UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Diplomarbeit

Scratchpad-Allokations-Strategien für Multiprozess-Systeme

Klaus Petzold

1. September 2004

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl 12 (Technische Informatik und Eingebettete Systeme)
Fachbereich Informatik
Universität Dortmund
Gutachter:
B. Tech. Manish Verma
Prof. Dr. Peter Marwedel

GERMANY · D-44221 DORTMUND

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die Zahl der Rechnersysteme hat in den letzten Jahren stark zugenommen. In immer mehr Bereichen werden Rechner eingesetzt, um z.B. die Effizienz in der industriellen Fertigung zu erhöhen oder auch um Menschen von gefährlichen Aufgaben zu entlasten. Es ist abzusehen, daß sich dieser Trend in den nächsten Jahren so fortsetzen wird. Die gestiegene Nachfrage nach Rechnersystemen führt zu einem verstärkten Konkurrenzkampf, der verkürzte Entwicklungszyklen und die Ausstattung der Produkte mit immer mehr Funktionalität mit sich bringt.

Ein Großteil der heute produzierten Prozessoren wird in eingebetteten Systemen eingesetzt. Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt eingebettet sind und deren Vorhandensein für den Nutzer nicht direkt sichtbar ist. Zu finden sind sie zu einem großen Teil in Telekommunikationsgeräten wie Mobiltelefonen, in Transportsystemen wie Fahrzeugen, Zügen und Flugzeugen und in Geräten aus dem medizinischen Bereich. Dabei wird eine hohe Anforderung an die Zuverlässigkeit und Effizienz dieser Systeme gestellt. Mehr über Einsatzgebiete und Anforderungen an eingebettete Systeme findet sich in [Mar03]. Das Buch gibt einen detaillierten Überblick über den Entwurf solcher Systeme, angefangen bei der Spezifikation, über die Auswahl von Hardware- und Software-Komponenten, das Hardware/Software Codesign und schließt ab mit Methoden zum Testen dieser Systeme.

Die Effizienzanforderung an eingebettete Systeme beinhaltet, daß sie kostengünstig sind, ein geringes Gewicht haben und möglichst energieeffizient arbeiten. Dabei stellt besonders der Energieverbrauch ein Problem dar. Mit einem steigenden Energieverbrauch nimmt auch die Wärmeentwicklung zu. Da die Wärme abgeführt werden muß, ist der Einsatz von Kühlsystemen notwendig, was die Systeme teurer und schwerer macht und damit den Effizienzanforderungen entgegenwirkt. Mit einer steigenden Wärmeentwicklung nimmt auch die Beanspruchung des Materials zu. Die Anforderung an die Zuverlässigkeit der eingebetteten Systeme beinhaltet unter anderem, daß

sie ausfallsicher und hoch verfügbar sind. Selbst wenn die durch die Abwärme verursachten Probleme gelöst werden können, steht die Anforderung der Energieeffizienz im Konflikt mit der gestiegenen Funktionalität der Systeme. Der Konkurrenzkampf zwingt die Hersteller, immer mehr Funktionalität zu implementieren. Dies läßt sich beispielhaft an den heutigen Mobiltelefonen erkennen, die sich immer mehr zu Personal Digital Assistents entwickeln und mit denen außerdem noch Spiele gespielt sowie Fotos und Videos aufgenommen und betrachtet werden können. Jede weitere Funktionalität läßt den Energieverbrauch steigen, doch kann die Weiterentwicklung der Akkus den gestiegenen Energiebedarf nicht kompensieren. Ein Ausweg würde darin bestehen, das System mit mehr Akkus auszustatten, doch steht dies erneut im Widerspruch zu den Anforderungen nach einem geringen Gewicht und niedrigen Preis.

Wie eben aufgezeigt, besteht ein Bedarf, den Energieverbrauch von Rechnersystemen zu senken. In vergangenen Jahren wurde vornehmlich versucht, die Hardware zu verbessern. Es zeigte sich jedoch, daß dies allein nicht ausreicht und auch auf der Softwareseite nach entsprechenden Lösungen gesucht werden muß. Die verkürzten Entwicklungszyklen lassen eine Optimierung der Software von Hand nicht mehr zu. Damit rückt die Entwicklung von Compilern, die diese Aufgabe automatisch übernehmen können, zunehmend ins Zentrum des Interesses.

1.1 Ziele der Arbeit

Es zeigt sich, daß ein Großteil der Energie in den Speichern verbraucht wird. Deshalb sind seit einiger Zeit verschiedene Arten von Speichern Gegenstand der Forschung, um den Energieverbrauch und die Ausführungszeit zu senken. Diese Speicher sind meist klein, Ziel ist es deshalb, häufig benutzten Code und Daten in diesen Speichern zu halten.

Caches erledigen diese Aufgabe automatisch, es ist keine Anpassung der Applikationen nötig. Forschungsergebnisse zeigen jedoch, daß durch den Einsatz von Compilertechniken die Effizienz der Cache-Speicher erhöht werden kann. In letzter Zeit wurde untersucht, ob sich durch die Verwendung von Scratchpad-Speichern Einsparungen erzielen lassen. Scratchpad-Speicher sind meist wenige Kilobyte groß und im Prozessor integriert. Pro Zugriff wird wenig Energie verbraucht und es fallen eine geringere Anzahl an Waitstates als beim Zugriff auf den Hauptspeicher an. Der Scratchpad-Speicher wird in den Adressraum eingeblendet und es ist Aufgabe des Compilers zu entscheiden, welcher Code oder welche Daten dort abgelegt werden sollen. Untersuchungen zeigen, daß Scratchpad-Speicher größere Einsparungen gegenüber Caches an Chipfläche, Energieverbrauch und Ausführungszeit möglich machen. Weiterhin ist für Realzeit-Systeme die Abschätzung der maximalen Ausführungszeit von besonderem Interesse. Für Caches können nur schlecht

gute obere Schranken gefunden werden, dies ist für Scratchpad-Speicher mit weniger Aufwand besser möglich.

Es sind für Scratchpad-Speicher bereits eine Vielzahl an Methoden entwickelt worden, um Ausführungszeit und Energieverbrauch von Applikationen zu senken. Wie bereits in der Einleitung erwähnt, erfüllen viele Rechnersysteme mehrere Funktionalitäten. Diese werden durch einzelne Prozesse realisiert. Ziel dieser Arbeit ist es, für Multiprozess-Systeme, die über einen Scratchpad-Speicher verfügen, Kostenfunktionen zu optimieren. Dies soll durch den Einsatz geeigneter Allokations-Strategien geschehen, die die Verteilung des Scratchpad-Speichers unter den Prozessen steuern. Mit diesen Allokations-Strategien ist es auch möglich, den Entwurf von Multiprozess-Systemen zu unterstützen. Nach einer Anwendung der Allokations-Strategien für verschiedene Scratchpad-Größen kann die günstigste Variante bezüglich einer Kostenfunktion ausgewählt werden.

1.2 Überblick

In Kapitel 2 werden einige wichtige Grundlagen eingeführt. Zuerst wird der ARM7TDMI-Prozessor und ein Energiemodell für diesen Prozessor vorgestellt. Anschließend folgt ein Überblick über die Aufgaben und Funktionsweise eines Dispatchers und Schedulers. Das Kapitel schließt ab mit einer Methode zur Darstellung von Optimierungsproblemen und einem Abschnitt über verwandte Arbeiten.

Kapitel 3 gliedert sich in drei Abschnitte, in denen jeweils eine Allokations-Strategie vorgestellt wird. Aufgabe der Strategien ist es, für ein Multiprozess-System, das über einen Scratchpad-Speicher verfügt, eine Kostenfunktion zu optimieren. Zu Anfang jedes Abschnitts wird die Grundidee der jeweiligen Strategie dargestellt, diese dann mathematisch formuliert und am Ende ein Algorithmus zur Berechnung einer optimalen Lösung vorgestellt.

Wie ausgehend von einer Menge von Applikationen und einem Scratchpad-Speicher der Energieverbrauch eines Multiprozess-Systems minimiert wird, zeigt Kapitel 4. Hierbei wird die Funktionsweise jedes an diesem Arbeitsprozess beteiligten Programmes besprochen.

In Kapitel 5 werden die mit den Allokations-Strategien erzielten Ergebnisse dargestellt und mit einem einfachen Ansatz zur Verteilung des zur Verfügung stehenden Scratchpads sowie der Verwendung eines Caches verglichen.

Abgeschlossen wird diese Arbeit mit Kapitel 6. Dort werden die erzielten Ergebnisse zusammengefasst und mögliche Erweiterungen diskutiert.

Kapitel 2

Grundlagen

Die in dieser Arbeit vorgestellten Strategien zur bestmöglichen Nutzung des Scratchpads durch mehrere Prozesse werden auf dem ARM7TDMI-Prozessor getestet. Dieser Prozessor wird in Abschnitt 2.1 vorgestellt.

Die Programme, die im Zuge der Auswertung entstehen, werden allerdings nicht auf realer Hardware ausgeführt, sondern im ARMulator, einer Simulationssoftware, die das Verhalten des ARM7TDMI-Prozessors zyklengenau nachbilden kann. Anhand des Execution-Trace eines im ARMulator simulierten Programms und eines Energiemodells für den ARM7TDMI-Prozessor kann der Energieverbrauch dieses Programms bestimmt werden. Das verwendete Energiemodell wird in Abschnitt 2.2 erläutert.

Damit mehrere Prozesse auf einem Prozessor quasi-parallel ausgeführt werden können, bedarf es eines Dispatchers und eines Schedulers. Mit deren Arbeitsweise, den verschiedenen Typen von Scheduling-Strategien und benötigten Datenstrukturen beschäftigt sich Abschnitt 2.3.

Die in dieser Arbeit entwickelten Allokations-Strategien werden zum besseren Verständnis zuerst als ILP-Gleichungen formuliert. Für das in Abschnitt 3.4.1 vorgestellte HAOP-Problem werden ILP-Gleichungen genutzt, um eine optimale Aufteilung von Programmobjekten auf verschiedene Speicher zu berechnen. Abschnitt 2.4 gibt eine Einführung in Integer Linear Programming.

Abgeschlossen wird dieses Kapitel mit einem Überblick über Arbeiten, auf denen diese Arbeit aufbaut und Arbeiten, die andere Ansätze zur Lösung der Problemstellung gewählt haben.

2.1 Der ARM7TDMI Prozessor

Eine detaillierte Übersicht über den ARM7TDMI-Prozessor findet sich in [ARM01a].

Der ARM7TDMI-Prozessor ist ein 32-Bit RISC Prozessor, der durch seine geringe Chipfläche, seinen niedrigen Energieverbrauch und seine ho-

he Verarbeitungsgeschwindigkeit besonders interessant für mobile Anwendungen ist. Er wird häufig in Pagern, Digitalkameras oder auch tragbaren Audio-Playern eingesetzt.

Der Prozessor besitzt eine Load/Store Architektur. Seine dreistufige Instruction Pipeline besteht aus den Stufen Fetch, Decode und Execute. Insgesamt hat der Prozessor 31 32-Bit Register und 6 Status-Register. Die Menge der aktuell zugreifbaren Register ist abhängig vom Betriebsmodus des Prozessors. Weiterhin hat er eine 32-Bit ALU, sowie ein 32-Bit Multiplizierwerk und einen separaten Barrelshifter, der beim Laden von Registern mit Konstanten und bei arithmetischen Operationen angesprochen werden kann.

Dem Programmierer stehen auf diesem Prozessor zwei Befehlssätze zur Verfügung. Der ARM-Befehlssatz umfasst 80 Kernbefehle, die 32-Bit breit sind. Im Gegensatz dazu sind die Befehle im Thumb-Befehlssatz mit Ausnahme eines Sprungbefehls 16-Bit breit. Untersuchungen zeigen, daß dadurch eine um bis zu 30% höhere Codedichte, sowie ein geringerer Energieverbrauch erreicht werden können. Durch Einschränkung der Befehlsworte auf 16-Bit enthält der Befehlssatz nur noch 36 Befehle und es kann nur noch auf eine Teilmenge der Register direkt zugegriffen werden. Jeder Thumb-Befehl wird vor seiner Ausführung ohne Geschwindigkeitsverlust in sein ARM-Äquivalent umgewandelt, intern verarbeitet der Prozessor also nur ARM-Instruktionen. Zwischen den Befehlssätzen kann mit einem speziellen Befehl gewechselt werden.

2.1.1 Unterschiede der Thumb- und ARM-Befehlssätze

Einige wichtige Unterschiede der beiden Befehlssätze werden nun im Detail aufgelistet:

- Die Breite der Befehlsworte beträgt 32-Bit im ARM-Befehlssatz und 16-Bit im Thumb-Befehlssatz.
- Insgesamt 80 Befehle stehen im ARM-Befehlssatz zur Verfügung, 36 im Thumb-Befehlssatz.
- Die Menge der 3-Register-Befehle ist im Thumb-Befehlssatz gegenüber der ARM-Befehlssatz verfügbaren Menge eingeschränkt.
- Der ARM-Befehlssatz sieht für jeden Befehl vor, daß er abhängig von der aktuellen Belegung der Status-Flags im CPSR[1] ausgeführt werden kann. Zusätzlich kann für arithmetische Befehle spezifiziert werden, ob das Ergebnis der Operation einen Einfluß auf die Status-Flags hat. Befindet sich der Prozessor im Thumb-Zustand, dann werden die Status-Flags nach arithmetischen Operationen immer aktualisiert.

[1] Current Program Status Register

- Der Thumb-Befehlssatz kennt weniger Adressierungsarten als der ARM-Befehlssatz.

- Im ARM-Befehlssatz kann vor arithmetischen Operationen und vor dem Laden von Registern mit Konstanten auf einen der Operanden der Barrelshifter angewendet werden. Dies ist im Thumb-Befehlssatz nicht möglich.

Bei der Wahl des Befehlssatzes muß der Programmierer abwägen zwischen den Vorteilen der hohen Codedichte beim Thumb-Befehlssatz und den vielfältigeren Möglichkeiten des ARM-Befehlssatzes. Die konditionale Ausführung der Befehle im ARM-Befehlssatz hilft z.B. Sprünge und damit ein Leeren und Neuladen der Pipeline zu vermeiden.

2.1.2 Registerorganisation

Der Prozessor stellt 31 32-Bit Register und 6 Status-Register bereit. Arbeitet der Prozessor im ARM-Zustand, dann sind 16 der Register frei zugreifbar. Die Register R0-R12 stehen zur freien Verfügung, R13 wird per Konvention als Stack-Pointer verwendet, R14 ist das Link-Register, das bei Funktionsaufrufen die Rücksprungadresse aufnimmt und R15 ist der Programmzähler. Die restlichen 32-Bit Register sind sogenannte Banked-Register.

Tritt der Prozessor in eine Ausnahme- bzw. Interrupt-Behandlung ein, dann hängt es vom neuen Betriebsmodus ab, wieviele Banked-Register ihr Äquivalent aus dem vorhergehenden Betriebsmodus überlagern. Im IRQ-Betriebsmodus, der zur Behandlung von allgemeinen Interrupts dient, werden die Register R13 und R14 durch die Banked-Register R13_irq und R14_irq überlagert. Während der Interrupt-Behandlung können diese Register ganz normal verwendet werden. Wird in den alten Betriebsmodus zurückgekehrt, dann findet das unterbrochene Programm die Register R13-R14 unverändert. In allen Betriebsmodi außer dem System- und User-Modus sind die Register R13 und R14 Banked-Register. Das Register R14 enthält die Rücksprungadresse und durch die Sicherung von R13 können auf einfache Art und Weise getrennte Stacks für das Benutzerprogramm und die Ausnahme- bzw. Interrupt-Handler realisiert werden.

Im Thumb-Befehlssatz sind wegen der eingeschränkten Befehlswortbreite nur acht der 16 im ARM-Befehlssatz verwendbaren Register direkt zugreifbar, nämlich R0-R7. Der Prozessor arbeitet im Thumb-Zustand auf derselben Registermenge wie im ARM-Zustand. Zum Beschreiben und Auslesen der Register R8-R15 dient der MOV-Befehl, der Inhalte zwischen den hohen Registern R8-R15 und den niedrigen Registern R0-R7 bewegen kann.

Das Status-Register CPSR enthält die Flags, einen Teil, der den aktuellen Betriebsmodus kodiert und zwei Bits, über die Interrupts und Fast-Interrupts ein- bzw. ausgeschaltet werden können. Wechselt der Prozessor

den Betriebsmodus, dann wird der Inhalt des CPSR in das SPSR[2] des neuen Betriebsmodus kopiert. Damit bleiben nach der Rückkehr aus einer Ausnahme-Behandlung die Flags erhalten und für den in Abschnitt 2.3 beschriebenen Dispatcher ist es mit Hilfe des SPSR möglich, die Flags eines unterbrochenen Prozesses zu speichern.

2.1.3 Betriebsmodi

Der ARM7TDMI kennt insgesamt sieben Betriebsmodi. Alle Modi außer dem User-Modus sind privilegierte Modi. Sie dienen dazu, Interrupt- oder Ausnahme-Behandlungen durchzuführen oder auf geschützte Ressourcen zuzugreifen. Tritt ein Interrupt oder eine Ausnahme auf, dann wechselt der Prozessor immer in den ARM-Befehlssatz.

User-Modus: In diesem Modus werden die meisten Programme ausgeführt.

Interrupt-Modus: Wird für die allgemeine Behandlung von Interrupts benutzt.

Fast-Interrupt-Modus: Dieser Modus wird für Datentransporte und Channel Prozesse verwendet.

Supervisor-Modus: Ein geschützter Modus für das Betriebssystem.

Abort-Modus: Der Prozessor wechselt in diesen Modus, wenn ein Daten- oder Instruktions-Prefetch-Abort auftritt.

System-Modus: Ein privilegierter User-Modus für das Betriebssystem.

Undefined-Modus: Stößt der Prozessor auf eine nicht definierte Instruktion, dann wird in diesen Modus gewechselt.

2.1.4 Die Behandlung von IRQs

Für diese Arbeit wurde ein preemptives Scheduling-Verfahren (siehe Abschnitt 2.3) implementiert. Bei diesem Verfahren wird periodisch ein Interrupt generiert. Dabei wechselt der Prozessor in den IRQ-Modus und führt folgende Schritte durch [ARM01a]:

1. Die Adresse des nächsten Befehls des unterbrochenen Programms wird im Register R14_irq gesichert.

[2]Saved Program Status Register

2. Der Inhalt des CPSR wird in das Status-Register SPSR_irq gesichert. War der Prozessor vorher im Thumb-Zustand, dann wird in den ARM-Zustand geschaltet. Um eine wiederkehrende Verschachtelung von Interrupt-Aufrufen zu vermeiden, werden die Interrupt- und Fast-Interrupt-Flags gesetzt.

3. Ab der unteren Grenze des Adressraums liegen in 4-Byte-Abständen die Ausnahme- und Interrupt-Handler. Gewöhnlich steht dort jeweils ein Sprungbefehl zur eigentlichen Handler-Routine. In diesem Schritt wird der Befehl, der zum IRQ-Handler gehört, geladen.

Nach der Beendigung des Handlers werden folgende Schritte abgearbeitet, um den alten Programmzustand wiederherzustellen:

1. Das Register R14_irq wird abzüglich eines Offsets in das Programmzähler-Register kopiert. Dies entspricht einem Rücksprung aus dem IRQ-Handler zurück in das unterbrochene Programm.

2. Der Inhalt des Status-Registers SPSR_irq wird in das CPSR kopiert.

2.2 Ein Energiemodell für den ARM7TDMI Prozessor

Tiwari et al. [TMW94] [TL98] haben in ihren Arbeiten ein allgemeines Energiemodell vorgeschlagen. Die Daten für dieses Modell bilden die Kosten, die durch die Abarbeitung der Instruktionen im Prozessor entstehen. Wendet man dieses Energiemodell auf alle Instruktionen eines Programms an, dann können mit hoher Genauigkeit die durch die Programmausführung im Prozessor verursachten Kosten berechnet werden.

In eingebetteten Systemen wird ein Großteil der Energie beim Zugriff auf Speicher verbraucht, die nicht im Prozessor integriert sind. In [SKWM01] wurde das Prozessor-Energie-Modell so erweitert, daß auch der Energieverbrauch durch Speicherzugriffe mit berücksichtigt wird. Michael Theokharidis hat in seiner Diplomarbeit [The00] die notwendigen Daten für den ARM7TDMI-Prozessor gemessen, womit nun ein Energiemodell für diesen Prozessor zur Verfügung steht, das die Berechnung des Energieverbrauchs von Programmen erlaubt.

Der Energieverbrauch in einem Rechnersystem setzt sich zusammen aus der Energie, die Mainboard, Prozessor, Speicher und die weiteren Peripheriegeräte verbrauchen. Da ein Compiler darauf Einfluß nehmen kann, aus welchen Instruktionen ein Programm zusammengesetzt wird und auf welche Speicher und wie oft auf diese zugegriffen wird, liegt das Hauptaugenmerk in dieser Arbeit auf dem Energieverbrauch von Prozessor und Speicher.

Fließt ein konstanter Strom I und liegt eine konstante Spannung U an, dann ist die in einem Zeitraum ΔT verbrauchte Energie gegeben durch:

$$E = U \cdot I \cdot \Delta T \tag{2.1}$$

Besteht der Zeitraum ΔT aus N Taktperioden der Länge t_c, dann kann 2.1 geschrieben werden als:

$$E = U \cdot I \cdot N \cdot t_c \tag{2.2}$$

Es stellt sich nun die Frage, woher der Energieverbrauch in einem Prozessor oder Speicher resultiert. Meist werden diese Komponenten durch CMOS-Schaltungen realisiert. Arbeitet ein Prozessor einen Befehl ab, oder wird auf einen Speicher zugegriffen, so führt dies zu Zustandsänderungen in den zugrunde liegenden CMOS-Schaltungen.

Die drei Hauptursachen für den Energieverbrauch in CMOS-Schaltungen sind [Syn96]:

- **Switching Power**
  Ein Strom, der beim Laden und Entladen der in den Schaltungen vorhandenen Kapazitäten fließt. Bei aktiven Schaltungen trägt dieser Strom hauptsächlich zum Energieverbrauch bei.

- **Short Circuit Power**
  Durch die unterschiedlichen Schaltzeiten der in den Schaltungen vorkommenden Transistoren fließt dieser Strom beim Schalten kurzzeitig.

- **Leakage Power**
  Da es keinen perfekten Isolator gibt, fließen selbst bei inaktiven Schaltungen Leckströme, deren Anteil an der Gesamtbilanz jedoch recht gering ist. In zukünftigen Systemen werden die Leckströme an Bedeutung gewinnen.

Die Kosten E_{instr} einer durch den Prozessor abgearbeiteten Instruktion ergeben sich wie folgt [SKWM01]:

$$E_{instr} = E_{cpu} + E_{mem} \tag{2.3}$$

E_{cpu} beschreibt die Kosten, die in der CPU durch die Verarbeitung der Instruktion entstehen. E_{mem} setzt sich aus den Kosten zusammen, die einmal durch das Laden der Instruktion verursacht werden und möglichen weiteren Kosten, die entstehen, wenn der Befehl ein Datum aus dem Speicher liest oder oder ein Datum schreibt.

Die CPU-Kosten können weiter in zwei Kostenarten aufgeschlüsselt werden [TMW94]:

$$E_{cpu} = E_{Base} + E_{InterInstr} \tag{2.4}$$

Basiskosten: Diese Kosten beschreiben den Energieverbrauch, der aus der Abarbeitung einer Instruktion resultiert und durch eine erhöhte Schaltkreisaktivität im Prozessor begründet ist.

Inter-Instruktions-Kosten: Arbeitet ein Prozessor eine Folge von Befehlen ab, dann entstehen Kosten durch Schaltkreisaktivitäten, die durch den Wechsel zwischen den Befehlen verursacht werden. Benutzen aufeinanderfolgende Befehle nicht dieselben Prozessor-Ressourcen, dann entstehen Kosten durch das Aktivieren bzw. Deaktivieren der entsprechenden Prozessor-Ressourcen. Die Fließbandverarbeitung der Instruktionen durch die CPU kann einen weiteren Kostenfaktor verursachen. Hierfür gibt es drei Gründe:

- **Resource Hazard**
  Versuchen zwei aufeinanderfolgende Instruktion auf die gleichen CPU-Ressourcen zuzugreifen, dann wird ein Anhalten der Pipeline erzwungen, da nur ein Zugriff pro Zeiteinheit erfolgen kann.
- **Data Hazard**
  Ein Pipeline-Stall wird durch diesen Konflikttyp verursacht, wenn eine Instruktion das Ergebnis einer vorhergehenden Instruktion benötigt, dieses Ergebnis aber noch nicht zur Verfügung steht.
- **Control Hazard**
  Control Hazards werden durch bedingte und unbedingte Sprünge verursacht. Da die Pipeline die nächsten abzuarbeitenden Befehle bereits enthält, muss nach Ausführung eines Sprungs der Inhalt der Pipeline verworfen und die Befehle, die sich an der neuen aktuellen Programmadresse befinden, geladen werden.

Die Speicherkosten E_{mem}, die eine Instruktion verursacht, können geschrieben werden als:

$$E_{mem} = E_{Fetch} + E_{MemAccess} \tag{2.5}$$

Durch das Laden jeder Instruktion entstehen Kosten E_{Fetch}. Liest oder schreibt diese Instruktion ein Datum, so fallen zusätzlich Kosten $E_{MemAccess}$ an, die von der Art des Zugriffs und dem angesprochenen Speicher abhängen.

Die einzelnen Kosten können nun zusammengefasst werden. Dazu werden die Gleichungen 2.4 und 2.5 in 2.3 eingesetzt.

$$E_{instr} = E_{Base} + E_{InterInstr} + E_{Fetch} + E_{MemAccess} \tag{2.6}$$

Da die Bestimmung der Inter-Instruktions-Kosten einen erheblichen Messaufwand darstellt, wird das Modell auf folgende Weise vereinfacht. Für eine kleine Menge von Instruktionen werden diese Kosten bestimmt und als

Durchschnittswert zu den Basiskosten addiert. Es entsteht ein Kostenpunkt $E_{Base'}$, der die Kosten E_{Base} und $E_{InterInstr}$ ersetzt.

Auf die Kosten $E_{Base'}$ in 2.6 wird nun Gleichung 2.2 angewendet, wobei der darin vorkommende Strom I als Durchschnittswert über die Zeit zu betrachten ist, die zur Abarbeitung der gesamten Instruktion benötigt wird.

$$E_{instr} = U \cdot (I_{Base'}) \cdot (n + m + r) \cdot t_c + E_{Fetch} + E_{MemAccess} \quad (2.7)$$

Hierbei entspricht n der Anzahl der Taktzyklen, die zur Abarbeitung der Instruktion benötigt werden, m der Anzahl der Waitstates, die beim Laden der Instruktion auftreten und r der Anzahl der Waitstates, die anfallen, falls die Instruktion ein Datum liest oder schreibt.

Nimmt man die Kosten für jede in einem Programm vorkommende Instruktion $E_{instr,i}$ und multipliziert sie mit der Anzahl ihrer Ausführungen N_i, dann erhält man den Energieverbrauch des gesamten Programms:

$$E_{prg} = \sum_i E_{instr,i} \cdot N_i$$

Gleichung 2.7 zeigt, was ein Compiler tun kann, um den Energieverbrauch eines Programms zu senken: Geht man davon aus, daß der Prozessor mit einer konstanten Spannung U arbeitet und die Taktrate t_c konstant ist, dann muß auf die fließenden Ströme und die Anzahl der Taktzyklen und Waitstates Einfluß genommen werden.

1. Die Basiskosten der Instruktionen sollten, wo möglich, gesenkt werden. Es können z.B. teure Multiplikationsbefehle durch Shift-Operationen ersetzt werden.

2. Durch Umordnen der Instruktionen ist es möglich, die Inter-Instruktions-Kosten zu senken. Plaziert man Befehle, die dieselben Prozessor-Ressourcen nutzen, direkt hintereinander, dann können die Kosten für das Aktivieren und Deaktivieren der entsprechenden Ressourcen entfallen und somit die Schaltkreisaktivitäten reduziert werden. Hierbei ist darauf zu achten, daß das Programmverhalten erhalten bleibt.

3. Die durch E_{Fetch} und $E_{MemAccess}$ repräsentierten Kosten lassen sich senken, wenn häufig benötigte Daten und Instruktionen in Speicher mit niedriger Zugriffsenergie geladen werden. Durch das Halten von häufig benötigten Daten in Registern lassen sich zusätzliche Einsparungen erzielen.

4. Da die Anzahl der Taktzyklen n einer Instruktion ebenfalls eine Auswirkung auf den Energieverbrauch hat, sollten Instruktionen oder auch Instruktionssequenzen mit einer geringen Anzahl von Taktzyklen gewählt werden.

5. Durch Auslagern von Code und Daten in Speicher mit möglichst wenig Waitstates werden die Zeiten minimiert, in denen der Prozessor nur wartet, aber Strom für die Ausführung einer Instruktion fließt.

Wird ein Programm optimiert, um die in Punkt 2 genannten Inter-Instruktions-Kosten zu minimieren, kann dies zum vermehrten Auftreten von Ressource und Data Hazards führen. Hieran läßt sich sehen, daß das Anwenden einer Optimierung weitere Kosten entstehen lassen kann. Auch reicht im allgemeinen der Platz in den kostengünstigen Speichern nicht aus, um den ganzen Programmcode und alle Daten aufzunehmen. Es ist somit Aufgabe des Compilers, unter den anwendbaren Optimierungen diejenigen zu wählen, die den Gesamtenergiebedarf minimieren.

2.3 Dispatcher und Scheduler

Wie bereits in Abschnitt 1.1 erwähnt laufen auf den meisten Rechnersystemen häufig mehrere Prozesse parallel, die jeweils individuelle Aufgaben erledigen. Da aus Kostengründen meist nur ein Prozessor zur Verfügung steht, der zu einem bestimmten Zeitpunkt jeweils nur einen Prozess ausführen kann, muß die zur Verfügung stehende Rechenzeit zwischen den Prozessen geteilt werden. Dazu ist es notwendig, daß die Prozesse entweder selbst die Kontrolle über den Prozessor abgeben, oder ihnen diese entzogen wird. Diese beiden Verfahren werden nicht-preemptives bzw. preemptives Scheduling genannt.

Wird non-preemptives Scheduling verwendet, dann entscheidet jeder Prozess für sich selbst, wann er die Kontrolle über den Prozessor abgibt. Bei diesem Typ ist es möglich, daß ein Prozess allein die Kontrolle über den Prozessor behält (z.B. durch einen Programmfehler) und alle anderen Prozesse nicht abgearbeitet werden können.

Beim preemptiven Scheduling unterbricht das Betriebssystem nach einer vorgegeben Zeitspanne, der sogenannten Zeitscheibe (engl. timeslice), den aktuell laufenden Prozess und entscheidet, welcher Prozess als nächstes die Kontrolle über den Prozessor erlangen soll. Üblicherweise geschieht dies, indem periodisch ein Interrupt generiert wird. Der Vorteil dieses Verfahrens liegt darin, daß alle Prozesse Rechenzeit bekommen und die Auslastung der Ressourcen optimiert werden kann, vorausgesetzt die Auswahl des nächsten aktiven Prozesses ist fair und kein Prozess wird vernachlässigt.

Erlangt ein unterbrochener Prozess wieder die Kontrolle über den Prozessor, dann muß er alle von ihm verwendeten Daten und Prozessorregister in dem Zustand wiederfinden, wie es vor seiner Unterbrechung der Fall war. Der Teil des Betriebssystems, der für das Sichern und Wiederherstellen aller notwendigen Daten und Prozessorregister verantwortlich ist, wird Dispatcher genannt. Für jeden Prozess hält der Dispatcher einen PCB (Process Control Block), in dem folgenden Daten gesichert werden:

- die Prozesskennung
- der Prozesszustand
- der Inhalt des Status-Registers
- der Inhalt der Prozessor-Register
- Speicherverwaltungsinformationen
- Liste der geöffneten Dateien

und eventuell weitere Informationen.

Der für diese Arbeit implementierte Dispatcher verwendet den ARM-Befehlssatz. Laufen die Prozesse im User-Modus (siehe Abschnitt 2.1), dann können mit einer speziellen Variante des ldmia/stmia Instruktionspaares mit einem Befehl alle Register eines Prozesses gesichert bzw. wiederhergestellt werden.

Die Entscheidung darüber, welcher unter den ausführungsbereiten Prozessen als nächster die Kontrolle über den Prozessor erlangen soll, trifft der Scheduler. Es gibt verschiedene Scheduling-Strategien, wobei viele davon auf bestimmte Aufgabenstellungen und Problemtypen hin optimiert sind. Es werden hier drei gängige Verfahren vorgestellt [Tan92].

Round Robin Scheduling Die Prozesse werden in einer zirkulären Datenstruktur gespeichert. Ist ein Prozess beendet, erhält der nächste in der Datenstruktur folgende Prozess die Kontrolle über den Prozessor.

Priority Scheduling Jedem Prozess ist eine Priorität zugeordnet. Je höher die Priorität, desto wichtiger ist der Prozess und desto häufiger sollte ihm der Prozessor zugeteilt werden. Ist die Zeitscheibe eines Prozesses abgelaufen, wird unter den anderen Prozessen derjenige mit der höchsten Priorität als nächster auszuführender ausgewählt. Um zu verhindern, daß Prozesse mit sehr niedriger Priorität verhungern, also nie den Prozessor zugeteilt bekommen, wird deren Priorität über die Zeit hinweg erhöht.

Shortest Job First Scheduling Dieses Verfahren wird in der Stapelverarbeitung eingesetzt und liefert eine minimale mittlere Wartezeit. Es wird derjenige Prozess als nächster Auszuführender ausgewählt, der die geringste Restlaufzeit hat.

Beendet ein Prozess seine Ausführung, dann ruft er eine Terminate-Funktion auf. Dieses Funktion meldet den Prozess beim Betriebssystem ab und gibt den gesamten Speicher frei, den dieser Prozess belegt hat.

2.4 Integer Linear Programming

Integer Linear Programming (ILP) bezeichnet die Optimierung linearer Funktionen unter linearen Nebenbedingungen. Viele Optimierungsprobleme lassen sich in dieser Form darstellen, so auch die in Kapitel 3 vorgestellten Allokations-Strategien. In ihrer Standardform werden ILP-Gleichungen beschrieben durch

$$f(x) = w_1 x_1 + \cdots + w_n x_n \tag{2.8}$$

$$\begin{aligned} a_{i1}x_1 + \cdots + a_{in}x_n &\geq b_i & 1 \leq i \leq m & \quad (2.9)\\ x_j &\in \mathbb{Z} & 1 \leq j \leq n & \quad (2.10)\\ w_j &\in \mathbb{R} & 1 \leq j \leq n & \\ a_{ij} &\in \mathbb{R} & 1 \leq i \leq m,\ 1 \leq j \leq n & \end{aligned}$$

Gleichung 2.8 ist die Zielfunktion, die entweder minimiert oder maximiert werden soll, wobei die in den Gleichungen 2.9 und 2.10 angegebenen Nebenbedingungen eingehalten werden müssen. Häufig betrachtet man Probleme, für die der Wertebereich der x_j auf $\{0, 1\}$ eingeschränkt ist. Probleme dieser Form mit nur einer Nebenbedingung sind bereits NP-hart [Weg01], womit man nicht auf eine effiziente Lösung dieser Probleme für alle Eingaben hoffen kann. Zur Lösung von ILP-Gleichungen gibt es eine Vielzahl an Software. Der in dieser Arbeit eingesetzte Compiler encc benutzt das Rucksackproblem, um für eine Menge von Programmobjekten eine Optimierung durchzuführen. Das Rucksackproblem wird durch ILP-Gleichungen dargestellt und zur Lösung der ILP-Löser ILOG CPLEX der Firma ILOG genutzt [ILO].

2.5 Verwandte Arbeiten

Die in dieser Arbeit vorgestellten Allokations-Strategien können dazu genutzt werden, Kostenfunktionen von Multiprozess-Systemen, die über einen Scratchpad-Speicher verfügen, zu optimieren. Die häufigsten betrachteten Kostenfunktionen sind die Codegröße, der Energieverbrauch und die benötigte Ausführungszeit. In diesem Abschnitt werden Arbeiten vorgestellt, die mittels Hardware- und Software-Techniken versuchen, diese Kostenfunktionen zu optimieren. Einige dieser Arbeiten bilden die Grundlage für diese Arbeit, andere wiederum verwenden andere Ansätze um ein ähnliches Ziel zu erreichen.

Damit der Nutzen der Software-Techniken bewertet werden kann, sind Modelle für Prozessor und Speicher, mit denen Energieverbrauch und Ausführungs- bzw. Zugriffszeit berechnet werden können, von grundlegender Bedeutung. Tiwari et al. [TMW94] haben ein Energiemodell für Prozessoren vorgestellt. Dieses Modell benötigt keine Informationen über den Aufbau

und die Realisierung des Prozessors, sondern betrachtet die bei der Ausführung von Instruktionen fließenden Ströme. Jeder Instruktion werden Grundkosten zugeordnet, die bei ihrer Ausführung anfallen. Die Abarbeitung einer Folge von Befehlen verursacht einmal Kosten durch die entstehende Schaltkreisaktivität beim Wechsel zwischen den Befehlen und weiterhin, wenn die einzelnen Befehle auf unterschiedliche Prozessor-Ressourcen zugreifen und diese aktiviert bzw. deaktiviert werden müssen. Dieser Kostentyp wird als Inter-Instruktions-Kosten bezeichnet und mit in das Modell aufgenommen. Für diesen Kostentyp zeigte sich, daß er sich für sämtliche Instruktionspaare in einem gewissen Bereich bewegt und wird deshalb als Durchschnittswert auf die Grundkosten aufgeschlagen. Die Daten für dieses Modell wurden erstmalig für einen Intel 486DX2-S gemessen und es wurde festgestellt, daß die Abweichung zwischen den vorhergesagten und gemessenen Kosten nicht mehr als 3% beträgt. Das Modell von Tiwari wurde durch Steinke et al. [SKWM01] so erweitert, daß der Energieverbrauch im Prozessor, das Bit-Toggling auf den Bussen und der Energieverbrauch des Speichers berücksichtigt werden. Letzterer wird mit in das Modell aufgenommen, da ein Großteil der Energie im Speicher verbraucht wird.

Ein Modell für Caches wurde von Wilton und Jouppi [WJ96] eingeführt. Die auf diesem Modell basierende Software erlaubt die Berechnung der Zugriffszeiten und des Energieverbrauchs von Caches, wobei als Eingabeparameter die Cache- und Block-Größe, die Assoziativität und die Technologie-Größe dienen.

Simunic et al. [SBM99] modellieren den Energieverbrauch eines Systems, das über einen ARM-Prozessor verfügt. Dabei werden die einzelnen Komponenten des Systems durch Erweiterungen des ARMulators [ARM01b] nachgebildet. Die Informationen über den Energieverbrauch der einzelnen Komponenten werden aus den zugehörigen Datenblättern entnommen, wobei hier nur zwischen den beiden Zuständen aktiv und inaktiv unterschieden wird. Für jeden Taktzyklus wird ermittelt, in welchem Zustand sich die Komponenten befinden. Eine Summation über die zugehörigen Verbrauchswerte liefert den Gesamtenergieverbrauch für die Ausführung eines Programms. Dieses Modell erlaubt es, während der Entwurfsphase Veränderungen an den Komponenten (z.B. Größe der eingesetzten Speicher) hinsichtlich des Energieverbrauchs zu bewerten.

Wendet man diese Modelle an, dann können die Auswirkungen von Optimierungen beurteilt werden. Die folgenden zwei Arbeiten konzentrieren sich dabei auf den Energieverbrauch des Prozessors.

Valluri und John [VJ01] haben für den DEC Alpha 21064 Prozessor untersucht, ob eine Optimierung, die eine höhere Ausführungsgeschwindigkeit zum Ziel hat, auch den Energieverbrauch im Prozessor sinken läßt. Dabei zeigte sich, daß Compiler-Optimierungen, die die Anzahl der ausgeführten Instruktionen vermindern, in der Regel auch den Energieverbrauch verbessern. Bei Optimierungen, die Parallelausführung zum Ziel haben, erhöht sich

aber meist die durchschnittlich verbrauchte Leistung.

Auf Parallelausführung von Instruktionen sind besonders VLIW-Architekturen ausgerichtet. Das Bit-Toggling auf dem Instruktions-Bus, das durch aufeinanderfolgende VLIW-Instruktionen hervorgerufen wird, hat eine Auswirkung auf den Energieverbrauch des Systems. Lee et al. [LLHT00] versuchen durch Umordnen von Instruktionen das Bit-Toggling zu minimieren. Dies kann auf zwei Arten geschehen: horizontal und vertikal. Bei der horizontalen Umordnung werden die Mikroinstruktionen der VLIW-Instruktionen vertauscht. Beim vertikalen Umordnen können die Mikroinstruktionen zwischen verschiedenen VLIW-Instruktionen getauscht werden.

Wie bereits erwähnt, wird ein Großteil der Energie beim Zugriff auf Speicher verbraucht. Es werden nun mehrere Arbeiten vorgestellt, die mit unterschiedlichen Ansätzen versuchen, diesen Kostentyp zu minimieren.

Inwieweit sich Energieeinsparungen durch das Abschalten von ungenutzten Speichern erzielen lassen, untersuchen Kandemir und Kolcu [KK02]. Es wird ein System mit mehreren Speicherbänken betrachtet, für die Stromsparzustände aktiviert werden können. Durch Schleifentransformationen soll erreicht werden, daß während der Programmausführung auf möglichst viele Speicherbänke über einen längeren Zeitraum nicht zugegriffen wird und diese somit in einen Stromsparzustand versetzt werden können. Zur Laufzeit wird mittels einer speziellen Hardware entschieden, welche Speicherbank in welchen Stromsparzustand geschaltet werden soll.

Kim et al. [KIVK00] untersuchten den Einfluß von häufig verwendeten Standard-Compiler-Optimierungen (Common Subexpression Elimination, Algebraic Simplification, Constant Propagation, Copy Propagation, Scalar Replacement, Loop Invariant Optimizations, Outer Loop Unrolling, Loop Interchange, Loop Fusion und Loop Tiling) auf den Energieverbrauch im Speicher. Dabei zeigte sich, daß diese Optimierungen die Datenlokalität erhöhen und damit auch den Energieverbrauch durch Datenzugriffe senken, aber der Energieverbrauch durch Zugriffe auf Instruktionen ansteigt. Als Konsequenz muß bei Systemen, für die ein möglichst geringer Energieverbrauch angestrebt wird, auf Lokalität sowohl bei Daten als auch bei Instruktionen geachtet werden.

Den Energieverbrauch im Instruktions-Cache versuchen Krishnaswamy und Gupta [KG02] dadurch zu senken, indem Funktionen entweder als Thumb- oder als ARM-Code übersetzt werden. Da der Thumb-Befehlssatz weniger Befehle zur Verfügung stellt, wird zum Teil Code generiert, der eine längere Ausführungszeit als sein ARM-Äquivalent hat, was auch den Energieverbrauch erhöhen kann. Es werden mehrere Heuristiken vorgeschlagen, anhand derer entschieden werden soll, ob eine Funktion als Thumb- oder ARM-Code zu übersetzen ist. Weiterhin wurde untersucht, ob durch das Mischen von Thumb- und ARM-Code innerhalb von Funktionen weitere Einsparungen erzielt werden können, es zeigten sich aber keine wesentlichen Verbesserungen gegenüber dem Funktions-basierten Ansatz.

Ghosh et al. [GMM99] versuchen die Daten-Cache-Miss-Rate zu senken. Es wird ein System von diophantischen Gleichungen aufgestellt, das einen Zusammenhang zwischen den in Schleifen vorkommenden Indizes, den Array-Größen und -Basisadressen und den Cache-Parametern herstellt. Die Lösungen der Gleichungen entsprechen der Anzahl der Daten-Cache-Misses. Durch Integration dieser Gleichungen in den Compiler erhält dieser Anhaltspunkte dafür, wie die Basisadressen und Spaltengrößen für Arrays bzw. die Blockgrößen beim Teilen von Schleifen gewählt werden müssen, um die Anzahl der Daten-Cache-Misses zu minimieren.

Die Plazierung von Basisblöcken im Speicher, mit dem Ziel Cache-Konflikte zu vermeiden, untersuchen Tomiyama und Yasuura [TY96]. Als Eingabe werden gewichtete Kontrollflußgraphen betrachtet. Zuerst wird entschieden, welche Basisblöcke hintereinander plaziert werden sollen. Die hieraus entstehenden Folgen von Basisblöcken werden Traces genannt. In einem zweiten Schritt werden die Traces so im Speicher verteilt, daß die Anzahl der Cache-Konflikte minimiert wird. Verma et al. [VWM04a] untersuchen ein System, das sowohl einen Cache als auch ein Scratchpad enthält. Ziel ist es, durch Auslagern von Traces in den Scratchpad-Speicher die Anzahl der Cache-Konflikte zu minimieren. Gelöst wird das Problem, indem über der Menge der Traces ein Konflikt-Graph gebildet und auf Basis dieser Informationen ein ILP-Modell aufgestellt wird. Die Zielfunktion des Modells beschreibt den Energiebedarf und die Belegung der Entscheidungsvariablen die Plazierung der Traces im Speicher.

Ob Scratchpad-Speicher eine Alternative zu Caches darstellen, haben Banakar et al. [BSL+02] untersucht. Es werden Modelle verwendet, mit denen sich die Größe und die Energie pro Zugriff von Caches und Scratchpads berechnen lassen. Das Modell des Scratchpads entspricht dem des Caches, nur wurden hier alle zum Tag-Speicher gehörenden Komponenten weggelassen. Die für diese Arbeit durchgeführten Experimente zeigen, daß ein Scratchpad bei gleicher Speichergröße weit weniger Fläche in Anspruch nimmt als ein Cache und auch die Ausführungsgeschwindigkeit gesteigert und der Energieverbrauch gesenkt werden kann.

Cooper und Harvey [CH98] schlagen einen speziellen Speicher vor, mit dem durch Spill-Code hervorgerufene Kosten reduziert werden können. Um den Prozessor gut auszulasten, versuchen einige Optimierungen möglichst viele Ressourcen des Prozessors zu nutzen. Dies und Optimierungen, die auf Datenlokalität ausgelegt sind, erhöhen die Menge des vom Compiler erzeugten Spill-Codes. Als Ausweg sollen möglichst viele Daten, die nicht in Registern gehalten werden können, in einen energieeffizienten Speicher ausgelagert werden. Es wird eine Compilererweiterung zur Auswahl der Daten für den energieeffizienten Speicher vorgeschlagen.

Der nächste Schritt nach der Betrachtung von Spill-Werten besteht darin, Code und Daten ins Scratchpad auszulagern. Steinke et al. [SWLM02] benutzen diesen Ansatz, um energieeffiziente Applikationen zu generieren.

Es wird für jedes der Code- und Datenobjekte der Nutzen bei einer Auslagerung ins Scratchpad bestimmt. Der Nutzen der Programmobjekte, deren Größe und die Größe des Scratchpad-Speichers bilden eine Eingabe für das bekannte Rucksackproblem [Weg99]. Ausgabe ist eine Verteilung der Programmobjekte auf das Scratchpad und den Hauptspeicher. Weitere Energieeinsparungen erzielen Verma et al. [VSM03], indem das Teilen von Arrays betrachtet wird. Es wird bewertet, für welche der in der Applikation vorkommenden Arrays es sich lohnt, einen Teil davon in den Scratchpad-Speicher auszulagern. Für diesen Ansatz ist es notwendig, daß für die geteilten Arrays der Code der Applikation angepaßt wird, damit stets auf die richtige Speicheradresse zugegriffen wird.

Einen anderen Ansatz für die Aufteilung des Scratchpads verfolgen Angiolini et al. [ABC03]. Hier wird nicht die Software dem Scratchpad angepasst, sondern das Scratchpad der Software. Für eine fertig erstellte Binärdatei wird der Execution-Trace analysiert. Mit Hilfe eines Algorithmus wird bestimmt, welche Speicherzellen zum Scratchpad gehören sollten. Auf Hardwareebene muß der korrekte Speicherzugriff auf die verschiedenen Speichertypen durch einen Adressdecoder sichergestellt werden. Als Nachteile der Software-orientierten Methoden wird die Grobheit der Verfahren genannt, da nur ganze Basisblöcke, Funktionen und Arrays auslagert werden können. Der Nachteil dieser Methode wiederum ist, daß eine einmal angepaßte Hardware nicht mehr optimal für ein eventuell durchgeführtes Software-Update ist.

Die bisher vorgestellten Verfahren zur Auslagerung von Code und Daten ins Scratchpad waren statisch, d.h. die einmal ausgelagerten Programmobjekte verbleiben für die gesamte Ausführungszeit im Scratchpad. In [SGW+02] wird das dynamische Kopieren von Code ins Scratchpad betrachtet. Dadurch können mehr Objekte das Scratchpad nutzen, wodurch der Energieverbrauch weiter gesenkt werden kann. Vermindert wird der Nutzen durch die entstehenden Kopierkosten. Als Objekte, die einen hohen Nutzen durch die Auslagerung ins Scratchpad erzielen, werden Schleifen untersucht, da ein Großteil der Ausführungszeit auf diese Programmobjekte entfällt und somit ein großer Teil der Energie dort verbraucht wird.

Viele Applikationen aus der Signal- oder Videoverarbeitung arbeiten auf großen Arrays mit teilweise mehrfach verschachtelten Schleifen. Kandemir et al. [KRI+01] betrachten Compileroptimierungen, die Schleifentransformationen einsetzen. Während der Abarbeitung von Schleifen sollen Teile von Arrays ins Scratchpad gebracht und nach der Bearbeitung in den Hauptspeicher zurück kopiert werden. Dafür wird bestimmt, in welchen Iterationen Teile von Arrays erneut verwendet werden. In einer Erweiterung [KC02] zu dieser Arbeit wird eine Hierarchie von Speichern mit unterschiedlicher Zugriffsenergie betrachtet. Es muß nun entschieden werden, in welchen der Speicher Teile der Arrays kopiert werden sollen, um den Gesamtenergiebedarf zu optimieren. Die Ergebnisse dieser Arbeit können auch für den

Entwurf einer Speicherhierarchie genutzt werden.

Poletti et al. [FMA+04] betrachten das dynamische Kopieren von Daten ins Scratchpad, wobei dieses Verfahren auch für Multitasking-Systeme verwendet werden kann. Der Ansatz ist eine Mischung aus einer Software-API, die Programme nutzen können und einer Hardware-Erweiterung. Ein DMA-Controller steuert das Kopieren von Daten zwischen Hauptspeicher und Scratchpad und hilft, die Kopierkosten so gering wie möglich zu halten. Die Software-API besteht aus einem konfigurierbaren dynamischen Memory-Manager, der versucht die Fragmentierung des Speichers gering zu halten und für Speicher-Allokations-Anfragen den besten passenden Block bestimmt.

Die dynamische Auslagerung von Code und Daten in den Scratchpad-Speicher untersuchen Verma et al. [VWM04b]. Um zu entscheiden, welche Programmobjekte zu welchem Zeitpunkt ausgelagert werden sollen, wird eine Liveness-Analyse durchgeführt. Mit den hieraus gewonnenen Daten wird in einem ersten Schritt bestimmt, welche Programmobjekte wann in den Scratchpad-Speicher ausgelagert werden sollen. In einem zweiten Schritt werden die Adressen berechnet, an die die Programmobjekte in das Scratchpad kopiert werden.

Zum Abschluß wird noch ein Scratchpad-basiertes Verfahren zur Minimierung der Ausführungszeit von Applikationen vorgestellt. Die dem Ansatz von Ozturk et al. [OKD+04] zugrunde liegende Idee ist besonders interessant, da sie mit der Verwendung von Kompressionstechniken auf bereits seit längerer Zeit erforschte Verfahren zurückgreift. Dabei wird auf der Basis der Wiederverwendung von Teilen von Arrays versucht, komprimierte Teile der Arrays vom Hauptspeicher in das Scratchpad zu bringen und diese, wenn ein Zugriff erfolgt, dort zu dekomprimieren. Um zu häufiges Auslagern zurück in den Hauptspeicher zu vermeiden, werden nicht mehr benutzte Teile wieder im Scratchpad komprimiert, um Platz für andere Daten zu schaffen. Die Entscheidung darüber, welcher Datenblock ins Scratchpad kopiert wird und wie lang dort gehalten wird, trifft der Compiler. Die von ihm erzeugten Tabellen werden zur Laufzeit ausgewertet und die entsprechenden Aktionen durchgeführt.

Kapitel 3

Allokations-Strategien für mehrere Prozesse

In diesem Kapitel werden die Allokations-Strategien SAMP, DAMP und HAMP vorgestellt. Aufgabe dieser Strategien ist es, für ein System, das mit einem Scratchpad-Speicher ausgestattet ist und auf dem mehrere Prozesse laufen, eine Kostenfunktion bezüglich dieser Prozesse zu optimieren. Als Kostenfunktion können z.B. der Energieverbrauch des Multiprozess-Systems oder die Gesamtlaufzeit gewählt werden. Wie bereits in der Einleitung erwähnt, stellt die Minimierung des Energieverbrauchs von eingebetteten Systemen momentan einen wichtigen Zweig der Forschung dar. Um die praktische Anwendbarkeit und den Nutzen der hier vorgestellten Allokations-Strategien bewerten zu können, wird aus diesem Grunde der Energieverbrauch als Kostenfunktion gewählt.

Im folgenden Abschnitt 3.1 wird ein Beispiel für ein Multiprozess-System vorgestellt, anhand dessen die Idee und die Funktionsweise der Allokations-Strategien in den folgenden Abschnitten erklärt wird.

Die in den Abschnitten 3.2 und 3.3 vorgestellten Allokations-Strategien sind vom Ansatz her verschieden. Die Allokations-Strategie SAMP teilt den Scratchpad-Speicher statisch unter den Prozessen auf, DAMP hingegen erlaubt es den Prozessen, das ganze Scratchpad zu nutzen, allerdings müssen die ausgelagerten Daten bei jeder Unterbrechung eines laufenden Prozesses durch den Dispatcher gesichert bzw. wiederhergestellt werden. Die HAMP-Allokations-Strategie, die in Abschnitt 3.4 eingeführt wird, stellt eine Kombination der ersten beiden Verfahren dar und versucht sich die Stärken dieser beiden Verfahren zunutze zu machen. Trifft man bei HAMP entsprechende Einschränkungen, dann erhält man als Spezialfälle die Allokations-Strategien SAMP und DAMP. Es wird sich weiterhin zeigen, daß aus theoretischer Sicht der kombinierte Ansatz schwieriger anzuwenden ist.

Prozess	0 kB SP-Anteil	4 kB SP-Anteil
mpegdec	90	57
receive	30	18
ui	15	8

Tabelle 3.1: Energieverbrauch in mJ bei 0 kB und 4 kB Scratchpad-Anteil

3.1 Vorbetrachtungen

Als einführendes Beispiel soll der Energieverbrauch eines mobilen Endgeräts, im weiteren Verlauf Handheld genannt, optimiert werden. Der Handheld wird dazu genutzt, ein Video zu empfangen und darzustellen. Insgesamt laufen auf diesem Gerät drei Prozesse: Der Prozess **receive** dient dazu, die Videodaten zu empfangen. **mpegdec** ist dafür zuständig, den empfangenen Datenstrom zu dekodieren, das Videobild anzuzeigen und die Soundausgabe zu produzieren. Der Prozess **ui** ist für die Darstellung des Benutzer-Interface und die Abfrage der Tasten des Gerätes zuständig. Der Handheld verfügt über einen 4 kB großen Scratchpad-Speicher, der in Einheiten von 1 kB belegt werden kann.

In Abschnitt 2.5 wurde der Ansatz von Steinke et al. [SWLM02] vorgestellt, mit dem es möglich ist, den Energieverbrauch einer Applikation unter Verwendung eines Scratchpads zu minimieren. Um nun den Energieverbrauch dieses Handhelds zu senken, könnte man wie folgt vorgehen:

1. Für jeden Prozess wird der Energiebedarf für das 4 kB große Scratchpad minimiert und der Energiebedarf berechnet, der entsteht, wenn der Prozess keinen Anteil am Scratchpad erhält. Anschließend wird für jeden Prozess ermittelt, um welchen Betrag sein Energieverbrauch sinkt, wenn man ihm das gesamte Scratchpad zuweist.

2. Derjenige Prozess, für den der Energieverbrauch um den größten Betrag gesunken ist, darf das ganze Scratchpad nutzen.

In Tabelle 3.1 ist für jeden Prozess die Energie angegeben, die er bei voller Nutzung des Scratchpad und ohne Scratchpad verbraucht. Wird nun das eben vorgestellte Verfahren angewendet, dann zeigt sich, daß der Prozess **mpegdec** den größten Nutzen daraus zieht, wenn er den gesamten Scratchpad-Speicher erhält. Der Energieverbrauch des Handhelds wäre in diesem Fall 102 mJ.

Es stellt sich nun die Frage, ob es möglich ist, noch mehr Energie zu sparen. Die Idee der in den folgenden Abschnitten vorgestellten Allokations-Strategien ist es, Anteile des Scratchpads an die einzelnen Prozesse zu vergeben und so den Energieverbrauch zu senken. Hierfür muß allerdings die Energie bekannt sein, die die Prozesse verbrauchen, wenn sie eine bestimmte Menge des Scratchpad-Speichers zugewiesen bekommen.

Prozess	0 kB	1 kB	2 kB	3 kB	4kB
mpegdec	90	84	63	63	57
receive	30	27	24	18	18
ui	15	9	8	8	8

Tabelle 3.2: Energiefunktionen der Prozesse

Für die drei Prozesse des Handheld muss also im ersten Schritt der Energieverbrauch bei einem Anteil von 0 kB, 1 kB, 2 kB, 3 kB und 4 kB am Scratchpad-Speicher gemessen werden. Dazu wird für jeden Anteil eine Optimierung des jeweiligen Prozesses mit dem Ansatz von Steinke durchgeführt, wobei eine Auswahl an Code- und Daten-Objekten zur Auslagerung in den Scratchpad-Speicher bestimmt wird. Der Energieverbrauch der resultierenden Binärdatei wird mit dem Profiler enprofiler bestimmt. Die durch diese Messung ermittelten Daten werden im weiteren Verlauf Energiefunktionen genannt. Tabelle 3.2 zeigt die Energiefunktionen der drei Prozesse des Handheld.

3.2 Statische Allokation für mehrere Prozesse (SAMP)

Die Idee hinter der Allokations-Strategie SAMP ist, daß ein Aufteilen der Ressource Scratchpad unter den beteiligten Prozessen eine größere Energieeinsparung bringen sollte, als dies bei der alleinigen Nutzung des Scratchpads durch einen Prozess der Fall ist. Betrachtet man die in Tabelle 3.2 aufgeführten Energiefunktionen, dann wird deutlich, daß sich ein geringerer Energieverbrauch erreichen läßt, wenn man **mpegdec** und **ui** jeweils 2 kB Scratchpad-Speicher zuweist. Der Energieverbrauch beträgt in diesem Fall 101 mJ.

Die Problemstellung läßt sich nun wie folgt formalisieren: Gegeben seien n Prozesse, ihre Energiefunktionen $f_i(x)$ und ein Scratchpad der Größe M. Gesucht ist eine Aufteilung des Scratchpad unter den Prozessen, die den Gesamtenergieverbrauch f_{HH} des Handheld minimiert:

$$f_{HH} = min\{f_1(x_1) + \cdots + f_n(x_n) \mid x_1 + \cdots + x_n \leq M\} \tag{3.1}$$

Die x_i in Gleichung 3.1 legen den Anteil fest, den Prozess i am Scratchpad hält. Wendet man Gleichung 3.1 auf die Prozesse des Handheld an, dann erhält man als minimalen Energieverbrauch 99 mJ, wobei **mpegdec** 2 kB, **receive** 1 kB und **ui** ebenfalls 1 kB an Scratchpad-Speicher erhält.

3.2.1 ILP

Bevor ein Algorithmus zur Lösung des eben definierten Minimierungsproblems vorgestellt wird, soll die Allokations-Strategie SAMP durch ILP-Gleichungen beschrieben werden.

n	Anzahl der Prozesse
M	Größe des Scratchpad
e_{ij}	Energieverbrauch von Prozess i bei j Scratchpadanteilen
x_{ij}	binäre Entscheidungsvariable, die den Wert 1 hat, wenn Prozess i j Scratchpadanteile nutzt

Die zu minimierende Zielfunktion lautet

$$\sum_{i=1}^{n}\sum_{j=0}^{M} x_{ij} \cdot e_{ij} \rightarrow minimiere$$

Da jeder Prozess nur einen Anteil am Scratchpad erhält, gilt folgende Einschränkung für die x_{ij}

$$\forall i: \quad \sum_{j=0}^{M} x_{ij} = 1$$

Weiterhin muß dafür gesorgt werden, daß die Summe aller Scratchpadanteile dessen Größe nicht übersteigt

$$\sum_{i=1}^{n}\sum_{j=0}^{M} x_{ij} \cdot j \leq M$$

3.2.2 Algorithmische Lösung

Zur Entwicklung eines Algorithmus für die bestmögliche Aufteilung des Scratchpad-Speichers bietet es sich an, zuerst den Fall nur zweier Energiefunktionen zu betrachten. Abbildungen 3.1 (a) und (b) zeigen die Energiefunktionen der beiden Prozesse **mpegdec** und **receive**.

Da das Scratchpad unter den beiden Prozessen aufgeteilt werden soll, könnte man in einem ersten Ansatz so vorgehen, daß alle Aufteilungen dieses Speichers unter den beiden Prozessen betrachtet werden. Jede Aufteilung teilt den gesamten Scratchpad-Speicher in zwei Teile. Betrachtet man nun alle möglichen Aufteilungen und berechnet jeweils anhand der Energiefunktionen den entstehenden Gesamtenergieverbrauch, dann erhält man eine kombinierte Energiefunktion. Sind zwei an je M Stellen definierte Energiefunktionen gegeben, dann ist die zugehörige kombinierte Energiefunktion ebenfalls an M Stellen definiert, da es insgesamt M zulässige Aufteilungen

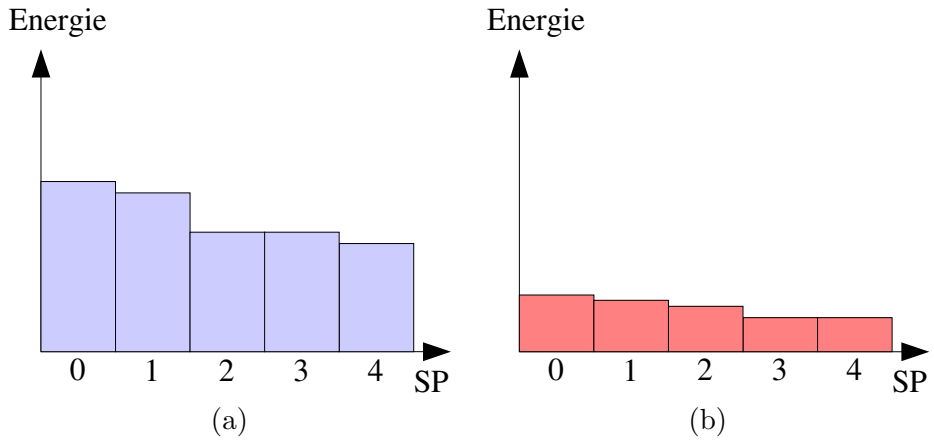

Abbildung 3.1: (a) Energiefunktion des **mpegdec** Prozesses, (b) Energiefunktion des **receive** Prozesses

eines Speichers der Größe M in zwei Teile gibt. Abbildung 3.2 zeigt die kombinierte Energiefunktion der Prozesse **mpegdec** und **receive**. Der erste Balken zeigt den Energieverbrauch, der entsteht, wenn **mpegdec** keinen Anteil am Scratchpad bekommt und **receive** den ganzen Scratchpad-Speicher nutzen darf. Der nächste Balken zeigt die Energie für die Aufteilung 1 kB für **mpegdec** und 3 kB für **receive** usw. Hat man Energiefunktionen mit n Werten gegeben, dann kann in Zeit $O(n)$ der minimale Energieverbrauch für zwei Prozesse bestimmt werden.

Dieses Verfahren funktioniert allerdings nur, wenn die Energiefunktionen monoton fallend sind. Für den Fall monoton fallender Funktionen kann sehr einfach gezeigt werden, daß nur die eben erwähnten Aufteilungen untersucht werden müssen.

Gegeben seien zwei Funktionen $f(x)$ und $g(x)$, die an n Stellen definiert sind und für die folgendes gilt

$$f(x_1) \geq f(x_2) \geq \cdots \geq f(x_n) \tag{3.2}$$

und

$$g(x_1) \geq g(x_2) \geq \cdots \geq g(x_n)$$

Gesucht ist ein minimaler Wert $f(x_i)+g(x_j)$ mit $i+j \leq n+1$. Das Minimum kann dann wie folgt bestimmt werden

$$min\{f(x_1) + g(x_n), f(x_2) + g(x_{n-1}), \ldots, f(x_n) + g(x_1)\} \tag{3.3}$$

Für alle in Gleichung 3.3 vorkommenden Werte $f(x_i)+g(x_j)$ gilt $i+j = n+1$. Es soll nun durch einen Widerspruchsbeweis gezeigt werden, daß

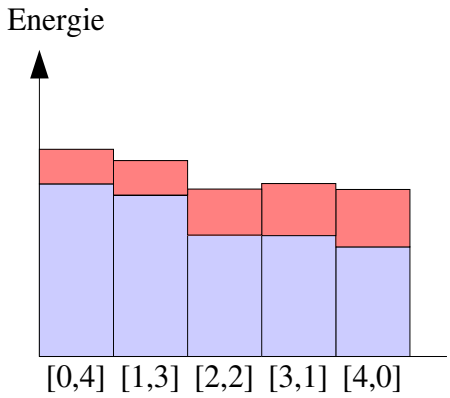

Abbildung 3.2: Kombinierte Energiefunktion von **mpegdec** und **receive**

es ausreicht, diese Werte zu untersuchen. Nimmt man an, daß es ein Paar $f(x_i) + g(x_j)$ mit $i + j < n + 1$ gibt, das unter allen zulässigen Paaren den kleinsten Wert hat, dann erhält man mit $k = (n + 1) - (i + j)$ und der Voraussetzung in Gleichung 3.2 einen Widerspruch zur Minimalitätsannahme von $f(x_i) + g(x_j)$: Da $f(x_i) \geq f(x_{i+k})$ gilt nämlich $f(x_i) + g(x_j) \geq f(x_{i+k}) + g(x_j)$.

In Tabelle 3.3 wurden die Energiefunktionen der Prozesse **mpegdec** und **receive** so abgeändert, daß sie nicht mehr monoton fallend sind. Die letzte Zeile zeigt die in Gleichung 3.3 auftretenden Werte, unter denen das Minimum gesucht wird. Würde man allerdings in diesem Fall 2 kB Scratchpad-Speicher an **mpegdec** und 1 kB an **receive** vergeben, dann erhält man mit 84 mJ eine bessere Lösung. Hiermit zeigt sich, daß dieser erste Ansatz nur für monoton fallende Kostenfunktionen eingesetzt werden kann. Da die Allokations-Strategie aber nicht auf diesen Typ von Funktionen beschränkt werden soll, muß ein anderer Weg gefunden werden.

Prozess	0 kB	1 kB	2 kB	3 kB	4 kB
mpegdec	90	84	61	63	57
receive	30	23	24	18	18
Kombinierte E-Fkt	108	102	85	86	87

Tabelle 3.3: Nicht monoton fallende Energiefunktionen

Versucht man sich zu verdeutlichen, warum das Minimum im obigen Beispiel nicht gefunden werden konnte, dann ergibt sich ein neuer Ansatz das Problem zu lösen.

Der minimale Energieverbrauch wird für eine Aufteilung erreicht, die nicht den gesamten Scratchpad-Speicher in Anspruch nimmt. Es muß also auch untersucht werden, ob auf einem Teil des Scratchpad-Speichers ein optimaler Energieverbrauch erreicht werden kann. Abbildung 3.3 verdeutlicht

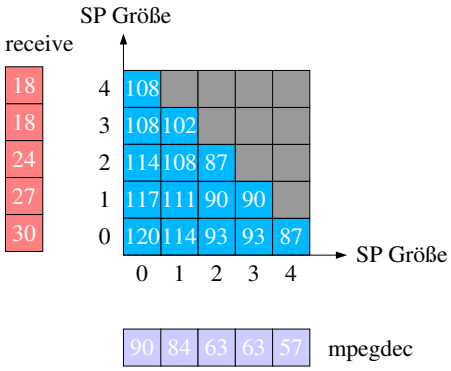

Abbildung 3.3: Kombinierte Energiefunktion für jede Scratchpad-Größe

das neue Vorgehen.

Am linken und am unteren Rand werden die Energiefunktionen der beiden Prozesse **mpegdec** und **receive** gezeigt, so wie sie in Tabelle 3.2 definiert wurden. In der Matrix ist für jede mögliche Aufteilung des Scratchpads in zwei Teile, wobei die Summe dieser beiden Teile nicht größer als das Scratchpad sein darf, der Energieverbrauch eingetragen. Der Eintrag in der linken unteren Ecke zeigt den Energieverbrauch, wenn beide Prozesse keinen Scratchpad-Speicher erhalten. Der Eintrag rechts daneben steht für den Fall, daß **mpegdec** 1 kB Scratchpad nutzen darf und **receive** keinen Anteil erhält.

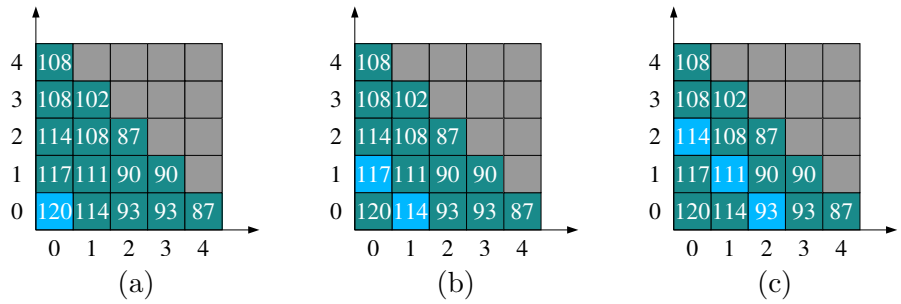

Abbildung 3.4: (a) Keiner der Prozesse erhält Anteile am Scratchpad, (b) 1 kB Scratchpad darf genutzt werden, (c) 2 kB Scratchpad dürfen genutzt werden

Die Abbildungen 3.4 (a)-(c) verdeutlichen, wie das neue Verfahren arbeitet. Am Anfang beträgt der minimale Energieverbrauch 120 mJ, dies entspricht dem Fall, daß beide Prozesse keinen Anteil am Scratchpad erhalten (Abbildung 3.4 (a)). Dann wird der Fall betrachtet, daß 1 kB des Scratchpads genutzt werden kann (Abbildung 3.4 (b)). Diesen Speicher kann entweder **mpegdec** oder **receive** nutzen. Der Energieverbrauch ist am geringsten,

	0 kB	1 kB	2 kB	3 kB	4 kB
Kombinierte E-Fkt	120	114	93	90	87

Tabelle 3.4: Kombinierte Energiefunktion $h(x)$

wenn **mpegdec** diesen Speicher zugewiesen bekommt und liegt unter dem, wenn beide Prozesse keinen Scratchpad-Speicher bekommen. Also ist die beste Lösung für einen Anteil von 1 kB am Scratchpad-Speicher 114 mJ, wobei **mpegdec** diesen Anteil erhält. Die nächste Möglichkeit besteht darin, die Prozesse 2 kB Scratchpad nutzen zu lassen. Hierbei können die 2 kB von einem Prozess allein genutzt werden oder von beiden Prozessen gemeinsam, wobei jeder 1 kB Scratchpad-Speicher erhält (Abbildung 3.4 (c)). Wieder ist es am besten, **mpegdec** den ganzen Speicher nutzen zu lassen.

Das Verfahren wird nun für jede mögliche Teilgröße des Scratchpads iteriert und dabei eine kombinierte Energiefunktion $h(x)$ aufgebaut. An der Stelle x_i liefert $h(x_i)$ den minimalen Energieverbrauch der beiden Prozesse, wobei sie maximal x_i kB Scratchpad-Speicher gemeinsam nutzen dürfen. Tabelle 3.4 zeigt diese Funktion für die beiden Prozesse **mpegdec** und **receive**. Algorithmus 1 zeigt die Minimumsberechnung im Pseudocode.

Algorithm 1 Kombinierte Energiefunktion $h(x)$ berechnen

Require: Energiefunktionen $f(x)$ und $g(x)$ für ein Scratchpad der Größe M
Ensure: $h(y) = min\{f(x_j) + g(x_k) \mid x_j + x_k \leq y\}$

```
1: min = ∞
2: for i = 0 to M do
3:    for j = 0 to i do
4:       if f(j) + g(i − j) < min then
5:          min = f(j) + g(i − j)
6:       end if
7:    end for
8:    h(i) = min
9: end for
10: Return h(x)
```

Die äußere Schleife iteriert über alle möglichen Scratchpad-Größen. In der inneren Schleife wird für jede dieser Größen die beste Unterteilung bestimmt. Diese wird nachher mit der besten bisherigen Lösung verglichen und das Minimum in die kombinierte Energiefunktion $h(x)$ übernommen.

Betrachtet man die in den Abbildungen 3.4 (a)-(c) gezeigten Matrizen, dann sieht man, daß der Algorithmus $M \cdot (M+1)/2$ Einträge dieser Matrizen untersucht. Da jeder Eintrag in Zeit $O(1)$ berechnet werden kann, beträgt die Rechenzeit dieses Verfahrens $O(M^2)$.

Im nächsten Schritt muß nun versucht werden, für den Fall von n Ener-

giefunktionen den optimalen Energieverbrauch für diese Art der Aufteilung des Scratchpads zu berechnen. Verdeutlicht man sich, was die kombinierte Energiefunktion $h(x)$ ausdrückt, nämlich den minimalen Energieverbrauch zweier Prozesse bei einer bestimmten Menge von Scratchpad-Speicher, dann sieht man, daß hier eigentlich dieselbe Information wie in den ursprünglichen Energiefunktionen dargestellt wird. Für den Fall der drei Prozesse des Handhelds liegt es deshalb nahe, Algorithmus 1 auf die kombinierte Energiefunktion $h(x)$ und die Energiefunktion des Prozesses **ui** anzuwenden.

Für den allgemeinen Fall von n Prozessen, die durch ihre Energiefunktionen repräsentiert werden, zeigt Algorithmus 2 die Berechnung des minimalen Energieverbrauchs bei einer statischen Aufteilung des Scratchpads.

Algorithm 2 Minimalen Energieverbrauch für n Prozesse berechnen

Require: n Energiefunktionen $f_i(x)$ für ein Scratchpad der Größe M
Ensure: $h(x_i) = min\{f_1(x_1) + \ldots + f_n(x_n) \mid x_1 + \ldots + x_n \leq x_i\}$
1: **if** $n > 2$ **then**
2: $h(x)$ = Ergebnis des rekursiven Aufrufs mit $f_1(x), \ldots, f_{n-1}(x)$
3: $h'(x)$ = Ergebnis des Aufrufs von Algorithmus 1 mit $h(x), f_n(x)$
4: Return $h'(x)$
5: **else**
6: $h(x)$ = Ergebnis des Aufrufs von Algorithmus 1 mit $f_1(x), f_2(x)$
7: Return $h(x)$
8: **end if**

Die Rechenzeit von Algorithmus 2 beträgt $O(n \cdot m^2)$, da $n-1$ mal Algorithmus 1 aufgerufen werden muß.

Zum Abschluß soll noch bewiesen werden, daß die beiden Algorithmen die korrekte Ausgabe produzieren. Es wird begonnen mit dem Korrektheitsbeweis von Algorithmus 1. Der Beweis geschieht über Induktion.

Induktionsanfang: Für $i = 0$ ist $min = h(0) = f(0) + g(0)$.

Induktionsannahme: Für $i - 1$ sind $h(0), \ldots, h(i-1)$ korrekt berechnet und min repräsentiert den minimalen Energieverbrauch bei einer Scratchpad-Größe von höchstens $i - 1$.

Induktionsschritt: In den Zeilen 3-7 von Algorithmus 1 wird der minimale Energieverbrauch bei einem Scratchpad-Anteil von i Einheiten berechnet. Nach Induktionsannahme wurde min korrekt für $i-1$ berechnet. Da in der inneren Schleife min aktualisiert wird, falls eine bessere Lösung gefunden wird, repräsentiert min auch nach dem Durchlaufen der inneren Schleife den besten bisher gefundenen Wert. Nachher wird $h(i)$ auf diesen Wert gesetzt und damit auch die kombinierte Energiefunktion korrekt berechnet.

□

Der Korrektheitsbeweis für Algorithmus 2 wird ebenfalls über Induktion geführt.

Induktionsanfang: Für $n = 2$ wird Algorithmus 1 aufgerufen.

Induktionsannahme: Für $n - 1$ wird $h'(x)$ korrekt berechnet.

Induktionsschritt: Nach Induktionsannahme enthält $h(x)$ das korrekte Ergebnis für $f_1, \ldots, f_{n-1}$. Auf $h(x)$ und f_n wird Algorithmus 1 angewendet, der für zwei Energiefunktionen den minimalen Energiebedarf für jede mögliche Scratchpad-Größe berechnet.

□

3.3 Dynamische Allokation für mehrere Prozesse (DAMP)

Die im letzten Abschnitt vorgestellte Allokations-Strategie SAMP vergibt im Extremfall das gesamte Scratchpad an einen Prozess. Für alle anderen Prozesse kann dann keine Energie durch eine Nutzung des Scratchpads gespart werden. Erhalten im anderen Extremfall alle n Prozesse einen gleich großen Anteil dieses Speichers, dann erhält jeder Prozess bei einem Scratchpad der Größe M einen Anteil der Größe M/n. In der Regel benötigen die Prozesse mehr Speicher um ihren Energiebedarf weiter zu reduzieren. Die Idee der Allokations-Strategie DAMP ist es, allen Prozessen den vollen Zugriff auf das Scratchpad zu ermöglichen. Hierbei zeigt sich aber das Problem, daß die Prozesse ihre Daten beim Wechsel zwischen den Prozessen im Scratchpad gegenseitig überschreiben und damit eine korrekte Programmausführung nicht mehr möglich ist. Das Problem könnte dadurch gelöst werden, daß die Prozesse den Code und die Daten, die sie im Scratchpad erwarten, dorthin kopieren, wenn sie die Kontrolle über den Prozessor erlangen und zurück in den Hauptspeicher sichern, wenn sie diese wieder abgeben müssen. Wird preemptives Scheduling eingesetzt, wissen die Prozesse allerdings nicht, wann sie unterbrochen werden. Das einzige Programmstück, das diese Aufgabe übernehmen kann, ist daher der Dispatcher. Er kennt den Prozess, dem gerade die Kontrolle über den Prozessor entzogen wurde und den Prozess, der als nächstes zur Ausführung kommt. Um die Allokations-Strategie DAMP nutzen zu können, muß der Dispatcher um eine Kopierfunktion erweitert werden.

Die zusätzliche Aufgabe des Dispatchers wird in den Abbildungen 3.5 (a) und (b) verdeutlicht. In Abbildung 3.5 (a) ist im oberen Teil das Scratchpad dargestellt, in dem sich momentan der dynamische Teil des **mpegdec** Prozesses befindet. Der untere Teil zeigt den Hauptspeicher, in dem sich der nicht ins Scratchpad ausgelagerte Code und die nicht ausgelagerten Daten befinden, sowie die Kopien der dynamischen Anteile. Das Bild zeigt die

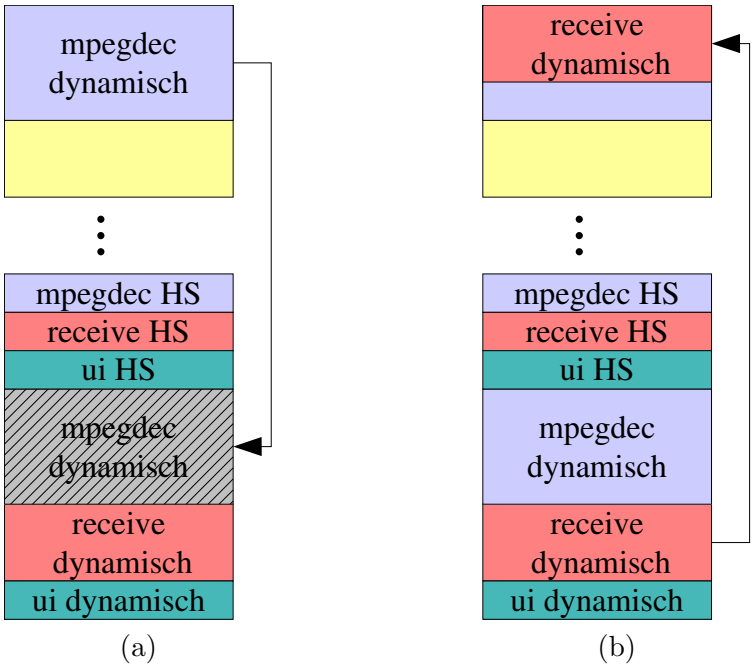

Abbildung 3.5: (a) Nach dem Unterbrechen von **mpegdec**, (b) Vor der Ausführung von **receive**

Situation, in der dem Prozess **mpegdec** die Kontrolle über den Prozessor entzogen wird. Aufgabe des Dispatchers ist es nun, den dynamischen Teil von **mpegdec** in den Hauptspeicher zu sichern, damit er von anderen Prozessen nicht überschrieben werden kann. Die Kopie des dynamischen Anteils von **mpegdec** im Hauptspeicher ist schraffiert dargestellt, da sie eine "veraltete" Version des dynamischen Anteils darstellt, denn **mpegdec** hat während seiner Ausführung auf den Daten im Scratchpad gearbeitet. Eigentlich müßte der ins Scratchpad ausgelagerte Code nicht gesichert werden, da er nicht verändert wird. Für den Code würden also jeweils nur die Kopierkosten vom Hauptspeicher in das Scratchpad anfallen. In dieser Arbeit wird dieser Fall allerdings nicht betrachtet, es werden Code und Daten zurück in den Hauptspeicher gesichert. Um durch die eben genannte Beobachtung weitere Energieeinsparungen erzielen zu können, ist es notwendig, den Optimierungsansatz von Steinke zu erweitern, der hier als Grundlage genutzt wird. Dies ist Teil der Weiterentwicklungen der hier vorgestellten Allokations-Strategien.

Nachdem der Dispatcher alle Daten gesichert hat, muß er, bevor der Prozess **receive** zur Ausführung gelangt, dessen dynamischen Anteil ins Scratchpad kopieren. Dieser Vorgang ist in Abbildung 3.5 (b) dargestellt.

Die Nutzbarkeit dieses Verfahrens wird dadurch eingeschränkt, daß bei

jedem Prozesswechsel zwei Speicherbereiche kopiert werden müssen, was wiederum Energie kostet. Die zwei wesentlichen Faktoren, die den Energieverbrauch bei dieser Allokations-Strategie beeinflussen, sind die durch die Kopierfunktion verursachten Kosten und die Länge der Zeitscheibe. Bei einer kurzen Zeitscheibe werden die Prozesse sehr oft unterbrochen und es entstehen demzufolge auch höhere Kopierkosten als dies bei einer längeren Zeitscheibe der Fall wäre. Um die entstehenden Kopierkosten gegen den Nutzen, der durch eine Auslagerung von Code und Daten ins Scratchpad entsteht, abwägen zu können, muß ein weiterer Funktionstyp definiert werden. Die Taktzyklenfunktion $t_i(x_j)$ gibt für Prozess i an, wieviele Taktzyklen er für seine Ausführung benötigt, wenn er x_j Anteile an Scratchpad-Speicher nutzt.

Um nun für diese Allokations-Strategie die optimale Zuweisung von dynamischem Scratchpad-Speicher an die Prozesse berechnen zu können, müssen neben den Energiefunktionen der Prozesse, die Taktzyklenfunktionen, die Kopierkosten und die Länge der Zeitscheibe bekannt sein.

Mit diesen Angaben läßt sich für jeden Prozess ermitteln, wieviel mal er unterbrochen wird und wie oft der dynamische Anteil kopiert werden muß. Bezeichnet c_{ts} die Länge der Zeitscheibe und nutzt Prozess i x_j Anteile des Scratchpads, dann müssen

$$b_i(x_j) = \left\lfloor \frac{t_i(x_j)}{c_{ts}} \right\rfloor + 1 \tag{3.4}$$

mal seine dynamischen Daten ins Scratchpad kopiert werden. Gleiches gilt für das Kopieren zurück in den Hauptspeicher. Der zweite Summand kommt dadurch zustande, daß bevor der Prozess das erste Mal ausgeführt wird und nach seiner letzten Zeitscheibe ebenfalls kopiert werden muß.

3.3.1 ILP

Die Allokations-Strategie DAMP wird nun in einer mathematischen Form als ILP-Modell vorgestellt.

n	Anzahl der Prozesse
M	Größe des Scratchpads
e_{ij}	Energieverbrauch von Prozess i bei j Scratchpadanteilen
x_{ij}	binäre Entscheidungsvariable, die den Wert 1 hat, wenn Prozess i j Scratchpadanteile nutzt
b_{ij}	Anzahl der Kopieraktionen, definiert wie in Gleichung 3.4
$KK_{SP,j}$	Kosten für das Kopieren von j Anteilen vom Hauptspeicher ins Scratchpad
$KK_{HS,j}$	Kosten für das Kopieren von j Anteilen vom Scratchpad in den Hauptspeicher

Hierbei ist zu beachten, daß die Kopierkosten vom Hauptspeicher in den Scratchpad-Speicher und umgekehrt durchaus unterschiedlich groß sein können. Geht man z.B. davon aus, daß die Schreib- und Lesekosten für den Scratchpad-Speicher gleich groß sind, aber das Lesen eines Datums aus dem Hauptspeicher höhere Kosten verursacht als das Schreiben, dann gilt $KK_{SP,j} > KK_{HS,j}$ für $j \neq 0$.

Die zu minimierende Zielfunktion lautet

$$\sum_{i=1}^{n} \sum_{j=0}^{M} x_{ij} \cdot [e_{ij} + b_{ij} \cdot (KK_{SP,j} + KK_{HS,j})] \rightarrow minimiere \tag{3.5}$$

Da jeder Prozess nur einen Anteil am Scratchpad erhält, gilt folgende Einschränkung für die x_{ij}

$$\forall i : \quad \sum_{j=0}^{M} x_{ij} = 1$$

In Gleichung 3.5 beschreibt der erste Summand der Klammer den Energieverbrauch des Prozess i bei Nutzung von j Scratchpadanteilen. Bei der hier vorgestellten Allokations-Strategie fallen allerdings noch Kopierkosten an. Diese ergeben sich aus der Anzahl der Kopieraktionen multipliziert mit den Kopierkosten, die für j Scratchpadanteile vom Hauptspeicher ins Scratchpad und in die andere Richtung anfallen.

3.3.2 Algorithmische Lösung

Aus der einfachen Struktur der Zielfunktion in Gleichung 3.5 und nur einer Nebenbedingung, läßt sich bereits erkennen, daß der minimale Energieverbrauch auf einfache Art und Weise berechnet werden kann. Die einzelnen Prozesse werden durch keine Nebenbedingung untereinander eingeschränkt, deshalb reicht es, die Minimierung für jeden Prozess einzeln durchzuführen. Algorithmus 3 berechnet für die Allokations-Strategie DAMP den minimalen Energieverbrauch.

Bevor man die Allokations-Strategie DAMP auf die Prozesse des Handheld-Beispiels anwenden kann, müssen noch die zugehörigen Taktzyklenfunktionen und die Kopierfunktion definiert werden. Die Taktzyklenfunktionen sind in Tabelle 3.5 aufgeführt. Die Kopierkosten seien durch folgende Funktion definiert:

$$KK_{SP}(x) := KK_{HS}(x) := \begin{cases} 0,1 & : \quad x = 0 \\ 0,33 \cdot x & : \quad x \neq 0 \end{cases}$$

Für einen Speicherblock der Länge x mit $x \neq 0$ sind die Kosten linear in der Größe des zu kopierenden Speicherbereichs. Soll kein Speicherblock kopiert

Algorithm 3 Minimalen Energieverbrauch für n Prozesse berechnen

Require: n Energiefunktionen $f_i(x)$ für ein Scratchpad der Größe M
Require: Kopierkostenfunktionen $KK_{SP}(x)$ und $KK_{HS}(x)$
Require: Kopieranzahlfunktionen $b_i(x)$
Ensure: $h(y) = min\{\sum_{j=1}^{n}[f_j(x_j) + c_j * (KK_{SP}(x_j) + KK_{HS}(x_j))] \mid x_1, \ldots, x_n \leq y\}$

```
1:  for i = 0 to M do
2:    E_min = 0
3:    for j = 1 to n do
4:      min = ∞
5:      for k = 0 to i do
6:        E = f_j(k) + b_j(k) · (KK_SP(k) + KK_HS(k))
7:        if E < min then
8:          min = E
9:        end if
10:     end for
11:     E_min = E_min + min
12:   end for
13:   h(k) = E_min
14: end for
15: Return h(x)
```

werden, dann fallen trotzdem geringe Fixkosten an, da auch in diesem Fall ein kleiner Teil der Kopierfunktion abgearbeitet werden muß.

Geht man von einer Zeitscheibenlänge c_{ts} von 50000 Zyklen aus und wendet Algorithmus 3 an, dann erhält man einen Energieverbrauch von 101,48 mJ, wobei **mpegdec** 4 kB, **receive** 3 kB und **ui** 2 kB Scratchpad-Speicher dynamisch nutzt.

Prozess	0 kB	1 kB	2 kB	3 kB	4 kB
mpegdec	360000	320000	320000	252000	228000
receive	120000	108000	96000	72000	72000
ui	60000	36000	32000	32000	32000

Tabelle 3.5: Taktzyklenfunktionen der Prozesse

3.4 Hybride Allokation für mehrere Prozesse (HAMP)

In den Abschnitten 3.2 und 3.3 wurden zwei Allokations-Strategien vorgestellt, denen zwei vollkommen unterschiedliche Ansätze zugrunde liegen. SAMP versucht den Scratchpad-Speicher so gut wie möglich unter den Pro-

zessen zu verteilen, DAMP ermöglicht es den Prozessen, das volle Scratchpad zu nutzen, allerdings müssen die Inhalte bei einem Prozesswechsel umkopiert werden. Die Allokations-Strategie SAMP sollte bei großen Scratchpad-Speichern bessere Ergebnisse liefern, da keine Kopierkosten anfallen und viel Speicher zur Verteilung zur Verfügung steht. DAMP sollte bessere Ergebnisse auf kleinen Scratchpad-Speichern liefern, da sich dort die Kopierkosten in Grenzen halten und der Nutzen durch die gemeinsame Verwendung des Scratchpads die Kopierkosten aufwiegt.

Eine weitere Möglichkeit Energie einzusparen ist es, die Vorteile der beiden Verfahren zu kombinieren. Es sollte versucht werden, den Prozessen so viel statischen Scratchpad-Speicher wie möglich zuzuweisen. Weiter gesenkt werden kann der Energieverbrauch nur, wenn ihnen noch mehr Scratchpad-Anteile zugewiesen werden. Um aber alle Prozesse gleichermaßen profitieren zu lassen und nicht einen der Prozesse zu bevorzugen, wird ein weiterer Teil des Scratchpads dynamisch genutzt. Die Aufgabe der Allokations-Strategie HAMP ist es, die beste Aufteilung des Scratchpads in jeweils einen statischen Anteil für jeden Prozess und einen gemeinsam genutzten dynamischen Anteil zu berechnen.

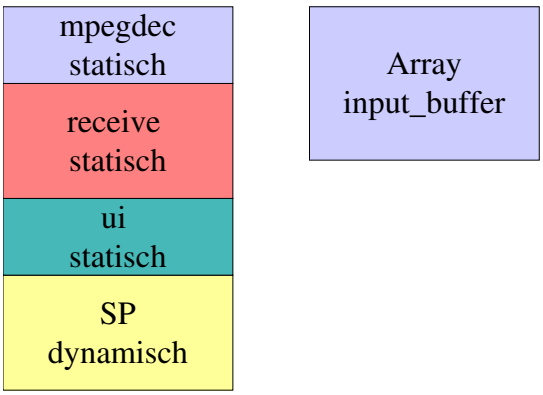

Abbildung 3.6: Probleme bei der Nutzung der Energiefunktionen für HAMP

Die bisher verwendeten Energiefunktionen können allerdings nicht als Eingabe für die Allokations-Strategie HAMP genutzt werden. Abbildung 3.6 verdeutlicht das Problem. Im linken Teil der Abbildung ist der Scratchpad-Speicher dargestellt. Jeder Prozess bekommt durch HAMP einen statischen Anteil zugewiesen. Ein weiterer Teil des Scratchpads wird dynamisch genutzt. Die Optimierungsmethode nach Steinke [SWLM02] optimiert die einzelnen Prozesse für einen zusammenhängenden Scratchpad-Speicher. In Abbildung 3.6 ist die Situation dargestellt, daß die Optimierung für den Prozess **mpegdec** das Array `input_buffer` für die Auslagerung in das Scratchpad ausgewählt hat. Ein Algorithmus für die Allokations-Strategie HAMP würde nun anhand der Energiefunktionen eine Aufteilung des Scratchpads in einen dynamischen und mehrere statische Anteile bestimmten. Dabei hat

der Algorithmus keine Kenntnis über die Anzahl und Größe der Objekte, die durch die Optimierung ins Scratchpad ausgelagert wurden. Die Abbildung zeigt nun das Problem, daß das Array `input_buffer` zwar in den dem Prozess zur Verfügung stehenden Scratchpad-Speicher (statischen und dynamischen Teil) passen würde, aber nur, wenn diese Teile direkt hintereinander angeordnet wären. Die Array-Adressierung verlangt, daß die Elemente eines Arrays aufeinanderfolgend im Speicher angeordnet sind. Durch eine Umordnung der statischen Teile könnte das Problem umgangen werden, doch ist dies nicht mehr möglich, sobald für einen weiteren Prozess eine Situation wie für **mpegdec** entsteht.

3.4.1 HAOP

Die soeben aufgezeigte Problematik motiviert, warum an der zugrundeliegenden Optimierungsmethode Veränderungen vorgenommen werden müssen. Das Problem, ein Programm für zwei Scratchpad-Speicher, einen statischen Teil und einen dynamischen Teil zu optimieren, wird von jetzt an als HAOP (Hybride Allokation für einen Prozess) bezeichnet.

HAOP erhält als Eingabe eine Applikation und die Größe eines statischen und eines dynamischen Scratchpad-Anteils. Um die Optimierung korrekt durchführen zu können, müssen die Kopierkosten für den dynamischen Teil berücksichtigt werden. Da die Anzahl der Prozesswechsel einen Einfluß auf die Kopierkosten hat, gehört die Anzahl der Unterbrechungen der Applikation ebenfalls zur Eingabe.

Ausgabe ist der minimale Energieverbrauch und eine mögliche Verteilung der Programmobjekte auf den statischen und dynamischen Scratchpad-Speicher, sowie auf den Hauptspeicher. Hiermit läßt sich eine erweiterte Energiefunktion bestimmen, die den Energieverbrauch für jede zulässige Kombination aus einem statischen und einem dynamischen Anteil definiert. Die erweiterten Energiefunktionen werden Eingabe für die Allokations-Strategie HAMP. Tabelle 3.6 zeigt die erweiterte Energiefunktion für den Prozess **mpegdec**. Die Spalten zeigen jeweils den Energieverbrauch bei einem statischen Anteil und variierendem dynamischen Anteil.

dyn ↓ *stat* →	0 kB	1 kB	2 kB	3 kB	4 kB
0 kB	92	89	75	73	71
1 kB	86	71	68	67	
2 kB	67	65	64		
3 kB	65	61			
4 kB	58				

Tabelle 3.6: Erweiterte Energiefunktion für **mpegdec**

3.4.1.1 Theoretische Diskussion (NP-Vollständigkeit)

HAOP soll jetzt formal definiert werden und es wird gezeigt, daß es sich dabei um ein NP-vollständiges Problem handelt.

HAOP erhält als Eingabe n Objekte, denen Größenwerte $g_1, \ldots, g_n$ zugeordnet sind. Diese Objekte sollen auf ein statisches und ein dynamisches Scratchpad verteilt werden. Die Größe des statischen Scratchpads ist S_{stat}, die des dynamischen S_{dyn}. Wird Objekt i im statischen Teil abgelegt, dann ist damit ein Nutzen von $v_{i,stat}$ verbunden. Im dynamischen Teil ist der Nutzen wegen der anfallenden Kopierkosten geringer und wird mit $v_{i,dyn}$ bezeichnet. Die Entscheidungsvariante von HAOP ist nun das Problem, für einen gegebenen Nutzwert A zu entscheiden, ob es eine Verteilung der Objekte auf das statische und dynamische Scratchpad gibt, die die Größen dieser Scratchpads respektiert und mindestens Nutzen A erreicht.

Hierbei ist anzumerken, daß diese Definition des HAOP-Problems nur eine Näherung darstellt. Sie wird verwendet, da mit Hilfe eines ILP-Lösers Lösungen berechnet werden sollen. Eine exakte Formulierung findet sich in Anhang A.

Zum Beweis der NP-Vollständigkeit wird HAOP auf das PARTITION-Problem polynomiell reduziert. PARTITION erhält als Eingabe $a_1, \ldots, a_n$ und es soll entschieden werden, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ gibt, so daß $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.

Ein Beweis für die NP-Vollständigkeit von PARTITION findet sich in [Weg99].

Satz: HAOP ist NP-vollständig.

Beweis: HAOP $\in$ NP, da eine Verteilung der Objekte auf die Speicher geraten und in polynomieller Zeit verifiziert werden kann, ob der Nutzen A erreicht wird.

Sei $(a_1, \ldots, a_n)$ eine Eingabe für PARTITION. Hieraus wird in polynomieller Zeit folgende Eingabe für HAOP konstruiert. $g_1 = a_1, \ldots, g_n = a_n$, $v_{1,stat} = v_{1,dyn} = \ldots = v_{n,stat} = v_{n,dyn} = 1$, $A = n$ und $S_{stat} = S_{dyn} = \lfloor (a_1 + \cdots + a_n)/2 \rfloor$.

Hat PARTITION eine Lösung, dann können die Objekte wegen der Definition von S_{stat}, S_{dyn} und der g_i in das statische und das dynamische Scratchpad verteilt werden. Da jedes Objekt einen Nutzenwert von eins hat und entweder im statischen oder dynamischen Teil abgelegt wird, kann auch der Nutzenwert A erreicht werden.

Hat HAOP eine Lösung, dann muß der Gesamtnutzen der zur Lösungsmenge gehörenden Objekte mindestens n betragen, was wiederum bedeutet, daß alle Objekte in einem der beiden Scratchpad-Speicher plaziert wurden. Die Wahl der Größen dieser beiden Speicher sichert, daß auch PARTITION eine Lösung hat. □

3.4.1.2 Pseudopolynomieller Algorithmus für HAOP

Es wird nun ein Algorithmus vorgestellt, mit dem für eine Eingabe für HAOP der Wert einer optimalen Lösung bestimmt werden kann. Das Verfahren basiert auf einem pseudopolynomiellen Algorithmus für das Rucksackproblem [Weg99] und benutzt wie dieser das Prinzip der dynamischen Programmierung.

Eingabe für HAOP sind die Größen der Objekte $g_1, \ldots, g_n$, die Nutzenwerte im statischen Scratchpad $v_{1,stat}, \ldots, v_{n,stat}$ und im dynamischen Scratchpad $v_{1,dyn}, \ldots, v_{n,dyn}$, sowie die Größen der beiden Speicher S_{stat} und S_{dyn}.

$HAOP(k, s_1, s_2)$, $1 \leq k \leq n$, $0 \leq s_1 \leq S_{stat}$, $0 \leq s_2 \leq S_{dyn}$ bezeichnet das Teilproblem, in dem nur die ersten k Objekte betrachtet werden und die Größen der beiden Speicher auf s_1 und s_2 eingeschränkt sind. Der Nutzen einer optimalen Lösung für $HAOP(k, s_1, s_2)$ wird durch $N(k, s_1, s_2)$ bezeichnet.

Während seiner Abarbeitung füllt der Algorithmus n Matrizen der Größe $S_{stat} \times S_{dyn}$ und greift dabei auf vorher berechnete Werte und folgende Randbedingungen zurück: $N(k, s_1, s_2) := -\infty$ für $s_1 < 0 \ \vee \ s_2 < 0$ und $N(0, s_1, s_2) := N(k, 0, 0) := 0$ für $s_1 \geq 0 \ \wedge \ s_2 \geq 0$.

Betrachtet man das Teilproblem $HAOP(k, s_1, s_2)$, dann gibt es für das Objekt k drei Möglichkeiten. Es kann in den statischen oder den dynamischen Scratchpad-Speicher gepackt werden, oder aber es verbleibt im Hauptspeicher. In den ersten beiden Fällen sinkt der zur Verfügung stehende Speicher im jeweiligen Scratchpad um g_k und der Wert des bis dahin erzielten Gesamtnutzens erhöht sich um $v_{k,stat}$ bzw. $v_{k,dyn}$. Im letzten Fall bleibt der Gesamtnutzen von $N(k-1, s_1, s_2)$ unverändert. Es folgt die Bellmansche Optimalitätsgleichung

$$\begin{aligned} N(k, s_1, s_2) \quad = \quad & max\{N(k-1, s_1, s_2), N(k-1, s_1 - g_k, s_2) + v_{k,stat}, \\ & N(k-1, s_1, s_2 - g_k) + v_{k,dyn}\} \end{aligned}$$

Jeder der Einträge in den Matrizen kann in Zeit $O(1)$ berechnet werden. Damit folgt eine Rechenzeit von $O(n \cdot S_{stat} \cdot S_{dyn})$.

3.4.1.3 ILP-Modell für HAOP

Da dem Autor kein Approximationsalgorithmus für HAOP bekannt ist und mit Hilfe von ILP-Gleichung Probleme dieser Art heutzutage gut gelöst werden können, wird hier ein ILP-Modell für HAOP angegeben.

Zu Beginn werden einige Variablen des Modells definiert.

n	Anzahl der Objekte
S_{stat}	Größe des statischen Scratchpads
S_{dyn}	Größe des dynamischen Scratchpads

g_i	Größe von Objekt i
$v_{i,stat}$	Nutzenwert von Objekt i im statischen Scratchpad
$v_{i,dyn}$	Nutzenwert von Objekt i im dynamischen Scratchpad
$x_{i,stat}$	binäre Entscheidungsvariable, die den Wert 1 hat, wenn Objekt i im statischen Scratchpad abgelegt wird
$x_{i,dyn}$	binäre Entscheidungsvariable, die den Wert 1 hat, wenn Objekt i im dynamischen Scratchpad abgelegt wird

Die Zielfunktion für HAOP lautet

$$\sum_{i=1}^{n}(x_{i,stat} \cdot v_{i,stat} + x_{i,dyn} \cdot v_{i,dyn}) \rightarrow maximiere$$

Jedes der Objekte darf in höchstens einem der Scratchpads abgelegt werden

$$\forall i: \quad x_{i,stat} + x_{i,dyn} \leq 1$$

Weiterhin darf für keinen der Scratchpad-Speicher die Gesamtgröße der darin abgelegten Objekte die Größe des Speichers überschreiten.

$$\sum_{i=1}^{n} x_{i,stat} \cdot g_i \leq S_{stat}$$

und

$$\sum_{i=1}^{n} x_{i,dyn} \cdot g_i \leq S_{dyn}$$

3.4.2 ILP-Modell für HAMP

In Abschnitt 3.4 wurde motiviert, warum für die Allokations-Strategie HAMP die Energiefunktionen nicht genutzt werden können. Die darauffolgenden Abschnitte stellten die erweiterten Energiefunktionen und eine Möglichkeit zu deren Berechnung vor. Diese erweiterten Energiefunktionen bilden die Eingabe für die Allokations-Strategie HAMP, die jetzt durch ILP-Gleichungen beschrieben wird. Zuerst werden folgende Variablen definiert.

n	Anzahl der Prozesse
M	Größe des Scratchpads
e_{ijk}	Energieverbrauch von Prozess i bei j statischen und k dynamischen Scratchpadanteilen
x_{ijk}	binäre Entscheidungsvariable, die den Wert 1 hat, wenn Prozess i j statische und k dynamische Scratchpadanteile nutzt

Die Zielfunktion lautet

$$\sum_{i=1}^{n}\sum_{j=0}^{M}\sum_{k=0}^{M-j} x_{ijk} \cdot e_{ijk} \rightarrow minimiere$$

Für jeden Prozess darf nur genau ein statischer und ein dynamischer Anteil ausgewählt werden

$$\forall i: \quad \sum_{j=0}^{M}\sum_{k=0}^{M-j} x_{ijk} = 1$$

Weiterhin dürfen für keinen Prozess der dynamische Anteil und die Summe der statischen Anteile aller Prozesse die Gesamtgröße des Scratchpads M überschreiten.

$$\forall l: \quad \sum_{i=1}^{n}\sum_{j=0}^{M}\sum_{k=0}^{M-j} x_{ijk} \cdot j + \sum_{j=0}^{M}\sum_{k=0}^{M-j} x_{ljk} \cdot k \leq M \tag{3.6}$$

Der erste Summand in Gleichung 3.6 summiert über die statischen Anteile aller Prozesse, wobei der Faktor j die Größe des jeweiligen statischen Anteils bezeichnet. Der zweite Summand summiert über die dynamischen Anteile eines Prozesses, der Faktor k bezeichnet hierbei die Größe des jeweiligen dynamischen Anteils. Für ein festes l sichert Gleichung 3.6, daß die statischen Anteile aller Prozesse und der dynamische Anteil von Prozess l nicht größer als das zur Verfügung stehende Scratchpad sind.

3.4.3 Algorithmische Lösung

Ähnlich wie im Fall der Allokations-Strategie SAMP wird hier als erstes die Aufteilung des Scratchpads für zwei Prozesse betrachtet. Es wird ein Algorithmus entwickelt, der für zwei erweiterte Energiefunktionen $f_e(x, y)$ und $g_e(x, y)$ eine kombinierte Energiefunktion $h_e(x, y)$ berechnet. Für (x, y) liefert $h_e(x, y)$ den minimalen Energieverbrauch, wobei die beiden statischen Anteile der Prozesse zusammen höchsten den Wert x haben und die dynamischen Anteile jeweils nicht größer als y sind.

Die Idee hinter Algorithmus 1 war es, für jede mögliche Teilgröße des Scratchpads eine optimale Aufteilung unter den Prozessen zu finden. Ähnlich geht Algorithmus 4 vor.

Die Schleife in Zeile 1 iteriert über jede mögliche Größe für den dynamischen Teil des Scratchpads. Für jede dieser Größen wird in den folgenden Zeilen jede mögliche Größe des statischen Teils auf die beste Aufteilung unter den Prozessen hin untersucht. Der Energieverbrauch der besten gefunden Aufteilung wird in die kombinierte erweiterte Energiefunktion übernommen.

Algorithm 4 Minimierung zweier erweiterter Energiefunktionen

Require: Erweiterte Energiefunktionen $f_e(x, y)$ und $g_e(x, y)$ für ein Scratchpad der Größe M

Ensure: $h_e(x_i, y_j) = min\{f_e(x_k, y_l) + g_e(x_p, y_r) | x_k + x_p \leq x_i \wedge y_l \leq y_j \wedge y_r \leq y_j\}$

```
1:  for d = 0 to M do
2:    min = ∞
3:    for s = 0 to M − d do
4:      for k = 0 to s do
5:        if min > f_e(k, d) + g_e(s − k, d) then
6:          min = f_e(k, d) + g_e(s − k, d)
7:        end if
8:      end for
9:      h_e(s, d) = min
10:   end for
11: end for
12: Return h_e(x, y)
```

Die Rechenzeit des Verfahrens kann durch $O(M^3)$ abgeschätzt werden, da jede der Schleifen maximal M Iterationen durchführt und in der innersten Schleife alle Berechnung in Zeit $O(1)$ durchgeführt werden können.

Algorithmus 5 führt die Minimierung, analog zu Algorithmus 2, für n erweiterte Energiefunktionen durch. Da es $n-1$ Aufrufe von Algorithmus 4 gibt, folgt daraus eine Rechenzeit von $O(n \cdot M^3)$ für Algorithmus 5.

Es soll nun die Korrektheit von Algorithmus 4 bewiesen werden. Das HAOP-Verfahren liefert erweiterte Energiefunktionen $f_e(x, y)$ mit der folgenden Eigenschaft: $f_e(x_1, y_1) \leq f_e(x_2, y_2)$ falls $x_1 \geq x_2 \ \wedge \ y_1 \geq y_2$. Diese Eigenschaft besagt, daß falls einem Prozess mehr statischer oder dynamischer Speicher zur Verfügung steht, sein Energieverbrauch gleich bleibt oder sinkt. Der Energieverbrauch kann nicht steigen, da das HAOP-Verfahren sonst eine Lösung mit weniger statischem und/oder dynamischem Anteil gewählt hätte. Es ist deshalb ausreichend, in Zeile 2 in Algorithmus 4 die Variable min, die den Wert der besten bisher gefundenen Aufteilung repräsentiert, mit unendlich zu initialisieren, da eine mindestens gleichwertige Lösung wie in der letzten Iteration auf jeden Fall gefunden wird.

Die Zeilen 2-10 von Algorithmus 4 sind nahezu äquivalent mit Algorithmus 1, mit der Ausnahme, daß hier die Iteration über alle möglichen statischen Anteile durch die Größe des momentan untersuchten dynamischen Anteils eingeschränkt ist. Die Korrektheit von Algorithmus 1 wurde bewiesen, also wird, zusammen mit der speziellen Eigenschaft der erweiterten Energiefunktionen, die kombinierte Energiefunktion $h_e(x, y)$ für $y = d$ und $x = 0, \ldots, M - d$ korrekt berechnet.

Die äußere Schleife iteriert über jeden zulässigen dynamischen Anteil. In

Algorithm 5 Minimalen Energieverbrauch für n Prozesse berechnen

Require: Energiefunktionen $f_{e,1}(x,y), \ldots, f_{e,n}(x,y)$

Ensure: $h_e(x_i, y_j) = min\{f_{e,1}(x_1,y_1) + \ldots + f_{e,n}(x_n,y_n) \mid x_1 + \ldots + x_n \leq x_i \wedge \forall k : y_k \leq y_j\}$

1: **if** $n > 2$ **then**
2: $h_e(x,y)$ = Ergebnis des rekursiven Aufrufs mit $f_{e,1}(x,y), \ldots, f_{e,n-1}(x,y)$
3: $h_e^{'}(x,y)$ = Ergebnis des Aufrufs von Algorithmus 4 mit $h_e(x,y), f_{e,n}(x,y)$
4: Return $h_e^{'}(x,y)$
5: **else**
6: $h_e(x,y)$ = Ergebnis des Aufrufs von Algorithmus 4 mit $f_{e,1}(x,y), f_{e,2}(x,y)$
7: Return $h_e(x,y)$
8: **end if**

den inneren Schleifen wird jeweils eine optimale Lösung für einen dynamischen Anteil und einen aufzuteilenden statischen Anteil berechnet. Hiermit folgt, daß die kombinierte Energiefunktion $h_e(x,y)$ korrekt berechnet wird. □

Algorithmus 5 berechnet den minimalen Energieverbrauch auf Basis erweiterter Energiefunktionen auf die gleiche Weise wie Algorithmus 2. Da der Korrektheitsbeweis analog durchgeführt werden kann, wird auf ihn verzichtet.

Bevor Algorithmus 5 auf die Prozesse des Handhelds angewendet werden kann, müssen noch die erweiterten Energiefunktionen der Prozesse `receive` und `ui` angegeben werden. Diese finden sich in den Tabellen 3.7 und 3.8. Die Allokations-Strategie HAMP berechnet als minimalen Energieverbrauch 99 mJ für die Prozesse des Handhelds, wobei `mpegdec` 3 kB statischen Anteil und `receive` und `ui` 0 kB statischen Anteil erhalten. Alle Prozesse nutzen 1 kB des Scratchpad dynamisch.

dyn ↓ *stat* →	0 kB	1 kB	2 kB	3 kB	4 kB
0 kB	31	29	27	25	22
1 kB	28	26	22	22	
2 kB	25	21	20		
3 kB	20	19			
4 kB	19				

Tabelle 3.7: Erweiterte Energiefunktion für **receive**

dyn ↓ *stat* →	0 kB	1 kB	2 kB	3 kB	4 kB
0 kB	16	10	10	10	10
1 kB	11	10	10	10	
2 kB	10	10	9		
3 kB	9	9			
4 kB	9				

Tabelle 3.8: Erweiterte Energiefunktion für **ui**

3.4.4 Bemerkungen zu HAMP

Abschließend soll noch auf eine weitere Problematik beim Einsatz der Allokations-Strategie HAMP hingewiesen werden.

In Abschnitt 3.4.1 wurde erwähnt, daß für das HAOP-Verfahren die Anzahl der Unterbrechungen einer Applikation bekannt sein muß. Bekannt ist die Länge einer Zeitscheibe des Systems. Die Taktzyklenanzahl, die zusammen mit der Zeitscheibenlänge die Anzahl der Unterbrechungen ergibt, wird jedoch erst durch das HAOP-Verfahren festgelegt. Hier steht man vor dem Problem, daß eine Information für einen Algorithmus benötigt wird, die dieser wiederum erst produziert.

Eine Näherung der Taktzyklenanzahl kann auf folgende Weise gefunden werden: Die Messung der Energiefunktion beginnt mit $f_e(0,0)$, also keinem statischen und dynamischen Anteil. Hierfür muß die Anzahl der Unterbrechungen nicht bekannt sein, da kein Objekt im dynamischen Teil abgelegt werden kann. Die Taktzyklenanzahl der resultierenden Binärdatei und die Anzahl der Unterbrechungen wird bestimmt. Für die nächste Messung mit einem kleinen Anteil am dynamischen Speicher wird diese Information dem HAOP-Verfahren als Näherung übergeben. Für die resultierende Binärdatei wird wieder die Taktzyklenanzahl bestimmt, die wiederum als Näherung für die nächste Messung dient usw.

3.5 Abschlußbemerkungen

Abschließend sollen noch eine Abschätzung der Gesamtrechenzeit durchgeführt werden, die sich nicht auf die einzelnen Algorithmen konzentriert, sondern das gesamte Verfahren, von der Messung der Kostenfunktionen bis zur Optimierung durch eine Allokations-Strategie, betrachtet.

Das Verfahren für die Optimierung einer Applikation bezüglich eines Scratchpad-Speichers formuliert ein Rucksackproblem für die Objekte des Programms. Für das Rucksackproblem sind gute polynomielle Approximationsalgorithmen bekannt [KPP04]. Führt man M Optimierungen für ein Scratchpad der Größe M durch, dann wird eine Energiefunktion erzeugt, die an M Punkten definiert ist. Die Allokations-Strategien SAMP und DAMP

haben Rechenzeiten von $O(n \cdot M^2)$ bzw. $O(n \cdot M)$. Dies ist polynomiell in der Länge der Eingabe, die sich aus n Energiefunktionen zu je M Punkten zusammensetzt.

Die Eingabe für das ganze Verfahren setzt sich zusammen aus den Applikationen, deren Anzahl n und der Größe des Scratchpad-Speichers M. Die Länge der Eingabe beträgt $O(n \cdot App_{max} + log(n) + log(M))$, wobei hier App_{max} die Länge der größten Applikation bezeichnet. Wird für eine Applikation die Energiefunktion bestimmt, dann müssen M Messungen durchgeführt werden und die Rechenzeit ist nicht mehr polynomiell in der Eingabelänge beschränkt, da $M = 2^{log(M)}$.

Sieht man das ganze Verfahren, von der Messung der Kostenfunktionen bis zur Optimierung durch eine Allokations-Strategie, als einen Algorithmus an, dann handelt es sich hierbei um einen Algorithmus mit pseudopolynomieller Laufzeit. Die Betrachtung gilt auch für die Allokations-Strategie HAMP, wenn für das HAOP-Verfahren ein polynomieller Approximationsalgorithmus verwendet wird.

Kapitel 4

Arbeitsablauf und Implementierung

Dieses Kapitel beginnt mit einem Überblick darüber, wie man für eine Menge von Applikationen mit Hilfe einer Allokations-Strategie eine energieoptimierte Multiprozess-Binärdatei erhält. Der Compiler encc und der Profiler enprofiler nehmen eine wichtige Stellung im Meßprozess und bei der Erstellung von Binärdateien ein. Sie werden in Abschnitt 4.2 vorgestellt. Das Energiemodell für die Kopierfunktion wird in Abschnitt 4.3 eingeführt. Die Abschnitte 4.4, 4.5 und 4.6 zeigen, wie das Messen der Energiefunktionen, das Ausführen der Allokations-Strategien und die Erzeugung der Multiprozess-Binärdatei im Detail abläuft. In Abschnitt 4.7 werden die Änderungen am enprofiler besprochen, die für das Messen des Energieverbrauchs einer Multiprozess-Binärdatei notwendig sind. Es folgt in Abschnitt 4.8 die Vorstellung von Dispatcher und Scheduler. Abgeschlossen wird dieses Kapitel mit einer Besprechung der notwendigen Erweiterungen des ARMulators.

4.1 Überblick über den Arbeitsablauf

Es soll nun beschrieben werden, wie man zu einer Menge von Applikationen eine bezüglich eines Allokationsverfahrens energieminimale Multiprozess-Binärdatei erhält. Abbildung 4.1 stellt diesen Ablauf graphisch dar, wobei die eckigen Boxen die verarbeiteten Daten darstellen und die abgerundeten Boxen ausgeführten Programmen entsprechen.

1. Gestartet wird mit einer Menge von N Applikationen, die sich einen Scratchpad-Speicher der Größe M teilen sollen.

2. Für jede der Applikationen muß der Energieverbrauch gemessen werden. Dies geschieht abhängig von der gewählten Allokations-Strategie.

SAMP & DAMP: Für jeden Scratchpad-Anteil $i \in \{0, \ldots, M\}$ Bytes wird eine Messung durchgeführt.

HAMP: Für jede Aufteilung des Scratchpads (i, j), $i, j \in \{0, \ldots, M\}$, $i+j \leq M$ in einen statischen Anteil i und einen dynamischen Anteil j Bytes wird eine Messung durchgeführt. Dies geschieht unter Angabe der Länge der Zeitscheibe, da bei diesem Messverfahren die Anzahl der Unterbrechungen der Applikationen bekannt sein müssen, um eine korrekte Optimierung auch für den dynamischen Anteil durchführen zu können.

Typischerweise werden diese Messungen mit einer Schrittweite d durchgeführt, da zum einen eine sehr feine Granularität der Meßdaten nicht erforderlich ist und dies zum anderen einen erheblichen zeitlichen Mehraufwand für die Messung bedeuten würde.

3. Als Ergebnis von Schritt 2 erhält man für jede Applikation eine Energiefunktion. Zusätzlich wurden die Ausführungszeiten der Applikationen für jeden möglichen Scratchpad-Anteil ermittelt. Mit diesen Daten und der Länge der Zeitscheibe wird bei der Allokations-Strategie DAMP die Anzahl der Unterbrechungen bestimmt.

4. Die eben erzeugten Daten werden nun genutzt, um mittels einer Allokations-Strategie eine optimale Verteilung des Scratchpad-Speichers unter den Applikationen zu berechnen. Im Falle der DAMP-Strategie muß hierbei die Länge der Zeitscheibe spezifiziert werden.

5. Jeder Prozess erhält einen statischen und einen dynamischen Anteil am Scratchpad-Speicher. Für die Strategie SAMP hat der dynamische und für die Strategie DAMP der statische Anteil keine Bedeutung.

6. Nachdem die Verteilung des Scratchpad unter den Prozessen bekannt ist, benutzt ein Programm zur Erstellung einer Multiprozess-Binärdatei den encc-Compiler, um aus jeder Applikation drei Assembler-Dateien zu generieren. Diese Assembler-Dateien enthalten den Code und die Daten, die im Hauptspeicher bzw. statischen und dynamischen Scratchpad-Speicher platziert werden. Nachdem ein Template für den Dispatcher mit entsprechenden Daten gefüllt wurde, werden alle Assembler-Dateien sowie der Dispatcher übersetzt und zu einer Binärdatei zusammengelinkt.

7. Resultat ist eine Multiprozess-Binärdatei. Der Energieverbrauch dieser Datei läßt sich mit dem enprofiler ermitteln und das Ergebnis kann mit denen anderer Allokations-Strategien verglichen werden.

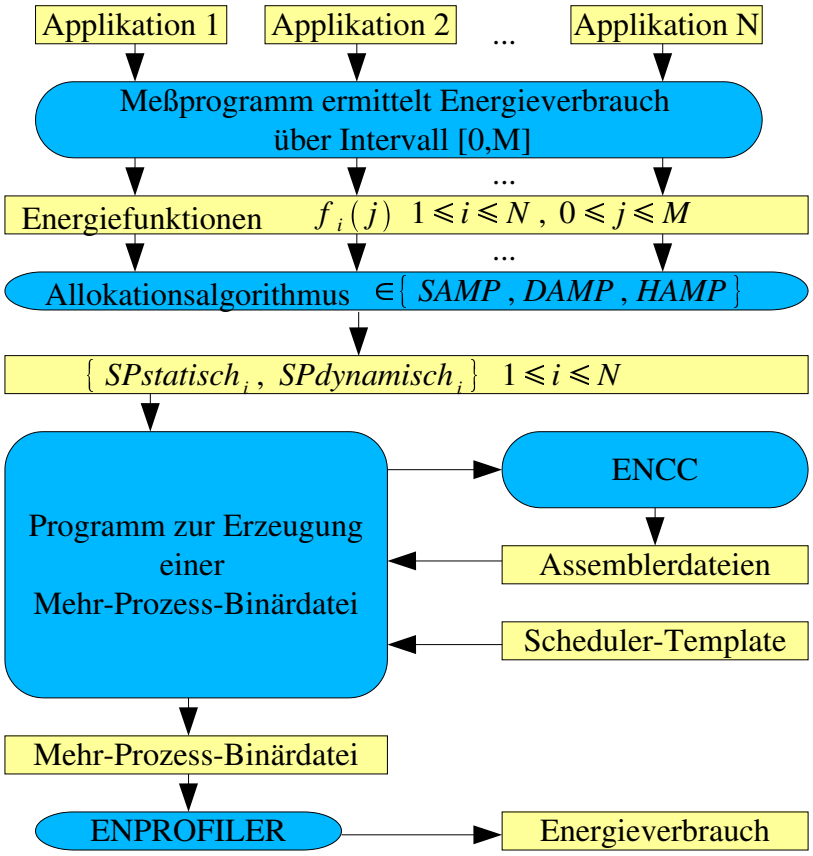

Abbildung 4.1: Erstellen einer energieminimalen Multiprozess-Binärdatei

4.2 encc & enprofiler

Der Energy Aware C Compiler encc wurde am Lehrstuhl 12 des Fachbereichs Informatik der Universität Dortmund entwickelt. Der Compiler besteht aus dem LANCE2 Frontend, einem Backend für den 16-Bit Thumb-Befehlssatz des ARM7TDMI-Prozessors, einem Backend für den 32-Bit Prozessor LEON und dem Profiler enprofiler.

Das LANCE2 Frontend transformiert eine C Quelldatei in eine Zwischenrepräsentation und kann einige High-Level Optimierungen wie Constant Propagation, Constant Folding, Copy Propagation, Jump Optimization, Common Subexpression Elimination und Dead Code Elimination anwenden. In den Backends wird der Code für die Zielplattformen generiert, die Registervergabe durchgeführt und weitere Optimierungen wie z.B. Instruction Scheduling, Register Pipelining und das Abbilden von Programmobjekten auf verschiedene Speichertypen durchgeführt.

enprofiler dient dazu, statistische Daten über Programme zu sammeln, die mit encc übersetzt wurden. Dies geschieht, indem ein Programm in einem

Simulator ausgeführt wird und enprofiler den erzeugten Execution-Trace analysiert. Folgende Informationen werden nach der Analyse bereitgestellt:

- Der Gesamtenergieverbrauch des Programms
- Anzahl der ausgeführten Instruktionen
- Anzahl der Taktzyklen
- Die in den verschiedenen Speichern verbrauchte Energie
- Die Energie, die beim Zugriff auf Variablen, Funktionen und Basis-Blöcke verbraucht wird
- Die Anzahl der Sprünge zwischen Basis-Blöcken
- Informationen darüber, aus welchen Basis-Blöcken auf welche Variablen zugegriffen wird

encc und enprofiler arbeiten auf folgende Art und Weise zusammen, um ein Programm für einen Scratchpad-Speicher bezüglich seines Energieverbrauchs zu optimieren:

1. Die Quelldateien werden so übersetzt, daß alle Programmobjekte in den Hauptspeicher abgebildet werden.
2. Die generierten Assemblerdateien werden übersetzt und gelinkt.
3. Die resultierende Binärdatei wird in einem Simulator ausgeführt und enprofiler analysiert den Execution-Trace.
4. Auf Basis der von enprofiler bereitgestellten statistischen Informationen über die Programmobjekte wird ein Rucksackproblem formuliert. Die Lösung dieses Optimierungsproblems liefert eine optimale Verteilung der Programmobjekte in die zur Verfügung stehenden Speicher.

Der soeben beschriebene Optimierungsvorgang nutzt das dynamische Profiling. Für eine Eingabe des Programms wird der Kontrollfluß und die Häufigkeit der Datenzugriffe bestimmt und auf Grundlage dieser Informationen die Optimierung durchgeführt. Der ebenfalls mögliche statische Ansatz hingegen benötigt keine Simulation des Programms. Hier wird für jede Schleife angenommen, daß sie zehn Mal durchlaufen wird und für jeden IF-Befehl wird eine Ausführungswahrscheinlichkeit von 50% für die beiden Zweige angenommen. Unter diesen Annahmen erfolgt die Verteilung der Programmobjekte in die Speicher.

Die in den Abschnitten 4.4 und 4.6 beschriebenen Programme zum Messen des Energieverbrauchs und zum Erstellen von Multiprozess-Binärdateien setzen auf einer erweiterten Version des encc auf. Urs Helmig hat in seiner Diplomarbeit [Hel04] das Aufteilen von Programmen auf partitionierte

Speicher betrachtet. Die Allokations-Strategie HAMP teilt das Scratchpad in einen statischen und einen dynamischen Teil, was als partitionierter Speicher betrachtet werden kann. Der von Urs Helmig weiterentwickelte encc wird für diese Arbeit auf zwei Scratchpad-Speicher eingeschränkt. Weiterhin wurde der Compiler so verändert, daß es möglich ist, nur Teile der Scratchpad-Speicher zu nutzen und Kopierkosten für Objekte, die in den dynamischen Scratchpad-Speicher abgebildet werden, zu betrachten.

4.3 Ein Energiemodell für die Kopierfunktion

Der Energieverbrauch der erstellten Multiprozess-Binärdateien soll mit Hilfe des enprofiler bestimmt werden. Dieses Tool ist jedoch nur in der Lage, den Energieverbrauch von Programmen, die den Thumb-Befehlssatz verwenden, präzise zu berechnen. Da der Dispatcher Befehle verwenden muß, die im Thumb-Modus nicht verfügbar sind, läuft er, ebenso wie die Kopierfunktion, im ARM-Modus. Es ist theoretisch möglich, beim Aufruf der Kopierfunktion in den Thumb-Befehlssatz zu wechseln, doch entstehen dadurch und durch das notwendige Wechseln des Befehlssatzes nach der Beendigung der Kopierfunktion zusätzliche Kosten. Um bei der Wahl der Allokations-Strategien DAMP und HAMP einen möglichst geringen Gesamtenergieverbrauch zu erreichen, ist es nötig, die Kosten der Kopierfunktion so niedrig wie möglich zu halten. Dies kann dadurch erreicht werden, daß eine minimale Anzahl von Iterationen für das Kopieren eines Datenblocks nötig ist. Durch die eingesparten Kopierkosten können den Prozessen größere dynamische Anteile zugewiesen werden, was wiederum den Gesamtenergieverbrauch sinken läßt. Der ARM-Befehlssatz bietet den Vorteil, daß pro Iteration die doppelte Anzahl von Datenworten kopiert werden kann. Allerdings hat die Wahl dieses Befehlssatzes für die Kopierfunktion den Nachteil, daß durch die 32-Bit breiten Befehle mehr Waitstates und eine höhere Zugriffsenergie anfallen. Wird die Kopierfunktion allerdings in den Scratchpad-Speicher ausgelagert, dann wird sie einen deutlich besseren Energieverbrauch als ihr Thumb-Modus-Äquivalent aufweisen, da für diesen Speicher keine Waitstates anfallen und die Energie pro Zugriff für 16-Bit und 32-Bit Datenworte dieselbe ist. Das in Abschnitt 4.6 vorgestellte Verfahren zur Erzeugung der Multiprozess-Binärdatei ist in der Lage, die Kopierfunktion in den Scratchpad-Speicher auszulagern.

In Abschnitt 2.2 wurde ein Energiemodell für den ARM7TDMI-Prozessor vorgestellt. Auf Basis dieses Modells wird nun ein Energiemodell für die Kopierfunktion des Dispatchers entwickelt, für das folgende Annahme getroffen wird. Da der Prozessor Thumb-Instruktionen vor ihrer Ausführung in ihre ARM-Äquivalente dekodiert, wird davon ausgegangen, daß derselbe Instruktionsstrom sowohl für Thumb- als auch für ARM-Instruktionen fließt. Dies sollte eine pessimistische Annahme darstellen, da für ARM-Befehle die

Dekodier-Einheit des Prozessors nicht angesprochen werden muß und die dadurch hervorgerufene Schaltkreisaktivität entfällt. Für den Energieverbrauch von ARM-Instruktionen wird hier also angenommen, daß er sich nur durch die höhere Speicher-Zugriffsenergie und eine andere Anzahl von Waitstates von dem der Thumb-Instruktionen unterscheidet.

Algorithm 6 Die Kopierfunktion in ARM-Assembler-Code

Require: r1 = Quelladresse, r2 = Zieladresse, r3 = Anzahl zu kopierender Bytes

```
1:                  movs  r3,r3,lsr 3
2:                  bcc   eight
3:                  ldmia r1!,{r4}
4:                  stmia r2!,{r4}
5: eight            movs  r3,r3,lsr 1
6:                  bcc   sixteen
7:                  ldmia r1!,{r4,r5}
8:                  stmia r2!,{r4,r5}
9: sixteen          movs r3,r3,lsr 1
10:                 bcc   thirtytwo
11:                 ldmia r1!,{r4-r7}
12:                 stmia r2!,{r4-r7}
13: thirtytwo       beq   done
14: thirtytwoloop   ldmia r1!,{r4-r11}
15:                 stmia r2!,{r4-r11}
16:                 subs  r3,r3,1
17:                 bne   thirtytwoloop
18: done
```

Unter dieser Annahme erfolgt die Umsetzung dieses Modells nun wie folgt. Um den Energieverbrauch eines Aufrufs der Kopierfunktion zu bestimmen, müssen die Anzahl der zu kopierenden Bytes, die Kopierrichtung (Scratchpad ⇒ Hauptspeicher oder Hauptspeicher ⇒ Scratchpad) und die Lage der Kopierfunktion (Scratchpad oder Hauptspeicher) bekannt sein. Mit Hilfe der Anzahl der zu kopierenden Bytes wird bestimmt, welche Codeblöcke innerhalb der Kopierfunktion ausgeführt werden. Nun wird auf jede ausgeführte Instruktion das Energiemodell aus Abschnitt 2.2 angewendet und nach Summation über alle ausgeführten Instruktionen erhält man den Energieverbrauch für einen Aufruf der Kopierfunktion.

Die Anwendung dieses Modells für die Kopierfunktion soll an einem Beispiel gezeigt werden. Algorithmus 6 zeigt den Assembler-Code der Kopierfunktion. Soll ein 68 Byte langer Datenblock kopiert werden, dann entspricht dies 17 Datenworten. Da die Kopierfunktion in der Schleife `thirtytwoloop` acht Datenworte pro Iteration zu kopieren kann, wird diese Schleife zweimal durchlaufen und vorher ein einzelnes Datenwort kopiert.

In Zeile 1 von Algorithmus 6 wird die Anzahl der zu kopierenden Byte durch acht geteilt. Mit einer Division durch vier wird die Anzahl der zu kopierenden Datenworte ermittelt, eine weitere Division durch zwei stellt fest, ob eine ungerade Anzahl von Datenworten kopiert werden soll. Das Carry-Bit wird gleich dem Wert des Bits gesetzt, das als letztes aus dem Register `r3` herausgeschoben wird. Für die genannte Eingabe (Binärdarstellung: 01000100b) ist dies ein gesetztes Bit. Der `bcc`-Befehl springt, wenn das Carry-Bit nicht gesetzt ist. Also wird er nicht ausgeführt und in den Zeilen 3 und 4 ein Datenwort kopiert. In Zeile 5 wird das nächste Bit herausgeschoben und, da es den Wert 0 hat, der Sprungbefehl in Zeile 6 ausgeführt. Gleiches gilt für die Zeilen 9 und 10. In Zeile 13 angelangt, hält das Register r3 den Wert 00000010b und es wird noch zweimal die Schleife durchlaufen und es werden zwei Datenblöcke zu je acht Datenworten kopiert.

Nachdem die ausgeführten Instruktionen bekannt sind, kann das Energiemodell auf diese angewendet werden. Als Stromwert für die Instruktionen werden die für den Thumb-Befehlssatz ermittelten Werte verwendet und die entsprechende Zugriffsenergie und Anzahl von Waitstates für die 32-Bit breiten Befehle berücksichtigt. Nach der Summation über alle Instruktionen erhält man den Gesamtenergieverbrauch für das Kopieren des 17 Datenworte langen Datenblocks.

4.4 Messen der Energiefunktionen

Algorithm 7 Energiefunktion messen für Allokations-Strategien SAMP und DAMP

Require: Scratchpad-Größe M, Schrittweite d, Applikation app

1: $i = 0$
2: **while** $i \leq M$ **do**
3: encc mit Scratchpad-Anteil i Byte und app aufrufen
4: Energieverbrauch und Taktzyklenanzahl mit enprofiler bestimmen
5: $i = i + d$
6: **end while**
7: Energiefunktion und Taktzyklenfunktion ausgeben

Gegeben ist ein Scratchpad der Größe M. Die Energiefunktion für eine Applikation wird bestimmt, indem für eine Menge von Anteilen am Scratchpad-Speicher der Energieverbrauch der Applikation gemessen wird. Zum Optimieren einer Applikationen für einen statischen und/oder dynamischen Anteil am Scratchpad wird der in Abschnitt 4.2 vorgestellte Compiler encc eingesetzt. Für die resultierende Binärdatei werden mit dem enprofiler der Energieverbrauch und die Anzahl der Taktzyklen, die die Applikation für einen Durchlauf benötigt, gemessen. Wie die Messung der Energiefunk-

tion konkret erfolgt, hängt von der Allokations-Strategie ab, die später auf die gewonnenen Daten angewendet werden soll.

Für den Fall der Allokations-Strategien SAMP und DAMP zeigt Algorithmus 7 im Pseudocode wie die Messung durchgeführt wird.

Algorithm 8 Energiefunktion messen für Allokations-Strategie HAMP

Require: Scratchpad-Größe M, Schrittweiten $d1$, $d2$, Applikation app, Zeitscheibenlänge t

1: encc mit 0 Byte statischem und dynamischen Scratchpad-Anteil und app aufrufen
2: Taktzyklenanzahl c mit dem ARMulator bestimmen
3: $c_{stat} = c$
4: $i = 0$
5: **while** $i \leq M$ **do**
6: $j = 0$
7: **while** $j \leq M - i$ **do**
8: **if** $j == 0$ **then**
9: Anzahl der Unterbrechungen $u = c_{stat}/t$
10: **else**
11: Anzahl der Unterbrechungen $u = c/t$
12: **end if**
13: encc mit i Byte statischem und j Byte dynamischem Scratchpad-Anteil, u und app aufrufen
14: Nutzenwert bestimmen
15: Taktzyklenanzahl c mit dem ARMulator bestimmen
16: **if** $j == 0$ **then**
17: $c_{stat} = c$
18: **end if**
19: $j = j + d2$
20: **end while**
21: $i = i + d1$
22: **end while**
23: Nutzenwerte und Taktzylenanzahlen ausgeben

Da das Durchmessen großer Applikationen einige Zeit in Anspruch nehmen kann, wurden zwei Verfahren implementiert, um den Ablauf zu beschleunigen. Der erste Ansatz versucht Bereiche, in denen die Energiefunktion konstant ist, zu erkennen. Dies geschieht, indem eine binäre Suche auf den Suchbereich angewendet wird. Ergeben die Messungen am Anfang und Ende des Suchintervalls denselben Wert, dann wird die Energiefunktion dort als konstant angenommen. Im anderen Fall wird der Suchbereich in der Mitte geteilt und das Verfahren rekursiv fortgesetzt. Manchmal reicht auch diese Beschleunigung nicht aus, um die Energiefunktion in angemessener Zeit zu bestimmen. Für einige große Applikationen kann es mehrere Minuten dau-

ern, bis der Energieverbrauch mit dem enprofiler berechnet wurde. Wie in Abschnitt 4.2 beschrieben, wird für eine zu optimierende Applikation ein Rucksackproblem definiert. Die Idee besteht nun darin, den Nutzenwert als Maß für den Energieverbrauch zu werten. Damit entfällt ein Durchmessen der Binärdatei mit dem enprofiler. Wird später eine Allokations-Strategie auf diese Daten angewendet, muß darauf geachtet werden, daß nur Daten, die mit der gleichen Meßmethode bestimmt wurden, miteinander kombiniert werden.

Das Messen der Energiefunktion für die Allokations-Strategie HAMP zeigt Algorithmus 8. Soll eine Applikation für einen statischen und dynamischen Anteil am Scratchpad optimiert werden, dann muß die Anzahl der Unterbrechungen u bekannt sein, damit die Kopierkosten für den dynamischen Anteil richtig berechnet werden können. Das Verfahren beginnt deshalb damit, die Applikation für 0 Byte statischen und dynamischen Anteil zu übersetzen und ermittelt die Anzahl der Taktzyklen c. Dann wird Schrittweise der dynamische Anteil erhöht, wobei die Taktzyklenanzahl c der letzten Messung eine Abschätzung dafür gibt, wie lang die Applikation ungefähr für einen Durchlauf benötigt. Nachdem der dynamische Anteil voll ausgeschöpft wurde, beginnt die Messung für den nächsthöheren statischen Anteil. Hier dient als Abschätzung der Taktzyklenanzahl der Wert c_{stat}, der zum letzten gemessenen statischen Anteil mit 0 Byte dynamischem Anteil gehört.

4.5 Minimierung des Energieverbrauchs für mehrere Prozesse

Für die Minimierung des Energieverbrauchs mehrerer Applikationen wird auf die gemessenen Energiefunktionen zurückgegriffen. Die Allokations-Strategien SAMP und DAMP benötigen als Eingabe die in Abschnitt 3.1 vorgestellten Energiefunktionen. HAMP greift auf die erweiterten Energiefunktionen, wie sie in Abschnitt 3.4.1 eingeführt wurden, zurück. Auf Basis dieser Eingaben berechnen die Allokations-Strategien unter Verwendung der in Kapitel 3 vorgestellten Algorithmen eine Verteilung des Scratchpad-Speichers unter den Applikationen. Die hierbei generierte Ausgabe dient als Eingabe für die im nächsten Abschnitt beschriebene Generierung einer Multiprozess-Binärdatei.

Die einzige Besonderheit stellt an dieser Stelle die Allokations-Strategie DAMP dar. Hier muß das Energiemodell für die Kopierfunktion, welches in Abschnitt 4.3 eingeführt wurde, genutzt werden, um die Optimierung korrekt durchführen zu können. Abhängig davon, ob die Daten der Energiefunktionen mit enprofiler bestimmt wurden oder Nutzenwerte des Rucksackproblems darstellen, muß auf die richtige Variante des Energiemodells für die Kopierfunktion zurückgegriffen werden.

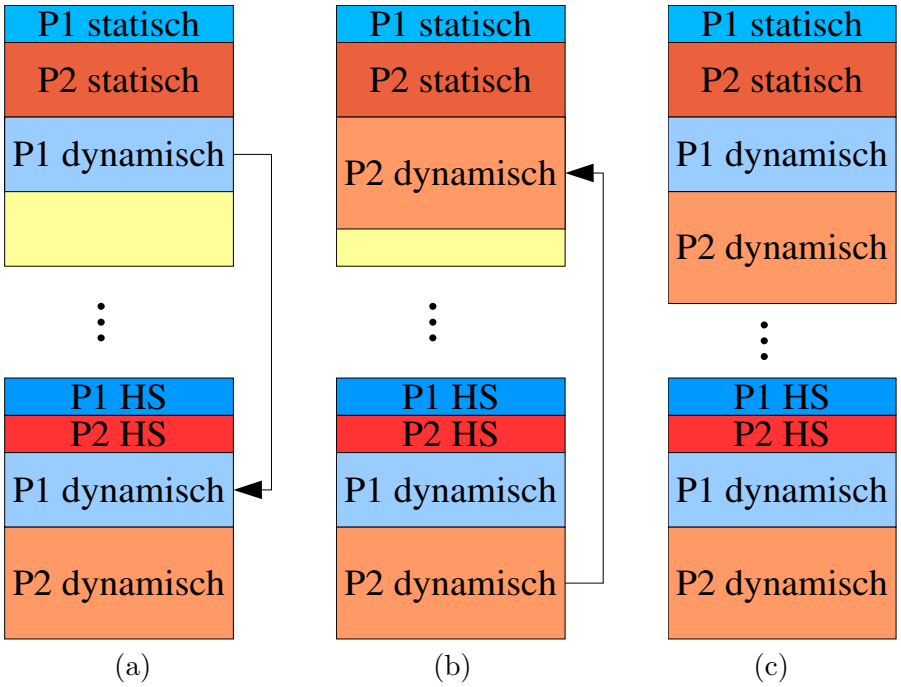

Abbildung 4.2: (a) Kopieren des dynamischen Anteils zurück in den Hauptspeicher, (b) Prozess P2 erlangt die Kontrolle über den Prozessor, sein dynamischer Anteil muß ins Scratchpad kopiert werden, (c) das für die Binärdatei gewählte Speicherlayout

4.6 Erstellung einer Multiprozess-Binärdatei

Für die Erzeugung einer Multiprozess-Binärdatei wird keine Unterscheidung zwischen den Allokations-Strategien SAMP, DAMP und HAMP getroffen. SAMP und DAMP sind Spezialfälle von HAMP, die keinen dynamischen bzw. statischen Anteil am Scratchpad-Speicher haben. Bevor schrittweise erläutert wird, wie die Erzeugung einer Multiprozess-Binärdatei abläuft, soll noch das gewählte Speicherlayout für die dynamischen Anteile vorgestellt werden.

In Abbildung 4.2 (a) sind die einzelnen Speicherbereiche zweier Prozesse P1 und P2 dargestellt. `P1 statisch` und `P1 dynamisch` kennzeichnen die statischen und dynamischen Anteile des Prozess P1 am Scratchpad. `P1 HS` bezeichnet den Teil von Prozess P1, der im Hauptspeicher abgelegt wird. Gleiches gilt für Prozess P2. In Abbildung 4.2 (a) ist die Situation dargestellt, in der der Dispatcher Prozess P1 die Kontrolle über den Prozessor entzieht und der dynamische Anteil von P1 zurück in den Hauptspeicher kopiert werden muß. Bevor Prozess P2 ausgeführt werden kann, muß der Dispatcher den dynamischen Anteil dieses Prozesses in den Scratchpad-Speicher

kopieren (Abbildung 4.2 (b)).

Möchte man den Energieverbrauch einer Binärdatei, die ein solches Speicherlayout verwendet, mit dem enprofiler berechnen, dann ergibt sich folgendes Problem: enprofiler führt genau Buch darüber, auf welche Codeblöcke und Variablen wie oft zugegriffen wird. Jedoch überlagern sich die in den dynamischen Anteilen von P1 und P2 liegenden Codeblöcke und Daten in ihren Speicheradressen. Ohne tiefgreifende Veränderungen am enprofiler kann nicht festgestellt werden, ob momentan z.B. auf eine Variable des Prozesses P1 oder P2 zugegriffen wird.

Um dieses Problem zu umgehen, wird ein Speicherlayout, wie es in Abbildung 4.2 (c) dargestellt ist, gewählt. Das Scratchpad wird soweit vergrößert, daß jeder Prozess seinen dynamischen Anteil am Scratchpad erhält, der sich mit dem anderer Prozesse nicht überlagert. Bevor der Dispatcher einem Prozess die Kontrolle über den Prozessor gibt und nachdem er sie ihm entzieht, wird der dynamische Anteil des Prozesses in seinen Teil des Scratchpads bzw. zurück in den Hauptspeicher kopiert. Würde man die Multiprozess-Binärdatei für ein reales System erstellen, dann würde es nur einen dynamisch genutzten Speicherbereich im Scratchpad geben, in den der Code und die Daten der Prozesse zwischen den Prozesswechseln kopiert werden.

Nach diesen Vorbetrachtungen erfolgt die Erzeugung der Binärdatei nun wie folgt:

1. Alle Applikationen werden mit dem encc für ihre Aufteilung in einen statischen und dynamischen Anteil übersetzt. Ergebnis sind drei Assemblerdateien pro Applikation, die den Code und die Daten enthalten, die im statischen und dynamischen Teil des Scratchpads sowie im Hauptspeicher abgelegt werden.

2. Damit die Assemblerdateien zu einer Binärdatei zusammengelinkt werden können, müssen eindeutige Bezeichner für Variablennamen und Sprungadressen generiert werden.

3. Alle Assemblerdateien werden übersetzt und zu einer Binärdatei gelinkt.

4. Damit der Dispatcher die dynamischen Anteile kopieren kann, müssen deren Startadressen und Größen, sowie deren Inhalt bekannt sein. Die Startadressen und Größen der dynamischen Anteile werden anhand der Ausgabe des Linkers bestimmt. Um die Inhalte dieser Anteile zu ermitteln, wird die Multiprozess-Binärdatei in den ARMulator geladen und ein Speicher-Dump durchgeführt.

5. Die Binärdatei wird erneut gelinkt, wobei jetzt das Speicherabbild der dynamischen Anteile mit in die Binärdatei aufgenommen und in den Hauptspeicher abgebildet wird.

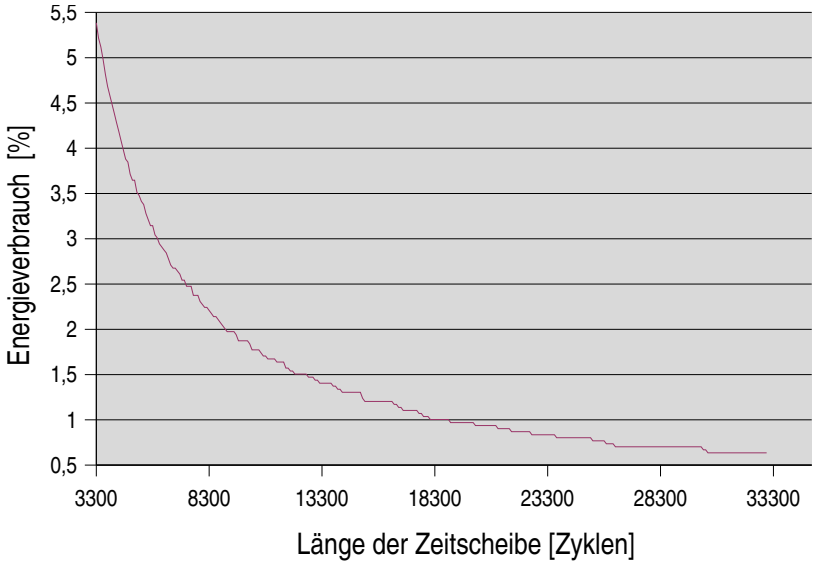

Abbildung 4.3: Prozentualer Anteil des Energieverbrauchs von Dispatcher und Scheduler am Gesamtenergieverbrauch

4.7 Messen des Energieverbrauchs einer Multiprozess-Binärdatei

Folgende Programmteile tragen einen Anteil am Gesamtenergieverbrauch der Multiprozess-Binärdatei:

- Dispatcher und Scheduler
- Die Kopierfunktion
- Die Terminate-Funktion
- Die einzelnen Prozesse

Der Energieverbrauch von Dispatcher, Scheduler und Terminate-Funktion wird nicht mitgezählt. In [ABR03] wurden Messungen an einem Realtime-OS durchgeführt, um den Energieverbrauch der einzelnen Komponenten des Betriebssystems zu bestimmen. Dabei wurde die Schalthäufigkeit des Dispatchers zwischen 100 Hz und 10 kHz variiert und es zeigte sich, daß Dispatcher und Scheduler zu weniger als 1% zum Gesamtenergieverbrauch beitragen. Anhand eines Energiemodells für Dispatcher und Scheduler wurden für diese Arbeit diesbezüglich eigene Experimente durchgeführt. Abbildung 4.3 zeigt eines davon. Es wurde für fünf Applikationen die Länge der Zeitscheibe von 3300 Zyklen (entspricht einer Schalthäufigkeit von 10 kHz bei 33 MHz Taktfrequenz) bis 33000 Zyklen (entspricht einer Schalthäufigkeit von 1 kHz)

variiert. Bei der kleinsten Zeitscheibenlänge tragen Dispatcher und Scheduler nur zu ca. 5% zum Gesamtenergieverbrauch bei, dieser Energieverbrauch wird deshalb als vernachlässigbar betrachtet.

Die Energie, die in der Terminate-Funktion verbraucht wird, ist von der Laufzeit der Prozesse und der Länge der Zeitscheibe abhängig. Die hier vorgestellten Allokations-Strategien nehmen keinen Einfluß auf die Laufzeit, die auf die Terminate-Funktion entfällt, außerdem stellen die hierdurch entstehenden Kosten feste Zusatzkosten dar. Es soll der Nutzen der Strategien bezogen auf den Energieverbrauch, der während der Laufzeit der Prozesse entsteht, bewertet werden. Deshalb fließt der Energieverbrauch der Terminate-Funktion nicht in den Gesamtenergieverbrauch ein, es werden nur der Energieverbrauch der Prozesse und der Kopierfunktion gemessen.

Der Energieverbrauch der Prozesse läßt sich auf die gewohnte Art und Weise mit dem enprofiler bestimmen. Die Kopierfunktion besteht wie der Dispatcher aus ARM-Code, den enprofiler nicht analysieren kann. In Abschnitt 4.3 wurde ein Energiemodell für die Kopierfunktion eingeführt. Dieses Modell wird bei der Analyse des Execution-Trace der Multiprozess-Binärdatei genutzt, um zusätzlich zum Energieverbrauch der Prozesse den Verbrauch der Kopierfunktion zu bestimmen. Wenn enprofiler bei der Analyse des Execution-Trace feststellt, daß der Dispatcher-Code erreicht wird, überprüft er, ob ein Sprungbefehl zur Kopierfunktion im Execution-Trace zu finden ist. Ist dies der Fall, wird anhand der in den nächsten Instruktionen geladenen Register bestimmt, wieviele Speicherelemente von welcher Quelladresse an welche Zieladresse kopiert werden sollen. Auf Basis dieser Daten kann das Energiemodell der Kopierfunktion die Energie berechnen, die für diesen Aufruf benötigt wurde. Am Ende der Analyse wird der durch die Aufrufe der Kopierfunktion verursachte Energieverbrauch zum Gesamtenergieverbrauch hinzugezählt.

4.8 Implementierung von Dispatcher und Scheduler

Der für diese Arbeit implementierte Dispatcher arbeitet preemptiv, der Scheduler wendet das Round-Robin-Verfahren an. Eine statische Priorisierung von Prozessen kann dadurch erreicht werden, daß einzelnen Prozessen mehrere aufeinanderfolgende Zeitscheiben zugewiesen werden.

Eine Besonderheit dieses Dispatchers ist, daß er vor und nach der Ausführung eines Prozesses zwei Speicherbereiche kopieren kann. Bei den Allokations-Strategien DAMP und HAMP wird der ganze bzw. ein Teil des Scratchpad-Speichers dynamisch genutzt, d.h. bevor ein Prozess die Kontrolle über den Prozessor erlangt, müssen Code und Daten, die er im dynamischen Teil des Scratchpads erwartet, dorthin kopiert werden. Nach der Ausführung des Prozesses müssen der ausgelagerte Code und die ausgelagerten Daten zu-

rück in den Hauptspeicher kopiert werden, da für nachfolgende Prozesse der dynamische Scratchpad-Speicher mit ihren Code- und Daten-Objekten überschrieben wird.

Der Dispatcher besteht aus folgenden Teilen: main-, dispatcher-, terminate und copy-Funktion, einer doppelt verketteten Liste von PCBs und allgemeinen Zustandsvariablen. Diese Teile werden nun im Detail beschrieben.

main-Funktion: Die main-Funktion schaltet den Prozessor in den User-Mode, installiert die dispatcher-Funktion als IRQ-Behandlungsroutine, setzt die Länge der Zeitscheibe und schaltet die periodische IRQ-Generierung ein. Erwartet der erste auszuführende Prozess Code oder Daten im dynamischen Scratchpad-Speicher, dann werden diese dorthin kopiert. Zum Abschluß wird die main-Funktion des Prozesses aufgerufen, dessen PCB der erste in der PCB-Liste ist.

Wie in Abschnitt 2.3 erwähnt, ist das ldmia/stmia Instruktionspaar besonders interessant für den Dispatcher, da mit nur einer Instruktion der Inhalt aller Register umgeladen werden kann. Voraussetzung ist allerdings, daß die Prozesse im User-Modus laufen. Deshalb schaltet die main-Funktion den Prozessor in diesen Betriebsmodus.

dispatcher-Funktion: Aufgabe der dispatcher-Funktion ist es, herauszufinden, ob der gerade unterbrochene Prozess noch weitere Zeitscheiben zugeteilt bekommt und falls dies nicht der Fall ist, einen Kontextwechsel durchzuführen. Innerhalb der dispatcher-Funktion werden folgende Schritte durchgeführt:

1. Bestimmen der Anzahl der noch verbleibenden Zeitscheiben. Stehen diesem Prozess noch Zeitscheiben zur Verfügung, wird die dispatcher-Funktion beendet und mit der Bearbeitung des unterbrochenen Prozesses fortgefahren. Andernfalls wird der nächste Punkt abgearbeitet.
2. Die Prozessor-Register werden in den PCB des unterbrochenen Prozesses gesichert.
3. Falls der Prozess Code oder Daten im dynamischen Scratchpad-Speicher hat, werden diese in den Hauptspeicher zurückkopiert.
4. Es wird nachgesehen, ob es noch aktive Prozesse gibt. Ist dies nicht der Fall, wird zu dem Programmteil zurückgekehrt, der die main-Funktion des Dispatchers aufgerufen hat.
5. Das Aktiv-Flag im PCB des unterbrochenen Prozesses gibt darüber Auskunft, ob dieser Prozess noch aktiv ist. Im negativen Fall wird er aus der Liste der PCBs entfernt.
6. Anhand des PCB wird der nächste auszuführende Prozess bestimmt.

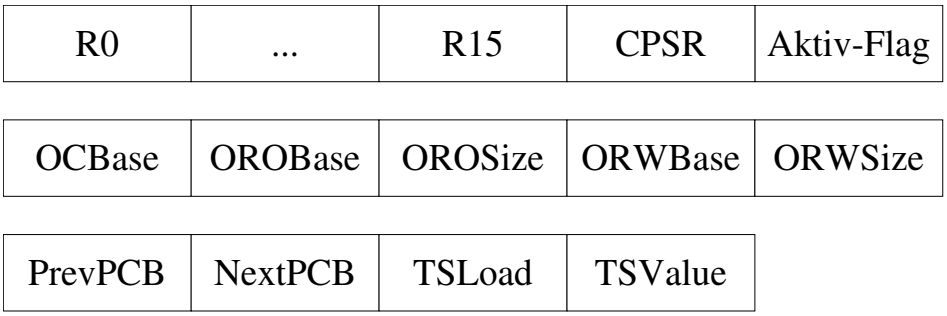

Abbildung 4.4: Aufbau eines Process-Control-Blocks

7. Falls für den nächsten auszuführenden Prozess Code oder Daten im dynamischen Scratchpad-Speicher erwartet werden, dann werden diese in diesem Schritt dorthin kopiert.
8. Die Anzahl der Zeitscheiben, die dieser Prozess aufeinanderfolgend zugeteilt bekommt, wird ermittelt.
9. Die Prozessor-Register werden aus dem PCB des Prozess wiederhergestellt und ihm wird die Kontrolle über den Prozessor übergeben.

terminate-Funktion: Wird ein Prozess beendet, dann springt er automatisch die terminate-Funktion an. Hier wird die Anzahl der noch aktiven Prozesse um eins verringert und das Aktiv-Flag im PCB des gerade beendeten Prozesses auf den Zustand inaktiv gesetzt. Abschließend wird in einer Endlosschleife auf die Unterbrechung durch die dispatcher-Funktion gewartet.

copy-Funktion: Die copy-Funktion kann Datenblöcke, deren Länge ein Vielfaches von vier Bytes beträgt, im Hauptspeicher bewegen. Um die Anzahl der Schleifendurchläufe und damit die Anzahl der notwendigen Instruktions-Fetches zu minimieren wird versucht, so oft wie möglich 32 Byte Datenblöcke zu kopieren. Nutzt man die ldmia/stmia Instruktionen mit acht Registern, so können pro Aufruf 32 Byte kopiert werden, da jedes Register vier Byte aufnehmen kann. Nähere Angaben zur Funktionsweise finden sich in Abschnitt 4.3.

Liste der PCBs: Abbildung 4.4 zeigt den Aufbau der PCBs. Im einzelnen haben die Einträge folgende Bedeutung:

1. Die ersten 16 Einträge nehmen die Register R0-R15 auf.
2. Das Status-Register CPSR wird im darauffolgenden Eintrag gespeichert.
3. Das Aktiv-Flag gibt darüber Auskunft, ob ein Prozess noch aktiv ist.

4. Der Eintrag OCBase zeigt auf einen Bereich im Hauptspeicher, in dem aufeinanderfolgend ein Read-Write und ein Read-Only Datenblock abgelegt sind. Die Einträge OROBase, OROSize sowie ORWBase und ORWSize kennzeichnen die Zieladressen dieser beiden Datenblöcke im dynamischen Teil des Scratchpads und deren Größe.
5. Die Zeiger NextPCB und PrevPCB zeigen auf PCBs, die zu aktiven Prozessen gehören und die in der doppelt verketteten Liste der PCBs den Vorgänger- bzw. Nachfolger-PCB darstellen.
6. Der Eintrag TSValue zeigt an, wieviele Zeitscheiben dem aktuell laufenden Prozess noch verbleiben. TSLoad enthält die Gesamtanzahl von aufeinanderfolgenden Zeitscheiben, die einem Prozess zugeteilt werden.

Zustandsvariablen: Die wichtigsten Zustandsvariablen sind `numproc`, Anzahl der noch aktiven Prozesse und `curproc`, ein Zeiger auf den PCB des aktuell laufenden Prozesses.

Die copy-Funktion hat einen wesentlichen Einfluß auf den Gesamtenergieverbrauch. Ihr Energieverbrauch schränkt die Allokation großer dynamischer Anteile ein. Um eine weitere Möglichkeit zum Senken des Gesamtenergieverbrauchs zu evaluieren, wurde der Dispatcher so realisiert, daß die copy-Funktion in den Scratchpad-Speicher ausgelagert werden kann.

4.9 Erweiterungen für den ARMulator

In der vorliegenden Version 4.60 des ARMulator fehlt die Möglichkeit, periodisch Interrupts generieren zu können. Der ARMulator wurde allerdings so realisiert, daß er um neue Funktionalitäten erweitert werden kann. Beispielsweise können durch entsprechende Erweiterungen ein Parallel-Port oder ein Coprozessor simuliert werden. Diese Erweiterungen werden Modelle genannt. [ARM01b] zeigt, wie diese Modelle erstellt werden.

Die in den beiden folgenden Abschnitten vorgestellten Erweiterungen basieren auf dem Tracer-Modell des ARMulators, das für die Generierung eines Execution-Trace eines ausgeführten Programms zuständig ist.

4.9.1 TracerIRQ-Modell

Das TracerIRQ-Modell löst folgende Probleme mit der vorliegenden Version des ARMulators:

1. Eine periodische Generierung von Interrupts wird nicht unterstützt.
2. Der ARMulator ist nicht in der Lage, Waitstates für Speicherzugriffe zu zählen. Würde man in dem neu zu entwickelnden Modell die

Funktion `ARMul_Schedule` verwenden, um nach einer festgelegten Zeitspanne einen Interrupt auszulösen, dann würden zu viele Instruktionen pro Zeitscheibe ausgeführt. Da z.B. beim Laden von Instruktionen aus dem Hauptspeicher Waitstates anfallen, bleiben weniger Zyklen in dieser Zeitscheibe für die auszuführenden Instruktionen. Werden keine Waitstates berücksichtigt, dann wird das Programm weniger oft unterbrochen und die Vorhersage nach dem DAMP- oder HAMP-Ansatz stimmt nicht mit der späteren Simulation der Multiprozess-Binärdatei im ARMulator überein.

3. Die vorliegende ARMulator-Version hat einen Programmfehler, der zur Folge hat, daß nur ein Zyklus für einen nicht ausgeführten bedingten Sprungbefehl gezählt wird. Laut Dokumentation von ARM ist aber jeder Sprungbefehl immer drei Zyklen lang.

Die Lösung besteht darin, das Tracer-Modell des ARMulators zu erweitern. Die Funktion `Tracer_Dispatch` dieses Modells wird für jede ausgeführte Instruktion und jeden Zugriff auf den Speicher aufgerufen.

Bei jedem Speicherzugriff muß nun innerhalb von `Tracer_Dispatch` überprüft werden, ob auf das Scratchpad oder den restlichen Speicher zugegriffen wird. Abhängig davon werden in einer Variable Waitstates gezählt. Für jede ausgeführte Instruktion muß ermittelt werden, ob es sich um einen konditionalen Sprung handelt und dieser nicht ausgeführt wurde. Ist dies der Fall, dann werden in einer Variable zwei Extra-Zyklen gezählt.

Bei jedem Aufruf von `Tracer_Dispatch` werden die Anzahl der verstrichenen Taktzyklen, ermittelt mit Hilfe der Funktion `ARMul_Time`, und die Extra-Waitstates bzw. -Zyklen mit der Länge der Zeitscheibe verglichen. Ist diese überschritten, dann wird ein neuer Interrupt ausgelöst.

4.9.2 CycleTracer-Modell

In Abschnitt 4.4 wurde besprochen, daß für die Energiemessungen für die Allokations-Strategie HAMP die Nutzenwerte des Rucksackproblems verwendet werden können. Es wird keine Messung des Energieverbrauchs und der Zyklenanzahl mit dem enprofiler durchgeführt. Für die Messungen müssen aber die Anzahl der Unterbrechungen des Prozesses bekannt sein, da hiervon die Kopierkosten und damit die Aufteilung der Objekte in den statischen und den dynamischen Scratchpad-Bereich abhängen.

Um die Zyklenanzahl des Programms bestimmen zu können, wird wieder das Tracer-Modell des ARMulators verändert. Wie im TracerIRQ-Modell werden Waitstates für Speicherzugriffe und Extra-Zyklen für nicht ausgeführte bedingte Sprungbefehle gezählt, es wird jedoch keine Trace-Datei geschrieben. Das Schreiben der Trace-Datei verlangsamt den Meßprozess um mehrere Größenordnungen und ist damit inakzeptabel, da es bei Messungen für die Allokations-Strategie HAMP im Gegensatz zu Messungen bei

SAMP und DAMP quadratisch viele Meßpunkte (bezogen auf die Größe des Scratchpad) gibt. Die Simulation selbst von sehr großen Applikationen gelingt mit diesem Ansatz im Bruchteil einer Sekunde.

Kapitel 5

Ergebnisse

In diesem Kapitel wird der erzielte Nutzen der Allokations-Strategien an vier Multiprozess-Systemen gezeigt. Dabei werden die einzelnen Strategien für verschiedene Scratchpad-Größen gegen das einfache Zuteilungsverfahren (im weiteren Verlauf SAOP genannt), das am Anfang von Abschnitt 3.1 vorgestellt wurde, und gegen ein System mit einem Cache gleicher Größe verglichen. Die vier Multiprozess-Systeme setzen sich aus folgenden Programmen zusammen:

Media: Dieses System enthält folgende Programme aus dem Multimedia-Bereich: `adpcm` (ein ADPCM Sprach-Kodierer/-Dekodierer), `g723` (G.723 Sprach-Kodierer/-Dekodierer), `edge-detection` (ein Programm aus der Bildverarbeitung zur Kantenextraktion) und `mpeg4` (ein MPEG4-Dekodierer). Die Gesamtgröße des Codes beträgt 7728 Byte und die der Daten 71068 Byte.

Mobile: Mit diesem Multiprozess-System wird eine mobile Anwendung, ähnlich wie das Handheld-Beispiel aus Kapitel 3, nachgebildet. Das Programm `gsm` dekomprimiert einen Sprach-Datenstrom nach dem GSM 06.10 Standard, `mpeg4` dekodiert einen MPEG4-Datenstrom. Der Code hat eine Gesamtgröße von 12800 Byte, die Daten sind 69468 Byte groß.

Sort: Die Sortieralgorithmen `bubblesort`, `heapsort`, `insertionsort`, `quicksort` und `selectionsort` bilden dieses Multiprozess-System. Die Codegröße beträgt 1428 Byte, die Datengröße 1720 Byte.

DSP: Dieses System besteht aus Programmen, die aus dem Bereich der digitalen Signalverarbeitung stammen und häufig in DSPs zu finden sind. Dazu gehören `fast-idct` (Fast Inverse Discrete Cosine Transform), `fft` (Fast Fourier Transform), `fir` (Fast Inverse Fourier Transform), `lattice-init` und `lattice-small` (Filterprogramme). Der Code dieser Programme hat eine Größe von 2592 Byte, die Daten belegen 24584 Byte.

Für diese Systeme ist folgendes zu beachten. Das Multiprozess-System **Media** wurde mit einer Zeitscheibenlänge von 33000 Zyklen gemessen. Für **Mobile** wurde eine Zeitscheibe der Länge 527000 Zyklen gewählt. Poletti et al. [FMA+04] betrachten, wie bereits in Abschnitt 2.5 erwähnt, eine Mischung aus Hardware- und Softwareansatz, um für ein Multiprozess-System den Energieverbrauch zu senken. Die Experimente wurden auf einem 200 MHz Prozessor durchgeführt, der verwendete Round-Robin-Scheduler hatte ein Schalthäufigkeit von 10 kHz. Aus diesem Grund wurde für den für diese Arbeit verwendeten 33 MHz ARM7TDMI-Prozessor eine Zeitscheibe von 33000 Zyklen gewählt, dies entspricht ebenfalls einer Schalthäufigkeit von 10 kHz. Die Applikationen `mpeg4` und `gsm`, aus denen **Mobile** besteht, haben eine sehr lange Laufzeit und sind eigentlich nicht für einen 33 MHz Prozessor geeignet. Ein Durchlauf dieser Applikationen dekodiert jeweils ein Datenpaket. Um eine ähnliche Anzahl von Unterbrechungen wie bei einem schnelleren Prozessor zu erhalten, wurde die Zeitscheibenlänge auf 527000 Zyklen festgelegt.

Sort und **DSP** wurden für die Zeitscheibenlängen 3300 Zyklen und 33000 Zyklen gemessen. Wird die Zeitscheibe weiter verkürzt, dann steigen die Kopierkosten für die Allokations-Strategien DAMP und HAMP stark an. Um den Nutzen zu bestimmen, der durch eine Auslagerung der Kopierfunktion auf das Scratchpad erzielt wird, wurden die Multiprozess-Systeme **Sort** und **DSP** zusätzlich mit einer Zeitscheibe von 3300 Zyklen gemessen, wodurch der prozentuale Anteil der Kopierkosten steigt. Somit sind die Verbesserungen bei einer Verlagerung der Kopierfunktion in das Scratchpad besser zu erkennen. Da die Kopierfunktion 72 Byte groß ist, kann sie für einen 64 Byte großen Scratchpad-Speicher nicht ausgelagert werden. Für die Multiprozess-Systeme **Media** und **Mobile** nicht variiert, da die Messung der erweiterten Energiefunktion für die einzelnen Applikationen dieser Systeme einen erheblichen zeitlichen Aufwand darstellt.

MP-System	Algorithmen [s]	Binärdatei-Erzeugung [min]	Energiemessung [min]
Media	13	268	127
Mobile	3	497	98
Sort	50	39	16
DSP	45	182	114

Tabelle 5.1: Laufzeit der Algorithmen, Binärdatei-Erzeugung und Energiemessungen

Tabelle 5.1 zeigt für jedes Multiprozess-System die Gesamtlaufzeit der SAMP-, DAMP- und HAMP-Algorithmen zur Berechnung der optimalen Aufteilung des zur Verfügung stehenden Scratchpad-Speichers, sowie die Laufzeiten für das Erzeugen der Binärdateien und das Berechnen des Ener-

gieverbrauchs. Bei der Erzeugung der Binärdateien wurde nicht auf bereits generierte Assemblerdateien zurückgegriffen, es wurden für jede Multiprozess-Binärdatei die einzelnen Applikationen neu übersetzt. Die Zeiten für das Messen der Energiefunktionen sind nicht in Tabelle 5.1 enthalten.

Alle Messungen wurden mit Hilfe des ARMulator durchgeführt. Für Systeme mit einem Scratchpad-Speicher wurde ein ARM7TDMI-Prozessor mit 33 MHz Taktfrequenz simuliert, für Systeme mit einem Cache-Speicher ein ARM710T-Prozessor mit der gleichen Taktfrequenz. In den Tabellen 5.2-5.4 werden Daten zu dem verwendeten Hauptspeicher, den Caches und Scratchpad-Speichern angegeben. Tabelle 5.2 macht Angaben zu den Waitstates und Zugriffsenergien für den Hauptspeicher für Byte-, Halbwort und Wortzugriffe, Tabelle 5.3 zeigt die Organisation und Zugriffsenergie der Caches. Die Cache-Speicher verwenden die Ersetzungsstrategie `Random`. In Tabelle 5.4 sind die Zugriffsenergien für Scratchpads verschiedener Größe aufgeführt. Diese Energien sind für die verschiedenen Zugriffsarten identisch.

Zugriffsart	Waitstates	Leseenergie [nJ]	Schreibenergie [nJ]
Byte	1	15,48	14,98
Halbwort	1	24,0	29,88
Wort	3	49,32	41,1

Tabelle 5.2: Zugriffsenergien und Waitstates für den Hauptspeicher

Größe [Byte]	Blöcke	Zugriffsenergie [nJ]
64	1	2,71
128	2	3,05
256	4	3,33
512	8	3,49
1024	16	3,75
2048	32	4,04
4096	64	4,71

Tabelle 5.3: Cache-Organisation und -Zugriffsenergien; die Assoziativität und Anzahl der Worte pro Cachezeile beträgt 4

Größe [Byte]	64	128	256	512	1024	2048	4096
Zugriffsenergie [nJ]	0,49	0,53	0,61	0,69	0,82	1,07	1,21

Tabelle 5.4: Scratchpad-Zugriffsenergien

5.1 SAMP

In Abschnitt 5.1.1 werden die vier Multiprozess-Systeme unter Verwendung der Allokations-Strategien SAMP und SAOP verglichen. Für alle Systeme erfolgt der Vergleich nur für Zeitscheiben der Länge 33000 bzw. 527000 Zyklen, da für keine der beiden Allokations-Strategien eine Änderung der Zeitscheibenlänge eine Auswirkung auf die Ergebnisse hat. In Abschnitt 5.1.2 wird SAMP gegen ein System mit einem Cache-Speicher verglichen. Da bei einer Verkürzung der Zeitscheibe die Prozesse öfter unterbrochen werden und somit potentiell mehr Cache-Misses auftreten, wenn sich Cachezeilen unterschiedlicher Prozesse gegenseitig verdrängen, hat die Länge der Zeitscheibe einen Einfluß auf die Güte der Ergebnisse für ein System mit Cache. Deshalb werden in Abschnitt 5.1.2 für die Multiprozess-Systeme **Sort** und **DSP** jeweils zwei Vergleiche für zwei unterschiedlich lange Zeitscheiben durchgeführt.

5.1.1 SAMP vs. SAOP

Die Abbildungen 5.1 (a)-(d) zeigen den Vergleich der Allokations-Strategien SAMP und SAOP. Auf der x-Achse sind die Scratchpad-Größen 64 Byte bis 4 kB aufgetragen. Die y-Achse zeigt den prozentualen Anteil des Energieverbrauchs von SAMP im Vergleich zur Allokations-Strategie SAOP. Hierbei entspricht die 100%-Basislinie dem Energieverbrauch der Allokations-Strategie SAOP.

Die größten Einsparungen werden für das Multiprozess-System **Sort** erzielt. Für eine Scratchpad-Größe von 4 KB spart der Einsatz der Allokations-Strategie SAMP mehr als 71% an Energie im Vergleich zu SAOP. Allerdings kann solch ein Multiprozess-System nicht in realen Anwendungen wiedergefunden werden. Das 4 kB große Scratchpad kann den gesamten Programmcode und alle Daten aufnehmen. Da SAOP nur einem Prozess eine Nutzung des Scratchpads ermöglicht, sind sehr große Einsparungen möglich. Würde die Allokations-Strategie SAOP so abgeändert, daß verbleibender Platz auf dem Scratchpad dem nächstbesten Prozess zugewiesen wird, dann würde SAMP auf den 1 und 2 kB großen Speichern wesentlich weniger Energie und auf dem 4 kB großen Scratchpad-Speicher keine Energie gegenüber SAOP einsparen können.

Für die Multiprozess-Systeme **Media**, **Mobile** und **DSP** fallen die Verbesserungen gegenüber SAOP mit durchschnittlich 13%, 9% und 18% weitaus geringer aus. Weiterhin läßt sich beobachten, daß für Scratchpad-Größen bis zu 256 Byte kaum mehr als 10% Einsparung erzielt werden. Dies ist darin begründet, daß jedes dieser Systeme eine Applikation beinhaltet, die den Energieverbrauch dominiert und für diese Applikation durch Nutzung von Scratchpad-Speicher große Energieeinsparungen erzielt werden. Für diese Situation berechnen die beiden Allokations-Strategien ähnliche Lösungen.

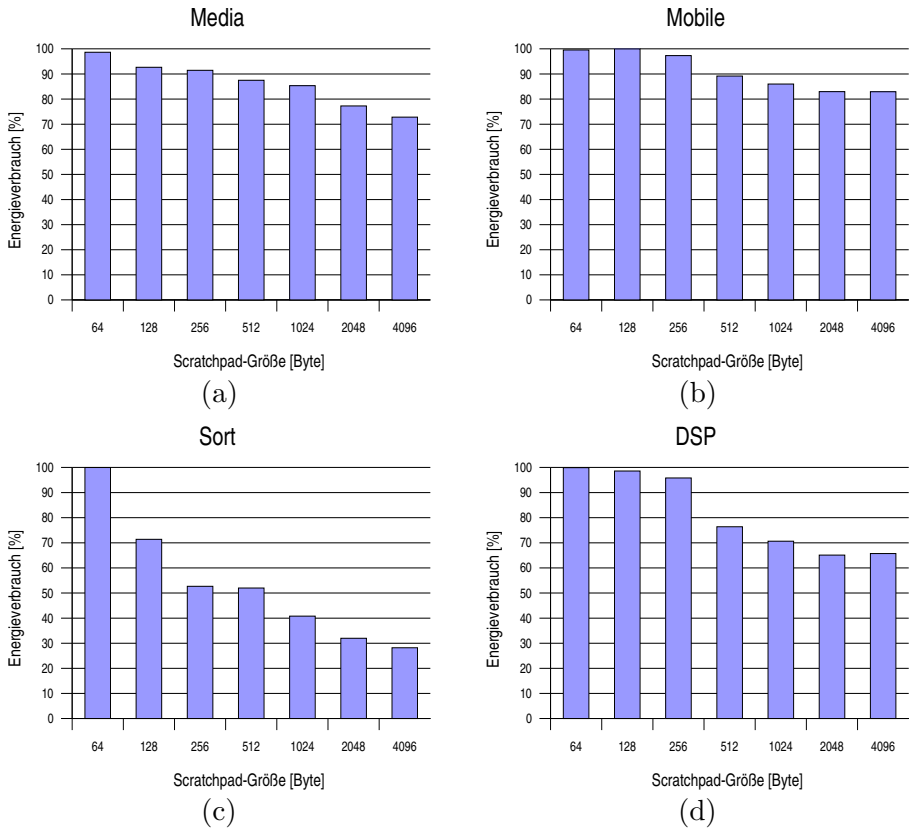

Abbildung 5.1: SAMP vs. SAOP: (a) **Media**, (c) **Sort** und (d) **DSP** bei einer Zeitscheibenlänge von 33000 Zyklen, (b) **Mobile** bei einer Zeitscheibenlänge von 527000 Zyklen

Im Falle von **DSP** ist anzumerken, daß für die Scratchpad-Größen 2 kB und 4 kB nicht der gesamte Speicherplatz ausgenutzt wird. Sämtlicher Code wurde ins Scratchpad ausgelagert, aber da nur noch große Arrays verbleiben, deren Kapazität die des Scratchpads übersteigt, können keine weiteren Einsparungen erzielt werden.

5.1.2 SAMP vs. Cache

Die Abbildungen 5.2 (a) und (b) sowie 5.3 (a)-(d) zeigen den Vergleich zwischen der Allokations-Strategie SAMP und einem System mit einem Cache. Da mehr Cache-Misses auftreten wenn die Zeitscheibe verkürzt wird, ist der Energieverbrauch des Multiprozess-Systems von dieser abhängig. Dieser Effekt wird in den Abbildungen 5.3 (a)-(d) verdeutlicht. In eingebetteten Systemen, die mit viel Hauptspeicher ausgestattet sind, kommt in der Regel eine Memory Mangement Unit (MMU) zum Einsatz. Bei einer einfachen

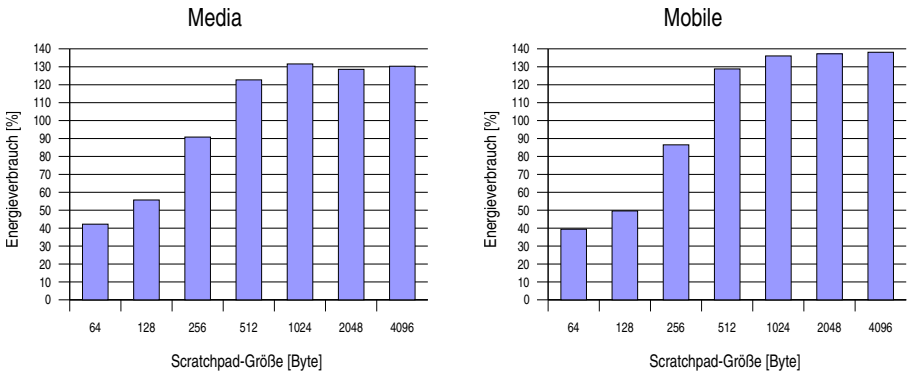

Abbildung 5.2: SAMP vs. Cache: (a) **Media** und (b) **Mobile**

Realisierung wird bei jedem Prozesswechsel der Cache-Inhalt gelöscht, damit keine ungültigen Daten und Instruktionen von anderen Prozessen verwendet werden. Durch diese Invalidierung des Cache-Inhalts beim Prozesswechsel wird bei den betrachteten Caches bei Speichergrößen von 1 bis 4 kB durchschnittlich 20% mehr Energie verbraucht.

Für alle Multiprozess-Systeme fallen die Energieeinsparungen von SAMP verglichen mit einem Cache deutlich schlechter aus, als dies bei einem Vergleich mit der einfachen SAOP Allokations-Strategie der Fall war. **Sort** verbraucht im Durchschnitt 6% weniger, **Media** genausoviel Energie, wenn SAMP eingesetzt wird. Bei **DSP** und **Mobile** wird durchschnittlich sogar 1% bzw. 2% mehr Energie verbraucht. Betrachtet man allerdings nur die Speichergrößen 64, 128 und 256 Byte, dann wird bei **Media** durchschnittlich 37%, bei **Mobile** 41% und bei **DSP** 40% weniger Energie beim Einsatz von SAMP verbraucht. Hieraus läßt sich schließen, daß besonders für kleine Systeme mit wenig Speicher der Einsatz eines Scratchpad und der Allokations-Strategie SAMP eine Verbesserung gegenüber einem Cache darstellen kann. Wird die Zeitscheibe auf 3300 Zyklen verkürzt, dann wird über den gesamten Bereich für **Sort** 17% und **DSP** 11% weniger Energie verbraucht, wenn SAMP verwendet wird.

Die Suche nach der Ursache für das schlechte Abschneiden bei den größeren Speichern erfordert eine genauere Betrachtung der Ergebnisse. Die Multiprozess-Systeme **Media**, **Mobile** und **DSP** zeigen ein ähnliches Verhalten. Bei kleinen Speichergrößen wird bei Verwendung eines Scratchpads weitaus weniger Energie verbraucht als bei einem System mit einem Cache. Das Multiprozess-System **Sort** zeigt ein abweichendes Verhalten, da ab einer bestimmten Größe soviel Code und Daten im Scratchpad abgelegt werden können, daß die durch die Prozesswechsel auftretenden Cache-Misses einen deutlich negativen Effekt auf den Energieverbrauch des Systems ha-

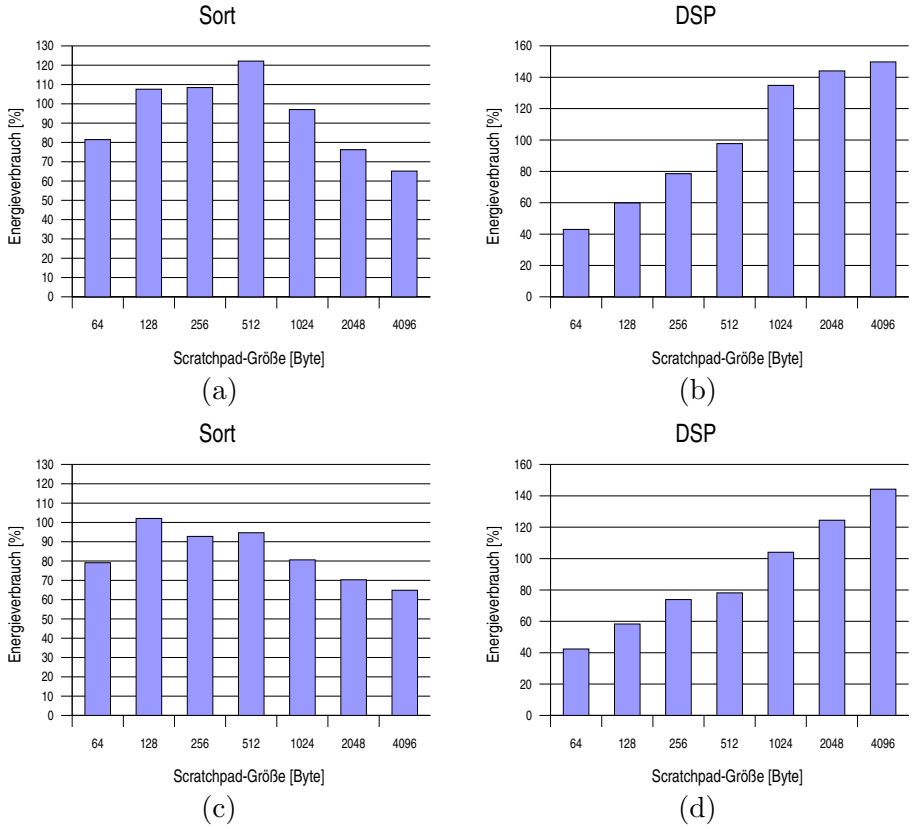

Abbildung 5.3: SAMP vs. Cache: (a) **Sort** und (b) **DSP** bei einer Zeitscheibenlänge von 33000 Zyklen, (c) **Sort** und (d) **DSP** bei einer Zeitscheibenlänge von 3300 Zyklen und der Kopierfunktion im Hauptspeicher

ben. Für die anderen Multiprozess-Systeme ist der höhere Energieverbrauch bei größeren Scratchpads in der zugrundeliegenden Optimierung nach Steinke [SWLM02] begründet. Im Gegensatz zu einem Cache-Speicher können nur die Programmobjekte einen Energiegewinn erzielen, die auf dem Scratchpad abgelegt wurden. Wird ein Cache verwendet, dann können potentiell alle Programmobjekte davon profitieren. Die drei Multiprozess-Systeme **DSP**, **Media** und **Mobile** enthalten Applikationen, die sehr große Arrays verwenden. **Media** z.B. enthält neun Arrays mit einer Größe von mehr als 1000 Byte, sieben Arrays mit mehr als 2000 Byte Größe und jeweils drei Arrays mit mehr als 4000 und 8000 Byte Größe. Es können nur wenige dieser Arrays überhaupt ins Scratchpad ausgelagert werden, für den Rest fällt die hohe Zugriffsenergie des Hauptspeichers an. Bei einem System mit Cache hingegen können beim wiederholten Zugriff auf diese Arrays Energieeinsparungen erzielt werden, da benötigte Teile der Arrays im Cache abgelegt werden. Der Einfluß der zugrundeliegenden Optimierung wird in Tabelle 5.5 verdeutlicht.

Es wurden für die Applikation `mpeg4` Messungen für einen Cache und ein Scratchpad mit Größen von 1 kB, 2 kB und 4 kB durchgeführt. Wird ein Cache verwendet, dann wird deutlich weniger Energie verbraucht. Da die verwendete Optimierung nicht in der Lage ist, für eine Applikation bessere Ergebnisse verglichen mit einem Cache zu erzielen, ist dies auch für ein Multiprozess-System nicht möglich, solang die Zeitscheibe nicht sehr kurz ist und sehr viele Cache-Misses auftreten.

Speichergröße [kB]	Energieverbrauch Cache [μJ]	Energieverbrauch Scratchpad [μJ]
1	47863	80011
2	47303	79907
4	47425	78896

Tabelle 5.5: Energieverbrauch für `mpeg4` bei Verwendung eines Caches bzw. eines Scratchpads und der Optimierung nach Steinke

5.2 DAMP

Abschnitt 5.2.1 zeigt den Vergleich der Allokations-Strategien DAMP und SAOP. In Abschnitt 5.2.2 wird DAMP mit einem Multiprozess-System, das einen Cache verwendet, verglichen. Da die Länge der Zeitscheibe einen Einfluß auf die Kopierkosten hat, wurden für die Systeme **Sort** und **DSP** Messungen mit einer 3300 und einer 33000 Zyklen langen Zeitscheibe durchgeführt. Wird die Kopierfunktion auf das Scratchpad ausgelagert, dann sinken die Kopierkosten und folglich ist es der Allokations-Strategie DAMP möglich, den Prozessen größere dynamische Anteile zuzuweisen, was wiederum den Energieverbrauch der einzelnen Prozesse sinken läßt. Aus diesem Grund wurden für die 3300 Zyklen lange Zeitscheibe für die Multiprozess-Systeme **Sort** und **DSP** zusätzlich Experimente mit einer ins Scratchpad ausgelagerten Kopierfunktion durchgeführt. Hierbei ist zu beachten, daß für kleine Speicher durch die Kopierfunktion soviel Platz belegt wird, daß sich nur geringe oder keine Einsparungen erzielen lassen.

Der Anteil der Kopierkosten am Gesamtenergieverbrauch liegt für **Media** zwischen 1 und 13%, für **Mobile** unter 1%, für **Sort** zwischen 2 und 20% und für **DSP** zwischen 1 und 7%. Wird die Zeitscheibe auf 3300 Zyklen verkürzt, dann steigen die Kopierkosten für **Sort** auf maximal 25% und für **DSP** auf maximal 28% an. Bei einer Verlagerung der Kopierfunktion ins Scratchpad liegen die Kopierkosten im Falle von **Sort** zwischen 6 und 39% und für **DSP** zwischen 2 und 19%.

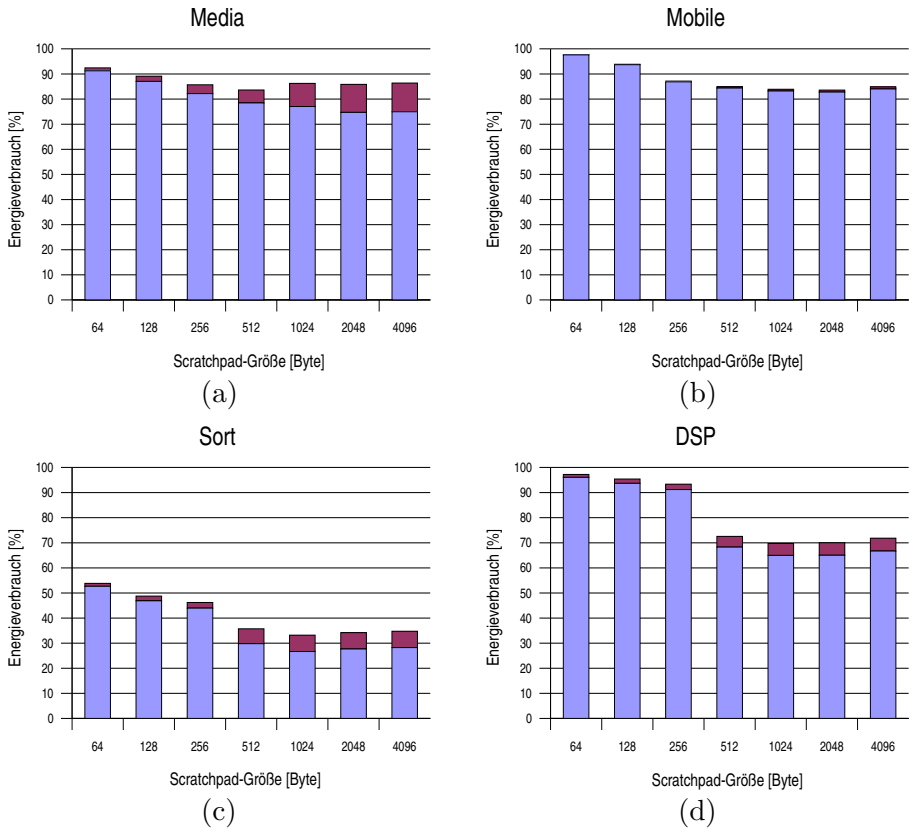

Abbildung 5.4: DAMP vs. SAOP: (a) **Media**, (c) **Sort** und (d) **DSP** bei einer Zeitscheibenlänge von 33000 Zyklen, (b) **Mobile** bei einer Zeitscheibenlänge von 527000 Zyklen

5.2.1 DAMP vs. SAOP

Die Abbildungen 5.4 (a)-(d) zeigen den Vergleich der Allokations-Strategien DAMP und SAOP für die großen Zeitscheiben. Es wird jeweils der Energieverbrauch der Prozesse der einzelnen Systeme dargestellt sowie im oberen Teil der Balken der Anteil der Kopierenergie. Für das Multiprozess-System **Media** wird bei Verwendung von DAMP durchschnittlich 13%, für **Mobile** 12%, für **Sort** 59% und für **DSP** 19% weniger Energie verbraucht. Die großen erzielten Einsparungen bei **Sort** sind in der niedrigen Programm- und Datengröße und geringen Ausführungszeit begründet. Bei allen Systemen zeigt sich das Verhalten, daß der relative Energieverbrauch zunächst bei zunehmender Scratchpadgröße sinkt und später wieder ansteigt. Ab einer gewissen Größe, die jeweils von den einzelnen Applikationen und deren Energieverbrauch abhängt, wird ein Kopieren von mehr Programmobjekten zu teuer. Dann können den Prozessen keine größeren dynamischen Antei-

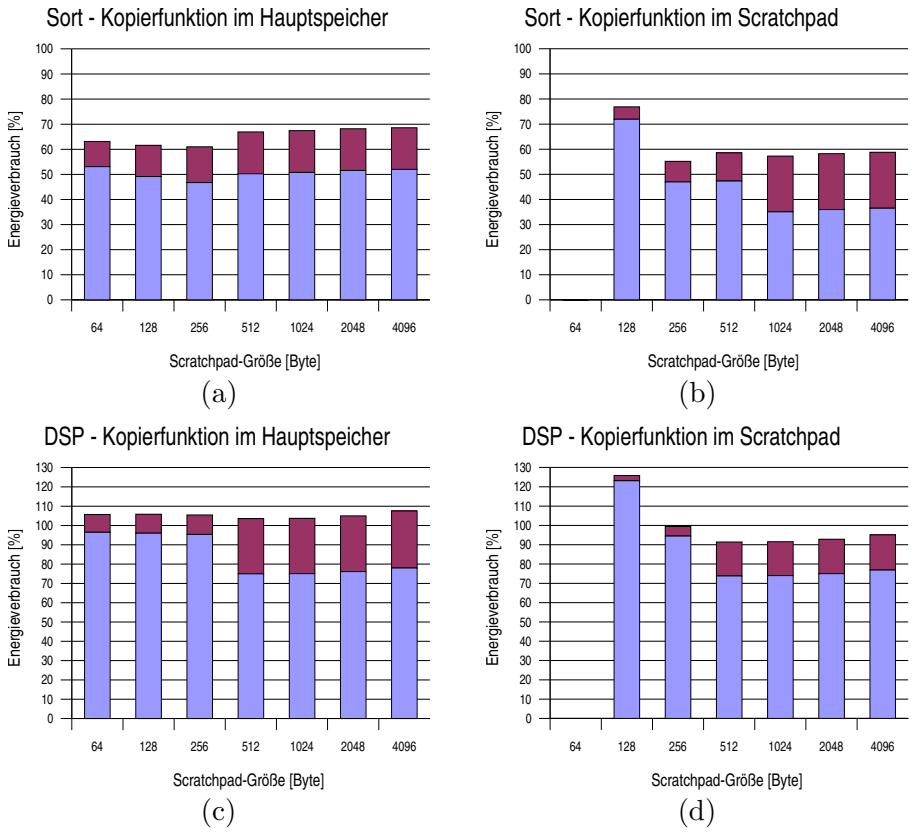

Abbildung 5.5: DAMP vs. SAOP: **Sort** und **DSP** bei Zeitscheibenlänge von 3300 Zyklen

le zugewiesen werden. Da aber die Zugriffsenergie für größere Scratchpads ansteigt, steigen auch die Kosten für die im Scratchpad abgelegten Programmobjekte und die Kosten für das Kopieren.

In den Abbildungen 5.5 (a)-(d) wird der Energieverbrauch von DAMP mit dem von SAOP verglichen. Dabei wird eine Zeitscheibe von 3300 Zyklen verwendet, um den Effekt der gestiegenen Kopierkosten besser sichtbar zu machen. Die Abbildungen 5.5 (b) und (d) zeigen die Ergebnisse für eine im Scratchpad abgelegte Kopierfunktion. Das Multiprozess-System **Sort** verbraucht beim Einsatz der Allokations-Strategie DAMP durchschnittlich 35% weniger Energie. Wird zusätzlich die Kopierfunktion im Scratchpad-Speicher abgelegt, dann liegen die durchschnittlichen Einsparungen bei 39%. Anders stellt sich die Situation für **DSP** dar. Liegt die Kopierfunktion im Hauptspeicher, dann wird im Durchschnitt 5% mehr Energie im Vergleich zur Allokations-Strategie SAOP verbraucht. Ein Verlagern der Kopierfunktion ins Scratchpad bringt im Durchschnitt 1% weniger Energieverbrauch beim Einsatz von DAMP. Auffällig ist, daß bei einem Scratchpad der Größe

128 Byte und einer dorthin ausgelagerten Kopierfunktion wesentlich mehr Energie verbraucht wird. Da für diese Speichergröße die Kopierfunktion den größten Anteil belegt, bleibt für die Prozesse weniger Scratchpad-Speicher und dementsprechend gering fällt auch der erzielte Nutzen aus. Betrachtet man den durchschnittlichen Energieverbrauch für die Speichergrößen 256 Byte bis 4 kB, dann spart **DSP** 6% und **Sort** 42% Energie beim Einsatz der Allokations-Strategie DAMP und einer im Scratchpad abgelegten Kopierfunktion.

5.2.2 DAMP vs. Cache

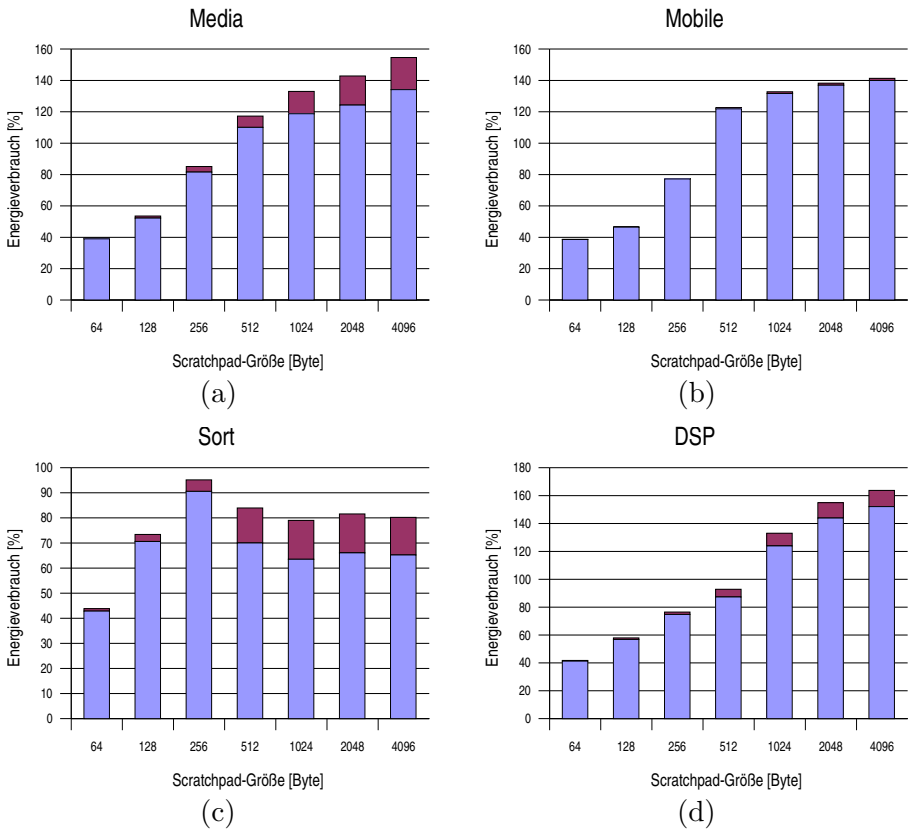

Abbildung 5.6: DAMP vs. Cache: (a) **Media**, (c) **Sort** und (d) **DSP** bei einer Zeitscheibenlänge von 33000 Zyklen, (b) **Mobile** bei einer Zeitscheibenlänge von 527000 Zyklen

Beim Vergleich von DAMP mit einem Cache zeigt sich ein ähnliches Verhalten wie beim Vergleich zwischen der Allokations-Strategie SAMP und einem System mit Cache. Für **Media**, **Mobile** und **DSP** (Abbildungen 5.6 (a), (b) und (d)) kann bis zu einer Speichergröße von 256 Byte Energie ein-

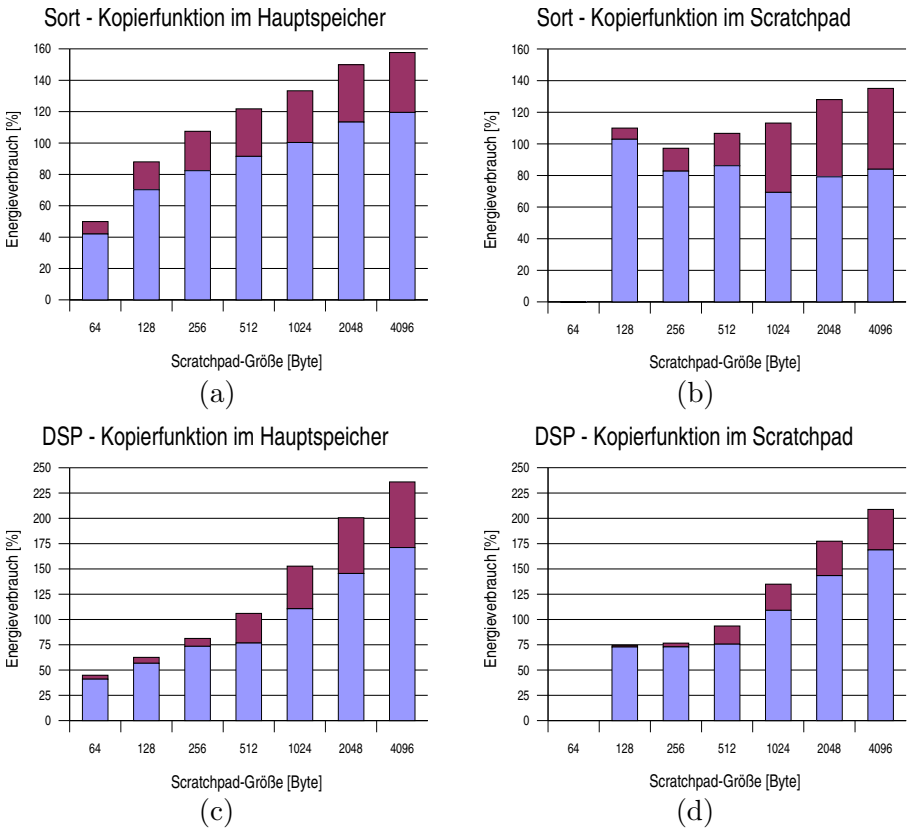

Abbildung 5.7: DAMP vs. Cache: **Sort** und **DSP** bei Zeitscheibenlänge von 3300 Zyklen

spart werden. Mit einer Ausnahme für **DSP** hat für 512 Byte und mehr Speicher ein System mit Cache klare Vorteile. Dies ist zum einen darin begründet, daß die Kopierkosten zu hoch werden und den Prozessen keine größeren dynamischen Anteile zugeteilt werden können und liegt desweiteren wie im Falle von SAMP auch in der zugrundeliegenden Optimierung begründet. Die Multiprozess-Systeme bestehen aus Applikationen, die große Arrays enthalten. Durch eine dynamische Nutzung des Scratchpads würde den einzelnen Applikationen das ganze Scratchpad zur Verfügung stehen, doch könnte selbst bei sehr geringen Kopierkosten nur ein Bruchteil der großen Arrays ins Scratchpad ausgelagert werden.

Für das Multiprozess-System **Sort** (Abbildung 5.6 (c)) können aufgrund der geringen Programmgröße Einsparungen gegenüber einem System mit Cache erzielt werden. Da die Zeitscheibe relativ lang ist und die einzelnen Applikationen eine geringe Größe haben, können nahezu der gesamte Code und alle Daten ins Scratchpad ausgelagert werden.

Im Durchschnitt wird beim Einsatz von DAMP für **Media** und **DSP**

3% mehr Energie verbraucht. Bei **Mobile** gibt es im Durchschnitt keine Einsparung gegenüber einem Cache, für **Sort** hingegen liegen sie bei 23%. Werden für die größeren Multiprozess-Systeme wieder nur die Speichergrößen von 64, 128 und 256 Byte betrachtet, dann wird für **Media** und **DSP** durchschnittlich 41% und für **Mobile** 46% Energie bei der Verwendung von DAMP gespart.

Wird die Zeitscheibe auf 3300 Zyklen verkürzt (Abbildungen 5.7 (a)-(d)), dann steigen die Kopierkosten sehr stark an und für **Sort** wird ab einer Speichergröße von 256 Byte mehr Energie für ein System mit Scratchpad als für ein System mit einem Cache-Speicher verbraucht. Durch eine Verlagerung der Kopierfunktion ins Scratchpad wird weiterhin mit 15% für **Sort** und 27% für **DSP** durchschnittlich mehr Energie gegenüber einem System mit Cache verbraucht. Bei den Speichergrößen von 1, 2 und 4 kB werden jedoch Verbesserungen von mehr als 20% gegenüber einer nicht ins Scratchpad ausgelagerten Kopierfunktion erzielt.

5.3 HAMP

In Abschnitt 5.3.1 werden die Allokations-Strategien SAOP und HAMP miteinander verglichen, Abschnitt 5.3.2 zeigt den Vergleich von HAMP mit einem Cache. Da die Ergebnisse für HAMP ebenfalls von den Kopierkosten und der Länge der Zeitscheibe abhängig sind, werden die Vergleiche für die Multiprozess-Systeme **Sort** und **DSP** hier ebenfalls für zwei unterschiedlich lange Zeitscheiben und mit einer Verlagerung der Kopierfunktion in das Scratchpad durchgeführt.

Die maximal anfallenden Kopierkosten sind im Vergleich zu DAMP geringer. Sie liegen für **Media** unter 5%, für **Mobile** unter 1%, für **Sort** unter 18% und für **DSP** unter 6%. Eine Verkürzung der Zeitscheibe auf 3300 Zyklen läßt die maximalen Kopierkosten für **Sort** auf 16% und für **DSP** auf 8% ansteigen. Wird die Kopierfunktion ins Scratchpad verlagert, sinken die Kosten jeweils um die Hälfte.

5.3.1 HAMP vs. SAOP

Die Abbildung 5.8 zeigt den Vergleich von HAMP mit der Allokations-Strategie SAOP für die langen Zeitscheiben. Die Idee hinter der Allokations-Strategie HAMP war es, die beste Aufteilung in mehrere statische Anteile für die einzelnen Prozesse und einen gemeinsam genutzten dynamischen Anteil zu berechnen. Da es bei kleinen Scratchpad-Speichern günstiger ist, einen größeren dynamischen Anteil bereitzustellen, sollte dort HAMP ähnliche Lösungen berechnen wie die Allokations-Strategie DAMP. Auf größeren Scratchpad-Speichern sollte HAMP wegen der gestiegenen Kopierkosten hingegen ähnliche Lösungen wie SAMP berechnen. Ein Vergleich der entsprechenden Abbildungen zeigt, das HAMP diesem Verhalten folgt. Dadurch ist

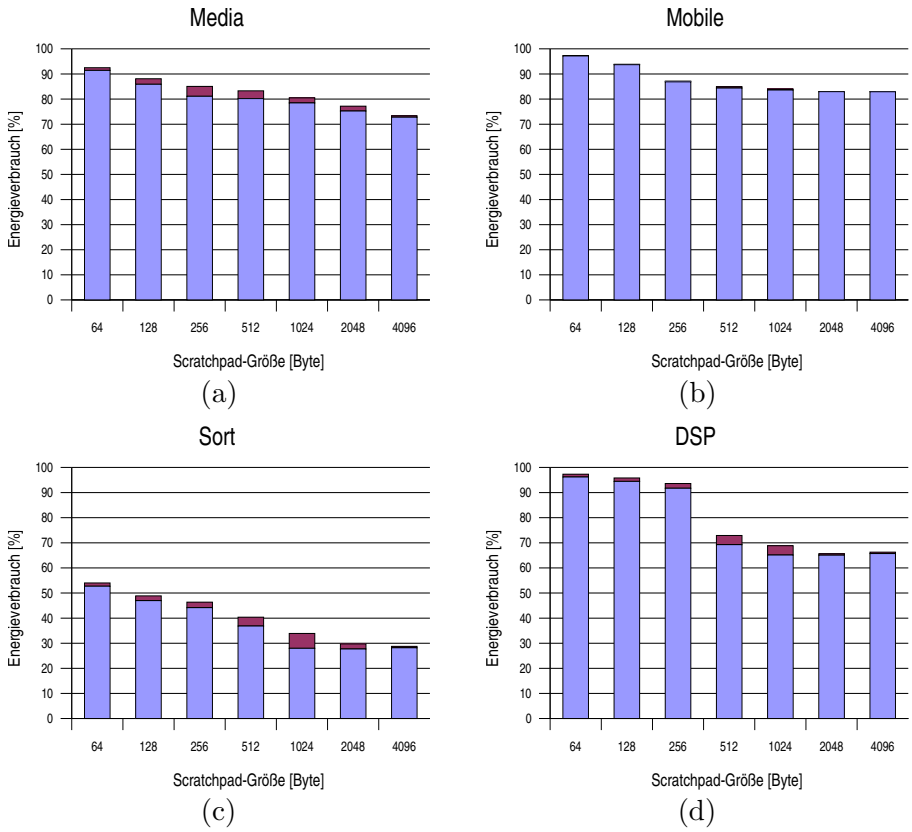

Abbildung 5.8: HAMP vs. SAOP: (a) **Media**, (c) **Sort** und (d) **DSP** bei einer Zeitscheibenlänge von 33000 Zyklen, (b) **Mobile** bei einer Zeitscheibenlänge von 527000 Zyklen

es möglich, bei HAMP etwas größere durchschnittliche Einsparungen zu erzielen. Diese liegen für **Media** bei 17%, für **Mobile** bei 12%, für **Sort** bei 60% und für **DSP** bei 23%.

Wie sich der Energieverbrauch für HAMP ändert, wenn eine sehr kurze Zeitscheibe gewählt wird, zeigen die Abbildungen 5.9 (a)-(d) für die Multiprozess-Systeme **Sort** und **DSP**. Die gestiegenen Kopierkosten führen dazu, daß die durchschnittlichen Energieeinsparungen auf 16% für **DSP** und 47% für **Sort** sinken. Wird die Kopierfunktion in das Scratchpad verlagert (Abbildungen 5.9 (b) und (d)), dann steigen die durchschnittlich erreichten Energieeinsparungen um jeweils ca. 3%. Der Energieverbrauch bei einem 128 Byte großen Scratchpad und einer dorthin verlagerten Kopierfunktion ist wie im Falle der Allokations-Strategie DAMP stark erhöht, da wegen der Größe der Kopierfunktion den Prozessen weniger als die Hälfte des Scratchpads zur Verfügung steht.

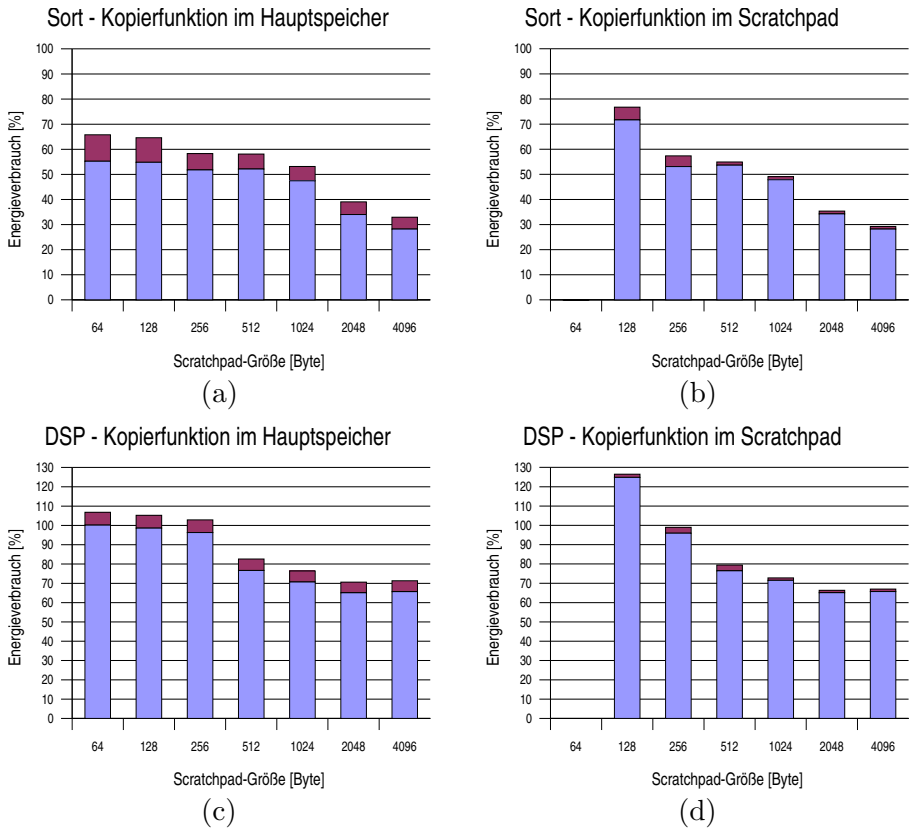

Abbildung 5.9: HAMP vs. SAOP: **Sort** und **DSP** bei Zeitscheibenlänge von 3300 Zyklen

5.3.2 HAMP vs. Cache

Wie beim Vergleich mit der Allokations-Strategie SAOP zeigt HAMP im Vergleich mit einem Cache bei kleinen Speichergrößen ähnliche Ergebnisse wie DAMP und bei größeren Speichern ähnliche Ergebnisse wie SAMP (vgl. Abbildung 5.10). Im Durchschnitt werden im Fall der Multiprozess-Systeme **Media**, **Mobile** und **Sort** 3% , 1% bzw. 25% Energie gegenüber einem System mit Cache eingespart. Bei **DSP** wird im Durchschnitt keine Energie eingespart. Werden wie im Falle der Strategien SAMP und DAMP nur die Speichergrößen 64, 128 und 256 Byte betrachtet, dann ergeben sich Einsparungen gegenüber einem Cache von durchschnittlich 41% für **Media** und **DSP** und 46% für **Mobile**. Warum Caches für größere Speicher mehr Energie einsparen, wurde bereits in Abschnitt 5.1.2 erläutert.

Die Abbildungen 5.11 (a)-(d) zeigen den Vergleich der Allokations-Strategie HAMP mit einem System mit Cache für die Multiprozess-Systeme **Sort** und **DSP** für eine Zeitscheibe der Länge 3300 Zyklen. Der Vergleich

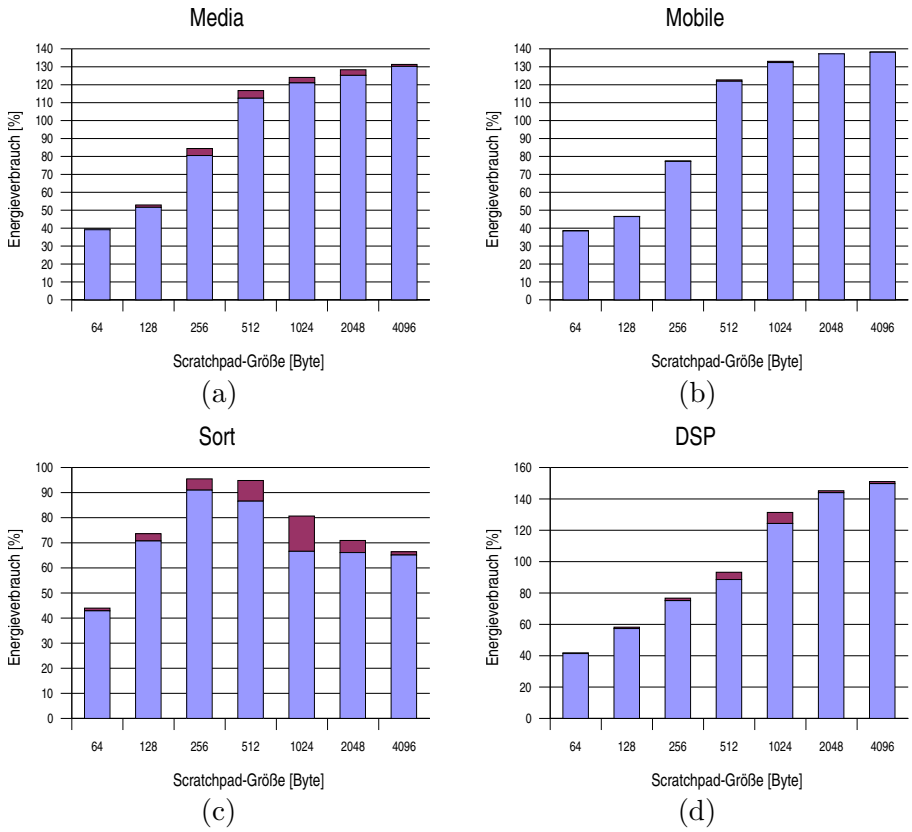

Abbildung 5.10: HAMP vs. Cache: (a) **Media**, (c) **Sort** und (d) **DSP** bei einer Zeitscheibenlänge von 33000 Zyklen, (b) **Mobile** bei einer Zeitscheibenlänge von 527000 Zyklen

mit der größeren Zeitscheibe zeigt, daß das System mit Cache beim Verkürzen der Zeitscheibe durch eine größere Anzahl von Cache-Misses einen Teil seiner Vorteile einbüßt. Die durchschnittlichen Energieeinsparungen liegen bei **Sort** bei 11% und für **DSP** bei 4%, falls die Kopierfunktion im Hauptspeicher abglegt wurde. Wird die Kopierfunktion ins Scratchpad verlagert, dann wird für die Scratchpad-Größen 128 Byte bis 4 kB durchschnittlich etwas mehr Energie verbraucht, jedoch können gegenüber einer im Hauptspeicher abgelegten Kopierfunktion bis zu 10% eingespart werden.

5.4 SAMP vs. DAMP vs. HAMP

In diesem Abschnitt werden die drei in dieser Arbeit vorgestellten Allokations-Strategien miteinander verglichen. Folgendes Verhalten wird erwartet: Auf kleinen Scratchpad-Speichern liefern DAMP und HAMP die besten Ergebnisse, da eine gemeinsame Nutzung eines kleinen Scratchpads einen größe-

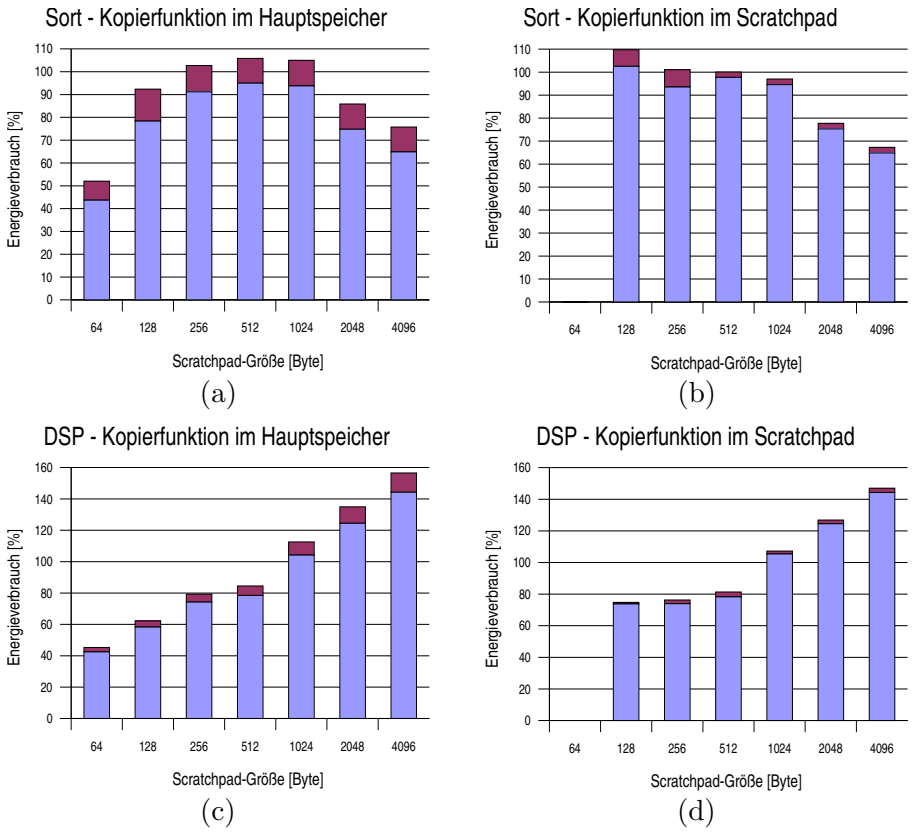

Abbildung 5.11: HAMP vs. Cache: **Sort** und **DSP** bei Zeitscheibenlänge von 3300 Zyklen

ren Nutzen liefert als dessen Zuweisung an einen Prozess oder eine Aufteilung unter mehreren Prozessen. Werden die größten verfügbaren Scratchpad-Größen betrachtet, dann liefert die Allokations-Strategie SAMP die besten Ergebnisse, da durch ein Aufteilen des Speichers in mehrere statische Anteil mehr Energie eingespart werden kann als durch das dynamische Nutzen eines großen Scratchpads mit hohen Kopierkosten. HAMP sollte für große Scratchpads ähnlich gute Ergebisse wie SAMP erreichen, doch durch den Mehraufwand der Kopierfunktion jeweils etwas mehr Energie verbrauchen. Bei den mittleren Scratchpad-Größen sollten durch mehrere statische Anteile und einen gemeinsam genutzten dynamischen Anteil die beste Ergebnisse durch HAMP erzielt werden.

Die Abbildungen 5.12 (a)-(d) zeigen den Vergleich von SAMP, DAMP und HAMP für die langen Zeitscheiben. In den Diagrammen werden jeweils absolute Energiewerte angegeben, der Übersichtlichkeit wegen werden die Kopierkosten weggelassen.

Es zeigt sich, daß das erwartete Verhalten größtenteils eintritt. HAMP je-

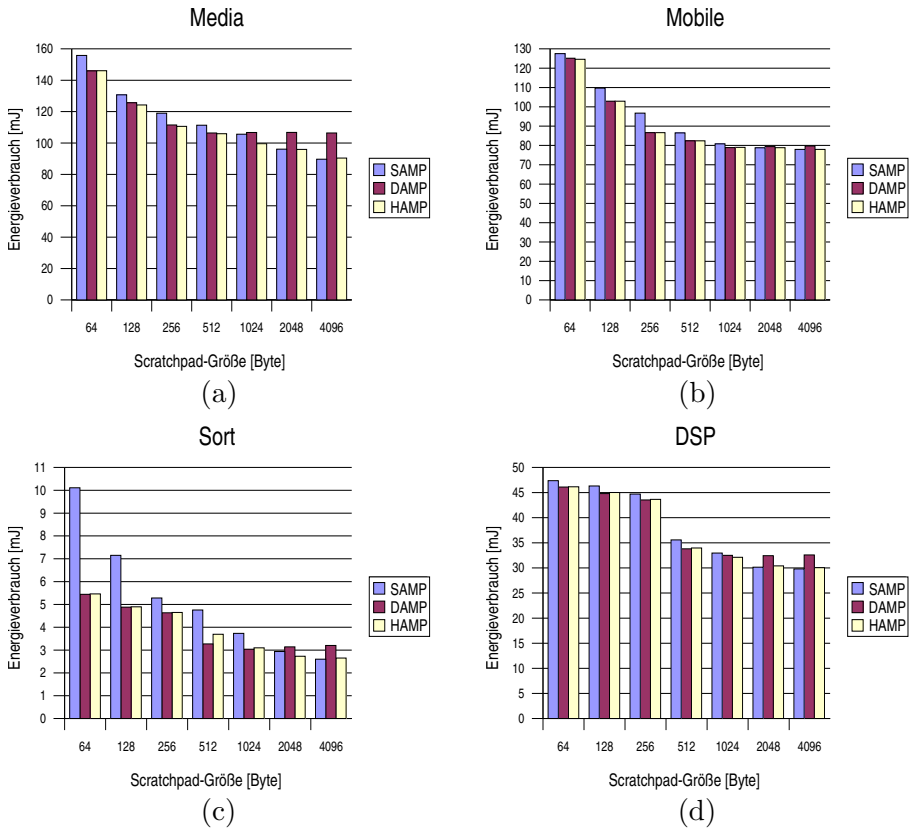

Abbildung 5.12: SAMP vs. DAMP vs. HAMP: (a) **Media**, (c) **Sort** und (d) **DSP** bei einer Zeitscheibenlänge von 33000 Zyklen, (b) **Mobile** bei einer Zeitscheibenlänge von 527000 Zyklen

doch sollte bei keinem Multiprozess-System für keine Speichergröße schlechtere Resultate als DAMP berechnen. Besonders deutlich ist dies für **Sort** bei 256 und 512 Byte zu sehen (vgl. Abbildung 5.12 (c)). Ursache dieses Verhaltens sind die erweiterten Energiefunktionen, also die Lösungen des HAOP-Problems. Vergleicht man die erweiterten Energiefunktion für einen statischen Anteil von 0 Byte mit den von DAMP berechneten Lösungen, dann zeigt sich, daß die für die erweiterten Energiefunktionen berechneten Nutzenwerte niedriger ausfallen. In Abschnitt 3.4.1 wurde HAOP auf diese Art und Weise definiert, weil die Lösungen mit einem ILP-Löser berechnet werden sollen. Diese Formulierung mit zwei Nutzenwerten für das statische und das dynamische Scratchpad stellt jedoch nur eine Näherung dar. Eine exakte Formulierung wird in Anhang A angegeben. Dort gibt es nur einen Nutzenwert pro Programmobjekt, der erzielte Gesamtnutzen wird jedoch um die Kosten reduziert, die für das Kopieren eines Speicherblocks anfallen, der aus den im dynamischen Anteil abgelegten Programmobjekten besteht.

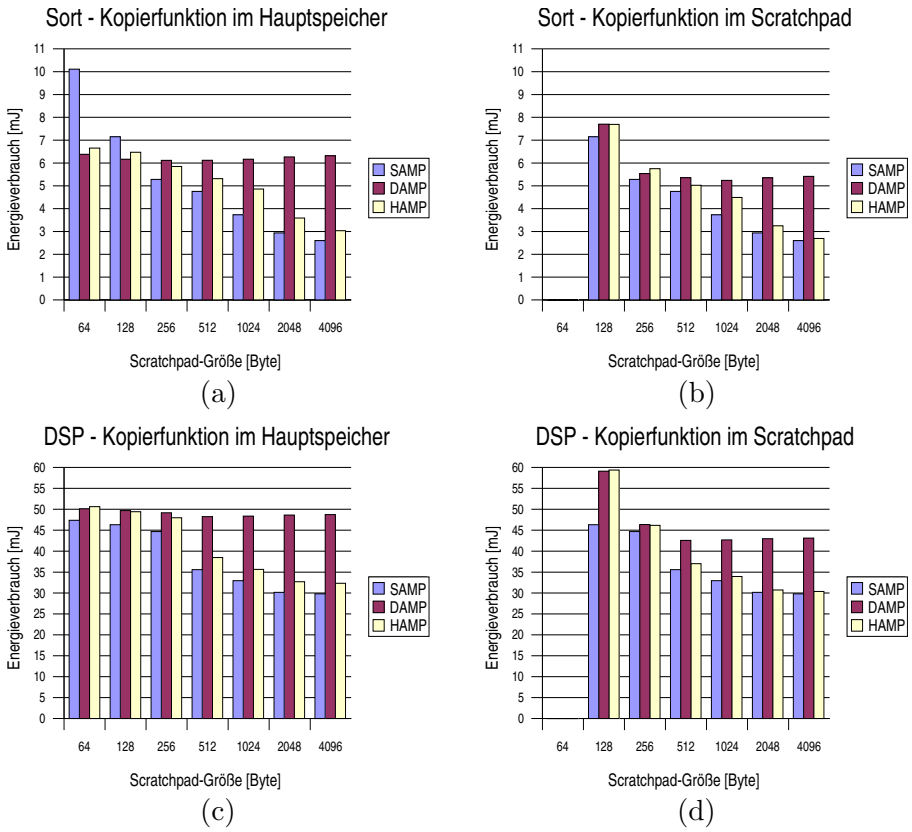

Abbildung 5.13: SAMP vs. DAMP vs. HAMP: **Sort** und **DSP** bei Zeitscheibenlänge von 3300 Zyklen

In der Abbildung 5.13 ist der Vergleich der Allokations-Strategien für eine 3300 Zyklen lange Zeitscheibe dargestellt. Es zeigt sich, daß die Kopierkosten so hoch sind, daß SAMP bereits bei kleinen Scratchpad-Größen bessere Ergebnisse als DAMP erzielt. Wird die Kopierfunktion ins Scratchpad verlagert (Abbildungen 5.13 (b) und (d)) dann sinken die Kopierkosten so weit, daß HAMP für große Speicher ähnlich gute Ergebnisse wie SAMP erzielt. Auch hier zeigt sich, daß HAMP teilweise schlechtere Ergebnisse als DAMP liefert. Würde zur Messung der erweiterten Energiefunktionen das in Anhang A formuliert HAOP-Problem gelöst, dann wird dieses Verhalten nicht mehr auftreten. Es ist weiterhin zu erwarten, daß dann HAMP auf den mittleren Speichergrößen bessere Ergebnisse erzielen wird, als dies momentan der Fall ist.

Kapitel 6

Zusammenfassung und Ausblick

Für eingebettete Systeme, denen häufig nur eine begrenzte Energiemenge zur Verfügung steht, ist eine Optimierung des Energieverbrauchs von entscheidender Bedeutung. Da die Weiterentwicklung von Akkus in den nächsten Jahren keine großen Verbesserungen erwarten läßt, treten neben Architekturverbesserungen Techniken zur Optimierung der Software immer mehr in den Vordergrund. Banakar et al. [BSL+02] haben gezeigt, daß der Einsatz von Scratchpad-Speichern in eingebetteten Systemen eine Alternative zu Caches darstellt. In der Folge wurden zahlreiche Verfahren entwickelt, um den Energieverbrauch oder die Ausführungszeit für einzelne Applikationen zu senken. Auf vielen Systemen laufen allerdings mehrere Prozesse parallel, deshalb wurden in dieser Arbeit drei Allokations-Strategien vorgestellt, mit denen eine Kostenfunktion eines Multiprozess-Systems optimiert werden kann.

Die ersten beiden Strategien, SAMP und DAMP, basieren auf unterschiedlichen Ansätzen. Die dritte Strategie, HAMP, versucht jeweils die Vorteile der beiden erstgenannten Strategien zu nutzen. SAMP weist jedem Prozess einen statischen Anteil am Scratchpad zu. Diese Allokations-Strategie erzielt besonders gute Ergebnisse, wenn eine größere Menge an Scratchpad-Speicher zur Verfügung steht. Auf kleinen Scratchpad-Speichern ist die Nutzbarkeit dieses Verfahrens beschränkt, da mit der Zuteilung von kleinen Anteilen des Scratchpads an die Prozesse nur geringe Energieeinsparungen erzielt werden können. Als Erweiterung bietet sich daher der Einsatz der Allokations-Strategie DAMP an: Den einzelnen Prozessen steht jeweils das ganze Scratchpad zur Verfügung, jedoch müssen der Code und die Daten, die ins Scratchpad ausgelagert wurden, bei jedem Prozesswechsel zwischen Hauptspeicher und Scratchpad hin und her kopiert werden. Der durch DAMP erreichte Nutzen wird durch die Kopierkosten und indirekt durch die Länge der Zeitscheibe des Systems eingeschränkt. Wird die Zeit-

scheibe verkürzt, dann werden die Prozesse immer öfter unterbrochen und dementsprechend öfter müssen Code und Daten bei den Prozesswechseln kopiert werden. Die Allokations-Strategie HAMP berechnet eine optimale Aufteilung des Scratchpad-Speichers in statische Anteile für die einzelnen Prozesse und einen gemeinsam genutzten dynamischen Anteil. Bei kleinen Scratchpad-Speichern verhält sich diese Strategie mehr wie DAMP, bei großen zunehmend wie SAMP. Auf großen Scratchpad-Speichern kann diese Allokations-Strategie allerdings nicht die gleichen Ergebnisse wie SAMP erzielen, da selbst dann Kosten durch die Kopierfunktion anfallen, wenn durch keinen Prozess der dynamische Anteil genutzt wird. Der durch dieses Verfahren erreichte Nutzen wird, ebenso wie bei DAMP, durch die Kopierkosten und die Länge der Zeitscheibe beschränkt. Ein weiteres Problem stellt die effiziente Gewinnung der Meßdaten dar. Gegenüber den ersten beiden Allokations-Strategien müssen für HAMP pro Prozess quadratisch viele Meßdaten bezogen auf die Scratchpad-Größe bestimmt werden. Es muß für jede möglich Aufteilung des Scratchpads in einen statischen und einen dynamischen Anteil das HAOP-Problem gelöst werden. Für diese Problemstellung ist nicht zu erwarten, daß ein echt polynomielles Approximationsschema oder ein Approximationsalgorithmus, der für alle Eingaben Lösungen berechnet, die nicht weiter als Faktor zwei von der optimalen Lösung entfernt sind, gefunden werden können.

Scratchpad-Größe	SAMP	DAMP	HAMP
64 Byte	0,46%	14,7%	14,71%
128 Byte	9,33%	18,22%	18,32%
256 Byte	15,69%	21,89%	21,96%
512 Byte	23,73%	30,78%	29,62%
1024 Byte	29,31%	31,73%	33,18%
2048 Byte	35,66%	31,58%	36,11%
4096 Byte	37,57%	30,54%	37,14%

Tabelle 6.1: Prozentuale Energieeinsparungen der Allokations-Strategien SAMP, DAMP und HAMP gegenüber SAOP

Die Allokations-Strategien wurden auf vier Multiprozess-Systeme angewendet und der resultierende Energieverbrauch gemessen. Tabelle 6.1 zeigt den für diese Systeme durchschnittlich erreichten Nutzen verglichen mit dem in Abschnitt 3.1 beschriebenen SAOP-Verfahren für die Scratchpad-Größen 64 Byte bis 4 kB. Der Vergleich mit einem System, das einen Cache verwendet, muß etwas differenzierter geführt werden. Für kleine Speichergrößen von 64, 128 und 256 Byte können Einsparungen zwischen 37% und 46% erzielt werden. Bei den größeren Speichern liegt der Energiebedarf beim Einsatz von SAMP, DAMP und HAMP deutlich über dem eines Caches. Dies läßt den Schluß zu, daß insbesondere bei kleinen Systemen Scratchpads und die vor-

gestellten Allokations-Strategien verwendet werden sollten. Die schlechteren Ergebnisse bei größeren Speichern haben ihre Ursache in dem zugrundeliegenden Optimierungsverfahren für die einzelnen Prozesse, das nicht in der Lage ist, Teile von Arrays in das Scratchpad auszulagern.

Der System-Designer, der ein eingebettetes System entwirft, wird durch die hier vorgestellten Allokations-Strategien durch ein automatisch ablaufendes Verfahren unterstützt. Alle Arbeitsschritte, von der Gewinnung der Meßdaten über die Optimierung der Kostenfunktionen und die Erstellung der Multiprozess-Binärdateien sind automatisiert. Der System-Designer kann entweder für eine gegebene Scratchpad-Größe diejenige Allokations-Strategie, die die beste Verteilung des Scratchpads unter den Prozessen bezüglich einer Kostenfunktion liefert, auswählen, oder aber die Optimierung für eine Menge von Scratchpad-Speichern durchführen und die beste Scratchpad-Größe für das System wählen.

Basierend auf den vorgestellten Allokations-Strategien bietet es sich an, folgende Fragestellungen und Erweiterungen zu untersuchen:

Verwendung weiterer Basis-Optimierungen: Für die Generierung der Ergebnisse wurde die statische Allokation von Code und Daten nach Steinke et al. [SWLM02] verwendet. Besonders beim Vergleich der Allokations-Strategien mit einem System, das einen Cache verwendet, zeigten sich die Einschränkungen dieser Optimierung. Verma et al. [VWM04b][VSM03] haben, wie bereits im Abschnitt 2.5 erwähnt, das dynamische Auslagern von Code und Daten ins Scratchpad sowie das Teilen von Arrays betrachtet. Durch Verwendung dieser oder weiterer Optimierungsmethoden für die einzelnen Prozesse sollte festgestellt werden, inwieweit die zugrundeliegende Optimierung Einfluß auf die Güte der Ergebnisse hat.

Erweiterung und Verbesserung der Algorithmen: Im Falle der Allokations-Strategien DAMP und HAMP bietet es sich an, vermehrt Code in den dynamischen Anteil auszulagern. Dieser muß nicht zurück in den Hauptspeicher kopiert werden, nachdem einem Prozess die Kontrolle über den Prozessor entzogen wurde. Für Code-Objekte reduzieren sich die Kopierkosten somit um die Hälfte gegenüber Daten-Objekten. Um diese Beobachtung auszunutzen, sind Veränderungen an der zugrundeliegenden Optimierung erforderlich.

Die Allokations-Strategien DAMP und HAMP gehen davon aus, daß einem Prozess die ganze Zeitscheibe zur Verfügung steht. Dies ist in realen System gewöhnlich nicht der Fall, dort konsumieren Dispatcher und Scheduler einen Teil der Zeitscheibe. Werden mehr Daten durch den Dispatcher kopiert, dann verringert sich die nutzbare Zeitscheibe, was zu einer erhöhten Laufzeit der Prozesse und einer höheren Anzahl von Unterbrechungen führt und damit die Kopierkosten steigen läßt.

Die Algorithmen für DAMP und HAMP sollten so erweitert werden, daß dieser Effekt mit berücksichtigt wird.

Bei der Auswertung der Ergebnisse in Abschnitt 5.4 zeigten sich die Grenzen des ILP-basierten Ansatz zum Lösen des HAOP-Problems. Hier ist es zum Messen der erweiterten Energiefunktionen notwendig die exaktere Formulierung aus Anhang A zu verwenden. Bis jetzt ist nur ein pseudopolynomieller Algorithmus zum Lösen dieses Problems bekannt. Hier sollten die Grenzen für Approximationsalgorithmen bestimmt und gute und schnelle Heuristiken gefunden werden.

Betrachtung weiterer Hardware: Poletti et al. [FMA+04] betrachten ein System mit einem DMA-Controller, um Daten dynamisch ins Scratchpad auszulagern. Die Kopierkosten für solch ein System sind wesentlich geringer als für einen Software-basierten Ansatz. Der Nutzen dieser Hardware für die Allokations-Strategien DAMP und HAMP kann durch Einstellen der Kopier-Kostenfunktionen ermittelt werden.

In der Diplomarbeit von Helmig [Hel04] werden Applikationen für mehrere Scratchpad-Speicher von unterschiedlicher Größe und Zugriffsenergie optimiert. Dieser Ansatz läßt sich auch auf Multiprozess-Systeme übertragen, wobei hier verschiedenste Konfigurationen möglich sind: Es können ein oder mehrere Scratchpad-Speicher die dynamischen Anteile aufnehmen. Die Speicher können jeweils einzelnen Prozessen zugeordnet oder unter den Prozessen geteilt werden. Weiterhin ist es möglich, die Scratchpad-Speicher in mehrere statische und einen dynamischen Anteil aufzuteilen. Hierbei ist zu beachten, daß mehr Freiheiten in der Aufteilung der Speicher auch einen deutlich gesteigerten Rechenaufwand erwarten lassen. Deshalb sollten für diese Problemstellungen Verfahren erarbeitet werden, die möglichst große Teile des Suchbereichs nicht betrachten müssen, aber trotzdem gute Näherungen berechnen können.

Anpassung an weitere Anforderungen: In den meisten Multiprozess-Systemen können den Prozessen unterschiedliche Prioritäten zugeordnet werden, wodurch Prozesse mit höherer Priorität häufiger die Kontrolle über den Prozessor erlangen als solche mit niedriger Priorität. Weitere Anforderungen sehen die Einhaltung von Zeitschranken vor. Für eine Menge von Prozessen muß deren Ausführung innerhalb einer festgelegten Zeitspanne sichergestellt werden. Die Betrachtung dieser Anforderungen stellt einen weiteren wichtigen Schritt zum Einsatz der Allokations-Strategien in Multiprozess-Systemen dar.

Anhang A

HAOP unter genauer Berücksichtigung der Kopierkosten

In Abschnitt 5.4 wurde motiviert, warum eine genauere Formulierung des HAOP-Problems notwendig ist. In den folgenden Abschnitten wird HAOP noch einmal definiert, wobei die entstehenden Kopierkosten diesmal exakt berücksichtigt werden. Weiterhin wird gezeigt, daß HAOP auch unter dieser Definition NP-vollständig ist und es wird ein pseudopolynomieller Algorithmus zur Lösung des Problems angegeben.

A.1 Definition HAOP

HAOP erhält als Eingabe n Objekte, denen Größenwerte $g_1, \ldots, g_n$ zugeordnet sind. Diese Objekte sollen auf ein statisches und ein dynamisches Scratchpad verteilt werden. Die Größe des statischen Scratchpads ist S_{stat}, die des dynamischen S_{dyn}. Wird Objekt i in einem der beiden Teile abgelegt, dann ist damit ein Nutzen von v_i verbunden. Bezeichnet D die Menge der Objekte, die im dynamischen Scratchpad abgelegt werden, dann gibt $x = \sum_{i \in D} g_i$ deren Gesamtgröße wieder und die Funktion $KK(x)$ liefert die entstehenden Kopierkosten. Der Gesamtnutzen ergibt sich durch Summation über alle Nutzenwerte der im statischen und dynamischen Scratchpad abgelegten Objekte abzüglich der entstehenden Kopierkosten. Die Entscheidungsvariante von HAOP ist nun das Problem, für einen gegebenen Nutzwert A zu entscheiden, ob es eine Verteilung der Objekte auf das statische und dynamische Scratchpad gibt, die die Größen dieser Scratchpads respektiert und mindestens Nutzen A erreicht.

A.2 Theoretische Diskussion (NP-Vollständigkeit)

Wie in Abschnitt 3.4.1.1 wird das PARTITION-Problem genutzt, um die NP-Vollständigkeit von HAOP zu zeigen.

Satz: HAOP ist NP-vollständig.

Beweis: HAOP $\in$ NP, da eine Verteilung der Objekte auf die Speicher geraten und in polynomieller Zeit verifiziert werden kann, ob der Nutzen A erreicht wird.

Sei $(a_1, \ldots, a_n)$ eine Eingabe für PARTITION. Hieraus wird in polynomieller Zeit folgende Eingabe für HAOP konstruiert. $g_i = a_i$, $v_i = 2a_i$, $A = \sum_i 2a_i - \lfloor (a_1 + \cdots + a_n)/2 \rfloor$, $S_{stat} = S_{dyn} = \lfloor (a_1 + \cdots + a_n)/2 \rfloor$ und $KK(x) = x$.

Hat PARTITION eine Lösung, dann können die Objekte wegen der Definition von S_{stat}, S_{dyn} und der g_i in das statische und das dynamische Scratchpad verteilt werden. Der erreichte Gesamtnutzen ist $\sum_i 2a_i - KK(\lfloor (a_1 + \cdots + a_n)/2 \rfloor) = \sum_i 2a_i - \lfloor (a_1 + \cdots + a_n)/2 \rfloor = A$.

Hat HAOP eine Lösung, dann muß der Gesamtnutzen der zur Lösungsmenge gehörenden Objekte mindestens $\sum_i 2a_i - \lfloor (a_1 + \cdots + a_n)/2 \rfloor$ betragen, was wiederum bedeutet, daß alle Objekte in einem der beiden Scratchpad-Speicher plaziert wurden. Die Wahl der Größen dieser beiden Speicher sichert, daß auch PARTITION eine Lösung hat. □

A.3 Pseudopolynomieller Algorithmus für HAOP

Die Eingabe für HAOP enthält die Größen der Objekte $g_1, \ldots, g_n$, die Nutzenwerte $v_1, \ldots, v_n$, die Kopierkostenfunktion $KK(x)$ und die Größen der beiden Speicher S_{stat} und S_{dyn}.

Die Einträge des Arrays $N(k, s_1, s_2)$ seien leer, ebenso werden die nicht vorhandenen Einträge $N(k, s_1, s_2)$ mit $s_1 < 0 \ \vee \ s_2 < 0$ als leer definiert. $N(1, g_1, 0) := v_1$ und $N(1, 0, g_1) := v_1 - KK(g_1)$.

Um einen Eintrag $N(k, s_1, s_2)$ zu füllen, sind acht Fälle zu unterscheiden:

1. $N(k-1, s_1, s_2)$, $N(k-1, s_1 - g_k, s_2)$ und $N(k-1, s_1, s_2 - g_k)$ sind leer, dann ist auch $N(k, s_1, s_2)$ leer.

2. Nur $N(k-1, s_1, s_2)$ ist nicht leer, dann ist $N(k, s_1, s_2) := N(k-1, s_1, s_2)$.

3. Nur $N(k-1, s_1 - g_k, s_2)$ ist nicht leer, dann ist $N(k, s_1, s_2) := N(k-1, s_1 - g_k, s_2) + v_k$.

4. Nur $N(k-1, s_1, s_2 - g_k)$ ist nicht leer, dann ist $N(k, s_1, s_2) := N(k-1, s_1, s_2 - g_k) + v_k - KK(s_2) + KK(s_2 - g_k)$. Für den Eintrag $N(k-1, s_1, s_2 - g_k)$ wurden bereits die Kopierkosten $KK(s_2 - g_k)$ gezählt.

Da hier aber Kopierkosten von $KK(s_2)$ anfallen, müssen die vorher gezählten Kosten wieder addiert werden.

5. $N(k-1, s_1, s_2)$ und $N(k-1, s_1 - g_k, s_2)$ sind nicht leer, dann ist $N(k, s_1, s_2) := max\{N(k-1, s_1, s_2), N(k-1, s_1 - g_k, s_2) + v_k\}$.

6. $N(k-1, s_1, s_2)$ und $N(k-1, s_1, s_2 - g_k)$ sind nicht leer, dann ist $N(k, s_1, s_2) := max\{N(k-1, s_1, s_2), N(k-1, s_1, s_2 - g_k) + v_k - KK(s_2) + KK(s_2 - g_k)\}$.

7. $N(k-1, s_1 - g_k, s_2)$ und $N(k-1, s_1, s_2 - g_k)$ sind nicht leer, dann ist $N(k, s_1, s_2) := max\{N(k-1, s_1 - g_k, s_2) + v_k, N(k-1, s_1, s_2 - g_k) + v_k - KK(s_2) + KK(s_2 - g_k)\}$.

8. Alle drei Einträge sind nicht leer:

$$\begin{aligned} N(k, s_1, s_2) \quad = \quad & max\{N(k-1, s_1, s_2), N(k-1, s_1 - g_k, s_2) + v_k, \\ & N(k-1, s_1, s_2 - g_k) + v_k - KK(s_2) + \\ & KK(s_2 - g_k)\} \end{aligned}$$

Literaturverzeichnis

[ABC03] Angiolini, F., L. Benini und A. Caprara: *Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning*. In: *Proc. of the Workshop on Compiler and Architectural Support for Embedded Computer Systems CASES'03*. ACM, Oktober 2003.

[ABR03] Acquaviva, A., L. Benini und B. Ricco: *Energy characterization of embedded real-time operating systems*. Kluwer Academic Publishers, 2003.

[ARM01a] Advanced RISC Machines Ltd (ARM): *ARM7TDMI*, September 2001. Dokument-Nr.: ARM DDI 0210B.

[ARM01b] Advanced RISC Machines Ltd (ARM): *The ARMulator*, August 2001. Dokument-Nr.: ARM DAI 0032E.

[BSL+02] Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan und P. Marwedel: *Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems*. In: *Proc. of 10th International Symposium on Hardware/Software Codesign*, Colorado, USA, Mai 2002.

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

[FMA+04] Francesco, P., P. Marchal, D. Atienza, L. Benini, F. Catthoor und J. M. Mendias: *An Integrated Hardware/Software Approach For Run-Time Scratchpad Management*. In: *Proc. of the 41st annual conference on Design automation*, San Diego, CA, USA, Juni 2004.

[GMM99] Ghosh, S., M. Martonosi und S. Malik: *Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior*. In: *ACM Transactions on Progamming Languages and Systems*. ACM, Juli 1999.

[Hel04] Helmig, U.: *Compilergestützte Optimierung von Zugriffen auf partitionierte Speicher*. Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 12, 2004.

[ILO] ILOG: *ILOG CPLEX*. http://www.ilog.com/products/cplex.

[KC02] Kandemir, M. und A. Choudhary: *Compiler-Directed Scratch Pad Memory Hierarchy Design and Management*. In: *Proc. of 39th Design Automation Conference*, New Orleans, Louisiana, USA, Juni 2002. ACM.

[KG02] Krishnaswamy, A. und R. Gupta: *Profile Guided Selection of ARM and Thumb Instructions*. In: *Proc. of LCTES - SCOPES 02, Berlin, Germany*, Juni 2002.

[KIVK00] Kim, H. S., M. J. Irwin, N. Vijaykrishnan und M. Kandemir: *Effect of Compiler Optimizations on Memory Energy*. In: *Workshop on Signal Processing Systems, Lousiana*, Oktober 2000.

[KK02] Kandemir, M. und I. Kolcu: *Influence of Loop Optimizations on Energy Consumption of Multi-Bank Memory Systems*. In: *Proc. of the 11th International Conference on Compiler Construction*, 2002.

[KPL99] Kwon, Y.-J., D. Parker und H. J. Lee: *TOE: Instruction Set Architecture for Code Size Reduction and Two Operations Execution*. In: *International Workshop on Compiler and Architecture Support for Embedded Systems*, Oktober 1999.

[KPP04] Kellerer, H., U. Pferschy und D. Pisinger: *Knapsack Problems*. Springer, 2004.

[KRI+01] Kandemir, M., J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif und A. Parikh: *Dynamic Management of Scratch-Pad Memory Space*. In: *Proc. of 38th Design Automation Conference*, Las Vegas, Nevada, USA, Juni 2001. ACM.

[LLHT00] Lee, C., J. Lee, T. Hwang und S. Tsai: *Compiler Optimization on Instruction Scheduling for Low Power*. In: *13th International Symposium on System Synthesis*. ACM, September 2000.

[Mar03] Marwedel, P.: *Embedded System Design*. Kluwer Academic Publishers, 2003.

[OKD+04] Ozturk, O., M. Kandemir, I. Demirkiran, G. Chen und M. J. Irwin: *Data Compression for Improving SPM Behavior*. In: *In Proc. the 41st Design Automation Conference (DAC'04)*, San Diego, CA, USA, Juni 2004.

[SBM99] Simunic, T., L. Benini und G. De Micheli: *Cycle-Accurate Simulation of Energy Consumption in Embedded Systems.* In: *In Proc. the 36th Design Automation Conference (DAC'99)*, New Orleans, Louisiana, USA, Juni 1999.

[SGW+02] Steinke, S., N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan und P. Marwedel: *Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory.* In: *Proc. of the 15th International Symposium on System Synthesis (ISSS)*, Kyoto Japan, Oktober 2002.

[SKWM01] Steinke, S., M. Knauer, L. Wehmeyer und P. Marwedel: *An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations.* In: *Proc. of International Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, Yverdon-Les-Bains, Switzerland, September 2001.

[SWLM02] Steinke, S., L. Wehmeyer, B.-S. Lee und P. Marwedel: *Assigning Program and Data Objects to Scratchpad for Energy Reduction.* In: *Proc. of Design Automation and Test in Europe (DATE)*, Paris Frankreich, März 2002.

[Syn96] Synopsis Inc.: *Power Products Reference Manual.* 1996.

[Tan92] Tanenbaum, A. S.: *Modern Operating Systems.* Prentice-Hall International Editions. Prentice-Hall, International, 1992.

[The00] Theokharidis, M.: *Energiemessung von ARM7TDMI Prozessor-Instruktionen.* Diplomarbeit, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 12, 2000.

[TL98] Tiwari, V. und M. T.-C. Lee: *Power Analysis of a 32-bit Embedded Microcontroller.* VLSI Design Journal, 7(3), 1998.

[TMW94] Tiwari, V., S. Malik und A. Wolfe: *Power Analysis of Embedded Software: A First Step towards Software Power Minimization.* IEEE Trans. On VLSI Systems, 2(4), Dezember 1994.

[TY96] Tomiyama, H. und H. Yasuura: *Optimal Code Placement of Embedded Software for Instruction Caches.* In: *Proc. of the 9th European Design and Test Conference*, Paris France, März 1996. ET&TC.

[VJ01] Valluri, M. und L. K. John: *Is Compiling for Performance == Compiling for Power?* In: *Proc. of the 5th Annual Workshop on Interaction between Compilers and Computer Architectures*, 2001.

[VSM03] Verma, M., S. Steinke und P. Marwedel: *Data Partitioning for Maximal Scratchpad Usage.* In: *Proc. of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2003.

[VWM04a] Verma, M., L. Wehmeyer und P. Marwedel: *Cache-aware Scratchpad Allocation Algorihm.* In: *Proc. of Design, Automation and Test in Europe (DATE)*, Februar 2004.

[VWM04b] Verma, M., L. Wehmeyer und P. Marwedel: *Dynamic Overlay of Scratchpad Memory for Energy Minimization.* In: *International Conference on Hardware/Software Codesign and System Synthesis*, Stockholm Schweden, September 2004.

[Weg99] Wegener, I.: *Theoretische Informatik - eine algorithmische Einführung.* Leitfäden der Informatik. B. G. Teubner, 1999.

[Weg01] Wegener, I.: *Stammvorlesung Effiziente Algorithmen.* Vorlesungsskript, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 12, 2001.

[WJ96] Wilton, S. J. E. und N. P. Jouppi: *CACTI: An Enhanced Cache Access and Cycle Time Model.* IEEE Journal of Solid-State Circuits, 31(5), Mai 1996.