durch Tiling erreicht werden kann.
- Es gibt ein Optimum für die Größe des Scratchpad-Speichers, das nicht am Rand möglicher Größen des Scratchpad-Speichers liegen muss und das vom Programm abhängt.

Um die Ergebnisse einzuordnen, ist ein Vergleich mit den Ergebnissen anderer Arbeiten sinnvoll. Kandemir [KRI+04] verwendet eine Architektur mit Cache und Scratchpad-Speicher, diese reduziert die Kosten für lokale Zugriffe, wie sie beim Kopieren auftreten. Die Kopierkosten, ein wesentlicher Bestandteil der Gesamtkosten, sind also nicht vergleichbar. Außerdem wird angeben, dass die Datengröße für Matrix-Multiplikation 196 KByte ist. Selbst mit 32 Elementen je Dimension und Berücksichtigung sowohl der Eingabe- als auch Ausgabe-Arrays werden in dieser Arbeit nur 12 KByte belegt, etwas 16-mal weniger. Der Vergleich der Ergebnisse für 16 und 32 in den Abbildungen 5.9 und 5.10 zeigt, dass die Ergebnisse sehr wohl von der Eingangsgröße abhängen. Ein weiterer Vergleich wird weiter dadurch erschwert, dass Kandemir die Instruktionen unberücksichtigt lässt. Es zeigt sich aber, dass gerade die Platzierung der Instruktionen einen Einfluss auf das Ergebnis hat, wie die vielen Verschlechterungen bei Verwendung statischer Analyse belegen. Kandemir erzielt Verbesserungen von bis zu 80% der Speicherzugriffe.

In [UB03] wird der Inhalt des Scratchpad-Speichers dynamisch ausgetauscht. Für den Benchmark Matrix-Multiplikation wird mit den beschriebenen Verfahren eine Verbesserung der Laufzeit um 40% gefunden.

Der Energievebrauch wird aber nicht beschrieben. Ein weiterer Unterschied ist, dass DRAM betrachtet wird. In dieser Arbeit wird als Hauptspeicher SRAM behandelt, der zwar langsamer ist als der interne Scratchpad-Speicher, aber wesentlich schneller als DRAM. Bei der Verwendung von DRAM als Hauptspeicher und SRAM als Scratchpad-Speicher kann erwartet werden, dass die erzielbaren Verbesserung weiter gesteigert werden können.

In [FV04] wird ebenfalls der FIR-Benchmark optimiert. Mit dem dort beschriebenen kombinierten Verfahren konnten insgesamt 20,3% Verbesserung für den FIR-Benchmark bei 1,8 KByte Speicher erreicht werden. Mit Tiling wird in dieser Arbeit bei 1 KByte Scratchpad-Speicher 41,4% Energie gespart.

Ein Vergleich mit anderen Arbeiten, die sich mit Tiling und Caches beschäftigen, ist unfair, weil in dieser Arbeit die anfallenden Kopierkosten mitberücksichtigt werden, die durch einen Cache aber vermindert werden. In [ZKKC03] werden verschiedene Hardware-Optimierungen und Software-Optimierungen für Caches evaluiert. Als Software-Optimierungen werden Tiling und andere Highlevel-Optimierungen eingesetzt. Desweiteren wird die Hardware optimiert, indem Blockbuffering und Cache Sub-Banking verwendet werden. Blockbuffering speichert eine Cache-Zeile so, dass nicht das gesamte Cache-Feld aktiviert werden muss, wenn immer auf die gleiche Cache-Zeile zugegriffen wird. Sub-Banking teilt das Cache Array in mehrere Teile; nur der Teil, auf den zugegriffen wurde, wird aktiviert. Mit Software-Optimierung wird für Matrix-Multiplikation eine Verbesserung um ca. 70% erreicht, Software-Optimierung und Hardware-Optimierung kombiniert liefern 80% Gewinn. Die Inputgröße ist wieder 120 KByte. 80% Gewinn insgesamt gegenüber dem Originalprogramm werden mit dem in dieser Arbeit beschriebenen Verfahren aber schon bei wesentlich kleineren Eingaben und kleineren Scratchpad-Speichern erzielt.

Das in dieser Arbeit beschriebene Verfahren kopiert die Daten während der Ausführung dynamisch. Das Verfahren von Verma [VWM04b] kopiert die Daten ebenfalls dynamisch. Ein direkter Vergleich einzelner Benchmarks ist nicht möglich, da sich die in den jeweiligen Arbeiten betrachteten Benchmarks nicht für die jeweiligen anderen Verfahren eignen. Der Vergleich einzelner Ergebnisse ist aber möglich. Durch das dynamische Overlay sind bis zu 60% erreicht. In dieser Arbeit werden bis zu 50% gegenüber dem statischen Ansatz erreicht. Beide Verfahren haben also ähnliches Potential.

# Kapitel 6

# Zusammenfassung und Ausblick

Nachdem die Ergebnisse und das Modell beschrieben sind, wird im Folgenden die Arbeit kurz zusammengefasst. Ein Ausblick liefert schließlich Ideen, wie das Verfahren verbessert werden kann und wie es in einen größeren Zusammenhang eingebunden werden kann.

## 6.1 Zusammenfassung

Tiling erhöht die Lokalität und ermöglicht die Ersetzung großer Arrays durch kleinere Arrays für die Laufzeit der Schleife. Der encc ist in der Lage, ganze Arrays und Code in den Scratchpad-Speicher zu verschieben. Das istf transformiert den Quelltext so, dass der encc ihn optimiert übersetzen kann und neue kleine Arrays in den Scratchpad-Speicher verschoben werden können. Die Auswirkungen der Code-Transformation werden im Modell des istf möglichst genau modelliert. Berücksichtigt werden die Kopierkosten, Schleifen-Overhead, Zugriffs-Overhead und Spillingkosten. Die modellierte verbrauchte Energie wird durch einen genetischen Algorithmus minimiert. Für die Modellierung der Energie wird eine Belegung des Scratchpad-Speichers angenommen. Durch die komplexen Zusammenhänge zwischen den Parametern ist es dem genetischen Algorithmus nicht immer möglich, das optimale Ergebnis zu finden. Durch wiederholte Ausführung kann jedoch die Chance, das Optimum zu finden, erhöht werden. Wird das Programm entsprechend dem Modell übersetzt, so tritt eine Verbesserung im Energieverbrauch ein. Die Anzahl ausgeführter Instruktionen ist wesentlich größer als die im Originalprogramm. Oft tritt durch die Optimierung auch eine Vergrößerung der Laufzeit auf. Statisches Profiling liefert bei einfachen Programmen in einigen Fällen gute Ergebnisse. Im Gegensatz zum dynamischen Profiling hat die statische Analyse nicht immer die optimale Belegung des Scratchpad-Speichers gefunden. Einen wesentlichen Einfluss auf den Overhead haben nicht nur die Kopierkosten, Zugriffskosten und Spillingkosten, die im Modell berücksichtigt werden, sondern auch Veränderungen in der Optimierbarkeit des übersetzen Programms. Eine Abschätzung der genauen Code-Größe des generierten Programms ist daher nicht möglich. Besonders bei kleinen Scratchpad-Speichern werden zusätzliche Sprünge eingefügt, die im Modell nicht ausreichend berücksichtigt werden. Auch die unterschiedlichen Möglichkeiten zur Erzeugung von Konstanten werden nicht berücksichtigt.

Trotz der Unwägbarkeiten, die Tiling auf den Code noch haben kann, werden Verbesserungen um bis zu 50% gegenüber den bisherigen Verfahren erreicht. Im Vergleich zu einem Programm ohne Nutzung des Scratchpad-Speichern wird der Energieverbrauch um bis zu 80% reduziert. Im Vergleich zu bisherigen Ansätzen zur Optimierung des Energieverbrauch ist es möglich, durch Tiling schon bei kleineren Scratchpad-Speichern mehr Energie zu sparen. Ist der Scratchpad-Speicher groß genug, dass Arrays komplett in den Scratchpad-Speicher passen, ist die Anwendung von Tiling nicht mehr notwendig, um Energie zu sparen.

Bei allen Ergebnissen mit Verbesserungen zeigt sich, dass es immer sinnvoll ist, sowohl Code als auch Daten in den Scratchpad-Speicher zu verschieben. Eine Betrachtung von Code oder Daten alleine ist nicht

sinnvoll, wenn der Energieverbrauch des Gesamtsystems betrachtet wird. Einen festen Anteil Scratchpad-Speicher für Code oder Daten zu reservieren, ist ebenfalls nicht von Vorteil.

## 6.2 Ausblick

Nachdem die wesentlichen Ziele dieser Arbeit erreicht sind, bleibt im Ausblick Gelegenheit, die getroffenen Entscheidungen mit dem Wissen der Ergebnisse neu zu bewerten. Es bietet sich auch die Möglichkeit für weitere Hinweise zur Weiterentwicklung.

Der Code, der durch das Frontend erzeugt wird, ist nicht optimal. Die Ergebnisse können verbessert werden. In anderen Arbeiten [KRI+04] wird Schleifen vertauschen in Betracht gezogen, um zunächst eine optimale Schleifenanordnung zu erhalten. In dieser Arbeit wird vorausgesetzt, dass die Schleifen schon im Original in einer optimalen Ordnung sind und dass die Tiling-Schleifen optimal genau so angeordnet werden wie im Original. Dies muss nicht in jedem Fall so sein. Bei Matrix-Multiplikation von 15 Elementen ist für nicht optimale Eingaben eine Verschlechterung um bis zu 30% gegenüber dem Optimum möglich.

Eine frühe Design-Entscheidung ist, ein Frontend für einen existierenden Compiler zu schreiben. Dieser Ansatz wurde gewählt, weil im Frontend die Schleifeninformationen vorhanden sind und die Code-Transformation sehr einfach ist. Die Informationen über den zu erwartenden Code fließen in Form einer Assembler-Datei und einer Datei mit Informationen über den Registerdruck in das Frontend, und eine C-Datei wird vom encc übersetzt. Der umgekehrte Informationsfluss existiert nicht.

Dynamisches Profiling liefert zwar die gewünschten Ergebnisse, erfordert aber eine Ausführung des Programms. Eine Verbesserung des Ansatzes mit statischer Analyse kann darin bestehen, dem encc zusätzliche Informationen aus der Analyse des Frontends zu geben.

Ein anderer Punkt, an dem in dieser Arbeit Energie verschenkt wird, ist die Kopierfunktion selbst. Auch der THUMB-Mode bietet Instruktionen, mit denen mehrere Daten auf einmal kopiert werden können. Hat ein Array im Scratchpad-Speicher in einer Dimension die Größe 1, dann kann eine Dimension weniger verwendet werden. Beide Optimierungen werden nicht durchgeführt, um das Modell zu vereinfachen und die Code-Größe klein zu halten.

Diese Funktion kann eine Compiler Known-Function sein und die Zugriffe und Kosten können im Modell des encc berücksichtigt werden. Eine andere Möglichkeit ist, die Schnittstelle für Profiling-Informationen des encc zu verwenden. Dazu wird das generierte Programm erneut übersetzt, um wieder eine Zuordnung zwischen den Basisblöcken und dem C-Programm zu finden. Die Informationen können auch in Form von Pragmas oder einer getrennten Datei übergeben werden.

Vielversprechend ist auch die Verwendung des Scratchpad-Overlay-Verfahrens von Verma. Das Verfahren wurde nicht verwendet, weil es im Frontend-Ansatz keine wesentlichen Vorteile bietet und der Highlevel-Ansatz zu viel Code erzeugt. Eine enge Integration zwischen Frontend und encc ist möglich, wenn der encc wesentlich modifiziert wird. Die Kopierfunktionen würden dann durch den Compiler eingefügt. Der Compiler muss dann aber auch in der Lage sein, die kleinen Arrays entsprechend zu erkennen und zu füllen. Durch die dynamische Belegung des Scratchpad-Speichers ist es dann auch sinnvoll, sehr komplexe Programme zu betrachten, die verschiedene häufig ausgeführte Schleifennester besitzen. Aber auch dieses Verfahren wäre dann auf ein dynamisches Profiling angewiesen.

Mit dem gewählten Ansatz ist die Nutzung anderer Compiler denkbar, da eine C nach C Transformation durchgeführt wird.

Um mit dem in dieser Arbeit dargestellten Verfahren größere Benchmarks sinnvoll zu übersetzen, ist eine dynamische Belegung des Scratchpad-Speichers notwendig. Nur durch eine dynamische Belegung können die Arrays verschiedener Schleifen den Scratchpad-Speicher gemeinsam nutzen. Bei Verwendung des Verfahrens von Verma können die eingefügten kleinen Arrays den Scratchpad-Speicher gemeinsam nutzen.

Weiter verbessern lässt sich das Modell, wenn der Code für Zugriffe und die Code-Erzeugung für Konstanten genauer vorhergesagt werden kann. Eine sehr einfache aber ungenaue Möglichkeit ist, dass für

bestimmte Merkmale Punkte vergeben werden. Die Punkte wären dann proportional zur Anzahl benötigter Befehle und könnten in das Modell einfließen. Punkte könnte es dafür geben, dass die Dimensionen mehrerer Arrays nicht gleich sind, dass keine Zweierpotenzen verwendet werden, dass die Zugriffsfunktionen verschieden sind u. s. w..

Besser wäre es, wenn ein Programmstück partiell übersetzt werden könnte. So wäre es möglich, zur Berechnung der Kosten einfach die Array-Zugriffe innerhalb des Nestes zu übersetzen und aus diesem Code-Teil abzulesen, welche Kosten für eine Zugriff entstehen. Dieses Konzept könnte weiter getrieben werden, indem an die Knoten des Syntaxbaums der ICD-C IR Informationen annotiert werden, welche Code-Fragmente für diesen Teil des C-Programms generiert worden sind. Es wäre dann nicht notwendig, die verschiedenen Kostenklassen zur Modellbildung zu verwenden. Mit einem Pass über den Teilbaum der generierten Schleife wird in Attributen annotiert, welche Kosten entstehen.

Um wirklich automatisch beliebige Programme zu übersetzen, fehlt in dieser Arbeit eine Abhängigkeitsanalyse. Auf diese wurde bewusst verzichtet. Für nicht perfekte Nester reicht auch die verwendete Notation nicht mehr aus. Aktuelle Arbeiten [LLL01] zeigen auf, wie beliebige Schleifennester mit affinen Transformationen modelliert und getiled werden können. Für einen umfassenden Energie optimierenden Compiler ist es sicher notwendig, auch diese Optimierungen zu implementieren. Das Modell muss erheblich modifiziert werden, wenn nicht die im vorherigen Abschnitt beschriebene Attributierung zum Einsatz kommt.

Eine andere interessante Fragestellung ist, wie das Modell vereinfacht werden kann, so dass es linear ist und trotzdem noch die Realität ausreichend modelliert, um Verbesserungen zu erreichen. Das in dieser Arbeit modellierte Problem gehört zu der Klasse der Packprobleme. Für viele Packprobleme existieren lineare oder polynomielle Approximationen, so dass Genauigkeit gegen Geschwindigkeit getauscht werden kann. Für das in dieser Arbeit dargestellte Problem gibt es keine offensichtliche Approximation.

Neuere Prozessoren für eingebettete Systeme besitzen neben einem integrierten Scratchpad-Speicher auch einen Cache. Eine genauere Betrachtung dieser Architektur wäre interessant. Einerseits sorgt der Cache dafür, dass die anfallenden Kopierkosten geringer sind. Die Kopierzugriffe besitzen örtliche und zeitliche Lokalität, so dass sie von einem Cache profitieren. Anderseits würde ein getiltes Programm auch ohne explizites Kopieren der Arrays schon profitieren. Es müsste das Modell der Kosten so erweitert werden, dass die Cache-Kosten ebenfalls berücksichtigt werden. Erst dann kann eine Abwägung getroffen werden, wie das Programm zu tilen ist und welche Arrays explizit kopiert werden sollten – wenn überhaupt.

Die Kopierkosten können nicht nur durch einen Cache minimiert werden, sondern auch durch eine DMA-Einheit. Die Anpassung des Modells, um diese Kosten zu berücksichtigen, wären sehr klein. Der encc mit statischer Analyse wird davon sicher auch profitieren, da dann die Kopierfunktion wegfällt, die nur sehr schlecht analysiert werden kann. Durch Verwendung einer DMA-Einheit kann der Code, der für die Kopierfunktionen notwendig ist, eingespart werden.

In [WM04] wird dargestellt, dass durch den Einsatz von Scratchpad-Speichern die Schätzung der Worst-Case Execution Time verbessert werden kann. Dies liegt im Wesentlichen daran, dass der Scratchpad-Speicher explizit belegt wird und daher einfacher zu analysieren ist, welche Zugriffszeiten für einzelne Zugriffe auftreten. Compiler, die die Worst-Case Execution Time verbessern wollen, werden also auch in Betracht ziehen, einen Scratchpad-Speicher zu nutzen. Da diese Arbeit einen Scratchpad-Speicher nutzt, ist es möglich, dass das in dieser Arbeit beschriebene Verfahren auch in solchen Compilern Anwendung findet. Auf eine genaue Modellierung der Energie kann dann verzichtet werden.

# Anhang A

# Anhang

## A.1 Cache-Belegung

Das folgende Programm wurde benutzt, um die Informationen zu den Cache Hit- und Miss-Raten zu gewinnen, die in Tabelle 4.1 dargestellt sind.

```
int hit;
int miss;
int looptest;
int cache[50];
void access(int pos){
  int i;
  int cachepos;
  cachepos=pos%50;
  if(cache[cachepos]==pos)
    hit++;
  else{
    miss++;
    cachepos-=cachepos % 5;
    pos-= cachepos % 5;
    for(i=0;i<5;i++)
      cache[cachepos+i]=pos+i;
   }
}
void init(){
 int i;
 hit=0;
  miss=0;
 for(i=0;i<50;i++)
  cache[i]=-1;
  looptest=0;
}
void result(){
  printf("Misses:%d\n",miss);
  printf("Hits:%d\n",hit);
  printf("Looptests:%d\n",looptest);
```

```
}
void main(){
 int i, z;
 init();
 printf("Progarmm A:\n");
 for(i=0;i<10;i++){
   looptest++;
   for(z=0;z<100;z++){
     looptest++;
     access(i);
     access(10+z);
     access(i);
   }
 }
 result();
 init();
 printf("Progarmm B:\n");
 for(z=0;z<100;z++){
   looptest++;
   for(i=0;i<10;i++){
     looptest++;
     access(i);
     access(10+z);
     access(i);
   }
 }
 result();
 init();
 printf("Progarmm A':\n");
 for(i=0;i<10;i++){
   looptest++;
   for(z=0;z<100;z++){
     looptest++;
     access(100+i);
     access(z);
     access(100+i);
   }
 }
 result();
 init();
 printf("Progarmm B':\n");
 for(z=0;z<100;z++){
   looptest++;
   for(i=0;i<10;i++){
     looptest++;
     access(100+i);
     access(z);
     access(100+i);
   }
 }
 result();

}
```

Das folgende Programm wurde benutzt um die Informationen zu den Cache Hit- und Miss-Raten zu gewinnen, wenn das Programm getiled ist, die in Tabelle 4.2 dargestellt sind.

```
int hit;
int miss;
int looptest;
int cache[50];
void access(int pos){
  int i;
  int cachepos;
  cachepos=pos%50;
  if(cache[cachepos]==pos)
    hit++;
  else{
    miss++;
    cachepos-=cachepos % 5;
    pos-= cachepos % 5;
    for(i=0;i<5;i++)
     cache[cachepos+i]=pos+i;
   }
}
void init(){
  int i;
  hit=0;
  miss=0;
  for(i=0;i<50;i++)
    cache[i]=-1;
  looptest=0;
}
void result(){
  printf("Misses:%d\n",miss);
  printf("Hits:%d\n",hit);
  printf("Looptests:%d\n",looptest);
}
int min(int a,int b){
  return a<b?a:b;
}

void main(){

 int A[10];
 int B[100];
 int i, z;
 int it, zt;
 for(i=0;i<10;i++)
   A[i]=i;
 for(i=0;i<100;i++)
   B[i]=i;
 init();
 for(i=0;i<10;i++){
   looptest++;
   for(z=0;z<100;z++){
     looptest++;
     access(i);
```

```
    access(10+z);
    access(i);
    A[i]=A[i]+B[z];
  }
}
result();
printf("Original:");
for(i=0;i<10;i++)
  printf("%d ",A[i]);
int zT;
int iT;
for(zT=5;zT<=100;zT+=5){
  for(iT=5;iT<=10;iT+=5){
    init();
    for(i=0;i<10;i++)
      A[i]=i;
    for(i=0;i<100;i++)
      B[i]=i;
    printf("\nTiled:   ");

    for(zt=0;zt<100;zt+=zT){
      looptest++;
      for(it=0;it<10;it+=iT){
        looptest++;
        for(z=zt;z<min(100,zt+zT);z++){
          looptest++;
          for(i=it;i<min(10,it+iT);i++){
            access(i);
            access(10+z);
            access(i);
            looptest++;
            A[i]=A[i]+B[z];
          }
        }
      }
    }
    for(i=0;i<10;i++)
      printf("%d ",A[i]);
    printf("\n");
    result();

    printf("Variante B:;zT:;%d;iT:;%d;Miss:;%d;Hit:;%d;
            Looptest:;%d\n",zT,iT,miss,hit,looptest);
  }
}
for(zT=5;zT<=100;zT+=5){
  for(iT=5;iT<=10;iT+=5){
    init();
    for(i=0;i<10;i++)
      A[i]=i;
    for(i=0;i<100;i++)
      B[i]=i;
    printf("\nTiled:   ");
```

```
    for(zt=0;zt<100;zt+=zT){
      looptest++;
      for(it=0;it<10;it+=iT){
        looptest++;
        for(i=it;i<min(10,it+iT);i++){
          looptest++;
            for(z=zt;z<min(100,zt+zT);z++){
            access(i);
            access(10+z);
            access(i);
            looptest++;
            A[i]=A[i]+B[z];
          }
        }
      }
    }
    for(i=0;i<10;i++)
      printf("%d ",A[i]);
    printf("\n");
    result();

    printf("Variante B'';zT:;%d;iT:;%d;Miss:;%d;Hit:;%d;
            Looptest:;%d\n",zT,iT,miss,hit,looptest);
  }
}

for(zT=5;zT<=100;zT+=5){
  for(iT=5;iT<=10;iT+=5){
    init();
    for(i=0;i<10;i++)
      A[i]=i;
    for(i=0;i<100;i++)
      B[i]=i;
    printf("\nTiled:   ");

    for(it=0;it<10;it+=iT){
      looptest++;
      for(zt=0;zt<100;zt+=zT){
        looptest++;
        for(i=it;i<min(10,it+iT);i++){
          looptest++;
            for(z=zt;z<min(100,zt+zT);z++){
            access(i);
            access(10+z);
            access(i);
            looptest++;
            A[i]=A[i]+B[z];
          }
        }
      }
    }
    for(i=0;i<10;i++)
      printf("%d ",A[i]);
    printf("\n");
```

```
    result();

    printf("Variante A:;zT:;%d;iT:;%d;Miss:;%d;Hit:;%d;
            Looptest:;%d\n",zT,iT,miss,hit,looptest);
  }
 }

 for(zT=5;zT<=100;zT+=5){
   for(iT=5;iT<=10;iT+=5){
    init();
    for(i=0;i<10;i++)
      A[i]=i;
    for(i=0;i<100;i++)
      B[i]=i;
    printf("\nTiled:   ");

    for(it=0;it<10;it+=iT){
      looptest++;
      for(zt=0;zt<100;zt+=zT){
        looptest++;
        for(z=zt;z<min(100,zt+zT);z++){
          looptest++;
          for(i=it;i<min(10,it+iT);i++){
            access(i);
            access(10+z);
            access(i);
            looptest++;
            A[i]=A[i]+B[z];
          }
        }
      }
    }
    for(i=0;i<10;i++)
      printf("%d ",A[i]);
    printf("\n");
    result();

    printf("Variante A'':; zT:;%d;iT:;%d;Miss:;%d;Hit:;%d;
            Looptest:;%d\n",zT,iT,miss,hit,looptest);
  }
 }
}
```

## A.2 Der Highlevel-Ansatz

Das Programm Matrix-Multiplikation wurde entsprechend dem Highlevel-Ansatz im Abschnitt 4.4 transformiert. Für die Teilung der Schleifen in zwei Hälften sind schon acht Schleifenkopien erforderlich. In jedem Schleifennest wird jeweils auf andere Teile der getilten Arrays zugegriffen. Die Kommentare geben jeweils an, welche Kombination gerade bearbeitet wird.

```
struct discontinuous{
```

```
    int  a0[25];
    int  a1[25];
};
union myArray {
     int  cont[50];
     struct discontinuous split;
};
struct discontinuous2D{
    union myArray  a0[25];
    union myArray  a1[25];
};
union myArray2D {
     union myArray  cont[50];
     struct discontinuous2D split;
};
union myArray2D a;
union myArray2D b;
union myArray2D c;
#include "matrix.h" //Hier sind die Eingabe-Arrays definiert
int main(int argc,char **argv){
   int i;
   int j;
   int k;
   int r;
   for(i=0;i<25;i++)
     for(j=0;j<25;j++){
        a.split.a0[i].split.a0[j]=A[i][j];
        b.split.a0[i].split.a0[j]=B[i][j];

        a.split.a0[i].split.a1[j]=A[i][25+j];
        b.split.a0[i].split.a1[j]=B[i][25+j];

        a.split.a1[i].split.a0[j]=A[25+i][j];
        b.split.a1[i].split.a0[j]=B[25+i][j];

        a.split.a1[i].split.a1[j]=A[25+i][25+j];
        b.split.a1[i].split.a1[j]=B[25+i][25+j];
   }

   // j 0 k 0 i 0
   for (j = 0; j < 25; j++) {
     for (k = 0; k < 25; k++) {
       r=c.split.a0[j].split.a0[k];
       for (i= 0; i < 25; i++) {
         r = r + a.split.a0[j].split.a0[i] * b.split.a0[i].split.a0[k];
       }
       c.split.a0[j].split.a0[k]=r;
     }
   }
   // j 0 k 0 i 1
   for (j = 0; j < 25; j++) {
     for (k = 0; k < 25; k++) {
       r=c.split.a0[j].split.a0[k];
       for (i= 0; i < 25; i++) {
```

```
        r = r + a.split.a0[j].split.a1[i] * b.split.a1[i].split.a0[k];
      }
      c.split.a0[j].split.a0[k]=r;
    }
  }

  // j 0 k 1 i 0
  for (j = 0; j < 25; j++) {
    for (k = 0; k < 25; k++) {
      r=c.split.a0[j].split.a1[k];
      for (i= 0; i < 25; i++) {
        r = r + a.split.a0[j].split.a0[i] * b.split.a0[i].split.a1[k];
      }
      c.split.a0[j].split.a1[k]=r;
    }
  }
  // j 0 k 1 i 1
  for (j = 0; j < 25; j++) {
    for (k = 0; k < 25; k++) {
      r=c.split.a0[j].split.a1[k];
      for (i= 0; i < 25; i++) {
        r = r + a.split.a0[j].split.a1[i] * b.split.a1[i].split.a1[k];
      }
      c.split.a0[j].split.a1[k]=r;
    }
  }
  // j 1 k 0 i 0
  for (j = 0; j < 25; j++) {
    for (k = 0; k < 25; k++) {
      r=c.split.a1[j].split.a0[k];
      for (i= 0; i < 25; i++) {
        r = r + a.split.a1[j].split.a0[i] * b.split.a0[i].split.a0[k];
      }
      c.split.a1[j].split.a0[k]=r;
    }
  }
  // j 1 k 0 i 1
  for (j = 0; j < 25; j++) {
    for (k = 0; k < 25; k++) {
      r=c.split.a1[j].split.a0[k];
      for (i= 0; i < 25; i++) {
        r = r + a.split.a1[j].split.a1[i] * b.split.a1[i].split.a0[k];
      }
      c.split.a1[j].split.a0[k]=r;
    }
  }
  // j 1 k 1 i 0
  for (j = 0; j < 25; j++) {
    for (k = 0; k < 25; k++) {
      r=c.split.a1[j].split.a1[k];
      for (i= 0; i < 25; i++) {
        r = r + a.split.a1[j].split.a0[i] * b.split.a0[i].split.a1[k];
      }
      c.split.a1[j].split.a1[k]=r;
```

```
    }
  }
  // j 1 k 1 i 1
  for (j = 0; j < 25; j++) {
    for (k = 0; k < 25; k++) {
      r=c.split.a1[j].split.a1[k];
      for (i= 0; i < 25; i++) {
        r = r + a.split.a1[j].split.a1[i] * b.split.a1[i].split.a1[k];
      }
      c.split.a1[j].split.a1[k]=r;
    }
  }



  for(i=0;i<50;i++){
    for(j=0;j<50;j++){
      printf("%d ",c.cont[i].cont[j]);
    }
    printf("\n");
  }

}
```

## A.3 Weitere Ergebnisse mit dynamischem Profiling

Da es nicht Ziel der Arbeit war, dynamisches Profiling zu verwenden, die Ergebnisse aber trotzdem vorliegen, werden sie der Vollständigkeit halber im Anhang aufgeführt. Die Abbildungen A.1, A.2, A.3, A.4 und A.3 zeigen die maximale Verbesserung des Energieverbrauchs, die durch verschiedene Läufe des genetischen Algorithmus gefunden wurden. Der Balken opt-Model zeigt die Verbesserung der Energie an, die erreicht wird, wenn das unter 11 Läufen dem Modell nach optimale Programm evaluiert wird. Sowohl das Originalprogramm als auch das getilte Programm wurden mit dynamischem Profiling durch den encc bei gleicher Scratchpad-Größe übersetzt und die relative Verbesserung aufgetragen.

Die Abbildungen A.6, A.7, A.8, A.9 und A.8 zeigen den absoluten Energieverbrauch für verschiedene Programmversionen. encc ist das Originalprogramm, das mit dynamischen Profiling übersetzt wurde. opt-Model ist das dem Modell nach optimale Programm. Der min-Balken zeigt den absolut gefunden minimalen Energieverbrauch unter den getilten Programm.

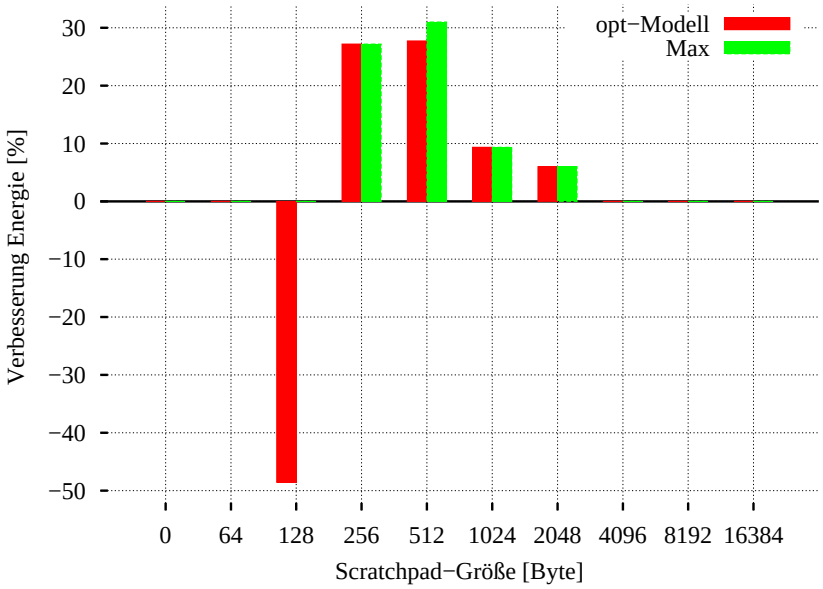


Abbildung A.1: Verbesserung der Energie, Matrix-Multiplikation mit 14 Elementen je Dimension, dynamisches Profiling

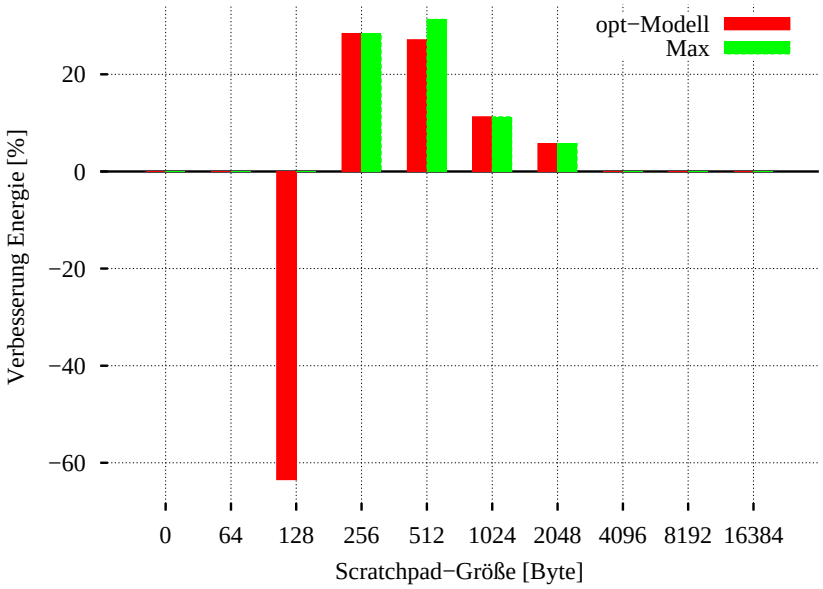


Abbildung A.2: Verbesserung der Energie, Matrix-Multiplikation mit 15 Elementen je Dimension, dynamisches Profiling

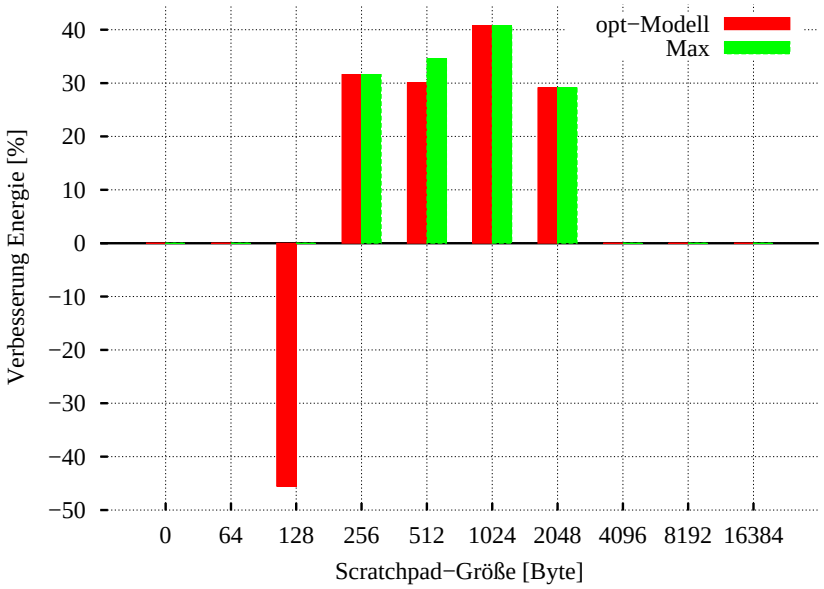


Abbildung A.3: Verbesserung der Energie, Matrix-Multiplikation mit 16 Elementen je Dimension, dynamisches Profiling

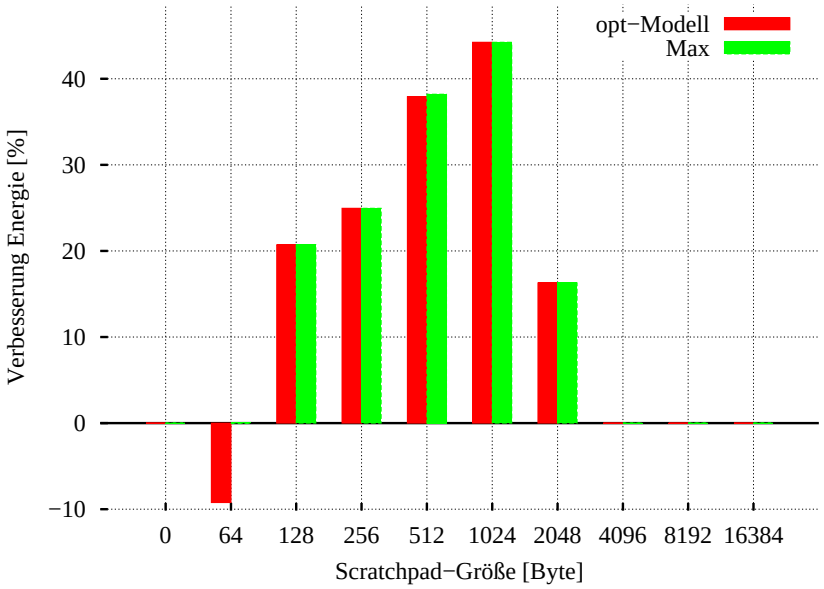


Abbildung A.4: Verbesserung der Energie, eindimensionale Faltung mit 401 und 40 Elementen, dynamisches Profiling

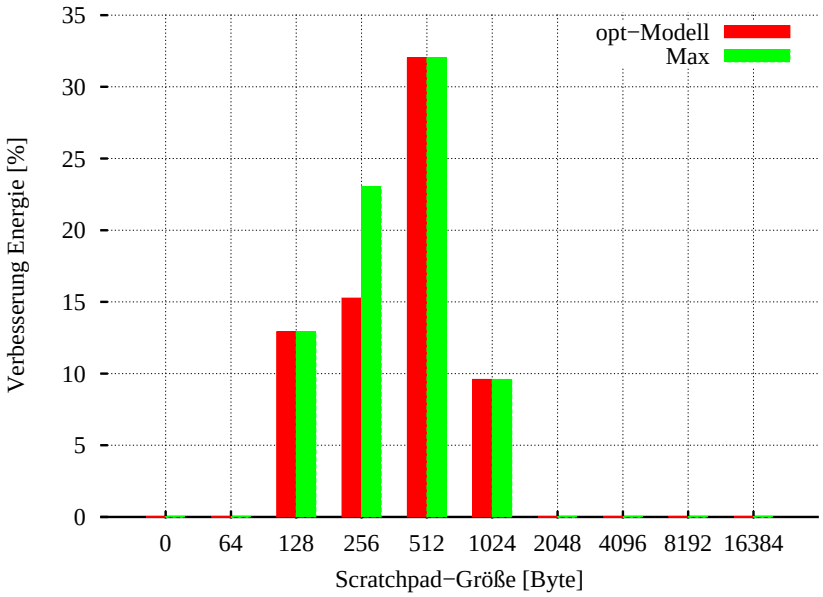


Abbildung A.5: Verbesserung der Energie, Tiefpass über ein zweidimensionales Feld mit 15 Elementen je Dimension und einer Fenstergröße von 3, dynamisches Profiling

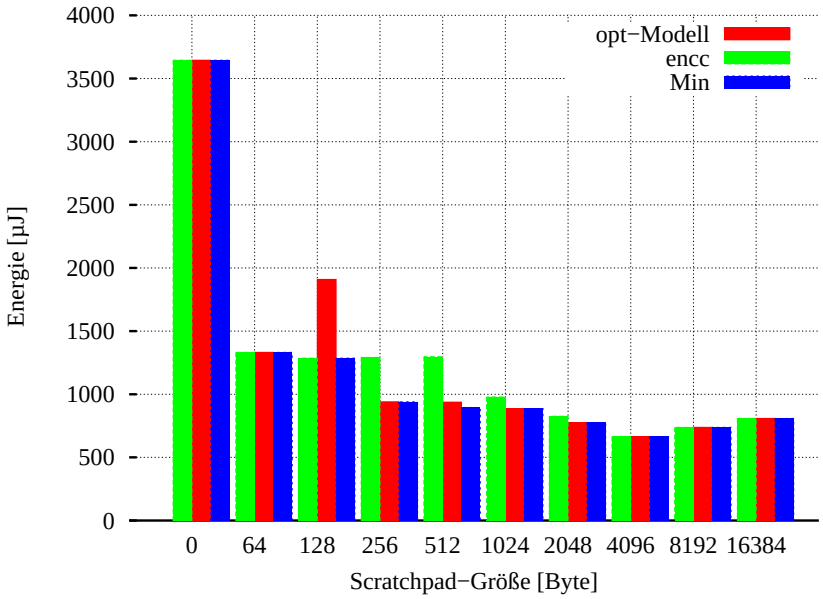


Abbildung A.6: Absolute Energie, Matrix-Multiplikation mit 14 Elementen je Dimension, dynamisches Profiling

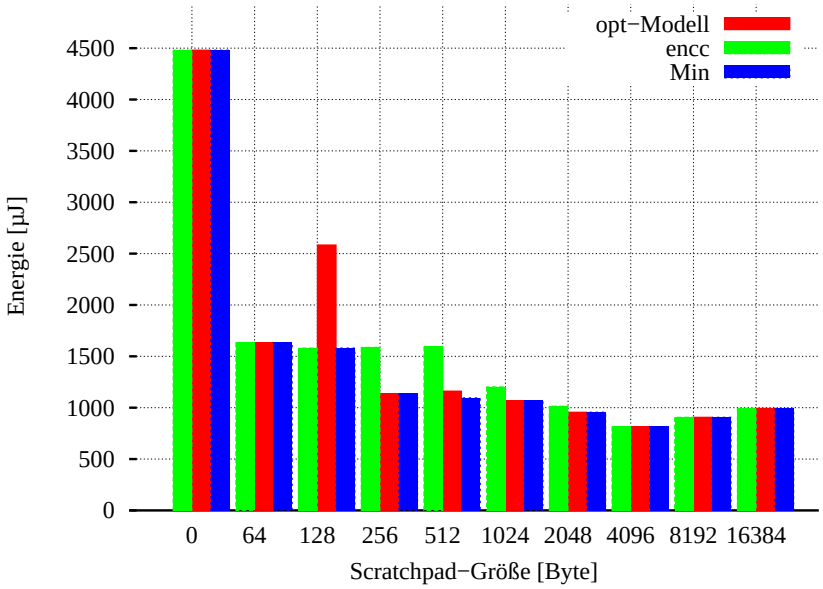


Abbildung A.7: Absolute Energie, Matrix-Multiplikation mit 15 Elementen je Dimension, dynamisches Profiling

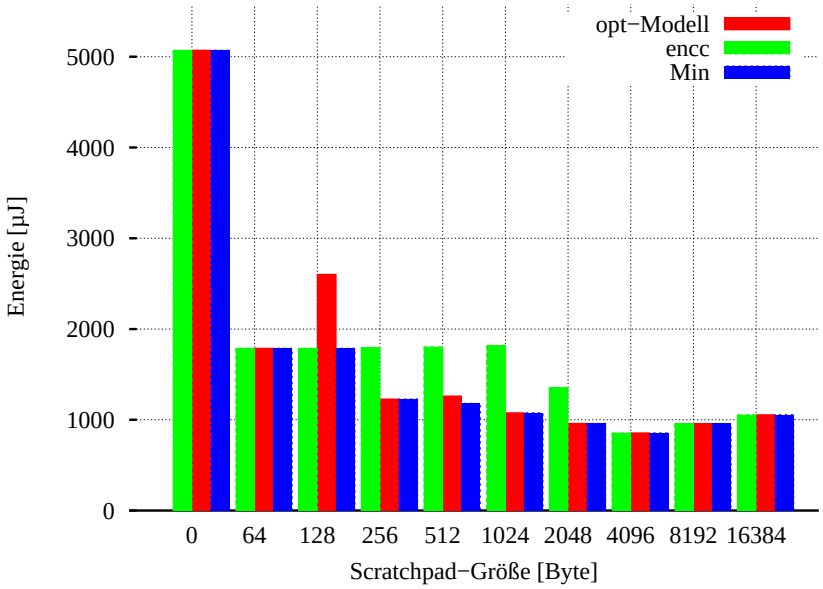


Abbildung A.8: Absolute Energie, Matrix-Multiplikation mit 16 Elementen je Dimension, dynamisches Profiling

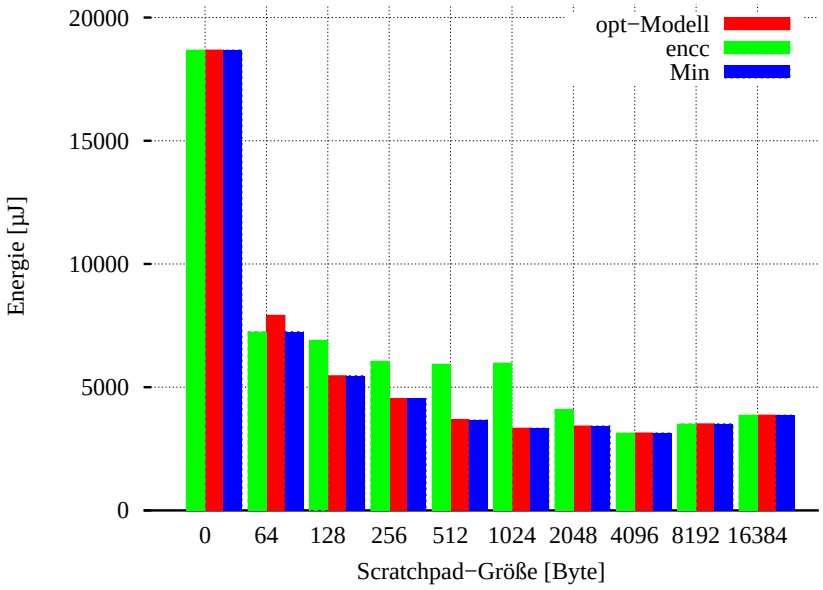


Abbildung A.9: Absolute Energie, eindimensionale Faltung mit 401 und 40 Elementen, dynamisches Profiling

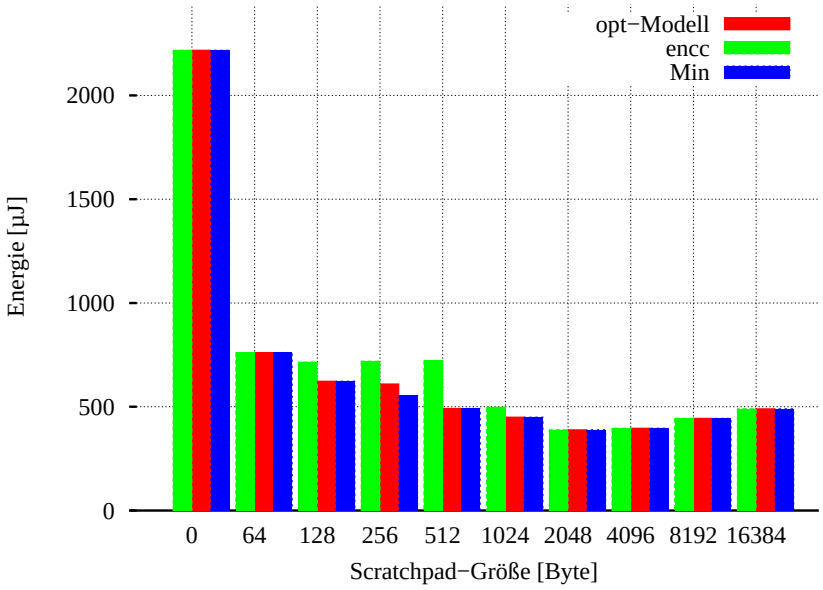


Abbildung A.10: Absolute Energie, Tiefpass über ein zweidimensionales Feld mit 15 Elementen je Dimension und einer Fenstergröße von 3, dynamisches Profiling

## A.4 Berücksichtigung bedingter Sprünge

Basisblöcke können zwei Nachfolger haben: Einer direkt im Code, der andere durch einen bedingten Sprung am Ende des Basisblocks. Die Versuche in Kapitel 5 wurden ohne Berücksichtigung des bedingten Sprungs durchgeführt. Zum Vergleich, welche Auswirkungen sich durch Betrachtung der Summe A.3 ergibt, kann Abbildung A.11 herangezogen werden.

$$E_{BB} = \sum_{b=0}^{bbc} oex_{bl_b} \cdot bc_{B_b,b} \tag{A.1}$$

$$+ \sum_{b=0}^{bbc-1} oex_{bl_b} \cdot e_{B_b,JUMP} \cdot (B_{b+1} \neq B_b) \tag{A.2}$$

$$+ \sum_{b=0}^{bbc} oex_{bl_b} \cdot e_{B_b,JUMP} \cdot (B_{bj_b} \neq B_b) \tag{A.3}$$

In Abbildung A.11 werden die gleichen Benchmarks wie in Abbildung 5.16 auf Seite 120 betrachtet. Der Unterschied ist, dass hier ein zusätzlicher Sprung angenommen wird, wenn zwei Basisblöcke durch einen bedingten Sprung verbunden sind und sie in unterschiedlichen Speichern liegen. Die Benchmarks wurden durch den encc mit dynamischem Profiling übersetzt.

Für kleinere Scratchpad-Speicher bis 512 Byte zeigen sich Unterschiede. Der FIR-Benchmark wird für 64 Byte Scratchpad-Speicher nun nicht mehr getiled. Die Verschlechterungen bei 128 Byte Scratchpad-Speicher sind für die Matrix-Multiplikationen mit 14, 15 und 16 Elementen kleiner. Der FIR-Benchmark kann bei 128 Byte Scratchpad-Speicher ein besseres Ergebnis erzielen. Wesentlich schlechter ist aber das Ergebnis der Matrix-Multiplikation mit 32 Elementen bei 256 Scratchpad-Speicher. Kleine Verbesserungen zeigen sich auch bei der Matrix-Multiplikation mit 14 Elementen.

Es zeigt sich also kein klares Bild, dass die Berücksichtigung der Teilformel A.3 immer zu einem besseren Ergebnis führt. Erst eine genauere Modellierung aller Basisblockübergänge, die auch die neu generierten Basisblöcke einschließt, kann eine wesentliche Verbesserung der Genauigkeit bewirken. Aufgrund der wesentlich höheren Modellierungskomplexität wurde auf diesen Schritt verzichtet.

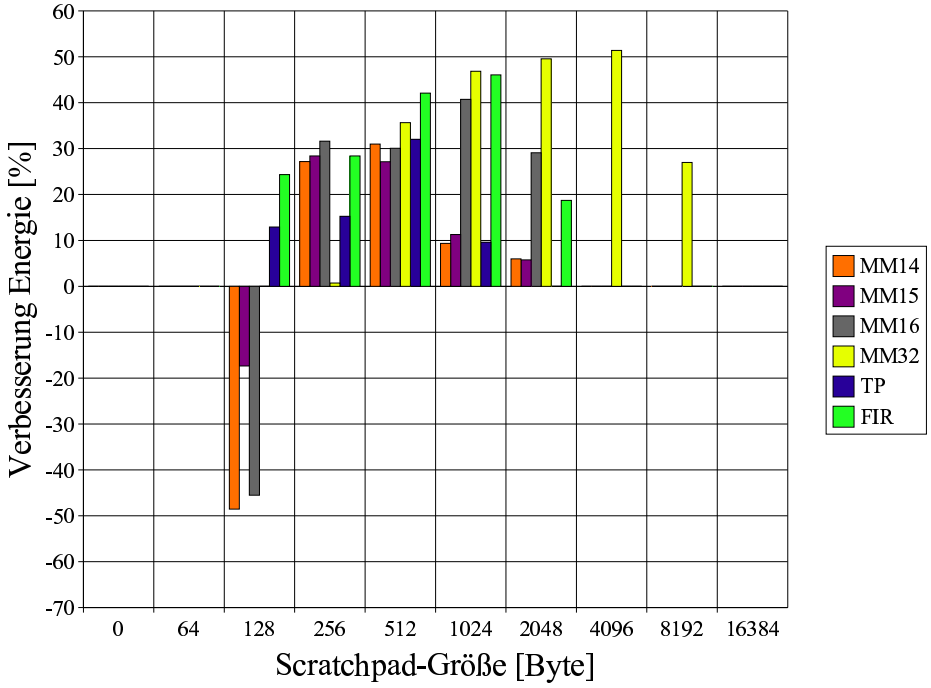


Abbildung A.11: Verbesserung des Energieverbrauchs verschiedener Benchmarks bei Verwendung unterschiedlicher Scratchpad-Speicher. Die getilten Programme wurden mit dynamischem Profiling übersetzt. Bedingte Aussprünge werden mit Kosten belegt, wenn der Speicher gewechselt wird.

# Literaturverzeichnis


[ADCL02] ACHTEREN, Tanja van ; DECONINCK, Geert ; CATTHOORAND, Francky ; LAUWEREINS, Rudy: Data Reuse Exploration Techniques for Loop–dominated Applications. In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (2002), März, S. 428–436

[Adv98a] ADVANCED RISC MACHINES LTD.: *SDT2.50 Reference Manual*. http://www.arm.com/pdfs/sdt250refman.pdf. Version: 1998. – Ref: DUI0041D

[Adv98b] ADVANCED RISC MACHINES LTD.: *SDT2.50 User Guide*. http://www.arm.com/pdfs/sdt250usrman.pdf. Version: 1998. – Ref: DUI0040D

[Adv01a] ADVANCED RISC MACHINES LTD.: *ARM Developer Suite Version 1.2 Assembler Guide*. http://www.arm.com/pdfs/DUI0068B_ADS1_2_Assembler.pdf. Version: 2001. – Ref: DUI0068B

[Adv01b] ADVANCED RISC MACHINES LTD.: *ARM7TDMI Technical Reference Manual Rev. 3*. http://www.arm.com/pdfs/DDI0029G_7TDMI_R1_trm.pdf. Version: 2001. – Ref: DDI0029G

[AK02] ALLEN, Randy ; KENNEDY, Ken: *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002. – ISBN 1–55860–286–0

[ASU86] AHO, Alfred V. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers*. Addison–Wesley, 1986. – ISBN 0–201–10194–7

[Atm99] ATMEL CORPORATION: *AT91 ARM Thumb Microcontrollers AT91M40400*. http://www.keil.com/dd/docs/datashts/atmel/at91m40400.pdf. Version: 1999

[Ban88] BANERJEE, Utpal: *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988. – ISBN 0–89838–289–0

[Bea05] BEASLEY, John E.: *Integer programming formulation examples*. http://people.brunel.ac.uk/~mastjjb/jeb/or/moreip.html, 2005

[Ben03] BENINI, Luca: Energy–Aware Design of Embedded Memories: A Survey of Technologies, Architectures, and Optimization Techniques. In: *ACM Trasactions on Embedded Computing Systems* 2 (2003), Februar, Nr. 1, S. 5–32

[BMCC03] BROCKMEYER, E. ; MIRANDA, M. ; CORPORAAL, H. ; CATTHOOR, F.: Layer Assignment Techniques for Low Energy in Multi–Layered Memory Organisations. In: *Proceedings of the Conference on Design, Automation and Test in Europe — Volume 1* (2003), März, S. 11070–11075

[Boo51] BOOTH, Andrew D.: A signed binary multiplication technique. In: *Quarterly Journal of Mechanics and Applied Mathematics* 4 (1951), S. 236–240

[Bre96] BREYMANN, Ulrich: *Die C++ Standard Template Library*. Addison–Wesley, 1996. – ISBN 3–8273–1067–9

[Buc04] BUCHWALD, Till: *Plattformabhängigkeit der Effizienz von Schleifentransformationen auf der Quellcodeebene*, Diplomarbeit, Fachbereich Informatik der Universität Dortmund, Dezember 2004

[Bäc96] BÄCK, Thomas: *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996. – ISBN 0–19–509971–0

[Cha82] CHAITIN, Gregory J.: Register allocation & spilling via graph coloring. In: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction* 17 (1982), Juni, S. 98–101

[CKL00] CHANG, Naehyuck ; KIM, Kwanho ; LEE, Hyung G.: Cycle–accurate energy consumption measurement and analysis: case study of ARM7TDMI. In: *Proceedings of the 2000 International symposium on Low power Electronics and Design* (2000), August, S. 185–190

[CSB92] CHANDRAKASAN, Anantha P. ; SHENG, Samuel ; BRODERSEN, Robert W.: Low Power CMOS Digital Design. In: *IEEE Journal of Solid–State Cirquits* 27 (1992), März, Nr. 4, S. 119–123

[Dan51] DANTZIG, George B.: Maximization of a linear function of variables subject to linear inequalities. In: *Activity Analysis of Production and Allocation* (1951), S. 339–347

[Fal04] FALK, Heiko: *Source Code Optimization for Data Flow Dominated Embedded Software*, Dissertation, Fachbereich Informatik der Universität Dortmund, 2004

[FMA+04] FRANCESCO, Poletti ; MARCHAL, Paul ; ATIENZA, David ; BENINI, Luca ; CATTHOOR, Francky ; MENDIAS, Jose M.: An integrated hardware/software approach for run–time scratchpad management. In: *Proceedings of the 41st annual conference on Design automation* (2004), September, S. 238–243

[FSF91] FSF (FREE SOFTWARE FOUNDATION): *General Public License*. http://www.gnu.org/licenses/gpl.txt. Version: 1991

[FV04] FALK, Heiko ; VERMA, Manish: Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization. In: *In Proceedings of the 8th Workshop on Software and Compiler for Embedded Systems, LNCS 3199* (2004), September, S. 137–151

[GKN81] GELLERT, Walter (Hrsg.) ; KÄSTNER, Herbert (Hrsg.) ; NEUBER, Dr. S. (Hrsg.): *Lexikon der Mathematik*. VEB Bibliographisches Institut Leipzig, 1981

[Gon05] GONSALVES, Antone: *31.6 mln mobile phones sold in the US in Q3 2005*. http://blogs.zdnet.com/ITFacts/?p=9322, Oktober 2005

[Gro05] GROUP, Stanford C.: *The SUIF Compiler – Software Distrubution*. http://suif.stanford.edu/suif/suif.html. Version: August 2005

[Gru02] GRUNWALD, Nils: *Energieminimierung eingebetteter Programme durch die dynamische Nutzung eines Scratchpad–Speichers*, Diplomarbeit, Fachbereich Informatik der Universität Dortmund, 2002

[GW02] GONZALEZ, Rafael C. ; WOODS, Richard E.: *Digital Image Processing*. Second Edition. Prentice Hall, 2002. – ISBN 0–201–18075–8

[Hei87] HEIMSOETH SOFTWARE GMBH & CO. PRODUKTIONS UND VERTRIEBS–KG (Hrsg.): *Turbo Pascal 3.0 Handbuch*. 1987

[Hel04] HELMIG, Urs: *Compilergestützte Optimierung von Zugriffen auf partitionierte Speicher*, Diplomarbeit, Fachbereich Informatik der Universität Dortmund, 2004

[HH95] HAFNER, Lutz ; HOFF, Peter: *Genetik*. Schroedel Schulbuchverlag, 1995. – ISBN 3–507–10526–8

[HKKR05] HITZ, Martin ; KAPPEL, Gerti ; KAPSAMMER, Elisabeth ; RETSCHITZEGGER, Werner: *UML@Work*. dpunkt, 2005. – ISBN 3–89864–261–5

[HP02] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach*. 3rd Edition. Morgan Kaufmann Publishers, 2002. – ISBN 1–55860–596–7

[IBMD04] ISSENIN, Ilya ; BROCKMEYER, Erik ; MIRANDA, Miguel ; DUTT, Nikil: Data Reuse Analysis Technique for Software–Controlled Memory Hierachies. In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (2004), S. 202–207

[ICD05a] ICD – INFORMATIK CENTRUM DORTMUND: *ICD–C Compiler framework Developer Manual*. 2005. – Confidential

[ICD05b] ICD – INFORMATIK CENTRUM DORTMUND: *ICD C Compiler framework*. http://www.icd.de/es/icd-c/icd-c.html. Version: August 2005

[KC02] KANDEMIR, M. ; CHOUDHARY, A.: Compiler–directed scratch pad memory hierarchy design and management. In: *39th ACM/IEEE Design Automation Conf.* (2002), June, S. 690–695

[Ker05] KERNCHEN, André: *Compilergestütze Energiereduktion für SDRAM– und Flash–basierte Speichertechnologien*, Diplomarbeit, Fachbereich Informatik der Universität Dortmund, 2005

[Kna01] KNAUER, Markus: *Energiemessung von ARM7TDMI Prozessor–Instruktionen*, Diplomarbeit, Fachbereich Informatik der Universität Dortmund, Juli 2001

[KRI+04] KANDEMIR, Mahmut ; RAMANUJAM, J. ; IRWIN, Mary J. ; VIJAYKRISHNAN, Narayanan ; KADAYIF, Ismail ; PARIKH, Amisha: A compiler based approach for dynamically managing scratch–pad memories in embedded systems. In: *IEEE Transactions on Computer–Aided Design* 23 (2004), Februar, Nr. 2, S. 243–260

[KRS92] KNOOP, Jens ; RÜTHING, Oliver ; STEFFEN, Bernhard: Lazy Code Motion. In: *ACM SIGPLAN Notices* 27 (1992), Juli, S. 24–234

[LCL99] LIM, Amy W. ; CHEONG, Gerald I. ; LAM, Monica S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: *Proceedings of the 13th international conference on Supercomputing* (1999), Mai, S. 228–237

[LEM01] LEE, Sheayun ; ERMEDAHL, Andreas ; MIN, Sang L.: An Accurate Instruction–Level Energy Consumption Model for Embedded RISC Processors. In: *ACM SIGPLAN Notices* 36 (2001), August, S. 1–10

[Leu97] LEUPERS, Rainer: *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997. – ISBN 0–7923–9958–7

[Leu04] LEUPERS, Rainer: *LANCE - Retargetable C compiler*. http://www.icd.de/es/lance/lance.html, 2004

[Lev00] LEVINE, David: *PGAPAck Parallel Genetic Algorithm Library*. http://www-fp.mcs.anl.gov/CCST/research/reports_pre1998/comp_bio/stalk/pgapack.html. Version: Januar 2000. – Philip Hallstrom, David Noelle, Greg Reeder, and Brian Walenz provided significant help.

[LL97] LIM, Amy W. ; LAM, Monica S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: *Proceedings of the 24th ACM SIGPLAN–SIGACT symposium on Principles of programming languages* (1997), Januar, S. 201–214

[LLL01] LIM, Amy W. ; LIAO, Shih-Wei ; LAM, Monica S.: Blocking and array contraction across arbitrarily nested loops using affine partitioning. In: *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming* 36 (2001), Juni, S. 103–112

[LMD+04] LORENZ, Markus ; MARWEDEL, Peter ; DRÄGER, Thorsten ; FETTWEIS, Gerhard ; LEUPERS, Rainer: Compiler based exploration of DSP energy savings by SIMD operations. In: *ASP–DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation* (2004), Januar, S. 838–841

[LW91] LAM, Monica S. ; WOLF, Michael E.: A data locality optimizing algorithm. In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (1991), Mai, S. 30–44

[Mar03] MARWEDEL, Peter: *Embedded System Design*. Kluwer Academic Publishers, 2003. – ISBN 1–4020–7690–8

[MEGC69] MCKELLAR, A. C. ; E. G. COFFMAN, Jr.: Organizing Matrices and Matrix Operations for Paged Memory Systems. In: *Communications of the ACM* 12 (1969), März, Nr. 3, S. 153–165

[Mic03] MICROCHIP TECHNOLOGY INC.: *PIC12F629/675 Data Sheet*. http://ww1.microchip.com/downloads/en/DeviceDoc/41190c.pdf. Version: 2003. – DS41190C

[Muc97] MUCHNICK, Steven S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. – ISBN 1–55860–320–4

[Mös89] MÖSCHWITZER, Albrecht (Hrsg.): *Formeln der Elektrotechnik und Elektronik*. 2., durchgesehene Auflage. VEB Verlag Technik, 1989. – ISBN 3–341–00480–7

[PDN97] PANDA, Preeti R. ; DUTT, Nikil D. ; NICOLAU, Alexandru: Efficient Utilization of Scratch–Pad Memory in Embedded Processor Applications. In: *Proceedings of the 1997 European conference on Design and Test* (1997), März, S. 7–11

[PW92] PUGH, William ; WONNACOTT, David: Eliminating false data dependences using the Omega test. In: *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* 27 (1992), Juli, S. 140–151

[RF92] RÖNZ, Bernd ; FÖRSTER, Erhard: *Regressions– und Korrelationsanalyse*. Gabler, 1992. – ISBN 3–0409–13019–5

[RJ00] REINMANN, Glen ; JOUPPI, Norman P.: CACTI 2.0: An Integrated Cache Timing and Power Model / COMPAQ Western Research Laboratory. 2000 (2000/7). – Forschungsbericht

[Sch86] SCHRIJVER, Alexander: *Theory of linear and integer programming*. Wiley, 1986. – ISBN 0–471–98232–6

[Sch95] SCHWEFEL, Hans-Paul: *Evolution and optimum seeking*. Wiley, 1995. – ISBN 0–471–57148–2

[SGW+02] STEINKE, Stefan ; GRUNWALD, Nils ; WEHMEYER, Lars ; BANAKAR, Rajeshwari ; BALAKRISHNAN, M. ; MARWEDEL, Peter: Reducing energy consumption by dynamic copying of instructions onto onchip memory. In: *Proceedings of the 15th international symposium on System Synthesis* (2002), October, S. 213–218

[SKWM01] STEINKE, Stefan ; KNAUER, Markus ; WEHMEYER, Lars ; MARWEDEL, Peter: An Accurate and Fine Grain Instruction–Level Energy Model supporting Software Optimizations. In: *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation* (2001), September, S. 3.2.1–3.2.10

[SSWM01] STEINKE, Stefan ; SCHWARZ, Ruediger ; WEHMEYER, Lars ; MARWEDEL, Peter: Low power code generation for a RISC processor by register pipelining / Fachbereich Informatik der Universität Dortmund. 2001 (754). – Forschungsbericht

[SW05] STEINKE, Stefan ; WEHMEYER, Lars: *The encc Energy aware C Compiler Homepage.* http://ls12-www.cs.uni-dortmund.de/research/encc/. Version: 2005

[SWLM02] STEINKE, Stefan ; WEHMEYER, Lars ; LEE, Bo-Sik ; MARWEDEL, Peter: Assigning Program and Data Objects to Scratchpad for Energy Reduction. In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (2002), März, S. 409–416

[SZWM01] STEINKE, Stefan ; ZOBIEGALA, Christoph ; WEHMEYER, Lars ; MARWEDEL, Peter: Moving Program Objects to Scratch–Pad Memory for Energy Reduction / Fachbereich Informatik der Universität Dortmund. 2001 (756). – Forschungsbericht

[The00] THEOKHARIDIS, Michael: *Energiemessung von ARM7TDMI Prozessor–Instruktionen*, Diplomarbeit, Fachbereich Informatik der Universität Dortmund, November 2000

[TMW94] TIWARI, Vivek ; MALIK, Sharad ; WOLFE, Andrew: Power analysis of embedded software: a first step towards software power minimization. In: *Proceedings of the 1994 IEEE/ACM international conference on Computer–aided design* (1994), November, S. 384–390

[UB03] UDAYAKUMARAN, Sumesh ; BARUA, Rajeev: Compiler–Decided Dynamic Memory Allocation for Scratch–Pad Based Embedded Systems. In: *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems* (2003), Oktober, S. 276–286

[VSM03] VERMA, Manish ; STEINKE, Stefan ; MARWEDEL, Peter: Data Partitioning for Maximal Scratchpad Usage. In: *ASP–DAC '03: Proceedings of the 2003 conference on Asia South Pacific design automation* (2003), Januar, S. 77–83

[VWM04a] VERMA, Manish ; WEHMEYER, Lars ; MARWEDEL, Peter: Cache–Aware Scratchpad Allocation Algorithm. In: *Proceedings of the conference on Design, automation and test in Europe* (2004), Februar, S. 2.1264–2.1269

[VWM04b] VERMA, Manish ; WEHMEYER, Lars ; MARWEDEL, Peter: Dynamic Overlay of Scratchpad Memory for Energy Minimization. In: *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis* (2004), September, S. 104–109

[WHM04] WEHMEYER, Lars ; HELMIG, Urs ; MARWEDEL, Peter: Compiler–optimized usage of partitioned memories. In: *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture* (2004), June, S. 114–120

[Wil93] WILDE, Doran K.: Memory Reuse Analysis in the Polyhedral Model / IRISA. 1993 (785). – Forschungsbericht

[WJ96] WILTON, Steven J. ; JOUPPI, Norman P.: CACTI: An Enhanced Cache Access and Cycle Time Model. In: *IEEE Journal of Solid State Circuits* 31 (1996), Mai, Nr. 5, S. 677–688

[WL05] WILDE, Doran ; LOECHNER, Vincent: *PolyLib – A library of polyhedral functions.* http://icps.u-strasbg.fr/PolyLib/. Version: September 2005

[WM04] WEHMEYER, Lars ; MARWEDEL, Peter: Influence of Onchip Scratchpad Memories on WCET prediction. In: *Proceedings of the 4th International Workshop on Worst–Case Execution Time (WCET) Analysis*, (2004), June

[Wol92] WOLF, Michael E.: *Improving Locality and Parallelism in Nested Loops*, Dissertation, Stanford University, 1992

[Wol95] WOLFE, Michael: *High Performance Compilers for Parallel Computing*. Addison Wesley Publishing Company, 1995. – ISBN 0–8053–2730–4

[ZKKC03] ZHANG, W. ; KARAKOY, M. ; KANDEMIR, M. ; CHEN, G.: A compiler approach for reducing data cache energy. In: *Proceedings of the 17th annual international conference on Supercomputing* (2003), Juni, S. 76–85

# Erklärung

Hiermit versichere ich, dass ich meine Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dortmund, den 10. November 2005

Thorsten Wilmer