

Diplomarbeit

Energieoptimierende,  
betriebssystemunterstützte,  
online Scratchpad-Allokation  
für  
Multiprozess-Applikationen

Christoph Faßbach

4. September 2006

INTERNE BERICHTE  
INTERNAL REPORTS

Lehrstuhl Informatik XII  
(Technische Informatik und Eingebettete Systeme)  
Fachbereich Informatik  
Universität Dortmund

**Gutachter:**

Robert Pyka  
Prof. Dr. Peter Marwedel



# Inhaltsverzeichnis

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Einführung</b>                                   | <b>1</b>  |
| 1.1      | Motivation . . . . .                                | 1         |
| 1.2      | Ziele der Diplomarbeit . . . . .                    | 2         |
| 1.3      | Aufbau der Diplomarbeit . . . . .                   | 3         |
| <b>2</b> | <b>Grundlagen</b>                                   | <b>5</b>  |
| 2.1      | Zielarchitektur . . . . .                           | 5         |
| 2.1.1    | ARM7-Prozessorfamilie . . . . .                     | 6         |
| 2.1.2    | Speicherarchitekturen . . . . .                     | 7         |
| 2.1.3    | MPARM-Simulator . . . . .                           | 8         |
| 2.1.4    | Energiemodell . . . . .                             | 10        |
| 2.2      | Dynamische Speicherallokation . . . . .             | 11        |
| 2.3      | Eingebettete Betriebssysteme . . . . .              | 12        |
| 2.3.1    | $\mu$ CLinux . . . . .                              | 14        |
| 2.3.2    | eCos . . . . .                                      | 15        |
| 2.3.3    | RTEMS . . . . .                                     | 18        |
| 2.4      | Mathematische Konzepte . . . . .                    | 21        |
| 2.4.1    | Integer Linear Programming . . . . .                | 21        |
| 2.4.2    | Pareto-Optimum . . . . .                            | 22        |
| 2.5      | Verwandte Arbeiten . . . . .                        | 22        |
| <b>3</b> | <b>Scratchpad-Allokation</b>                        | <b>29</b> |
| 3.1      | Vorbetrachtungen . . . . .                          | 29        |
| 3.2      | Belegung mit Blockieren der Objekte . . . . .       | 33        |
| 3.2.1    | Optimale Lösung mittels ILP . . . . .               | 34        |
| 3.2.2    | Statische Belegung . . . . .                        | 39        |
| 3.2.3    | Heuristische Belegung mit First-Fit . . . . .       | 40        |
| 3.2.4    | Heuristische Belegung mit Best-Fit . . . . .        | 42        |
| 3.2.5    | Heuristische Belegung in einem Durchlauf . . . . .  | 43        |
| 3.2.6    | Heuristische Belegung in drei Durchläufen . . . . . | 45        |
| 3.3      | Belegung mit Replatzierung der Objekte . . . . .    | 48        |
| 3.3.1    | Optimale Lösung mittels ILP . . . . .               | 48        |
| 3.3.2    | Dynamische Belegung . . . . .                       | 50        |

|          |  |            |
|----------|--|------------|
| 3.3.3    | Heuristische Belegung mit Blockstrategie . . . . . | 51         |
| 3.4      | Bewertung . . . . .                                | 53         |
| <b>4</b> | <b>Arbeitsablauf und Implementierung</b>           | <b>57</b>  |
| 4.1      | Arbeitsablauf zur Compilezeit . . . . .            | 57         |
| 4.2      | Schnittstellenbeschreibung des SPMM . . . . .      | 61         |
| 4.3      | Erweiterung der Prozesserzeugung . . . . .         | 65         |
| 4.4      | Implementierung des SPMM . . . . .                 | 67         |
| 4.4.1    | ILP-basierte Belegung . . . . .                    | 75         |
| 4.4.2    | Heuristische Belegung . . . . .                    | 77         |
| 4.5      | Integration in das Zielbetriebssystem . . . . .    | 78         |
| 4.5.1    | $\mu$ CLinux . . . . .                             | 78         |
| 4.5.2    | eCos . . . . .                                     | 79         |
| 4.5.3    | RTEMS . . . . .                                    | 79         |
| <b>5</b> | <b>Ergebnisse</b>                                  | <b>81</b>  |
| 5.1      | Beschreibung der Benchmarks . . . . .              | 81         |
| 5.2      | Diskussion der Ergebnisse . . . . .                | 84         |
| 5.3      | Rechenaufwand und Lösungsgüte . . . . .            | 93         |
| 5.4      | Vergleich des SPMM mit Caches . . . . .            | 95         |
| <b>6</b> | <b>Zusammenfassung und Ausblick</b>                | <b>97</b>  |
| <b>A</b> | <b>ILP-Gleichungen zur Adresszuweisung</b>         | <b>103</b> |
|          | <b>Literaturverzeichnis</b>                        | <b>105</b> |

# Abbildungsverzeichnis

|     |   |    |
|-----|---|----|
| 2.1 | MARM-Systemarchitektur . . . . .                                  | 9  |
| 2.2 | MARM-Prozessorsubsystem . . . . .                                 | 10 |
| 2.3 | Speicherstruktur des Slab-Allocators . . . . .                    | 16 |
| 2.4 | Beispielhafter Aufbau einer eCos-Applikation . . . . .            | 17 |
| 2.5 | Beispiel für die Verwendung des Rate Monotonic Managers . . . . . | 21 |
|     |   |    |
| 3.1 | Beispiel für die zusätzliche Dereferenzierungsstufe . . . . .     | 31 |
| 3.2 | Zugriffsfehler bei Benutzung gepufferter Adressen . . . . .       | 32 |
| 3.3 | Schutz der Datenstrukturen des SPMM . . . . .                     | 33 |
| 3.4 | Beispiel für Fragmentierung . . . . .                             | 38 |
| 3.5 | Umsortierung der Speicherobjekte . . . . .                        | 41 |
| 3.6 | Belegung in einem Durchlauf . . . . .                             | 44 |
| 3.7 | Belegung in drei Durchläufen . . . . .                            | 47 |
| 3.8 | Beispiel für Replatzierung . . . . .                              | 49 |
|     |   |    |
| 4.1 | Darstellung des Arbeitsablaufes . . . . .                         | 61 |
| 4.2 | Begründung des durchschnittlichen Nutzenmaßes . . . . .           | 68 |
| 4.3 | Aufbau der Liste der Scratchpad-Speicherbereiche . . . . .        | 70 |
| 4.4 | Aufbau der Skiplisten-Struktur . . . . .                          | 71 |
| 4.5 | Beispiele für Suchvorgänge auf Skiplisten . . . . .               | 72 |
| 4.6 | Aufbau eines Buddy-Baumknotens . . . . .                          | 73 |
| 4.7 | Abbildung des Baumes auf das Feld . . . . .                       | 73 |
| 4.8 | Beispiel für die Benutzung des Buddy-Systems . . . . .            | 74 |
| 4.9 | Einfluß der Komponenten des ILP-Verfahrens aufeinander . . . . .  | 76 |
|     |   |    |
| 5.1 | Ergebnis des AUTO-Benchmarks . . . . .                            | 85 |
| 5.2 | Ergebnis des TELECOM-Benchmarks . . . . .                         | 86 |
| 5.3 | Ergebnis des MEDIA <sup>-</sup> -Benchmarks . . . . .             | 87 |
| 5.4 | Ergebnis des MEDIA <sup>+</sup> -Benchmarks . . . . .             | 88 |
| 5.5 | Ergebnis des SORT-Benchmarks . . . . .                            | 89 |
| 5.6 | Abweichung der blockierenden Heuristiken vom Optimum . . . . .    | 91 |
| 5.7 | Abweichung der replatzierenden Heuristiken vom Optimum . . . . .  | 92 |
| 5.8 | Abweichung der Blockierenden von den Replatzierenden . . . . .    | 93 |
| 5.9 | Energiebedarf über Belegungsaufwand des SPMM . . . . .            | 94 |

|   |    |
|---|----|
| 5.10 SPMM vs. Cache; SORT-Benchmark . . . . . | 95 |
| 5.11 SPMM vs. Cache; AUTO-Benchmark . . . . . | 96 |

# Tabellenverzeichnis

|     |  |    |
|-----|--|----|
| 3.1 | Laufzeiten der Heuristiken . . . . .                                     | 54 |
| 5.1 | Übersicht über die Benchmarks . . . . .                                  | 83 |
| 5.2 | Übersicht über das Ergebnis des AUTO-Benchmarks . . . . .                | 85 |
| 5.3 | Übersicht über das Ergebnis des TELECOM-Benchmarks . . . . .             | 86 |
| 5.4 | Übersicht über das Ergebnis des MEDIA <sup>-</sup> -Benchmarks . . . . . | 87 |
| 5.5 | Übersicht über das Ergebnis des MEDIA <sup>+</sup> -Benchmarks . . . . . | 88 |
| 5.6 | Übersicht über das Ergebnis des SORT-Benchmarks . . . . .                | 89 |
| 5.7 | Durchschnittlicher Energie- und Zyklenbedarf aller Benchmarks . . . . .  | 91 |



# Kapitel 1

## Einführung

Die Gestalt und die Verwendung der Rechnersysteme waren und sind einem stetigen Wandel unterworfen. Waren die ersten Computer noch teuer, groß und wurden von mehreren Personen Tag und Nacht genutzt, so benutzen heute einzelne Personen Tag und Nacht mehrere Rechner, größtenteils ohne sich dessen bewußt zu sein. Während man die Arbeitsrechner bei steigender Rechenleistung immer weiter bis zum Personal Computer verkleinerte, wurden Dinge des täglichen Bedarfs zu Rechnersystemen. Diese so genannten eingebetteten Systeme sind der Definition[Mar03] nach informationsverarbeitende Systeme, die in ein größeres Produkt eingebettet und daher normalerweise für den Benutzer nicht direkt sichtbar sind.

Die Anwendungsbereiche solcher eingebetteten Systeme sind vielfältig. Von Automobilen, Flugzeugen und allen anderen Arten von Fahrzeugen über Haushaltselektronik wie Waschmaschinen, Herden und Mikrowellen bis zu elektronischen Notizbüchern, so genannten persönlichen digitalen Assistenten (PDA), haben diese Rechnersysteme nahezu überall Einzug gehalten. Telefone wurden hierdurch nicht nur zu Mobiltelefonen, sondern entwickelten sich so rasant weiter, dass sie beim weltgrößten Hersteller von Mobiltelefonen nicht mehr Telefone genannt werden dürfen[Dur06]. Dort heißen sie Multimedia Computer.

Die Benutzung eingebetteter Systeme in allen Formen und Ausprägungen ist so selbstverständlich geworden, dass heute ein Großteil der produzierten Prozessoren in eingebetteten Systemen eingesetzt wird. Gerade mobile Systeme wie PDAs und Mobiltelefone begleiten ihre Benutzer ständig und müssen daher besonderen Kriterien genügen.

### 1.1 Motivation

Mobile Geräte müssen immer mehr bieten, damit die Hersteller ihre Geräte weiterhin verkaufen können, da die Marktdurchdringung bereits extrem hoch ist. Bei Mobiltelefonen liegt sie beispielsweise bei über 80%[DZ05]. Po-

sitive Verkaufsargumente für mobile Systeme sind unter anderem geringere Größe, kleineres Gewicht, neue Fähigkeiten und längere netzunabhängige Laufzeiten.

Um weitere, neue Fähigkeiten zu integrieren wird immer mehr Rechenleistung benötigt. Die Rechenleistung bereitzustellen ist nicht schwierig, da die Halbleiterindustrie ständig schnellere Prozessoren entwickelt. Zusätzliche Rechenleistung bedeutet aber auch zusätzlichen Energiebedarf, der in mobilen Geräten nicht auf Kosten der Laufzeit gehen soll.

Mobile eingebettete Systeme beziehen die benötigte Energie üblicherweise aus Akkumulatoren, kurz Akkus. Die Kapazität von Akkus ist begrenzt, womit auch die netzunabhängige Laufzeit mobiler Geräte begrenzt ist. Die Weiterentwicklung der Akkus in Bezug auf Kapazitätssteigerungen ist sehr gering. In den ersten sieben Jahren seit ihrer Markteinführung 1991 konnten Lithium-Ionen Akkus ihre Energiedichte verdoppeln [GK03], während das Mooresche Gesetz eine Verdoppelung der Anzahl integrierter Transistoren je Chip alle 18 Monate vorsieht. Die Steigerung der Akkukapazität kann mit den Fortschritten in der Halbleiterindustrie nicht Schritt halten. Auch der Einsatz größerer oder schwererer Akkus verbietet sich für tragbare Systeme aufgrund der Optimierungskriterien Größe und Gewicht von selbst.

Mehr Energieverbrauch bedeutet aber auch, dass mehr Wärme entsteht. Wärme ist in mobilen Endgeräten nicht erwünscht. Zum einen sollte der Benutzer sein portables Gerät berühren können, ohne sich zu verbrennen. Zum anderen wirken sich hohe Temperaturen im Geräteinneren negativ auf die Akkus aus[Buc01]. Kühlsysteme, die der Wärmeentwicklung entgegen wirken, machen die Geräte schwerer, größer und teurer.

Aus Gründen der Laufzeit und der Wärmeentwicklung ist es also sinnvoll, Energie zu sparen. Allein der Cache verbraucht bis zu 40% der Energie bei ARM-Prozessoren[KG02]. Das ganze Speichersubsystem wurde als einer der Hauptenergieverbraucher identifiziert und ist daher ein geeigneter Ansatzpunkt zum Energiesparen, zum Beispiel durch den Einsatz sparsamerer Speicherbausteine.

## 1.2 Ziele der Diplomarbeit

Das Speichersubsystem ist, wie im vorherigen Abschnitt beschrieben, ein geeigneter Ansatzpunkt für Energiesparmaßnahmen, da dort ein Großteil der Energie des Gesamtsystems verbraucht wird. Deshalb beschäftigt sich seit einigen Jahren die Forschung mit verschiedenen gestalteten Speicherhierarchien. Die Idee ist es, möglichst viele Zugriffe auf den Hauptspeicher durch Zugriffe auf einen kleinen Speicher zu ersetzen. Da kleine Speicher schneller sind als große, kann so nicht nur die Ausführungszeit verbessert werden, sondern es wird auch Energie gespart, da Wartezyklen entfallen. Zusätzlich ist jeder einzelne Zugriff auf einen kleinen Speicher weniger energieaufwän-

dig als ein entsprechender Zugriff auf einen großen Speicher. Zwei mögliche Ausprägungen dieser kleinen Speicher sind Caches und Scratchpad-Speicher.

Caches werden durch zusätzliche Hardware belegt und dann automatisch vom Prozessor benutzt. Diese zusätzliche Hardware verbraucht nicht nur Chipfläche, sondern auch signifikante Mengen Energie, so dass alternative Speicherhierarchien, zum Beispiel die des Scratchpad-Speichers, untersucht wurden. Scratchpad-Speicher werden nicht durch Hardware belegt. Sie werden im Gegensatz zu den Caches im Adressraum eingebündelt und müssen explizit von den Applikationen belegt und benutzt werden. Neben der Ersparnis von Energie und Chipfläche ist ein weiterer Vorteil dieser Architektur, dass zu jeder Zeit die Belegung des Scratchpad-Speichers bekannt sein kann. Daher sind Scratchpad-Speicher in der Lage, die für Realzeitsysteme wichtige maximale Ausführungszeit positiv zu beeinflussen, während für Caches die Vorhersage ihres Inhaltes weit schwieriger ist, und sie daher meist nur die durchschnittliche Ausführungszeit verbessern.

Um Scratchpad-Speicher gewinnbringend einzusetzen, wurden verschiedene Ansätze untersucht. Die meisten davon betrachten Optimierungen zur Designzeit für einen Prozess oder eine feste Auswahl von Prozessen. Wie bereits im vorangegangenen Abschnitt erwähnt, sind die Hersteller mobiler Systeme gezwungen, ständig neue Funktionen in ihre Produkte zu integrieren. Dazu gehört auch, dass nicht nur mehrere Prozesse gleichzeitig laufen, sondern auch die Fähigkeit, Benutzerprogramme auszuführen. Programme, die ein Benutzer selbst vorgibt, sind nicht zur Design-Zeit bekannt und können von einem solchen Ansatz nicht erfasst werden. Auch die Optimierung dynamisch verwalteten Speichers ist diesen Ansätzen unmöglich. Um jedem Prozess eines Multiprozesssystems zur Laufzeit Zugriff auf einen Teil des Scratchpad-Speichers zu geben, ohne dass die Prozesse sich dabei gegenseitig behindern, ist eine automatische Verwaltung dieser Ressource nötig.

Ziel der vorliegenden Diplomarbeit ist es, eine Scratchpad-Allokationstechnik zu entwickeln, die sich in bestehende Betriebssysteme integrieren lässt. Der Scratchpad-Speicher soll zur Laufzeit vom Betriebssystem belegt werden, so dass mehrere gleichzeitig laufende Applikationen hinsichtlich ihres Energieverbrauchs optimiert werden. Die Applikationen liefern hierzu lediglich eine Gewichtung der von ihnen verwendeten Speicherobjekte. Aufgrund der Integration in das Betriebssystem wird Unterstützung präemptiven Multitaskings verlangt. Neben der Güte der Allokationstechnik im Vergleich zu einer optimalen Belegung wird auch der Einfluss der Allokationsstrategie auf Echtzeitfähigkeit und Vorhersagbarkeit betrachtet.

### 1.3 Aufbau der Diplomarbeit

Nach dem Überblick über den Aufbau der Diplomarbeit werden in Kapitel 2 einige wichtige Grundlagen eingeführt. Zunächst wird die Zielarchitektur aus

dem ARM7TDMI-Prozessor und der Speicherhierarchie beschrieben. Eine Übersicht über die verwendete Simulationsumgebung rundet die Beschreibung der Zielarchitektur ab. Es folgt eine kurze Einführung in die dynamische Speicherallokation, bevor auf die Grundlagen eingebetteter Betriebssysteme und die drei Open-Source Betriebssysteme  $\mu$ CLinux, eCos und RTEMS eingegangen wird. Das Kapitel endet mit einer Darstellung über die mathematischen Grundlagen und einem Abschnitt über verwandte Arbeiten.

In Kapitel 3 werden verschiedene Möglichkeiten der Scratchpad-Allokation für Multiprozess-Applikationen vorgestellt. Nach einer Vorbetrachtung der Problemstellung werden in den folgenden zwei Abschnitten zwei verschiedene Ansätze mit verschiedenen Strategien entwickelt, die die Probleme zu lösen versuchen. Jede der Strategien wird beschrieben, ihre Laufzeit analysiert und ihre Auswirkungen auf Echtzeitfähigkeit und Vorhersagbarkeit untersucht.

Kapitel 4 beschreibt zunächst, welche Schritte nötig sind, um Applikationen in mehreren Schritten an die Scratchpad-Allokationstechnik des Betriebssystems anzupassen. Anschließend wird die Schnittstelle des Scratchpad-Speichermanagers, sowie dessen Implementierung und Integration in die verschiedenen Betriebssysteme, erklärt.

Die mit Hilfe des Scratchpad-Speichermanagers erzielten Ergebnisse werden in Kapitel 5 vorgestellt. Zuerst werden die verwendeten Benchmarks und ihre Ergebnisse beschrieben. Es folgt ein Vergleich der heuristischen Lösungsstrategie mit der Optimalen des jeweiligen Ansatzes und der beiden Ansätze miteinander. Dann wird der von der Allokationsstrategie erzeugte Overhead der erzielten Lösungsgüte gegenübergestellt. Abschließend werden die Ergebnisse des Scratchpad-Speichermanagers und die einer Cache-basierten Architektur gegeneinander abgewogen.

Die vorliegende Arbeit endet mit einer Zusammenfassung der Ergebnisse und dem Ausblick auf mögliche Erweiterungen in Kapitel 6.

# Kapitel 2

## Grundlagen

Die im Rahmen dieser Arbeit entwickelten Scratchpad-Allokationstechniken sind an eine bestimmte Zielarchitektur angepasst, die in Abschnitt 2.1 beschrieben wird. Der darauf folgende Abschnitt 2.2 erklärt die Aufgaben der dynamischen Speicherverwaltung durch das Betriebssystem. Abschnitt 2.3 beleuchtet eingebettete Betriebssysteme genauer und geht auf drei Beispielsysteme ein.

Optimale Lösungen, sowohl für die Allokation als auch für die Platzierung, werden in dieser Arbeit durch ILP-Gleichungen formuliert. Die Online-Heuristiken, die die gleichen Aufgaben zu bewältigen haben, versuchen die Anzahl der Speicherzugriffe auf den Hauptspeicher zu minimieren und dabei den Overhead für ihre Berechnungen und das Kopieren von Speicherobjekten gering zu halten. Die Lösungen für diese beiden gegensätzlichen Ziele sollten nach Möglichkeit Pareto-optimal gewählt werden. Eine minimale Einführung in diese beiden grundlegenden mathematischen Konzepte gibt Abschnitt 2.4.

Am Ende dieses Kapitels wird in Abschnitt 2.5 ein Überblick über Arbeiten Anderer gegeben, die dieser Arbeit als Grundlage dienen oder ähnliche Ansätze oder Zielsetzungen verfolgen.

### 2.1 Zielarchitektur

In diesem Abschnitt wird die Zielarchitektur vorgestellt, für welche die im Rahmen der Diplomarbeit entwickelte Scratchpad-Allokationstechnik gedacht ist. Sie umfasst neben einem Prozessorkern aus der ARM7-Prozessorfamilie noch eine Speicherhierarchie. Da die Programme, die es zu optimieren gilt, nicht auf realer Hardware, sondern auf dem MPARM-Simulator ausgeführt werden, wird auch dieser beschrieben. Abgeschlossen wird dieser Abschnitt mit einer Beschreibung des vom MPARM-Simulator verwendeten Energiemodells.

### 2.1.1 ARM7-Prozessorfamilie

Die Repräsentanten der ARM7-Familie sind 32 Bit RISC Prozessoren, die von ARM Limited als eingebettete Mikroprozessoren vertrieben werden. Ihr geringer Bedarf an Chipfläche, ihr niedriger Energieverbrauch und ihre hohe Rechenleistung macht sie besonders geeignet für mobile, eingebettete Anwendungen[ARM01b], wie Pager, Mobiltelefone, PDAs oder MP3-Player.

Neben vollständigen Prozessoren sind auch ARM7-Prozessorkerne als synthetisierbare IP (Intellectual Property) in Verilog oder VHDL verfügbar, so dass Hersteller eigene Varianten, siehe z.B. [Cir01], mit zusätzlicher Peripherie auf einem Chip verwirklichen können.

Die wesentlichen, gemeinsamen Eigenschaften der ARM7-Prozessoren sind:

- 32 Bit Von Neumann Load/Store RISC Architektur
- 3-stufige Pipeline: Fetch, Decode, Execute<sup>1</sup>
- 37 Register mit je 32 Bit, deren Sichtbarkeit vom Betriebsmodus abhängig ist
- 4 GB Adressraum
- Byte (8 Bit), Halbwort (16 Bit) und Wort (32 Bit) Datentypen
- 2 Instruktionssätze: ARM- und Thumb-Befehlssatz<sup>1</sup>

Programmen stehen auf Prozessoren dieser Familie zwei verschiedene Befehlssätze zur Verfügung.

Der ARM-Befehlssatz umfasst 80 grundlegende, 32 Bit breite Befehle. Jeder davon kann bedingt, in Abhängigkeit von Status-Flags im Status-Register, ausgeführt werden. Vor dem Laden von Registern mit Konstanten oder arithmetischen Instruktionen kann ohne Verzögerung auf einen der Operanden der Barrelshifter angewendet werden. Im ARM-Mode stehen dem Programmierer 15 32 Bit breite Register zur freien Verfügung.

Der Thumb-Befehlssatz besteht aus lediglich 36 16 Bit breiten Befehlen. Aufgrund der kleineren Befehlswortlänge sind lediglich 8 Register frei verwendbar. Gegenüber dem ARM-Mode muss im Thumb-Mode auf verschiedene Adressierungsarten, den Barrelshifter und die bedingte Ausführung aller Befehle verzichtet werden. Zudem wird bei manchen Instruktionen das 3-Register-Format auf ein 2-Register-Format beschränkt.

Jeder Thumb-Befehl wird vor seiner Ausführung ohne Zeitverlust in sein ARM-Äquivalent umgewandelt, intern werden also nur ARM-Instruktionen verwendet. Der Wechsel zwischen beiden Betriebsmodi ist mit einem speziellen Befehl möglich. Trotz halber Befehlswortlänge sind identische Programme im Thumb-Mode nur um etwa 35% kleiner als im ARM-Mode, denn es

<sup>1</sup>Ausnahme: ARM7EJ-S[ARM01a] mit nativer Java-Bytecode Ausführung

werden mehr Befehle benötigt, um die gleichen Berechnungen durchzuführen. Da die Befehls Worte jedoch kleiner sind, können sich die Cache-Hit-Rate und der Speicherdurchsatz positiv entwickeln, so dass Aussagen bezüglich Ausführungsgeschwindigkeit und Energieeffizienz ebenfalls Gegenstand der Forschung sind.

Eine ausführliche Darstellung der Merkmale der ARM7-Familie findet sich in [ARM04]. Auf die Einzelheiten von ARM- und Thumb-Befehlssatz wird in [Sea00] eingegangen.

### 2.1.2 Speicherarchitekturen

Grundsätzlich wird zwischen dynamischen Speichern (DRAMs) und statischen Speichern (SRAMs) unterschieden. Bei beiden Technologien werden die Speicher in Bitfeldern zu Zeilen und Spalten organisiert. Eine Dekodierlogik steuert die Lese- und Schreibzugriffe. Zunächst übernimmt ein Zeilendekodierer einen Teil der Adresse und wählt die entsprechende Speicherzeile aus. Sobald die Zeile selektiert ist, wählt ein Spaltendekodierer die gewünschte Speicherzelle aus. Die Busbreite gibt dabei an, wieviele Speicherzellen parallel ausgelesen werden.

Bei DRAMs wird die Information eines Bits in einem geladenen oder nicht geladenen Kondensator gespeichert. Problematisch ist dabei, dass die gespeicherten Ladungen dieser winzigen Kondensatoren ebenfalls sehr klein sind. Da die Ladungen so klein sind, müssen sie durch Leseverstärker aufbereitet werden, um nutzbar zu sein. Um die Informationen nicht durch Leckströme zu verlieren, müssen sie regelmäßig in sogenannten Refreshzyklen neu eingeschrieben werden. Jeder Lesevorgang einer Zeile zerstört dabei alle darin gespeicherten Informationen, die Zeile muss daher abermals neu gespeichert werden.

SRAMs speichern die Informationen in Flip-Flops, die jeweils aus mehreren Transistoren bestehen [Gos91]. Technologiebedingt ist die Zugriffszeit bei SRAMs geringer, wenn sie in ähnlicher Strukturgröße wie DRAMs gefertigt werden. Eine Gegenüberstellung verschiedener Speichertechnologien für eingebettete Systeme liefert [BMP03].

Kleinere Speicher können generell schneller betrieben werden und brauchen weniger Energie als größere Speicher gleicher Technologie. Große Speicher brauchen mehr Chipfläche, so dass längere Leitungen mit höherer Kapazität nötig sind. Das Umladen der Kapazitäten benötigt Zeit und Energie. Mit kleineren Strukturen lässt sich der Kapazität in gewissem Maße entgegenwirken, es entstehen dann jedoch andere Probleme [Hil04]. Kleinere Speicher besitzen weniger und nicht so tiefe Zeilendecoder, so dass auf sie mit geringerer Verzögerung zugegriffen werden kann.

Die Rechenleistung von Prozessoren wächst jedes Jahr um 50 bis 100%, während die Geschwindigkeit der Hauptspeicher lediglich um 7% pro Jahr zunimmt [HP02]. Es ist offensichtlich und seit langem bekannt, dass die Spei-

cherzugriffe den Prozessor ernsthaft ausbremsen, der wachsenden Rechenleistung sozusagen eine Wand in den Weg stellen[WM95]. Um trotz langsamen Hauptspeichers immer schnellere Prozessoren sinnvoll einsetzen zu können, werden Speicherhierarchien eingesetzt, die die Anzahl der Hauptspeicherzugriffe und die damit verbundenen Wartezeiten verringern sollen.

Eine Möglichkeit ist die Verwendung von kleinen, schnellen Speichern, die nicht direkt vom Prozessor adressiert werden können. Diese als Caches bezeichneten Speicher sind in mehreren Cache-Zeilen organisiert. Jede Cache-Zeile enthält Informationen aus einem zusammenhängenden Teil des Hauptspeichers. Im Tag-Speicher wird zu jeder Zeile die Startadresse der enthaltenen Informationen abgelegt. Wird auf eine Adresse zugegriffen, vergleicht zusätzliche Hardware die benötigte Adresse mit den im Tag-Speicher vorhandenen Adressen. Bei einem sogenannten Cache-Hit wird eine Übereinstimmung gefunden und die benötigte Adresse befindet sich im Cache, andernfalls spricht man von einem Cache-Miss. Bei einem Cache-Hit kann der Hauptspeicherzugriff entfallen. Bei einem Cache-Miss wird der Inhalt, der zu der Adresse gehörenden Cache-Zeile, aus dem Hauptspeicher in den Cache übernommen. Hierzu ist ebenfalls zusätzliche Hardware erforderlich. Für Details zum Cache sei auf [Sch03] verwiesen. Durch den großen Hardwareaufwand sind Cachezugriffe relativ energieaufwändig.

Eingebettete Systeme auf Basis von Mikroprozessoren, wie dem ARM7, umfassen meist auch eine Speicherhierarchie. Neben Caches kommen hierbei auch Scratchpad-Speicher zum Einsatz. Scratchpad-Speicher sind, verglichen mit dem Hauptspeicher, kleinere und schnellere Speicher, die direkt vom Prozessor adressierbar sind und deren Belegung nicht durch zusätzliche Hardware, sondern softwaregestützt erfolgt. Diese Softwareunterstützung kann durch den Programmierer, den Compiler oder durch das Betriebssystem erfolgen. Üblicherweise werden Scratchpad-Speicher aus SRAM-Zellen gefertigt und in den Prozessor integriert. So sind Scratchpad-Speicher nicht nur sehr schnell, sondern sie bieten auch einen geringen Energiebedarf. Viele Prozessoren, die für eingebettete Systeme gedacht sind, enthalten sowohl einen Cache als auch einen oder mehrere Scratchpad-Speicher. Zugriffe auf Scratchpad-Speicher werden nicht vom Cache abgefangen.

### 2.1.3 MPARM-Simulator

Anstelle von realer Hardware und aufwändigen Messungen[IY97] des Energieverbrauches wird im Rahmen dieser Diplomarbeit der MPARM-Simulator benutzt. Der Simulator ermittelt nicht nur die exakte Anzahl der benötigten Zyklen und erzeugt detaillierte Trace-Dateien. Er enthält ebenfalls ein Energiemodell, mit dessen Hilfe er den Energiebedarf des Systems während eines Simulationslaufs abschätzen kann. Auf das zugrunde liegende Energiemodell wird im nächsten Abschnitt eingegangen.

Der MPARM-Simulator wurde ursprünglich zur Simulation von sogenannten MPSoCs (Multi Processor Systems on a Chip) entwickelt [LAB<sup>+</sup>04]. Dem entsprechend simuliert MPARM nicht nur mehrere Prozessorkerne gleichzeitig, sondern implementiert zwischen den einzelnen Prozessoren auch zwei verschiedene Bussysteme wie ARM AMBA AHB oder ST Microelectronics STBus. Private oder gemeinsame Speicher der Prozessoren, ein Interruptcontroller und ein Semaphorencontroller werden ebenso mit SystemC modelliert wie der Systembus, der sie verbindet. Die Verwendung verschiedener, in C oder C++ realisierter Instruktionssatzsimulatoren ist durch SystemC-Wrapper möglich. Simulatoren für ARM7, StrongARM, PowerPC 750 und MIPS R3000 Prozessoren wurden und werden bereits portiert. Heterogene und homogene Multiprozessorsysteme sind möglich.

Zur Simulation von ARM7-Prozessoren wird der Befehlssatzsimulator SoftWare ARM, kurz SWARM [Dal00], benutzt. Ausser der Simulation des ARM-Befehlssatzes mit möglichst exaktem Datenpfad bietet SWARM einen einfachen UART, einen LCD-Controller und die Möglichkeit, verschiedene Cache-Konfigurationen auszuwählen. Der Thumb-Befehlssatz wird nicht unterstützt.

Die im MPARM implementierte Systemarchitektur wird in Abbildung 2.1 dargestellt. Jeder Prozessorkern ist in ein eigenes Prozessorsubsystem integriert. Mit Hilfe des Interruptcontrollers ist es möglich, zwischen den Prozessoren Nachrichten auszutauschen. Der Semaphorencontroller ermöglicht es, test-and-set-Operationen auszuführen.

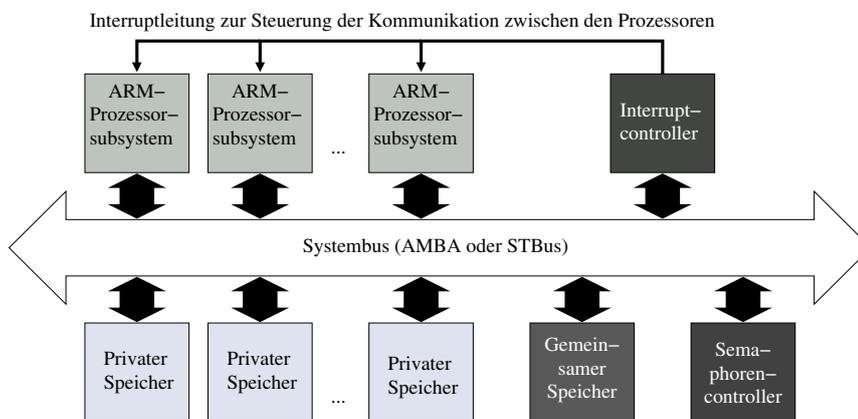


Abbildung 2.1: MPARM-Systemarchitektur

Die Prozessorsubsysteme sind gemäß Abbildung 2.2 aufgebaut. Cache und Scratchpad-Speicher sind mit einem lokalen Speicherbus verbunden. Wenn Daten aus dem Hauptspeicher in den Cache oder aus dem Cache in den Hauptspeicher transportiert werden sollen, transferiert der Speichercontroller die Cache-Zeilen in schnellen Burst-Zugriffen über den Busmaster und den Systembus. Um Daten zwischen Hauptspeicher und Scratchpad zu

transferieren, werden die Daten zuerst in den Cache transferiert und erst dann an ihr eigentliches Ziel geschrieben. Zugriffe auf das Scratchpad erfolgen jedoch immer am Cache vorbei. Sowohl Cache als auch Scratchpad sind abschaltbar. Fehlt das Scratchpad, so ist es nicht adressierbar, fehlt der Cache, so erfolgen alle Zugriffe direkt auf den Hauptspeicher.

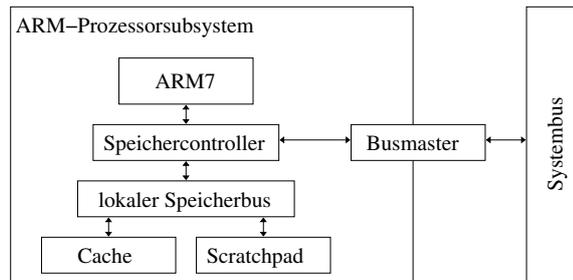


Abbildung 2.2: M-ARM-Prozessorsubsystem

#### 2.1.4 Energiemodell

M-ARM umfasst für alle verwendeten Komponenten, mit Ausnahme des Interrupt- und des Semaphorencontrollers, eigene Energiemodelle. Die Auswertung erfolgt während des Simulationslaufes. Die Ergebnisse werden in einen Statusbericht geschrieben. Die Energieverbrauchswerte entstammen alle der gleichen  $0,13\ \mu\text{m}$  Technologie und wurden von STMicroelectronics zur Verfügung gestellt.

Für die Prozessorkerne wird ein instruktionsbasiertes Energiemodell verwendet, wie es in [TMW94] vorgeschlagen wurde. Das Modell ist zustandsbasiert implementiert, wobei lediglich zwischen den Zuständen *RUNNING* und *IDLE* unterschieden wird. Der *RUNNING*-Zustand wird eingenommen, wenn die Pipeline Instruktionen verarbeitet. Steht die Pipeline, zum Beispiel weil ein Cache-Miss auftritt, wird der *IDLE*-Zustand angenommen. Gemäß [BBC<sup>+</sup>01] beträgt die Abweichung dieses Modells gegenüber einer Energieschätzung auf Registertransferebene unter 5%.

Die Energiebestimmung für Hauptspeicher, Scratchpad-Speicher und Caches erfolgt mit Hilfe eines analytischen Modells, das aus [CZG98] abgeleitet ist. Es wird primär zwischen Schreib- und Lesezugriffen unterschieden. In Abhängigkeit von der Größe des gelesenen oder geschriebenen Speichers und der Art des Zugriffs, lesend oder schreibend, wird eine Energie je Zugriff ermittelt. Für Caches werden zusätzlich zum reinen Informationszugriff in Abhängigkeit von der Cache-Organisation Kosten für den Tag-RAM-Zugriff bestimmt. Die Details zum Energiemodell für Speicher und Cache sind in [LPB04] zu finden.

Das Energiemodell für den STBus stammt aus [BZZ04]. Es berechnet die aufgewandte Energie pro Taktzyklus anhand der Anzahl der Datenpa-

kete, die auf dem Bus übertragen werden sollen. Dank der signalbasierten Implementierung des MPARM-Simulators in SystemC ist durch Analyse der Bus-Request- und Bus-Grant-Signale die genaue Zahl der Geräte bekannt, die auf dem Bus Daten übertragen.

## 2.2 Dynamische Speicherallokation

Dynamische Speicherallokation[WJNB95] ist eine grundlegende Eigenschaft der meisten Rechnersysteme seit ungefähr 1960. Unter dynamischer Speicherallokation versteht man die Vergabe von Arbeitsspeicherbereichen an Prozesse während das System läuft. Bei jedem Programmstart muss das Betriebssystem dem Programm Speicher zur Verfügung stellen, in dem zum einen die auszuführenden Instruktionen und zum anderen die Daten abgelegt werden. Der Datenbereich läßt sich weiter untergliedern in:

- Statische Daten, deren Größe sich während der gesamten Programmausführung nicht verändert,
- Stack-Speicher, der die lokalen Daten einzelner Funktionen enthält und mit Funktionsaufrufen oder Rücksprüngen wächst beziehungsweise schrumpft,
- Heap-Speicher, in dem die Speicherobjekte abgelegt werden, die das Programm zur Laufzeit anlegt. In C wird solcher Speicher mit `malloc()` angefordert, in C++ werden mit `new()` erzeugte Objekte ebenfalls dort angelegt.

Das Zuteilen des Speichers geschieht sowohl beim Programmstart als auch während der Programmausführung letztlich durch das Betriebssystem. Im Betriebssystem wird diese Aufgabe von Speichermanagern übernommen. Der Speichermanager entscheidet, welches Speicherobjekt wo abgelegt werden soll. Trotz dieser scheinbar einfachen Aufgabe ist das Problem der Dynamic Storage Allocation, kurz DSA, als NP-hart[Weg03] nachgewiesen[GJ79]. Damit der Speichermanager Programmen Speicher zuweisen kann und die Programme den Speicher wieder freigeben können, benötigt der Speichermanager Informationen sowohl über die freien als auch über die belegten Speicherbereiche. Die Entscheidungen müssen auf Basis der momentanen Situation und der aktuellen Speicheranforderung getroffen werden. Jede getroffene Entscheidung beeinflusst jedoch die Zukünftigen. Bei besonders ungünstiger Platzierung eines Speicherobjektes können weitere Objekte unter Umständen nicht mehr platziert werden. Dieses Problem nennt man Fragmentierung, bei der zwischen externer und interner Fragmentierung unterschieden wird. Bei externer Fragmentierung kann freier Speicher nicht genutzt werden, weil eines oder mehrere Speicherobjekte so platziert sind, dass kein zusammenhängender, freier Speicherblock ausreichender Größe existiert. Als

interne Fragmentierung bezeichnet man freien Speicher, der in belegten Blöcken existiert. Diese Art von Fragmentierung tritt immer dann auf, wenn die Größe von Speicherblöcken nicht frei gewählt werden darf.

Der Designspielraum der Speichermanager umfasst die Grundfragen der Verwaltung benutzter Speicherblöcke, des freien Speichers und des Übergangs zwischen beidem.

Für benutzte Speicherblöcke muss entschieden werden, welche Größen sie annehmen dürfen, welche Informationen für die Blöcke abgelegt werden, wo die Blockinformationen abgelegt werden und wie auf die Möglichkeit der Veränderung der Blockgröße reagiert werden soll.

Für die freien Speicherbereiche muss neben verschiedenen Datenstrukturen zu ihrer Verwaltung auch die Möglichkeit erwogen werden, dass verschiedene freie Speicherbereiche für verschiedene Anfragen genutzt werden können. Man spricht dann von sogenannten Memory Pools.

Wird Speicher angefordert, muss entschieden werden, wo dieser Speicher angelegt werden soll. Dies kann nach verschiedenen Strategien wie Best-Fit, First-Fit, Next-Fit, Exact-Fit oder Worst-Fit erfolgen. Bei der Auswahl eines Platzes muss zudem jeweils berücksichtigt werden, ob eine minimale Größe für Speicherblöcke eingehalten werden soll und wann größere Speicherblöcke in kleinere Bereiche zerteilt werden dürfen. Verschiedene Designoptionen stehen auch beim Freigeben von Speicher zur Auswahl. Freigegebene Blöcke können miteinander zu größeren verschmolzen werden, dabei können eine oder mehrere maximale Blockgrößen zu beachten sein, die jeweils fest oder variabel sein können. Zudem stellt sich die Frage, wann freie Blöcke verschmolzen werden sollen; nie, manchmal oder immer. Die Verschmelzung kann sofort oder nach einer Verzögerung erfolgen.

Zwischen diesen Designoptionen gibt es zudem verschiedene Abhängigkeiten, die Details zu den Designmöglichkeiten sind in [AMC<sup>+</sup>04] zu finden.

## 2.3 Eingebettete Betriebssysteme

Die Software, die von eingebetteten Systemen ausgeführt wird, wird immer umfangreicher und immer komplexer. Um die Entwicklung zu vereinfachen und zu beschleunigen, werden immer häufiger eingebettete Betriebssysteme eingesetzt.

Während für Desktop-Betriebssysteme besonders eine ansprechende, einfach zu bedienende Benutzeroberfläche wichtig zu sein scheint, müssen eingebettete Betriebssysteme ganz andere Anforderungen [Mar03] erfüllen.

- **Konfigurierbarkeit:** Damit keine Hardwareressourcen verschwendet werden, sollten sich nicht benötigte Funktionen des eingebetteten Betriebssystems durch Konfiguration daraus entfernen lassen.

- **Treiberflexibilität:** Da die Hardware eingebetteter Systeme anwendungsspezifisch verschieden ist, liegt es nahe, die Treiber als spezielle Tasks flexibel zu implementieren, anstatt sie in den Betriebssystemkern zu integrieren.
- **Schutzmechanismen:** Eine hohe Ausführungsgeschwindigkeit ist bei vielen eingebetteten Systemen wichtiger, als es vor Eingriffen durch Software zu schützen. Statt dessen ist es sinnvoll, jedem Prozess direkten Hardwarezugriff zu gestatten.
- **Interrupthandling:** Aus Geschwindigkeitsgründen ist es sinnvoll, jedem Prozess die Handhabung von Interrupts, idealerweise unter Verwendung einer einfachen Programmierschnittstelle, zu ermöglichen.
- **Realzeitfähigkeit:** Viele eingebetteten Systeme sind Realzeitsysteme, das heißt, sie müssen bestimmte Aufgaben innerhalb eines vorgegebenen Zeitfensters lösen. Die Aufgaben müssen immer in dieser Zeit erfüllt werden, deshalb wird hier die Ausführungszeit im schlechtesten Fall (worst-case execution time, kurz WCET) betrachtet. Realzeitbetriebssysteme kommen dem entgegen, indem sie
  - für die Dauer von Betriebssystemaufrufen obere Schranken garantieren,
  - neben hochauflösenden Timern auch geeignete Scheduling-Verfahren[Mar03], die die Deadlines der verschiedenen Prozesse berücksichtigen, zur Verfügung stellen und
  - auf Geschwindigkeit optimiert sind, um Prozesse mit extrem kurzen Zeitfenstern zu unterstützen.

Ein Problem bei der Erstellung realzeitfähiger Systeme ist die sogenannte Prioritätsumkehr (Priority Inversion). Hierbei blockiert ein Prozess  $P_1$  niedriger Priorität einen Prozess  $P_2$  höherer Priorität, indem  $P_1$  eine Ressource belegt, die  $P_2$  benötigt. Erst wenn  $P_1$  die Ressource freigibt, kann  $P_2$  weiterarbeiten. Für den Prozess hoher Priorität wird es ohne den Einsatz spezieller Verfahren unmöglich, Laufzeitschranken einzuhalten.

Es ist eine große Anzahl verschiedener kommerzieller und frei verfügbarer Betriebssysteme für eingebettete Systeme erhältlich. Im Folgenden werden nun die drei Open-Source-Betriebssysteme  $\mu$ CLinux, eCos und RTEMS vorgestellt. Besonders die Real-Zeit-Fähigkeiten, die Speicherverwaltung, die Prozesserzeugung und die Scheduler sind dabei für die vorliegende Arbeit von speziellem Interesse.  $\mu$ CLinux ist ein eingebettetes Betriebssystem, während eCos und RTEMS als Realzeitbetriebssysteme gelten.

### 2.3.1 $\mu$ CLinux

$\mu$ CLinux[WW06] ist eine Linuxvariante für Mikroprozessoren ohne virtuelle Adressierung. Es kommt daher ohne Memory Management Unit (MMU) aus. Es wurde zuerst auf den Motorola MC68328 (DragonBall) portiert.  $\mu$ CLinux konnte so erstmals auf einem PalmPilot mit einem TRG SuperPilot Board [Han03] gestartet werden. Bei vielen Prozessoren für eingebettete Systeme wird auf die MMU verzichtet, um die Produktionskosten und die Verlustleistung des Systems zu reduzieren. Gewöhnliche Linux-Kernel benötigen jedoch eine MMU, um mit Hilfe des virtuellen Adressraumes positionsabhängige Maschinenprogramme zu starten, virtuellen Speicher zu verwenden und Speicherschutz zu realisieren. Bei eingebetteten Systemen fehlt zumeist die Festplatte und damit die Möglichkeit des Swap-Spaces, zudem werden sie gewöhnlich zur gleichen Zeit nur von einem einzelnen Benutzer verwendet. Daher kann durch sorgfältigen Anwendungsentwurf bei eingebetteten Systemen auf die Verwaltung virtuellen Speichers verzichtet werden. Das ist aber nicht ohne umfangreiche Änderungen am Kernel möglich [Eng01]. Entsprechende Änderungen an einem Linux Kernel 2.0 sind in  $\mu$ CLinux implementiert [Emb06] worden. Auch Kernel der Versionen 2.4 und 2.6 sind seither für  $\mu$ CLinux konvertiert worden.

Der Linux Kernel mußte umfassend angepasst werden, um ihn zu verkleinern und an die Umgebung ohne MMU anzupassen. Dabei konnten die wichtigsten Vorteile des Linuxsystems erhalten werden: Stabilität, umfassende Netzwerkfähigkeiten und weitreichende Dateisystemunterstützung.

Die Linux API konnte beibehalten werden, jedoch wurde die C-Bibliothek glibc durch die kleinere und an die veränderten Gegebenheiten angepasste  $\mu$ Clibc ersetzt.  $\mu$ Clibc ist konfigurierbar gestaltet, so dass man nicht benötigte Teile deaktivieren kann. Zudem fehlen manche Funktionen und andere ändern ihr Verhalten. Der  $\mu$ CLinux Kernel inklusive der grundlegenden Programme kommt mit einer Gesamtgröße von unter 900 kB aus. Der vollständige TCP/IP-Stack[KR02], die Netzwerkprotokolle und damit volle Internetfähigkeit blieben ebenso erhalten wie die Unterstützung für verschiedene Dateisysteme, darunter auch NFS, ext2 und FAT16/32. Aufgrund der fehlenden MMU musste das Multitasking eingeschränkt werden, statt `fork()` steht nun `vfork()` zur Verfügung. `vfork()` blockiert im Gegensatz zu `fork()` den aufrufenden Prozess, bis der neu erzeugte Task mittels `exec()` zum selbstständigen Prozess wird oder durch `exit()` beendet wird. Während ein Linux für ein System mit MMU den Speicher für den neu erzeugten Task erst bei Schreibzugriffen anlegt, wird auf Systemen ohne MMU der komplette Speicher sofort kopiert. Multitasking ist dennoch möglich, auch wenn jeglicher Speicherschutz fehlt. Systeme mit symmetrischen Multiprozessoren (SMP) werden ebenso wie durch Desktop-Linux Kernel unterstützt.  $\mu$ CLinux ist für mehrere Simulationsumgebungen verfügbar, unter anderem auch für MPARM. Dort wird jedoch lediglich ein Uniprozessorsys-

tem verwendet. Außer für ARM-Prozessoren ist  $\mu$ CLinux auch für Motorola ColdFire, Intel i960 und den Microblaze RISC soft-core für FPGAs erhältlich.  $\mu$ CLinux wird kommerziell zum Beispiel in der AXIS 2100 Network Camera [Axi02] und dem Aplio Voice-over-IP Telefon [Apl00] eingesetzt. Die Vorteile von  $\mu$ CLinux sind seine geringe Codegröße, die große Auswahl an möglichen Prozessoren und Plattformen, sowie seine freie Verfügbarkeit. Zu seinen Nachteilen gehört der fehlende Speicherschutz, das eingeschränkte Multitasking und seine mangelnde Realzeitfähigkeit, da es lediglich ein verkleinertes Desktop-Linux ist.

$\mu$ CLinux bietet die üblichen Linux-Scheduler an, unter anderem einen prioritätenbasierten *Round-Robin*-Scheduler, der mit einer konstanten Laufzeit arbeiten kann. Realzeittasks bekommen zwar höchste Priorität, dennoch bleibt ihr Timing das Ergebnis einer Best-Effort-Strategie. Es werden keine besonderen Schedulingverfahren für Prozesse mit Deadlines angeboten. Auch Verfahren, die das Problem der Prioritätsumkehr lösen, werden nicht angeboten.

Linux und auch  $\mu$ CLinux verwalten ihren Speicher seit Kernelversion 2.4 mit Hilfe eines sogenannten Slab-Allocators[Vah95, Bon94]. Der Speicher ist organisiert in Caches. Jeder Cache enthält ausschließlich Objekte eines Typs, zum Beispiel verschiedene Dateisystemcacheobjekte oder Speicherobjekte der Programme. Die Caches wiederum bestehen aus vielen Slabs, kleinen Speicherbereichen, die üblicherweise nur die Größe einer Speicherseite haben und immer zusammenhängend sind. Alle Slabs können mehrere initialisierte Objekte enthalten. Um die Speicherfragmentierung zu verringern, werden die Slabs in drei Gruppen unterteilt:

- vollständig mit Objekten belegte Slabs,
- teilweise von Speicherobjekten belegte Slabs und
- leere Slabs, die keine Objekte enthalten.

Wenn teilweise belegte Slabs existieren, werden diese, soweit möglich, für neue Speicheranforderungen benutzt. Existieren freie Objekte, so werden diese belegt bevor neue Slabs erzeugt werden. Die Suche nach einem geeigneten Platz für ein Objekt folgt dabei der Verkettung der Slabs. Die Erzeugung neuer Slabs ist durch den virtuellen Adressraum beschränkt, im Falle von  $\mu$ CLinux durch den physikalisch vorhandenen Speicher, da keine MMU vorhanden ist. Abbildung 2.3 zeigt den Aufbau der Slab-Strukturen.

### 2.3.2 eCos

Cygnus entwickelte eCos[Mas02], um eine kosteneffiziente, qualitativ hochwertige Betriebssystemlösung für den Markt der eingebetteten Systeme anzubieten. Ein weiteres, wichtiges Designkriterium war der Speicherbedarf.

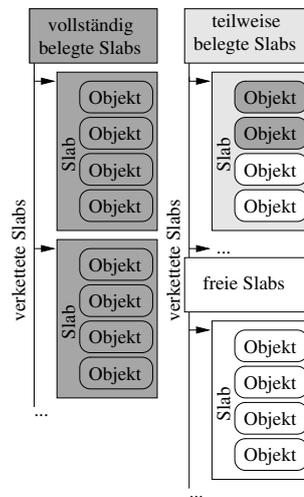


Abbildung 2.3: Speicherstruktur des Slab-Allocators

Um dennoch möglichst viele Bedürfnisse der Hersteller eingebetteter Systeme zu befriedigen, wurde eCos hochgradig konfigurierbar gestaltet. Unter Mitarbeit mehrerer Halbleiterhersteller wurde so ein Realzeitbetriebssystem geschaffen, das die Hardware durch eine Softwareschicht abstrahiert (hardware abstraction layer, HAL) und sich so leicht auf verschiedenste Hardware anpassen läßt. Da Software, die auf der HAL aufbaut, hardwareunabhängig funktioniert, muss lediglich die HAL angepaßt werden, um eCos auf neue Hardwareplattformen zu portieren.

Durch das hohe Maß an Konfigurierbarkeit läßt sich eCos von wenigen hundert Byte bis zu einigen hundert Kilobyte skalieren, wenn zusätzliche Dienste wie Web-Server hinzugefügt werden. eCos umfasst neben Interrupt- und Ausnahmenverwaltung, Fehlerbehandlung, Threadsynchronisation, Scheduling, Timern und Gerätetreibern, die alle standardmäßig von eingebetteten Betriebssystemen erwartet werden können, eine HAL, die der Software einen allgemeineren Zugriff auf die Hardware bietet. Eine ISO C- sowie mathematische Bibliotheken stehen ebenfalls zur Verfügung. Aus Geschwindigkeitsgründen laufen sowohl die Anwendungsprogramme als auch das Betriebssystem vollständig im Supervisor Modus des jeweiligen Prozessors. Damit haben auch Benutzerprogramme vollen Zugriff auf die Hardware. Neben ARM, Intel x86, MIPS, PowerPC und SPARC werden noch weitere Prozessorarchitekturen unterstützt. Einen Überblick über die eCos-Systemarchitektur bietet Abbildung 2.4, für die Details sei hier auf [Mas02] verwiesen.

Der eCos-Kernel ist der Betriebssystemkern von eCos. Er umfasst, natürlich voll konfigurierbar, die grundlegenden Funktionalitäten, die von einem Realzeitbetriebssystem erwartet werden. Dazu zählen Interrupt- und Excep-

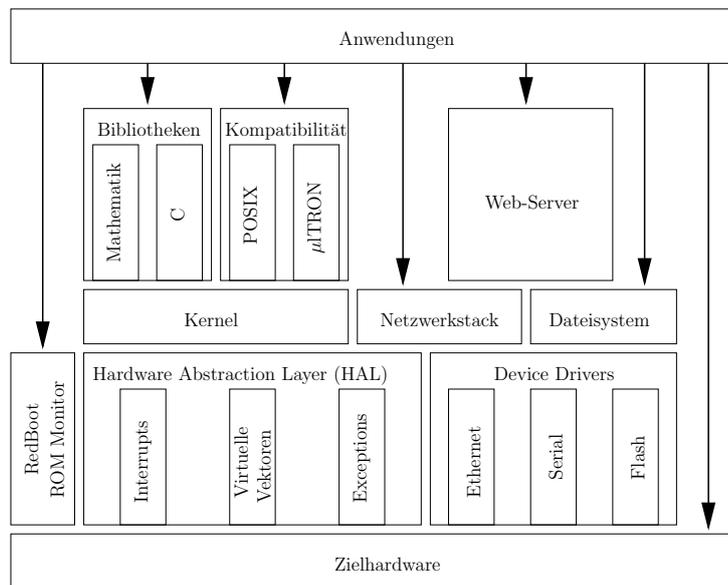


Abbildung 2.4: Beispielhafter Aufbau einer eCos-Applikation

tionverarbeitung, Scheduling, Threads und deren Synchronisation. Um die Realzeitfähigkeit des Systems zu erzielen, wurde an mehreren Stellen optimiert. Die benötigte Zeit, um auf einen Interrupt zu reagieren, wurde so gering wie möglich und außerdem absolut vorhersagbar gehalten. Gleiches gilt für die Zeit, die der Dispatcher benötigt, um einen Thread zur Ausführung zu bringen. Der Speicherbedarf für Code und Daten wurde minimal und ebenfalls vorhersagbar gehalten. Die dynamische Speicherallokation ist ebenfalls konfigurierbar, so dass das eingebettete System keinen Speicherüberläufen ausgesetzt wird. Die Ausführung aller Kernelaufufe verläuft absolut vorhersagbar, so dass ein auf eCos basierendes eingebettetes System Realzeitaufgaben gewachsen ist.

eCos stellt zwei stabile und einen experimentellen Scheduler zur Auswahl, von denen immer nur einer genutzt werden kann. Als stabil gelten der Multilevel-Queue-Scheduler und der Bitmap-Scheduler, der Lottery-Scheduler ist noch experimentell und soll hier nicht weiter behandelt werden. Während der Arbeit des gewählten Schedulers sind Interrupts zugelassen, um deren Reaktionszeit zu verringern. Als Realzeitbetriebssystem unterstützt eCos außerdem Priority Inheritance[Mar03] und Priority Ceiling, um das Problem der Prioritätsumkehr zu lösen.

Nur der Multilevel-Queue-Scheduler unterstützt symmetrisches Multiprocessing. Der Multilevel-Queue-Scheduler ist präemptiv zwischen Threads unterschiedlicher Priorität, die auch in unterschiedlichen Queues verwaltet werden. Zwischen Threads gleicher Priorität wird nach dem Zeitscheiben-

verfahren umgeschaltet. Die Threads werden in doppelt verketteten Listen gehalten, die als Queues verwendet werden.

Der Bitmap-Scheduler arbeitet mit maximal einem Thread je Priorität. Alle Prioritäten werden als Bit in einem Bitvektor verwaltet und es wird immer derjenige Thread ausgeführt, der zum einen in seinem Bit als ausführbereit markiert ist und der momentan die höchste Priorität aller ausführbereiten Threads hat. Da es höchstens einen Thread je Prioritätsniveau gibt, ist kein auf Zeitscheiben basierendes Verfahren nötig.

Bedauerlicherweise unterstützt eCos kein dynamisches Speichermanagement[GLL<sup>+</sup>03]. Die ISO C-Bibliothek ermöglicht dennoch bei passender Konfiguration in eingeschränktem Rahmen die Funktionalitäten von `malloc()` und den verwandten Funktionen. Zur Verwendung kommt dabei eine Konvertierung von Doug Leas Malloc (`dmalloc`)[Lea96], das auch in früheren Versionen der GNU C/C++ Bibliotheken zum Einsatz kam und sich besonders durch seine hohe Geschwindigkeit, gute Portabilität und geringen Bedarf an zusätzlichem Speicherplatz pro verwaltetem Speicherobjekt auszeichnet.

Die Erzeugung neuer Prozesse erfolgt mit Hilfe des Aufrufes von `cyg_thread_create()`. Ein Zeiger auf den Prozessstack und dessen Größe müssen dabei übergeben werden. Es kann leicht eine beliebige Anzahl Prozesse auf diese Weise initialisiert werden, die dann mit Hilfe von `cyg_thread_resume()` für ausführungsbereit erklärt werden können. Den eigentlichen Start des Prozesses übernimmt dann der Scheduler, der zu diesem Zweck seinerseits einmalig mittels `cyg_scheduler_start()` gestartet worden sein muss.

### 2.3.3 RTEMS

RTEMS[OAR03b] wurde von der On-Line Applications Research Corporation (OAR) kommerziell für das U.S. Army Missile Command entwickelt. Ziel war es dabei, ein portables, standardkonformes Realzeitbetriebssystem zu schaffen, dessen Quellcode öffentlich verfügbar ist und für das sämtliche anfallenden Lizenzgebühren bezahlt sind.

Unter anderem unterstützt RTEMS ARM, Intel x86, MIPS, Motorola MC68k, PowerPC und SPARC Prozessoren. Außer homogenen Multiprozessorsystemen sind dabei durch eine integrierte Message-Passing-Architektur zur Kommunikation und Synchronisation zwischen verschiedenen Prozessen auch heterogene Systeme möglich. RTEMS wurde für MPARM portiert, da MPARM mehrere, unterschiedlich getaktete Prozessorkerne und damit ein heterogenes Multitaskingsystem, simulieren kann.

RTEMS verfügt über ereignisbasiertes, präemptives, prioritätsbasiertes Scheduling und schnelles Interruptmanagement. Ein BSD-Sockets kompatibler TCP/IP-Stack gehört ebenso zum Betriebssystem wie die Unterstützung verschiedener Netzwerkprotokolle wie HTTP, FTP und NFS. Auch ei-

nige Dateisysteme, wie FAT, sind im Funktionsumfang des Betriebssystems enthalten. Die RTEMS-API umfasst dabei neben einer nativen API auch die Standards POSIX 1003.1b und uTRON 3.0. RTEMS ist verfügbar für ANSI/ISO C/C++ und Ada95. Neben GNU C/C++-Bibliotheken ist auch eine mathematische Bibliothek für RTEMS portiert worden. Speicherschutz ist auch in RTEMS nicht implementiert.

Wie eCos ist auch RTEMS hochgradig konfigurierbar, jedoch ist die Konfigurierbarkeit anders aufgebaut. Während eCos mittels einer großen Anzahl Präprozessoranweisungen, die in einer graphischen Oberfläche ausgewählt werden können, konfiguriert wird, müssen die Optionen bei RTEMS im Quelltext zum Beispiel durch Headerdateien gesetzt werden, bevor ein Konfigurationsheader eingefügt wird, der dann aus den gesetzten Definitionen eine gültige Konfiguration erzeugt. Über diese Schnittstelle lassen sich Optionen wählen, die dem Umfang nach eCos nahe kommen. Zusätzlich bietet RTEMS die Möglichkeit, Objekte[OAR03a] einer fest definierten Typenmenge zu verwenden, um das gewünschte Systemverhalten zu erzeugen. Diese Typen beinhalten Prozesse, Nachrichten zur Kommunikation zwischen Prozessen, Semaphoren, Timer und Zeitschranken für den Scheduler.

Alle erschaffenen Objekte haben neben einem Namen auch eine ID, die sich aus dem Objekttyp, dem Rechenknoten, auf dem sie erschaffen wurden und einem fortlaufenden Index besteht. Eine weitere Verwendung der Objekte ist, auf einfache Weise, ohne Änderung an den Quelldateien und ohne RTEMS neu zu kompilieren, in vielen Kernbereichen des Betriebssystems eigene Funktionen zu verankern. So kann man auf einfache Art Funktionen durch Timerobjekte zeitgesteuert auslösen lassen, Interruptroutinen plazieren oder, mit Hilfe des User-Extension-Objektes, Benutzerroutinen einfügen, die aufgerufen werden, wenn

- Prozesse erzeugt werden,
- Prozesse erstmals ausführungsbereit werden,
- Prozesse zum wiederholten Male ausführungsbereit werden,
- Prozesse gelöscht werden,
- Prozesswechsel durchgeführt werden,
- Prozessausführungen beginnen,
- Prozessausführungen enden oder
- fatale Fehler auftreten.

So können zum Beispiel einzelne Prozesse bei jedem Wiedereintritt in ihren Ausführungsbereich bestimmte Aktionen ausführen.

Auch die dynamische Speicherverwaltung in RTEMS ist konfigurierbar. Es stehen zwei Speichermanager zur Auswahl, der Partition Manager und der Region Manager.

Der Partition Manager optimiert vor allem die Zeit, die für die Speicherallokation benötigt wird. Dazu wird der verfügbare Arbeitsspeicher in eine oder mehrere Partitionen unterteilt, die wiederum in gleich große Blöcke partitioniert werden. Belegte Blöcke stehen vollständig der Applikation zur Verfügung, in leeren Blöcken werden je zwei Zeiger abgelegt, um die leeren Blöcke zu einer doppelt verketteten Liste zu verbinden. Blöcke werden vom Anfang der Liste aus alloziert, freigegebene Blöcke werden am Ende der Liste angefügt.

Der Region Manager verwaltet Regionen wählbarer Größe, in denen der Speicher in Vielfachen einer festen Seitengröße zugeteilt wird. Die Regionen sind organisiert in doppelt verketteten Listen von Blöcken variabler Größe, die aus einer oder mehreren Seiten bestehen. Speicheranforderungen werden nach einer First-Fit-Strategie erfüllt. Dabei wird auf ein Vielfaches der Seitengröße aufgerundet, und frei bleibende Seiten des Speicherbereiches werden abgespalten. Wird Speicher freigegeben, so werden benachbarte, freie Speicherbereiche nach Möglichkeit sofort zusammengefasst. RTEMS muss zu jedem Block zusätzliche Informationen, wie zum Beispiel dessen Größe, verwalten. Durch die Strategie des Aufteilens und Verschmelzens von Speicherbereichen werden stets Speicherblöcke minimaler, unter Beachtung der Seitengröße möglicher, Größe zugeteilt, was der Fragmentierung des Speichers entgegenwirkt. Dazu sind jedoch größere Verwaltungsstrukturen erforderlich, was den Aufwand der Speicherallokation erhöht.

Auch RTEMS lässt während der Arbeit des Schedulers Interrupts zu, um die Reaktionszeit auf Interruptanfragen zu verringern. Auch in RTEMS sind alle wichtigen Funktionen, wie Interrupts und Dispatcher mit vorher-sagbaren Laufzeiten implementiert. RTEMS stellt, wie bereits beschrieben, über das Objektsystem alle von Realzeitsystemen benötigten Funktionen zur Verfügung. RTEMS lässt es zu, Priority Inheritance und Priority Ceiling für jeden Prozeß einzeln zu konfigurieren, um das Problem der Prioritätsumkehr für genau die Prozesse zu lösen, bei denen aus der Prioritätsinversion kritische Fehler erwachsen können. Für alle übrigen Prozesse wird zusätzlicher Aufwand vermieden.

Voreingestellt bietet RTEMS einen prioritätsbasierten Round-Robin-Scheduler. Um minimale Laufzeit zu erreichen, werden alle ausführbereiten Prozesse einer Priorität in einer Warteschlange organisiert, so dass immer der erste Prozess als nächster ausgeführt wird. So sind minimale Kontextwechselzeiten garantiert, da nicht erst die Liste aller Prozesse durchlaufen werden muss, um den nächsten Prozess zu ermitteln. Prioritäten, Präemptionskontrolle, Zeitscheibenverfahren und manuelle Abgabe des Kontrollflusses sind unabhängig voneinander für jeden Prozess verwendbar.

Zur Unterstützung von Rate Monotonic Scheduling (RMS) können Prozesse dynamisch durch den Rate Monotonic Manager verzögert werden. Dazu muss der Rate Monotonic Manager mittels `rtems_rate_monotonic_create()` erzeugt werden. Starten kann man den Manager für eine anzugebende Laufzeit durch `rtems_rate_monotonic_period()`. Läuft der aufgerufene Manager bereits, so blockiert er bei seinem erneuten Start den Kontrollfluß für den Rest der Laufzeit. Danach gibt er den Kontrollfluß frei und wird im Anschluß neu gestartet. Läuft er noch nicht oder nicht mehr, so wird der Manager gestartet und der Kontrollfluß blockiert nicht. Wie in Abbildung 2.5 dargestellt, können so Intervalle definiert werden, in denen bestimmte Programmteile bis zu einer Deadline ausgeführt werden müssen.

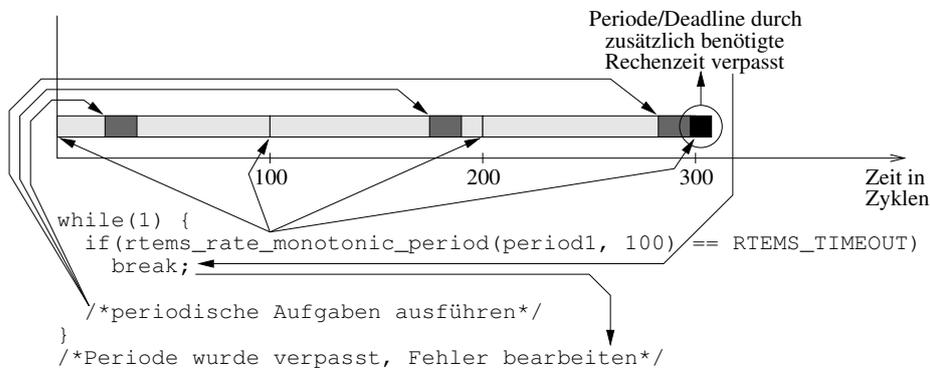


Abbildung 2.5: Beispiel für die Verwendung des Rate Monotonic Managers

Um Prozesse zu starten, muss ein Task Manager Objekt mittels `rtems_task_create()` erzeugt werden. Der zugehörige Prozess kann dann durch `rtems_task_start()` gestartet werden. Weitere Funktionen des Task Managers dienen unter anderem dazu, Prozesse zu pausieren, wieder aufzunehmen, ihre Priorität zu ändern oder sie für eine bestimmte Zeit zu verzögern.

## 2.4 Mathematische Konzepte

Im Folgenden sollen kurz die Konzepte des Integer Linear Programming und der Pareto-Optimalität vorgestellt werden, da sie im späteren Verlauf der Arbeit benutzt werden.

### 2.4.1 Integer Linear Programming

Als Integer Linear Programming (ILP) bezeichnet man die Optimierung einer linearen Zielfunktion über einer ganzzahligen Lösungsmenge, die durch

eine Nebenbedingungen aus linearen Gleichungen und Ungleichungen eingeschränkt ist. Dieses Verfahren wird häufig zur Lösung von Optimierungsproblemen eingesetzt, wenn keine spezialisierten Algorithmen bekannt sind. Zu diesem Zweck wird das zu optimierende Problem in Gleichungen transformiert, um es dann von einem ILP-Löser wie CPLEX[ILO06] oder `lp_solve`[LP06] lösen zu lassen. ILP-Gleichungen nehmen die Standardform

$$\sum_{i=1}^n c_i x_i = f(x) \quad (2.1)$$

$$\sum_{i=1}^n a_{i,j} x_i \leq b_j, \quad 1 \leq j \leq m \quad (2.2)$$

$$x_i \in \mathbb{Z}, \quad 1 \leq i \leq n$$

$$c_i \in \mathbb{R}, \quad 1 \leq i \leq n$$

$$a_{i,j} \in \mathbb{R}, \quad 1 \leq i \leq n, 1 \leq j \leq m$$

$$b_j \in \mathbb{R}, \quad 1 \leq j \leq m$$

an. Die Zielfunktion, Gleichung (2.1), soll entweder maximiert oder minimiert werden, wobei die Nebenbedingungen in Form der Gleichungen (2.2) zu erfüllen sind.

Wird der Wertebereich der  $x_i$  eingeschränkt, wird das Problem der Suche nach einer optimalen Lösung NP-hart[Weg99]. Dennoch sind ILP-Löser für die meisten Probleme in der Lage, in akzeptabler Zeit optimale Lösungen zu bestimmen, da viele Probleme sich als gutartig erweisen.

### 2.4.2 Pareto-Optimum

Bei Optimierungsproblemen mit mehreren Zielen, die nicht zu einem Kriterium zusammengeführt werden können, ist keine eindeutig beste Lösung definiert. Statt dessen läßt sich eine Menge Lösungen bestimmen, bei der eine Verbesserung eines Zieles nur durch eine Verschlechterung in einem anderen Kriterium erreicht werden kann. Diese Lösungsmenge der optimalen Kompromisse nennt man Pareto-Optimum. Die Lösungen lassen sich in einem  $n$ -dimensionalen Diagramm der  $n$  Optimierungskriterien als Punkte darstellen. Diese optimalen Punkte bilden eine  $(n - 1)$ -dimensionale Hyper-Grenzfläche.

## 2.5 Verwandte Arbeiten

In dieser Arbeit wird eine Speicherallokationsstrategie entwickelt, um zur Laufzeit das Scratchpad anhand eines Nutzenmaßes und einer Kostenfunktion für die Kopiertätigkeiten mit Funktionsblöcken und Daten zu belegen. Sowohl das Nutzenmaß als auch die Kostenfunktion können dabei beliebig

gewählt werden. In den folgenden Kapiteln wird lediglich eine Minimierung des Energiebedarfes angestrebt, aber auch eine Optimierung der Laufzeit oder des Hauptspeicherbedarfes sind denkbar. In diesem Abschnitt werden Arbeiten vorgestellt, die Grundlagen dieser Arbeit sind, ähnliche Ansätze oder ähnliche Ziele verfolgen.

Optimierungen des dynamischen Speichermanagers haben wie bereits in Abschnitt 2.2 beschrieben eine lange Geschichte. Daher sollen hier keine Optimierungen an den grundlegenden Strukturen der Speichermanager beschrieben werden, hierfür sei auf [WJNB95] verwiesen. Im Folgenden werden einige neue Ideen aufgezeigt, die Speichermanager für andere Zielsetzungen als hohe Allokationsgeschwindigkeit oder geringe Fragmentierung zu optimieren.

Zhao et al. [ZRW05] untersuchten, ob man mit Hilfe von verbesserten Speicherverwaltungsstrategien das Cache-Verhalten beeinflussen kann, um die Ausführungsgeschwindigkeit eines Systems zu verbessern. Sie verwendeten eine Allokationsstrategie, die auf Pools ähnlicher Speicherobjekte basiert. Die dadurch verbesserte zeitliche und örtliche Lokalität zusammengehöriger Speicherobjekte sowie den erleichterten Einsatz von Prefetching bewirken Geschwindigkeitsverbesserungen zwischen 13% und 17% auf x86-basierten Systemen.

Durch den Einsatz eines Speichermanagers, der an die auszuführende Applikation angepasst ist und auch den Scratchpad-Speicher verwendet, lassen sich nach Mamagkakis et al. [MAP<sup>+</sup>06] bis zu 82% der im Speicher verbrauchten Energie und bis zu 27% Laufzeit einsparen. Dabei erfolgt die Auswahl des Speichermanagers aus einer Menge Pareto-optimaler Lösungen. Die Lösungsmenge wird automatisch generiert, die letzte Auswahl eines Speichermanagers erfolgt jedoch nach subjektiven Kriterien manuell.

Seit einiger Zeit sind Arbeitsspeicherbausteine verfügbar, die ebenso wie mobile Prozessoren mit verschiedenen Energiesparmodi ausgestattet sind. Ein Speicherbaustein im Standby-Zustand braucht weniger Energie als einer im aktiven Zustand. Lebeck et al. [LFZE00] haben untersucht, wie mit Hilfe von Softwareunterstützung durch den Speichermanager die bisherigen Hardwareverfahren zur Nutzung dieser Energiesparfunktion verbessert werden können. Für ihr kombiniertes Kostenmaß aus Verzögerung und Energiebedarf erreichten die Autoren mit Hilfe optimierter Seitenallokation in Verbindung mit dynamisch wechselnden Energiesparmodi bis zu 55% Energieersparnis gegenüber einem reinen Hardwareansatz.

Seit einigen Jahren sind Scratchpad-Speicher Gegenstand der Forschung. Es wurde versucht, die Hardware der Scratchpad-Speicher an verschiedene Applikationen anzupassen. Andererseits wurden auch verschiedene Verfahren entwickelt, die Software so zu verändern, dass sie Scratchpad-Speicher benutzt. Die Software-Lösungen lassen sich grob in zwei Ansätze unterteilen. Zum einen gibt es statische Scratchpad-Allokationstechniken, bei denen sich der Inhalt des Scratchpad-Speichers während der Programmausführung

nicht mehr verändert. Zum zweiten gibt es dynamische Ansätze, die den Inhalt des Scratchpads während der Programmausführung verändern. Grundsätzlich kann außerdem zwischen Belegungstechniken unterschieden werden, die zur Design-/Compilezeit angewandt werden, und solchen, die zur Laufzeit entscheiden, was im Scratchpad-Speicher sein soll.

Liegt die auszuführende Applikation lediglich als Maschinenprogramm vor und darf aus rechtlichen Gründen nicht verändert werden, so ist eine Anpassung der Software an die Hardware nicht möglich. Ein Ausweg ist es dann, den Scratchpad-Speicher, wie von Angiolini et al. [ABC03, ABC05] beschrieben, an die Applikation anzupassen. Um nicht-kontinuierliche Teile der Anwendung aus dem Scratchpad-Speicher zu nutzen, muss der Adressbereich des Scratchpad-Speichers in mehrere Teile partitioniert werden, die genau dort in den Adressraum eingeblendet werden, wo sie bei der speziellen Applikation den größten Gewinn erzielen. Die Aufteilung des Scratchpad-Speichers auf die verschiedenen Programmteile erfolgt mit Hilfe eines Algorithmus, der die Technik der dynamischen Programmierung anwendet. Dies ist möglich, da die Problemstellung nicht NP-hart ist, weil die Objekte beliebig unterteilbar sind.

Eine Möglichkeit, den Scratchpad-Speicher statisch zur Compilezeit zu belegen, stellten Steinke et al. [SZWM01] vor. Ihr Verfahren bestimmt zunächst die Anzahl der Zugriffe auf die einzelnen Variablen, Funktionen und Basisblöcke. So kann die potentielle Energieersparnis bei Verschiebung eines Objektes in den Scratchpad-Speicher ermittelt werden. Bei der Verschiebung von Basisblöcken werden zusätzlich nötig werdende Sprungbefehle ebenfalls beachtet. Im nächsten Schritt werden mit Hilfe eines Branch-and-Bound-Verfahrens diejenigen Speicherobjekte ausgewählt, die in den Scratchpad-Speicher verschoben werden sollen. Da für diese Speicherobjekte keine weiteren Unterteilungen betrachtet werden, ist die Problemstellung grundsätzlich NP-hart. Für reale Eingaben findet das Verfahren dennoch Lösungen, so dass im letzten Schritt die ausgewählten Variablen, Funktionen und Basisblöcke in den Scratchpad-Speicher verschoben werden und die nötigen zusätzlichen Sprungbefehle eingefügt werden können. Die vorgestellte Scratchpad-Allokationstechnik wurde in einen Compiler integriert. Die erzeugten Ergebnisse verringern den Energieverbrauch des betrachteten Systems um bis zu 78%.

Einen Vergleich der Scratchpad-Speicher mit Caches im Hinblick auf Chipfläche, Energiebedarf und Systemgeschwindigkeit präsentieren Banakar et al. [BSL<sup>+</sup>01]. Die statische Belegung des Scratchpad-Speichers mit Variablen, Funktionen und Basisblöcken eines Prozesses durch den Compiler erfolgt durch ILP-Gleichungen zur Compilezeit. Da für diese Speicherobjekte ebenfalls keine beliebigen Unterteilungen zugelassen sind, ist das Problem, welche Objekte in den Scratchpad-Speicher verschoben werden sollen, NP-hart. Im direkten Vergleich mit einem gleich großen, zweifach assoziativen Cache verbessert sich die Laufzeit durch die Verwendung des Scratchpad-

Speichers um 16%. Dabei ist die Fläche, die der Scratchpad-Speicher im Chip belegt bei gleicher Beschleunigung um 34% kleiner. Der Energieverbrauch des Scratchpad-Speichers ist dabei zwischen 60% und 82% geringer als der eines Caches. So lassen sich nach Steinke et al. [SWLM02], die den gleichen Ansatz verwenden, durchschnittlich 23% des Energieverbrauchs des Gesamtsystems einsparen.

Wehmeyer et al. [WHM04] erweiterten diesen Ansatz der Belegung des Scratchpad-Speichers durch den Compiler mit Hilfe eines ILPs für den Fall mehrerer oder partitionierter Scratchpads. Gegenüber der ursprünglichen Variante konnte der Energiebedarf noch einmal um 22% reduziert werden.

Wilmer [Wil05] kombinierte Steinkes Ansatz mit Loop-Tiling, um Teile von Arrays im Scratchpad-Speicher zu halten. Hierbei ist dynamisches Umkopieren erforderlich. Die Belegung des Scratchpad-Speichers wird hier durch einen genetischen Algorithmus erzeugt.

In [VSM03] betrachten Verma et al. eine andere Erweiterung des Ansatzes nach Steinke. Zusätzlich zu Basisblöcken, Funktionen und globalen Variablen wird der Scratchpad-Speicher nun auch mit Teilen von Arrays belegt. Hierzu wird das zu optimierende Programm modifiziert und es müssen alle möglichen Aufteilungspunkte betrachtet werden. Auch hier werden ILP-Gleichungen eingesetzt, um das Problem der Aufteilung zusammen mit dem der Scratchpad-Speicherbelegung zu lösen. So werden Verbesserungen zwischen 5% und 17% gegenüber dem ursprünglichen Ansatz möglich.

Eine andere Idee verfolgen Verma et al. mit [VWM04a]. Wenn Caches und Scratchpad-Speicher in einem System vorhanden sind, lässt sich der Scratchpad-Speicher benutzen, um die Anzahl der auftretenden Cache-Misses zu reduzieren. Hierzu wird anhand von Traces ein Konfliktgraph aufgebaut. Die Speicherobjekte bilden dabei die Knoten und werden durch Kanten, die die Cache-Konflikte darstellen, verbunden. Das Kantengewicht entspricht der Anzahl der durch diesen Cache-Konflikt verursachten Cache-Misses. Auf diesem Konfliktgraphen wird nun die Anzahl der Cache-Misses mit Hilfe von ILP-Gleichungen minimiert, indem möglichst viele Konflikte durch Verschiebung eines der beiden Speicherobjekte in den Scratchpad-Speicher behoben werden. Obwohl weniger Speicherzugriffe in den Scratchpad-Speicher umgeleitet werden als bei Steinkes Ansatz, ist dennoch eine Verbesserung des Energieverbrauches von 8% bis 29% möglich, da die Anzahl der Instruktionscache Misses, die besonders energieaufwändig sind, minimiert wird.

Petzold [Pet04] überträgt Steinkes Ansatz auf Multiprozesssysteme, weiterhin erfolgt die Belegung des Scratchpads zur Compilezeit. Die Menge der zu optimierenden Prozesse muss deshalb bekannt und fest sein. Petzold präsentiert drei Ansätze, mit deren Hilfe der Scratchpad-Speicher zwischen den Prozessen aufgeteilt werden kann. Der statische Ansatz, kurz SAMP genannt, partitioniert den Scratchpad-Speicher. Jeder Prozess erhält einen Teil des Scratchpad-Speichers mit fester Größe. Für die optimale Aufteilung des

Scratchpad-Speichers auf die Prozesse werden eine Lösung mit Hilfe von ILP-Gleichungen und ein algorithmisches Verfahren präsentiert. Der dynamische Ansatz (DAMP) stellt jedem der laufenden Prozesse jeweils das gesamte Scratchpad zur Verfügung. Eine Erweiterung des Dispatchers sorgt dafür, dass bei jedem Prozesswechsel die Daten des vorherigen Prozesses aus dem Scratchpad-Speicher zurück in den Hauptspeicher geschrieben und die Daten und Instruktionen des neuen Prozesses in den Scratchpad-Speicher kopiert werden. Für jedes Objekt im Scratchpad entstehen dadurch Kopierkosten, die bei der Entscheidung, mit welchen Objekten der Scratchpad-Speicher belegt werden soll, berücksichtigt werden müssen. Auch für dieses Belegungsproblem wird eine Formulierung in ILP-Gleichungen und in algorithmischer Form angegeben. Der dritte von Petzold vorgestellte Ansatz kann als eine Kombination der beiden Vorhergehenden verstanden werden. Der sogenannte hybride Ansatz (HAMP) teilt das Scratchpad in zwei Teile. Ein Teil wird dynamisch durch alle Prozesse benutzt, was den bereits erwähnten Kopieraufwand verursacht. Der andere Teil des Scratchpads wird statisch auf die Prozesse aufgeteilt. Dabei umfasst der hybride Ansatz im Extremfall sowohl den statischen als auch den dynamischen Ansatz, wenn der entsprechende dynamische beziehungsweise statische Teil des Scratchpads nicht genutzt wird. Auch für die hybride Allokationsstrategie wird neben einer Lösung durch ILP-Gleichungen ein algorithmischer Ansatz präsentiert. Während der statische Ansatz vor allem für große Scratchpad-Speicher geeignet und der dynamische besonders für kleine Scratchpads erfolgreich ist, ist die hybride Lösung für alle Scratchpad-Größen zu empfehlen, da sie die beiden anderen umfasst. Gegenüber einer Zuweisung des Scratchpads an den Prozess mit der größten Energieersparnis und einer Bestimmung der Belegung durch die Allokationstechnik nach Steinke lassen sich weitere Einsparungen von bis zu 37% erzielen. Petzolds Zielsetzung ist mit der, der vorliegenden Arbeit nahezu identisch, jedoch soll die Belegung bei dieser Arbeit zur Laufzeit generiert werden, um beliebige Zusammenstellungen auch zur Designzeit des Systems unbekannter Programme zu unterstützen.

Steinke et al. [SGW<sup>+</sup>02] betrachteten ebenfalls ein dynamisches Verfahren, um den Scratchpad-Speicher mit Instruktionen zu belegen. Eine Analyse des Ausgangsprogramms liefert neben den verschiebbaren Funktionen und Basisblöcken auch deren jeweilige Größe und die Anzahl der Zugriffe darauf. Nach einer Bestimmung der möglichen Punkte für das Umkopieren der Speicherobjekte lassen sich aus den erwarteten Energieeinsparungen und den Kopierkosten abermals ILP-Gleichungen machen, so dass ein ILP-Löser eine optimale, dynamische Belegung bestimmen kann. Gegenüber dem statischen Ansatz von Steinke erreicht dieses dynamische Verfahren eine Verringerung des Energiebedarfs von bis zu 38%, gegenüber einem Cache von 29%.

Einen vollständig dynamischen Ansatz der Scratchpad-Nutzung zur Compilezeit verfolgen Verma et al. in [VWM04b]. Dazu werden in einem ersten Schritt alle möglichen Speicherobjekte bestimmt. Diese umfassen skalare und

nicht-skalare globale Variablen, nicht-skalare lokale Variablen und Traces. Traces sind häufig ausgeführte Codesegmente, die, neben weiteren Eigenschaften, mit unbedingten Sprüngen enden und als atomare Einheit von Befehlen an beliebiger Stelle im Speicher platziert werden können, ohne dass Modifikationen an anderen Traces nötig sind. Für all diese Speicherobjekte wird im zweiten Schritt eine Lebendigkeitsanalyse anhand des Kontrollflußgraphen durchgeführt, um herauszufinden, welche Speicherobjekte wann benutzt werden. Damit läßt sich im dritten Schritt, vergleichbar mit der Registerallokation für CISC Architekturen, eine Belegung für das Scratchpad ermitteln. Dazu werden ILP-Gleichungen aufgestellt, deren Lösung von einem ILP-Löser bestimmt wird. Im letzten Schritt erfolgt die Positionierung der Objekte an Adressen im Scratchpad durch weitere ILP-Gleichungen, die getrennt von denen zur Erstellung der Belegung gelöst werden können. Gegenüber der ursprünglichen Scratchpad-Allokationstechnik von Steinke [SWLM02] lassen sich im Durchschnitt 34% Energie einsparen.

Einen weiteren Ansatz verfolgen Poletti et al. [PMA<sup>+</sup>04]. Die Belegung des Scratchpad-Speichers erfolgt zur Laufzeit, allerdings müssen die Speicherobjekte, die im Scratchpad-Speicher abgelegt werden sollen, um Energie zu sparen, vom Programmierer zur Compilezeit explizit angelegt werden. Hierzu steht eine Programmierschnittstelle für den vorgestellten Scratchpad-Speichermanager zur Verfügung. Zur Laufzeit entscheidet der Scratchpad-Speichermanager, der eine Erweiterung des in Abschnitt 2.3.3 auf Seite 20 vorgestellten Speichermanagers von RTEMS ist, lediglich anhand des freien Scratchpad-Speichers wo ein Speicherobjekt angelegt wird. Zudem wird eine DMA-Einheit eingesetzt, um die Kopieraufgaben zu übernehmen. Der Ansatz von Poletti ist mit dem der vorliegenden Arbeit eng verwandt, jedoch soll hier die Belegung vollständig zur Laufzeit und automatisiert generiert werden, um mehreren Applikationen ohne Wissen über die anderen Zugriff auf das Scratchpad zu ermöglichen.



## Kapitel 3

# Scratchpad-Allokation für mehrere Prozesse

Im Folgenden sollen die verschiedenen, im Rahmen dieser Arbeit entwickelten, Scratchpad-Allokationstechniken beschrieben werden. In Abschnitt 3.1 werden zunächst die grundlegenden Konzepte der Verfahren vorgestellt. Dabei ergeben sich zwei Strategien, die die Korrektheit der Applikationen erhalten. Die Erste basiert auf dem Blockieren der benutzten Speicherobjekte an ihrer momentanen Position und wird in Abschnitt 3.2 zusammen mit einer Auswahl möglicher Heuristiken beschrieben. Die andere Strategie und entsprechende Heuristiken werden in Abschnitt 3.3 dargestellt. Sie verschiebt auch benutzte Objekte im Speicher und replaziert sie an ihre ursprüngliche Adresse, sobald es nötig ist. Das Kapitel endet mit einer Betrachtung der vorgestellten Allokationstechniken in Abschnitt 3.4.

### 3.1 Vorbetrachtungen

Um den Energiebedarf von Multiprozesssystemen zu verbessern, kann man den Scratchpad-Speicher einem der Prozesse zuweisen. Dessen Energiebedarf wird dadurch verringert und auch die Summe aller Prozesse wird weniger Energie verbrauchen. Wie in Abschnitt 2.5 beschrieben, hat Petzold [Pet04] gezeigt, dass es möglich ist, mehr Energie in Multiprozesssystemen zu sparen, wenn der Scratchpad-Speicher von mehr als einem Prozess benutzt wird. Hier werden nun Laufzeitansätze vorgestellt, die nicht nur mit zur Compilzeit unbekanntem Programmen arbeiten können, sondern von denen auch alle, bis auf eine Ausnahme, fähig sind, dynamisch im Heap-Speicher angelegte Speicherobjekte zu optimieren.

Eine Verteilung des Scratchpad-Speichers durch die Applikationen selber ist nicht sinnvoll, da sie so nicht mehr voneinander unabhängig sind. Alle Prozesse sind dem Betriebssystem bekannt, daher ist es naheliegend, das Betriebssystem um einen Scratchpad Memory Manager (kurz SPMM)

zu erweitern, der den Scratchpad-Speicher zuteilt und verwaltet. Jeder Prozess, der Scratchpad-Speicher benutzen möchte, muss diesen beim SPMM anfordern.

Wenn ein Prozess dynamischen Speicher beim SPMM reserviert, so ist das erzeugte Speicherobjekt dem SPMM bekannt. Zusätzlich sollen jedoch auch statische Speicherobjekte, globale Variablen und Funktionen in den Scratchpad-Speicher kopiert werden. Der SPMM benötigt deswegen auch Informationen über die statischen Objekte. Deshalb werden beim Erzeugen eines neuen Prozesses auch dessen statische Speicherobjekte durch eine Betriebssystemerweiterung dem SPMM bekannt gemacht. Bei jedem Wechsel in den Kontext eines Prozesses wird versucht, dessen Speicherobjekte durch Verschiebung in den Scratchpad-Speicher zu optimieren. Hierzu werden Scheduler oder Dispatcher um einen Aufruf des SPMM erweitert. Jeder Kontextwechsel eines Prozesses sorgt so dafür, dass die Allokationsstrategie aufgerufen wird, um das Scratchpad nach Möglichkeit mit den Speicherobjekten des Prozesses zu belegen. Ändern sich die Speicherobjekte eines Prozesses durch dynamische Speicheranforderungen oder Freigaben, so wird die Allokationsstrategie ebenfalls aufgerufen um den Energieverbrauch in der veränderten Situation zu verringern.

Jedes vom SPMM verwaltete Speicherobjekt muss, wie beschrieben, angelegt werden. Dabei teilt der anfordernde Prozess dem SPMM ein Nutzenmaß mit, anhand dessen der SPMM entscheiden kann, welche Speicherobjekte tatsächlich im Scratchpad-Speicher sein sollen, um möglichst viel Energie zu sparen. Dieses Nutzenmaß ist beliebig wählbar, solange ein korrespondierendes Kostenmaß verwendet wird, mit dessen Hilfe ein Vergleich zwischen dem möglichen Nutzen eines Speicherobjektes und dessen Kosten für Kopiervorgänge, sowohl zwischen Hauptspeicher und Scratchpad-Speicher, als auch in der entgegengesetzten Richtung, möglich ist. Zudem werden, abhängig von der gewählten Belegungsstrategie, bei jedem Wechsel in den Kontext des Prozesses dessen Objekte in den Scratchpad-Speicher kopiert, daher treten die Kopierkosten im schlechtesten Fall bei jedem Kontextwechsel in diesen Prozess wieder auf. Um diesen Kopieraufwand lediglich für Speicherobjekte zu betreiben, bei denen dennoch Gewinn erzielt wird, ist es sinnvoll, ein Nutzenmaß je Zeitscheibe des Prozesses zu definieren und zu verwenden. Objekte, für die sich der Kopieraufwand nicht lohnt, werden so ausgeschlossen.

Hauptspeicher wird als nicht knappe Ressource angenommen, alle Speicherobjekte werden lediglich in den Scratchpad-Speicher kopiert. Im Hauptspeicher verbleiben die Originale. Dorthin können veränderte Objekte bei Verdrängung aus dem Scratchpad-Speicher zurückgeschrieben werden. Da Instruktionsobjekte nicht verändert werden dürfen, müssen sie im Gegensatz zu Datenobjekten auch nicht zurückgeschrieben werden. Im Folgenden wird häufig von der Verschiebung von Speicherobjekten gesprochen. Damit ist oben beschriebenes Verhalten gemeint.

Wenn ein Speicherobjekt zur Laufzeit des Programmes verschoben werden soll, so müssen alle Zeiger des Programmes auf dieses Speicherobjekt angepasst werden. Es gibt in C/C++ keine Buchführung über die vorhandenen Zeiger auf ein Speicherobjekt, daher ist es zur Laufzeit nicht möglich, alle Zeiger auf ein Objekt zu finden. Um dennoch Speicherobjekte verschieben zu können, wird eine zweite Dereferenzierungsstufe eingefügt. Ein Programm darf per Definition keine Zeiger mehr auf seine Speicherobjekte verwenden, wenn diese durch den SPMM verwaltet werden. Statt dessen verwenden Programme Zeiger auf die SPMM-Objekte, um von dort Zeiger auf das eigentliche Speicherobjekt zu beziehen. So ist es möglich, die Speicherobjekte zu bewegen, während die Programme mit unveränderten Referenzen auf die SPMM-Objekte arbeiten können. Dies ist in Abbildung 3.1 dargestellt. Die Verschiebung von Speicherobjekten äußert sich in der Veränderung des Zeigers des SPMM-Objekts auf das eigentliche Speicherobjekt.

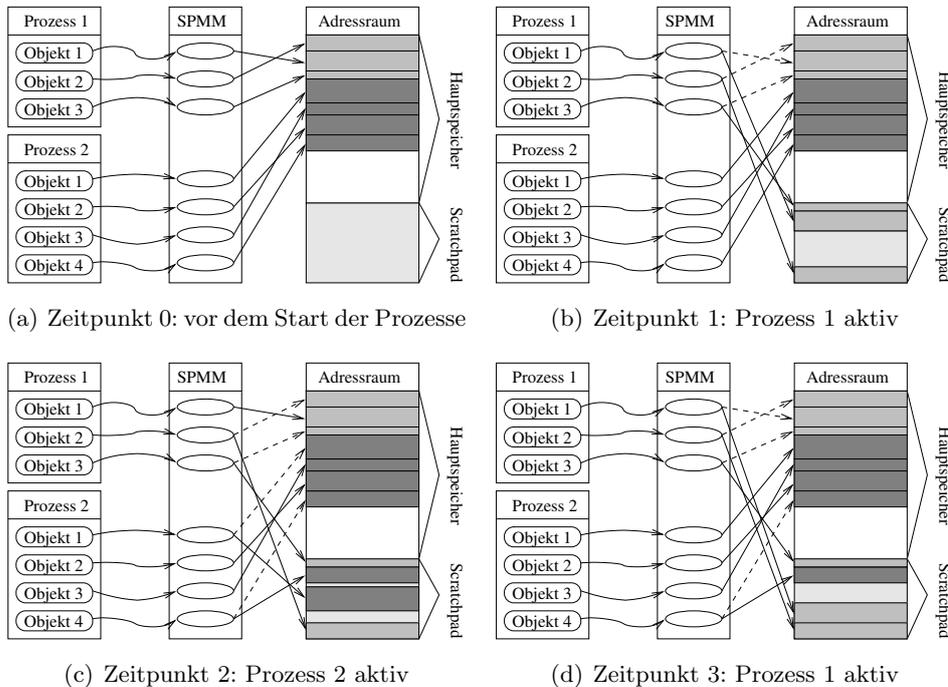


Abbildung 3.1: Beispiel für die zusätzliche Dereferenzierungsstufe

Programme speichern oft Zeiger auf Speicherobjekte in Registern zwischen. Wird das Speicherobjekt, dessen Adresse zwischengespeichert wurde, verschoben, so verfehlen alle weiteren Zugriffe, die den Registerinhalt als Adresse benutzen, das Speicherobjekt und greifen auf beliebige andere Speicherobjekte zu. Abbildung 3.2 visualisiert die beschriebene Problemstellung.

Das Programm wird in diesem Fall nicht mehr korrekt ausgeführt. Ein Ausweg ist es, alle Zugriffe auf optimierte Speicherobjekte indirekt über die

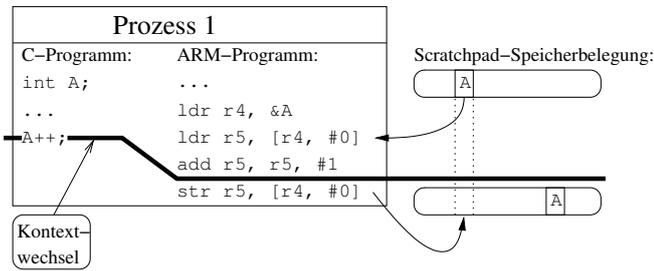


Abbildung 3.2: Zugriffsfehler bei Benutzung gepufferter Adressen

festen Adresse des SPMM-Objektes aus dem Speicher zu adressieren. Zum einen ist diese Adressierungsart extrem ineffizient, zum anderen unterstützt der ARM7-Prozessor sie nicht. Ein anderer Ausweg ist es, SPMM-Objekte als benutzt zu markieren, bevor ein Prozess die Adresse seines Speicherobjektes von einem SPMM-Objekt erhält. Die Scratchpad-Allokationsstrategie sorgt dann dafür, dass das Speicherobjekt für diesen Prozess an genau dieser Stelle zu finden bleibt. Werden die Referenzen auf das Speicherobjekt gelöscht, so darf das SPMM-Objekt wieder als unbenutzt gekennzeichnet und unbeschränkt bewegt werden.

Um Speicherobjekte im Scratchpad-Speicher platzieren zu können, muss dem SPMM bekannt sein, wo freie Bereiche im Scratchpad-Speicher zu finden sind. Da Speicherobjekte eines Prozesses durch die eines anderen verdrängt werden können, müssen die belegten Bereiche des Scratchpad-Speichers einem SPMM-Objekt zugeordnet werden können. Es ist also eine Buchführung über die freien und die belegten Bereiche des Scratchpad-Speichers nötig.

Eingebettete Betriebssysteme benutzen ebenso wie Desktop-Betriebssysteme üblicherweise präemptives Multitasking. Unterbrechungen eines Prozesses können dadurch und durch Interrupts zu jeder Zeit auftreten. Wird so die Arbeit des SPMM unterbrochen, können sich die Buchführungsstrukturen in einem inkonsistenten Zustand befinden, was zu undefiniertem Verhalten führen kann. Um das zu verhindern, können alle Interrupts deaktiviert werden. Dafür muss jedoch die maximale Laufzeit des SPMM auf alle Taskwechsel- und Interruptlaufzeiten addiert werden. Alternativ ist auch der Einsatz von Semaphoren möglich, jedoch ist selbst mit Verfahren wie Priority Ceiling immer noch die maximale Laufzeit des SPMM auf alle Taskwechsel- und Interruptlaufzeiten aufzuschlagen, da erst nach Abschluss des SPMM-Aufrufes die Semaphore freigegeben werden kann. Beide beschriebenen Möglichkeiten, die Datenstrukturen des SPMM zu schützen, werden in Abbildung 3.3 verdeutlicht. Eine absolut konstant begrenzte Laufzeit des SPMM ist daher absolut notwendig, um die Vorhersagbarkeit des Zeitverhaltens des Systems zu erhalten.

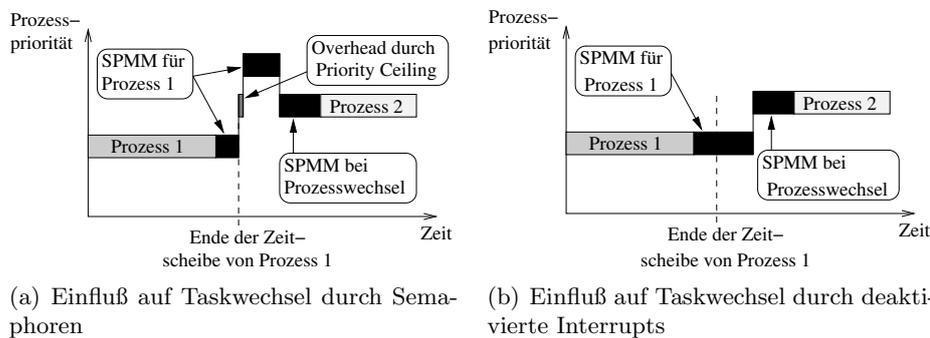


Abbildung 3.3: Gegenüberstellung beider Möglichkeiten zum Schutz der Datenstrukturen des SPMM

Der SPMM muss bei jedem Belegungsschritt zwei Probleme lösen. Zum einen entscheidet er, welche der Speicherobjekte im Scratchpad abgelegt werden sollen. Jedes Speicherobjekt verspricht dabei einen Nutzen in Form einer Energieersparnis und erzeugt durch seine Größe Kosten. Dies läßt sich als Knapsack-Problem[Weg03] auffassen. Die Speicherobjekte entsprechen den Gegenständen und der Scratchpad-Speicher dem Rucksack. Damit ist das Problem, eine Belegung zu generieren eine Variante des Knapsack-Problems und ebenso NP-hart.

Zum anderen muss der SPMM entscheiden, an welchen Adressen im Scratchpad-Speicher die Objekte abzulegen sind. Diese Aufgabe entspricht exakt dem Problem der Dynamic Storage Allocation (DSA), das ebenfalls als NP-hart nachgewiesen wurde[GJ79].

### 3.2 Belegung mit Blockieren der Objekte

Wie im vorangehenden Abschnitt beschrieben, ist es notwendig, dass die Adressen benutzter Speicherobjekte eines Prozesses unverändert sind, wenn der Prozess aktiv ist. Die einfachste Lösung ist es daher, als benutzt markierte Objekte weder in den Scratchpad-Speicher zu verlagern noch hinaus zu bewegen. Diese Objekte gelten dann als blockiert und dürfen nicht mehr bewegt werden.

Durch die blockierten Objekte ändert sich die Aufgabenstellung des SPMM. Die Belegung zu einem Aufrufzeitpunkt lässt sich nicht mehr als Knapsack-Problem formulieren. Die blockierten Objekte unterteilen den Scratchpad-Speicher in mehrere Bereiche, in denen Objekte abgelegt werden können. Jeder dieser Bereiche kann nun als eigener Rucksack betrachtet werden. Die Problemstellung entspricht damit einem Multiple Knapsack-Problem[KBH94], das ebenso NP-hart ist. Erschwerend kommt hinzu, dass die Speicherobjekte in verschiedenen Bereichen unterschiedlichen Nutzen erzielen, denn jede Verdrängung kann zusätzliche Kosten durch das Kopieren

des Objektes in den Hauptspeicher verursachen. Diese Kosten lassen sich als verminderter Nutzen modellieren. Für das Knapsack-Problem sind gute und gleichzeitig schnelle Heuristiken bekannt, gerade für das Multiple Knapsack-Problem sind jedoch nur vergleichsweise langsame Approximationen verfügbar.

Nach getroffener Belegungsentscheidung müssen die Speicherobjekte platziert werden. Wie beschrieben, verursachen Speicherobjekte innerhalb eines als verfügbar betrachteten Speicherbereiches weitere Kosten, da nicht blockierte Objekte in Teilen des Zielbereiches liegen können und in anderen nicht. Dahingehend muss auch das in Abschnitt 2.2 ab Seite 11 beschriebene DSA-Problem erweitert werden, es bleibt jedoch NP-hart, da DSA weiterhin als Sonderfall bei vollständig freiem Bereich auftritt.

Die Vorhersagbarkeit der Scratchpad-Belegung wird durch das Blockieren von Objekten für jeden einzelnen Prozess ohne Wissen über den Zustand der anderen Prozesse unmöglich. Der Scratchpad-Speicher kann zu jedem Zeitpunkt vollständig durch blockierte Objekte anderer Prozesse belegt sein. Aussagen über das Zeitverhalten des betrachteten Prozesses werden so, wie bei Caches auch, nur im durchschnittlichen, nicht aber im für Realzeitsysteme wichtigen, schlechtesten Fall positiv beeinflusst. Ein Ausweg ist es, Realzeitprozessen an dieser Stelle zu ermöglichen, blockierte Objekte kurzzeitig auszulagern. Der Realzeitprozess verschiebt dazu den Inhalt eines beliebigen Teils des Scratchpad-Speichers in den Hauptspeicher und belegt diesen Teil dann mit eigenen Objekten. Bevor andere Prozesse wieder aktiv werden können, müssen die ausgelagerten Daten wieder zurück an ihre ursprüngliche Position verschoben werden. So läßt sich für die benötigten Objekte eine Vorhersagbarkeit herstellen. Dieser Ansatz läßt sich sowohl in den Echtzeitapplikationen als auch im SPMM verankern. Im letzteren Fall entsteht im SPMM ein Hybrid zwischen blockierender und replatzierender Belegungsstrategie, die in Abschnitt 3.3 auf Seite 48 beschrieben wird.

Im Folgenden werden verschiedene Methoden vorgestellt, die die Auswahl der Speicherobjekte und deren Platzierung im Scratchpad-Speicher übernehmen.

### 3.2.1 Optimale Lösung mittels ILP

Aufgrund der NP-harten Problemstellungen des SPMM ist nicht zu erwarten, dass eine zur Laufzeit generierte Lösung Optimalität garantieren kann. Statt dessen soll eine optimale Lösung zur Compilezeit generiert und zur Laufzeit benutzt werden. Auf die Realisierung dieses Ansatzes wird in Abschnitt 4.4.1 ab Seite 75 eingegangen.

Die optimale Lösung kann durch ILP-Löser bestimmt werden. Dazu ist es nötig, die Problemstellung in Form von Gleichungen und Ungleichungen darzustellen. Zunächst werden die Variablen und Konstanten des ILP-Modells definiert.

|                            |  |
|----------------------------|--|
| $T$                        | Menge der betrachteten Zeitpunkte, $T \subset \mathbb{N}_0$  |
| $t \in T$                  | Zeitpunkt  |
| $I$                        | Menge der Indizes der Speicherobjekte, $I \subset \mathbb{N}_0$ ,<br>jedem Speicherobjekt wird ein eindeutiger Index<br>aus $\mathbb{N}$ zugeordnet            |
| $I_t$                      | Menge der Indizes der zum Zeitpunkt $t$ existierenden<br>Speicherobjekte, $I_t \subseteq I$  |
| $i \in I_t$                | Index eines Speicherobjektes $i$ zum Zeitpunkt $t$   |
| $MO$                       | Menge aller Speicherobjekte  |
| $mo_i$                     | Speicherobjekt mit dem Index $i$   |
| $SIZE_i$                   | Konstante, Größe des Speicherobjektes $mo_i$   |
| $SIZE_{SPM}$               | Konstante, Größe des Scratchpad-Speichers  |
| $DATA_i$                   | Konstante, 1, wenn $mo_i$ ein Datenobjekt ist, 0 sonst   |
| $STAT_i$                   | Konstante, 1, wenn $mo_i$ ein statisches Speicherobjekt<br>ist, 0 sonst  |
| $C_{MM \rightarrow SPM,i}$ | Konstante, Kopierkosten von Objekten der<br>Größe $SIZE(mo_i)$ vom Hauptspeicher in den<br>Scratchpad-Speicher   |
| $C_{SPM \rightarrow MM,i}$ | Konstante, Kopierkosten von Objekten der<br>Größe $SIZE(mo_i)$ vom Scratchpad-Speicher in den<br>Hauptspeicher   |
| $P_i$                      | Konstante, möglicher Profit, wenn Speicherobjekt $mo_i$<br>im Scratchpad ist, über alle Zeitscheiben des<br>Prozesses gemittelt                                |
| $T_{i,t}$                  | Konstante, 1, wenn zum Zeitpunkt $t$ der Prozess, zu dem<br>das Speicherobjekt $mo_i$ gehört, läuft, 0 sonst   |
| $B_{i,t}$                  | Konstante, 1, wenn Speicherobjekt $mo_i$ zum<br>Zeitpunkt $t$ blockiert ist, 0 sonst   |
| $x_{i,t}$                  | Binäre Entscheidungsvariable, 1, wenn das Speicher-<br>objekt $mo_i$ zum Zeitpunkt $t$ im Scratchpad-Speicher liegt,<br>0 sonst                                |
| $\psi_{i,t}$               | Binäre Entscheidungsvariable, 1 wenn das Speicher-<br>objekt $mo_i$ zum Zeitpunkt $t$ vom Hauptspeicher<br>in den Scratchpad-Speicher verschoben wird, 0 sonst |
| $\omega_{i,t}$             | Binäre Entscheidungsvariable, 1 wenn das Speicher-<br>objekt $mo_i$ zum Zeitpunkt $t$ vom Scratchpad-Speicher<br>in den Hauptspeicher verschoben wird, 0 sonst |

Durch die Möglichkeit, Objekte zu blockieren, verändern Belegungsentscheidungen auch den Spielraum für die nachfolgenden Entscheidungen. Daher sind zwei Arten optimaler Lösungen möglich. Lokal optimale Lösungen erstellen die Belegung ausgehend von dem einen betrachteten Zeitpunkt. Werden die lokal optimalen Lösungen aller Zeitpunkte bestimmt und hintereinander angewandt, so entsteht nicht zwingend eine global optimale Lösung, weil keine der lokalen Lösungen nachfolgende Belegungsschritte berücksichtigt. Daher ist der Lösungsraum eines Ansatzes, der auf einer globalen Sicht über alle Zeitpunkte basiert, größer. Auch zur Lösung des DSA-Problems ist eine globale Sicht über alle Zeitpunkte erforderlich, da jede Platzierungsentscheidung die Entscheidungen der späteren Zeitpunkte beeinflusst. Hier wird nach einer optimalen Lösung gesucht, anhand der die heuristischen Verfahren bewertet werden können. Die Heuristiken sollen sich über die Laufzeit vollständiger Programme bewähren, daher wird hier der globale Ansatz gewählt. Somit lautet die zu optimierende Zielfunktion:

$$\begin{aligned}
& \sum_{t \in T} \left( \sum_{i \in I_t} x_{i,t} \cdot P_i \cdot T_{i,t} \right. \\
& \quad - \sum_{i \in I_t \cap I_{t-1}} (\psi_{i,t} \cdot C_{MM \rightarrow SPM, SIZE_i}) \\
& \quad - \sum_{i \in I_t \cap I_{t-1}} (\omega_{i,t} \cdot C_{SPM \rightarrow MM, SIZE_i} \cdot DATA_i) \\
& \quad \left. - \sum_{i \in I_t \setminus I_{t-1}} (x_{i,t} \cdot C_{MM \rightarrow SPM, SIZE_i} \cdot STAT_i) \right) \\
& \longrightarrow \max
\end{aligned} \tag{3.1}$$

Die erste Zeile der Zielfunktion (3.1) modelliert die erwarteten Gewinne. Ist ein Objekt im Scratchpad-Speicher und ist der zugehörige Prozess aktiv, wird der Gewinn aufaddiert. Die zweite Zeile modelliert die Kopierkosten für alle Objekte vom Hauptspeicher in den Scratchpad-Speicher. Neu angelegte Objekte erzeugen dabei keinerlei Kosten, da sie nicht in der aufsummierten Menge vorhanden sind. In der dritten Zeile werden die Rückkopierkosten von Objekten aus dem Scratchpad- in den Hauptspeicher berücksichtigt. Für Instruktionsobjekte fallen diese Kosten nicht an. Die vierte Zeile fügt Kopierkosten für neu beim SPM angemeldete, statische Speicherobjekte, also globale Variablen und Funktionen, hinzu. Dynamisch erzeugte Speicherobjekte können direkt im Scratchpad-Speicher erzeugt werden. Da sie zu diesem Zeitpunkt noch keine Daten enthalten, fallen auch keine Kopierkosten an.

Bei der Optimierung der Zielfunktion müssen mehrere Beschränkungen beachtet werden. Zunächst müssen  $\psi_{i,t}$  und  $\omega_{i,t}$  die in der Definition genann-

ten Werte annehmen. Die Gleichung (3.2) zwingt  $\psi_{i,t}$  auf 1, wenn  $mo_i$  vom Hauptspeicher in den Scratchpad-Speicher verschoben wird.

Gleichung (3.3) zwingt analog  $\omega_{i,t}$  auf 1, wenn  $mo_i$  vom Scratchpad-Speicher in den Hauptspeicher verschoben wird. Automatisch nehmen  $\psi_{i,t}$  und  $\omega_{i,t}$  den Wert 0 an, denn so wird die Zielfunktion maximiert.

$$\forall t \in T \forall i \in I_t \psi_{i,t} \geq x_{i,t} - x_{i,t-1} \quad (3.2)$$

$$\forall t \in T \forall i \in I_t \omega_{i,t} \geq x_{i,t-1} - x_{i,t} \quad (3.3)$$

Blockierte Speicherobjekte dürfen weder vom Scratchpad-Speicher in den Hauptspeicher noch in anderer Richtung verschoben werden.

$$\forall t \in T \forall i \in I_t x_{i,t} - x_{i,t-1} \leq 1 - B_{i,t} \quad (3.4)$$

$$\forall t \in T \forall i \in I_t x_{i,t-1} - x_{i,t} \leq 1 - B_{i,t} \quad (3.5)$$

Die Summe des Platzbedarfes der Speicherobjekte im Scratchpad-Speicher darf die Größe des Scratchpads nicht übersteigen.

$$\forall t \in T \sum_{i \in I_t} x_{i,t} \text{SIZE}_i \leq \text{SIZE}_{SPM} \quad (3.6)$$

Die Anzahl der binären Variablen in der ILP-Formulierung beträgt  $O(|MO| \cdot |T|)$ . Dies ist nicht nur die obere Schranke für die Anzahl der  $x_{i,t}$ , sondern durch die Gleichungen (3.2) und (3.3) auch für  $\psi_{i,t}$  und  $\omega_{i,t}$ . Auch die Anzahl der Ungleichungen, die die Nebenbedingung bilden, ist durch  $O(|MO| \cdot |T|)$  beschränkt. CPLEX ist in der Lage, diese ILP-Gleichungen für die in Kapitel 5 betrachteten Benchmarks in unter einer Stunde Rechenzeit zu lösen.

Der Belegungsschritt geht davon aus, dass es ausreichend ist, wenn die Summe der Größe der Objekte im Scratchpad-Speicher dessen Speicherplatz nicht übersteigt. Diese Annahme kann fehlschlagen, wenn Scratchpad-Speicher durch Fragmentierung verloren geht. Dies ist durch ungeschickte Platzierung der Objekte möglich. Dieses Problem wird in Abbildung 3.4 dargestellt.

Falls eine Platzierung existiert, lässt diese sich ebenfalls durch eine ILP-Formulierung ermitteln. Zusätzlich zu den bisherigen Definitionen werden weitere verwendet.

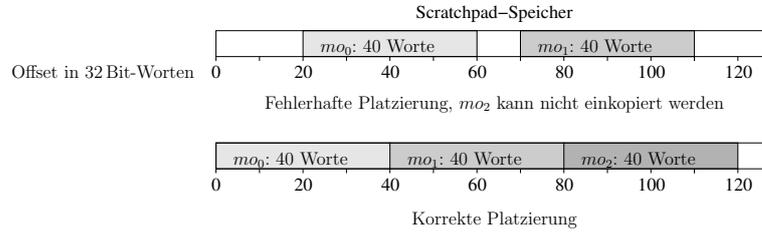


Abbildung 3.4: Fehlerhafte und korrekte Platzierung für drei Objekte mit je 40 Worten

|                   |   |
|-------------------|---|
| $I'_t$            | Menge der Indizes der Speicherobjekte zum Zeitpunkt $t$ , für die $x_{i,t} = 1$ gilt                                    |
| $i', j' \in I'_t$ | Index der Speicherobjekte zum Zeitpunkt $t$   |
| $START_{SPM}$     | Konstante, Startadresse des Scratchpad-Speichers  |
| $END_{SPM}$       | Konstante, höchste Adresse einer Scratchpad-Speicherzelle   |
| $a_{i',t}$        | Ganzzahlige Entscheidungsvariable, Startadresse des Speicherobjektes $mo_{i'}$ zum Zeitpunkt $t$ im Scratchpad-Speicher |

Das Positionierungsproblem lässt sich durch Ungleichungen wie folgt beschreiben. Speicherobjekte müssen sich vollständig im Scratchpad-Speicher befinden.

$$\forall t \in T \forall i' \in I'_t \quad START_{SPM} \leq a_{i',t} \leq END_{SPM} - SIZE_{i'} \quad (3.7)$$

Jede Speicherzelle des Scratchpad-Speichers darf von höchstens einem Speicherobjekt belegt werden, Speicherobjekte dürfen sich also nicht überschneiden. Die Gleichung modelliert, dass Speicherobjekt  $mo'_i$  entweder an einer höheren (erste Ungleichung) oder an einer niedrigeren (zweite Ungleichung) Speicherstelle im Scratchpad abgelegt wird als  $mo'_j$ .

$$\forall t \in T \forall i' \in I'_t \forall j' \in I'_t, i' \neq j' \quad \begin{aligned} a_{i',t} - a_{j',t} &\geq SIZE_{j'} & XOR \\ a_{j',t} - a_{i',t} &\geq SIZE_{i'} \end{aligned} \quad (3.8)$$

Speicherobjekte dürfen sich nur bewegen, wenn sie vom Hauptspeicher in den Scratchpad-Speicher oder zurück verschoben werden. Umsortierungen des Scratchpad-Speichers sollen nicht betrachtet werden.

$$\forall t \in T \forall i' \in I'_t \cap I'_{t-1} \quad a_{i',t} = a_{i',t-1} \quad (3.9)$$

Daraus lässt sich, wie im Anhang A ab Seite 103 beschrieben, ein ILP-Gleichungssystem formen, das sich mit CPLEX lösen lässt, wenn eine Lösung existiert. Der Zeitaufwand hierfür ist entsprechend des größeren Suchraumes ganzzahliger Entscheidungsvariablen erheblich größer. Die Anzahl der ganzzahligen Entscheidungsvariablen ist durch  $O(|MO| \cdot |T|)$  nach oben beschränkt, die Anzahl der binären Entscheidungsvariablen und der beschränkenden Ungleichungen ist im Anhang A angegeben.

Da die Lösung der ILP-Gleichungssysteme nur zur Compilezeit generiert wird und zur Laufzeit lediglich benutzt wird, ist die Vorhersagbarkeit des Verhaltens des Systems wie bei jeder Compilezeitlösung gegeben.

### 3.2.2 Statische Belegung

Eine einfache, heuristische Realisierung der blockierenden Strategie zur Belegung des Scratchpad-Speichers ist es, die Objekte aller Prozesse als blockiert anzunehmen. Speicherobjekte werden dann nicht mehr durch andere Objekte aus dem Scratchpad-Speicher verdrängt. In diesem Sinne kann man von einer statischen Belegung sprechen. Freie Bereiche des Scratchpad-Speichers werden mit Objekten aufgefüllt, ein vollständig belegtes Scratchpad bleibt voll, solange kein Objekt durch den besitzenden Prozess entfernt wird. Daher ist nicht zu erwarten, dass sich die Belegung des Scratchpad-Speichers häufig ändert, nachdem er einmal gefüllt wurde.

Dieser Belegungsansatz wird in [Pet04] SAMP genannt und zur Compilezeit verwendet. Die in [PMA<sup>+</sup>04] vorgestellte Programmierschnittstelle zur Scratchpad-Nutzung sieht ebenfalls keine Verdrängung von Speicherobjekten aus dem Scratchpad durch andere Objekte vor.

In der vorliegenden Arbeit soll der SPMM anhand des Nutzenmaßes zur Laufzeit entscheiden, welche Speicherobjekte in das Scratchpad verschoben werden sollen. Dazu wird bei der statischen Belegung entsprechend dem Greedy-Algorithmus [Weg02] für Knapsack-Probleme die Effektivität der Speicherobjekte aller Prozesse bestimmt. Dabei gilt

$$\text{Effektivität} = \frac{\text{möglicher Gewinn}}{\text{Objektgröße}}.$$

Die Belegungsstrategie setzt zunächst einen Zeiger  $p$  auf den Beginn des Scratchpad-Speichers. Dann wird das Speicherobjekt  $s$  mit der höchsten Effektivität ausgewählt, das im Hauptspeicher liegt, nicht blockiert ist und dessen Kopierkosten geringer sind als der erwartete Profit. Wird kein geeignetes  $s$  gefunden, so ist der Belegungsversuch beendet.

Im nächsten Schritt wird ein geeigneter Zielbereich für das Speicherobjekt gesucht. Dabei wird der Zeiger  $p$  über die im Scratchpad-Speicher befindlichen Speicherbereiche aufsteigend verschoben, bis ein ausreichend großer, freier Speicherbereich  $d$  gefunden wird, der  $s$  aufnehmen kann. Wird

dabei das Ende des Scratchpad-Speichers erreicht, so wird der Belegungsversuch abgebrochen. Wenn ein Bereich  $d$  gefunden wird, dann wird er vollständig oder teilweise  $s$  zugewiesen. Die darauf folgende Iteration beginnt mit der Auswahl des Speicherobjektes der nächst geringeren Effektivität.

Implizit wird mit Hilfe von  $p$  eine roving-Pointer[WJNB95] genannte Strategie benutzt. Der roving-Pointer wird für alle betrachteten Objekte eines Aufrufes der Belegungsstrategie nicht wieder auf den Anfang des Scratchpad-Speichers gesetzt, sondern wandert lediglich einmal darüber. Erst beim nächsten Aufruf der Strategie, bei einem Prozesswechsel oder bei der Erzeugung eines dynamischen Objektes wird der roving-Pointer  $p$  zurück auf den Anfang des Scratchpad-Speichers gesetzt. So wird ausgenutzt, dass aufgrund der statischen Belegungsstrategie nahezu keine dynamischen Objekte im Scratchpad platziert werden und auch selten freier Speicher im Scratchpad vorhanden ist. Dennoch können freie Speicherbereiche entstehen, wenn Objekte freigegeben werden, die sich im Scratchpad-Speicher befinden. Wenn diese vom roving-Pointer übersprungen werden, werden sie im gleichen Aufruf der Belegungsstrategie nicht erneut betrachtet. Strategien, die roving-Pointer verwenden, nehmen die so entstehende Fragmentierung des Scratchpad-Speichers in Kauf, um die Laufzeit der Heuristik zu verringern.

Aus den  $n$  Objekten aller Prozesse können in  $O(n)$  Schritten diejenigen ausgewählt werden, die im Scratchpad platziert werden sollen. Dieser Worst-Case-Fall tritt ein, wenn alle Speicherobjekte blockiert oder im Scratchpad-Speicher sind. Die Platzierung der Objekte ist bei einer Scratchpad-Größe  $S$  und einer Mindestgröße für Speicherobjekte im Scratchpad-Speicher  $M$  in höchstens  $O(S/M)$  Schritten möglich. Für jedes Objekt wird der roving-Pointer mindestens um  $M$  verschoben. Wird das Ende des Scratchpad-Speichers erreicht, so ist der Aufruf der Belegungsstrategie ebenso beendet wie im Fall, dass nicht genügend geeignete Objekte zur Verfügung stehen. Die asymptotische Laufzeit dieser Belegungsstrategie beträgt  $O(n + S/M)$ .

Die Vorhersagbarkeit der Belegung des Scratchpad-Speichers ist nicht gegeben. Speicherobjekte beliebiger Prozesse, die sich bereits im Scratchpad-Speicher befinden, können verhindern, dass weiteren Speicherobjekten Scratchpad-Speicher zugewiesen werden kann. Für Realzeitprozesse, die Scratchpad-Speicherplatz benötigen, ist eine Erweiterung erforderlich, wie in Abschnitt 3.2 beschrieben wurde.

### 3.2.3 Heuristische Belegung mit First-Fit

Die Auswahl der Objekte zur Belegung des Scratchpad-Speichers wird auch bei dieser Strategie durch die bereits im vorherigen Abschnitt erwähnte Greedy-Heuristik durchgeführt. Die Speicherobjekte der Prozesse sollen sich gegenseitig aus dem Scratchpad verdrängen können, so dass die Belegungsstrategie das Scratchpad für den jeweils aktuellen Prozess vorbereitet.

Dazu werden die Speicherobjekte des Prozesses nach absteigender Effektivität angeordnet. Mit dem Objekt der höchsten Effektivität beginnend, werden dann alle Speicherobjekte der Reihe nach betrachtet. Dabei wird ein Objekt, dessen möglicher Profit größer als seine Kopierkosten ist und das weder blockiert noch bereits im Scratchpad-Speicher ist, ausgewählt. Mit einer Adress-Ordered-First-Fit-Strategie wird nach einem entsprechend großen Speicherplatz im Scratchpad-Speicher für das ausgewählte Speicherobjekt gesucht, der entweder frei oder von einem nicht blockierten Objekt eines anderen Prozesses belegt ist. Der erste gefundene Speicherplatz wird vollständig oder teilweise benutzt, um das Objekt in den Scratchpad-Speicher zu kopieren. Darauf folgend, aber auch wenn kein mögliches Ziel gefunden wird, fährt die Strategie mit dem nächsten Speicherobjekt fort. Der Vorgang wird wiederholt, solange noch mögliche Objekte vorhanden sind.

Die Idee dieser Belegungsstrategie ist, so viele Speicherobjekte wie möglich im Scratchpad-Speicher unterzubringen und dabei nur überschaubare Rechenzeit in die Platzierung der Speicherobjekte zu investieren. Untersuchungen zufolge liefern Adress-Ordered-First-Fit-Strategien für Probleme der dynamischen Speicherallokation gute Ergebnisse mit geringer Fragmentierung bei hoher Platzierungsgeschwindigkeit. Objekte anderer Prozesse werden aus dem Scratchpad-Speicher verdrängt, um die durch die momentane Objektauswahl mögliche Energieersparnis für den aktuellen Prozess zu maximieren. Speicherobjekte des gleichen Prozesses werden nicht verdrängt. Ein Auskopieren von Objekten höherer Effektivität ist nicht sinnvoll, da sie mehr Ersparnis versprechen. Werden Objekte niedrigerer Effektivität ersetzt, so kann eine unnötige Umsortierung der Objekte im Scratchpad-Speicher, wie in Abbildung 3.5 dargestellt, eine mögliche Folge sein. Um diese Problematik zu vermeiden, werden auch Objekte niedriger Priorität des gleichen Prozesses nicht ersetzt.

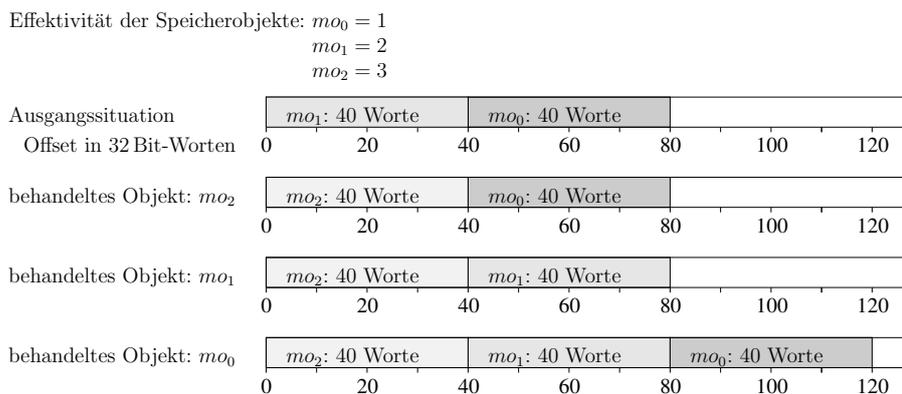


Abbildung 3.5: Unnötige Umsortierung der Speicherobjekte, wenn die Verdrängung der Objekte des gleichen Prozesses möglich ist

Bei  $n$  Objekten des betrachteten Prozesses, einer minimalen Größe  $M$  für Objekte im Scratchpad-Speicher und einer Scratchpad-Speichergöße  $S$  beträgt die Worst-Case-Laufzeit der Allokationsstrategie  $O(n \cdot (S/M))$ . Damit ist die Laufzeit linear von der Anzahl der vorhandenen Speicherobjekte und der Anzahl der bereits im Scratchpad befindlichen Objekte, die auch zu anderen Prozessen gehören können, abhängig.

Es ist nicht vorhersagbar, wann welche Speicherobjekte eines Prozesses im Scratchpad-Speicher abgelegt werden können. Ein Prozess kann keine eigenen Speicherobjekte im Scratchpad platzieren, wenn es mit blockierten Objekten anderer Prozesse belegt ist.

### 3.2.4 Heuristische Belegung mit Best-Fit

Nach der Auswahl der Speicherobjekte durch die bereits beschriebene Greedy-Heuristik kann die Platzierung der Objekte auch durch eine Best-Fit-Strategie erfolgen.

Statt den zuerst gefundenen, passenden Speicherplatz zu benutzen, wird eine erschöpfende Suche über alle möglichen Plätze im Scratchpad durchgeführt. Dabei wird der Speicherplatz anhand zweier Kriterien ausgewählt. Zuerst werden Ziele bevorzugt, bei denen keine Objekte aus dem Scratchpad entfernt werden müssen. So lassen sich nicht nur zu diesem Zeitpunkt Kopierkosten sparen, sondern auch in eventuell folgenden Schritten, da potenziell weniger Objekte erneut in den Scratchpad-Speicher kopiert werden müssen. Das untergeordnete Kriterium ist die Größe des Zielbereiches. Kleinere, möglichst exakt passende Bereiche werden bevorzugt. So sollen große, zusammenhängende Bereiche den größeren Objekten vorbehalten bleiben, um so mehr Objekte im Scratchpad unterzubringen. Nach der vollständigen Suche wird der gefundene Platz vollständig oder teilweise für das betrachtete Speicherobjekt verwendet. Wird kein geeigneter Speicherplatz gefunden, so wird mit dem nächsten Objekt fortgefahren, wobei erneut abgebrochen werden kann, sobald keine Kandidaten mehr übrig sind, die weder blockiert noch bereits im Scratchpad-Speicher sind und deren möglicher Profit größer ist als ihre Kopierkosten.

Die Best-Fit-Strategie benötigt durchschnittlich mehr Rechenzeit als die Adress-Ordered-First-Fit-Strategie, doch verspricht sie noch bessere Ausnutzung des vorhandenen Speicherplatzes durch weniger Fragmentierung. Auch hier werden Objekte ausschließlich anderer Prozesse aus dem Scratchpad-Speicher verdrängt, doch minimiert die Best-Fit-Strategie die Anzahl der verdrängten Objekte.

Die Worst-Case-Laufzeit der Best-Fit-Platzierungsstrategie ist ebenfalls  $O(n \cdot (S/M))$ , wobei  $n$  Objekte des aktuellen Prozesses mit einer minimalen Größe  $M$  in einem Scratchpad der Speichergöße  $S$  untergebracht werden sollen. Diese Strategie betrachtet immer mindestens so viele Zielbereiche im Scratchpad-Speicher wie die First-Fit-Strategie, im Durchschnitt jedoch

mehr. Die Laufzeit ist nicht nur linear in der Anzahl der Objekte des eigenen Prozesses, sondern auch in der Anzahl der bereits im Scratchpad befindlichen Objekte. Es ist nicht möglich, vorherzusagen, welche Objekte eines Prozesses im Scratchpad untergebracht werden können, da es mit blockierten Speicherobjekten anderer Prozesse belegt sein kann.

### 3.2.5 Heuristische Belegung in einem Durchlauf

Eine heuristische Belegung des Scratchpad-Speichers mit Objekten kann auf Kosten der Güte der Lösung auch in geringerer asymptotischer Laufzeit erfolgen. Eine Möglichkeit ist hierbei die Erweiterung der statischen Belegungsstrategie.

Im Vorbereitungsschritt werden, wie bereits beschrieben, die Speicherobjekte des zu optimierenden Prozesses nach ihrer Effektivität angeordnet. Zudem wird der roving-Pointer  $p$  auf den Anfang des Scratchpad-Speichers gesetzt.

Die Heuristik bestimmt den nächsten Kandidaten  $s$ , also das noch nicht betrachtete Speicherobjekt mit der höchsten Effektivität, das weder blockiert ist noch sich im Scratchpad befindet. Der Kandidat muss außerdem mehr Profit versprechen als seine Kopierkosten betragen. Wenn kein Kandidat gefunden werden kann, wird die Belegung des Scratchpad-Speichers damit abgeschlossen. Ist ein Objekt  $s$  größer als der ab  $p$  im Scratchpad insgesamt verfügbare Speicherplatz, so wird  $s$  übersprungen.

Im Anschluß wird eine mögliche Zieladresse  $d$  im Scratchpad-Speicher bestimmt. Dabei beginnt die Suche ab dem roving-Pointer  $p$ . Wird das Ende des Scratchpads erreicht, so wird die Scratchpad-Belegung damit beendet. Mögliche Ziele  $d$  sind freie Bereiche oder solche, die von unblockierten Speicherobjekten anderer Prozesse belegt werden. Hierbei können mehrere Speicherobjekte verdrängt werden, um größeren Kandidaten  $s$  ausreichend Platz zu schaffen. Abschließend wird der gefundene Speicherplatz  $d$  dem Objekt  $s$  vollständig oder teilweise zugewiesen und mit der Bestimmung des nächsten Kandidaten fortgefahren.

Abbildung 3.6 stellt das Verhalten dieser Heuristik in einem UML Aktivitätsdiagramm dar. Die Aktionen des Belegungsverfahrens werden durch abgerundete Rechtecke dargestellt, die durch den als Pfeile dargestellten Kontrollfluß verbunden werden. Bedingungen werden in eckigen Klammern an Fallunterscheidungen annotiert. Die exakte Bedeutung der Symbole wird in [HK03] erklärt.

Abermals wird für jeden Aufruf der Belegungsstrategie der roving-Pointer nur am Anfang zurückgesetzt. Diese Designentscheidung beschränkt die Zeit, die dafür aufgewendet wird, alle ausgewählten Speicherobjekte zu platzieren. Der Platzierungsschritt kann bei einer Scratchpad-Größe  $S$  und einer Mindestgröße für Speicherobjekte im Scratchpad-Speicher  $M$  in  $O(S/M)$  Schritten ausgeführt werden. Für jedes Objekt wird der roving-Pointer min-

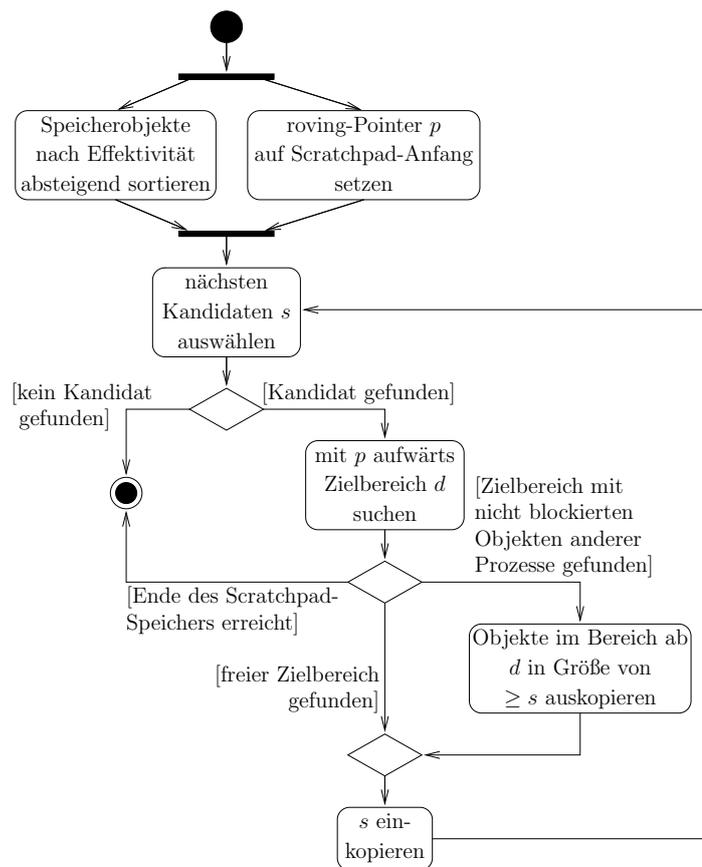


Abbildung 3.6: Ablauf der heuristischen Belegung in einem Durchlauf als UML Aktivitätsdiagramm

destens um  $M$  verschoben. Wird das Ende des Scratchpad-Speichers nach dessen Größe von  $S$  erreicht, endet der Belegungsschritt. Die Auswahl der zu platzierenden aus den  $n$  Speicherobjekten des aktuellen Prozesses erfordert im Worst-Case, der eintritt, wenn alle Objekte blockiert oder im Scratchpad sind,  $O(n)$  Schritte. Damit beträgt die Gesamtlaufzeit der Belegungsstrategie  $O(n + S/M)$ .

Die Idee hinter dieser Allokationsstrategie ist, bei möglichst geringer Laufzeit eine Belegung des Scratchpad-Speichers zu erzeugen. Statt die Energieersparnis durch Verschiebung der Speicherobjekte zu maximieren, wird der Aufwand der Belegungsstrategie und damit ihr Energiebedarf minimiert. Dadurch kann zusätzliche Energie eingespart werden, wenn die behandelte Problem Instanz ausreichend einfach ist, so dass trotz der geringen Laufzeit eine gute Belegung erzeugt wird.

Erneut ist die Belegung des Scratchpad-Speichers nicht vorhersagbar, da durch blockierte, im Scratchpad-Speicher befindliche Speicherobjekte beliebiger Prozesse verhindert werden kann, dass Objekte des gerade aktuellen Prozesses im Scratchpad abgelegt werden können.

### 3.2.6 Heuristische Belegung in drei Durchläufen

Die Belegung in drei Durchläufen versucht, Scratchpad-Belegungen mit geringerem Energiebedarf zu erzeugen, als es die Belegung in einem Durchlauf zu leisten vermag. Hierzu wird ein größerer Belegungsaufwand in Kauf genommen, wobei die asymptotische Laufzeit gleich bleibt. Bei dieser Strategie handelt es sich um einen Verbesserungsversuch gegenüber der Vorhergehenden. Er ist erfolgreich, wenn durch den zusätzlichen Aufwand mehr Energie eingespart als verbraucht wird.

Zunächst werden die Speicherobjekte nach ihrer Effektivität angeordnet und der roving-Pointer  $p_1$  auf den Anfang des Scratchpad-Speichers gesetzt.

Das unbetrachtete Speicherobjekt mit der nächsthöheren Effektivität, das weder blockiert noch bereits im Scratchpad vorhanden ist und außerdem noch mehr Profit verspricht als seine Kopierkosten betragen, wird zum nächsten Kandidaten  $s$  für eine Verschiebung in den Scratchpad-Speicher. Ist kein derartiges Objekt verfügbar, so wird der Belegungsversuch beendet.

Für die Verschiebung des Kandidaten  $s$  in das Scratchpad wird nun eine Zieladresse  $d$  ab dem roving-Pointer  $p_1$  gesucht. Im ersten Durchlauf werden lediglich Zielbereiche als mögliche Ziele akzeptiert, die vollständig unbelegt und ausreichend groß sind, um  $s$  aufzunehmen. Wird ein solches Ziel gefunden, so wird der Bereich vollständig oder teilweise dem Objekt  $s$  zugewiesen und mit dem nächsten möglichen Kandidaten fortgefahren. Erreicht der roving-Pointer  $p_1$  das Ende des Scratchpads, weil kein geeigneter Zielbereich gefunden wird, beginnt der zweite Durchlauf. Der Belegungsversuch endet, wenn kein weiterer geeigneter Kandidat gefunden wird.

Im zweiten Durchlauf werden im Gegensatz zum Ersten Objekte anderer Prozesse, die nicht blockiert sind, aus dem Scratchpad-Speicher verdrängt. Zur Platzierung wird ein neuer roving-Pointer  $p_2$  verwendet, der zu Beginn des zweiten Durchlaufes auf den Anfang des Scratchpads zeigt. Nach der Auswahl eines geeigneten Platzes  $d$ , der unbelegt oder von Objekten anderer Prozesse belegt ist, die nicht blockiert sind, werden dort eventuell vorhandene Objekte anderer Prozesse aus dem Scratchpad entfernt und, wenn es sich um Datenobjekte handelt, zurück in den Hauptspeicher geschrieben. Anschließend wird der Bereich  $d$  vollständig oder teilweise dem Kandidaten  $s$  zugewiesen und mit der Auswahl des nächsten Kandidaten, wie oben beschrieben, fortgefahren. Allerdings beginnt die Platzierung dieses Kandidaten sofort im zweiten Durchlauf, weil anzunehmen ist, dass das Scratchpad bereits zu einem Großteil belegt ist, wenn der erste Kandidat im zweiten Durchlauf angekommen ist. Erreicht der roving-Pointer  $p_2$  das Ende des Scratchpad-Speichers, so beginnt der dritte Durchlauf. Wird kein weiterer geeigneter Kandidat gefunden, endet der Belegungsversuch.

Im dritten Durchlauf werden zusätzlich Objekte des gleichen Prozesses aus dem Scratchpad-Speicher verdrängt, wenn sie über eine geringere Effektivität verfügen als das aktuell Betrachtete. Der Fall kann vorkommen, wenn zum Beispiel in einem vorherigen Belegungsschritt ein Objekt hoher Effektivität nicht platziert werden konnte, weil blockierte Objekte es verhinderten und der Platz von einem Objekt niedrigerer Effektivität eingenommen wurde. Sind die blockierten Objekte nun nicht mehr blockiert, wird plötzlich eine Platzierung des betrachteten Objektes möglich. Für den dritten Durchlauf wird abermals ein neuer roving-Pointer  $p_3$  verwendet, der zu Beginn des Durchlaufes auf den Anfang des Scratchpad-Speichers zeigt. Nachdem ein Zielbereich gefunden wurde, werden die dort befindlichen, nicht blockierten Speicherobjekte beliebigen Prozesses aus dem Scratchpad entfernt und gegebenenfalls in den Hauptspeicher zurückgeschrieben. Im Anschluß daran wird der gefundene Bereich teilweise oder vollständig dem Kandidatenobjekt  $s$  zugewiesen und das nächste Objekt wie oben beschrieben ausgewählt. Die Platzierung der so ausgewählten Kandidaten beginnt abermals im dritten Durchlauf, da damit zu rechnen ist, dass nur wenige nicht blockierte Objekte anderer Prozesse im Scratchpad vorhanden sind, wenn ein Objekt den dritten Durchlauf erreicht hat. Wird kein weiterer geeigneter Kandidat gefunden oder erreicht der roving-Pointer  $p_3$  das Ende des Scratchpad-Speichers, so endet der Belegungsversuch.

Abbildung 3.7 stellt den Ablauf der Heuristik in einem UML Aktivitätsdiagramm dar.

Für jeden Aufruf der Belegungsstrategie wird der roving-Pointer höchstens drei mal zurückgesetzt. Da konstante Faktoren in der O-Notation entfallen, kann der Platzierungsschritt für alle Objekte bei einer Scratchpad-Größe  $S$  und einer Mindestgröße für Speicherobjekte im Scratchpad-Speicher  $M$  in höchstens  $O(S/M)$  Schritten ausgeführt werden. Allerdings

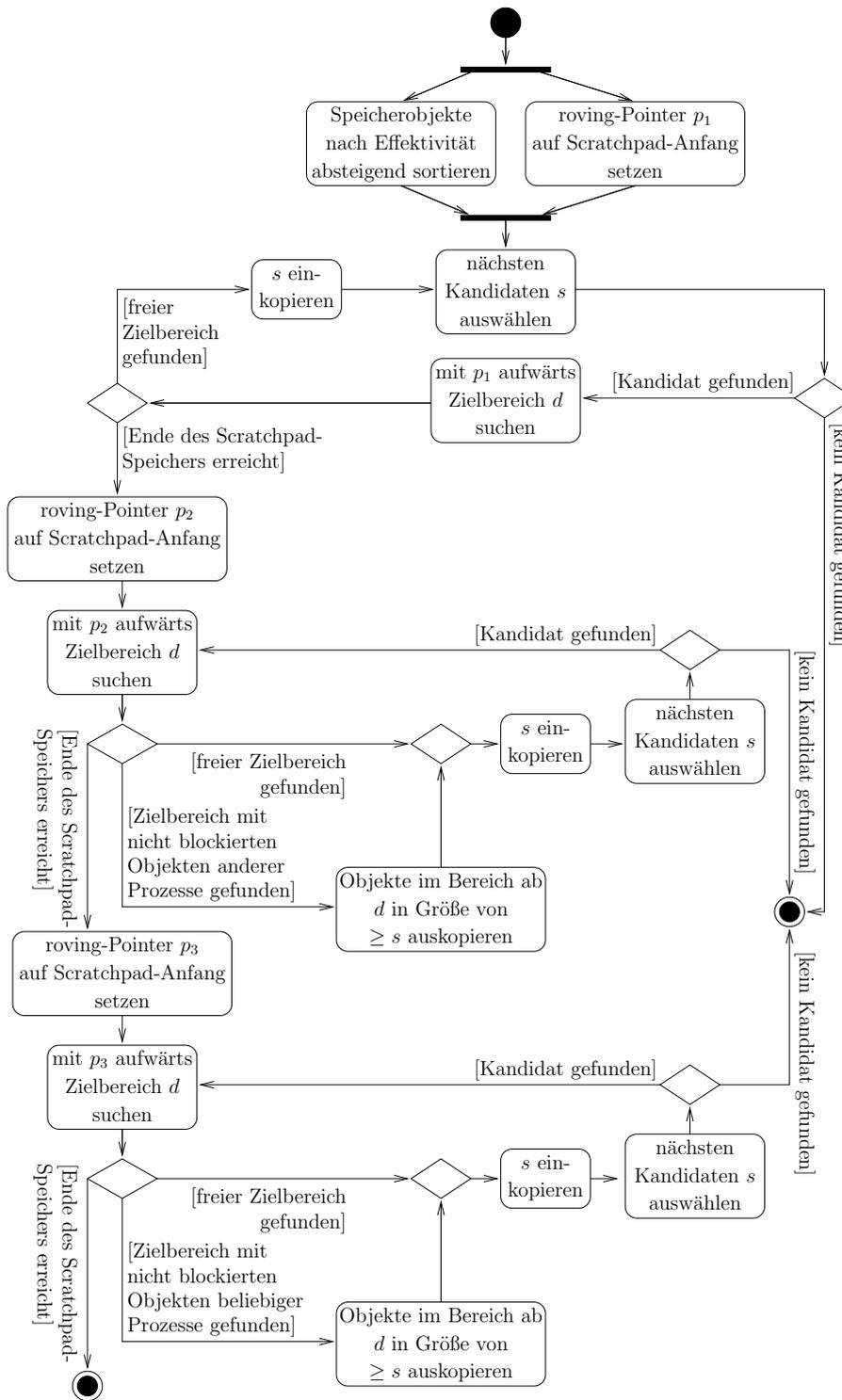


Abbildung 3.7: Ablauf der heuristischen Belegung in drei Durchläufen als UML Aktivitätsdiagramm

benötigt die Auswahl der zu platzierenden Objekte aus den  $n$  Speicherobjekten des aktuellen Prozesses im Worst-Case abermals  $O(n)$  Schritte. Die Laufzeit von Belegung und Positionierung beträgt damit insgesamt  $O(n + S/M)$ .

Diese Allokationsstrategie soll trotz geringer Laufzeit eine möglichst hohe Ausnutzung des Scratchpad-Speichers erzielen. Die Hoffnung ist, dass eine hohe Ausnutzung auch in einer hohen Energieersparnis resultiert.

Auch hier ist die Belegung des Scratchpad-Speichers nicht vorhersagbar, da durch blockierte Objekte beliebiger Prozesse verhindert werden kann, dass Objekte des gerade ausgeführten Prozesses im Scratchpad abgelegt werden können.

### 3.3 Belegung mit Replatzierung der Objekte

Wie in Abschnitt 3.1 beschrieben, ist zur korrekten Programmausführung notwendig, dass Prozesse blockierte Speicherobjekte dort wiederfinden, wo die Objekte waren, als sie als benutzt markiert wurden. Während der Ausführung anderer Prozesse ist es möglich, die Speicherobjekte im Hauptspeicher zu halten und das Scratchpad dem jeweils aktiven Prozess zur Verfügung zu stellen. Wird ein Prozess erneut aktiv, so müssen alle dazugehörigen blockierten Speicherobjekte wieder genau dort abgelegt werden, wo sie waren, als der Prozess zuletzt aktiv war. Die Objekte werden replaziert, so dass der Prozess bei seinen blockierten Objekten auf eine scheinbar unveränderte Situation trifft. Das beschriebene Konzept wird in Abbildung 3.8 dargestellt.

Auch bei Verwendung der Replatzierungsstrategie lässt sich das Problem der Objektauswahl nicht als Knapsack-Problem formulieren. Sobald blockierte Objekte an feste Adressen replaziert werden müssen, entstehen wie im blockierenden Fall mehrere Unterteilungen des Scratchpad-Speichers. Wie in Abschnitt 3.2 beschrieben, lässt sich diese Problemstellung als Multiple-Knapsack-Problem modellieren und ist NP-hart. Die Platzierung der ausgewählten Speicherobjekte in vollständig freien Speicherbereichen tritt weiterhin als Sonderfall auf und kann dann als DSA-Problem aufgefasst werden, das ebenfalls NP-hart ist.

#### 3.3.1 Optimale Lösung mittels ILP

Auch für die replatzierende Allokationstechnik sollen optimale Lösungen mittels ILP-Gleichungen erzeugt werden. Die Gleichungen werden zur Compilezeit erzeugt, von einem ILP-Löser gelöst und die Lösung dann, wie in Abschnitt 4.4.1 auf Seite 75 beschrieben, zur Laufzeit benutzt.

Da einmal im Scratchpad-Speicher befindliche Speicherobjekte auch dort hin replaziert werden müssen, beeinflussen die einmal getroffenen Belegungsentscheidungen auch die zukünftigen, so dass es sinnvoll ist, eine global

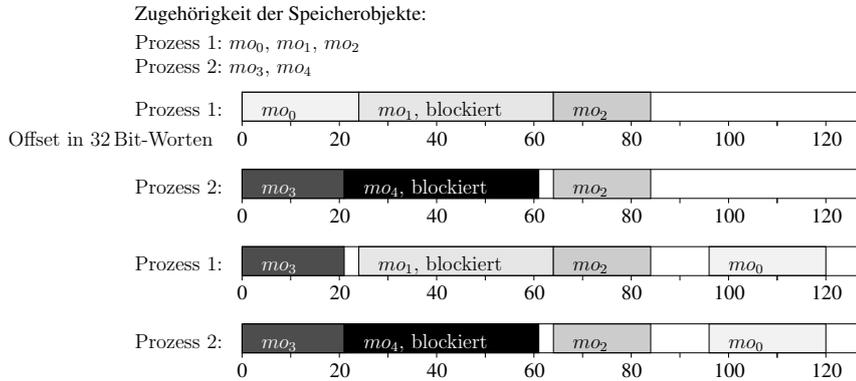


Abbildung 3.8: Beispiel für Replatzierung der Speicherobjekte

optimale Lösung über alle Zeiten zu bestimmen. Die Vereinigung lokal optimaler Lösungen führt nicht zwangsläufig zu einer global Optimalen.

Lediglich geringe Veränderungen sind nötig, um das in Unterabschnitt 3.2.1 ab Seite 34 beschriebene ILP-Gleichungssystem für die Scratchpad-Belegung mit blockierenden Objekten an die replatzierende Belegung anzupassen. Die Gleichungen (3.4) und (3.5) von Seite 37 werden ersetzt durch Gleichung (3.10).

$$\forall t \in T \forall i \in I_t \cap I_{s_{i,t}} T_{i,t} B_{i,t} x_{i,t} - T_{i,t} B_{i,t} x_{i,s_{i,t}} = 0 \quad (3.10)$$

Dabei ist  $s_{i,t} = \max(\{r \in T | r < t, T_{i,r} = 1\} \cup \{-1\})$  der letzte Zeitpunkt vor  $t$ , an dem sich das Objekt  $mo_i$  im Scratchpad-Speicher befand, wobei  $I_{-1} = \emptyset$  benötigt wird, um einen definierten Zustand auch für die erste Verschiebung eines Objektes in das Scratchpad zu ermöglichen. Somit sorgt Gleichung (3.10) dafür, dass die blockierten Objekte eines Prozesses erneut im Scratchpad abgelegt werden, wenn diese zum letzten vorangegangenen Zeitpunkt, an dem der Prozess ausgeführt wurde, ebenfalls dort waren.

Die Anzahl der binären Entscheidungsvariablen ist gegenüber dem blockierenden Ansatz unverändert in der Größenordnung  $O(|MO| \cdot |T|)$ , da die Gleichungen (3.2) und (3.3) von Seite 37 unverändert übernommen wurden. Damit besteht auch die Nebenbedingung weiterhin aus  $O(|MO| \cdot |T|)$  Ungleichungen. CPLEX löst diese ILP-Gleichungen mit geringem Zeitaufwand.

Die Positionierung der Objekte kann ebenfalls wie bereits in Abschnitt 3.2.1 beschrieben erfolgen. Zusätzlich muss jedoch Gleichung (3.11) verwendet werden. Sie sorgt dafür, dass die blockierten Objekte eines Prozesses im Scratchpad an der gleichen Adresse abgelegt werden, an der sie sich zum letzten, vorangegangenen Zeitpunkt befanden, an dem der Prozess ausgeführt wurde.

$$\forall t \in T \forall i' \in I'_t \cap I'_{s_{i',t}} B_{i',t} a_{i',t} - B_{i',t} a_{i',s_{i',t}} = 0 \quad (3.11)$$

Auf Gleichung (3.9) von Seite 38 kann nicht verzichtet werden. Ihr Fehlen würde Umsortierungen im Scratchpad-Speicher erlauben, die zu zusätzlichen Kosten führen, die im ILP-Modell für die Erzeugung der Objektauswahl nicht berücksichtigt werden und so die Optimalität der Lösung in Frage stellen. Die Benutzung beider Beschränkungen verkleinert die Menge möglicher Lösungen, erhält aber deren Optimalität.

Um CPLEX zur Lösung des entstehenden Gleichungssystems zu verwenden, ist eine Transformation in eine ILP-Eingabe nötig, wie sie in Anhang A ab Seite 103 beschrieben wird. Da gegenüber der Variante mit Blockieren der Speicherobjekte mehr Objekte im Scratchpad-Speicher untergebracht werden können, ist die tatsächliche Anzahl der Beschränkungen ebenfalls größer als bei dem blockierenden Ansatz. Dennoch bleibt die Anzahl der ganzzahligen Entscheidungsvariablen durch  $O(|MO| \cdot |T|)$  beschränkt. Im Anhang A wird die Zahl der beschränkenden Ungleichungen und der binären Entscheidungsvariablen angegeben.

Auch hier handelt es sich um eine zur Compilezeit generierte Belegung, die zur Laufzeit lediglich benutzt wird. Daher ist die Vorhersagbarkeit wie bei jeder Compilezeitlösung gegeben.

### 3.3.2 Dynamische Belegung

Eine einfache Realisierung der replatzierenden Strategie ist es, jedem Prozess das Scratchpad vollständig zur Verfügung zu stellen. Dieser Ansatz wird in [Pet04] DAMP genannt. Der SPMM reserviert dazu für jeden Prozess einen Puffer im Hauptspeicher, der exakt die Größe des Scratchpad-Speichers annimmt. Bei Prozesswechseln wird der Scratchpad-Inhalt in den Puffer des vorerst endenden Prozesses kopiert. Damit ist der Scratchpad-Speicher frei und der Inhalt des Puffers des aufkommenden Prozesses kann in den Scratchpad-Speicher kopiert werden.

Die Auswahl der Speicherobjekte, die in den Scratchpad-Speicher sollen, kann nun für jeden Prozess unabhängig nach einer beliebigen Strategie erfolgen. Wichtig ist jedoch, dass blockierte Objekte nicht aus dem Scratchpad in den Hauptspeicher verschoben werden.

Die Belegung wird wie in Unterabschnitt 3.2.2 beschrieben erzeugt. Speicherobjekte anderer Prozesse können aufgrund des Umkopierverfahrens nicht im Scratchpad-Speicher des betrachteten Prozesses liegen und brauchen daher nicht berücksichtigt zu werden. Hingegen erscheint es, wie im Unterabschnitt 3.2.3 beschrieben, wenig sinnvoll, Objekte des gleichen Prozesses zu verdrängen. Um den Rechenaufwand der Heuristik zu verringern, werden daher keine Verdrängungen von Speicherobjekten aus dem Scratchpad-Speicher betrachtet.

Entsprechend können Belegung und Platzierung der  $n$  Objekte in einem Scratchpad der Größe  $S$  und einer Mindestgröße  $M$  für Speicherobjekte im Scratchpad-Speicher in  $O(n + S/M)$  Schritten ausgeführt werden, wie in Unterabschnitt 3.2.2 begründet wird. In diesem Falle ist  $n$  jedoch die Anzahl der Objekte des aktuellen Prozesses, nicht aller Prozesse, wie bei der statischen Strategie.

Die Belegung des Scratchpad-Speicher ist vom Standpunkt eines Prozesses aus vorhersagbar. Jedem Prozess steht der volle Scratchpad-Speicher zur Verfügung. Anhand der bekannten Effektivität aller Objekte lässt sich dann leicht vorhersagen, welche Speicherobjekte bei Prozessstart im Scratchpad platziert werden und welche später noch hinzu kommen. Um Prozessen die Einhaltung von Zeitschranken zu ermöglichen, können die Nutzenwerte der Speicherobjekte gezielt erhöht oder gesenkt werden, um genau die benötigten Objekte im Scratchpad unterzubringen.

### 3.3.3 Heuristische Belegung mit Blockstrategie

Bei der Belegung des Scratchpad-Speichers nach der im vorangehenden Abschnitt beschriebenen dynamischen Methode wird zum einen von jedem Prozess ein Puffer in Größe des Scratchpads benötigt, zum anderen finden unnötige Kopieraktionen statt, die nicht zulassen, bei geringer Scratchpad-Ausnutzung Daten über einen Prozesswechsel hinweg im Scratchpad-Speicher zu halten. Die Blockstrategie löst beide Probleme. Dynamische Speicherobjekte werden jedoch nicht behandelt, um die Platzierungsstrategie zu vereinfachen. Diese Entscheidung basiert auf der Hoffnung, dass dennoch ausreichend viele Speicherobjekte zur Belegung des Scratchpad-Speichers zur Verfügung stehen, auf die ausreichend häufig zugegriffen wird.

Die Speicherobjekte des betrachteten Prozesses werden nach ihrer Effektivität absteigend angeordnet. Hieraus werden Objekte ausgewählt, die noch nicht im Scratchpad-Speicher liegen und deren mögliche Energieeinsparung größer ist als die Kopierkosten. Würde die Auswahl eines Speicherobjektes die Summe der Größen der ausgewählten Objekte über die Größe des Scratchpad-Speichers hinaus erhöhen, so wird dieses Objekt übersprungen. Objekte, die im Scratchpad waren und blockiert sind, werden bevorzugt ausgewählt. Die ausgewählten Objekte bilden einen Verbund, den sogenannten Block.

Beim ersten Aufruf der Strategie werden die Speicherobjekte ohne Zwischenräume innerhalb des Blocks in der Reihenfolge ihrer Auswahl angeordnet. Der ganze Block von Speicherobjekten wird nun platziert. Wenn möglich, wird der Block in einem freien Bereich des Scratchpad-Speichers abgelegt. Wenn der Block nicht in freiem Speicher untergebracht werden kann, wird er am Anfang des Scratchpad-Speichers untergebracht, nachdem dort liegende Speicherobjekte aus dem Scratchpad-Speicher entfernt und zurück in den Hauptspeicher geschrieben wurden, wenn es sich um Da-

tenobjekte handelt. Sobald genug Platz frei ist, wird der Block mit den enthaltenen Objekten dort einkopiert. Im hinteren Bereich des Scratchpad-Speichers sollen sich so Objekte anlagern, die bis zum nächsten Wechsel in deren zugehörigen Prozess nicht mehr verschoben werden.

Bei weiteren Aufrufen der Scratchpad-Allokationsstrategie können Blöcke entstehen, die Objekte enthalten, die bereits im Scratchpad-Speicher waren und blockiert sind. Da ausschließlich statische Speicherobjekte betrachtet werden, können die Speicherobjekte wie beim ersten Aufruf der Strategie der Reihenfolge nach angeordnet werden. Die blockierten Objekte nehmen innerhalb des Blocks den gleichen Platz ein, so dass sie wieder an den ursprünglichen Positionen im Scratchpad abgelegt werden. Die Startadresse des Blocks wird durch die darin enthaltenen blockierten Objekte bestimmt. Ab dieser Startadresse bis zum Ende des Blocks werden alle Speicherobjekte aus dem Scratchpad-Speicher entfernt und zurück in den Hauptspeicher geschrieben, falls es sich um Datenobjekte handelt. Da Blöcke lediglich aus statischen Objekten bestehen und zusammenhängend erzeugt wurden, können in den Blöcken keine freien Bereiche sein. Damit werden auch nicht mehr Objekte aus dem Scratchpad entfernt, als nötig.

Die Blockstrategie hat zum Ziel, mit möglichst geringem Aufwand für Belegung und Platzierung eine Scratchpad-Belegung zu erzeugen. Der Erfolg der Allokationsstrategie ist dabei wesentlich von der Applikation abhängig. Die Hoffnung ist, dass die Blockstrategie bei häufig vorkommenden Mustern von Speicherobjekten gut funktioniert und so Energie einsparen kann.

Die Auswahl der Speicherobjekte zur Erzeugung eines Blocks kann in  $O(n)$  Schritten erfolgen, dabei ist  $n$  die Anzahl der Speicherobjekte des betrachteten Prozesses. Diese Worst-Case-Laufzeit tritt auf, wenn ausreichend viele Objekte zur Auswahl stehen, die nicht ausgewählt werden dürfen, weil sie beispielsweise zu groß sind, um in das Scratchpad verschoben zu werden. Die Worst-Case-Laufzeit der Platzierung der Objekte liegt in der Größenordnung  $O(S/M)$ , wobei  $S$  die Scratchpad-Größe und  $M$  die Mindestgröße der Objekte im Scratchpad-Speicher ist. Bei der Suche nach einem Platz für einen Block mit Mindestgröße  $M$  werden im schlechtesten Fall so viele Speicherobjekte im Scratchpad als Ziel in Betracht gezogen, wie hinein passen, bevor der Block am Anfang platziert wird. Höchstens  $S/M$  Speicherobjekte können im Scratchpad-Speicher sein, was die Laufzeit bestimmt. Die Gesamtlaufzeit für Belegung und Platzierung beträgt somit  $O(n + S/M)$ .

Zur Compilezeit lässt sich vorhersagen, welche Speicherobjekte für jeden Prozess ausgewählt werden, da die Effektivität der statischen Objekte bereits zur Compilezeit bekannt ist. Sollen bestimmte Objekte im Scratchpad platziert werden, zum Beispiel um Realzeitvorgaben zu erfüllen, so können die Nutzenwerte der betreffenden Objekte gegenüber anderen gezielt erhöht werden. Prozesse können den Scratchpad-Inhalt anderer Prozesse nicht beeinflussen. Allerdings muss damit gerechnet werden, dass die Speicherobjekte der verschiedenen Prozesse sich ständig gegenseitig überschreiben und

so immer wieder zusätzliche Kopieraktionen nötig werden, die die Ausführungszeiten beeinflussen.

### 3.4 Bewertung

Zumindest der optimalen Lösung der replatzierenden Strategie durch die ILP-Gleichungen steht die Möglichkeit offen, die Speicherobjekte nicht umzukopieren. Die Lösungsmenge des blockierenden Verfahrens ist nur eine Teilmenge der Lösungsmenge des replatzierenden Verfahrens. Deshalb ist zu erwarten, dass die optimale, replatzierende Lösung mindestens so gute Ergebnisse erzielt, wie die Blockierende. Wenn die Kopierkosten relativ zu den Profiten niedrig sind, werden die Lösungen der replatzierenden Strategie mehr Energie einsparen als die der Blockierenden.

Bei beiden Strategien liefert die Lösung durch ILP-Gleichungen optimale Ergebnisse, wenn eine Lösung gefunden wird. Nicht bei allen Eingaben muss eine Lösung gefunden werden, gerade bei großen Eingaben, die viele Speicherobjekte, viele betrachtete Zeitpunkte oder große Scratchpad-Speicher betrachten, steigt die Rechenzeit der ILP-Löser extrem an.

Den ILP-basierten Verfahren stehen mehr Informationen zur Verfügung als den online Heuristiken. In den ILP-Formulierungen wird die globale Sicht über alle Zeitpunkte verwendet, um optimale Ergebnisse zu garantieren. Die online Heuristiken müssen mit der Information des aktuell betrachteten Zeitpunktes und der Vorangegangenen auskommen. Bei der replatzierenden Strategie bedeuten Fehler durch diese Einschränkung der Heuristik zusätzliche Kopierkosten. Wenn Kopieren von Speicherobjekten geringe Energiekosten verursacht, sind Fehlentscheidungen mit entsprechend geringem Mehraufwand korrigierbar und daher nicht sehr schwerwiegend. Im Fall der blockierenden Strategie sind Entscheidungen, die sich in späteren Schritten als Fehler erweisen, nicht so leicht korrigierbar. Ist ein Objekt blockiert, so wird es nicht mehr bewegt, auch wenn dadurch Objekte eines anderen Prozesses, die mehr Energie sparen könnten, zur Verfügung stehen. Daher ist zu erwarten, dass die Heuristiken mit blockierender Strategie gegenüber der optimalen Lösung weit schlechter abschneiden als die replatzierenden Heuristiken gegenüber der entsprechenden optimalen Lösung. Zusätzlich hierzu müssen die blockierenden Heuristiken ebenso wie die dynamische Heuristik das schwerere Multiple-Knapsack-Problem lösen, während die beschriebene Blockstrategie sich aufgrund der einschränkenden Annahmen lediglich mit dem einfachen Knapsack-Problem beschäftigen muss. Daher ist davon auszugehen, dass die replatzierenden Heuristiken sehr viel bessere Ergebnisse erzielen, als die Blockierenden.

Alle vorgestellten Heuristiken sind, wie in Tabelle 3.1 zu sehen ist, auf geringe Laufzeit optimiert. Dennoch beträgt die Worst-Case-Laufzeit je Aufruf bei der blockierenden heuristischen Belegung mit First-Fit beziehungsweise

|               | Worst-Case-Laufzeit je |                     |                    |
|---------------|------------------------|---------------------|--------------------|
|               | Belegungsschritt       | Platzierungsschritt | Aufruf insgesamt   |
| blockierend   |                        |                     |                    |
| statisch      | $O(n)$                 | $O(S/M)$            | $O(n_+ + S/M)$     |
| First-Fit     | $O(n)$                 | $O(S/M)$            | $O(n \cdot (S/M))$ |
| Best-Fit      | $O(n)$                 | $O(S/M)$            | $O(n \cdot (S/M))$ |
| 1 Durchlauf   | $O(n)$                 | $O(S/M)$            | $O(n + S/M)$       |
| 3 Durchläufe  | $O(n)$                 | $O(S/M)$            | $O(n + S/M)$       |
| replatzierend |                        |                     |                    |
| dynamisch     | $O(n)$                 | $O(S/M)$            | $O(n + S/M)$       |
| Block         | $O(n)$                 | $O(S/M)$            | $O(n + S/M)$       |

Tabelle 3.1: Übersicht über die Worst-Case-Laufzeiten der Heuristiken

Best-Fit  $O(n \cdot (S/M))$ , bei den anderen Verfahren  $O(n + S/M)$  und ist damit von der Eingabe abhängig. Für die Einhaltung von Echtzeitschranken kann das fatale Folgen haben. Darauf soll kurz eingegangen werden. Bei der statischen Heuristik ist die Worst-Case-Laufzeit  $O(n_+ + S/M)$ , wobei  $n_+$  die Gesamtzahl der Speicherobjekte aller Prozesse ist. Damit können andere Prozesse durch die Anzahl ihrer Speicherobjekte die Zeit beeinflussen, die der Scheduler benötigt, um den Realzeitprozess zur Ausführung zu bringen. Zusagen über die Realzeitfähigkeiten des Systems sind so nicht mehr möglich. Für alle anderen blockierenden Heuristiken sind die Laufzeiten jedoch von der Anzahl der Speicherobjekte  $n$  des aufrufenden Prozesses abhängig, so dass jeder Prozess die Laufzeit der Heuristik bestimmen kann, die der Scheduler aufruft, wenn der Prozess aktiviert werden soll. Da aufgrund der blockierenden Strategie, wie in Abschnitt 3.2 beschrieben, nicht vorhergesagt werden kann, ob ein Prozess seine Objekte im Scratchpad-Speicher unterbringen kann, ist die Benutzung von Scratchpad-Speicher über die Heuristiken für Realzeitprozesse nicht nutzbringend in Bezug auf deren Worst-Case-Laufzeit. Deshalb sollten solche Prozesse auf die Benutzung des SPMM mit blockierenden Heuristiken verzichten. Wenn die bereits erwähnten Erweiterungen benutzt werden, um die Vorhersagbarkeit für Realzeitprozesse zu verbessern, wird keine der Heuristiken benutzt und deren Laufzeiten entfallen. Bei beiden replatzierende Verfahren ist die Laufzeit ebenfalls von der Anzahl  $n$  der Speicherobjekte des betrachteten Prozesses abhängig, nicht aber von anderen Prozessen zu beeinflussen. Beschränkt ein Realzeitprozess die Anzahl  $n$  seiner Speicherobjekte auf eine ausreichend kleine Konstante, so ist die Laufzeit beider Heuristiken mit  $O(S/M)$  kleiner als die Zeit  $O(S)$ , die für das Ein- beziehungsweise Auskopieren der Speicherobjekte benötigt wird. Bei konstanter Scratchpadgröße  $S$  ist damit auch die Laufzeit der Heuristiken durch eine Konstante beschränkt und lässt sich in Echtzeit ausdrücken. Bei beiden replatzierenden Heuristiken ist die Scratchpad-Belegung

vorhersagbar, deshalb können Realzeitprozesse unter oben genannter Bedingung durch Benutzung des SPMM nicht nur Energie sparen, sondern auch ihre Realzeitschranken einhalten.

Die Energieersparnis der verschiedenen Verfahren zur Belegung des Scratchpad-Speichers wird anhand einer Auswahl von Benchmarks im Kapitel 5 ab Seite 81 verglichen.



## Kapitel 4

# Arbeitsablauf und Implementierung

Das vorangegangene Kapitel erklärt, wie der SPMM Belegungen des Scratchpad-Speichers erzeugt. Das vorliegende Kapitel geht nun auf die Realisierung dieser Verfahren ein. Der nachfolgende Abschnitt 4.1 beschreibt, wie ein Programm erzeugt wird, dass den SPMM benutzt. Die vom SPMM angebotene Programmierschnittstelle wird im Abschnitt 4.2 beschrieben. Zur Vereinfachung der Prozessverwaltung werden auf diese Programmierschnittstelle zugeschnittene Funktionen angeboten, die in Abschnitt 4.3 vorgestellt werden. Die möglichen und tatsächlich gewählten Wege bei der Implementierung des SPMM stellt Abschnitt 4.4 dar. Abschnitt 4.5 zeigt anhand von drei Beispielen, wie der SPMM in verschiedene eingebettete Betriebssysteme integriert werden kann.

### 4.1 Arbeitsablauf zur Compilezeit

Alle Programme, die den SPMM benutzen, müssen für ihre Speicherobjekte das bereits angesprochene Nutzenmaß zur Verfügung stellen, damit der SPMM entscheiden kann, welche Objekte im Scratchpad-Speicher platziert werden sollen. Der `profitAnnotator` übernimmt die Berechnung des Nutzenmaßes für alle Speicherobjekte.

Dynamische Speicherobjekte werden explizit vom Programm erzeugt. Im verwendeten Arbeitsablauf müssen sie manuell durch SPMM-Objekte ersetzt und das Programm an die in Abschnitt 3.1 auf Seite 31 beschriebene doppelte Dereferenzierung der dynamischen Objekte angepasst werden. Der Nutzen der dynamischen Speicherobjekte wird durch den `profitAnnotator` bestimmt und eingetragen.

Für die statischen Speicherobjekte, also globale Variablen und Funktionen, können die SPMM-Objekte bereits zur Compilezeit durch den `profitAnnotator` erzeugt werden. Dabei wird auch der Nutzen der Speicherobjek-

te in den SPMM-Objekten vermerkt. Alle Zugriffe auf die Speicherobjekte werden automatisch auf die doppelte Dereferenzierungsstufe in Form der SPMM-Objekte umgelenkt.

Bei statischen und dynamischen Speicherobjekten sind die Markierungen der benutzten Objekte vom Programmierer einzubringen. Eine automatische Positionierung dieser Markierungen kann zusätzlichen Gewinn bringen und das Ziel von Optimierungen sein, ist jedoch nicht Gegenstand der vorliegenden Arbeit.

Der profitAnnotator bereitet die Applikation auf die Benutzung des SPMM vor. Die Applikation kann aus mehreren Prozessen bestehen. Vereinfachend verarbeitet der profitAnnotator jedoch nur eine einzelne C-Quelldatei. In dieser Datei sind bereits die Funktionsaufrufe zur Markierung benutzter Objekte und die SPMM-Aufrufe zur Erzeugung und Freigabe dynamischer Objekte enthalten.

Die Eingabedatei wird zunächst kompiliert. Dabei wird eine memory-map-Datei erzeugt, in der die Positionierung und die Größe aller Funktionen und globalen Variablen angegeben wird. Durch Verwendung spezieller Parameter bei der Kompilierung des SPMM werden bei der Simulation mit Hilfe des MPARAM die Aufrufe der Speicherallokations- und Speicherfreigabefunktionen des SPMM mit ihren Parametern und Ergebnissen protokolliert. Die beiden eigentlichen Funktionen des SPMM werden deaktiviert. Statt dessen werden dynamische Speicherobjekte ausschließlich im Hauptspeicher angelegt und nicht mehr freigegeben. So ist jedes dynamische Objekt eindeutig einem Speicherbereich zuzuordnen.

Aus der memory-map-Datei und dem Aufrufprotokoll des SPMM lässt sich ein eindeutiges Abbild des Speichers der simulierten Prozesse entwerfen. Die in der, während der Simulation erzeugten, Trace-Datei abgelegten Lese- und Schreibzugriffe werden so den dynamischen und statischen Speicherobjekten zugeordnet. Daraus lässt sich der vorläufige Profit anhand des verwendeten Nutzenmaßes, das in Abschnitt 4.4 auf Seite 67 beschrieben wird, bestimmen. Für die dynamischen Speicherobjekte wird der durchschnittliche Nutzen aller von einem SPMM-Aufruf erzeugten Speicherobjekte an diesen Aufruf annotiert. Im Anschluß werden für die statischen Speicherobjekte SPMM-Objekte angelegt, die unter anderem auch Größe und Nutzen des Speicherobjektes enthalten.

Die statischen SPMM-Objekte werden außerdem in Speicherstrukturen eingetragen, die von der Erweiterung des Programmladers durch den SPMM verarbeitet werden können. Dabei werden die Objekte den Prozessen durch zusätzliche, im Quellcode abgelegte Informationen zugeordnet.

Schließlich werden noch alle Funktionen in separate Quelldateien verschoben. Dies ist nötig, um den verwendeten GCC zu zwingen, ausführbaren Code zu erzeugen, der auf Instruktionsobjekte an beliebig entfernten Adressen zugreifen kann. Wenn sich die Funktionen in der gleichen Quelldatei befinden, werden Funktionen aus Optimierungsgründen positionsabhän-

gig aufgerufen. Wird eine Funktion in den Scratchpad-Speicher verschoben, so kann die Funktion nicht mehr relativ adressiert werden und der Aufruf schlägt fehl.

Die beschriebenen Quellcode-Transformationen werden mit Hilfe des ICD-C[ICD06] durchgeführt. Kompiliert man das modifizierte Programm, so ist die Größe der Funktionen gegenüber dem Ergebnis des ersten Kompilervorgangs verändert, weil die Zugriffe auf die Speicherobjekte in der modifizierten Version über die zweite Dereferenzierungsstufe der SPMM-Objekte erfolgen und außerdem die Speicherallokationsanforderungen an den SPMM mit veränderten Nutzenwerten ausgestattet sind. Die Größe der Speicherobjekte muss in den SPMM-Objekten korrigiert werden, hierzu ist ein erneutes Kompilieren und Auswerten der neu erzeugten memory-map-Datei nötig. Die Änderung der Funktionsgrößen kann zu einer minimal veränderten Zugriffszahl auf Instruktionsobjekte führen. Um die Nutzenwerte anzupassen, ist ein weiterer Simulationslauf notwendig, bei dem eine neue Trace-Datei erzeugt wird. Daraus lassen sich die nun gültigen Nutzenwerte berechnen und in die SPMM-Objekte eintragen. Da sich die Nutzenwerte der Allokationsanforderungen an den SPMM dabei nicht verändern, sind keine weiteren Korrekturen nötig. Ein einziger Simulationslauf genügt jedoch nicht, um eine exakte Berechnung der Nutzenwerte zu gewährleisten, da für die dynamischen Speicherobjekte die Nutzenwerte im Quelltext der Funktionen eingetragen werden müssen, was die Nutzenwerte der Funktion verändert. Die Veränderung der Nutzenwerte ist jedoch so gering, dass der zweite Simulationslauf optional weggelassen werden kann, ohne die Belegungsentscheidungen des SPMM wesentlich zu beeinflussen.

Für die heuristischen Verfahren sind damit die Voraussetzungen geschaffen und sie können den Energieverbrauch bei Ausführung der Applikation verringern. Für die optimalen, auf ILP-Gleichungen basierenden Verfahren müssen die Belegungen zur Compilezeit erzeugt werden, was im Folgenden beschrieben wird.

Alle zur Erzeugung der ILP-Gleichungen benötigten Informationen lassen sich einer generierten Profildatei entnehmen. Diese Profildatei enthält die Aufrufe der verschiedenen SPMM-Funktionen zusammen mit deren Parametern und Ergebnissen. Das Markieren von benutzten Speicherobjekten, das Entfernen selbiger Markierungen, Speicherallokationsaufrufe, Speicherfreigaben und Prozesswechsel werden in chronologischer Reihenfolge festgehalten. Die Profildatei wird vom SPMM während eines Simulationslaufes erzeugt, wenn der SPMM mit speziellen Parametern compiliert wurde. Der outputGenerator automatisiert die Erzeugung der Profildatei.

Der assignmentILPgenerator übernimmt die Profildatei, verfolgt den Verlauf aller Aufrufe in ebenfalls chronologischer Reihenfolge und bildet deren Wirkung nach. Dabei wird für jeden betrachteten Zeitpunkt der Zustand aller zu diesem Zeitpunkt vorhandenen Speicherobjekte abgelegt. Der erste betrachtete Zeitpunkt ist  $t = 0$ . Alle Aufrufe von Funktionen des SPMM bis

zu einer Anwendung der Belegungsstrategie werden dem gleichen Zeitpunkt zugeordnet. Die Belegungsstrategie wird bei der Anforderung oder Freigabe dynamischer Speicherobjekte und bei Prozesswechseln aufgerufen. Werden die Funktionsaufrufe dieser SPMM-Operationen gefunden, so endet der aktuelle Zeitpunkt dort. Für jeden Aufruf der Belegungsstrategie wird auch ein Zeitpunkt  $t$  in den ILP-Gleichungen angelegt. Der momentane Zustand aller Speicherobjekte wird in den neuen Zeitpunkt übernommen.

Wird bei der Verarbeitung der Profildatei ein SPMM-Aufruf gefunden, der ein statisches oder dynamisches SPMM-Objekt erzeugt, wird auch im `assignmentILPgenerator` ein Objekt ab dem aktuell betrachteten Zeitpunkt angelegt. Bei einem Aufruf der Funktion des SPMM zum Markieren von Objekten als benutzt wird die entsprechende Markierung auch in den Objekten des `assignmentILPgenerator` durchgeführt. Dabei können Objekte mehrfach als benutzt markiert werden, um Rekursion zu ermöglichen. Auch die Freigabe von Speicher durch den SPMM wird im `assignmentILPgenerator` widergespiegelt. Dabei wird das freigegebene Objekt aus der Menge der Objekte des aktuellen Zeitpunktes entfernt. Beim nächsten Aufruf der Belegungsstrategie wird das Objekt entsprechend nicht mehr behandelt.

Abschließend werden die ILP-Gleichungen zur Berechnung der optimalen Belegung erzeugt und in eine Datei zur Verarbeitung durch CPLEX geschrieben. Dazu werden die gesammelten Informationen über die Zeitpunkte und Speicherobjekte entsprechend den Quantoren und Summenzeichen der ILP-Gleichungen durchlaufen. Die Scratchpad-Speichergröße und die replatzierende oder blockierende Methode sind an dieser Stelle frei wählbar.

CPLEX erzeugt die Lösung der Gleichungen und schreibt sie in eine Protokolldatei. Damit stehen die vom `positioningILPgenerator` zu jedem Zeitpunkt zu platzierenden Speicherobjekte fest. Um die ILP-Gleichungen zur Platzierung für die Strategie mit blockierenden Objekten zu erzeugen, reichen die in der Lösung des Zuweisungsproblems enthaltenen Informationen, zusammen mit der Größe des Scratchpad-Speichers, aus. Wenn die ILP-Gleichungen für die Strategie mit Replatzierung der Speicherobjekte generiert werden sollen, muss zusätzlich zu jedem Zeitpunkt bekannt sein, welche Prozesse aktiv und welche Objekte benutzt sind und replatziert werden müssen. Diese Informationen werden, wie beim `assignmentILPgenerator` beschrieben, aus der Profildatei extrahiert. Die ILP-Gleichungen des Positionierungsproblems können dann generiert und durch CPLEX gelöst werden. Die Lösung wird in eine zweite Protokolldatei geschrieben.

Der `lplug2header` genannte Konverter wandelt beide von CPLEX erzeugten Protokolldateien in eine C-Header-Datei um. Diese Datei übernimmt, wie in Abschnitt 4.4.1 ab Seite 75 beschrieben, die Steuerung des SPMM bei Benutzung ILP-basierter Lösungen. Aus der Lösung des Belegungsproblems werden die zu platzierenden Objekte eingelesen. Diese Informationen sind auch in der Protokolldatei des Platzierungsproblems enthalten, jedoch ist die Verarbeitung beim gewählten Weg weniger aufwändig. Anschließend

wird für jedes der zu platzierenden Objekte aus der Lösung des Platzierungsproblems die zu verwendende Zieladresse extrahiert. Der Konverter kann damit die Belegung und die Positionierung der Speicherobjekte in die Zielfeile schreiben. Zusätzlich werden noch die Zeitpunkte der Prozeßwechsel vom SPMM benötigt, wenn eine ILP-basierte Lösung zur Scratchpad-Belegung verwendet werden soll. Die Zeitpunkte werden von dem Konverter aus der Profildatei in den Header übertragen.

Abbildung 4.1 fasst den Arbeitsablauf zusammen. Das untere gestrichelte Rechteck umfasst den Teil, der zur Erzeugung optimaler Lösungen zusätzlich nötig ist. Die Laufzeitlösung wird von den Bibliotheken in dem anderen gestrichelten Kasten erzeugt.

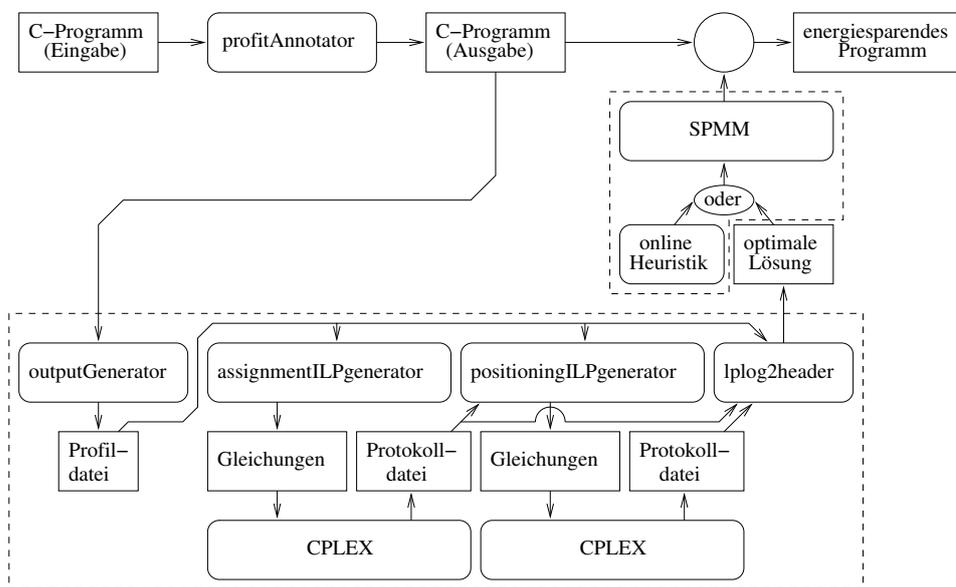


Abbildung 4.1: Darstellung des Arbeitsablaufes

## 4.2 Schnittstellenbeschreibung des SPMM

Der SPMM stellt Anwendungsprogrammen eine Programmierschnittstelle (Application Programming Interface, kurz API) zur Verfügung. Sie kann, wie im vorangegangenen Abschnitt beschrieben, teilautomatisiert oder manuell genutzt werden. Im Folgenden wird diese API beschrieben. Dabei wird zuerst die grundlegende Datenstruktur des SPMM-Objektes eingeführt, dann werden die Funktionen des SPMM vorgestellt.

SPMM-Objekte werden durch den Typ `mem_obj` repräsentiert. An dieser Stelle werden die von den Belegungsstrategien zusätzlich genutzten Felder des SPMM-Objektes, ebenso wie die zur Bildung der Verwaltungsstrukturen des SPMM nötigen Referenzen, nicht dargestellt. Der Typ kann nicht direkt

mit `typedef` vereinbart werden, da die genannten Referenzen durch Verweise auf Objekte gleichen Typs in der Struktur gebildet werden. Die Verwaltung der SPMM-Objekte wird in Abschnitt 4.4 auf Seite 69 behandelt. Auf die von den Belegungsstrategien abhängigen Elemente wird in den Abschnitten 4.4.1 und 4.4.2 ab Seite 75 eingegangen.

```
struct mem_obj
{
    uint size;
    uint profit;
    task_id owner;
    uint locked;
    uint write_back;
    void * cur_addr;
    void * mm_addr;
    [...]
};

typedef struct mem_obj mem_obj;
```

Die Größe des zum SPMM-Objekt gehörenden Speicherobjektes wird in `size` abgelegt. Der mögliche Nutzen wird in der Variable `profit` gehalten. Die `task_id` des Prozesses, zu dem das Speicherobjekt gehört, ist im Element `owner` abgelegt. Der Nutzungszähler verwendet die Variable `locked` des SPMM-Objektes. Wenn das Speicherobjekt beim Entfernen aus dem Scratchpad-Speicher in den Hauptspeicher zurückgeschrieben werden soll, ist `write_back` 1, sonst 0. Der Zeiger `cur_addr` zeigt auf die aktuelle Position des Speicherobjektes, an der darauf zugegriffen werden soll. Wenn das Objekt im Scratchpad-Speicher liegt, zeigt `cur_addr` dort hin. Liegt es nicht darin, zeigt `cur_addr` auf die Adresse des Objektes im Hauptspeicher, auf die immer auch `mm_addr` zeigt.

```
void spmm_init();
```

Die Funktion `spmm_init` verschiebt die Kopierfunktion `copy_spm` in den Scratchpad-Speicher und initialisiert die vom SPMM benötigten Datenstrukturen. Welche das im Einzelnen sind, ist von der gewählten Implementierung abhängig und wird in Abschnitt 4.4 ab Seite 69 beschrieben. `spmm_init` muss einmal aufgerufen werden, bevor der SPMM von Programmen benutzt werden kann.

```
void copy_spm(uint * src, uint * dest, uint len);
```

Die Kopierfunktion `copy_spm` wird vom SPMM verwendet, um Speicherobjekte zwischen dem Scratchpad und dem Hauptspeicher zu kopieren. Da die Kopierfunktion relativ zu dem überwiegenden Teil der Speicherobjekte klein ist und häufig darauf zugegriffen wird, ist die Kopierfunktion selber

ein ideales Objekt zur Verschiebung in den Scratchpad-Speicher. Die Kopierfunktion wird daher statisch im Scratchpad-Speicher abgelegt. Damit wird die Laufzeit der Kopiervorgänge stark verkürzt, was sich positiv auf den gesamten Zeitbedarf des SPMM auswirkt. Auch der Energiebedarf der Kopiervorgänge wird so verringert. Dadurch wird der Spielraum für Objektverlagerungen durch die Belegungsstrategien des SPMM erheblich erweitert, was das Energiesparpotential vergrößert.

Die Kopierfunktion kopiert einen Datenblock von `1en` 32 Bit-Worten vom Quellzeiger `src` zum Zielzeiger `dest`. Quelle und Ziel müssen an Wortgrenzen liegen. Beim Kopieren werden `Load-multiple`- und `Store-multiple`-Instruktionen benutzt, die bis zu acht Worte gleichzeitig bewegen.

```
mem_obj * spmalloc(uint size, uint profit);
```

Mit Hilfe der Funktion `spmalloc` können Anwendungsprogramme SPMM-Objekte zusammen mit einem dynamischen Speicherobjekt der angegebenen Größe `size` anfordern. Der mögliche Nutzen des Objektes wird in `profit` angegeben. Das SPMM-Objekt und das zugehörige Speicherobjekt werden im Hauptspeicher angelegt. Größe und Nutzen werden im SPMM-Objekt ebenso gespeichert wie der Zeiger auf das Speicherobjekt im Hauptspeicher.

Bevor Änderungen an den Verwaltungsstrukturen erfolgen, müssen nun Maßnahmen zu deren Schutz für den Fall eines Prozesswechsels getroffen werden. Die hierzu gewählte Lösung wird in Abschnitt 4.4 ab Seite 74 beschrieben. Im Anschluß daran wird das neue SPMM-Objekt der Verwaltungsstruktur hinzugefügt und die Belegungsstrategie auf den dann vorhandenen Objekten ausgeführt. Dabei wird berücksichtigt, dass für das neu angelegte Speicherobjekt keine Kopierkosten bei Verschiebung in den Scratchpad-Speicher anfallen. Wird das Objekt in das Scratchpad verschoben, wird nur der Zeiger auf die aktuelle Position des Speicherobjektes angepasst. Wenn es nicht verschoben wird, wird dieser Zeiger ebenfalls auf den Speicherbereich im Hauptspeicher gesetzt.

Dann werden die Verwaltungsstrukturen wieder freigegeben und abschließend der Zeiger auf das SPMM-Objekt zurückgegeben.

```
void spfree(mem_obj * ptr);
```

SPMM-Objekte, die zu dynamischen Speicherobjekten gehören und mit `spmalloc` erzeugt wurden, können mit `spfree` wieder gelöscht werden. Nach dem Schutz der Verwaltungsstrukturen des SPMM gegen Inkonsistenzen wird das SPMM-Objekt, auf das der Verweis `ptr` zeigt, aus der Verwaltungsstruktur entfernt. Dabei werden sowohl vom eigentlichen Speicherobjekt, als auch vom SPMM-Objekt, verwendete Bereiche des Hauptspeichers, wie auch eventuell belegter Scratchpad-Speicher, wieder freigegeben.

Die Belegungsstrategie wird auf den veränderten SPMM-Objekten aufgerufen, wodurch andere Speicherobjekte eventuell freigewordenen Platz im Scratchpad nutzen können. Abschließend wird der Schutz der Verwaltungsstrukturen wieder entfernt.

```
void splock(mem_obj * ptr);
```

Die Prozesse müssen ihre Speicherobjekte als benutzt markieren, damit die Belegungsverfahren wissen, welche Objekte bewegt werden dürfen und welche nicht. Die Funktion `splock` markiert das Objekt, auf das `ptr` zeigt. Da Speicherobjekte im Pfad rekursiver Aufrufe liegen können, wird ein Benutzungszähler (counting semaphore) benutzt. Die Anzahl der Benutzungsebenen wird so exakt festgehalten, jedoch ist aus Geschwindigkeitsgründen kein Schutz gegen Überläufe des Zählers implementiert. Bei korrekter Benutzung sind Überläufe des 32 Bit breiten Zählers unwahrscheinlich.

Ein Schutz der Verwaltungsstrukturen des SPMM ist nicht nötig, da sie nicht verändert werden, sondern lediglich das eine manipulierte SPMM-Objekt. Das SPMM-Objekt gilt als benutzt, sobald ein Wert ungleich null im Benutzungszähler steht. Unterbrechungen durch andere Prozesse sind bei dieser Funktion unkritisch, da der Prozess, der `splock` aufruft, erst auf den Zeiger auf Speicherobjekt zugreifen darf, wenn `splock` beendet wurde.

```
void sprelease(mem_obj * ptr);
```

Der Benutzungszähler des durch den Zeiger `ptr` angegebenen Speicherobjektes wird von `sprelease` um eine Stufe verringert. Erst wenn der Benutzungszähler vollständig abgebaut ist, darf das Objekt wieder verschoben werden. Es gibt keinen Schutzmechanismus gegen Überläufe des Zählers nach unten, da die hierzu nötigen Operationen die benötigte Rechenzeit extrem erhöhen.

Auch hier sind wie bei `splock` keine Schutzmaßnahmen gegen Unterbrechungen nötig.

```
void spglobal(mem_obj * ptr);
```

Statische Speicherobjekte werden mit Hilfe der Funktion `spglobal` beim SPMM angemeldet. Der Parameter `ptr` ist dabei ein Verweis auf das bereits existierende, anzumeldende SPMM-Objekt. Diese Objekte können manuell oder durch den im vorherigen Abschnitt beschriebenen `profitAnnotator` angelegt werden. Die Funktion `spglobal` wird vor dem Start eines Prozesses benutzt, um dessen Speicherobjekte beim SPMM anzumelden. Daher kann der Prozess, zu dem das neu angemeldete Speicherobjekt gehört, nicht aktiv sein, weshalb das neu hinzugekommene Objekt noch nicht im Scratchpad-Speicher platziert werden muss. Deshalb wird die Belegungsstrategie nicht aufgerufen.

Um das SPMM-Objekt beim SPMM anzumelden, muss es lediglich in die Verwaltungsstrukturen des SPMM eingefügt werden. Dabei müssen selbige für den entsprechenden Zeitraum vor Inkonsistenzen durch Prozesswechsel geschützt werden, wenn bereits Prozesse gestartet wurden, die den SPMM benutzen.

```
void spdrop(task_id id);
```

Wenn Prozesse enden, werden alle statischen Objekte dieses Prozesses nicht mehr benutzt. Sie müssen nicht länger vom SPMM verwaltet werden und können gegebenenfalls aus dem Scratchpad-Speicher entfernt werden. Dazu wird die Funktion `spdrop` verwendet. Alle statischen SPMM-Objekte, die zu Prozess `id` gehören, werden aus den Verwaltungsstrukturen des SPMM entfernt. Gleichzeitig wird von den zugehörigen Speicherobjekten belegter Scratchpad-Speicher freigegeben. Die Verwaltungsstrukturen werden während dieser Änderungen gegen weitere Zugriffe durch andere Prozesse und damit einhergehende Inkonsistenzen geschützt.

```
void sptask(task_id old_task, task_id new_task);
```

Der SPMM muss im Scheduler verankert werden, damit bei jedem Prozesswechsel das Scratchpad für den nächsten aktiven Prozess vorbereitet werden kann. Zu diesem Zweck wird der Scheduler um einen Aufruf der Funktion `sptask` erweitert. Dabei müssen der alte und der neue Prozess, zum Beispiel in Form der jeweiligen `task_id` oder ähnlichem, übergeben werden. Der nächste Prozess, für den das Scratchpad belegt werden soll, wird in `new_task` angegeben. Auf die Verwendung des Parameters `old_task` wird in Abschnitt 4.4.2 auf Seite 77 eingegangen.

Der Schutz der Verwaltungsstrukturen des SPMM gegen Inkonsistenz durch Prozesswechsel muss auch hier gewährleistet werden, bevor die Belegungsstrategie aufgerufen wird. Alle Heuristiken ausser der Statischen legen die Speicherobjekte des aktiven Prozesses bevorzugt im Scratchpad ab, da nur der aktive Prozess auf seine Objekte zugreifen kann. Hierzu verwenden die Heuristiken `new_task`. Anschließend werden die Verwaltungsstrukturen wieder frei gegeben.

### 4.3 Erweiterung der Prozesserzeugung

Die Prozesserzeugung des Betriebssystems muss erweitert werden, damit neue Prozesse ihre statischen Speicherobjekte beim SPMM automatisch anmelden können. Die hierzu verwendeten Strukturen und Funktionen werden im Folgenden beschrieben.

Zu jedem Prozess wird zur Compilezeit ein `task_init_block` angelegt. Darin sind alle zur Erzeugung des Prozesses nötigen Informationen enthal-

ten. Welche Informationen benötigt werden, ist vom verwendeten Betriebssystem abhängig. Eine `task_id`, die anfängliche Priorität, die zu verwendende Stackgröße, so wie weitere betriebssystemspezifische Attribute gehören üblicherweise dazu. Diese Parameter werden vom Betriebssystem verlangt und haben nichts mit dem SPMM zu tun. Zusätzlich jedoch wird der Zeiger `globals` auf ein Feld von Zeigern auf alle statischen Speicherobjekte des Prozesses angegeben. Ein `NULL`-Zeiger markiert das Ende des Feldes. Dieses Feld kann manuell angelegt oder vom `profitAnnotator` automatisch erzeugt werden.

```
int start_task(task_init_block * tib);
```

Mit einem Aufruf der Funktion `start_task` wird der durch den Zeiger `tib` beschriebene Prozess erzeugt und gestartet. Zunächst wird der Prozess durch das Betriebssystem erstellt. Danach wird das Feld der Zeiger auf die statischen Speicherobjekte des Prozesses durchlaufen, bis die Markierung am Feldende gefunden wird. Jedes dieser statischen Objekte wird durch `spglobal` beim SPMM angemeldet. Anschließend wird der neue Prozess ausführbereit gemacht. Dabei kann der aktuelle Prozess unterbrochen werden, wenn der neu erschaffene Prozess eine höhere Priorität hat als der, der momentan `start_task` ausführt.

Die Funktion gibt einen kodierten Fehlerstatus zurück. Ein Rückgabewert von 0 bedeutet, dass kein Fehler auftrat, der neue Prozess erzeugt wurde und nun auf seine Ausführung wartet. Fehler können zum Beispiel auftreten, wenn der Stack des Prozesses aus Speichermangel nicht angelegt werden kann. Alle möglichen Fehler treten auf, bevor die statischen Speicherobjekte beim SPMM angemeldet werden. Die Verwaltungsstrukturen werden daher von Fehlern nicht beschädigt.

```
void Init();
```

In eingebetteten Systemen werden nach dem Start des Betriebssystemkerns automatisch diejenigen Prozesse gestartet, die die Aufgaben des Systems erfüllen sollen. Prozesse, die den SPMM nicht benutzen, können weiterhin nach Belieben gestartet werden, solange sie keinen Scratchpad-Speicher verwenden. Prozesse, die den SPMM benutzen, dürfen aber erst gestartet werden, nachdem `spmm_init` ausgeführt wurde. Um die Initialisierung des SPMM und den Start der Prozesse zu vereinfachen, wird am Ende der Startprozedur des Betriebssystems die Funktion `Init` aufgerufen. Sie sorgt zunächst dafür, dass sie nicht unterbrochen wird, indem sie sich selbst die höchstmögliche Priorität zuordnet. Danach ruft sie `spmm_init` auf und bereitet so den SPMM zur Verwendung durch Applikationsprozesse vor. Anschließend wird das `NULL`-terminierte Feld `startup` durchlaufen, dass für jeden zu startenden Prozess einen Zeiger auf den entsprechenden `task_init_block` enthält. Für

jeden der Zeiger wird `start_task` mit dem Zeiger als Argument aufgerufen. Keiner der gestarteten Prozesse kommt dabei sofort zur Ausführung, da die höchste Priorität bereits für den aufrufenden Prozess genutzt wird. Sind alle Prozesse erzeugt, wird der aufrufende Prozess beendet. Der Scheduler sorgt dann dafür, dass alle anderen Prozesse ausgeführt werden.

Ein Multiprozesssystem kann somit automatisch gestartet werden und den SPMM benutzen, indem es alle Prozesse im Feld `startup` angibt. Dabei wird auf je einen `task_init_block` verwiesen, der wiederum die Zeiger auf die SPMM-Objekte enthält.

Darüber hinaus ist es natürlich auch möglich, die Prozesse explizit einzeln mit `start_task` anzulegen und damit nur die Anmeldung der SPMM-Objekte zu automatisieren.

Eine dritte Möglichkeit der Prozesserzeugung ist es, den Prozess manuell beim Betriebssystem zu erzeugen, dann die statischen SPMM-Objekte mittels `spglobal` anzumelden und schließlich den Prozess zu starten.

```
void stop_task(task_id id);
```

Die Funktion `stop_task` vereinfacht das Beenden von Prozessen. Es werden alle zum Prozess `id` gehörenden SPMM-Objekte mittels `spdrop` aus den Verwaltungsstrukturen des SPMM und die zugehörigen Speicherobjekte aus dem Scratchpad-Speicher entfernt, bevor der Prozess beendet wird.

Zur Vereinfachung der Simulationsvorgänge wird MPARM beendet, sobald keine Prozesse mehr ausgeführt werden.

## 4.4 Implementierung des SPMM

Die Implementierung des SPMM beeinflusst neben den bisher beschriebenen Designentscheidungen die Effizienz des SPMM maßgeblich.

Das verwendete Kosten- und Nutzenmaß ist eine dieser Implementierungsentscheidungen. Da die Energie optimiert werden soll, lassen sich die Kosten von Kopieroperationen und der Nutzen von Verlagerungen der Objekte in den Scratchpad-Speicher in Joule, der Einheit der Energie, ausdrücken. Der jeweilige Wert der Kosten beziehungsweise des Nutzens lässt sich aus der, von MPARM erzeugten, Trace-Datei mit Hilfe des bereits beschriebenen Energiemodells des Simulators berechnen. In der Trace-Datei sind alle Speicherzugriffe notiert. Alle weiteren Parameter, die Zugriffszeiten der verschiedenen Speicherbereiche, die resultierenden Prozessorzustände und die Energieverbrauchswerte dieser Zustände und der einzelnen Zugriffe, sind ebenfalls bekannt.

Die Kopierkosten wurden für alle von MPARM unterstützten Scratchpad-Kapazitäten und alle möglichen Objektgrößen für die Richtungen vom Hauptspeicher in den Scratchpad-Speicher und zurück vorberechnet und in Tabellenform abgelegt. Der SPMM kann die Kopierkosten daraus in konstanter

Zeit ablesen. Auch bei der Erzeugung der ILP-Gleichungen kommen diese Tabellen zum Einsatz.

Die mögliche Energieersparnis der Objekte werden durch den profit-annotator auf dem gleichen Weg ermittelt. Sie ergeben sich als Differenz des Energieverbrauchs für die Zugriffe auf das Objekt, wenn es im Hauptspeicher liegt, und des Verbrauchs, wenn das Objekt sich im Scratchpad befindet. Über die Verteilung der Zugriffe auf das Speicherobjekt innerhalb der Ausführungszeit des Prozesses können zur Laufzeit keine Angaben ausgewertet werden. Deshalb wird hierfür eine Gleichverteilung angenommen. So erhält man die Anzahl der Zugriffe zwischen zwei Aufrufen der Belegungsfunktion, indem man die Summe aller Aufrufe durch die Anzahl der betrachteten Zeitpunkte teilt, zu denen das Speicherobjekt existiert. Entsprechend läßt sich auch die mögliche Energieersparnis mittels Division durch die Anzahl der Zeitpunkte in eine durchschnittliche Energieersparnis umwandeln, die hier als Nutzenmaß verwendet wird. So werden lang- und kurzlebige Objekte zu jedem einzelnen Aufrufzeitpunkt der Belegungsstrategie miteinander vergleichbar ohne eine globale Sicht zu benötigen. Für die heuristischen Ansätze ist dies unerlässlich, wie Abbildung 4.2 verdeutlicht.

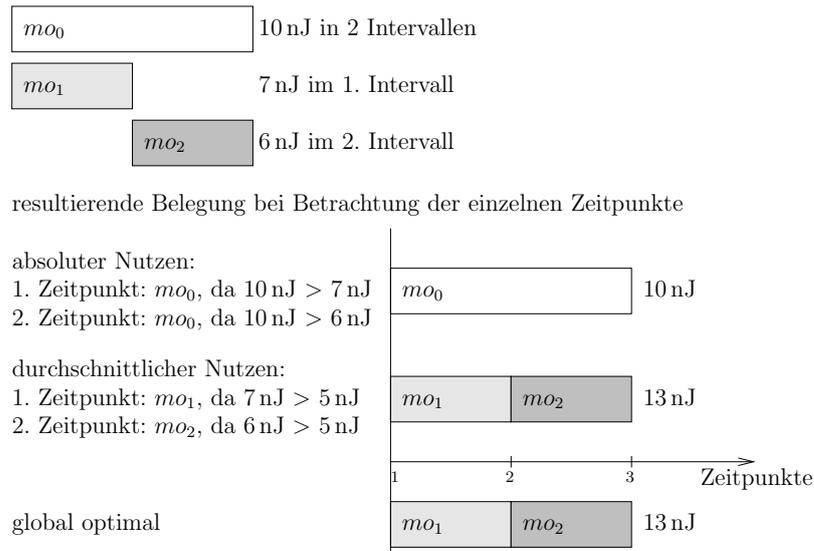


Abbildung 4.2: Vergleich der lokalen Entscheidungen bei Verwendung des absoluten beziehungsweise des durchschnittlichen Nutzenmaßes

Durch die Wahl des Nutzenmaßes sollen nur Objekte in den Scratchpad-Speicher kopiert werden, für die es sich in jedem betrachteten Zeitintervall lohnt, die Kopierkosten zu investieren. Aufgrund der insgesamt niedrigen Kopierkosten ist es nicht nötig, dieses Nutzenmaß für die statische Belegungsstrategie anzupassen, obwohl dort die Kopierkosten nur einmalig auftreten.

Der SPMM muss zu jeder Zeit auf alle SPMM-Objekte zugreifen können, um daraus diejenigen auszuwählen, die in den Scratchpad-Speicher verlagert werden sollen. Da SPMM-Objekte zur Laufzeit erzeugt und gelöscht werden können, muss eine Datenstruktur verwendet werden, die das Einfügen und Entfernen von Objekten unterstützt. Die Belegungsverfahren sind darauf angewiesen, dass auf die SPMM-Objekte sequentiell zugegriffen werden kann. Deshalb werden die SPMM-Objekte in doppelt verketteten Listen gehalten. Operationen zum Einfügen und Entfernen sind in konstanter Zeit möglich, ebenso wie die Bestimmung des nächsten Objektes bei sequentiellen Zugriffen. Die Verwendung einfach verlinkter Listen erhöht den Aufwand von Operationen zum Löschen von SPMM-Objekten erheblich, weil trotz gegebenem Objekt kein Verweis auf das vorhergehende Objekt in der Liste vorhanden ist.

```
struct mem_obj
{
    [...]
    struct mem_obj * next;
    struct mem_obj * prev;
    [...]
};
```

Die `mem_obj`-Datenstruktur aus Abschnitt 4.2 von Seite 62 wird deshalb um die Elemente `next` und `prev` erweitert, die auf das nächste beziehungsweise vorhergehende Objekt in der Liste verweisen.

Sowohl die freien als auch die belegten Bereiche des Scratchpad-Speichers müssen dem SPMM bekannt sein, damit einerseits Speicherobjekte in freie Bereiche verschoben werden können und andererseits belegte Bereiche freigegeben werden können, um Objekte im Scratchpad durch Andere zu ersetzen. Da so auch belegte Speicherbereiche durch Auskopieren des momentanen Inhaltes verwendet werden können, um neue Speicherobjekte aufzunehmen, ist es sinnvoll, belegte und freie Speicherbereiche in einer gemeinsamen Struktur zu halten.

In der Literatur[WJNB95] wurden verschiedene Ansätze zur Verwaltung von Speicherbereichen untersucht. Im Rahmen der vorliegenden Arbeit wurden drei Ansätze experimentell überprüft.

Der erste Ansatz verwendet eine doppelt verkettete, adresssortierte Liste aller Scratchpad-Speicherbereiche. Jedem Bereich wird ein Header vorangestellt.

```
struct spm_area
{
    uint16 prev;
    uint16 next;
    mem_obj * obj;
    /* Instruktionen oder Daten */
};
```

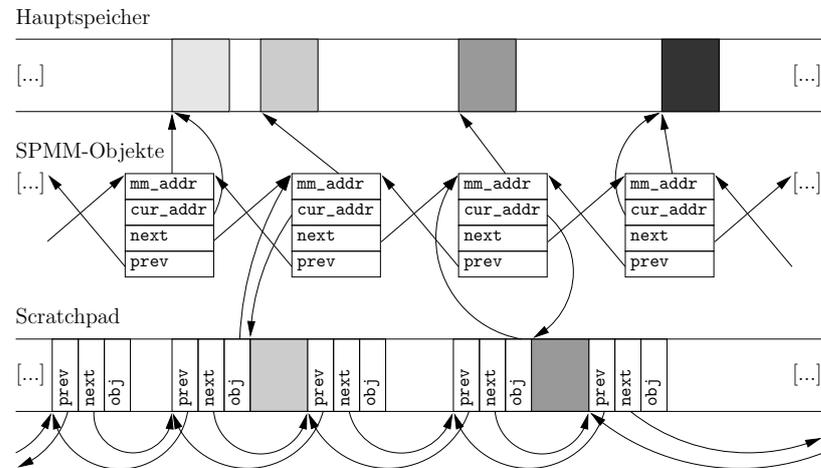


Abbildung 4.3: Zeigerstruktur der SPM-Objekte und der Scratchpad-Speicherbereiche bei Verwendung der Listenstruktur

Da Scratchpad-Speicher relativ klein sind, genügen jeweils 16 Bit, um zusammen mit der bekannten Startadresse des Scratchpad-Speichers die Position des vorherigen Speicherbereiches durch `prev` beziehungsweise des Nachfolgenden durch `next` anzugeben. Da die Bereiche an Wortgrenzen ausgerichtet sind, genügt diese Implementierung für Scratchpad-Speicher mit bis zu 256 kB Kapazität. Der `prev`-Zeiger des ersten Bereiches zeigt auf den Anfang des Scratchpad-Speichers. Der `next`-Zeiger des letzten Bereiches zeigt auf das Ende des Scratchpads. Jeder Scratchpad-Speicherbereich enthält einen Zeiger auf das zugehörige SPM-Objekt, wodurch zu jedem belegten Bereich beispielsweise die Adresse des Speicherobjektes zum Zurückkopieren in den Hauptspeicher ermittelt werden kann. Für freie Speicherbereiche enthält der `obj`-Zeiger den Wert NULL. Die eigentlichen Daten eines belegten Speicherbereiches schließen sich an die Verwaltungsverweise an. Damit müssen Speicherbereiche aus mindestens 3 32 Bit Worten bestehen und können eine beliebige Größe bis zu der des Scratchpad-Speichers annehmen. Dazu werden freie, größere Speicherbereiche aufgeteilt und einer der beiden Teile verwendet. Passende Bereiche können sofort verwendet werden. Wenn beim Aufteilen eines Speicherbereiches ein Bereich kleiner als die Mindestgröße entsteht, wird statt dessen der ganze Bereich zugeteilt. Wenn ein belegter Bereich freigegeben wird, wird er mit angrenzenden freien Bereichen zusammengefasst und bildet einen Speicherbereich maximaler Größe. Durch die gemeinsame Verwaltung freier und belegter Speicherbereiche werden Aufspalten und Zusammenfassen von Bereichen erheblich beschleunigt, da vorhergehende und folgende Bereiche in konstanter Zeit gefunden werden.

Der zweite Ansatz verwendet mehrere Listen, um für zu verschiebende Objekte schneller geeignete Zielbereiche zu finden. Die verwendete Daten-

struktur entspricht Skiplisten. Die Listen sind nach der Mindestgröße der Speicherbereiche sortiert. Innerhalb der Listen werden die Bereiche nach Adressen sortiert. Jede der Listen enthält Speicherbereiche, die mindestens die vorgegebene Größe haben. Je größer der Bereich ist, in desto mehr Listen ist er enthalten. Wenn ein Zielbereich gesucht wird, dann wird in der Liste der kleinsten passenden Bereiche nach einem geeigneten Ziel gesucht. Im idealen Fall, einer Bereichsliste für jede Objektgröße, wird eine Address-Ordered-First-Fit-Strategie realisiert. Da die Listenzeiger Platz benötigen, können nur begrenzt viele Listen realisiert werden. In der verwendeten Implementierung werden daher Listen für die Größen von  $2^0$  und  $2^3$  bis  $2^8$  Worten benutzt. Wieder werden 16 Bit breite Zeiger verwendet, die von jedem Bereich aus auf den Vorherigen und den nächsten Bereich zeigen. Zusätzlich wird ein Zeiger auf das zugehörige Speicherobjekt belegter Bereiche verwendet. Insgesamt werden also 8 32 Bit-Wort große Header jedem Speicherbereich vorangestellt. Die Mindestgröße der Bereiche liegt bei 9 Worten, die Maximale gibt die Scratchpad-Größe vor. Die Mechanismen zum Vereinigen und Aufteilen von Speicherbereichen sind die gleichen wie unter Verwendung des ersten Ansatzes, jedoch um einiges komplexer. Der Aufbau der Listen wird in Abbildung 4.4 dargestellt. Abbildung 4.5 verdeutlicht den Verlauf der Suche nach einem freien Speicherbereich.

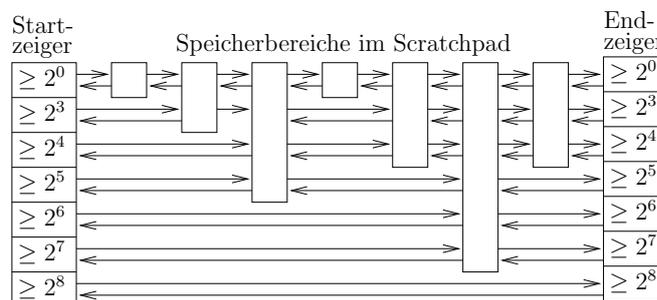
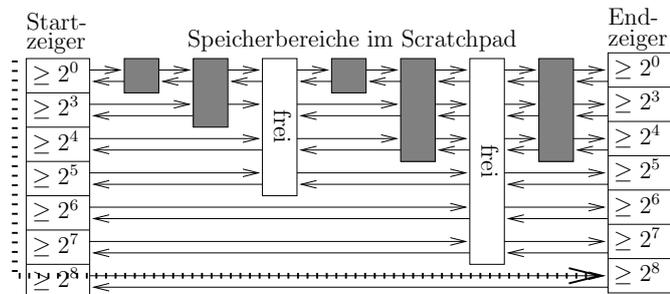


Abbildung 4.4: Aufbau der Skiplisten-Struktur

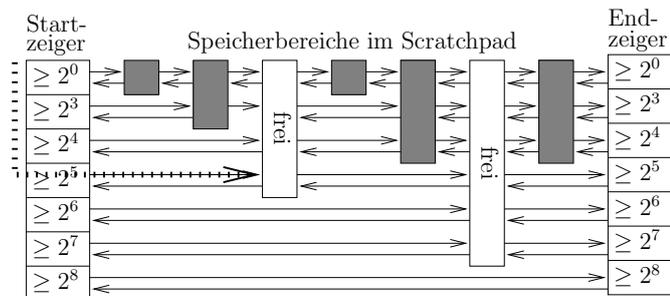
Da der Scratchpad-Speicher relativ klein ist, lohnen sich die zusätzlichen Suchstrukturen nicht. Der entstehende Overhead übersteigt den Nutzen, zudem geht zusätzlicher Scratchpad-Speicher verloren. Der Energiebedarf durch den SPMM optimierter Anwendungen ist unter Verwendung der Skiplisten etwa 10% größer als unter Benutzung des zuerst vorgestellten einfachen Listenansatzes.

Der dritte Ansatz implementiert ein binäres Buddy-System. Konzeptionell wird der gesamte Scratchpad-Speicher auf zwei gleich große Bereiche aufgeteilt, die jeweils immer wieder in zwei Bereiche unterteilt werden. Dabei entstehende Speicherbereiche jeder Unterteilungsebene können den Speicherobjekten zugewiesen werden. Angrenzende Bereiche werden sofort verschmolzen, wenn sie vollständig unbesetzt sind und durch die gleiche Unterteilung



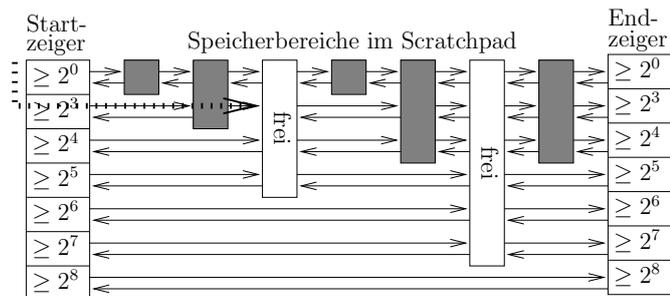
Verlauf der Suche.....

(a) Suche nach freiem Bereich von 200 Worten



Verlauf der Suche.....

(b) Suche nach freiem Bereich von 20 Worten



Verlauf der Suche.....

(c) Suche nach freiem Bereich von 7 Worten

Abbildung 4.5: Beispiele für verschiedene Suchvorgänge auf der Skiplisten-Struktur

entstanden sind. Die möglichen Speicherbereiche bilden einen vollständigen binären Baum, der im Hauptspeicher getrennt von den Speicherbereichen gelagert wird. Abgespeichert werden müssen alle Wege von der Wurzel des Baumes zu einem vollständig freien oder vollständig belegten Knoten. Unterteilungen unterhalb brauchen nicht betrachtet werden, daher bilden vollständig belegte beziehungsweise vollständig freie Knoten die Blätter des Baumes. An den inneren Knoten werden die Zustände der nachfolgenden Knoten festgehalten, also ob der linke oder der rechte Nachfolger belegt sind. Diese Informationen lassen sich zusammen mit dem Zeiger auf das belegende SPMM-Objekt wie in Abbildung 4.6 dargestellt in einem 32 Bit-Wort darstellen, weil die Adressen der SPMM-Objekte an Wortgrenzen ausgerichtet sind.

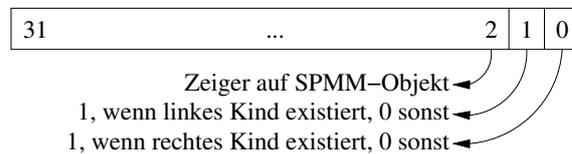


Abbildung 4.6: Aufbau eines Buddy-Baumknotens

Bei unbelegten Blättern ist der Zeiger NULL. Der Baum wird auf ein Feld abgebildet. In der Position jedes Knotens ist nicht nur seine Unterteilungsebene, sondern auch sein Elterknoten, mögliche Kinderknoten, die Startadresse des zugehörigen Bereiches im Scratchpad und sein Buddy, also der Knoten, der mit ihm zusammen durch Unterteilung entstand, kodiert.

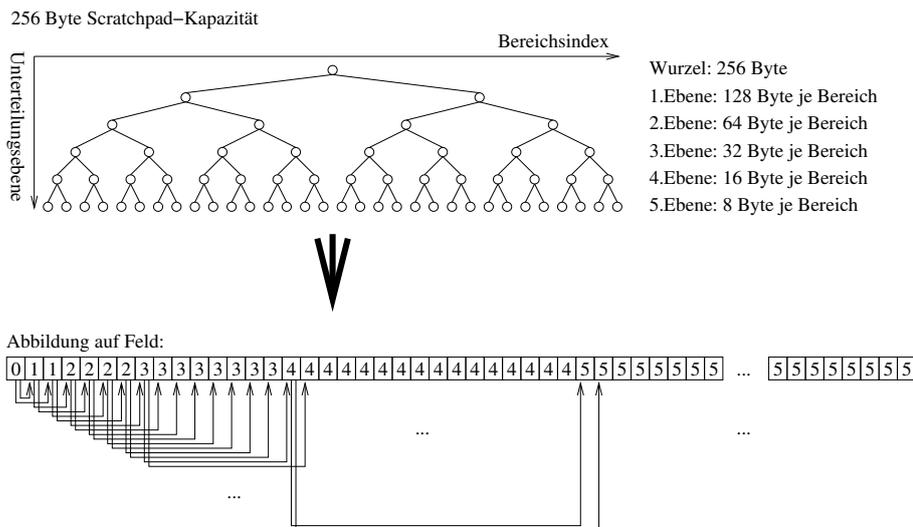


Abbildung 4.7: Abbildung des Baumes auf das Feld

Im Gegenzug wird jedem Speicherbereich die Angabe der zugehörigen Unterteilungsebene und des Bereichsindex in jeweils 16 Bit kodiert vorangestellt. Die Suche nach möglichen Zielbereichen wird durch Listen gleich großer Objekte unterstützt, die die Bereiche adresssortiert enthalten. Wenn kein geeigneter Zielbereich der kleinsten ausreichenden Größe für ein Speicherobjekt vorhanden ist, wird in den Listen der nächstgrößeren Bereiche nach einem geeigneten Bereich zum Unterteilen in zwei Bereiche der kleineren Größe gesucht. Beim Unterteilen und Zusammenfassen von Bereichen müssen unter Umständen logarithmisch viele Knoten zwischen dem Ort der Änderung und der Wurzel des Baumes manipuliert werden.

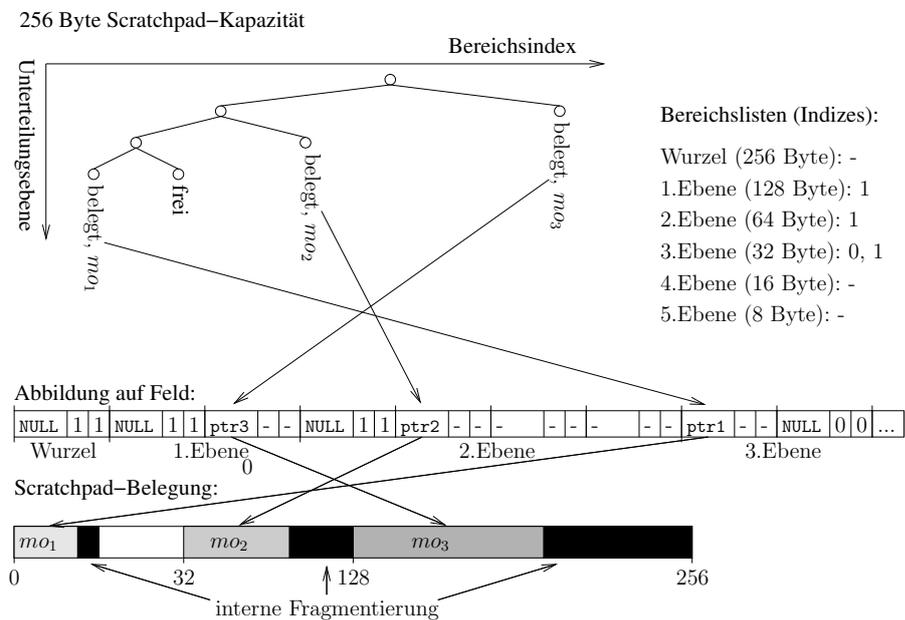


Abbildung 4.8: Beispiel für die Benutzung des Buddy-Systems

Abbildung 4.8 stellt die Benutzung des Buddy-Systems grafisch dar. Wie bereits aus vorhergehenden Arbeiten bekannt, leiden binäre Buddy-Systeme massiv an interner Fragmentierung. Da Scratchpad-Speicher klein sind, sind die Wirkungen der internen Fragmentierung umso verheerender. Die Fragmentierung lag bei den betrachteten Applikationen bei ungefähr 35%. Ein Vergleich der Energieersparnis mit dem ersten, listenbasierten Ansatz ist kaum möglich, weil aufgrund der Fragmentierung kaum Speicherobjekte in das Scratchpad verlagert werden.

Der erste, listenbasierte Ansatz zur Verwaltung der Speicherobjekte wird im Folgenden für alle Heuristiken eingesetzt, weil die damit beobachtete Energieersparnis am größten ist.

Wie in Abschnitt 3.1 auf Seite 32 beschrieben, können die Datenstrukturen des SPMM durch Semaphoren oder durch das Aussetzen der Inter-

raptverarbeitung vor Inkonsistenzen geschützt werden. In MPARM sind neben dem Timer und dem Interruptcontroller keine weiteren Interruptquellen vorhanden. Der Interruptcontroller wird nur in Multiprozessorumgebungen verwendet und wird daher bei der vorliegenden Arbeit nicht betrachtet. Der Timer generiert für den Scheduler das Signal, einen Taskwechsel einzuleiten. In der vorliegenden Arbeit verwenden alle Prozesse den SPMM. Deshalb greift der Scheduler bei jedem Aufruf auf SPMM-Funktionen zu, um das Scratchpad für den nächsten Prozess vorzubereiten. Diese SPMM-Funktionen benötigen die Datenstrukturen des SPMM. Wenn diese durch Semaphoren geschützt sind und der Scheduler einen Prozess unterbricht, der die Semaphoren besitzt, würde es zum Deadlock kommen. Werden Verfahren zur Vermeidung von Deadlocks wie Priority Ceiling verwendet, so fällt zusätzlicher Aufwand für dessen Verwendung an. Daher ist in diesem Fall sinnvoll, die Datenstrukturen des SPMM durch ein temporäres Aussetzen der Interruptbehandlung zu schützen.

Um die Interruptverarbeitung auszusetzen, muss ein Flag des Timers gelöscht werden. Die Datenstrukturen des SPMM sind dann geschützt, denn auftretende Interrupts werden verzögert, bis das Flag gesetzt wird und die Datenstrukturen so wieder freigegeben sind. Anschließend wird der Interrupt genauso bearbeitet, als wäre er nicht verzögert worden.

#### 4.4.1 ILP-basierte Belegung

Die Übernahme von Scratchpad-Belegungen durch den SPMM aus der Lösung von ILP-Gleichungen erfolgt durch eines von zwei Verfahren.

Das erste Verfahren wurde auf geringen Speicherplatzbedarf optimiert. Für jeden Zeitpunkt wird für jedes Objekt in einem fortlaufenden Bitstring ein Bit reserviert. Ist das Bit 0, wird das Speicherobjekt im Hauptspeicher bleiben oder zurück in den Hauptspeicher verschoben. Wenn das Bit 1 ist, bleibt das Speicherobjekt im Scratchpad-Speicher oder wird dorthin verlagert. Im letzteren Fall wird zusätzlich die Zieladresse aus einer Adressliste gelesen. Die Reihenfolge der SPMM-Objekte muss mit der Reihenfolge der Bits im Bitstring und der Adressen übereinstimmen, damit die richtigen Objekte verschoben werden. Die Reihenfolge wird deshalb auf die Erstellungsreihenfolge der SPMM-Objekte in der durch den outputGenerator erzeugten Profildatei festgelegt.

Im Belegungsbitstring sind die Informationen über die ein- und auszukopierenden Speicherobjekte vermischt. Da erst alle Auskopiervorgänge abgeschlossen sein müssen, bevor neue Speicherobjekte einkopiert werden können, ist zusätzlicher Aufwand nötig. Entweder der Belegungsstring wird zweimal gelesen, wobei im ersten Durchlauf nur in den Hauptspeicher zurück kopiert wird. Dann wird die Position des Bitstrings zurück auf den anfänglichen Wert gesetzt und erneut durchlaufen. Im zweiten Durchlauf werden die neuen Speicherobjekte in den Scratchpad-Speicher kopiert. Alternativ

kann der Belegungsbitstring auch nur einmal gelesen werden und alle Kopieroperationen vom Hauptspeicher in das Scratchpad werden bis nach dem Durchlauf verzögert. Hierzu wird in der Struktur der SPMM-Objekte ein weiterer Zeiger, `copy_in_next`, verwendet, durch den eine Liste der einzukopierenden Speicherobjekte realisiert werden kann.

```
struct mem_obj
{
    [...]
    struct mem_obj * copy_in_next;
};
```

Das zweite Verfahren, um die ILP-Lösung zur Scratchpad-Belegung zu verwenden, ist vorteilhaft, wenn viele SPMM-Objekte vorhanden sind, aber nur wenige in den Scratchpad-Speicher verschoben werden können. Alle SPMM-Objekte werden in der Erstellungsreihenfolge der Profildatei in eine Tabelle übernommen. Die Größe der Tabelle kann ebenfalls der Profildatei entnommen werden. Die Belegung ist in einer Liste abgelegt. Für jeden Zeitpunkt werden zunächst die aus dem Scratchpad-Speicher zu entfernenden SPMM-Objekte anhand ihres Tabellenindex gefolgt von dem Trennzeichen (`0xFFFF FFFF`) angegeben. Dann werden die einzukopierenden SPMM-Objekte, zusammen mit der jeweiligen Zieladresse, angegeben. Abgeschlossen wird auch diese Informationseinheit durch das Trennzeichen.

Die Menge der verarbeiteten Worte ist bei diesem Verfahren größer, aber die Implementierung ist erheblich einfacher, so dass kaum Overhead entsteht und weniger Energie verbraucht wird als mit dem ersten Verfahren.

Die Aufrufe der Belegungsschritte und mit ihnen die Prozesswechsel müssen bei Verwendung einer ILP-basierten Lösung an den exakt gleichen Stellen im Programmablauf stattfinden, wie sie in der Profildatei auftreten, damit exakt die gleichen SPMM-Objekte blockiert sind. Andernfalls ist die Korrektheit des Programmes nicht gewährleistet. Da die Verschiebung von Speicherobjekten in den Scratchpad-Speicher die Ausführung des Programms aber beschleunigt, kann jede Belegungsentscheidung den Zeitpunkt verändern, für den diese Entscheidung getroffen werden soll. Damit verändert sich die Problemstellung mit jedem Belegungsschritt und eine Lösung wird weiter erschwert.

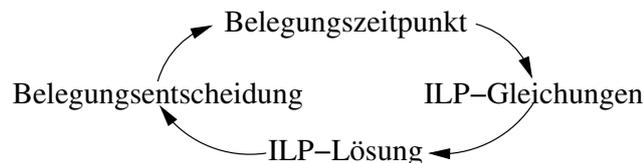


Abbildung 4.9: Einfluß der Komponenten des ILP-Verfahrens aufeinander

Diese Problematik wird umgangen, wenn die Prozesswechsel nicht mehr durch Timer, sondern durch die Prozesse selber ausgelöst werden. Die Prozesswechsel werden abhängig gemacht von der Anzahl der vorhergehenden Aufrufe der Funktionen `splock`, `sprelease`, `spmalloc`, `spfrees` und `sptask`. Ist die gleiche Anzahl der Aufrufe erreicht, die bei dem nächsten Prozesswechsel in der Profildatei eingetragen ist, wird im aktuellen Simulationslauf ebenfalls ein Prozesswechsel durchgeführt.

#### 4.4.2 Heuristische Belegung

Bei allen heuristischen Belegungsstrategien wird die SPMM-Objektstruktur um die Komponente `effect` erweitert. Darin wird die Effektivität abgelegt, um sie nicht ständig erneut berechnen zu müssen.

```
struct mem_obj
{
    [...]
    uint effect;
};
```

Hinzu kommt noch das Element `spm_addr`, in dem die letzte Adresse des Speicherobjektes im Scratchpad-Speicher abgelegt wird, wenn die Blockstrategie benutzt wird.

```
struct mem_obj
{
    [...]
    void * spm_addr;
    uint effect;
};
```

Die SPMM-Objekte werden in einer nach der Effektivität absteigend sortierten, doppelt verketteten Liste gehalten. Im Fall der statischen Heuristik werden die SPMM-Objekte aller Prozesse in einer gemeinsamen Liste verwaltet, weil die Belegung prozessübergreifend erzeugt werden muss, da sich die Objekte nicht gegenseitig verdrängen. Bei allen weiteren Strategien wird eine Liste der SPMM-Objekte je Prozess verwendet, weil bei jedem Belegungsschritt ausschließlich Speicherobjekte des aktiven Prozesses im Scratchpad platziert werden.

Die Funktion zum Auskopieren des gesamten Scratchpad-Speichers in den Puffer des vorherigen Prozesses beziehungsweise zum Einkopieren des Pufferinhaltes des neuen Prozesses in den Scratchpad-Speicher wird bei der dynamischen Belegungsstrategie in die Funktion `sptask` integriert. Der Puffer des alten Prozesses wird anhand des Parameters `old_task` identifiziert.

## 4.5 Integration in das Zielbetriebssystem

Grundsätzlich kann der SPMM auf verschiedene Arten in das Betriebssystem integriert werden. Beispielsweise kann man Teile des ursprünglichen Speichermanagements durch den SPMM ersetzen. Dazu integriert man die in Abschnitt 4.3 ab Seite 65 beschriebene Erweiterung der Prozesserzeugung in die entsprechende Funktion des Betriebssystems. Diese Variante ist sehr effizient, aber jede Änderung des SPMM erfordert, dass das Betriebssystem neu kompiliert wird. Zudem sind nur noch Programme lauffähig, die auf die Änderungen des Betriebssystems vorbereitet sind.

Alternativ lässt sich der SPMM als Bibliothek zusätzlich zu den Betriebssystemfunktionen verwenden. Diese zweite Variante ist geringfügig ineffizienter, da sie zusätzlich zu den Betriebssystemroutinen implementiert wird und diese teilweise aufruft. Die Kompatibilität zu anderer Software, die den SPMM nicht verwendet, ist vollständig gegeben. Bei Änderungen des SPMM müssen nur dieser und gegebenenfalls die Applikationen neu übersetzt werden.

Um diese Variante in ein Betriebssystem zu integrieren sind einige Grundlagen nötig.

- Die Erweiterung der Prozesserzeugung kann, wie bereits beschrieben, verwendet werden, wenn `start_task` und `stop_task` implementiert werden können.
- Ein Aufruf von `sptask` muss in den Scheduler integriert werden. Wenn dies zur Compilezeit erfolgt, müssen SPMM und Betriebssystem bei jeder Änderung des SPMM zusammen gelinkt werden.
- Der SPMM fordert dynamisch Speicher an, um die SPMM-Objekte anzulegen. Dazu wird die Unterstützung des Betriebssystems benötigt.

Im Folgenden wird beschrieben, wie der SPMM als Bibliothek mit den in den Abschnitten 2.3.1 bis 2.3.3 ab Seite 14 näher beschriebenen Betriebssystemen verwendet werden kann. Da keines der Systeme über Speicherschutz verfügt oder den Scratchpad-Speicher automatisch benutzt, kann der SPMM direkt, ohne Umweg über das Betriebssystem, darauf zugreifen. Eine weitergehende Integration in das Betriebssystem erfordert umfangreichere Änderungen, die aber an den gleichen Stellen ansetzen.

### 4.5.1 $\mu$ CLinux

Der Aufruf von `sptask` kann in der Hauptfunktion `schedule` des Schedulers, die sich in der Datei `linux-2.6.x/kernel/sched.c` befindet, vor dem Kontextwechsel durch die Quelltextzeile `context_switch(rq, prev, next)` erfolgen. Der Parameter `old_task`, der den vorherigen Prozess erwartet, wird mit `prev`, der Parameter des nächsten Prozesses `new_task` mit `next` belegt.

Unter  $\mu$ CLinux verwendet `start_task` die bereits angesprochene `fork/exec`-Kombination, um den Prozess für das Betriebssystem zu erzeugen. Um den Prozess zu beenden, verwendet `stop_task` die Funktion `exit`.

Der für die Verwaltungsstrukturen des SPMM nötige Hauptspeicher kann direkt bei dem  $\mu$ CLinux-Kernel angefordert werden.

### 4.5.2 eCos

Im Verzeichnis `kernel/v2.0/src/sched/` in den Dateien `bitmap.cxx`, `lottery.cxx` und `mlqueue.cxx` sind die verschiedenen, von eCos zur Verfügung gestellten, Scheduler zu finden. Die Schedulerfunktion `schedule` wird jeweils um einen Aufruf von `sptask` erweitert. Die Parameter `old_task` und `new_task` werden mit den Variablen `current` beziehungsweise `thread` belegt.

Unter eCos verwendet `start_task` die Betriebssystemfunktion `cyg_thread_create`, um einen neuen Prozess zu erzeugen. Mit `cyg_thread_resume` wird der Prozess dann gestartet. Der Prozess wird durch `cyg_thread_exit` beendet und aus dem System mittels `cyg_thread_delete` entfernt. Diese Aufgaben können von `stop_task` übernommen werden.

Da eCos nicht über eine Verwaltung dynamischen Speichers durch das Betriebssystem verfügt, muss, wie in Abschnitt 2.3.2 auf Seite 18 beschrieben, die C-Bibliothek benutzt werden, um Speicher für die SPMM-Objekte anzufordern.

### 4.5.3 RTEMS

Mit den User-Extension-Objekten bietet RTEMS die komfortabelste Anbindung des SPMM. Die Funktion `sptask` wird zur Laufzeit einfach als User-Extension-Objekt, das bei Prozesswechseln aufgerufen werden soll, eingebunden. Die Signatur von `sptask` passt genau auf diese Verwendung.

Um Prozesse unter RTEMS zu erzeugen, verwendet `start_task` die Funktion `rtems_task_create`. Der Prozess kann dann mit einem Aufruf von `rtems_task_start` gestartet werden. Die Funktion `stop_task` kann den Prozess durch Aufruf von `rtems_task_suspend` beenden und ihn durch `rtems_task_delete` aus dem Betriebssystem löschen.

Die SPMM-Objekte können unter Verwendung der vom Betriebssystem zur Verfügung gestellten Speicherallokationsfunktionen angelegt werden.

Der SPMM ist prototypisch unter Verwendung von RTEMS implementiert und integriert worden, weil nur unter RTEMS Änderungen am SPMM vorgenommen werden können, ohne das Betriebssystem neu zu übersetzen. Der Aufruf von `sptask` aus dem Scheduler heraus macht dies bei eCos und  $\mu$ CLinux notwendig, was die Entwicklung stark behindert. Zudem ist das Fehlen eines dynamischen Speichermanagements in eCos ein weiteres, entscheidendes Hindernis, denn die entsprechende Funktion der C-Bibliothek ist vergleichsweise ineffizient.



# Kapitel 5

## Ergebnisse

Die Ergebnisse der verschiedenen Scratchpad-Belegungsstrategien lassen sich anhand ihrer Energieersparnis und der benötigten Rechenzeit vergleichen. Dazu ist es jedoch nötig, eine Auswahl von Applikationen zu bestimmen, anhand derer die benötigten Größen ermittelt werden. Diese Applikationen werden zu Benchmarks zusammengefasst, die in Abschnitt 5.1 vorgestellt werden.

Mit Hilfe des MPARM wird der Einfluß der verschiedenen Scratchpad-Belegungsstrategien des SPMM auf Energiebedarf und Laufzeit der Benchmarks ermittelt und in Abschnitt 5.2 diskutiert. Die Heuristiken werden zunächst untereinander, dann mit den ILP-basierten, optimalen Lösungen verglichen.

Die Belegung des Scratchpad-Speichers zur Laufzeit bewirkt zusätzlichen Rechenaufwand, der die Ausführungszeit des Programms verlängert. Dadurch wird mehr Energie benötigt. Der SPMM kann Energie sparen, wenn die Energieersparnis durch die Verschiebung der Speicherobjekte in den Scratchpad-Speicher größer ist als der zusätzliche Energiebedarf. Auf diesen Trade-Off wird in Abschnitt 5.3 eingegangen.

Abgeschlossen wird das Kapitel mit einer Gegenüberstellung der Scratchpad-Belegung durch den SPMM und Caches in Abschnitt 5.4.

### 5.1 Beschreibung der Benchmarks

Als Benchmarks werden Multiprozesssysteme verwendet, bei denen mehrere Applikationen parallel ausgeführt werden. Die vier verwendeten Benchmarks sind:

**AUTO:** Dieser Benchmark besteht aus den Automotive- und Industrieapplikationen der MiBench[GRE<sup>+</sup>01] Benchmark-Suite. Sie umfasst `basicmath`, `bitcount`, `qsort` und `susan`, mit den Unterprogrammen `edges`, `corners` und `smooth`. Der `basicmath`-Test führt einfache mathematische Berechnungen, wie das Lösen quadratischer Gleichungen, die

Bestimmung von Quadratwurzeln und Konvertierungen von Grad nach Bogenmaß durch. Mit `bitcount` werden die Bitmanipulationsfähigkeiten des Prozessors mit Hilfe von mehreren Algorithmen zum Zählen von Einsen in Bitstrings überprüft. Der `qsort`-Test sortiert ein Datenfeld in aufsteigender Ordnung. Das Bilderkennungspaket `susan` besteht aus Teilprogrammen zur Kantenerkennung (`edges`), Eckenbestimmung (`corners`) und zum Weichzeichnen (`smooth`).

**TELECOM:** Die Programme `CRC32`, `FFT`, `IFFT`, sowie `ADPCM` und `GSM`, jeweils im Kodier- und Dekodierbetrieb, werden zu dem TELECOM genannten Multiprozesssystem zusammengefasst. Die Programme bilden zusammen den Telekommunikationsbenchmark der MiBench Benchmark-Suite. Das Testprogramm `CRC32` führt einen Checksummentest nach dem 32 Bit CRC-Verfahren (Cyclic-Redundancy-Check) durch. Mit `FFT` wird die Fast-Fourier-Transformation berechnet. Durch `IFFT` wird die inverse Fouriertransformation bestimmt. Mit `ADPCM` (Adaptive Differential Pulse Code Modulation) werden Audiodaten mit einer festen Kompressionsrate von 4:1 kodiert beziehungsweise dekodiert. `GSM` kodiert beziehungsweise dekodiert einen Sprachdatenstrom nach dem GSM 06.10 Standard. In Europa und vielen anderen Ländern wird GSM (Global System for Mobile communications) in kommerziellen Mobilfunknetzen verwendet.

**MEDIA:** Dieses Multiprozesssystem besteht in der ersten Variante `MEDIA-` neben dem bereits in TELECOM verwendeten Audiodatenkodierer und -dekodierer `ADPCM` aus den Teilprogrammen `edge-detection` und `G723`, wobei Letzteres im Kodier- und Dekodierbetrieb verwendet wird. `G723` ist ein Sprachkodierer und -dekodierer nach dem G.723 Standard. Die zweite Variante `MEDIA+` beinhaltet zusätzlich noch den Videodekodierer `MPEG2`. Dieser dominiert die Laufzeit von `MEDIA+` und wird deshalb in `MEDIA-` nicht betrachtet. Damit entspricht `MEDIA+` dem in Petzolds Diplomarbeit [Pet04] verwendeten Media-Benchmark, wobei der `MPEG4`-Dekodierer durch einen `MPEG2`-Dekodierer ersetzt wird.

**SORT:** Die Sortieralgorithmen `bubblesort`, `heapsort`, `insertionsort`, `quicksort` und `selectionsort` bilden das Multiprozesssystem Sort in Petzolds Diplomarbeit [Pet04]. SORT beinhaltet zusätzlich die Sortierverfahren `combosort`, `mergesort` und `shellsort`.

Tabelle 5.1 stellt die Prozessanzahl, die Codegröße und die Eingabegröße der Benchmarks gegenüber. Die resultierenden Binärdateien enthalten zusätzlich die Erweiterung der Prozesserzeugung zuzüglich der Strukturen zur Verwendung selbiger durch den Benchmark, das vollständige, nicht auf minimale Größe konfigurierte RTEMS, die C-Bibliothek, den SPMM und die verwendeten Tabellen der Kopierkosten.

Zur Anpassung an den SPMM wurden die Benchmarks den in Abschnitt 4.1 auf Seite 57 beschriebenen Modifikationen unterworfen. In Tabelle 5.1 wird deshalb auch die Veränderung der Laufzeit und des Energiebedarfs der Benchmarks nach der Anpassung an den SPMM relativ zum unveränderten Benchmark angegeben. Der Scratchpad-Speicher wird dabei noch nicht benutzt, daher steigen der Energiebedarf und die Ausführungszeit an. Besonders groß ist der Unterschied bei den MEDIA-Benchmarks, da dort viel häufiger Speicherobjekte über die SPMM-Objekte dereferenziert werden müssen als bei den anderen Benchmarks. Bei dem SORT-Benchmark verringern sich Zyklen- und Energiebedarf leicht. Der SPMM führt nur zu geringem zusätzlichen Aufwand, weil die Sortieralgorithmen nur aus vergleichsweise wenigen Funktionen bestehen und wenige Datenobjekte bearbeiten, wodurch die zusätzliche Dereferenzierungsstufe selten benutzt wird. Der Compiler erzeugt aber erheblich günstigeren Maschinencode, so dass in der Summe minimal weniger Energie und Prozessorzeit benötigt wird.

|                    | Codegröße | Eingabegröße | Prozessanzahl | Bedarf der an den SPMM angepassten Version |         |
|--------------------|-----------|--------------|---------------|--|---------|
|                    |           |              |               | Energie                                    | Zeit    |
| AUTO               | 13 936    | 15 100       | 6             | 103,36%                                    | 103,28% |
| TELECOM            | 27 552    | 24 401       | 7             | 104,09%                                    | 104,01% |
| MEDIA <sup>-</sup> | 7 628     | 4 864        | 5             | 145,95%                                    | 142,81% |
| MEDIA <sup>+</sup> | 23 300    | 11 660       | 6             | 140,34%                                    | 139,40% |
| SORT               | 7 044     | 6 796        | 8             | 99,72%                                     | 99,82%  |

Tabelle 5.1: Übersicht über die Benchmarks, Größenangaben in Bytes

Die Experimente wurden nach dem in Kapitel 4 beschriebenen Arbeitsablauf durchgeführt. MPARM verfügt über Energiemodelle für Scratchpad-Speichergrößen der Zweierpotenzen zwischen 256 Byte ( $2^8$ ) und 16 kB ( $2^{14}$ ). Da das Energiemodell von MPARM verwendet werden sollen, sind die möglichen Scratchpad-Speichergrößen vorgegeben.

Die im Rahmen dieser Arbeit verwendeten Benchmarks sind, verglichen mit Petzolds Benchmarks, groß und verfügen über viel mehr Speicherobjekte. Auch die verwendeten Scratchpad-Speicher sind größer. Die Rechenzeit des ILP-Lösers bei der Positionierung der Objekte steigt dadurch extrem an. Eine mögliche Lösung ist es, die Anzahl der betrachteten Zeitpunkte zu verringern. Die verwendete Zeitscheibenlänge beträgt 1 000 000 Prozessorzyklen beziehungsweise 5 ms. Damit bleibt die Anzahl der betrachteten Prozesswechsel gering und der ILP-Löser kann die Positionierungsprobleme lösen. Zudem benötigt die Belegungsstrategie aufgrund der Dauer der Kopiervorgänge bis zu 100 000 Zyklen ( $500 \mu\text{s}$ ), so dass ein Verhältnis der Rechenzeit der Belegungsstrategie zu der Zeitscheibenlänge von 1 : 10 erreicht wird.

## 5.2 Diskussion der Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Benchmarks vorgestellt und diskutiert.

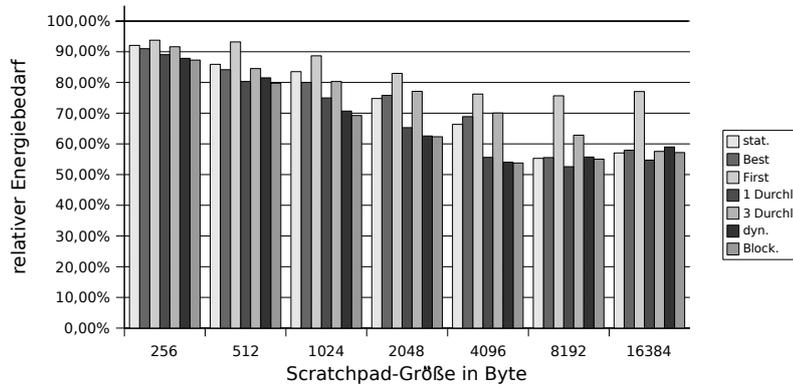
Die Namen der Heuristiken werden abgekürzt verwendet, *stat.* bezeichnet das statische Verfahren aus Abschnitt 3.2.2, *Best* die Belegung mit Best-Fit nach 3.2.4, *First* meint das in Abschnitt 3.2.3 vorgestellte First-Fit-Verfahren. Die Belegung in einem Durchlauf nach Abschnitt 3.2.5 wird als *1 Durchl.* abgekürzt. Folglich meint *3 Durchl.* die Belegung in drei Durchläufen, die in Abschnitt 3.2.6 beschrieben wird. Die dynamische Belegung nach 3.3.2 wird als *dyn.* und die Belegung mit der Blockstrategie aus Abschnitt 3.3.3 wird als *Block.* abgekürzt.

In den nachfolgenden Abbildungen 5.1 bis 5.5 werden die Ergebnisse der einzelnen Heuristiken für die Benchmarks über alle Scratchpad-Größen dargestellt. Dabei sind sowohl der Energiebedarf als auch die Anzahl der benötigten Zyklen auf die Werte des unmodifizierten Benchmarks normiert, die auf einem System ohne Cache und ohne Verwendung des Scratchpad-Speichers erzielt werden. Werte unter 100% bedeuten daher eine Ersparnis von Energie beziehungsweise Ausführungszeit, während Werte darüber eine entsprechende Verschlechterung repräsentieren. Die Tabellen 5.2 bis 5.6 fassen die Benchmarkergebnisse zu Durchschnitts- und Minimalwerten je Heuristik zusammen.

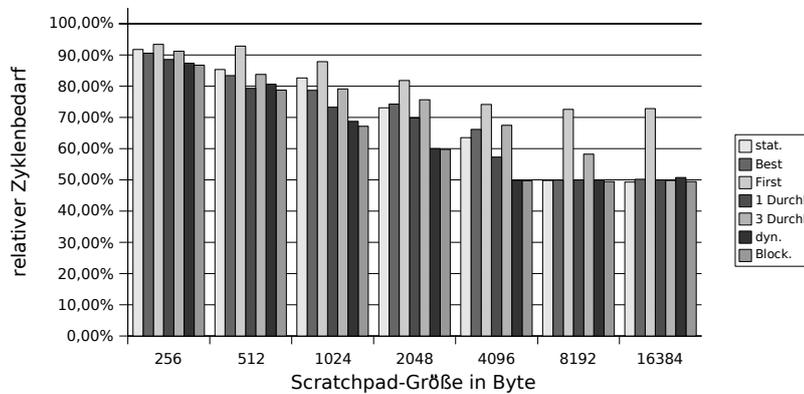
Der Energieverbrauch verhält sich bei allen Benchmarks über alle Belegungsstrategien wie erwartet. Der Energieverbrauch sinkt bis zu einer optimalen Scratchpad-Größe. Er steigt geringfügig, wenn ein größeres Scratchpad verwendet wird, weil alle profitablen Speicherobjekte sich bereits im Scratchpad befinden und größere Speicher mehr Energie je Zugriff benötigen. Dieser Effekt ist bei dem *MEDIA<sup>+</sup>*-Benchmark für die betrachteten Scratchpad-Speichergrößen nicht zu beobachten, weil die optimale Scratchpad-Größe des Benchmarks bei mindestens 16 kB liegt. Für größere Scratchpad-Speicher ist auch hier ein Anstieg des Energiebedarfs zu erwarten.

Idealerweise zeigt die Laufzeit ein ähnliches Verhalten. Sie verringert sich bis zu einer optimalen Scratchpad-Größe, steigt danach aber nicht mit an, denn die Zugriffsgeschwindigkeit bleibt unverändert. Die Heuristiken zur Scratchpad-Belegung werden jedoch zur Laufzeit aufgerufen, so dass ein größeres Scratchpad zu längeren Laufzeiten der Belegungsstrategie führen kann. Solche Effekte treten bei dem *AUTO*-Benchmark einzig für *First* und *dyn.* in minimaler Form auf.

Trotz der allgemein guten Erfahrungen mit der First-Fit-Positionierung von Speicherobjekten bei der dynamischen Speicherverwaltung ist der Erfolg des entsprechenden Verfahrens bei der Scratchpad-Allokation eher mäßig. Bei den Benchmarks *MEDIA<sup>-</sup>* und *MEDIA<sup>+</sup>*, die in den Abbildungen 5.3 und 5.4 sowie den Tabellen 5.4 und 5.5 dargestellt werden, erreicht die First-Fit-Strategie die Gewinnzone nicht. Statt für geringe Fragmentierung zu



(a) Energie

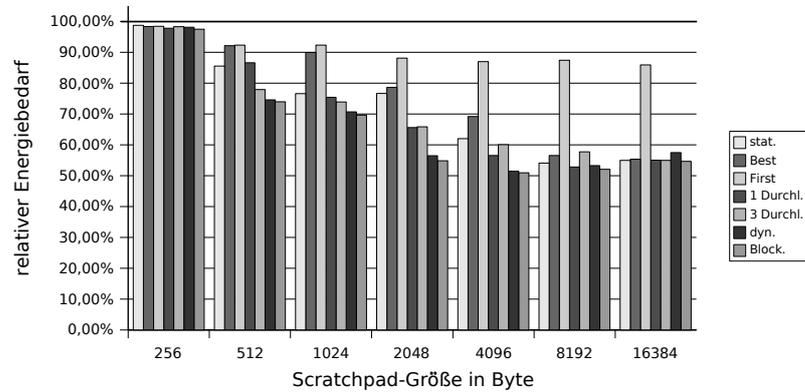


(b) Laufzeit

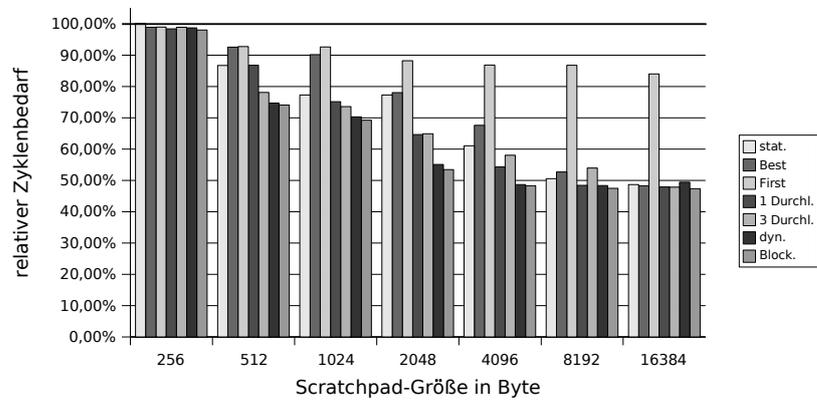
Abbildung 5.1: Ergebnis des AUTO-Benchmarks

|   | stat. | Best | First | 1 Durchl. | 3 Durchl. | dyn. | Block. |
|---|-------|------|-------|-----------|-----------|------|--------|
| durchschnittlicher, relativer Energiebedarf | 74%   | 73%  | 84%   | 68%       | 75%       | 67%  | 66%    |
| minimaler, relativer Energiebedarf          | 55%   | 56%  | 76%   | 53%       | 58%       | 54%  | 54%    |
| durchschnittlicher, relativer Zyklusbedarf  | 71%   | 70%  | 82%   | 67%       | 72%       | 64%  | 63%    |
| minimaler, relativer Zyklusbedarf           | 49%   | 50%  | 73%   | 50%       | 50%       | 50%  | 49%    |

Tabelle 5.2: Übersicht über das Ergebnis des AUTO-Benchmarks



(a) Energie

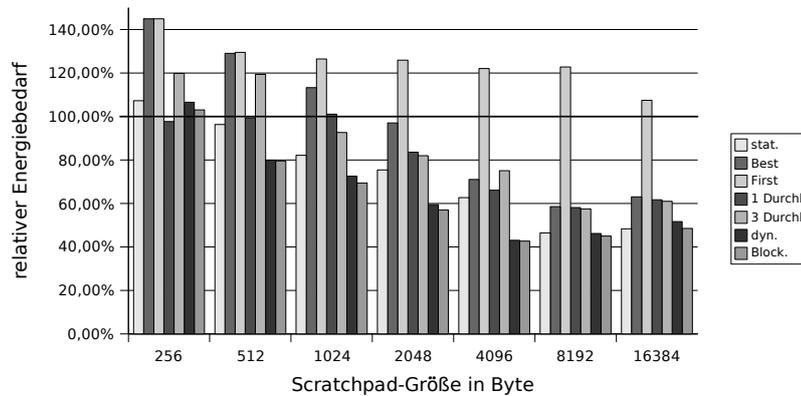


(b) Laufzeit

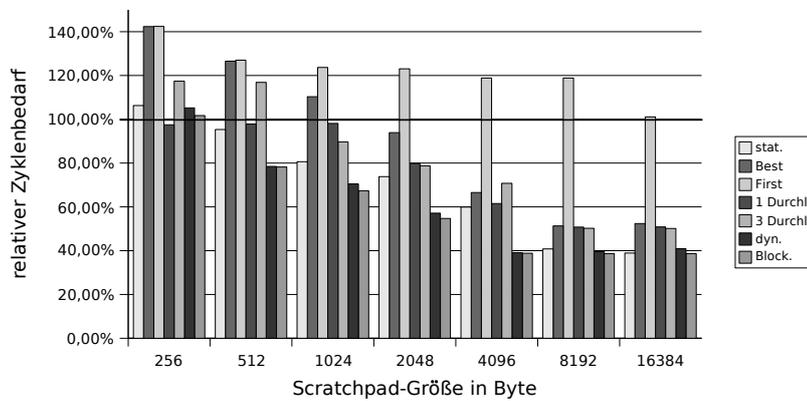
Abbildung 5.2: Ergebnis des TELECOM-Benchmarks

|   | stat. | Best | First | 1 Durchl. | 3 Durchl. | dyn. | Block. |
|---|-------|------|-------|-----------|-----------|------|--------|
| durchschnittlicher, relativer Energiebedarf | 73%   | 77%  | 90%   | 70%       | 70%       | 66%  | 65%    |
| minimaler, relativer Energiebedarf          | 54%   | 55%  | 86%   | 53%       | 55%       | 51%  | 51%    |
| durchschnittlicher, relativer Zyklusbedarf  | 72%   | 75%  | 90%   | 68%       | 68%       | 64%  | 63%    |
| minimaler, relativer Zyklusbedarf           | 49%   | 48%  | 84%   | 48%       | 48%       | 48%  | 47%    |

Tabelle 5.3: Übersicht über das Ergebnis des TELECOM-Benchmarks



(a) Energie

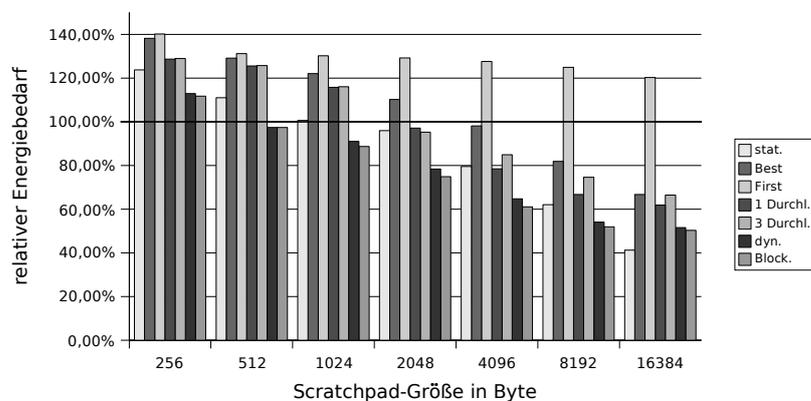


(b) Laufzeit

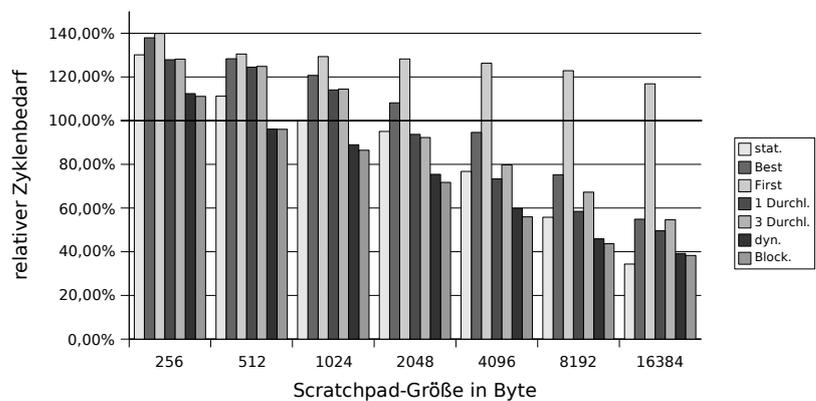
Abbildung 5.3: Ergebnis des MEDIA<sup>-</sup>-Benchmarks

|   | stat. | Best | First | 1 Durchl. | 3 Durchl. | dyn. | Block. |
|---|-------|------|-------|-----------|-----------|------|--------|
| durchschnittlicher, relativer Energiebedarf | 74%   | 97%  | 126%  | 81%       | 87%       | 66%  | 64%    |
| minimaler, relativer Energiebedarf          | 46%   | 59%  | 107%  | 58%       | 57%       | 43%  | 43%    |
| durchschnittlicher, relativer Zyklusbedarf  | 71%   | 92%  | 122%  | 77%       | 82%       | 62%  | 60%    |
| minimaler, relativer Zyklusbedarf           | 39%   | 51%  | 101%  | 51%       | 50%       | 39%  | 39%    |

Tabelle 5.4: Übersicht über das Ergebnis des MEDIA<sup>-</sup>-Benchmarks



(a) Energie

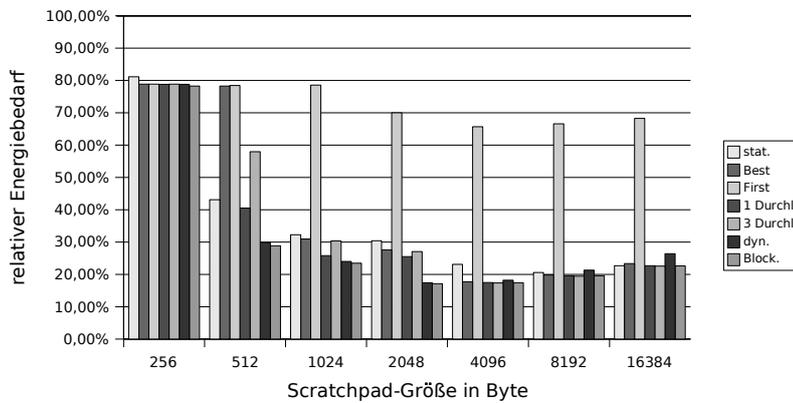


(b) Laufzeit

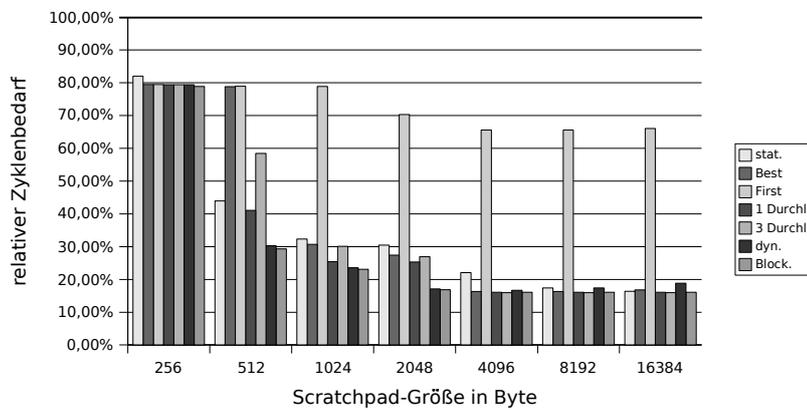
Abbildung 5.4: Ergebnis des MEDIA<sup>+</sup>-Benchmarks

|   | stat. | Best | First | 1 Durchl. | 3 Durchl. | dyn. | Block. |
|---|-------|------|-------|-----------|-----------|------|--------|
| durchschnittlicher, relativer Energiebedarf | 88%   | 107% | 129%  | 96%       | 99%       | 79%  | 77%    |
| minimaler, relativer Energiebedarf          | 41%   | 67%  | 120%  | 62%       | 66%       | 52%  | 50%    |
| durchschnittlicher, relativer Zyklusbedarf  | 86%   | 103% | 128%  | 92%       | 94%       | 74%  | 72%    |
| minimaler, relativer Zyklusbedarf           | 34%   | 55%  | 117%  | 50%       | 55%       | 39%  | 38%    |

Tabelle 5.5: Übersicht über das Ergebnis des MEDIA<sup>+</sup>-Benchmarks



(a) Energie



(b) Laufzeit

Abbildung 5.5: Ergebnis des SORT-Benchmarks

|   | stat. | Best | First | 1 Durchl. | 3 Durchl. | dyn. | Block. |
|---|-------|------|-------|-----------|-----------|------|--------|
| durchschnittlicher, relativer Energiebedarf | 36%   | 40%  | 72%   | 33%       | 36%       | 31%  | 30%    |
| minimaler, relativer Energiebedarf          | 21%   | 18%  | 66%   | 17%       | 17%       | 17%  | 17%    |
| durchschnittlicher, relativer Zyklusbedarf  | 35%   | 38%  | 72%   | 31%       | 35%       | 29%  | 28%    |
| minimaler, relativer Zyklusbedarf           | 16%   | 16%  | 66%   | 16%       | 16%       | 17%  | 16%    |

Tabelle 5.6: Übersicht über das Ergebnis des SORT-Benchmarks

sorgen, entfernt die Belegungsstrategie unnötig häufig unblockierte Speicherobjekte anderer Prozesse aus dem Scratchpad und belegt deren Platz neu. Dieses Verhalten führt neben aufwändigen Kopieraktionen dazu, dass blockierte Objekte sich an ungünstigen Positionen im Scratchpad anlagern und so zu Fragmentierung führen. Die Best-Fit-Strategie vermeidet diese Problematik, indem sie freie Speicherplätze belegt, bevor sie Objekte aus dem Scratchpad verdrängt.

Trotz der schlechten Ergebnisse der roving-Pointer basierten Verfahren bei allgemeinen Problemstellungen der dynamischen Speicherverwaltung, erzielen die heuristische Belegung in einem beziehungsweise in drei Durchläufen hier sehr gute Ergebnisse. Das letztgenannte Verfahren erzielt gegenüber der einfacheren Variante jedoch keine signifikanten Einsparungen. Der Verbesserungsversuch der Belegung in drei Durchläufen gegenüber der Belegungsstrategie mit einem Durchlauf wird daher im Folgenden als gescheitert betrachtet. Der TELECOM-Benchmark untermauert dies mit Abbildung 5.2 und Tabelle 5.3.

Die statische Belegung erzielt bei großen Scratchpad-Speichern ihre besten Ergebnisse. Das dynamische Verfahren ist den anderen Verfahren besonders bei kleinen Scratchpad-Speichern überlegen, da das Umkopieren hier keine hohen Kosten verursacht und jedem Prozess das gesamte Scratchpad zur Verfügung steht. Zudem werden kleine Scratchpads bei allen betrachteten Benchmarks vollständig ausgenutzt.

Das blockbasierte Belegungsverfahren erzielt eine höhere Energieersparnis als das dynamische. Für die Scratchpad-Größen von 8 kB und 16 kB werden beim AUTO-Benchmark jedoch zu viele Kopieroperationen ausgeführt, so dass das Verfahren der Belegung in einem Durchlauf weniger Energie verbraucht.

Die Heuristiken erreichen beim SORT-Benchmark bei weitem die größten Energieeinsparungen, wie in Abbildung 5.5 und Tabelle 5.6 zu sehen ist. Die First-Fit-Strategie erzielt zwar Energieeinsparungen bis zu 34%, doch ab einer Scratchpad-Größe von 1 kB wird unter Verwendung einer der anderen Strategien nur halb so viel Energie zur Ausführung des Benchmarks benötigt.

Die Tabelle 5.7 fasst die Ergebnisse der unterschiedlichen Heuristiken bei Durchführung aller Benchmarks zu je einem durchschnittlichen Energie- und Zyklusbedarf für alle Scratchpad-Größen zusammen. Beide sind bei der blockbasierten Belegungsstrategie von allen Heuristiken am Geringsten. Die Vorhersagbarkeit der Belegung ist bei diesem Belegungsverfahren für jeden Prozess gegeben. Die Blockstrategie ist damit das Empfehlenswerteste der vorgestellten Online-Allokationsverfahren.

Um die optimalen, durch ILP-Gleichungen erzielten Einsparungen mit denen der Heuristiken vergleichen zu können, werden auch für die heuristischen Verfahren die Prozesswechsel durch die in Abschnitt 4.4.1 auf Seite 76 beschriebenen Mechanismen gesteuert. Dabei hat sich das zweite, tabellenba-

|           | Energiebedarf | Zyklenbedarf |
|-----------|---------------|--------------|
| Block.    | 60,20%        | 57,05%       |
| dyn.      | 61,68%        | 58,41%       |
| stat.     | 68,85%        | 66,88%       |
| 1 Durchl. | 69,56%        | 66,90%       |
| 3 Durchl. | 73,30%        | 70,24%       |
| Best      | 78,66%        | 75,73%       |
| First     | 100,23%       | 98,84%       |

Tabelle 5.7: Durchschnittlicher Energie- und Zyklenbedarf aller Benchmarks

sierte Verfahren zur Verwendung der ILP-Lösungen zur Laufzeit als schneller und energiesparender erwiesen.

Die optimalen, ILP-basierten Lösungen des replatzierenden Belegungsverfahrens sparen mindestens so viel Energie, wie die des blockierenden Ansatzes, weil die Lösungsmenge des Blockierenden eine Untermenge der Lösungsmenge des replatzierenden Falles ist. Da die blockierende, optimale Lösung bei den verwendeten Benchmarks die Energieersparnis der replatzierenden, optimale Lösung jedoch nicht erreicht, werden die blockierenden und die replatzierenden Belegungsstrategien im Folgenden einzeln mit den jeweiligen, optimalen Lösungen verglichen. So läßt sich die Güte der Heuristiken unabhängig von dem zu Grunde liegenden Verfahren abschätzen.

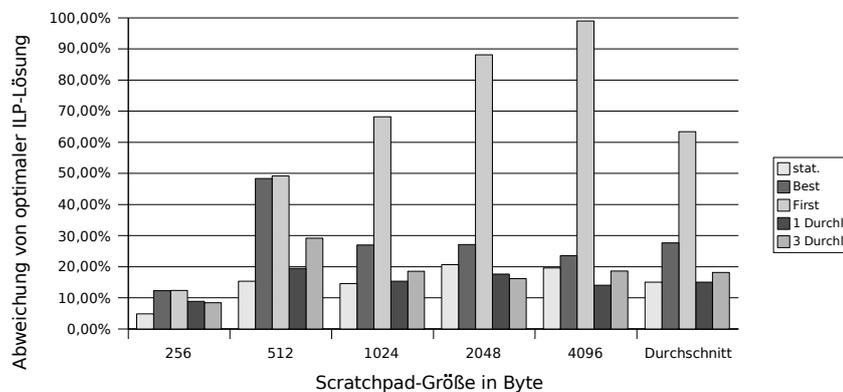


Abbildung 5.6: Abweichung des Energiebedarfs der blockierenden Heuristiken vom Optimum

Abbildung 5.6 stellt die durchschnittliche Abweichung des Energieverbrauches aller Benchmarks bei Verwendung der blockierenden, heuristischen Verfahren gegenüber dem optimalen Energiebedarf bei Benutzung der blockierenden, ILP-basierten Belegung dar. Je besser die von einer Heuristik erzeugte Belegung ist, desto geringer ist die Abweichung. Dabei werden die Scratchpad-Größen von 8 kB und 16 kB nicht betrachtet, da nicht für alle

Benchmarks die Lösungen der ILP-Gleichungen des Positionierungsproblems generiert werden konnten.

Die Strategie der Belegung in einem Durchlauf und die statische Heuristik erreichen etwa die gleiche durchschnittliche Abweichung von rund 15% über alle Benchmarks und Scratchpad-Größen.

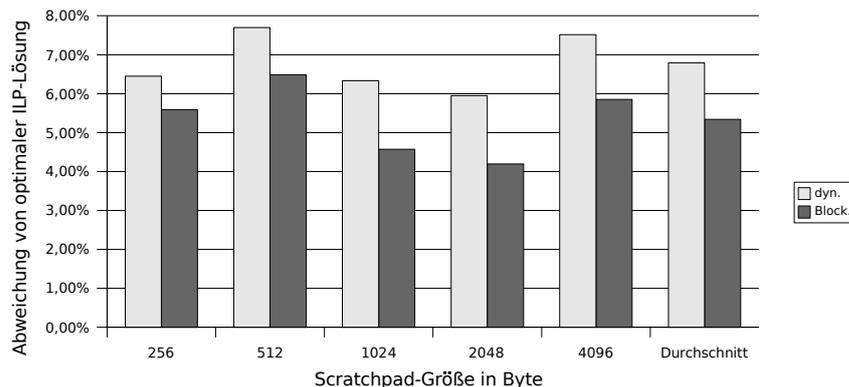


Abbildung 5.7: Abweichung des Energiebedarfs der replatzierenden Heuristiken vom Optimum

Das Belegungsverfahren mit dem Replatzieren der Speicherobjekte wurde geschaffen, um den heuristischen Verfahren zu ermöglichen, die optimale Belegung besser zu approximieren. Die Abbildung 5.7 zeigt die Abweichung der replatzierenden Heuristiken von der optimalen, replatzierenden Lösung analog zu Abbildung 5.6 für den blockierenden Ansatz.

Dabei erreicht die dynamische Belegung eine Abweichung von rund 7% und die Blockstrategie rund 5% Abweichung gegenüber der optimalen Belegung. Damit erzielen die replatzierenden Belegungsverfahren im Durchschnitt deutlich geringere Abweichungen von der entsprechenden optimalen Lösung als die blockierenden Verfahren.

Die Abweichung der blockierenden, optimalen Lösung gegenüber der optimalen, replatzierenden Variante stellt Datenreihe (b) in Abbildung 5.8 dar. Datenreihe (a) zeigt die Abweichung der besten, blockierenden Heuristik, der Belegung in einem Durchlauf, gegenüber der erfolgreichsten, replatzierenden Strategie, der blockbasierten Belegung.

Bei beiden MEDIA-Benchmarks zeigen sich deutlich größere Abweichungen der blockierenden Belegung von der replatzierenden Belegung als bei den anderen Benchmarks. Diese Abweichungen sind bei den Heuristiken noch größer als bei den ILP-basierten Verfahren. Wenn bei der Belegung des Scratchpad-Speichers aufgrund des blockierenden Ansatzes Entscheidungskonflikte zwischen mehreren Objekten entstehen, können diese durch das replatzierende Verfahren umgangen werden. Gerade bei der heuristischen

Belegung des Scratchpad-Speichers wird so deutlich mehr Energie eingespart.

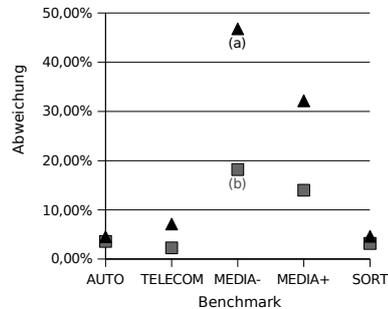


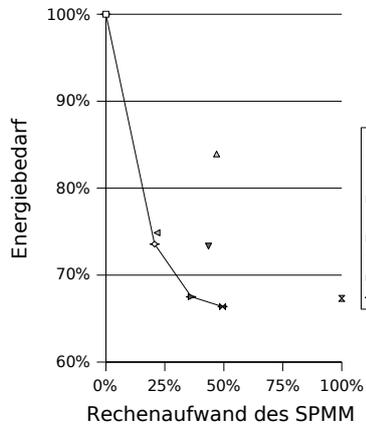
Abbildung 5.8: Abweichung der blockierenden Belegungsverfahren von den Replatzierenden, (a) beste Heuristik, (b) optimale Lösung

### 5.3 Rechenaufwand und Lösungsgüte

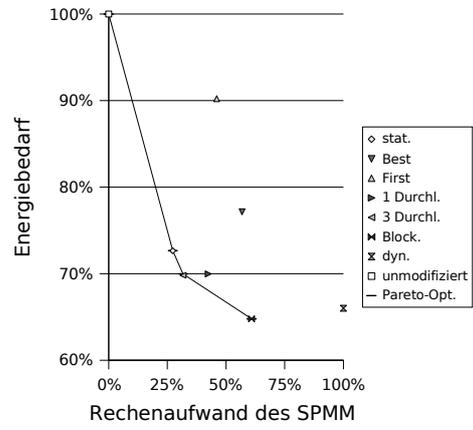
Die Belegungsverfahren des SPMM verfolgen zwei Ziele. Zum einen soll durch den SPMM möglichst wenig zusätzliche Rechenzeit verbraucht werden, zum anderen soll so viel Energie wie möglich eingespart werden. Zusätzlicher Rechenzeitbedarf bedeutet auch zusätzlichen Energieverbrauch. Der SPMM sollte daher nur verwendet werden, wenn dadurch mehr Energie eingespart als zusätzlich benötigt wird.

Gute Belegungsstrategien müssen also mit steigendem Belegungs- aufwand auch entsprechend mehr Energie einsparen. In den Diagrammen der Abbildung 5.9 wird der Energieverbrauch bei Ausführung der Benchmarks relativ zu dem Energiebedarf des unmodifizierten Benchmarks über dem Rechen- und Kopieraufwand des SPMM aufgetragen. Der Aufwand wird dabei relativ zu der jeweils aufwändigsten Strategie angegeben. Für jeden Benchmark wurde über alle Scratchpad-Größen gemittelt. Der Durchschnitt aller Benchmarks wird in Abbildung 5.9(f) dargestellt.

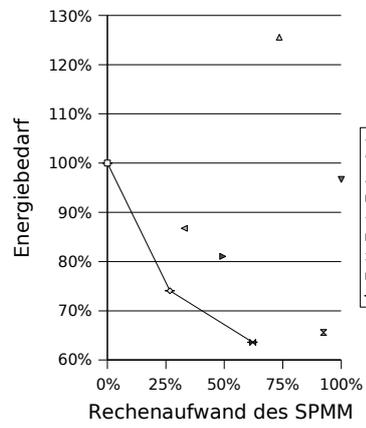
Lediglich die blockbasierte Belegungsstrategie liefert neben der Statis- chen für alle Benchmarks Pareto-optimale Ergebnisse. Die Best-Fit- und First-Fit-Strategien erzeugen viel Aufwand bei der Erstellung der Belegung, doch die erzeugte Belegung ist nicht gut genug, um den Aufwand zu rechtfertigen. Beide bleiben im Vergleich zu der Belegung in einem beziehungs- weise drei Durchläufen deutlich zurück. Die letztgenannten erzielen, ausser in beiden MEDIA-Benchmarks, gute Ergebnisse. Das blockbasierte und das dynamische Belegungsverfahren erreichen die größten Energieeinsparungen, sind jedoch mit massivem Kopieraufwand verbunden. Bei der dynamischen Strategie wird der zusätzliche Kopieraufwand nicht durch besseren Energie-



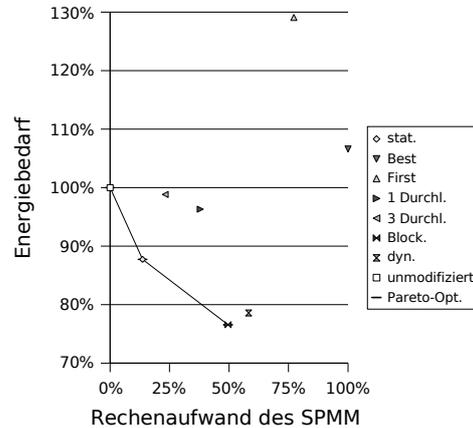
(a) AUTO



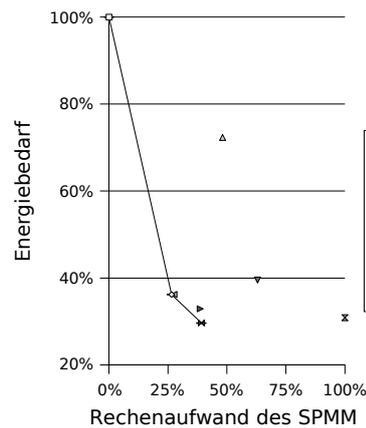
(b) TELECOM



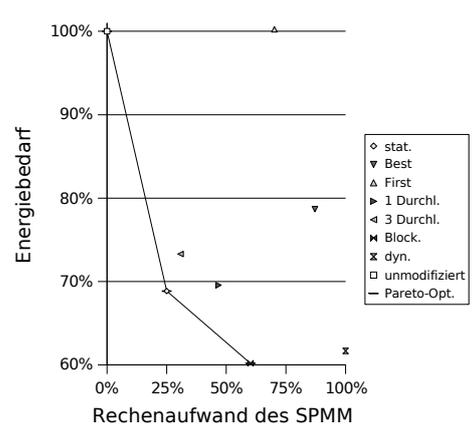
(c) MEDIA<sup>-</sup>



(d) MEDIA<sup>+</sup>



(e) SORT



(f) Mittel über die Benchmarks

Abbildung 5.9: Pareto-Optima des Energiebedarfes über dem Belegungsaufwand des SPMM

verbrauch als bei der Blockstrategie gerechtfertigt. Daher werden hier keine Pareto-Optima erreicht.

## 5.4 Vergleich des SPMM mit Caches

Auch Caches sparen bei der Ausführung von Programmen gegenüber dem Hauptspeicher Zeit und Energie. Allerdings erfolgt die Belegung der Caches vollständig durch Hardware, so dass der Inhalt der Caches zu jedem Zeitpunkt schwer vorherzusagen ist.

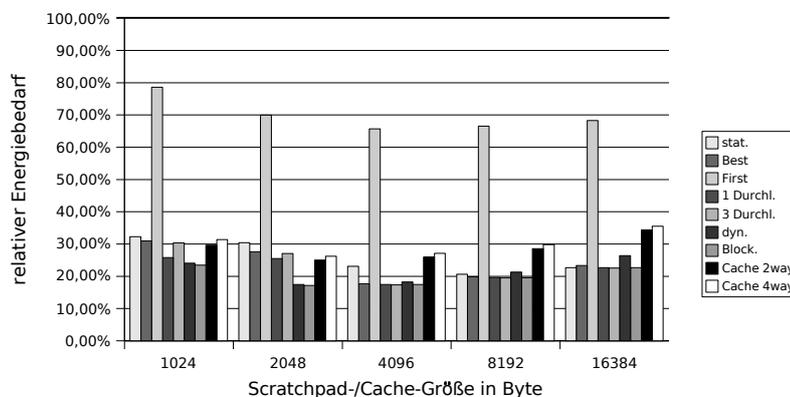


Abbildung 5.10: Vergleich des SPMM mit Caches anhand des SORT-Benchmarks

Eine Belegung des Scratchpads zur Compilezeit ist nicht nur vollständig vorhersagbar, sondern kann auch gegenüber Caches Energie einsparen, wie einige der in Abschnitt 2.5 ab Seite 22 beschriebenen Arbeiten zeigen. Auch eine Belegung des Scratchpad-Speichers zur Laufzeit durch den SPMM kann gegenüber einem System mit Caches Energie einsparen. Die Vorhersagbarkeit der durch den SPMM erzeugten Scratchpad-Belegung ist dabei von der verwendeten Belegungsstrategie abhängig. Das Vergleichssystem verfügt in den durchgeführten Benchmarks über Instruktions- und Datencaches gleicher Größe. Die Größe der Caches liegt zwischen 1 kB und 16 kB und ist durch MPARAM bedingt, dessen Energiemodell erneut zur Bestimmung des Energiebedarfes dient. Abbildung 5.10 stellt den jeweiligen Energiebedarf eines Systems mit zweifach assoziativen (Cache 2way) beziehungsweise vierfach assoziativen (Cache 4way) Caches bei Ausführung des SORT-Benchmarks relativ zu dem unmodifizierten Benchmark auf einem System ohne Caches dar. Der jeweilige Energieverbrauch des Benchmarks bei Verwendung der heuristischen Scratchpad-Belegungsstrategien ist ebenfalls eingetragen. Die maximale Energieersparnis des SPMM gegenüber dem System mit Caches liegt bei 34% beziehungsweise 37%.

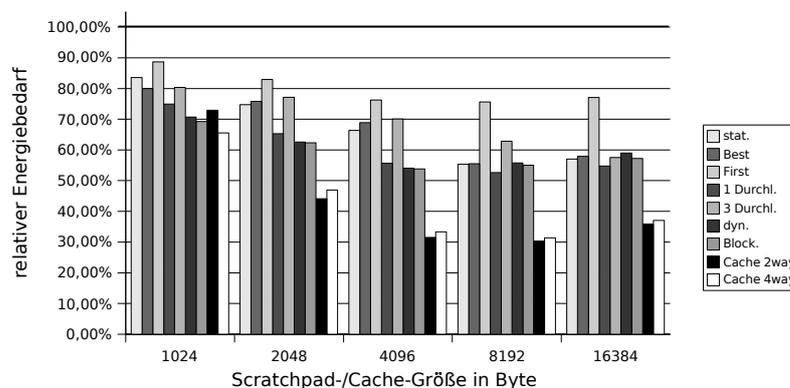


Abbildung 5.11: Vergleich des SPMM mit Caches anhand des AUTO-Benchmarks

Auch für die anderen Benchmarks spart der SPMM gegenüber dem unmodifizierten Benchmark auf einem System ohne Caches Energie und Laufzeit ein, wie in Abschnitt 5.2 ab Seite 84 beschrieben wird. Die Energie- und Laufzeitersparnis ist bei der Verwendung von Caches anstelle des SPMM für TELECOM-, AUTO- und beide MEDIA-Benchmarks dennoch größer. Abbildung 5.11 stellt dies am Beispiel des Energiebedarfs des AUTO-Benchmarks, ebenfalls relativ zu einem unmodifizierten Benchmark auf einem System ohne Caches, dar. Die anderen genannten Benchmarks und die jeweiligen Laufzeiten zeigen ein ähnliches Verhalten.

Der Grund dafür liegt in dem Verhältnis der Anzahl der optimierten Speicherzugriffe zu der Anzahl der Zugriffe insgesamt. Der Cache wird bei potentiell allen Speicherzugriffen eines Systems verwendet. Der SPMM optimiert dagegen nur Zugriffe auf durch ihn verwaltete Speicherobjekte. Weder die Funktionsaufrufe und Datenstrukturen des zugrunde liegenden Betriebssystems RTEMS noch der verwendeten C-Bibliotheken können daher vom SPMM profitieren. Während bei dem SORT-Benchmark 97% der Speicherzugriffe auf vom SPMM verwaltete Speicherobjekte entfallen, sind es beim AUTO-Benchmark 57%, beim TELECOM-Benchmark 66%, beim  $MEDIA^+$ -Benchmark 80% und beim  $MEDIA^-$ -Benchmark 82%. Deshalb ist der Energiebedarf dieser Benchmarks bei Verwendung des SPMM nicht objektiv mit dem bei Verwendung von Caches vergleichbar.

## Kapitel 6

# Zusammenfassung und Ausblick

Tragbare eingebettete Systeme beziehen ihre Energie aus Akkus mit begrenzter Kapazität. Es ist nicht zu erwarten, dass die Energiedichte der Akkus in den nächsten Jahren große Weiterentwicklungen erfährt, so dass die Akkukapazität bei gleichem Gewicht und Volumen kaum zunehmen wird. Die Laufzeit dieser mobilen Systeme wird daher weiter durch den Energieverbrauch bestimmt. Um den Energiebedarf zu senken, sind deshalb neben Optimierungen der Hardware auch energiesparende Softwarelösungen Gegenstand der Forschung. Durch die Verwendung von Scratchpad-Speichern, die von der Software belegt werden, anstelle von Caches kann Energie eingespart werden, wie in [BSL<sup>+</sup>01] gezeigt wurde.

Der überwiegende Teil bisher untersuchter Verfahren zur Belegung der Scratchpad-Speicher verwendet Designzeitansätze, die gegenüber Caches auch eine verbesserte Vorhersagbarkeit bieten. Da eingebettete Systeme häufig Multiprozesssysteme sind, wurden auch hierfür Verfahren entwickelt. Diese Verfahren können nicht verwendet werden, wenn der Benutzer des Systems eigene Applikationen zur Laufzeit hinzufügen können soll. An dieser Stelle sind Belegungsstrategien nötig, die zur Laufzeit des Systems entscheiden können, wie das Scratchpad verwendet werden soll. Deshalb wurden im Rahmen dieser Arbeit sieben Laufzeitverfahren zur Scratchpad-Belegung vorgestellt. Sie können nicht nur die statischen Objekte der Applikationen, globale Variablen und Funktionen, sondern auch dynamisch angelegte Speicherobjekte in das Scratchpad verlagern.

Die vorgestellten Laufzeitverfahren lassen sich in blockierende und replatzierende Strategien unterteilen. Während die Blockierenden benutzte Speicherobjekte tatsächlich nicht bewegen, sorgen die Replatzierenden dafür, dass Prozesse ihre benutzten Objekte immer dann an den gleichen Stellen vorfinden, wenn die Prozesse vom Prozessor ausgeführt werden.

Bei den blockierenden Strategien ist die Vorhersagbarkeit der Belegung durch den Einfluß der Prozesse aufeinander eingeschränkt, da die im Scratchpad-Speicher befindlichen blockierten Objekte eines Prozesses verhindern können, dass Objekte anderer Prozesse im Scratchpad Platz finden.

Die statische Belegung des Scratchpad-Speichers betrachtet keine Ersetzungen von Objekten im Scratchpad durch andere. Um dennoch profitabel arbeiten zu können, wird eine globale Sicht auf die Speicherobjekte aller Prozesse verwendet. Vor allem bei großen Scratchpad-Speichern, die möglichst viele profitable Speicherobjekte aufnehmen können, ist diese Strategie erfolgreich. Der Aufwand bei der Erzeugung der Belegung und zum Kopieren der Speicherobjekte ist sehr gering.

Eine Belegung des Scratchpads auf Basis der, bei der dynamischen Speicherverwaltung erfolgreichen, First-Fit-Strategie erzielt die geringsten Energieeinsparungen, da sie Speicherobjekte anderer Prozesse auch dann ersetzt, wenn geeigneter freier Speicher zur Verfügung steht. Neben zusätzlichem Kopieraufwand hat dies auch Fragmentierungsprobleme zur Folge, wenn Objekte ungünstig ersetzt werden.

Der ebenfalls aus der dynamischen Speicherverwaltung bekannten Best-Fit-Strategie gelingen durch die Bevorzugung freien Speichers weit größere Einsparungen. Diese Strategie erfordert jedoch für jedes Speicherobjekt eine vollständige Suche über die belegten und freien Bereiche des Scratchpad-Speichers, was einen sehr großen Aufwand bedeuten kann.

Einen anderen Weg geht die Belegung des Scratchpad-Speichers in einem Durchlauf. Sie verfolgt eine roving-Pointer genannte Strategie, die bei der dynamischen Speicherverwaltung im allgemeinen Fall sehr schlechte Ergebnisse in Bezug auf die Fragmentierung erzielt. Hier jedoch erzeugt sie in geringer Ausführungszeit eine sehr gute Belegung.

Die Belegung in drei Durchläufen stellt dem gegenüber keine Verbesserung dar. Der zusätzliche Aufwand bei der Suche nach geeigneten Zielbereichen zum Einkopieren der Speicherobjekte sorgt nicht für eine signifikant größere Energieersparnis.

Bei den replatzierenden Strategien beeinflussen sich die Objekte der Prozesse nicht gegenseitig, somit wird die Belegung aus der Sicht jeden Prozesses vorhersagbar. Dafür steigt der Kopieraufwand enorm an, denn die Speicherobjekte werden immer wieder von anderen verdrängt, so dass die Belegung des Scratchpad-Speichers ständigen Wechseln unterworfen ist.

Die dynamische Belegung des Scratchpads kopiert deshalb den Scratchpad-Inhalt jedes Prozesses gleich vollständig um. Diese Strategie erreicht die größten Einsparungen, weil jedem Prozess der gesamte Scratchpad-Speicher zur Verfügung steht. Dieser maximalen Energieersparnis steht der maximale Kopieraufwand gegenüber. Dieser benötigt so viel zusätzliche Energie, dass insgesamt die Blockstrategie einen geringeren Energieverbrauch bewirkt.

Die Blockstrategie gruppiert einzukopierende Speicherobjekte und platziert neue Blöcke, wenn möglich, in freiem Speicher. Ist dies nicht möglich,

so wird der vordere Bereich des Scratchpad-Speichers von allen Prozessen dynamisch benutzt. Blöcke, die bereits einmal im Scratchpad-Speicher waren, werden stets wieder an den gleichen Platz kopiert, wenn eines der darin enthaltenen Speicherobjekte in Benutzung war. Diese Heuristik ist die einzige der hier Vorgestellten, die sowohl die statische als auch die dynamische Lösung nachbilden kann. Zur Vereinfachung verzichtet sie jedoch auf die Verwaltung dynamischer Speicherobjekte.

Die erzielten Einsparungen sind trotz der hohen theoretischen Komplexität der behandelten Probleme bei allen Strategien ausser der First-Fit-Belegung relativ hoch und nah an der optimalen Lösung, wie eine experimentelle Überprüfung anhand mehrerer Benchmarks zeigt. Bei vier der fünf verwendeten Benchmarks erzielt der vorgestellte Scratchpad-Speichermanager (SPMM) dennoch keine Energieeinsparungen gegenüber der Verwendung von Caches. Die Begründung liegt in der Struktur der verwendeten Benchmarks, bei denen bis zu 47% ihrer Speicherzugriffe auf Bibliotheks- und Betriebssystemfunktionen entfallen, die nicht vom SPMM optimiert werden können.

Für den Programmierer von Anwendungen für mobile, eingebettete Systeme stellt der SPMM eine einfach zu verwendende Programmierschnittstelle zur Verfügung, wobei ein Teil der Anpassung der Programme an den SPMM durch den vorgestellten Arbeitsablauf automatisiert werden kann. Für die einzelne Applikation spielt es keine Rolle, welche weiteren Applikationen beliebiger Herkunft nebenläufig ausgeführt werden, solange diese keine schwerwiegenden Fehler verursachen.

Ausgehend von den vorgestellten Online-Belegungsstrategien und den vorgestellten Designzeit-Programmen zu deren Unterstützung, bietet sich die Untersuchung folgender Fragestellungen und Erweiterungen an:

**Verwendung zusätzlicher Compilezeit-Optimierungen:** Andere Verfahren zur Scratchpad-Belegung haben eine positive Reaktion auf zusätzliche Compilezeit-Optimierungen gezeigt. Ein Beispiel dafür ist der Loop-Tiling-Ansatz in [Wil05]. Mit dem dort vorgestellten Verfahren können auch größere Datenfelder in den Scratchpad-Speicher verschoben und dort sequentiell verarbeitet werden. Es wäre zu überprüfen, ob dies mit dem hier vorgestellten Ansatz gewinnbringend kombiniert werden kann.

Function Inlining, also das Einkopieren von Funktionen an die Stelle ihrer Aufrufe, wird verwendet, um durch verbesserte Lokalität Geschwindigkeitsvorteile zu erzielen. Dabei werden die übrig gebliebenen Funktionen größer, was ihre Verschiebung in den Scratchpad-Speicher erschwert. Die Umkehrung dieser Optimierung, Function Exlining, lagert Schleifenrumpfe in eigene Funktionen aus und verkleinert so die Funktionen, auf die oft zugegriffen wird. Sie sind so erheblich leichter und in größerer Anzahl in den Scratchpad-Speicher zu verschieben.

Erste Experimente mit dieser Idee zeigten vielversprechende Ergebnisse mit den vorgestellten Heuristiken. Das Ausmaß der möglichen zusätzlichen Energieersparnis ist zu erforschen. Ebenso ist die Grenze auszuloten, ab der eine weitere Aufteilung der Funktionen bis hin zu Basisblöcken keine weiteren Einsparungen erzielen kann.

**Automatisierung der Verwendung der API:** Die Verwendung der SPMM-API erfordert in ihrem momentanen Zustand erheblichen manuellen Aufwand. Jeder neue Zugriff auf ein Speicherobjekt muss von den Funktionen zur Markierung des Objektes als benutzt umschlossen werden. In der automatisierten, optimierten Positionierung dieser Markierungen liegt vermutlich weiteres Energiesparpotenzial.

Auch die Umstellung der dynamischen Speicherobjekte auf die Verwendung der SPMM-API erfordert massiven zusätzlichen Programmieraufwand, der in Form einer Quelltexttransformation durchgeführt werden kann. An dieser Stelle herrscht ebenfalls noch Handlungsbedarf.

**Verbesserung der Integration:** Wie in Abschnitt 5.4 auf Seite 95 beschrieben, kann der SPMM in seiner jetzigen Form bei vier von fünf Benchmarks gegenüber einem Cache keine weiteren Einsparungen erzielen. Um das volle Potential zu verifizieren, müssen entweder die Betriebssystem- und Bibliotheksaufrufe aus den Benchmarks entfernt werden, oder selbige an die Verwendung des SPMM angepasst werden. Wie der SORT-Benchmark zeigt, können dadurch auch gegenüber Caches Energieeinsparungen erzielt werden. Zudem stellt sich die Frage, ob nicht auch Teile des SPMM durch den SPMM verwaltet werden sollten.

**Suche nach besseren Heuristiken:** Die im Rahmen dieser Arbeit vorgestellten Heuristiken wurden auch hinsichtlich ihrer Pareto-Optimalität untersucht. Es ist wahrscheinlich, dass die Mehrzahl der Pareto-optimalen Heuristiken zur Scratchpad-Belegung noch nicht gefunden ist. In diesem Sinne bessere Heuristiken sollten gefunden werden können.

**Anpassung an Priorität der Prozesse:** In eingebetteten Systemen werden den Prozessen häufig verschiedene Prioritäten zugewiesen. Momentan verfügt der SPMM nicht über eine Gewichtung der Speicherobjekte anhand der Priorität der zugehörigen Prozesse. Die Heuristiken sollten an diese Gegebenheit angepasst werden, denn die darin enthaltenen, zusätzlichen Informationen können ausgenutzt werden, um die Energieersparnis zu vergrößern.

**Erweiterung für Realzeitprozesse:** In Abschnitt 3.2 auf Seite 34 wurde auf eine mögliche Erweiterung der blockierenden Verfahren zur

Wiederherstellung der Vorhersagbarkeit der Scratchpad-Belegung für Realzeitprozesse eingegangen. Das tatsächliche Verhalten einer solchen Erweiterung ist mangels Realisierung nicht bekannt.

**Betrachtung zusätzlicher Hardware:** In [WHM04] wurde der Fall betrachtet, dass mehrere Scratchpad-Speicher vorhanden sind, für die die Applikationen optimiert werden. Dieser Ansatz kann auf Multiprozessysteme übertragen werden. Der SPMM kann für diesen Fall angepasst werden, wobei zusätzlicher Raum für weitere Forschungen entsteht, wenn man auch die Möglichkeit in Betracht zieht, verschiedene heuristische Belegungsverfahren für unterschiedliche Scratchpad-Speicher zu verwenden.

Eine weitere Hardwareerweiterung, an die der SPMM angepasst werden kann, ist eine DMA-Einheit, wie sie in [PMA<sup>+</sup>04] verwendet wird, um Daten zwischen Hauptspeicher und Scratchpad-Speicher zu transferieren. Die damit möglichen Energieeinsparungen sind ebenfalls zu erkunden.

**Überprüfung der Verschiebung der Kopierfunktion:** Wie bereits beschrieben, fiel früh die Designentscheidung, die Kopierfunktion fest in den Scratchpad-Speicher zu verschieben. Das Verhalten der heuristischen Belegungsverfahren in dem Fall, dass die Kopierfunktion im Hauptspeicher verbleibt, ist unbekannt. Gerade bei kleinen Scratchpad-Speichern kann es sinnvoll sein, die Kopierfunktion im Hauptspeicher zu belassen um das vollständige Scratchpad mit Speicherobjekten der Anwendungen zu belegen, weil die Kopierfunktion weniger häufig durchlaufen wird, als bei größeren Scratchpad-Speichern.

**Dynamische Verwaltung der Energieersparnis:** Die mögliche Energieersparnis durch die Verschiebung von Speicherobjekten in den Scratchpad-Speicher ist nicht, wie in dieser Arbeit zur Vereinfachung der Problemstellung verwendet, konstant über die Laufzeit des zugehörigen Prozesses. Vielmehr entfallen die möglichen Einsparungen genau auf die Zeit, in der das Speicherobjekt als benutzt markiert ist. Es ist denkbar, den SPMM so anzupassen, dass Speicherobjekte durch den SPMM genau dann in den Scratchpad-Speicher verschoben werden, wenn sie als benutzt markiert werden. Sie können entfernt werden, sobald sie nicht mehr benutzt werden. Dieser Ansatz ist mit dem Einfügen von Instruktionen zur Optimierung der Cache-Belegung (Software Prefetching[McI98]) bei Desktop-Prozessoren und -Anwendungen vergleichbar.



## Anhang A

# ILP-Gleichungen zur Adresszuweisung

Die Gleichungen (3.7) bis (3.9) in Abschnitt 3.2.1 auf Seite 38 modellieren die Positionierung der Speicherobjekte im Scratchpad-Speicher für die Belegung mit Blockieren der Objekte. Im Folgenden wird daraus eine ILP-Formulierung hergeleitet. Hierzu sind in Ergänzung zu den Definitionen auf den Seiten 35 und 38 weitere Variablen erforderlich.

|                    |   |
|--------------------|---|
| $L$                | Konstante, groß genug   |
| $\rho_{i',t}$      | Binäre Entscheidungsvariable, muss für gültige Ergebnisse 0 sein  |
| $\sigma_{i',j',t}$ | Binäre Entscheidungsvariable, 1, wenn Gleichung A.4 durch die $a_{i',t}$ beziehungsweise $a_{j',t}$ erfüllt werden soll, 0 wenn Gleichung A.3 durch die $a_{i',t}$ beziehungsweise $a_{j',t}$ erfüllt werden soll |

Speicherobjekte müssen sich nach Gleichung (3.7) vollständig im Scratchpad-Speicher befinden. Dazu ist lediglich die Zerlegung in zwei Gleichungen nötig.

$$\forall t \in T \forall i' \in I'_t a_{i',t} \geq START_{SPM} \quad (A.1)$$

$$\forall t \in T \forall i' \in I'_t a_{i',t} \leq END_{SPM} - SIZE_{i'} \quad (A.2)$$

In Gleichung (3.8) kommt ein *XOR* vor, dass in ILP-Lösern nicht verwendet werden kann. Es wird ersetzt durch die weitere binäre Entscheidungsvariable  $\sigma_{i',j',t}$ . Je nach angenommenem Wert der Variable muss dann der erste oder der zweite Teil der Gleichung erfüllt werden. Zudem müssen die zwei Argumente des *XOR* auf zwei eigene Gleichungen aufgeteilt werden.

$$\forall t \in T \forall i' \in I'_t \forall j' \in I'_t, i' \neq j' \quad a_{i',t} - a_{j',t} + L \cdot \sigma_{i',j',t} \geq \text{SIZE}_{j'} \quad (\text{A.3})$$

$$\forall t \in T \forall i' \in I'_t \forall j' \in I'_t, i' \neq j' \quad a_{j',t} - a_{i',t} - L \cdot \sigma_{i',j',t} \geq \text{SIZE}_{i'} - L \quad (\text{A.4})$$

Gleichung (3.9) wird umgestellt und  $\rho_{i',t}$  wird eingebracht.  $\rho_{i',t}$  kann nur 0 sein, sonst wird die Gleichung unerfüllbar.

$$\forall t \in T \forall i' \in I'_t \cap I'_{t-1} \quad a_{i',t} - a_{i',t-1} - L \cdot \rho_{i',t} = 0 \quad (\text{A.5})$$

Die  $\rho_{i',t}$  sollen nicht wirklich optimiert werden. Sie sind nur dazu da, die Beschränkungen in Anlehnung an [Ver06] mit der Zielfunktion (A.6) zu verknüpfen.

$$\sum_{t \in T} \sum_{i' \in I'_t \cap I'_{t-1}} \rho_{i',t} \longrightarrow \min \quad (\text{A.6})$$

Der replatzierende Ansatz fügt der Positionierung lediglich Gleichung (3.11) hinzu. Alle anderen Gleichungen werden übernommen. Daher sind auch die folgenden Aussagen für beide Ansätze gleichermaßen gültig.

Die Anzahl der binären Entscheidungsvariablen  $\sigma_{i',j',t}$  ist ebenso wie die Anzahl der beschränkenden Ungleichungen nach oben durch  $O(|MO|^2 \cdot |T|)$  beschränkt. CPLEX benutzt zur Lösung ganzzahliger Optimierungsprobleme ein branch-and-bound-Verfahren. Während die Gleichungen zur Erzeugung der Scratchpad-Belegung in unter einer Stunde durch CPLEX gelöst werden können, kann die Rechenzeit zur Lösung des Platzierungsproblems je nach Problem Instanz im Bereich von Wochen liegen.

# Literaturverzeichnis

- [ABC03] ANGIOLINI, Federico ; BENINI, Luca ; CAPRARA, Alberto: Polynomial-time algorithm for on-chip scratchpad memory partitioning. In: *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems (CASES)*, ACM Press, 2003. – ISBN 1581136765, S. 318–326
- [ABC05] ANGIOLINI, Federico ; BENINI, Luca ; CAPRARA, Alberto: An efficient profile-based algorithm for scratchpad memory partitioning. In: *IEEE Transactions on CAD of Integrated Circuits and Systems* 24 (2005), Nr. 11, S. 1660–1676
- [AMC<sup>+</sup>04] ATIENZA, David ; MAMAGKAKIS, Stylianos ; CATTLOOR, Franky ; MENDIAS, Jose M. ; SOUDRIS, Dimitris: Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications. In: *Proceedings of the conference on Design, automation and test in Europe (DATE)*, IEEE Computer Society, 2004. – ISBN 07695208551, S. 532–537
- [Apl00] APLIO, INC. *Aplio/Phone User Guide*.  
<http://www.aplio.com/ug20/ctUGintro20.htm>. 2000
- [ARM01a] ARM LTD. *ARM7EJ-S Technical Reference Manual*. Dokument-Nr.: ARM DDI 0214B. 2001
- [ARM01b] ARM LTD. *ARM7TDMI Product Overview*. Dokument-Nr.: ARM DVI 0027B. 2001
- [ARM04] ARM LTD. *ARM7TDMI Technical Reference Manual*. Dokument-Nr.: ARM DDI 0210C. 2004
- [Axi02] AXIS COMMUNICATIONS AB. *Datenblatt AXIS 2100 Network Camera*.  
<http://www.axis.com>. 2002
- [BBC<sup>+</sup>01] BENINI, Luca ; BRUNI, Davide ; CHINOSI, Mauro ; SILVANO, Cristina ; ZACCARIA, Vittorio ; ZAFALON, Roberto: A Power Mode-

- ling and Estimation Framework for VLIW-based Embedded Systems. In: *International Workshop-Power And Timing Modeling, Optimization and Simulation (PATMOS)*, 2001, S. 2.1.2–2.1.10
- [BMP03] BENINI, Luca ; MACII, Alberto ; PONCINO, Massimo: Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. In: *ACM Transactions on Embedded Computing Systems* 2 (2003), Nr. 1, S. 5–32
- [Bon94] BONWICK, Jeff: The Slab Allocator: An Object-Caching Kernel Memory Allocator. In: *USENIX Summer Technical Conference Proceedings*, USENIX Association, 1994. – ISBN 1880446626, S. 87–98
- [BSL<sup>+</sup>01] BANAKAR, Rajeshwari ; STEINKE, Stefan ; LEE, Bo-Sik ; BALAKRISHNAN, M. ; MARWEDEL, Peter: Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption / Universität Dortmund. 2001 ( 762). – Forschungsbericht
- [Buc01] Kap. 5.4 In: BUCHMANN, Isidor: *Batteries in a Portable World: A Handbook on Rechargeable Batteries for Non-Engineers*. Cadex Electronics Inc., 2001. – ISBN 0968211828
- [BZZ04] BONA, Andrea ; ZACCARIA, Vittorio ; ZAFALON, Roberto: System Level Power Modeling and Simulation of High-End Industrial Network-on-Chip. In: *Proceedings of the conference on Design, automation and test in Europe (DATE)*, IEEE Computer Society, 2004. – ISBN 07695208553, S. 318–324
- [Cir01] CIRRUS LOGIC INC. *Datenblatt CS89712*. Dokument-Nr.: DS502PP2. 2001
- [CZG98] CHINOSI, Mauro ; ZAFALON, Roberto ; GUARDIANI, Carlo: Automatic characterization and modeling of power consumption in static RAMs. In: *Proceedings of the 1998 international symposium on Low power electronics and design (ISLPED)*, ACM Press, 1998. – ISBN 1581130597, S. 112–114
- [Dal00] DALES, Michael. *SWARM 0.44 Documentation*.  
[http://www.cl.cam.ac.uk/~sim\\$mw24/phd/swarm.html](http://www.cl.cam.ac.uk/~sim$mw24/phd/swarm.html).  
2000
- [Dur06] DURYEE, Tricia: The phone of the future: wired to run your life. In: *The Seattle Times* (14.05.2006)
- [DZ05] 664 Millionen verkaufte Handys in 2004. In: *Die Zeit* (08.02.2005)

- [Emb06] EMBEDDED LINUX MICROCONTROLLER PROJECT. *µCLinux Homepage*.  
[www.uclinux.org](http://www.uclinux.org). 2006
- [Eng01] ENGEL, Michael. *Linux auf Embedded Systemen ohne MMU*.  
<http://www.linux-community.de/Neues/story?storyid=889>. 2001
- [GJ79] GAREY, Michael R. ; JOHNSON, David S.: *Computers and Intractability: A Guide To the Theory of NP-Completeness*. Freeman, 1979. – ISBN 0716710455
- [GK03] GURUN, Selim ; KRINTZ, Chandra: Addressing the Energy Crisis in Mobile Computing With Developing Power Aware Software / University of California, Santa Barbara. 2003 ( 2003-15). – Forschungsbericht
- [GLL<sup>+</sup>03] GARNETT, Nick ; LARMOUR, Jonathan ; LUNN, Andrew ; THOMAS, Gary ; VEER, Bart. *eCos Reference Manual*.  
<http://ecos.sourceforge.org/>. 2003
- [Gos91] GOSER, Karl: *Großintegrationstechnik Teil II*. Hüthig, 1991. – ISBN 3778516167
- [GRE<sup>+</sup>01] GUTHAUS, Matthew R. ; RINGENBERG, Jeffrey S. ; ERNST, Dan ; AUSTIN, Todd M. ; MUDGE, Trevor ; BROWN, Richard B.: MiBench: A free, commercially representative embedded benchmark suite. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization (WWC)*, IEEE Computer Society, 2001. – ISBN 0780373154, S. 3–14
- [Han03] HANDERA. *TRGpro product specification*.  
<http://www.handera.com/Products/TRGpro.aspx>. 2003
- [Hil04] HILLERINGMANN, Ulrich: *Silizium-Halbleitertechnologie*. B.G. Teubner Verlag, 2004. – ISBN 3519301490
- [HK03] HITZ, Martin ; KAPPEL, Gerti: *UML@Work*. dpunkt.verlag, 2003. – ISBN 3898641945
- [HP02] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002. – ISBN 1558605967
- [ICD06] ICD - INFORMATIK CENTRUM DORTMUND E.V. *ICD-C Compiler framework*.  
<http://www.icd.de/es/icd-c/icd-c.html>. 2006

- [ILO06] ILOG INC. *ILOG CPLEX*.  
<http://www.ilog.com/products/cplex/>. 2006
- [IY97] ISHIHARA, Tohru ; YASUURA, Hiroto: Experimental Analysis of Power Estimation Models of CMOS VLSI Circuits. In: *IEICE Transactions Fundamentals* E80-A (1997), Nr. 3, S. 480–486
- [KBH94] KHURI, Sami ; BÄCK, Thomas ; HEITKÖTTER, Jörg: The zero/one multiple knapsack problem and genetic algorithms. In: *Proceedings of the 1994 ACM symposium on Applied computing (SAC)*, ACM Press, 1994. – ISBN 0897916476, S. 188–193
- [KG02] KRISHNASWAMY, Arvind ; GUPTA, Rajiv: Profile guided selection of arm and thumb instructions. In: *Proceedings of the joint conference on Languages, compilers and tools for embedded systems (LCTES/SCOPES)*, ACM Press, 2002. – ISBN 1581135270, S. 56–64
- [KR02] KUROSE, James F. ; ROSS, Keith W.: *Computernetze*. Addison-Wesley, 2002. – ISBN 3827370175
- [LAB<sup>+</sup>04] LOGHI, Mirko ; ANGIOLINI, Federico ; BERTOZZI, Davide ; BENINI, Luca ; ZAFALON, Roberto: Analyzing On-Chip Communication in a MPSoC Environment. In: *Proceedings of the conference on Design, automation and test in Europe (DATE)*, IEEE Computer Society, 2004. – ISBN 0769520852
- [Lea96] LEA, Doug: A Memory Allocator. In: *unix/Mail* (1996), Nr. 6. – auch verfügbar unter  
<http://gee.cs.oswego.edu/dl/html/malloc.html>
- [LFZE00] LEBECK, Alvin R. ; FAN, Xiaobo ; ZENG, Heng ; ELLIS, Carla: Power aware page allocation. In: *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, ACM Press, 2000. – ISBN 1581133170, S. 105–116
- [LP06] *Introduction to lp\_solve 5.5.0.7*.  
<http://lpsolve.sourceforge.net/5.5/>. 2006
- [LPB04] LOGHI, Mirko ; PONCINO, Massimo ; BENINI, Luca: Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In: *Proceedings of the 14th ACM Great Lakes symposium on VLSI (GLSVLSI)*, ACM Press, 2004. – ISBN 1581138539, S. 406–410
- [MAP<sup>+</sup>06] MAMAGKAKIS, Stylianos ; ATIENZA, David ; POU CET, Christophe ; CATT HOOR, Francky ; SOUDRIS, Dimitrios ; MENDIAS, Jose M.: Automated exploration of pareto-optimal configurations

- in parameterized dynamic memory allocation for embedded systems. In: *Proceedings of the conference on Design, automation and test in Europe (DATE)*, European Design and Automation Association, 2006. – ISBN 3981080106, S. 874–875
- [Mar03] MARWEDEL, Peter: *Embedded System Design*. Kluwer Academic Publishers, 2003. – ISBN 1402076908
- [Mas02] MASSA, Anthony J.: *Embedded Software Development with eCos*. Prentice Hall, 2002. – ISBN 0130354732
- [McI98] MCINTOSH, Nathaniel: Compiler Support for Software Prefetching / Rice University, Houston. 1998 ( TR98-303). – Forschungsbericht
- [OAR03a] OAR CORPORATION. *RTEMS C User's Guide*. <http://www.rtems.com/>. 2003
- [OAR03b] OAR CORPORATION. *RTEMS Frequently Asked Questions*. <http://www.rtems.com/>. 2003
- [Pet04] PETZOLD, Klaus: *Scratchpad-Allokations-Strategien für Multiprozess-Systeme*, Universität Dortmund, Lehrstuhl 12, Diplomarbeit, 2004
- [PMA<sup>+</sup>04] POLETTI, Francesco ; MARCHAL, Paul ; ATIENZA, David ; BENINI, Luca ; CATHOOR, Francky ; MENDIAS, Jose M.: An integrated hardware/software approach for run-time scratchpad management. In: *Proceedings of the 41th Design Automation Conference (DAC)*, ACM Press, 2004. – ISBN 1581138288, S. 238–243
- [Sch03] SCHWIEGELSHOHN, Uwe. *Technische Informatik*. Vorlesungsskript, Universität Dortmund, Institut für Roboterforschung, Lehrstuhl für Datenverarbeitungssysteme. 2003
- [Sea00] SEAL, David (Hrsg.): *ARM Architecture Reference Manual*. Addison Wesley, 2000. – auch verfügbar unter Dokument-Nr.: ARM DDI 0100. – ISBN 0201737191
- [SGW<sup>+</sup>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 (ISSS)*, ACM Press, 2002. – ISBN 1581135769, S. 213–218

- [SWLM02] STEINKE, Stefan ; WEHMEYER, Lars ; LEE, Bo-Sik ; MARWEDEL, Peter: Assigning Program and Data Objects to Scratchpad for Energy Reduction. In: *Design, Automation and Test in Europe Conference and Exposition (DATE)*, IEEE Computer Society, 2002. – ISBN 0769514715, S. 409–417
- [SZWM01] STEINKE, Stefan ; ZOBIEGALA, Christoph ; WEHMEYER, Lars ; MARWEDEL, Peter: Moving Program Objects to Scratch-Pad Memory for Energy Reduction / Universität Dortmund. 2001 ( 756). – Forschungsbericht
- [TMW94] TIWARI, Vivek ; MALIK, Sharad ; WOLFE, Andrew: Power analysis of embedded software: a first step towards software power minimization. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2 (1994), Nr. 4, S. 437–445
- [Vah95] VAHALIA, Uresh: *UNIX Internals*. Prentice Hall, 1995. – ISBN 0131019082
- [Ver06] VERMA, Manish: *Advanced Memory Optimization Techniques for Low Power Embedded Processors*, Universität Dortmund, Lehrstuhl 12, Dissertation, 2006
- [VSM03] VERMA, Manish ; STEINKE, Stefan ; MARWEDEL, Peter: Data partitioning for maximal scratchpad usage. In: *Proceedings of the 2003 conference on Asia South Pacific design automation (ASPDAC)*, ACM Press, 2003. – ISBN 0780376609, S. 77–83
- [VWM04a] VERMA, Manish ; WEHMEYER, Lars ; MARWEDEL, Peter: Cache-Aware Scratchpad Allocation Algorithm. In: *Design, Automation and Test in Europe Conference and Exposition (DATE)*, IEEE Computer Society, 2004. – ISBN 0769520855, S. 1264–1269
- [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 (CODES+ISSS)*, ACM Press, 2004. – ISBN 1581139373, S. 104–109
- [Weg99] WEGENER, Ingo: *Theoretische Informatik - eine algorithmenorientierte Einführung*. B.G. Teubner Verlag, 1999. – ISBN 3519121239
- [Weg02] WEGENER, Ingo. *Datenstrukturen, Algorithmen und Programmierung 2*. Vorlesungsskript, Universität Dortmund, Lehrstuhl für Effiziente Algorithmen und Komplexitätstheorie. 2002

- [Weg03] WEGENER, Ingo: *Komplexitätstheorie*. Springer, 2003. – ISBN 3540001611
- [WHM04] WEHMEYER, Lars ; HELMIG, Urs ; MARWEDEL, Peter: Compiler-optimized usage of partitioned memories. In: *Proceedings of the 3rd workshop on Memory performance issues (WMPI)*, ACM Press, 2004. – ISBN 159593040X, S. 114–120
- [Wil05] WILMER, Thorsten: *Energieeffiziente Belegung von Scratchpad-Speichern mit Code und Arrays durch Loop-Tiling*, Universität Dortmund, Lehrstuhl 12, Diplomarbeit, 2005
- [WJNB95] WILSON, Paul R. ; JOHNSTONE, Mark S. ; NEELY, Michael ; BOLES, David: Dynamic Storage Allocation: A Survey and Critical Review. In: *Proceedings of the International Workshop on Memory Management (IWMM)*, 1995
- [WM95] WULF, Wim A. ; MCKEE, Sally A.: Hitting the Memory Wall: Implications of the Obvious. In: *Computer Architecture News* 23 (1995), Nr. 1, S. 20–24
- [WW06] WAGNER, Jens (Hrsg.) ; WEHMEYER, Lars (Hrsg.): *ROAMIX. Books on Demand*, 2006. – ISBN 3833443669
- [ZRW05] ZHAO, Qin ; RABBAH, Rodric ; WONG, Weng-Fai: Dynamic memory optimization using pool allocation and prefetching. In: *SIGARCH Computer Architecture News* 33 (2005), Nr. 5, S. 27–32. – ISSN 01635964