Diplomarbeit:
Compilergestützte Energiereduktion
von SDRAM- und Flash-basierten Speichertechnologien

André Kernchen

19. Januar 2005

Inhaltsverzeichnis

Kapitel 1

Einleitung

Zu einer der wichtigsten Eigenschaften eines mobilen eingebetteten Systems zählt heutzutage, nebst kleinen Abmessungen und hoher Zuverlässigkeit, ein geringer Energieverbrauch, der erheblichen Einfluss auf die Batterie-Betriebsdauer hat. Systemkomponenten wie Prozessor und Speicher gehören zu den typischen Verbrauchern des Gesamtsystems, so dass es wichtig ist, die Leistungsaufnahme so zu regulieren, dass die Verlustleistung insgesamt möglichst gering ist. Wird weniger Energie für Prozessor und Speicher benötigt, können andere Merkmale wie Farbdisplay oder Hintergrundbeleuchtung, die sich zunehmend zum Standard in tragbaren Geräten entwickeln, davon profitieren.

In der Vergangenheit reduzierte man die Leistungsaufnahme in eingebetteten Systemen mit Hilfe einer Reihe von Power-Management-Techniken, die im wesentlichen auf die längeren Leerlauf-Phasen des Systems zugeschnitten waren. Inzwischen gehören anspruchsvollere und rechenintensivere Applikationen aus dem Multimedia-Bereich, die über beträchtliche Zeitspannen hinweg aktiv sind, zum typischen Anwendungsbereich. Dies hat zur Folge, dass das zeitliche Verhältnis zwischen aktivem Betrieb und Leerlauf-Phasen immer größer wird. Während konventionelle Power-Management-Techniken in Zeiten des Leerlaufs äußerst effektiv sind, tragen sie nichts zur Senkung des Verbrauchs während des regulären Betriebs bei.

Eine Möglichkeit, den Forderungen nach hoher Rechenleistung und geringer Leistungsaufnahme gerecht zu werden, besteht darin, den Prozessor je nach seinem Arbeitsauf-

kommen mit unterschiedlicher Leistung zu betreiben. Zum Beispiel erfordert die Videokodierung eine deutlich höhere Leistung als die Audiowiedergabe, so dass der Prozessor mit deutlich niedrigerer Taktfrequenz arbeiten könnte, ohne das Ergebnis der Wiedergabe zu beeinflussen. Folglich wäre ein möglicher Ansatz, die Versorgungsspannung des Prozessors bei einer Absenkung der Taktfrequenz zu reduzieren, um die Leistungsaufnahme zu senken.

Neben dem Prozessor gehören aber auch Speicher zu den Verbrauchern eines Systems, bei denen ein gewisses Potential zur Senkung der Leistungsaufnahme vorhanden ist. Die Verwendung von energiesparenden Speicherlösungen ermöglicht dem Entwickler die Bereitstellung neuer Funktionalität, ohne die Leistungsfähigkeit zu beeinträchtigen.

Die bei einigen Speichern vorhandenen Leistungsmerkmale *Temperature-Compensated-Self-Refresh-Sensors, Partial-Array-Self-Refresh, Power-Down- und Deep-Power-Down-Mode* bieten Entwicklern die Möglichkeit, mit dem Verbrauch besser hauszuhalten und so die Batteriebetriebsdauer in *handheld applications* wie SmartPhones, MP3-Playern und PDAs deutlich zu verlängern.

1.1 Inhalt der Arbeit

Der Inhalt dieser Arbeit setzt sich aus mehreren Komponenten zusammen. Grundlegend wird zunächst eine Recherche moderner Speichertechnologien vorgenommen, die aufgrund ihrer Eigenschaften eine besondere Eignung für den Einsatz in mobilen eingebetteten Systemen vorweisen.

Eine wesentliche Anforderung dieser Arbeit stellt die Aufstellung eines Energiemodells zur Bewertung des Energieverbrauchs dynamischer Speicher und dessen Integration in die vorhandene Arbeitsumgebung dar.

Die Untersuchung von Flash- und SDRAM-basierten Speichertechnologien bilden den Hauptteil dieser Arbeit. Im diesem Rahmen werden Optimierungen beschrieben sowie die damit erzielten Ergebnisse vorgestellt und diskutiert. Mögliche Erweiterungsvor-

schläge schließen diese Arbeit ab.

1.2 Aufbau der Arbeit

Im folgenden wird nun ein Überblick der Arbeit erstellt und der Inhalt der einzelnen Kapitel kurz skizziert. Die vorliegende Arbeit gliedert sich wie folgt:

- **Kapitel 2 - Grundlagen**
  In diesem Kapitel werden die grundlegenden Thematiken behandelt, die ein allgemeines Verständnis dieser Arbeit erleichtern. Desweiteren werden Begrifflichkeiten definiert, die in den folgenden Kapiteln vorausgesetzt werden.
  Anfänglich wird der physikalische Begriff der Energie und die Messung des Verbrauchs von Energie erläutert. Da diese Arbeit eine Reduktion des Energieverbrauchs durch Speicher anstrebt, werden im Anschluss verschiedene Speichertechnologien vorgestellt und in ihrer Funktionsweise diskutiert. Die eigentliche Aufgabe der Energiereduktion wird compilergesteuert vorgenommen, so dass konsequenterweise eine grundlegende Einführung in die Funktionsweise von Compilern geliefert wird.

- **Kaptel 3 - Verwandte Arbeiten**
  Die compilergesteuerte Energiereduktion stellt ein großes Forschungsgebiet im Bereich der eingebetteten System dar. Die Arbeiten, die sich mit dieser Thematik beschäftigen, bilden den Inhalt dieses Kapitels. Insbesondere die am Lehrstuhl XII entstandenen Forschungsarbeiten, die zu Teilen auch Grundlage für die vorliegende Arbeit bilden, werden hier vorgestellt.

- **Kapitel 4 - Low-Power Speicher in SDRAM-Technologie**
  Die zuvor in Kapitel 2 grundlegend erläuterten Speicher werden nun anhand ihrer speziellen Funktionalität für den Einsatz in eingebetteten Systemen untersucht. Hierzu wird ein Energiemodell aufgestellt, welches zur Analyse des Energieverbrauchs solcher Speicher unabdingbar ist. Dieses Energiemodell wird anschließend in die bestehende Arbeitsumgebung integriert, wodurch eine zyklengenaue Messung des Energieverbrauchs möglich wird. In diesem Kapitel werden dann auch die notwendigen Anpassungen der Umgebung detailliert beschrieben.

Abschluss dieses Kapitels bildet eine Optimierungsmethode und deren Formalisierung, die auf die Funktionalität obengenannter Speicher in Kombination mit Scratchpad-Speichern abzielt.

- **Kapitel 5 - SDRAM- und Flash-basierte Speichertechnologien**
  Eine weitere Anforderung an diese Arbeit ist die Integration einer im betrachteten System vorhandenen Speicherkomponente, die in der bisherigen Arbeitsumgebung unberücksichtigt blieb. Die Kombination von Flash-Speichern mit den zuvor beschriebenen SDRAM-Speichern rundet diese Arbeit ab. Die Ausnutzung der Vorteile beider Speichertechnologien werden untersucht und durch eine compilergesteuerte Optimierung demonstriert. Insbesondere die Formalisierung der Optimierungsproblematik bildet den Schwerpunkt dieses Kapitels.

- **Kapitel 6 - Ergebnisse**
  Die erzielten Ergebnisse dieser Arbeit werden grafisch präsentiert und ausgewertet. Hierzu wurde eine Reihe von Testprogrammen mit dem Compiler übersetzt und hinsichtlich ihres Energieverbrauchs bei verschiedenen Speicherkonfigurationen analysiert.

- **Kapitel 7 - Zusammenfassung und Ausblick**
  In diesem Kapitel werden die Inhalte dieser Arbeit rekapituliert und die Ergebnisse zusammengefasst. Abgeschlossen wird die Zusammenfassung mit einem Ausblick über mögliche Erweiterungen dieser Arbeit.

Kapitel 2

Grundlagen

Das folgende Kapitel befasst sich mit dem physikalischen Begriff der Energie sowie mit verschiedenen Ansatzmöglichkeiten, eine Reduktion des Energieverbrauchs zu erzielen. Zu den typischen Energieverbrauchern zählen die Speicher eines Systems, so dass im Anschluss die heute gängigen Speichertechnologien vorgestellt werden. Den Abschluss dieses Kapitels bildet eine allgemeine Einführung in den Aufbau von Compilern.

2.1 Leistung und Energie

Für die Beschreibung des Energieverbrauchs wird zunächst der Begriff der Leistung P eingeführt. Die Leistung P basiert auf der Spannung U und dem Strom I.
Berücksichtigt man den zeitlichen Verlauf von Strom und Spannung, so erhält man für den Augenblickswert der Leistung

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

Die Energie, als elektrische Arbeit bezeichnet, ergibt sich aus dem Integral der Leistung über eine gewisse Zeit t. Im allgemeinen berechnet sich die Energie E, die durch einen elektrischen Verbraucher in dem zeitlichen Intervall $[t_0, t_1]$ umgewandelt wird, durch

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

Werden Spannung und Strom als konstant angenommen oder über die Zeit gemittelt, erhält man

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

2.1.1 Leistungsverbrauch in CMOS-Schaltungen

Beim Leistungsverbrauch in CMOS-Schaltungen unterscheidet man für gewöhnlich statische und dynamische Verlustleistung. Die dynamische Verlustleistung ergibt sich durch Zustandsänderungen, die durch Schaltvorgänge hervorgerufen werden.

- Durch das Laden und Entladen der Lastkapazitäten am Ausgang der CMOS-Schaltung fließen Schaltströme, die den maßgeblichen Anteil der dynamischen Verlustleistung ausmachen. Diese Leistung wird im folgenden als *switching power* (P_{SW}) bezeichnet.
- Während des Wechsels des Eingangssignals tritt kurzzeitig eine überlappte Leitfähigkeit der n-MOS und p-MOS-Transistoren auf, so dass ein Kurzschlusstrom fließt. Die hiermit verbundene Verlustleistung wird mit *shortcircuit power* (P_{SC}) angegeben.

Der statische Leistungsverbrauch ist unabhängig von den Schaltvorgängen und wird ausschließlich durch unerwünschte, kontinuierliche Leckströme hervorgerufen. Proportional zur Chipfläche und Temperatur entsteht hierdurch der statische Leistungsverbrauch *leakage power* (P_{LK}). Durch Addition der statischen und dynamischen Verlustleistung erhält man

$$P_{TOT} = P_{SW} + P_{SC} + P_{LK} \tag{2.4}$$

und mit Gl. 2.3 den Energieverbrauch

$$E_{TOT} = P_{TOT} \cdot t \tag{2.5}$$

2.1.2 Leistungs- und Energiereduktion

Anhand der bisherigen Feststellungen lässt sich erkennen, dass für den Energieverbrauch einer CMOS-Schaltung die jeweils anfallende Leistung über die Zeit verant-

wortlich ist. Folglich lassen sich Energiereduktionen durch Verringerung der Leistung und/oder der Zeit, über die diese Leistung anfällt, erzielen. Die folgende Auflistung stellt auszugsweise Ansatzmöglichkeiten für die Energiereduktion auf verschiedenen Entwicklungsebenen (Hard- und Software) dar.

- **Hardware-Ebene:**
  Anhand der zuvor aufgestellten Gleichungen stellt man fest, dass durch Verringerung von Betriebsspannung, Stromstärken und Lastkapazitäten eine Reduktion der Verlustleistung bereits auf Schaltungsebene realisierbar ist. Der Entwurf von Technologien mit geringer Leistungsaufnahme wird häufig als *low-power design* bezeichnet.

- **Software-Ebene:**
  Auf Software-Ebene bestehen ebenfalls Möglichkeiten für den Entwickler, Einfluss auf den Energieverbrauch zu nehmen. Besonders beim Entwurf von optimierenden Compilern sollten folgende Entwurfsanforderungen berücksichtigt werden:

  - Erzeugung von Code mit wenig Schaltaktivität, um die zuvor beschriebene dynamische Verlustleistung klein zu halten
  - Erzeugung von schnellem Code, um direkt auf die zeitliche Komponente Einfluss zu nehmen
  - Erzeugung von kompaktem Code, um Speicherplatz zu sparen, denn der Zugriff auf größere Speicher verursacht einen größeren Energieverbrauch
  - Reduktion von Speicherzugriffen, da Speicherzugriffe entsprechende Schaltaktivitäten auf hochkapazitiven Busleitungen verursachen
  - Ausnutzung heterogener Speicher durch optimale Code- und Datenverteilung
  - Ausnutzung von *power managment*, dazu zählen die Funktionalitäten von modernen Speichern

Die unvollständige Aufzählung verdeutlicht das Potential der compilergesteuerten Energiereduktion, das sich durch Code- und Speicheroptimierungen ergibt.

2.2 Speichertechnologien

Im allgemeinen werden Speicher anhand ihrer Zugriffsgeschwindigkeit, Kapazität und Kosten beurteilt, wobei beim Einsatz in eingebetteten Systemen (ES) zusätzlich ein geringer Leistungsbedarf gefordert wird.
Hierbei stellen die Halbleiterspeicher gegenwärtig die Gruppe der gängigsten Speicher dar. Technologisch unterscheiden sich diese in folgende Gruppen [Sch04]:

- MOS (CMOS)
- Bipolar (Si,GaAs)
- BiCMOS

Die aufwendig herzustellenden BiCMOS-Schaltungen vereinigen Vorteile der beiden anderen Schaltungen:

- hohe Zugriffsgeschwindigkeit (Bipolar)
- hohe Schaltungsdichte (CMOS)
- geringer Leistungsbedarf (CMOS)

Allgemein sind alle Speicher ähnlich aufgebaut: die Anordnung der einzelnen Zellen ergibt eine Speichermatrix, auf die zeilen- und spaltenweise über einen Adressdecoder zugegriffen wird. Im folgenden werden nun zunächst die flüchtigen, anschließend die nichtflüchtigen Speicher genauer erläutert. Diese Erläuterungen basieren auf [Mar03] und [Bäh02] und werden an passender Stelle ergänzt.

2.2.1 Flüchtige Speicher

Die Zellen flüchtiger Speicher verlieren nach dem Ausschalten der Betriebsspannung ihren Informationsgehalt. Da alle Zellen uneingeschränkt adressiert werden können,

spricht man auch von *random access memories* (RAM), eine weitere Unterteilung in **statische RAM** (SRAM) und **dynamische RAM** (DRAM) ergibt sich durch die unterschiedlichen Mechanismen beim Zugriff auf einen solchen Speicher.

Statisches RAM

SRAM findet aufgrund seiner geringen Zugriffszeit von bis zu 20 Nanosekunden (ns) vorwiegend Verwendung als *on-chip* Speicher (*cache, scratch-pad*). Die Zugriffszeiten variieren je nachdem, ob der Speicher *on-chip* oder *off-chip* in das System integriert wird. Eine SRAM-Zelle basiert auf dem Flip-Flop-Konzept und wird aus 6 Transistoren aufgebaut.

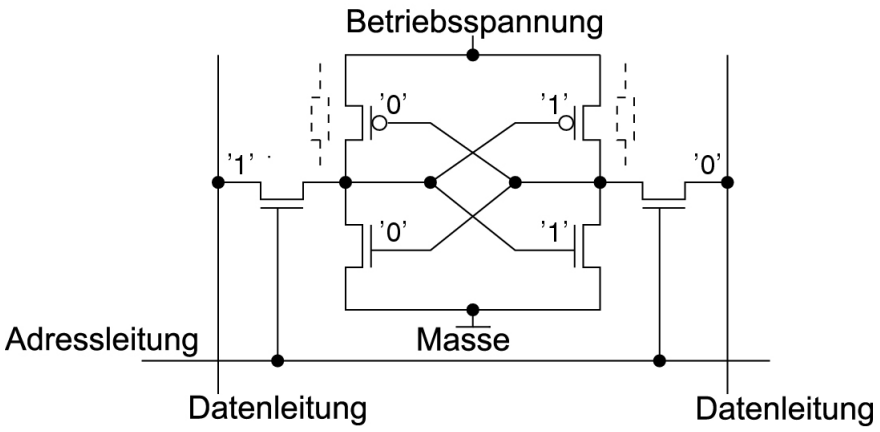


Abbildung 2.1: Aufbau einer SRAM-Zelle [Mar03]

Adressiert wird eine SRAM-Zelle über eine Adressleitung und zwei komplementäre Datenleitungen (s. Abb 2.1). Wird die Adressleitung auf '1' gesetzt, schalten zwei Transistoren und stellen damit eine Verbindung der Speicherzelle mit den beiden Datenleitungen her. Über eine zusätzliche Steuerleitung wird mit dem Steuersignal *write enable* (WE) angegeben, ob es sich um einen schreibenden oder einen lesenden Zugriff handelt. Beim Schreibzugriff werden die ausgewählten Datenleitungen mit den beiden komplementären Werten des Eingangs verbunden. Der resultierende Speicherinhalt der Zelle ist somit stabil und behält auch dann seinen Inhalt, wenn die Adressleitung auf '0' gesetzt wird. Beim Lesezugriff werden die Datenleitungen mit einem Differenzverstärker

am Ausgang verbunden. Dieser stellt fest, welche der beiden Datenleitungen das höhere Potential hat und erzeugt den eigentlichen Ausgabewert.

Eine Vielzahl architektonischer Veränderungen der SRAM wurden bisher vorgeschlagen, von denen einige Realisierungen im folgenden erläutert werden.

Die Aufteilung der E/A-Anschlüsse in zwei voneinander getrennte Anschlüsse (*dual-port*) erlaubt schnellere Speicherzugriffe, da Lese- und Schreibzugriffe auf unterschiedliche Zellen gleichzeitig stattfinden können. Über ein zusätzliches Steuersignal *output enable* (OE) können die Ausgänge des SRAM geschaltet werden. Dadurch wird ein überlappender Zugriff ermöglicht, da die Datenleitungen nicht während der Adressbearbeitung belegt werden.

Synchrone SRAM (SSRAM) enthalten zusätzliche, mit dem Prozessortakt verbundene Adress-und Datenregister, die einen Durchsatz steigernden Pipelinebetrieb im SRAM ermöglichen. SSRAM, die im *burst mode* betrieben werden, inkrementieren die aktuell angelegte Adresse automatisch. Hierdurch entfällt der zusätzliche Aufwand für die Übertragung der nächsten Adresse und die damit verbundene Busbelastung.

Gegenwärtig lässt sich ein deutlicher Trend zu **low-power SRAM** mit niedrigen Betriebsspannungen erkennen, wobei ein Standard von 1,8 Volt angestrebt wird. Zukünftige Lösungen zielen auf eine weitere Reduzierung von 50% ab.
Ein Nachteil konventioneller SRAM war bisher der große Flächenbedarf. Durch eine neue Technologie, die SRAM-Zellen unter Verwendung eines *thin film* Transistor (TFT) und aus der DRAM-Technologie stammenden *stacked capacitor* verbindet, strebt die Halbleiterindustrie eine Flächenreduktion von 50% auf 30 mm^2 an.

Embedded SRAM

Speicher in SRAM-Technologie werden zumeist als Cache oder Scratchpad-Speicher verwendet. Unabhängig von ihrer Verwendung werden SRAM, sofern sie in den Prozessor-Chip eingebettet sind, als **embedded SRAM** (eSRAM) bezeichnet.

Der *on-Chip* oder kurz L1-Cache, ist ein relativ kleiner eSRAM, der häufig benötigte Daten und Instruktionen zwischenspeichert. Zur Laufzeit werden dann durch eine aufwendige, im Cache integrierte Logik, auszugsweise diejenigen Hauptspeicherinhalte im Cache eingelagert, die mit hoher Wahrscheinlichkeit von der CPU angefordert werden (örtliche und zeitliche Lokalität). Eine ausführliche Beschreibung von Cache-Speichern wird in [HP96] durchgeführt.

Ein neuer Trend im Bereich der ES sieht eine Verwendung von eSRAM ohne die zuvor erwähnte aufwendige Verwaltungslogik vor. Als *Scratchpad* (SP) bezeichnet, stellen diese Speicher aus Sicht des Prozessors einen frei adressierbaren Speicher dar, der ebenfalls einen schnellen und vor allem energiesparsamen Zugriff garantiert [BSL+01]. Da allerdings nur speziell angepasste Programme von diesem Speicher profitieren, spricht man häufig auch von einem *compiler-controlled memory* (CCM). Die Speicherinhalte eines Scratchpad-Speichers werden vom Compiler bestimmt und sind nicht ohne erheblichen Aufwand zur Laufzeit veränderbar. Folglich sollte der Compiler bereits zur Übersetzungszeit bestimmen, welche Instruktionen oder Daten mit großer Wahrscheinlichkeit häufig von der CPU angefordert werden.

Dynamisches RAM

DRAM wird heute in der Regel als Hauptspeicher eingesetzt. Die DRAM-Zelle besteht aus lediglich einem Transistor und einem Kondensator und kann gegenüber der SRAM-Zelle auf etwa einem Zehntel der Fläche realisiert werden.

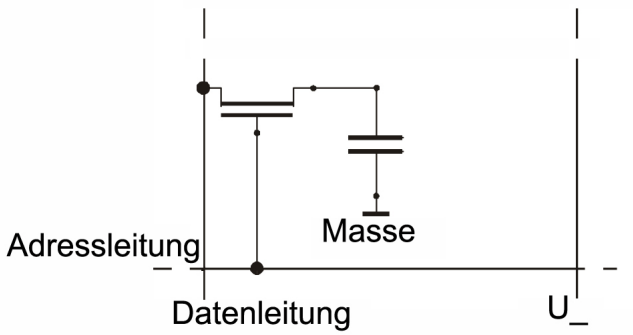


Abbildung 2.2: Aufbau einer DRAM-Zelle

Für die Adressierung eines DRAM wird eine Zeitmultiplexsteuerung verwendet. Diese Steuerung erfolgt über die zwei Steuersignale *row address strobe* (RAS) und *column address strobe* (CAS). Abhängig von der verwendeten Technologie ergeben sich folgende wichtige Kenngrößen:

- CAS Latency (CL)
- RAS-to-CAS Delay
- RAS Precharge Delay

Die erstaufgeführte Verzögerung CAS Latency gibt die Zeitspanne in Speichertaktzyklen an, die bis zur Bereitstellung der Daten am Ausgang des Speichers benötigt werden. Die Kenngröße RAS-to-CAS Delay beschreibt die Verzögerung in Speichertaktzyklen, die das auf RAS folgende Steuersignal CAS angelegt sein muss, bevor die Spaltenadresse als gültig angenommen werden kann. Der RAS Precharge Delay als letztgenannte Kenngröße gibt die Anzahl der Speichertaktzyklen an, die der Speicher nach dem Auslesen der Speicherzelle warten muss, bevor die nächste Zeilenadresse angelegt werden kann. Mit dieser Verzögerung verbunden sind das notwendige Zurückschreiben des Datums in die Speicherzelle und das Vorladen der Datenleitungen (*precharge*).

Beim Schreibvorgang wird ein entsprechendes Potential an die gewünschte Datenleitung gelegt und somit der Transistor der Zelle geschaltet. Da allerdings über eine Adressleitung alle Zellen einer Zeile aktiviert werden, ist das Schreiben einer isolierten Zelle nicht möglich. Somit erfordert ein Schreibvorgang zunächst stets das Lesen, das Ändern der entsprechenden Werte und anschließend das Rückschreiben der Speicherinhalte.

Erschwert wird das Lesen beim DRAM durch die Kapazität der Datenleitung. Diese ist in Abhängigkeit von der Länge der Leitung zumeist wesentlich größer als die des Kondensators in der DRAMZelle. Der Lesevorgang läuft daher wie folgt ab: Zunächst wird die Zeilenadressierung vorgenommen. Die entsprechenden Kondensatorladungen dieser Zeile werden auf die Datenleitungen übertragen und mit einem Eingang des zugehörigen Verstärkers verbunden. Die zweiten Eingänge der Verstärker werden auf eine Spannung U_{-} gesetzt, die zwischen der einer anliegenden '0' und einer anliegenden '1' liegt. Nachdem die Differenzverstärker die Inhalte der Zellen durch Auswertung

der Signale auf den jeweiligen Datenleitungen durch Vergleich mit U_ bestimmt haben, wird mit der Spaltenadressierung die Auswahl der gewünschten Spalte und somit des gewünschten Verstärkers vorgenommen. Da durch den Lesevorgang die Zelleninhalte der Zeile verloren gehen (zerstörendes Lesen), wird im Anschluss die gesamte Zeile zurückgeschrieben und die Datenleitungen für den nächsten Zugriff vorgeladen.

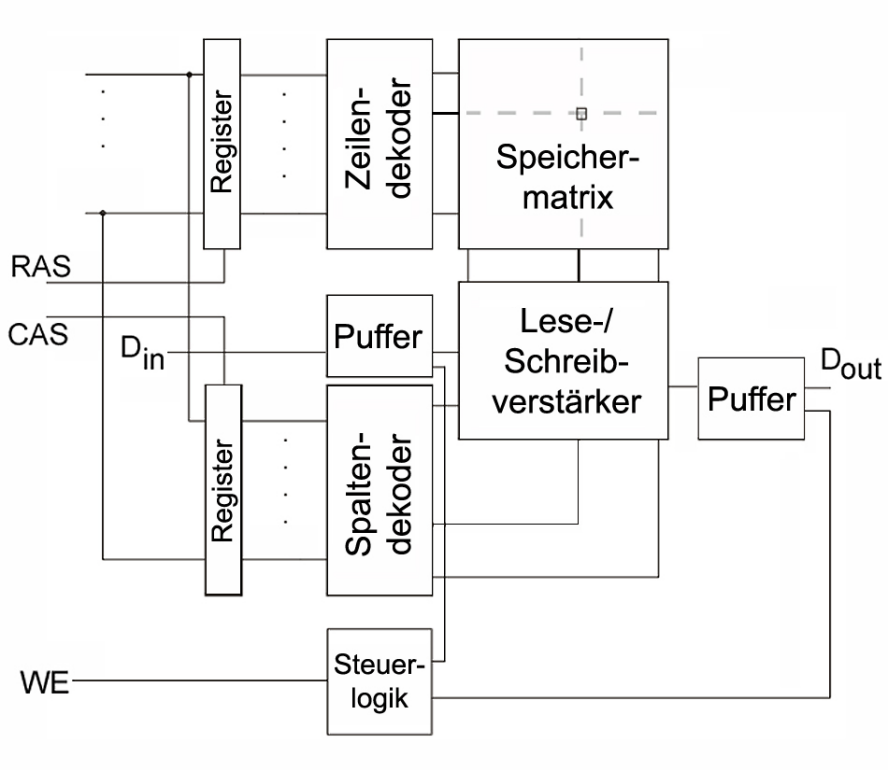


Abbildung 2.3: vereinfachtes Blockdiagramm - DRAM

Hinzu kommt, dass die Ladung der Kondensatoren auch ohne Zugriff auf die jeweilige Zelle nicht beliebig lang erhalten bleibt. Da Leckströme diese Ladungen reduzieren, würden nach einer gewissen Zeitspanne die Speicherinhalte der Zellen verloren gehen. Besonders nachteilig wirkt sich hierbei der Effekt aus, dass mit zunehmender Anzahl von Zugriffen auf einen Speicher auch die Temperatur in diesem ansteigt. Als Konsequenz erhöhen sich die unerwünschten Leckströme und fördern somit eine schnellere Entladung der Kondensatoren.
Daher muss der Speicherinhalt eines DRAM in periodischen Abständen zunächst komplett ausgelesen und anschließend wieder restaurierend beschrieben werden. Dieser

Vorgang, als Refresh bezeichnet, findet gegenwärtig in Abständen von wenigen Millisekunden (64*ms*) statt.

Moderne DRAM lösen diesen Refresh selbst aus, dazu benötigen sie einen Adressenzähler und einen internen Timer. Andernfalls geben externe Steuersignale den Anstoß zum Refresh. Der Refresh erfolgt gewöhnlich zeilenweise, da eine komplette Auffrischung des gesamten Speichers aufgrund des hohen Strombedarfs zu Spannungseinbrüchen führen könnte.

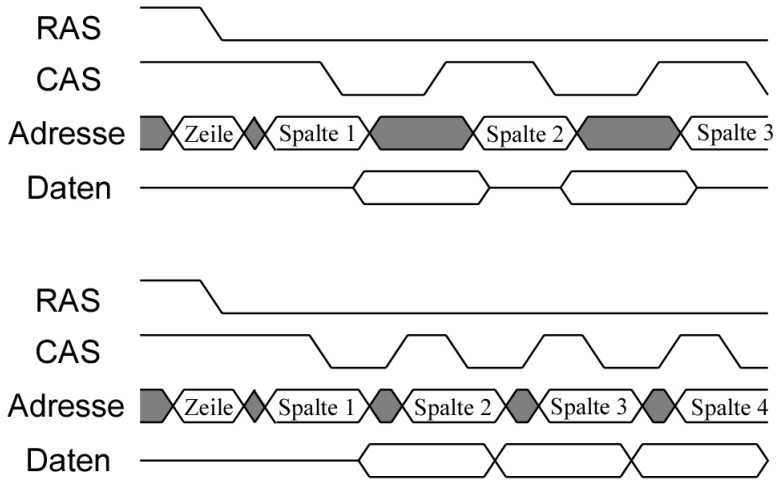


Abbildung 2.4: Vergleich zwischen Fast-Page und Hyper-Page Modus [Sch04]

Das Geschwindigkeitswachstum heutiger DRAM-Speicher verhält sich eher träge gegenüber dem der Prozessoren. Daher sind viele Ansätze gemacht worden, DRAM mit einem schnelleren Datenzugriff zu realisieren.

1. Im **Nibble Modus** wird das Auslesen 4 aufeinander folgender Zellen einer Zeile erlaubt. Ausgenutzt wird hierbei, dass diese Zelleninhalte sowieso an den Verstärkern vorliegen und dementsprechend nacheinander adressiert werden können. Das erneute Adressieren einer Zeile entfällt somit, die Spaltenadresse wird intern inkrementiert.

2. Der **Page Modus** als Erweiterung des Nibble Modus erlaubt eine kürzere Zugriffszeit auf Zeilen mit gleicher Adresse. Hierbei wird allerdings nicht verlangt, dass

die Spaltenadressen direkt aufeinander folgen müssen. Daher wird eine erneute Übertragung der Spaltenadresse notwendig.

3. Der **Fast-Page Modus** erlaubt eine weitere Verbesserung des Datenzugriffs. Während im Page-Modus eine gewisse Zeitspanne nach der fallenden CAS-Flanke vergehen muss, bevor die Spaltenadresse übernommen wird, geschieht dies im Fast-Page Modus direkt mit der fallenden Flanke.

4. Im **Hyper-Page Modus** bleiben die gelesenen Zelleninhalte bis zur nächsten fallenden CAS-Flanke gültig. Die Inhalte sind für den Prozessor auch während der nächsten Spaltenadressierung noch lesbar.
   Dieser Modus wird auch häufig als *extended data out* (EDO) bezeichnet.

Neben den verschiedenen Ansätzen, die einen schnelleren Datenzugriff garantieren, werden heutzutage viele Weiterentwicklungen des DRAM angeboten.

1. Der **synchrone DRAM** (SDRAM) stellt die wohl populärste Weiterentwicklung des asynchronen, mittels verschiedener Steuersignale (RAS, CAS, WE, OE) betriebenen DRAM dar. Aufgrund der großen Diskrepanz zwischen Zugriffszeit des Speichers und Zykluszeit des Prozessors mussten lange Wartezeiten und die damit verbundene Belegung des Speicherbusses in Kauf genommen werden.

   Im synchronen Betrieb ist diese Wartezeit genau definiert, da der Speicher mit dem externen Takt betrieben wird. Dadurch kann der Bus zwischenzeitlich anderweitig benutzt werden.
   Ebenso wie SSRAM verfügen SDRAM über zusätzliche, mit dem Prozessortakt verbundene Register, die einen Pipelinebetrieb ermöglichen. Ebenfalls erlauben solche Speicher den *data prefetch*, bei dem mehrere Zelleninhalte einer Zeile in getrennte Ausgabepuffer geladen werden, auf die dann mit einem Vielfachen der internen Frequenz (100-133 MHz) zugegriffen werden kann. Abhängig von dieser Frequenz werden SDRAM mit der Bezeichnung PC100 bzw. PC133 vermarktet.
   Die Aufteilung der SDRAM in interne Bänke, auf die möglichst abwechselnd zugegriffen (*bank interleaving*) wird, kann Verzögerungen verstecken, die durch einen Refresh oder Adressierung verursacht werden.

2. Beim **double data rate SDRAM** (DDR-SDRAM) wird die Zugriffsgeschwindigkeit bei gleicher Taktrate verdoppelt, da sowohl mit der fallenden als auch mit

der steigenden Taktflanke Daten übertragen werden. Dies erfordert verbesserte Schaltungen zur Synchronisierung. Hierzu wird ein differentieller Takt verwendet, der generell störfester und mit höherer Präzision auswertbar ist. Zusätzlich verfügen DDR-SDRAMs über eine *delay locked loop*-Schaltung welche ein *data strobe*-Signal erzeugt, das die Gültigkeit der Daten anzeigt. Die Speicherzellen von DDR-SDRAM arbeiten nicht schneller als bei SDRAM, werden aber paarweise angesprochen und dann nacheinander ausgegeben, so dass theoretisch doppelte Bandbreiten möglich sind. Im direkten Vergleich mit SDRAM werden bei DDR-SDRAM oftmals doppelte interne Frequenzen (200-266 MHz) angegeben. In Anlehnung an die Bezeichnung von SDRAM spricht man bei diesem Speicher daher in der Vermarktung auch von DDR200 bzw. DDR266.

Low-Power SDRAM

Verschiedene Halbleiterhersteller erweitern derzeit ihre Produktion um SDRAM, die speziell für mobile Endgeräte hergestellt werden. Solche Speicher werden daher auch häufig als **mobile SDRAM** bezeichnet. Hierbei sorgt ein im Speicher integriertes *power managment* (siehe Kapitel 4) für einen besonders geringen Leistungsverbrauch, insbesondere der Verbrauch durch die Refresh-Zyklen wird durch Anpassung an Temperatur und Speicherinhalt gesenkt.
Neben der geringen Leistungsaufnahme spielt auch die effiziente Flächenausnutzung des Speichers für den Einsatz in mobilen Endgeräten und generell in ES eine wichtige Rolle. Im Vergleich zu konventionellem SDRAM, der in *thin small outline package*-Gehäusen (TSOP) verbaut wird, können bis zu zwei Drittel der Fläche durch *fine pitch ball grid array*-Gehäuse (FBGA) eingespart werden. FBGA-Gehäuse verfügen über kleinere Kontaktabstände und kürzere Leiterbahnen, so dass die Fläche derzeitiger Realisierungen bei Kapazitäten von bis zu 512 MBit unter 100mm^2 liegt.

Seit der Standardisierung des low-power DRAM durch den *Joint Electron Device Engineering Council* (JEDEC) wird auch insbesondere im Entwurf von *application specific integrated circuits* (ASIC) und *systems on chip* (SOC) der leistungseinsparende Vorteil des mobile RAM berücksichtigt.

Eine Weiterentwicklung des mobilen RAM stellen die **pseudo-statischen RAM** (PSRAM) dar. Diese Speicher vereinen die Performance von SRAM mit der hohen Speicherdichte von DRAM und integrieren die energieeinsparenden Funktionen des mobile RAM. Da PSRAM anschlusskompatibel zu SRAM produziert wird, werden letztere als zukünftiger Ersatz für SRAM gehandelt. Derzeitig werden Speicherkapazitäten von 16 bis 128 MBit in Aussicht gestellt.

Embedded DRAM

Einen anderen Ansatz, den Flächenbedarf und den Leistungsverbrauch konventioneller SDRAM zu senken, stellt der **embedded DRAM** (eDRAM) dar. Die eDRAM-Technologie erfüllt diese Anforderungen durch die Integration mit dem ASIC als SOC, wodurch externe Busse und der damit verbundene Flächenbedarf wegfallen (s Abb. 2.5).

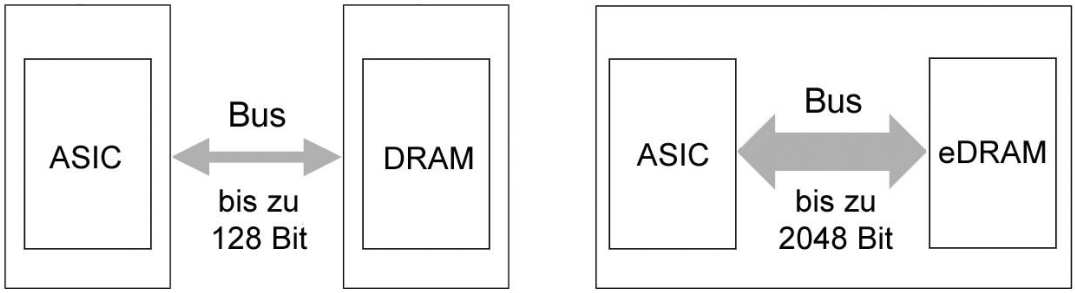


Abbildung 2.5: ASIC mit externem und integrierten Speicher

Zusätzlich wird dadurch das auf geringer Fläche entstehende Problem der elektromagnetischen Verträglichkeit durch nah aneinanderliegende Busleitungen kompensiert.
Weiterhin lassen sich Einsparungen beim Leistungsbedarf durch Wegfall von Treiberschaltungen und Puffer für externe Speicher erzielen [Bin99].
Eine weitere Stärke dieser Technologie liegt in der äußerst flexiblen Gestaltung des Speichers, so dass Kapazitäten als Vielfaches eines Megabit (MBit) realisierbar sind. Dadurch lassen sich anwendungsspezifische Größen für ASICs als Vielfaches eines MBit, aber auch Standardgrößen mit einer Kapazität von bis zu 64 MBit fertigen. Auch die Organisation des Speichers lässt sich hinsichtlich der Busbreite und des Flächenbedarfs weitgehend an seine Umgebung anpassen. Als Richtgröße für den Flächenbedarf wer-

den 1,5 MBit/mm^2 angegeben.

Ergänzt werden die Vorteile des eDRAM durch schnelle Zugriffszeiten, die unter 5 ns liegen und die geringe Betriebsspannung im Bereich von 1,5 Volt. Aufgrund dieser zahlreichen Vorteile wird der eDRAM auch als potentieller Konkurrent für SRAM angesehen. Besonders der immense Unterschied im Flächenbedarf machen den eDRAM zu einer attraktiven Alternative zum SRAM. Längst verfügen Hochleistungsprozessoren über L3-Caches, die in dieser Technologie implementiert werden.
Oft genannter Nachteil des eDRAM sind die höheren Herstellungskosten, die durch aufwendigere Lithographie-Techniken bei der Einbettung des Speichers entstehen. Derzeit sieht die Halbleiterindustrie 3 bis 4 zusätzliche Masken für die *on-chip* Integration des eDRAM vor.

2.2.2 Nichtflüchtige Speicher

Vorteil nichtflüchtiger Speicher ist die Beibehaltung ihrer Speicherinhalte auch nach dem Abschalten der Betriebsspannung. Hierbei muss allerdings auch in Kauf genommen werden, dass Speicherinhalte während des Betriebes nur langsam oder gar nicht geändert werden können. Abhängig davon, ob der Speicherinhalt eines solchen Speichers überhaupt änderbar ist, werden diese als reversible, andernfalls als irreversible Speicher bezeichnet. Da letztere Speicher ausschließlich für den lesenden Zugriff geeignet sind, werden diese auch als *read only memory* (ROM) bezeichnet.

Die Inhalte irreversibler ROM werden bereits in der Fertigung durch Maskenprogrammierung, bei programmierbaren ROMs (PROM) nach der Fertigung mittels spezieller Programmiergeräte, festgelegt. Hierbei besteht jede PROM-Zelle aus einer Diode und einer Schmelzsicherung (*fusable link*). Diese Sicherung wird durch Anlegen einer Überspannung mit dem Programmiergerät durchgeschmolzen (s. Abb.2.6).

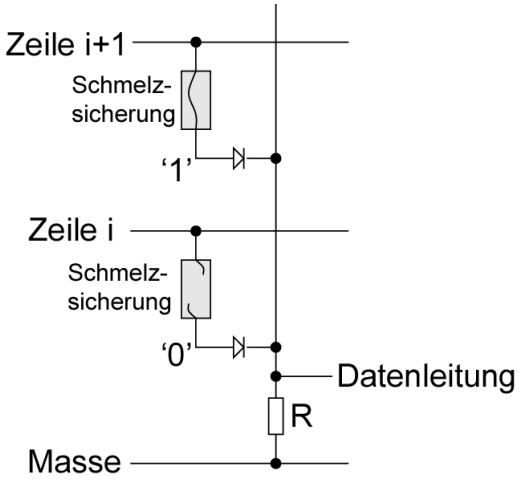


Abbildung 2.6: Aufbau einer PROM-Zelle [Mar03]

Die Möglichkeit einer eingeschränkten Wiederbeschreibbarkeit nichtflüchtiger Speicher bilden die reversiblen *erasable programmable* ROM (EPROM), die je nach verwendeter Lösch-Methode in *ultra-violet erasable programmable* ROM (UV-EPROM), *electrically erasable programmable* ROM (EEPROM) und *flash programmable* ROM (Flash) eingeteilt werden. Diese Speicher sind besonders wirtschaftlich herzustellen und finden dort Anwen-

dung, wo Informationen auf längere Sicht unverändert bereitstehen müssen.

1. Bei den UV-EPROM befinden sich an den Kreuzungspunkten der Speichermatrix spezielle MOS-Transistoren, die zwischen *gate* und *drain* eine isolierte leitende Zone (*floating gate*) besitzen. Auf diese kann mit hoher Spannung Ladung aufgebracht werden, entladen werden diese mittels ultraviolettem Licht. Hierfür werden an den entsprechenden Stellen des Keramikgehäuses Quarzfenster angebracht, durch welche die Bestrahlung stattfinden kann. Ein wesentlicher Nachteil besteht darin, dass diese Speicher entweder nur ganz, oder zumindest in größeren Bereichen löschbar sind.

2. Beim EEPROM kann der Löschvorgang elektrisch ausgelöst werden. Dies erfordert allerdings eine komplexere Schaltung, pro Speicherzelle wird ein weiterer Transistor benötigt. Nachteilig wirkt sich in diesem Zusammenhang natürlich der größere Flächenbedarf durch den zusätzlichen Transistor aus. Beschrieben werden EEPROM analog zu EPROM, wobei sich hier allerdings der größere Flächenbedarf wiederum rentiert: Einzelne Zellen sind beschreib- und löschbar.

3. Die heute gängigsten nichtflüchtigen, wiederbeschreibbaren Speicher stellen die Flash-Speicher dar. Das Beschreiben solcher Speicher erfordert nicht die zuvor beschriebenen aufwendigen Methoden. Sie werden normal adressiert und mittels spezieller Befehle beschrieben oder gelöscht.

   Generell unterscheidet man bei der Realisierung zwei Strukturen (s. Abb 2.7):

   - Bei der NOR- oder parallelen Struktur sind die Zellen in einer Speichermatrix angeordnet, der Zugriff auf eine Speicherzelle erfolgt wie beim RAM. Beschrieben werden können diese Speicher allerdings nur in ganzen Blöcken (*erase-blocks*), die bei dieser Struktur verhältnismäßig groß ausfallen, so dass Nachteile bei der Schreibgeschwindigkeit in Kauf genommen werden müssen. Gängige Speicher dieser Struktur weisen Kapazitäten von 64 MBit vor.
   - Bei der NAND- oder seriellen Struktur liegen die Schwerpunkte auf höherer Schreibleistung, höherer Speicherkapazität und minimierten Herstellungskosten. Die höhere Schreibleistung wird im wesentlichen durch eine geringere Größe der *erase-blocks* erreicht. Mit Speicherkapazitäten, die teilweise über 512 MBit liegen, stellt sich eine besondere Eignung als Hintergrundspeicher

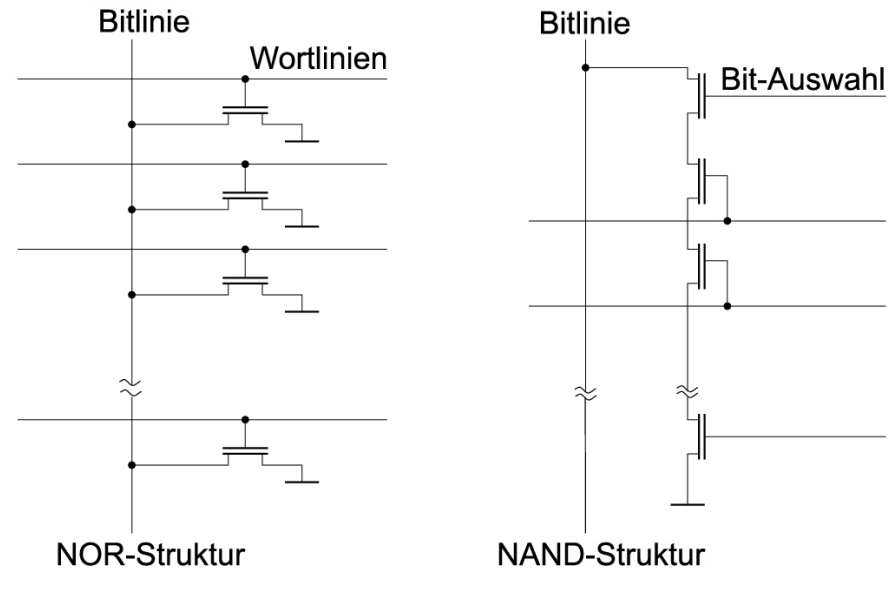


Abbildung 2.7: NOR- und NAND-Struktur [Sch04]

heraus. Besonders nachteilig an dieser Struktur ist der ausschließlich seitenweise Zugriff, so dass das Lesen einer bestimmten Adresse mit einem erheblichen Zeitaufwand verbunden ist.

Flash-Speicher sind das Mittel der Wahl, wenn ES größeren Instruktions- oder Datenspeicher benötigen. Die *on-chip* Integration sowie eine Reduktion der Betriebsspannung auf 1,8 V und die Bereitstellung energieeinsparender Betriebmodi lässt einen eindeutigen Trend zu **low-power flash rom** erkennen. Obwohl es sich bei Flash-Speichern mit Zugriffszeiten unter 100 ns (NOR-Struktur) um verhältnismäßig schnelle nichtflüchtige Speicher handelt, sind die jeweiligen Nachteile beider Strukturen und die begrenzte Anzahl an möglichen Schreibvorgängen (10^5 bis 10^6), so gravierend, dass von einer Verwendung als Hauptspeicher abzusehen ist.

2.3 Compiler für eingebettete Systeme

Ein Compiler [ASU86] erzeugt aus einem Quellprogramm, das zumeist in einer Hochsprache geschrieben ist, ein Zielprogramm (Objektcode). Gängige Hochsprachen bieten einen hohen Abstraktionsgrad und ersparen dem Programmierer unter Verwendung eines Compilers die mühselige Assemblerprogrammierung. Der vom Compiler erzeugte Objektcode muss natürlich korrekt und lauffähig sein. Zusätzlich sollten die Übersetzungszeiten kurz gehalten und gute Fehlerdiagnosen ermöglicht werden.

Bei dem Entwurf von Compilern für eingebettete Systeme sollten zusätzliche Auflagen beachtet werden [Leu01]. Zum einen werden deutlich höhere Anforderungen an die Codequalität gestellt, da sich die Codegröße bei ES mit eingebetteten Speichern unmittelbar auf die Chipfläche auswirkt und Zugriffe auf externe Speicherpartitionen deutlich mehr Energie verbrauchen. Ineffizienter Code kann möglicherweise negativen Einfluss auf die Ausführungszeit des Programms haben und somit zur Verletzung von Echtzeit-Bedingungen führen. Zum anderen sollte mehr Zeit in die Codegenerierung und Optimierung investiert werden, denn eingebettete Prozessoren weisen häufig spezielle Befehlssätze auf, die auch dementsprechend vom Compiler genutzt werden sollten.

Die Aufteilung des Compilers in Frontend und Backend ermöglicht die Wiederverwendbarkeit dieser beiden Komponenten. Während das Frontend auf die Hochsprache zugeschnitten ist und ein standardisiertes Austauschformat generiert, wird das Backend speziell für die betrachtete Zielarchitektur implementiert. So ist es möglich, für verschiedene Zielarchitekturen das gleiche Frontend und für verschiedene Hochsprachen das gleiche Backend wiederzuverwenden (retargierbare Compiler, [ML01]).

2.3.1 Frontend eines Compilers

Das Frontend eines Compilers erstellt aus dem Quellcode eines in einer Hochsprache geschriebenen Programms eine maschinenunabhängige Zwischendarstellung *(Intermediate Representation, IR)*, auf der bereits an dieser Stelle unabhängig vom Zielprozessor Optimierungen vorgenommen werden können. Abschließendes Ziel ist die Darstellung der IR durch Flussgraphen, die als Schnittstelle zum Backend des Compilers dient.

Korrektheitsüberprüfung

Die wesentlichen Teilschritte zur Erzeugung einer Zwischendarstellung bestehen aus Analysen des Programms sowie der Erstellung von Symboltabellen.

- **Lexikalische Analyse:**
  Die Aufgabe der lexikalischen Analyse ist es, aus dem Eingabestrom Zeichenfolgen zu erkennen und zu sogenannten Token (Schlüsselwörter, Zahlen, Bezeichner) zusammenzufassen. Kommentare hingegen werden in dieser Analyse herausgefiltert. Diese Aufgaben werden von einem *Scanner* übernommen, die Kenntnis über die in der Programmiersprache definierten Schlüsselwörter besitzt.

- **Syntaktische Analyse:**
  Aufgabe der syntaktischen Analyse ist es, einen Syntaxbaum entsprechend einer kontextfreien Grammatik G aufzubauen. Diese Grammatik besteht in der Regel aus einem Quadrupel $G = (\Sigma_T, \Sigma_N, R, S)$, welche sich aus Terminalen, Nichtterminalen, Regeln und dem Startsymbol zusammensetzt. Ziel der syntaktischen Analyse ist die Reduktion der Eingabe auf das Startsymbol. Hierzu fordert eine weitere Komponente, *Parser* genannt, fortlaufend vom *Scanner* Token an und entscheidet dann über Anwendung der in der Grammatik definierten Regeln.

- **Semantische Analyse:**
  In der semantischen Analyse wird einerseits eine kontextsensitive Korrektheitsüberprüfung durchgeführt, zum anderen eine Symboltabelle aufgebaut. Die deklarierten Bezeichner werden zusammen mit ihren Typinformationen in dieser Tabelle gespeichert, die im Anschluss vom Backend zur Codegenerierung verwertet wird.

Zwischendarstellung

Das Frontend eines Compilers erzeugt, wie bereits erläutert, eine maschinenunabhängige Zwischendarstellung des Quellprogramms. Die IR dient dabei als Austauschformat zwischen dem Frontend und dem Backend des jeweiligen Compilers. Üblicherweise wird die IR in einem 3-Adress-Code (3AC) dargestellt, der aus IR-Ausdrücken und IR-Statements besteht.

Definition 2.1 *Eine IR-Primitive ist entweder ein Symbol (Bezeichner) oder eine Konstante.*

Definition 2.2 *Ein (allgemeiner) IR-Ausdruck ist entweder*

- *ein primitiver IR-Ausdruck*
- *ein unärer Ausdruck für einen Operator und eine Primitive*
- *ein binärer Ausdruck e_1 op e_2 für einen Operator op und zwei Primitive*
- *ein Funktionsaufruf $f(e_1,...,e_n)$ für eine Funktion f und Primitive $e_1,...,e_n$*

Definition 2.3 *Ein IR-Statement ist*

- *eine Zuweisung $e_1 = e_2$ für einen primitiven IR-Ausdruck e_1 und einen IR-Ausdruck e_2*
- *ein Label (string) L*
- *ein unbedingter Sprung für ein Label L*
- *ein bedingter Sprung für ein Label L und eine Primitive e*
- *ein Return-Statement für eine Primitive e*

Optimierung der Zwischendarstellung

Optimierungen der IR sind maschinenunabhängig und dienen der vereinfachten Darstellung. Dabei sollen redundante Berechnungen und überflüssiger IR-Code entfernt werden. Weitere Vereinfachungen werden durch Auswertung von Berechnungen während des Übersetzungsvorgangs erzielt.

Im Rahmen der Forschungsarbeit am Lehrstuhl XII wurde das ANSI-C Frontend *LANCE* entwickelt, welches eine Vielzahl der obengenannten Vereinfachungen durchführt. Optimierungen für *constant folding, constant propagation, copy propagation, CSE elimination, dead code elimination* und *jump optimization* sowie die iterative Anwendung aller Optimierungen sind in diesem Frontend implementiert.

Für einige dieser Optimierungen ist eine Analyse des Datenflusses zwischen einzelnen IR-Statements erforderlich. Hierzu können für Ausdrücke und Statements die Mengen *define* und *use* gebildet werden.

Definition 2.4 *Zu einem Symbol e als Primitive (auf der rechten Seite) in einer Zuweisung s_1 sind die Definitionen def(e) die Menge aller IR-Zuweisungen s_2, so dass s_2 einen Wert an e zuweist und dieser Wert potentiell in s_1 verwendet wird. s_1 heißt dann datenabhängig von s_2.*

Definition 2.5 *Zu einer IR-Zuweisung s mit einem Symbol e auf der linken Seite sind die Benutzungen use(s) die Menge aller IR-Ausdrücke e mit $s \in def(e)$.*

In der Flussanalyse soll die Zwischendarstellung so strukturiert werden, dass die anschließende Codegenerierung erleichtert wird. Hierzu wird eine Darstellung des 3AC durch Graphen vorgenommen.

Definition 2.6 *Eine Folge von IR-Statements $B = (s_1, ..., s_n)$ heißt Basisblock (BB), falls*

- *der Einsprung in B nur nach s_1 erfolgen kann und*
- *eine Verzweigung aus B nur in s_n erfolgen kann.*

Definition 2.7 *Der Kontrollflussgraph (CFG) zu einer Funktion f ist ein gerichteter Graph $G = (V, E)$, wobei $V = (B_1, ..., B_n)$ die Menge der Basisblöcke in der IR ist. Falls zwei Basisblöcke B_i und B_j unmittelbar nacheinander ausgeführt werden können, so enthält E die Kante (B_i, B_j).*

Kontrollflussgraphen repräsentieren somit die Basisblockstruktur eines Programms, für die Analyse der Datenabhängigkeiten innerhalb eines Basisblocks wird jedoch eine weitere strukturierte Darstellung erforderlich.

Definition 2.8 *Der Datenflussgraph (DFG) zu einem Basisblock $B = (s_1, ..., s_n)$ ist ein gerichteter, azyklischer Graph $G = (V, E)$. Die Knotenmenge $V = (v_1, ..., v_m)$ wird durch die primären Eingangswerte, die primären Ausgangswerte und die Operatoren in den IR-Statements gebildet. Die Menge E enthält eine Kante (v_i, v_j), falls eine Datenabhängigkeit zwischen den zu v_i und v_j gehörenden Statements besteht.*

2.3.2 Backend eines Compilers

Maschinenunabhängige Zwischendarstellungen dienen als Austauschformat für die Codegenerierung. In dieser Phase werden maschinenabhängige Zwischendarstellungen (*low-level IR, LIR*) erzeugt, bei denen das Programm durch Vorstufen realer Maschineninstruktionen dargestellt wird.

Die für imperative Sprachen meist verbreitete Form einer Zwischendarstellung ist, wie bereits erläutert, der 3AC. Diese einfachere Struktur der Programmdarstellung erleichtert die Durchführung von architekturspezifischen Optimierungen und ermöglicht eine schnelle Umordnung von Anweisungen, da bereits eine starke Ähnlichkeit zu Assemblerbefehlen gegeben ist.

Codegenerierung

Die Aufgabe der Codegenerierung ist die Erzeugung eines korrekten, ausführbaren Codes, der die Ressourcen der Zielarchitektur effizient nutzt. Die einzelnen Aufgaben gliedern sich in die drei Bereiche Instruktionsauswahl, Registerallokation und Instruktionsanordnung. Einige Probleme dieser Phase sind NP-hart, so dass eine effiziente Durchführung der Codegenerierung eine weitere Anforderung darstellt.

- **Instruktionsauswahl:**
  Die Instruktionsauswahl (*code selection*) hat die Aufgabe, die abstrakten Operationen der Zwischendarstellung möglichst kostenoptimal mit Instruktionen der Zielarchitektur zu überdecken. Die Schwierigkeit dieser Auswahl ist stark vom Aufbau der Zielarchitektur abhängig. Je irregulärer die Architektur, desto mehr Spezialbehandlungen erfordert die eigentliche Auswahl. Der meist verbreitete Ansatz zur Instruktionsauswahl basiert auf einem Verfahren der Mustererkennung. Hierzu wird der DFG an gemeinsamen Teilausdrücken in Bäume aufgespalten.

  Definition 2.9 *Ein Knoten $v_i \in V$ eines DFG $G = (V,E)$ mit mehr als einer ausgehenden Kante heißt gemeinsamer Teilausdruck (common-subexpression, CSE). Der DFG G ohne CSE heißt Datenflussbaum (DFT).*

Für einen DFT kann in linearer Zeit eine optimale Auswahl von Assemblerbefehlen durch Überdeckung (*tree parsing*) auf Basis von Baumgrammatiken erzielt werden [Leu01].

- **Registerallokation:**
  In dieser Phase werden die zunächst noch virtuellen Register durch physikalisch vorhandene Register ersetzt. Da Zugriffe auf Register im allgemeinen schneller und energieeffizienter als Speicherzugriffe sind, sollten insbesondere häufig verwendete Werte in Registern gehalten werden.
  Hierzu wird eine Analyse der Lebensspannen der einzelnen Register vorgenommen, die bis hin zur interprozeduralen Sichtweise des Datenflusses reichen kann. Durch Betrachtung größerer Ausschnitte kann zwar ein besseres Ergebnis erzielt werden, gleichzeitig aber auch ein deutlicher Mehraufwand entstehen.
  Einen systematischen Ansatz für die Registerallokation stellt die Färbung von Interferenzgraphen dar. Hierbei entstehende Konflikte erzwingen den Abwurf eines Registers und dessen Auslagerung im Hauptspeicher (*spilling*, [App98]).

- **Instruktionsanordnung:**
  Die Veränderung der Instruktionsanordnung (*instruction scheduling*) kann durchaus positiven Einfluss auf die Effizienz der Codegenerierung haben, da beispielsweise das zuvor erwähnte Abwerfen eines Registers und der damit verbundene *spill code* verhindert werden können. Bei Veränderung der bestehenden Anordnung müssen Abhängigkeiten zwischen den einzelnen Instruktionen zur Wahrung der Korrektheit des Programms unbedingt beachtet werden. Zur Erläuterung der Begrifflichkeiten Ausgabe-, Daten- und Antidatenabhängigkeit sei an dieser Stelle auf [Mar03] verwiesen.

Die vorgestellten Teilaufgaben der Codegenerierung wurden zwar getrennt betrachtet, können jedoch von einer gemeinsamen Behandlung der Teilaufgaben profitieren. Man beachte allerdings, dass die sukzessive Behandlung das Problem der Phasenkopplung hervorruft. Die Berechnung der optimalen Lösung durch gemeinsame Betrachtung aller Teilaufgaben stellt ein NP-vollständiges Entscheidungsproblem dar. Die Teilaufgabe der Registerallokation (Färbungsproblem für allgemeine Grpahen) ist bereits NP-hart.

Kapitel 3

Verwandte Arbeiten

Im Rahmen der allgemeinen Anforderungen eingebetteter Systeme stellt die compilergesteuerte Optimierung ein mächtiges Instrument zur Reduktion von Laufzeit, Energieverbrauch und der Erzeugung von kompaktem Code dar. Diese Ziele werden einerseits durch Codeoptimierungen und andererseits durch die compilergesteuerte Nutzung heterogener Speichersysteme erzielt, da insbesondere die Speicher einen deutlichen Einfluss auf Laufzeit und Energieverbrauch haben.

3.1 Compilergestützte Reduktion des Energieverbrauchs

Besonders die in ES vorhandenen schnellen Scratchpad-Speicher sind gegenwärtig Gegenstand der Forschung. In diesem Zusammenhang wurde zunächst die compilergesteuerte statische Nutzung durch Verlagerung von Daten [PDN99] und Instruktionen [IY00] getrennt betrachtet. Am LS12 wurde in [SLWM02] eine gemeinsame Betrachtung von Instruktionen und Daten für die Verlagerung in den Scratchpad vorgenommen.

Ein erster Ansatz für die dynamische Nutzung des Scratchpads wurde in [Kan01] vorgenommen, bei der Datenstrukturen partitioniert werden und sich bei Abarbeitung gegenseitig aus dem Scratchpad-Speicher verdrängen. In [Gru02],[SGW$^+$02] wurde die Energieminimierung eingebetteter Programme durch die dynamische Nutzung eines Scratchpad-Speichers als Instruktionsspeicher untersucht. Im Rahmen der Forschungs-

arbeit am LS12 wurde in [VWM04] eine gemeinsame Nutzung des Scratchpad-Speichers für die dynamische Verlagerung von Instruktionen und Daten vorgenommen.

Weiterhin sind ES häufig mit Flash-Speichern ausgestattet, die durch einen entsprechenden Hauptspeicher in adäquat schneller Technologie ergänzt werden. Unter der Annahme eines Hauptspeichers mit geringer Kapazität wurde in [LPL+04] ein compilergestützter Nachlademechanismus (*demand paging*) entworfen, durch den der Hauptspeicherbedarf um 33% gesenkt werden konnte. Eine Methode zur Energiereduktion (*energy-aware demand paging*) in Speichersystemen mit NAND-Flash- und SDRAM-basierten Speichertechnologien wird in [Par04] vorgeschlagen.

Die compilergesteuerte Reduktion des Energieverbrauchs von Flash-basierten Speichertechnologien wird in Kap.5 der Arbeit vorgenommen.

Die in Kap.4 formulierte Optimierungsmethode zur statischen Nutzung von Scratchpad-Speichern basiert auf den Ergebnissen von zwei vorhergehenden Diplomarbeiten am LS12, die im folgenden genauer erläutert werden.

3.1.1 Statische Ausnutzung von Scratchpad-Speichern

Zobiegala hat sich in seiner Arbeit [Zob01] dem Ziel gewidmet, durch Nutzung eines Scratchpad-Speichers eine Reduktion des Energieverbrauchs der untersuchten Programme herbeizuführen. Der wesentliche Teil seiner Arbeit bestand darin, häufig ausgeführte Teile des Programms (Programmobjekte) in den Scratchpad zu verlagern.

Die Auswahl der Programmobjekte (Funktionen, Basisblöcke und globale Variablen) erfolgt statisch durch Analyse des Programms anhand des *Function-Call Graph* (FCG) und den in diesem Graph enthaltenen Informationen über Größe und Ausführungshäufigkeit. Bei Aufspaltung des Kontrollflusses werden die ausgehenden Kanten mit einer Wahrscheinlichkeit von 50% belegt. Weiterhin werden Schleifen mit einer festen Anzahl an Durchläufen versehen.

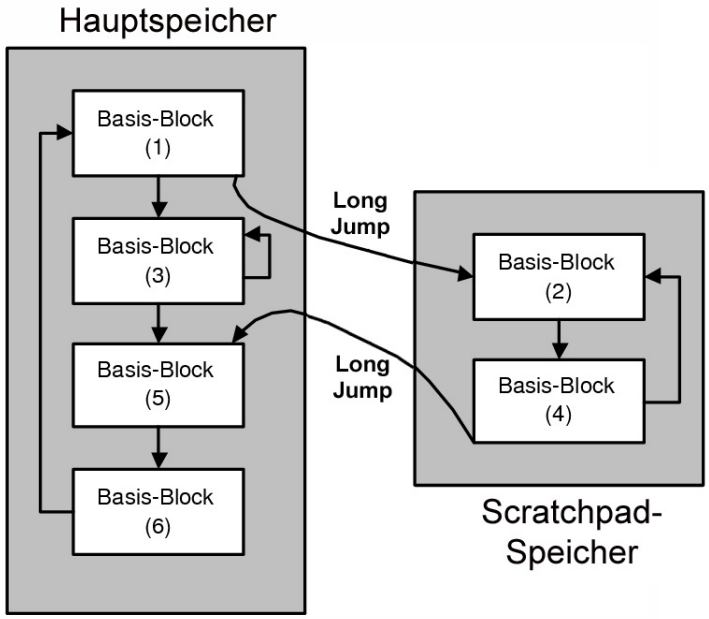


Abbildung 3.1: Statische Nutzung eines Scratchpad-Speichers [Hel04]

Die Auswahl geeigneter Programmobjekte, die in den Scratchpad-Speicher zur Reduktion des Energieverbrauchs verschoben werden sollen, stellt eine Analogie zur Formulierung des Rucksack-Problems dar. Hierbei muss beachtet werden, dass das Ablegen zweier aufeinanderfolgender Basisblöcke in unterschiedlichen Partitionen mit dem Einfügen eines zusätzlichen Befehls verbunden ist. Zur Überwindung der hohen Sprungdistanz (durch weit auseinanderliegende Adressen) wird ein zusätzlicher *long jump* benötigt (s. Abb3.1).

Hinsichtlich der berücksichtigten Abhängigkeiten und den damit verbundenen Problemgrößen macht Zobiegala folgende Fallunterscheidung:

- **Eindimensionales Rucksack-Problem:**
  Lediglich die Abhängigkeiten zwischen einer Funktion und den in ihr enthaltenen Basis-Blöcken werden berücksichtigt, so dass entweder eine ganze Funktion oder aber eine Menge von Basisblöcken dieser Funktion verschoben werden. Zur Auswahl der Programmobjekte wird ein *Branch & Bound*-Algorithmus gewählt.

- **Mehrdimensionales Rucksack-Problem:**
  Der Mehraufwand, der durch die zusätzlichen Sprünge zwischen Partitionen entsteht, lässt sich dann vermeiden, wenn der nachfolgende Basisblock ebenfalls in

den Scratchpad verlagert wird. Um dies zu berücksichtigen, bildet Zobiegala *Multi-Basis-Blöcke*, die eine Konkatenation hintereinander liegender Basisblöcke darstellen. Da die Problemgrösse durch hinzukommende Abhängigkeiten zwischen den Basisblöcken innerhalb der betrachteten Funktion wächst, wird hier ein ILP-Modell erzeugt und mittels des IP-Solver CPLEX [Ilo] gelöst.

Als Ergebnis der Arbeit ergab sich eine erhebliche Energieeinsparung von bis zu 80% bei gleichzeitiger Performancesteigerung gegenüber einem System ohne Scratchpad. Dieser enorme Vorteil lässt sich im wesentlichen dadurch begründen, dass häufig die kompletten Programme durch den Scratchpad aufgenommen werden konnten. Von Abweichungen dieser Ergebnisse bei Untersuchung größerer *benchmarks* war also auszugehen.

3.1.2 Partitionierung von Scratchpad-Speichern

Helmig hat in seiner Arbeit [Hel04] das Energiesparpotential durch die Nutzung partitionierter Speicher nachgewiesen. Hierbei wird ausgenutzt, dass die Zugriffsenergie sich proportional zur Größe des Speichers verhält. Der wesentliche Teil seiner Arbeit bestand darin, den vorhandenen Scratchpad-Speicher in mehrere kleine Speicherpartitionen zu zerlegen und das Programm auf die einzelnen Partitionen zu verteilen.

Im Rahmen seiner Untersuchungen hat Helmig herausgestellt, dass es sinnvoll sein kann, Basisblöcke zu zerlegen. Dies vergrößert auf der einen Seite zwar die Anzahl der zu verteilenden Programmobjekte, auf der anderen Seite gibt diese Zerlegung dem ILP-Solver jedoch die Möglichkeit, geeignete Basis-Blöcke in die bestmöglichste Speicherpartition zu verschieben. Dies wäre ohne eine Teilung unter Umständen nicht möglich, da zu große Basisblöcke nicht in kleinen Speicherpartitionen abgelegt werden können und somit sogar möglicherweise der Scratchpad nicht nutzbar wäre.

Zur Bestimmung der Ausführungs- und Zugriffshäufigkeiten wurde die statische und dynamische Analyse verwendet.

Helmig kam zu dem Ergebnis, dass mit einer neu formulierten Optimierungsmethode der Speicherenergieverbrauch bis zu 97% reduziert werden konnte. Im Vergleich zu

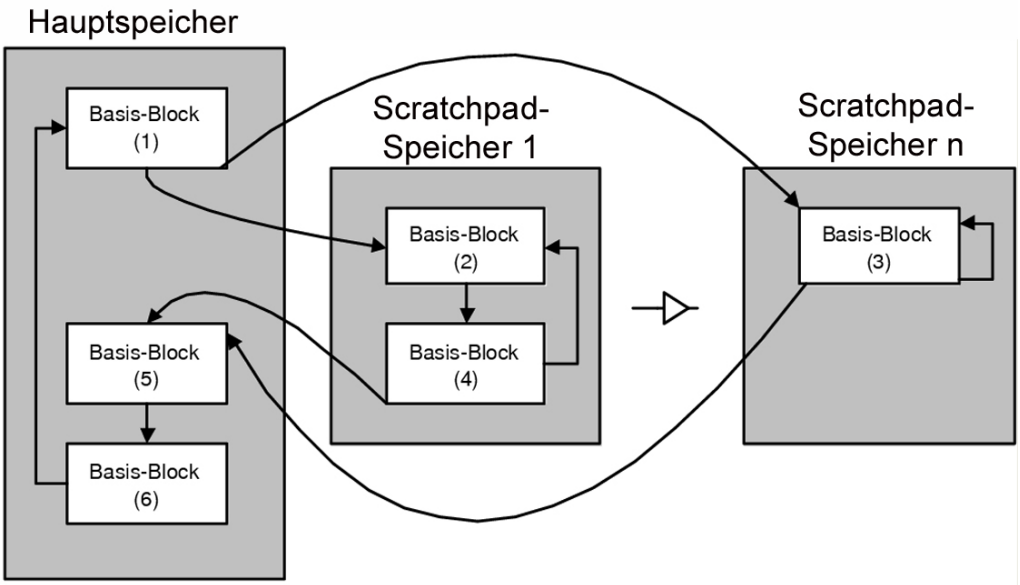


Abbildung 3.2: Nutzung partitionierter Scratchpad-Speicher [Hel04]

[Zob01] konnte durch die Partitionierung eine Verbesserung von bis zu 20% erzielt werden.

Kapitel 4

Low-Power Speicher in SDRAM-Technologie

Eine zentrale Anforderung an diese Arbeit stellt die Untersuchung moderner Speicher in SDRAM-Technologie dar. Die typisch für diese Technologie schnellen *low-power* SDRAM (LP-SDRAM) mit großer Kapazität, kleinen Abmessungen und geringer Leistungsaufnahme, bieten sich aufgrund ihrer Eigenschaften für die weiteren Betrachtungen an.

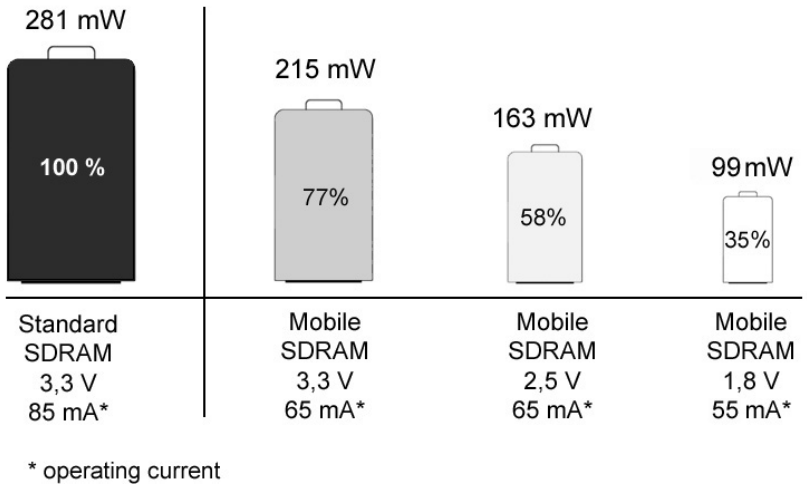


Abbildung 4.1: Leistungsaufnahmen verschiedener SDRAM

Die zu der Kategorie der LP-SDRAM gehörenden Mobile SDRAM weisen dank erheblicher Spannungsreduktion eine adäquat geringere Leistungsaufnahme vor. Herkömmliche SDRAM werden mit einer Spannung von 3,3 Volt betrieben, Mobile SDRAM und deren Treiberschaltungen hingegen benötigen je nach Ausführung nur 1,8 oder 2,5 Volt. Weiterhin kann durch besondere Schaltungstechnik und Herstellungstechnologie die Stromaufnahme im typischen Betrieb (*operating current*) auf 70% gegenüber konventionellem SDRAM abgesenkt werden (s. Abb 4.1).

Zunächst wird das in diesen Speichern integrierte *power managment* vorgestellt. Zentrale Komponente dieses Kapitels stellt die im Anschluss folgende Leistungskalkulation von SDRAM-basierten Speichertechnologien dar. Anhand der Analyse der Speicherzugriffe und Zugriffszeiten ergibt sich ein Energiemodell, mit dem der applikationsspezifische Energieverbrauch eines SDRAM präzise abgeschätzt werden kann. Basierend auf der Auswertung des Energieverbrauchs wird zum Abschluss dieses Kapitels eine Optimierungsmethode unter Nutzung von SDRAM und Scratchpad-Speichern motiviert und formalisiert.

4.1 Funktionalität von Low-Power SDRAM

Im folgenden wird die spezielle Funktionalität moderner LP-SDRAM erläutert, durch die sich diese Speicher maßgeblich von konventionellen SDRAM unterscheiden [Mic02b].

- Herkömmliche SDRAM nehmen eine konstant hohe Refresh-Rate an, um den Betrieb auch unter maximaler Wärmeentwicklung zu gewährleisten. Hierbei wird durchgehend eine *worst case* Temperatur auf dem Speicher angenommen. Die in den betrachteten Speichern integrierte Funktionalität *temperature compensated self refresh* (TCSR) steuert eine temperaturabhängige Refresh-Rate, die bei geringer Wärmeentwicklung die Refresh-Zyklen und somit die Leistungsaufnahme reduziert.
- Beim Refresh wird nicht berücksichtigt, dass möglicherweise einzelne Bänke des Speichers nicht genutzt werden, so dass immer der komplette Speicher aufgefrischt wird. Die Funktionalität *partial array self refresh* (PASR) sorgt an dieser Stelle für erhebliche Reduktion des Leistungsverbrauchs. Mittels PASR werden die

Refresh-Operationen auf die Bereiche mit gespeicherten Daten beschränkt.

Ergänzt wird die Funktionalität durch den *power-down* (Abschalten der Augangstreiber- und Pufferschaltungen) und *deep power-down*-Modus (DPD). Der DPD wird dann aktiviert, wenn der Speicher für eine längere Zeitspanne nicht mehr benötigt wird, um maximale Einsparungen zu erzielen [Sam02]. Die Speicherinhalte gehen allerdings im DPD verloren.

Mobile SDRAM

Der Mobile SDRAM mit Kapazitäten von 64 bis 256 MBit ist ein aktueller Vertreter der LP-SDRAM und verfügt über das zuvor beschriebene *power managment*. Integriert in ein FBGA-Gehäuse weist dieser Speicher in seiner kleinsten Ausführung einen Flächenbedarf von 64mm^2 vor. Die verschiedenen Betriebsparameter und die integrierte Funktionalität werden in zwei Registern programmiert.

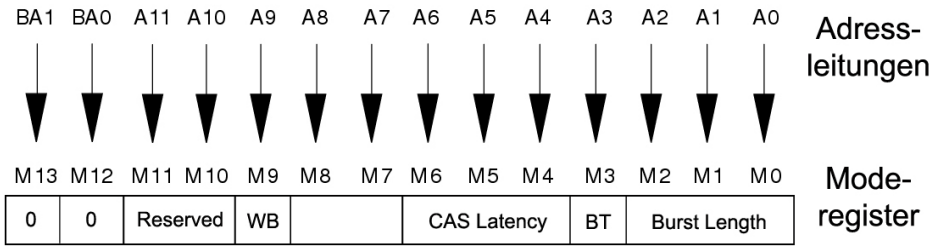


Abbildung 4.2: Moderegister [Mic04c]

Im *mode register* (s. Abb 4.2) werden die Parameter für den Speicherzugriff konfiguriert. Die jeweiligen Parameter lassen sich wie folgt interpretieren:

- Die Anzahl der Spalten, die bei einem Speicherzugriff im *burst mode* adressiert werden, wird durch den Parameter *burst length* bestimmt. In Kombination mit dem Parameter *write burst mode* (WB) wird dann festgelegt, ob sich die *burst length* auf den Lese- oder Schreibzugriff bezieht.

- Die Zugriffe im *burst mode* erfolgen je nach Konfiguration des *burst type*-Registers (BT) sequentiell (innerhalb einer Bank) oder *interleaved* (abwechselnder Zugriff auf verschiedene Bänke).
- Der Betrieb des Speichers mit unterschiedlichen Taktfrequenzen wirkt sich auf die *CAS latency* (CL) aus. Die jeweilige CL kann im *mode register* entsprechend der Betriebsfrequenz angepasst werden.

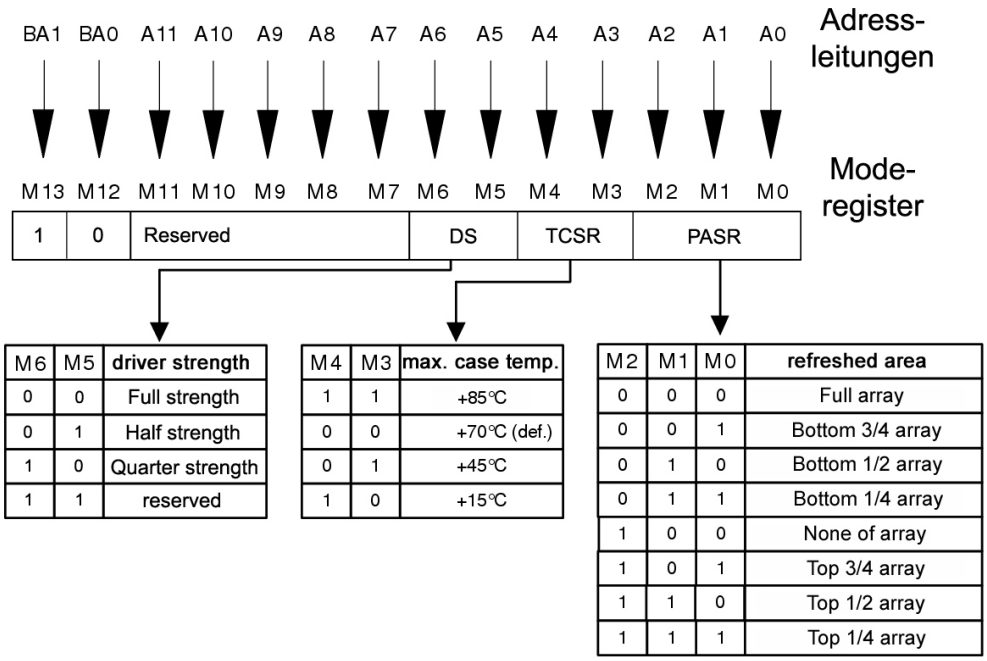


Abbildung 4.3: Extended Moderegister [Mic04c]

Das *extended mode register* beinhaltet die Parameter für die integrierte *self-refresh* Funktionalität und konfiguriert somit das *power-managment* des mobilen SDRAM. Programmiert werden beide Register durch den Befehl `LOAD MODE REGISTER` durch entsprechende Werte auf den Adressleitungen BA1 und BA0. Hierbei ist das *extended mode register* für den *auto-refresh mode* vorprogrammiert (*full array, 85°C*) . Im *self-refresh mode* wird dann im Gegensatz zum *auto-refresh mode* der Speicher so konfiguriert, dass die Aufrechterhaltung der Speicherinhalte durch das *power-managment* gewährleistet wird. Der *self-refresh mode* ist dann zu aktivieren, wenn auf diesen Speicher für eine gewisse Zeit nicht zugegriffen wird.

Hierzu bieten sich folgende Konfigurationsmöglichkeiten:

- Für die Codierung von Regionen stehen drei Bit im *mode register* zur Verfügung, auf die sich der Refresh (PASR) beschränkt.
- Insgesamt vier adäquate Refresh-Raten (TCSR) stehen für die von Sensoren gemessenen Temperaturen im Bereich von 15° bis 85° C zur Verfügung.
- Ausgehend von der *full drive strength* ($C_{LOAD} = 30pF$, s. Kap. 4.2.1) kann die Ausgangstreiberstärke *driver strength* auf die Hälfte oder ein Viertel reduziert werden.

Das zuvor beschriebene *power managment* wird derzeit auch in asynchrone pseudo-statische RAM (PSRAM) integriert. Die Bezeichnung pseudo-statisch ergibt sich durch die Anschlusskompatibilität zu SRAM (*drop-in replacement*) und die Realisierung der Speicherzellen in DRAM-Technologie. Der eigentliche Betrieb erfolgt analog zum SRAM, indem eine logische Schaltung den jeweiligen Speicherzugriff auf die DRAM-Steuerung umsetzt. Diese Speicher werden, trotz ihrer internen Speicherorganisation, anhand ihrer Datenblattangaben in die Kategorie der SRAM eingeordnet und daher von der weiteren Betrachtung ausgeschlossen.

4.2 Energieverbrauch von SDRAM

Das folgende Kapitel befasst sich detailliert mit der Leistungsaufnahme eines Speichers in SDRAM-Technologie. Hierzu werden sukzessive Gleichungen aufgestellt, die letztlich zur Generierung eines Werkzeugs dienen, um die applikationsspezifische Leistungsaufnahme eines solchen Speichers abschätzen zu können.

Grundlage für dieses Kapitel stellt ein leistungsbasiertes Energiemodell aus [Mic01] dar, welches in ähnlicher Form von [Raw04] genutzt wird. Die verwendeten Bezeichnungen der Kenngrößen (s. Tabelle 4.1) orientieren sich hierbei an den typischen Datenblattangaben von Micron. Die Beschreibungen der jeweiligen Kenngrößen werden allerdings allgemein gehalten, so dass die herstellerunabhängige Verwendbarkeit der angestellten Überlegungen gegeben ist.

Ein alternatives Energiemodell für Speicher in SDRAM-Technologie wird in [LKC02] vorgestellt. Dieses Modell basiert auf dem Tripel (M, Φ, Ξ), wobei M einen endlichen Zustandsautomaten darstellt. Mit Φ wird der statische Energieverbrauch im Zustand S angegeben, Ξ stellt die dynamische Energie dar, die mit einem Zustandsübergang T verbunden ist. Somit lässt sich in jedem Zustand und Zustandswechsel der jeweilige Energieverbrauch erfassen.
Der dynamische Energieverbrauch setzt sich aus folgenden Komponenten zusammen:

- *hamming-distance dependant dynamic energy*
- *weight dependant dynamic energy*
- *common-mode dynamic energy*

Die statische Energieverbrauch wird durch die beiden Komponenten *weight dependant static energy* und *constant leakage energy* gebildet. Eine detailliertere Erläuterung der Energiekomponten wird in [LKC02] vorgenommen. Anhand der Simulation wird dann unter Hinzunahme des Zustandsautomaten der Gesamtenergieverbrauch berechnet.

4.2.1 Leistungs- und Energieberechnung

Als Einleitung wird in Abb.4.4 ein vereinfachter Zustandsautomat eines SDRAM abgebildet, anhand dessen sich die im folgenden angestellten Überlegungen gut nachvollziehen lassen. Dieser Zustandsautomat stellt die Grundlage für die Berechnung des Energieverbrauchs eines Speichers in SDRAM-Technologie dar. Durch die Simulation eines Programms bei Ausführung aus einem SDRAM-Speicher lässt sich anhand dieses Automaten ein zustandsbasiertes Leistungsmodell ableiten. Die in Tab. 4.1 angegebenen Kenngrößen repräsentieren die in den jeweiligen Zuständen gemessenen Ströme und können aus dem Datenblatt eines Speichers bezogen werden. Da die Angaben zu konventionellem *single data rate*-SDRAM (SDR-SDRAM) von den Angaben zu einem DDR-SDRAM abweichen, werden im folgenden alle Überlegungen parallel angestellt und entsprechend mit dem Kürzel SDR bzw. DDR deklariert. Schwerpunkt dieser Arbeit stellen allerdings die SDR-SDRAM dar, so dass die Abbildungen sich prinzipiell auf diese Speicher beziehen.

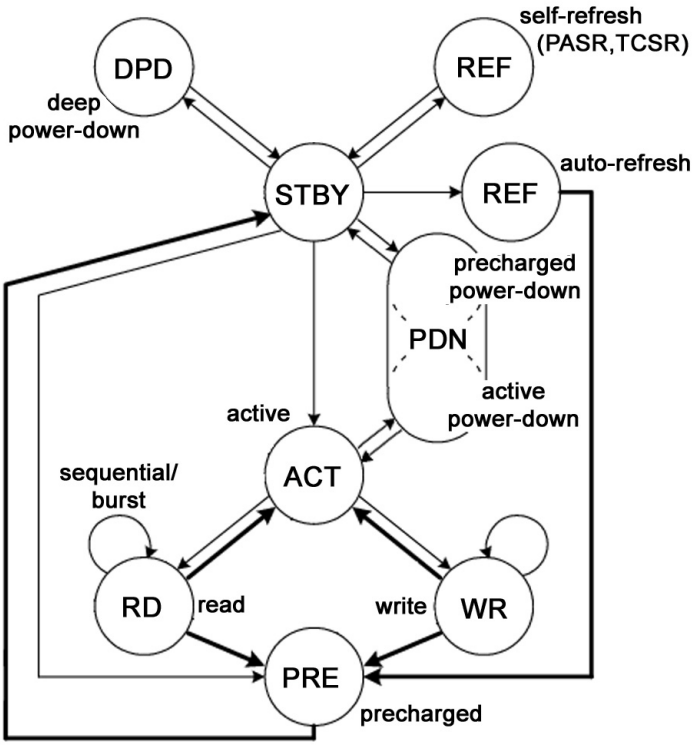


Abbildung 4.4: Vereinfachter Zustandsautomat eines SDR-SDRAM

Zustandsbedingte Leistung

Das Steuersignal *clock enable* (*CKE*) stellt die zentrale Ansteuerungskomponente dar. Wenn *CKE* eine '0' führt, sind die E/A-Puffer abgeschaltet und der Takteingang deaktiviert. Der Speicher befindet sich im *power-down state* (*PDN*). Erst durch einen Signalwechsel von CKE auf *high* werden Takteingang und Pufferschaltungen aktiviert und der Speicherzugriff ermöglicht. Der Zustand wechselt in den *standby state* (*STBY*).

Ein Speicherzugriff erfordert zunächst das Aktivieren einer Zeile in einer selektierten Bank. Durch den Befehl *ACTIVATE* wird das 'Öffnen' einer Bank, die Adressierung der gewünschten Zeile und anschließende Übertragung der Zeileninhalte an die Verstärker angestoßen. Der Speicher befindet sich dann im sogenannten *active state* (*ACT*). Der Stromverbrauch in diesem Zustand wird beim DDR-SDRAM durch die Kenngröße I_{DD3N} und beim SDRAM durch I_{DD3} (*active standby current*) angegeben. Wechselt der Speicher aus ACT in PDN wird die Leistung durch die Stromstärken I_{DD3P}(*active power-down*

Kenngröße	Symbol	SDRAM	DDR-SDRAM
active precharge current	I_{DD0}	-	×
active operating current	I_{DD1}	×	-
precharge power-down standby current	I_{DD2P}	-	×
idle standby current	I_{DD2F}	-	×
power-down standby current	I_{DD2}	×	-
active power-down standby current	I_{DD3P}	-	×
active standby current	I_{DD3N}	-	×
	I_{DD3}	×	-
read current	I_{DD4R}	-	×
write current	I_{DD4W}	-	×
operating current	I_{DD4}	×	-
auto refresh current	I_{DD5}	×	×

Tabelle 4.1: Verwendete Kenngrößen der Firma Micron

standby current) bzw. I_{DD2} (*power-down standby current*) bestimmt.
Die Leistungsaufnahmen in den jeweiligen Zuständen ergeben sich zu:

$$p_{DDR\_ACT\_PDN} = I_{DD3P} \cdot V_{DD} \tag{4.1}$$

$$p_{DDR\_ACT\_STBY} = I_{DD3N} \cdot V_{DD} \tag{4.2}$$

$$p_{SDR\_ACT\_PDN} = I_{DD2} \cdot V_{DD} \tag{4.3}$$

$$p_{SDR\_ACT\_STBY} = I_{DD3} \cdot V_{DD} \tag{4.4}$$

Die gewünschten Spalten einer Zeile können nun adressiert und die Inhalte am Verstärker abhängig vom Steuersignal WE ausgelesen oder überschrieben werden.

Der zu *ACTIVATE* korrespondierende Befehl *PRECHARGE* entspricht dem 'Schließen' einer Zeile. Die betroffene Zeile wird zurückgeschrieben und die Bitleitungen auf logisch '1' vorgeladen. Der Speicher befindet sich im Anschluss im *precharged state* (*PRE*). Der Stromverbrauch wird dann ebenfalls abhängig davon, ob Takteingang und E/A-Puffer aktiviert sind, durch die Kenngrößen I_{DD2P} (*precharge power-down standby current*) und I_{DD2F} (*idle standby current*) angegeben, so dass sich für die Leistungsaufnahme die

folgenden Gleichungen ergeben:

$$p_{DDR\_PRE\_PDN} = I_{DD2P} \cdot V_{DD} \tag{4.5}$$

$$p_{DDR\_PRE\_STBY} = I_{DD2F} \cdot V_{DD} \tag{4.6}$$

$$p_{SDR\_PRE\_PDN} = I_{DD2} \cdot V_{DD} \tag{4.7}$$

$$p_{SDR\_PRE\_STBY} = I_{DD3} \cdot V_{DD} \tag{4.8}$$

An den bisher aufgestellten Gleichungen erkennt man, dass beim SDRAM die Produkte für die Leistungen in den jeweiligen *power-down* und *standby states* identisch sind. Diese Feststellung erlaubt folgende Vereinfachungen:

$$p_{SDR\_PDN} = p_{SDR\_PRE\_PDN} = p_{SDR\_ACT\_PDN} \tag{4.9}$$

$$p_{SDR\_STBY} = p_{SDR\_PRE\_STBY} = p_{SDR\_ACT\_STBY} \tag{4.10}$$

Das Abschalten des Speichers (*deep power-down*, *sleep-mode*), verbunden mit vollständigem Verlust der Speicherinhalte, reduziert die Leistungsaufnahme auf ein Minimum, da folglich auch keine Leistung für den Refresh des Speichers anfällt. In den Datenblättern von Speichern, die diesen Zustand ermöglichen, wird hierzu ein äußerst schwacher Strom I_{ZZ} angegeben, der meist im Bereich von $10\mu A$ liegt.

$$p_{DPD} = I_{ZZ} \cdot V_{DD} \tag{4.11}$$

Aktivierungsbedingte Leistung

Aufgrund der durch die Technologie gegebenen Eigenschaften bilden *ACTIVATE* und *PRECHARGE* ein Befehlspaar, welches unabhängig vom Speicherzugriff erhebliche Aktivität durch die notwendige Befehlsdekodierung, Zeilenadressierung und Übertragung der Daten an die Verstärker auf dem Speicher verursacht.

Der mit I_{DD0} (*active precharge current*) angegebene Strom stellt die durchschnittlich gemessene Stromstärke für ein solches Befehlspaar, unter der Annahme einer Zeitspanne t_{RC} zwischen zwei Aktivierungen dar.

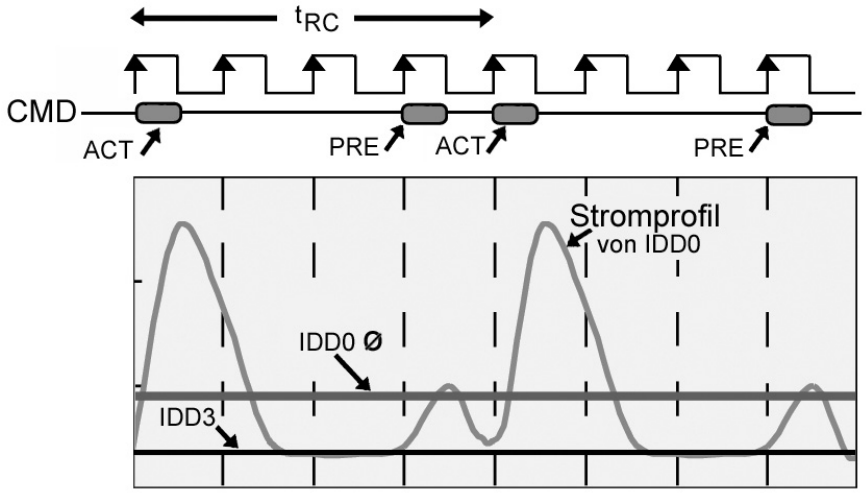


Abbildung 4.5: Stromprofil und Durchschnittsstrom IDD0 [Mic01]

Durch Subtraktion der fixen Stromanteile (s. Abb 4.5) erhält man:

$$p_{DDR\_ACT} = (I_{DD0} - I_{DD3N}) \cdot V_{DD} \tag{4.12}$$

Da I_{DD0} beim SDRAM nicht explizit angegeben wird, muss diese Größe für nachfolgende Berechnungen ermittelt werden. Der Strom I_{DD1} (*active operating current*) gibt beim SDRAM die durchschnittlich gemessene Stromstärke für zwei aufeinanderfolgende Lesezugriffe innerhalb einer Zeile an. Wird der Stromanteil für die beiden Leseoperationen (mit einer Dauer von zwei Taktzyklen) subtrahiert, erhält man

$$I_{DD0} = I_{DD1} - \frac{(I_{DD4} - I_{DD3}) \cdot 2 \cdot t_{CK}}{t_{RC}} \tag{4.13}$$

und somit

$$p_{SDR\_ACT} = (I_{DD0} - I_{DD3}) \cdot V_{DD} \tag{4.14}$$

Zugriffsbedingte Leistung

Nachdem eine Zeile geöffnet wurde, können die eigentlichen Speicherzugriffe stattfinden. Hierzu werden für den Lese- und Schreibzugriff die beiden Kenngrößen I_{DD4R} und I_{DD4W} (*read/write current*) eingeführt.
Da Speicherzugriffe zwischen zwei Aktivierungen stattfinden, in denen beim DDR-

SDRAM grundsätzlich der Strom I_{DD3N} angenommen wird, muss zur Berechnung der Leistung für die reine Lese-/Schreiboperation die Differenz zwischen I_{DD4R} bzw. I_{DD4W} und I_{DD3N} gebildet werden.

$$p_{DDR\_RD} = (I_{DD4R} - I_{DD3N}) \cdot V_{DD} \tag{4.15}$$

$$p_{DDR\_WR} = (I_{DD4W} - I_{DD3N}) \cdot V_{DD} \tag{4.16}$$

In den Datenblattangaben eines SDR-SDRAM wird häufig nur eine Stromstärke I_{DD4} (*operating current*) angegeben, die dann sowohl für den *write* als auch für den *read current* angenommen werden kann. Bei expliziter Angabe beider Kenngrößen wird man feststellen, dass beide Stromstärken zumeist identisch sind oder nur unerheblich voneinander abweichen.

$$p_{SDR\_RD} = p_{SDR\_WR} = (I_{DD4} - I_{DD3}) \cdot V_{DD} \tag{4.17}$$

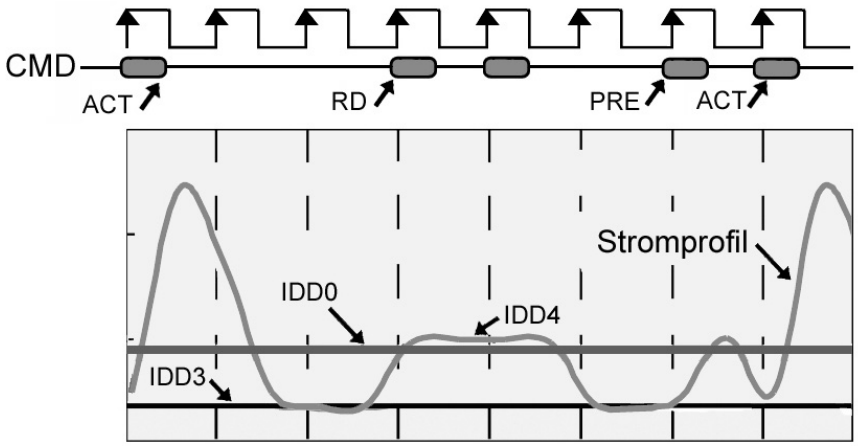


Abbildung 4.6: Stromprofil bei Speicherzugriff [Mic01]

Bei der Berechnung der Leistung für den lesenden Zugriff muss berücksichtigt werden, dass die Ausgangstreiber der Datenleitungen (*DQ*) freigeschaltet werden, da der Speicher die Leitungen auf den entsprechenden Wert treiben muss. Die Belastung der Ausgänge des SDRAM wird hierbei durch die kapazitive Last C_{LOAD} modelliert.

Multipliziert mit der Anzahl der Datenleitungen (abhängig von der Busbreite) und den Steuerleitungen (*data-strobe, DQS*) erhält man, abhängig von der Taktfrequenz *CK* und

der Spannung der Treiberschaltung V_{DQ}, die benötigte Leistungsaufnahme

$$p_{SDR\_DQ} = \frac{1}{2} \cdot C_{LOAD} \cdot (V_{DQ})^2 \cdot CK \cdot (\sum DQ + \sum DQS) \tag{4.18}$$

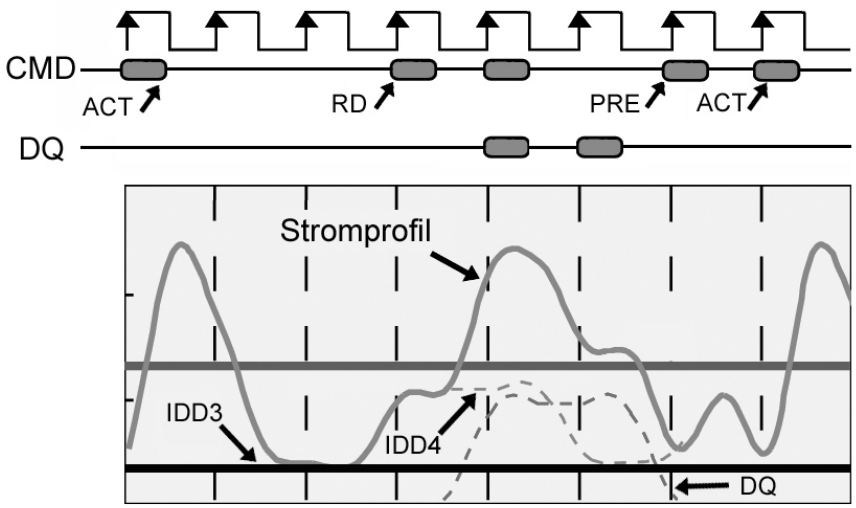


Abbildung 4.7: Stromprofil beim Lesezugriff mit Datenausgabe [Mic01]

Refreshbedingte Leistung

Jede Zeile muss in einem bestimmten Intervall neu geschrieben werden, um die Gültigkeit der Inhalte zu gewährleisten. Der Strom I_{DD5} (*auto-refresh current*), der über eine gewisse Zeitspanne im *power-down state* ohne Zugriffe auf den Speicher gemessen wird, stellt einen Durchschnittswert für diese Operation dar. Folglich ergibt sich für den kontinuierlich angenommenen *auto-refresh* ohne die fixen Anteile für den *power-down state*:

$$p_{DDR\_REF} = (I_{DD5} - I_{DD2P}) \cdot V_{DD} \tag{4.19}$$

$$p_{SDR\_REF} = (I_{DD5} - I_{DD2}) \cdot V_{DD} \tag{4.20}$$

Nachdem nun alle Einzelleistungen bestimmt worden sind, erfolgt die Berechnung der Gesamtleistung.

Berechnung der Gesamtleistung für den SDR-SDRAM

Die Berechnung der Gesamtleistung erfordert eine prozentuale Verteilung der Betriebszeit (Laufzeit des betrachteten Programms) auf die jeweiligen Speicherzustände. In dieser Arbeit werden anhand von Simulationen applikationsspezifische Profile erstellt. Hierzu werden die Zugriffe auf den SDRAM-Speicher protokolliert und unter Kenntnis des Zustandsautomaten die entsprechenden Zustände gewichtet. Nach Ablauf der Simulation können die während dieser gesammelten Informationen zur Energieberechnung verwertet werden. In anderen Ansätzen wird anhand adäquater Profile (*moderate usage, high-stress workload*) die Gesamtleistung geschätzt, wobei natürlich Abweichungen zum tatsächlichen Resultat entstehen können. Die Auswertung in dieser Arbeit ermöglicht eine sehr exakte Verteilung der Laufzeit t_{CPU} auf die einzelnen Zustände.
Folgende Auflistung gibt Aufschluss über benötigte Angaben für die Berechnung der Gesamtleistung:

- $BNK_{PRE}\%$=$(1 - BNK_{ACT}\%)$: prozentualer Anteil der Programmzyklen, in dem sich der Speicher im 'vorgeladenen' Zustand befindet
- $CKE_{LO\_PRE}\%$: prozentueller Anteil von $BNK_{PRE}\%$, in denen CKE eine '0' führt
- $CKE_{LO\_ACT}\%$: prozentueller Anteil von $1 - BNK_{PRE}\%$, in denen CKE eine '0' führt
- $RD\%/WR\%$: prozentueller Anteil der Programmzyklen, in denen Daten gelesen oder geschrieben werden

Weiterhin ist davon auszugehen, dass im realen Betrieb die Zeitspanne zwischen zwei Aktivierungen nicht stets minimal ist. Daher wird der Parameter n_{ACT} eingeführt, der die durchschnittliche Anzahl der Zyklen zwischen zwei *ACT*-Befehlen angibt. Dieser Wert variiert natürlich je nach Zugriffshäufigkeit auf den Speicher und wird ebenfalls für das jeweilige applikationsspezifische Profil bestimmt. Daraus ergibt sich:

$$p_{SDR\_ACT} = (I_{DD0} - I_{DD3}) \cdot \frac{t_{RC}}{n_{ACT} \cdot t_{CK}} \cdot V_{DD} = (I_{DD0} - I_{DD3}) \cdot \frac{t_{RC}}{t_{ACT}} \cdot V_{DD} \tag{4.21}$$

Die zuvor aufgestellten, mit p bezeichneten Gleichungen beziehen sich prinzipiell auf die im Datenblatt spezifizierte maximale Taktfrequenz des Speichers. Für einige Gleichungen müssen Anpassungen bezüglich der tatsächlichen Frequenz, mit der der Spei-

cher betrieben wird, vorgenommen werden (*frequency scaling*, f_s). Ausgenommen hiervon sind:

- $p_{SDR\_PRE\_PDN}$ und $p_{SDR\_ACT\_PDN}$ bzw. p_{SDR_PDN}, da CKE eine '0' führt und in diesen Zuständen kein Taktsignal anliegt
- p_{SDR_ACT}, da diese Leistung ausschließlich von der Zeitspanne t_{ACT} abhängig ist
- p_{SDR_REF}, da in der Berechnung von einem zeitlichen Intervall unabhängig vom Takt ausgegangen wird

Weiterhin werden *worst-case* Annahmen in Bezug auf die Betriebsspannung vorgenommen. Variiert die tatsächliche Betriebsspannung von der Spezifikation im Datenblatt sind weitere Anpassungen notwendig (*voltage scaling*,VDD_s). Ausgenommen hiervon ist die Leistung p_{DQ}, da diese durch die Spannung der Treiberschaltung V_{DQ} bestimmt wird.

$$f_s = \frac{f_{use}}{f_{spec}} \quad ; \quad VDD_s = \frac{(V_{DD_{use}})^2}{(V_{DD_{spec}})^2}$$

Die Vereinfachung in Gl.4.10 erlaubt für die Berechnung der Gesamtleistung eines SDR-SDRAM die Zusammenfassung von $CKE_{LO\_PRE}\%$ und $CKE_{LO\_ACT}\%$. $CKE_{LO}\%$ entspricht somit dem prozentuellen Anteil der Programmzyklen, in denen der Speicher im *power-down state* ist.

$$P_{SDR\_PDN} = p_{SDR\_PDN} \cdot CKE_{LO}\% \cdot VDD_s \tag{4.22}$$

$$P_{SDR\_STBY} = p_{SDR\_STBY} \cdot (1 - CKE_{LO}\%) \cdot VDD_s \cdot f_s \tag{4.23}$$

Für die Leistungen, die ausschließlich mit Lese- oder Schreiboperationen verbunden sind, erhält man anhand der Zugriffszyklen:

$$P_{SDR\_RD} = p_{SDR\_RD} \cdot RD\% \cdot VDD_s \cdot f_s \tag{4.24}$$

$$P_{SDR\_WR} = p_{SDR\_WR} \cdot WR\% \cdot VDD_s \cdot f_s \tag{4.25}$$

$$P_{SDR\_DQ} = p_{SDR\_DQ} \cdot RD\% \cdot f_s \tag{4.26}$$

Da p_{SDR_ACT} bereits durch t_{ACT} bewertet wurde und p_{SDR_REF} über die gesamte Laufzeit anfällt, muss an dieser Stelle gemäß der Vorüberlegungen lediglich das *voltage scaling*

berücksichtigt werden.

$$P_{SDR\_ACT} = p_{SDR\_ACT} \cdot VDD_s \quad (4.27)$$

$$P_{SDR\_REF} = p_{SDR\_REF} \cdot VDD_s \quad (4.28)$$

Durch Aufsummierung der Einzelleistungen erhält man die applikationsspezifische Gesamtleistung. Auf die explizite Angabe des SDR-Kürzels wird in der folgenden Gleichung verzichtet.

$$P_{SDR} = P_{PDN} + P_{STBY} + P_{ACT} + P_{WR} + P_{RD} + P_{DQ} + P_{REF} \quad (4.29)$$

Durch die geleistete Vorarbeit ergibt sich der Energieverbrauch des SDRAM durch Multiplikation mit der Laufzeit zu:

$$E_{SDR} = P_{SDR} \cdot t_{CPU} \quad (4.30)$$

4.2.2 Speichertiming

Nachdem nun beschrieben wurde, wie man anhand von Datenblattangaben die Gesamtleistung dynamischer Speicher berechnen kann, soll nun erläutert werden, welche Zugriffszeiten sich für die zugrundeliegenden Speicher ergeben.

Anhand der Darstellung des Speichertimings, welche gewöhnlicherweise in den Datenblättern zu den jeweiligen Speichern in Diagrammform vorliegt, wird die Integration der betrachteten Speicher in die Arbeitsumgebung vorgenommen.

Im folgenden wird nun als Einführung exemplarisch erläutert, wie die Speicherzugriffszeiten (in Taktzyklen) abhängig vom Takt variieren können. Die in Abb. 4.8 dargestellten Diagramme sind aus [Mic00] übernommen worden und visualisieren die Auswirkungen des Betriebs mit verschiedenen Taktfrequenzen auf die Zugriffszeit des Speichers.

Der im folgenden betrachtete Speicher ist auf den Betrieb mit einer Taktfrequenz von 125 MHz ausgelegt, dies entspricht einer Taktdauer von 8ns ($t_{CK} = 8ns$). Anhand des se-

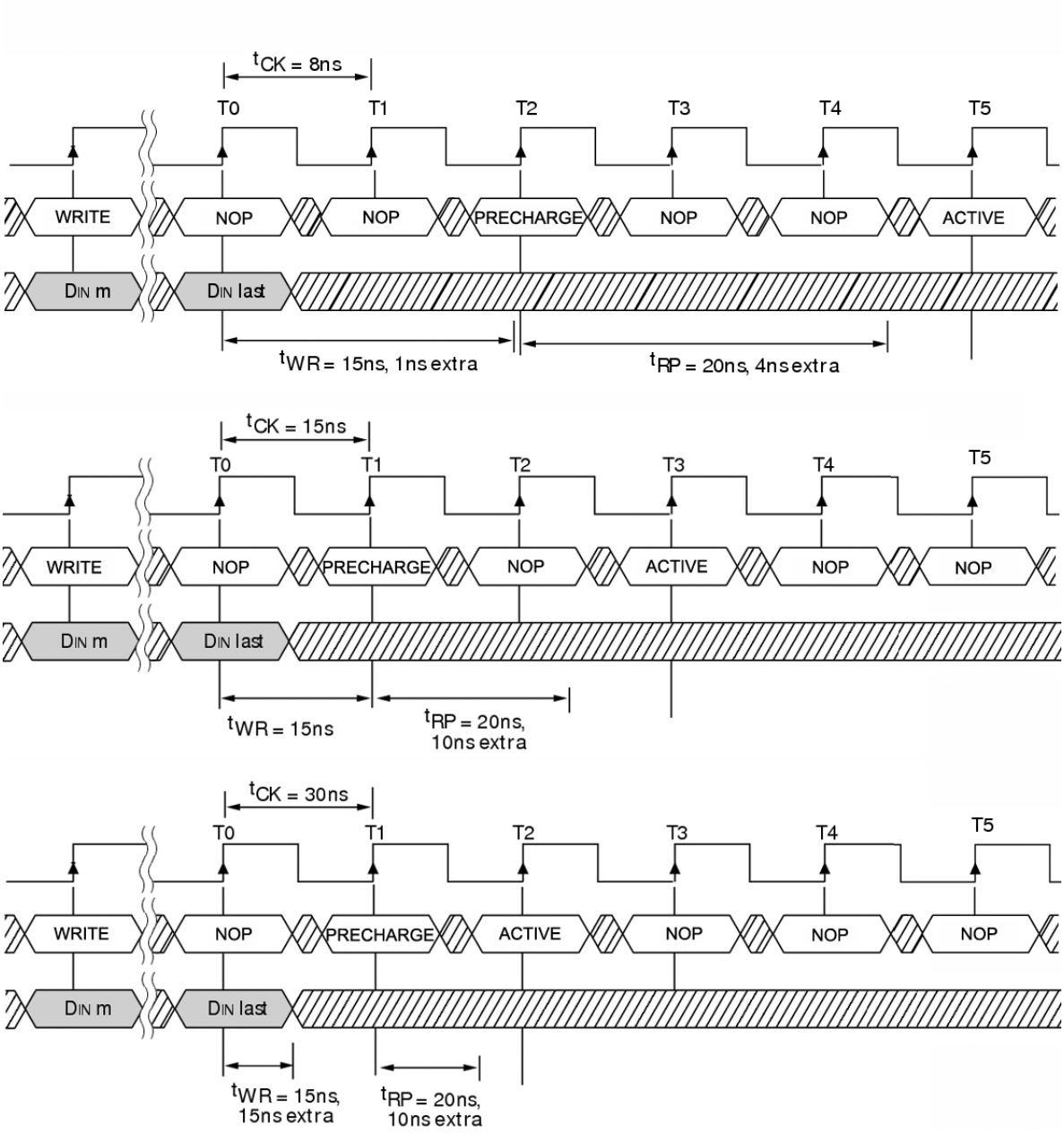


Abbildung 4.8: Write-Timing bei verschiedenen Frequenzen [Mic00]

quentiellen Schreibzugriffs auf diesen Speicher werden die Auswirkungen des Betriebs mit deutlich niedrigeren Frequenzen von 66 MHz ($t_{CK} = 15ns$) und 33 MHz ($t_{CK} = 30ns$) dargestellt.

In den Diagrammen sind weiterhin die zeitlichen Kenngrößen *write recovery time* ($t_{WR} =$

15*ns*) und *precharge command period* ($t_{RP} = 20ns$) eingetragen. Betrachtet man nun das oberste Diagramm der Abb. 4.8 erkennt man, dass bei $t_{CK} = 8ns$ insgesamt $T = 5$ Taktzyklen vom betrachteten Zeitpunkt (D_{INlast}) bis zur nächstmöglichen Speicheraktivierung vergehen. Im folgenden sei diese Zeitspanne mit $t_f(D_{INlast}, ACT)$ für $f = (33, 66, 125)$, bezeichnet. Somit ergibt sich für $t_{125}(D_{INlast}, ACT)$

$$t_{125}(D_{INlast}, ACT) = T \cdot t_{CK}(125) = 40ns \tag{4.31}$$

Berechnet man nun die minimale Zeitspanne $t_{min}(D_{INlast}, ACT)$ bis zur nächstmöglichen Aktivierung erhält man eine untere zeitliche Schranke für die folgenden Überlegungen.

$$t_{min}(D_{INlast}, ACT) = t_{WR} + t_{RP} = 35ns \tag{4.32}$$

Durch Bildung der Differenz aus $t_f(D_{INlast}, ACT)$ und $t_{min}(D_{INlast}, ACT)$ bestimmt man den zeitlichen Mehraufwand, welcher zugunsten der Zugriffgeschwindigkeit möglichst klein sein sollte.

$$t_{diff}(min, 125) = t_{125}(D_{INlast}, ACT) - t_{min}(D_{INlast}, ACT) = 5ns \tag{4.33}$$

Das mittlere Diagramm der Abb. 4.8 stellt das Timing bei einem 66 MHz schnellen Speicherbus dar. Konsequenterweise nimmt zwar die Anzahl der Takte bis zur nächstmöglichen Aktivierung ab, die Differenz zum Minimalwert aber zu.

$$t_{diff}(min, 66) = t_{66}(D_{INlast}, ACT) - t_{min}(D_{INlast}, ACT) = 10ns \tag{4.34}$$

Das unterste Diagramm der Gegenüberstellung veranschaulicht den betrachteten Sachverhalt für eine Taktfrequenz von 33 MHz. Auch hier reduziert sich zwar die Anzahl der Takte bis zur nächstmöglichen Aktivierung, die Zeit $t_{diff}(min, f)$ nimmt aber für die betrachteten Fälle das Maximum an.

$$t_{diff}(min, 33) = t_{33}(D_{INlast}, ACT) - t_{min}(D_{INlast}, ACT) = 25ns \tag{4.35}$$

Vorhergehende Berechnungen rechtfertigen die Argumentation, dass die betrachteten Speicher an einen adäquat schnellen Speicherbus angebunden sein müssen, um ihre Geschwindigkeitsvorteile ausnutzen zu können. Die Auswirkung der unterschiedlichen Taktfrequenzen auf die Zugriffszeit wird in Tab.4.2 auf Seite 59 anhand des Speichers

[Mic04c] demonstiert.

Im folgenden werden nun anhand von Datenblattauszügen [Mic04c] die Zugriffsarten *random* (*RND*) und *burst/sequential access* (*SEQ*) bei Betrieb des Speichers mit 100MHz analysiert.

Zugriff im *random access mode*

Der lesende Speicherzugriff, unter der Annahme eines Speichers mit einer Datenwortbreite von 16 Bit, erfordert zunächst das Aktivieren der betreffenden Zeile. Die zeitliche Verzögerung *active to read/write delay* wird mit t_{RCD} bezeichnet und beschreibt die Dauer des *RAS-to-CAS delay*, bevor die Zeilenadressierung gültig ist. Im Rahmen der Zeitmultiplexsteuerung folgt die Spaltenauswahl und die Übertragung der Zelleninhalte an die Leseverstärker, mit der in Abb. 4.9 eine speicherinterne *CAS-latency* von T_{CAS} Takten verbunden ist. Nach der Bereitstellung der Daten erfolgt die zuvor beschriebene *precharge command period* (t_{RP}), so dass eine nächste Aktivierung erfolgen kann und der aktuelle Lesezugriff als abgeschlossen gilt.

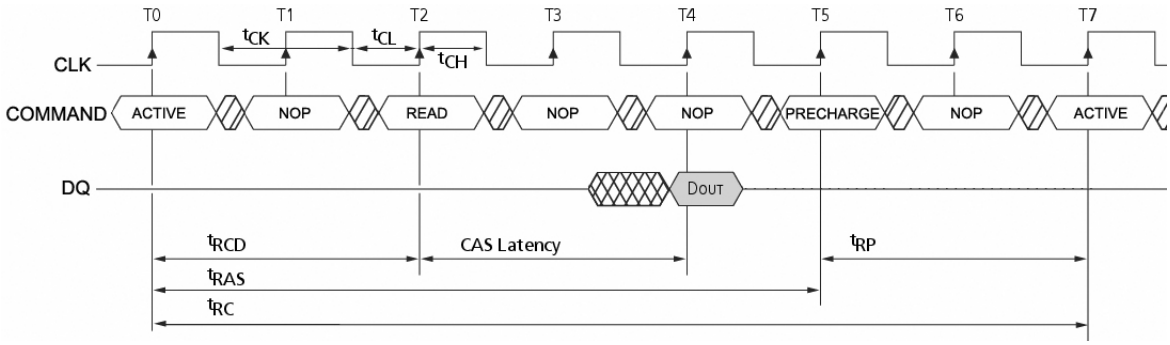


Abbildung 4.9: Random-Access Timing beim Lesezugriff [Mic04c]

Insgesamt $T_{SDR\_RND16(RD)}$ Taktzyklen sind für den lesenden Speicherzugriff im *random access mode* bei Annahme einer Taktdauer von t_{CK} notwendig. Für die Ausgabe eines 16-Bit Wortes werden hierbei T_{DOUT} Taktzyklen angenommen. Es wird an dieser Stelle davon ausgegangen, dass je zwei aufeinanderfolgende Lesezugriffe unterschiedliche Zeilen adressieren und dementsprechend für jeden Zugriff eine erneute Aktivierung

erforderlich wird.

$$T_{SDR\_RND16(RD)} = \left\lceil \frac{t_{RCD}}{t_{CK}} \right\rceil + T_{CAS} + T_{DOUT} + \left\lceil \frac{t_{RP}}{t_{CK}} \right\rceil \tag{4.36}$$

Für den schreibenden Speicherzugriff muss noch die *write recovery time* in die Gleichung für $T_{RND(WR)}$ aufgenommen werden. Bei t_{WR} handelt es sich um eine feste zeitliche Größe, die beginnend mit dem letzten D_{IN} angibt, wieviel Zeit bis zum *precharge command period* vergehen muss.

$$T_{SDR\_RND16(WR)} = \left\lceil \frac{t_{RCD}}{t_{CK}} \right\rceil + \left\lceil \frac{t_{WR}}{t_{CK}} \right\rceil + \left\lceil \frac{t_{RP}}{t_{CK}} \right\rceil \tag{4.37}$$

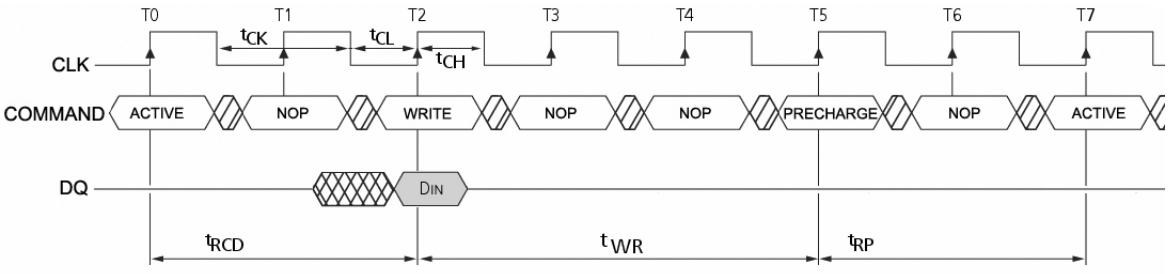


Abbildung 4.10: Random-Access Timing beim Schreibzugriff [Mic04c]

Zugriff im *burst mode*

Bei dem 32-Bit-Speicherzugriff muss berücksichtigt werden, dass das zweite 16-Bit Wort sequentiell gelesen werden kann, da Worte für gewöhnlich an zwei hintereinanderfolgenden Adressen abgelegt werden. Daraus resultiert, dass ein erneutes Aktivieren und der damit verbundene Mehraufwand an Zeit und Leistung eingespart werden kann, sofern der Speicher den Zugriff im *burst mode* unterstützt. Ein sequentieller Lesezugriff minimaler Länge wird in Abb. 4.11 dargestellt. Analog zum Speicherzugriff im *random access mode* wird eine Verzögerung von t_{RCD} angenommen, die in dieser Abbildung nicht eingezeichnet ist.

Im *burst-mode* werden aufeinanderfolgende Spaltenadressen anhand einer angelegten Basisadresse im Speicher durch interne Inkrementierung generiert, wodurch erhebliche

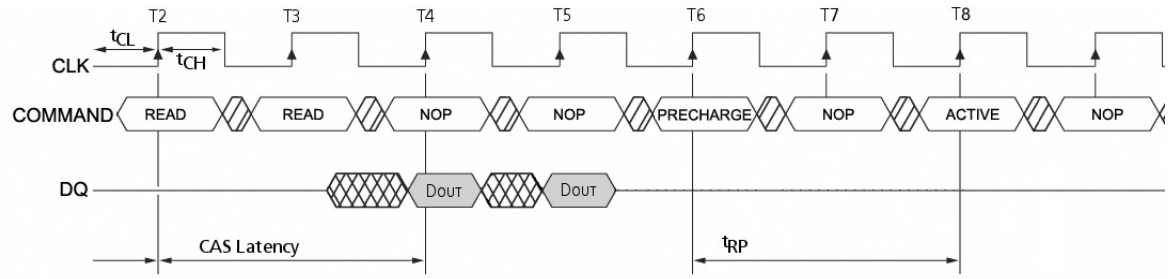


Abbildung 4.11: Burst-Mode Timing beim Lesezugriff [Mic04c]

Geschwindigkeitsvorteile entstehen. In Abb. 4.11 erkennt man, dass jedes sequentiell gelesene 16-Bit Wort zum nächsten Takt zur Verfügung steht. Unter der Annahme, dass T_{DOUT} einen Taktzyklus beansprucht, erhält man für den sequentiellen Zugriff auf ein 16-Bit Wort

$$T_{SDR\_SEQ16(RD)} = T_{DOUT} \tag{4.38}$$

Zusammenfassend ergibt sich für den lesenden 32-Bit-Speicherzugriff eine Dauer von

$$T_{SDR\_RND32(RD)} = T_{RND16(RD)} + T_{SEQ16(RD)} \tag{4.39}$$

Zyklen.

Analog erhält man für das sequentielle Schreiben eines 16-Bit Wortes

$$T_{SDR\_SEQ16(WR)} = T_{DIN} \tag{4.40}$$

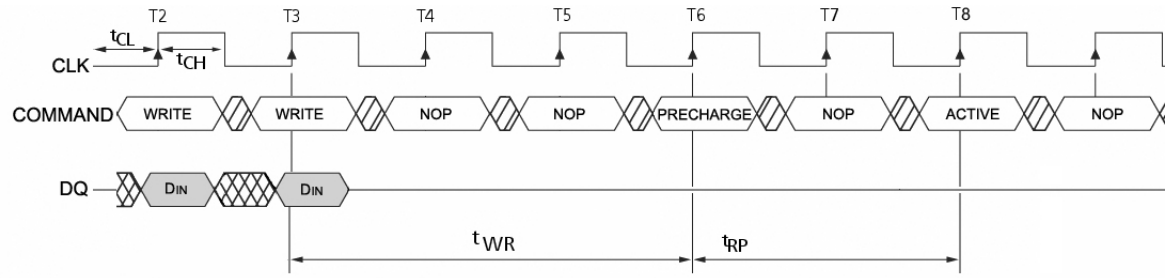


Abbildung 4.12: Burst-Mode Timing beim Schreibzugriff [Mic04c]

Die *burst length* eines Speichers in SDRAM-Technologie legt die Anzahl der Adressen innerhalb einer Zeile fest, auf die im *burst mode* zugegriffen werden kann. Je nach Speicherorganisation kann dieser Parameter variieren, prinzipiell gilt aber, dass die Anzahl

der Speicherzugriff im *burst mode* auf eine Zeile beschränkt ist, da der Zugriff auf eine andere Zeile mit einer erneuten Aktivierung verbunden ist.

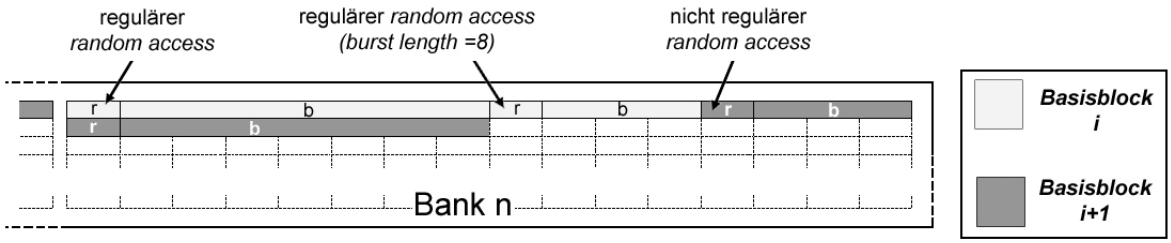


Abbildung 4.13: Speicherzugriffe im *burst mode*

Aus den bisherigen Erläuterungen geht nicht hervor, dass der Linker einen erheblichen Einfluss auf das Zugriffsschema eines Basisblocks ausübt. Die folgende Feststellung führt zu der Formulierung einer Regel, die für die in dieser Arbeit durchgeführte Bewertung von Speicherzugriffen notwendig ist. Abb. 4.13 verdeutlicht, dass die Zugriffe auf einen Basisblock abhängig von dessen Positionierung im Speicher variieren können. Die Basisblöcke i und $i+1$ haben gleiche Größe und sind prinzipiell identisch. Der Basisblock i liegt am Anfang einer Zeile, wodurch der Zugriff auf die erste Instruktion stets mit einer Zeilenaktivierung verbunden ist. Weiterhin wird durch die Größe des Basisblocks die für diesen Fall angenommene maximale *burst length* überschritten, so dass bei Ausführung insgesamt zwei Zugriffe im *random access mode* stattfinden werden.

Der Basisblock $i+1$ liegt im Speicher direkt hinter dem Basisblock i. Sollte während der Programmausführung Basisblock $i+1$ direkt nach Basisblock i ausgeführt werden, so wäre ein sequentieller Zugriff auf die erste Instruktion des Basisblocks i möglich. Erst nach der erneuten Zeilenaktivierung würde ein Zugriff im *random access* stattfinden. Dieser Sachverhalt führt möglicherweise zur Fehleinschätzung des Energieverbrauchs eines Basisblocks, falls dieser im Rahmen einer compilergesteuerten Energiereduktion an anderer Stelle im Speicher abgelegt wird. Aus diesem Grunde wird jeder Zugriff auf die erste Instruktion eines Basisblocks stets als *random access* bewertet.

Häufig besteht für den Benutzer die Möglichkeit, das *burst-mode timing* im System je

	$T_{SDR\_RND16(RD)}$		$T_{SDR\_RND32(RD)}$	
Kenngröße Micron 64MBit	CLK=33MHz t_{CK}=30ns	CLK=100MHz t_{CK}= 10ns	CLK=33MHz t_{CK}=30ns	CLK=100MHz t_{CK}= 10ns
t_{RP}	20 ns $\sim$ 1T	$\sim$ 2 T	$\sim$ 1 T	$\sim$ 2 T
$t_{RCD} = t_{RP}$	$\sim$ 1 T	$\sim$ 2 T	$\sim$ 1 T	$\sim$ 2 T
CAS Latency	1 T	2 T	1 T	2 T
DOUT	1 T	1 T	2 T	2 T
T_{TOT}	4 T	7 T	5 T	8 T
t_{TOT}	120 ns	70 ns	150 ns	80 ns

Tabelle 4.2: Zugriffszeit in Takten [Mic04c]

nach 'Qualität' des verwendeten Speichers zu konfigurieren. Ausgehend von einem *four-read burst* wird das *burst-mode timing* als Quadrupel $(T_{INIT}, T_B, T_B, T_B)$ angegeben. Die erste Angabe T_{INIT} gibt die initiale Verzögerung in Taktzyklen für den ersten Zugriff an und entspricht der Variablen T_{SDR_RND16}, unter der Annahme, dass diesem Zugriff die *precharge command period* zugeschrieben wird. Nach der initialen Verzögerung beanspruchen die weiteren Zugriffe jeweils T_B Taktzyklen. Diese Variable ergibt sich durch T_{SDR_SEQ16}. Ein typisches *burst-mode timing* unter Vernachlässigung der *precharge command period* wird in [Bäh02] mit (5,1,1,1) angegeben.

Energieverbrauch am Beispiel des *Micron Mobile SDRAM*

Nachdem nun alle erforderlichen Vorbereitungen für die Berechnung der Zugriffsenergie eines SDRAM abgeschlossen sind, soll nun am Beispiel des bereits in Tab.4.2 verwendeten Speichers die Energie für den Lesezugriff auf ein 16-Bit Wort berechnet werden. Die hierzu erforderlichen Kenngrößen sind in Tab. 4.3 aufgelistet [Mic04c]. Hierzu wird das zuvor aufgestellte leistungsbasierte Energiemodell ausgenutzt, indem ein Programm betrachtet wird, welches aus nur einem Speicherzugriff besteht. Die Speicherzustände lassen sich aus dem Stromprofil in Abb.4.14 ableiten.

Der Speicher ist an einen Bus mit einer Taktfrequenz von 100MHz angebunden und wird mit einer Spannung von 1,8V betrieben. Dies entspricht den im Datenblatt spezifizierten Angaben, so dass keine Skalierungen bezüglich der Frequenz oder Spannung vorgenommen werden müssen. Mit t_{CK}= 10ns und t_{RC} = 70ns als Referenzwert erhält

Stromstärke	[mA]	Leistung	[mW]
I_{DD1}	50	$\times$	-
I_{DD2}	0,15	p_{PDN}	0,27
I_{DD3}	35	p_{STBY}	63
I_{DD4}	80	p_{RD}/p_{WR}	81
I_{DD5}	2	p_{REF}	3,33

Tabelle 4.3: Verwendete Kenngrößen

man

$$I_{DD0} = 50mA - \frac{(80mA - 35mA) \cdot 2 \cdot 10ns}{70ns} = 38,75mA$$

Geht man von einem kontinuierlichen *random access* aus, so wird jeder Speicherzugriff nach 7 Zyklen abgeschlossen (s. Abb.4.14) und mit t_{ACT}= 70ns erhält man

$$P_{SDR\_ACT} = (38,75mA - 35mA) \cdot \frac{70ns}{70ns} \cdot 1,8V = 7,71mW$$

Der Takteingang des Speichers ist während des gesamten Zugriffs aktiviert, so dass folgende zugriffsunabhängigen Leistungen angenommen werden:

$$\begin{aligned} P_{SDR\_STBY} &= 35mA \cdot 1,8V = 63mW \\ P_{SDR\_REF} &= (2mA - 0,15mA) \cdot 1,8V = 3,33mW \end{aligned}$$

Der Lesezugriff wird anteilig auf den gesamten Zugriff skaliert. Da von den insgesamt 7 Takten lediglich für einen Takt der Strom I_{DD4} fließt, berechnet man

$$P_{SDR\_RD} = \frac{1}{7} \cdot 81mW = 11,57mW$$

Werden die Steuerleitungen vernachlässigt, erhält man bei einer Busbreite von 16 Bit und einer angenommenen Kapazität C_{LOAD} von 30 pF

$$p_{SDR\_DQ} = \frac{1}{2} \cdot 30pF \cdot (1,8V)^2 \cdot 10ns \cdot 16 = 77,8mW$$

Analog zum Lesezugriff erhält man durch Skalierung der Zeitdauer für die Datenausgabe auf die Gesamtzeit

$$P_{SDR\_DQ} = \frac{1}{7} \cdot p_{DQ} = 11,11mW$$

Die Gesamtleistung für den betrachteten Lesezugriff ergibt sich durch Aufsummierung

zu

$$P_{SDR\_RND16(RD)} = 7,71mW + 63mW + 11,57mW + 11,1mW + 3,33mW = 96,71mW$$

Vernachlässigt man die konstanten Leistungsanteile erhält man

$$P^*_{SDR\_RND16(RD)} = 7,71mW + 11,57mW + 11,1mW = 30,38mW$$

als ausschließlich durch den Zugriff bedingte Leistung.
Mit $T_{SDR\_RND16(RD)} = 7$ und t_{CK}= 10ns lässt sich der Energieverbrauch des betrachteten Zugriffs zu

$$\begin{aligned} E_{SDR\_RND16(RD)} &= 96,71mW \cdot 70ns = 6,77nJ \\ E^*_{SDR\_RND16(RD)} &= 30,38mW \cdot 70ns = 2,13nJ \end{aligned}$$

abschätzen.

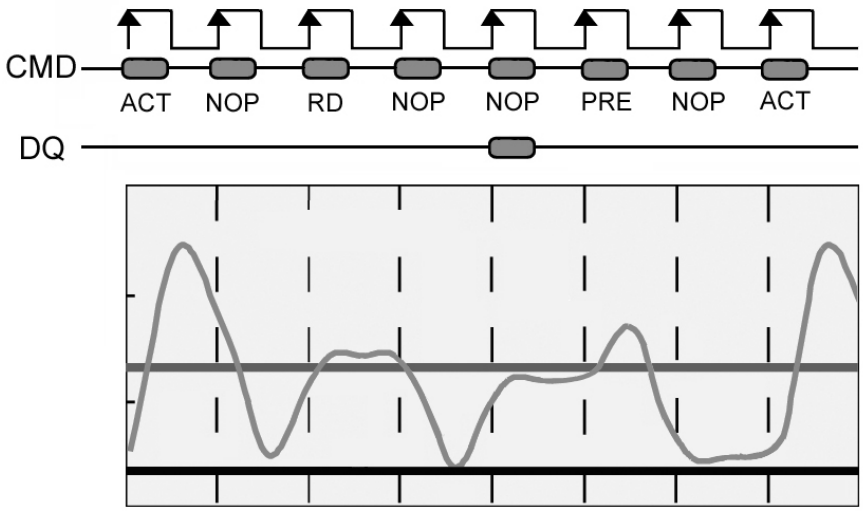


Abbildung 4.14: Stromprofil im *random access mode* [Mic01]

Analog erhält man für den Zugriff auf ein 32-Bit Wort bei gleicher Vorgehensweise

$$\begin{aligned} E_{SDR\_RND32(RD)} &= 112,78mW \cdot 80ns = 9,02nJ \\ E^*_{SDR\_RND32(RD)} &= 46,45mW \cdot 80ns = 3,72nJ \end{aligned}$$

Zieht man von der Leistung für den Zugriff auf ein 32-Bit Wort die Leistung für die Ausgangstreiberschaltung und die Refresh-Leistung

$$\begin{aligned} P_{SDR\_DQ} &= \frac{2}{8} \cdot p_{DQ} = 19,45mW \\ P_{SDR\_REF} &= (2mA - 0,15mA) \cdot 1,8V = 3,33mW \end{aligned}$$

ab, erhält man

$$E^{-}_{SDR\_RND32(RD)} = 112,78mW - 19,45mW - 3,33mW = 90mW$$

Durch Division mit der Betriebsspannung erhält man $I_{DD1} = 50mA$, dessen Stromprofil genau dem eines 32-Bit Speicherzugriffes entspricht (s. Tab.4.3).

Da für den sequentiellen Zugriff keine erneute Aktivierung benötigt wird, bestimmt man für den mit diesem Zugriff verbundenen Energieverbrauch mit $T_{DOUT} = 1$ zu

$$\begin{aligned} E_{SDR\_SEQ16(RD)} &= (77,8mW + 63mW + 81mW + 3,33mW) \cdot 10ns \\ &= E_{SDR\_RND32(RD)} - E_{SDR\_RND16(RD)} = 2,25nJ \\ E^{*}_{SDR\_SEQ16(RD)} &= 3,72nJ - 2,13nJ = 1,59nJ \end{aligned}$$

4.3 Energieverbrauch der CPU

Der Energieverbrauch der durch die Instruktionensverarbeitung in der CPU anfällt, lässt sich nach Tiwari durch zwei Ursachen beschreiben ([TMW94],[Tiw96]).

- *instruction basic costs*
- *inter-instruction costs*

Zum einen sind mit der Instruktionsverarbeitung Schaltaktivitäten im Prozessor verbunden. Der dadurch entstehende Energieverbrauch bildet die Basiskosten einer Instruktion. Die Messung aller Thumb-Instruktionen des ARM7TDMI hat Theokaridis im Rahmen seiner Diplomarbeit vorgenommen [The00]. Der Messwert wurde durch Wiederholen einer Instruktion in einer Schleife bestimmt.

Weiterhin finden im Prozessor zwischen zwei aufeinanderfolgenden Instruktionen Zustandswechsel durch unterschiedlich benötigte Funktionseinheiten statt, die im folgenden als *inter-instruction costs* bezeichnet werden. Ein weiterer Inter-Instruction-Effekt ergibt sich durch die Hamming-Distanz zweier aufeinanderfolgender Instruktionen auf den Busleitungen.
Zur Messung von *inter-instruction costs* hat Theokaridis verschiedene Befehlsklassen aufgestellt, die so ausgewählt wurden, dass die Instruktionen einer Klasse möglichst gleiche Funktionseinheiten des ARM7TDMI-Prozessors beanspruchen.

- *inter-instruction costs* innerhalb einer Befehlsklasse
- *inter-instruction costs* zwischen verschiedenen Befehlsklassen

Aus jeder Befehlsklasse wurden genau zwei Instruktionen ausgewählt und zu einem Instruktions-Tupel zusammengefasst. Weiterhin bilden je zwei Instruktionen aus unterschiedlichen Klassen ein Tupel. Die so ausgewählten Tupel wurden analog zur Bestimmung der Basiskosten durch wiederholtes Ausführen des Tupels in einer Schleife gemessen.
Zusammenfassend erhält man für den Energieverbrauch der CPU bei Abarbeitung der Instruktionen

$$E_{CPU} = \sum E_{BasicCost} + \sum E_{InterCost} \tag{4.41}$$

In diesen Messungen sind noch nicht die Energiekosten berücksichtigt worden, die durch den Speicherzugriff entstehen. Basierend auf dem grundlegenden Modell von Tiwari hat Steinke ein erweitertes Modell für den Energieverbrauch der CPU unter Berücksichtigung von Speicherzugriffsenergien aufgestellt ([SKWM01]). Das erweiterte Modell wird in dieser Form von dem im Anschluss beschriebenen *enProfiler* zur Energieabschätzung verwendet.

4.4 Arbeitsumgebung

Dieses Kapitel beschreibt die in dieser Arbeit verwendete Arbeits- und Simulationsumgebung, die sich aus mehreren Komponenten zusammensetzt (s. Abb.4.15).

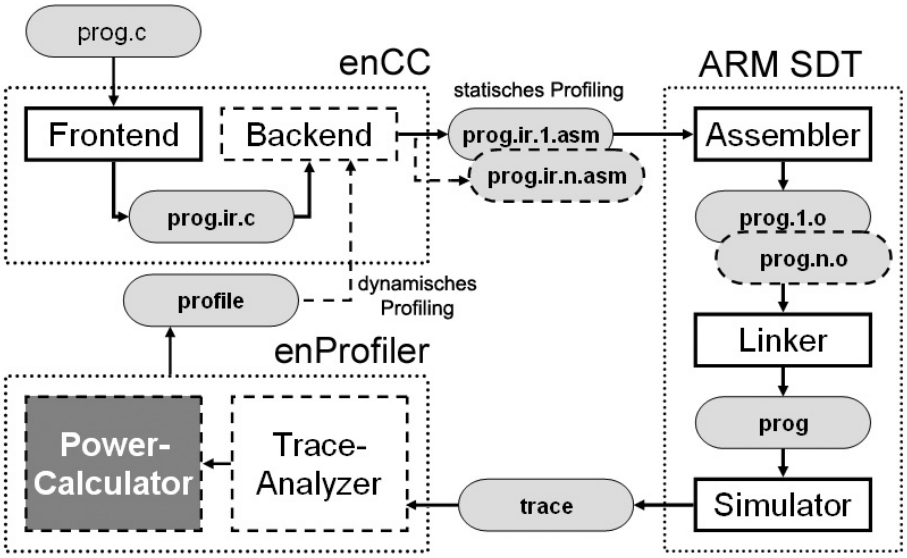


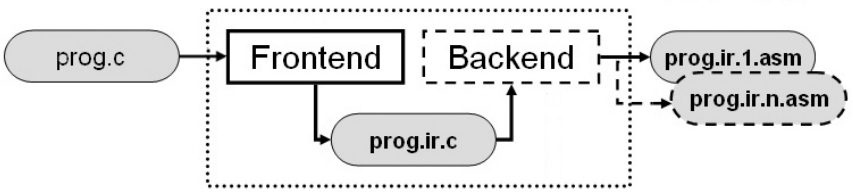
Abbildung 4.15: Arbeits- und Simulationsumgebung

Der verwendete *energy aware c-compiler* (*enCC*), der im Rahmen der Forschungsarbeit am LS 12 für Informatik an der Universität Dortmund entwickelt wurde, setzt sich aus einem ANSI-C Frontend (*LANCE*) und einem Backend für den Thumb-Befehlssatz des ARM7TDMI-Prozessors zusammen. Weiterhin steht mit dem *Software Development Toolkit* von ARM (*ARM SDT*) eine kommerzielle Software zur Verfügung, deren Komponenten Assembler, Linker und Simulator den Compiler in eine vollständige Entwicklungs- und Simulationsumgebung einbetten.

Mit dem ebenfalls am LS 12 entwickelten *enProfiler* lassen sich dann die vom Simulator erstellten Protokolle detailliert auswerten und Energieverbräuche abschätzen, so dass durch Kombination des Simulators mit dem *enProfiler* aufwendige Versuchsaufbauten und Messungen vermieden werden können.

Im folgenden soll nun der Prozess von der Übersetzung des Quellcodes bis zur Auswertung der Simulation sukzessiv erläutert werden.
Das Frontend *LANCE* erwartet ein ANSI-C Programm (*prog.c*) als Eingabe und erzeugt daraus die maschinenunabhängige Zwischendarstellung (*prog.ir.c*) als Austauschformat

für das Backend.
Hierbei können bereits Vereinfachungen der Zwischendarstellung vorgenommen werden. Optimierungen für *constant folding, constant propagation, copy propagation, CSE elimination, dead code elimination* und *jump optimization* sowie die iterative Anwendung aller Optimierungen sind in diesem Frontend implementiert.



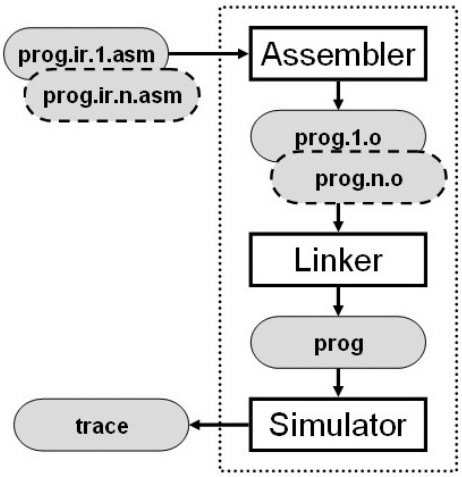
Das Backend des *enCC* erzeugt dann aus der Eingabe in der Phase der Codegenerierung die maschinenabhängige Zwischendarstellung (*prog.ir.asm*). Diese Phase besteht aus Codeselektion, Registerallokation, Instruktionsanordnung und verschiedenen Optimierungsmethoden, zu denen insbesondere die compilergestütze Optimierung von Speicherzugriffen auf heterogene Speicher zählt. Das Backend zerlegt hierzu die erzeugte Zwischendarstellung anhand des *profiling* entsprechend der verfügbaren Speicherpartitionen nach gewissen Optimierungskriterien:

$$prog.ir.asm \rightarrow (prog.ir.1.asm,...,prog.ir.n.asm)$$

Diese Zerlegung beinhaltet notwendige Adress- und Sprunganpassungen, die durch die Paritionierung anfallen.

Statisches Profiling: *Die Ausführungs- und Zugriffshäufigkeiten werden datenunabhängig bestimmt. Hierzu wird bei der Kontrollflussanalyse vereinfacht angenommen, dass jede Schleife zehnmal durchlaufen und jede Programmverzweigung mit einer Wahrscheinlichkeit von 50% stattfinden wird. Offensichtlich können diese vorhergesagten Häufigkeiten stark vom tatsächlichen Programmablauf abweichen.*

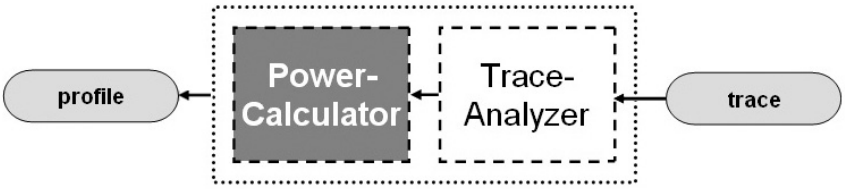
Der *enProfiler* analysiert den *trace* des Simulators zur Bestimmung der Laufzeit und des Energieverbrauchs und hält die Ausführungshäufigkeiten der Funktionen und Basisblöcke sowie den Zugriff auf globale Variablen in einem applikationsspezifischen Profil (*profile*) fest. Mit diesem Profil als zusätzliche Eingabe kann das Backend in einem zwei-



Der Assembler des *ARM SDT* übersetzt die einzelnen *asm*-Dateien zu Objektdateien (*prog.1.o,...,prog.n.o*), die anschließend vom Linker zu einem ausführbaren Programm (*prog*) zusammengefasst werden. Mit dem ARM-Simulator (ARMulator) kann dann die Ausführung des Programms simuliert und ein Protokoll (*trace*) aller ausgeführten Befehle und Speicherzugriffe generiert werden.

ten Durchlauf eine verbesserte Verteilung auf die zur Verfügung stehenden Speicherpartitionen vornehmen.

Dynamisches Profiling: *Das Progamm wird mit dem ARM-Simulator ausgeführt und anhand des Trace-Protokolls durch den enProfiler eine exakte Analyse der Ausführungs- und Zugriffshäufigkeiten erstellt. Vorteil dieser Analyse ist die hohe Genauigkeit für gegebene Eingabedatenmengen, wobei konsequenterweise bei Programmen mit Eingabedaten die ausgeführten Programmteile und deren Ausführunghäufigkeiten bei Modifikation der Eingabedatenmenge sehr unterschiedlich sein können. Zum anderen entsteht bei dieser Methode ein deutlicher Mehraufwand durch zweimaliges Übersetzen des Programms.*



An dieser Stelle wird das neue Modul *power calculator* (PWC) zur Auswertung des Energieverbrauchs von SDRAM-Speichern in den *enProfiler* integriert. Anhand der zuvor festgelegten Adressbereiche kann dann bei der Analyse ein Zugriff auf einen Speicher in SDRAM-Technologie als solcher erkannt werden. Die Parameter, die zur Bestimmung des Energieverbrauchs vom PWC verwertet werden, lassen sich anhand eines Auszuges aus einem generierten *trace* des ARMulator konkretisieren.

```
1.      MNR2O__ 00400000 B500
2.      MSR2O__ 00400002 2100
3.      MSR2O__ 00400004 1C02
4.      IT 00400000 b500 PUSH     {r14}
5.      MNW4___ 005FFFF8 004000DD
6.      MNR2O__ 00400006 2000
7.      IT 00400002 2100 MOV      r1,#0
8.      MSR2O__ 00400008 F000
9.      IT 00400004 1c02 MOV      r2,r0
```

Man erkennt am vorhergehenden Auszug, dass in jeder Zeile Speicherzugriffe (M) und Instruktionsausführungen (I) gesondert betrachtet werden. Bei Speicherzugriffen wird darauffolgend festgelegt, ob der Zugriff nicht-sequentiell (N) oder sequentiell (S) stattfindet. Prinzipiell lassen sich folgende Regeln für den sequentiellen Zugriff feststellen:

- Ein Speicherzugriff kann genau dann im *burst mode* stattfinden, wenn die aktuelle Speicheradresse einen Abstand von 2 oder 4 Byte zum vorhergehenden Speicherzugriff hat.
- Ein lesender Speicherzugriff, der auf einen schreibenden Speicherzugriff folgt, ist stets im *random access mode*.
- Ein schreibender Speicherzugriff, der auf einen lesenden Speicherzugriff folgt, ist stets im *random access mode*.
- Instruktionen, die einen Datenzugriff auf den Speicher verursachen, unterbrechen den *burst mode*. Die Zugriff auf die Folgeinstruktion erfolgt nach dem Datenzugriff im *random access mode*.
- Die Anzahl sequentieller Zugriffe sind durch die *burst length* begrenzt.
- Die erste Instruktion eines Basisblocks wird stets im *random access mode* gelesen (nicht-regulärer Zugriff im *random access mode*).

Durch letztere Regel wird sicherlich die Anzahl der möglichen sequentiellen Speicherzugriffe unterschätzt, der Energiebedarf des Speichers dafür aber nicht unterbewertet. Hierdurch wird das zuvor beschriebene Problem der Lage des Basisblocks im Speicher

	Parameter	PWC-Parameter
Zugriffsmodus	random/sequential 2 byte/4byte	n_{act}
Zugriffsart	read/write	*RD%/WR%*
Zugriffsintervall	-	CKE_{LO_ACT}%,CKE_{LO_PRE}

Tabelle 4.4: Parameter

umgangen.

Ebenfalls unterschieden wird, ob auf den Speicher geschrieben (W) oder vom Speicher gelesen (R) wird. Die Breite des Speicherzugriffs in Byte wird durch eine Nummerierung (2/4) deklariert. Weiterhin wird festgehalten, ob der Speicherzugriff einen *opcode fetch* (O) darstellt. Es folgt die hexadezimale Darstellung der Adresse des Speicherzugriffes, anhand der die zugehörige Speicherpartition identifiziert werden kann. Liegt diese Adresse im Adressbereich einer SDRAM-Partition, werden die zuvor erläuterten Parameter vom PWC festgehalten (s. Tab 4.4).

Weiterhin werden während der Analyse die CPU-Zyklen mitgezählt, so dass jeder Speicherzugriff zeitlich eindeutig zugeordnet werden kann. Unter Kenntnis dieser Informationen können dann die Intervalle festgelegt werden, in denen auf die jeweilige Speicherpartition (SDRAM-Partition) zugegriffen wird. Anhand dieser Intervalle wird nach Abschluss der Analyse eine prozentuale Verteilung der Laufzeit auf die verschiedenen Speicherzustände vorgenommen.

4.5 Ausnutzung der Funktionalität von Low-Power SDRAM

In vorhergehenden Diplomarbeiten am LS12 ergeben sich die Optimierungskriterien durch eine Modellvorstellung bestehend aus Speichertechnologien, die entweder keine oder vernachlässigbar kleine kontinuierliche, nichtzugriffsbedingte Energieverbräuche vorweisen. Die Erweiterung des Modells durch den PWC ermöglicht die Bewertung kontinuierlicher Energieverbräuche und eröffnet somit die Möglichkeit, diese in der

Formulierung der Optimierungsproblematik zu berücksichtigen. Im Anschluss an die folgenden Vorüberlegungen, wird nach einem kurzem Exkurs in die Optimierungstheorie die Formalisierung der Optimierungsproblematik vorgenommen.

4.5.1 Optimierungspotential

Wie bereits zuvor erläutert, bieten moderne SDRAM vielerlei Funktionalität, um einen energiesparsamen Betrieb zu ermöglichen. Da insbesondere die nichtzugriffsbedingten Leistungen einen erheblichen Einfluss auf die Gesamtleistung darstellen, zielen die im folgenden angestellten Überlegungen auf die geringe Leistungsaufnahme eines SDRAM auf den Moment ab, in dem der Speicher nicht genutzt wird.

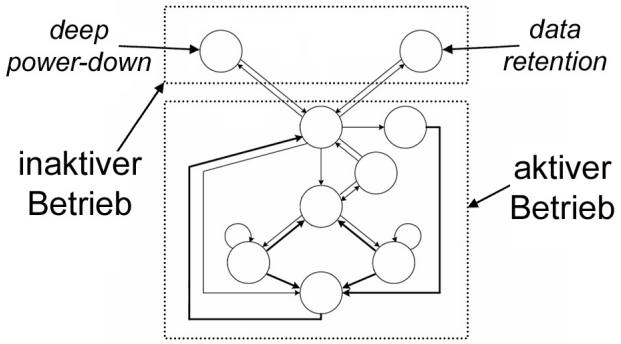


Abbildung 4.16: Schematische Darstellung des Zustandsautomaten

Abb. 4.16 zeigt nochmals schematisch den eingangs dargestellten Zustandsautomaten. Wie aus der Abbildung hervorgeht, lässt sich der Zustandsautomat in zwei wesentliche Betriebsmodi, den aktiven und inaktiven Betrieb, unterteilen. Der inaktive Betrieb wiederum teilt sich in den *sleep*- und *data retention mode* auf.

Je nachdem, ob auf den gegenwärtigen Speicherinhalt des Mobile SDRAM verzichtet werden kann, erlangt man durch den DPD maximale Energieeinsparungen. Betrachtet man nun das Optimierungpotential des DPD, stellt man fest, dass zwei Probleme für eine compilergestützte Optimierung entstehen:

- Der Wechsel aus dem aktiven Betrieb in den DPD führt zu einem vollständigen Datenverlust und erfordert somit das Auslagern des Speicherinhaltes in andere Partitionen, sofern der Verlust der Daten nicht in Kauf genommen werden kann.

- Der Wechsel aus dem DPD in den aktiven Betrieb erfordert einen erheblichen Zeitaufwand von typischerweise 150 μs (15000 Takte bei 100Mhz), in dem der Speicher nicht genutzt werden kann. Erst nach Ablauf dieser Vorlaufzeit können die Speicherinhalte wiederhergestellt werden, indem von anderen Partitionen kopiert wird.

Die Leistungsaufnahme im DPD ohne Refresh beträgt 0,018mW [Mic04c], die Leistungsaufnahme im Power-Down-Modus mit Refresh 3,6mW (s.Tab. 4.3), im Standby-Modus 66,63mW und im typischen Betrieb 99mW (s.Abb. 4.1).

Leistung	[mW]
p_{DPD}	0,018
$p_{PDN} + p_{REF}$	3,6
$p_{STBY} + p_{REF}$	66,63

Tabelle 4.5: Leistungsaufnahme eines SDRAM ohne Zugriff [Mic04c]

Geht man nun davon aus, dass für einen gewissen Zeitraum nicht auf den Speicher zugegriffen wird, lässt sich durch Wechsel in den Power-Down-Modus gegenüber dem Standby-Modus die Leistungsaufnahme um fast 95% reduzieren. Durch Wechsel in den DPD wäre sogar eine Reduktion von 99,9% möglich, wodurch eine weitere Verringerung von 5% gegenüber dem Power-Down-Modus entsteht. Diese Annahme bezieht sich aber auf ein Profil, in dem keine Speicherzugriffe stattfinden, so dass sich unter Annahme eines typischen Zugriffsprofils diese Einsparungen relativieren. Unter Berücksichtigung der beiden obigen Aufzählungspunkte wird von einer Ausnutzung des DPD durch das Kopieren von Codesektionen in den SDRAM zur Laufzeit abgesehen.

Soll der Speicherinhalt hingegen über einen gewissen Zeitraum im Nichtbetrieb erhalten bleiben (*data retention mode*), wird der Speicher in den *self-refresh mode* versetzt. Hier sorgen dann TCSR und PASR für einen adäquat geringen Leistungsverbrauch. Einerseits senkt TCSR mit abnehmender Temperatur auf dem Speicher die Refresh-Rate ab, andererseits beschränkt PASR den Refresh auf einzelne Regionen des Speichers. Die Motivation für die compilergestützte Ausnutzung des PASR wird in Abb.4.17 dargestellt.

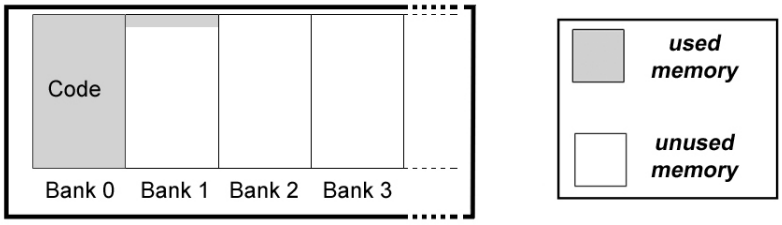


Abbildung 4.17: Codeverteilung auf Speicherbänke

Man erkennt, dass der Code eines Programms eine komplette Speicherbank und einen kleinen Anteil einer weiteren Bank beansprucht. Im *self-refresh mode* müssten aber die kompletten Speicherinhalte beider Bänke erhalten werden, so dass ein Optimierungskriterium für den Compiler die möglichst vollständige Nutzung einzelner Bänke darstellten könnte. Unter der Annahme, dass dem Compiler mehrere Speicher für die Programmverteilung zur Verfügung stehen, könnte es sich somit als sinnvoll erweisen, eine Aufteilung so vorzunehmen, dass lediglich eine Speicherbank des SDRAM genutzt wird. Da allerdings alle am LS12 zur Verfügung stehenden *benchmarks* nicht annähernd die Kapazität der kleinstmöglich konfigurierbaren Region ausschöpfen, wird dieser Fall nicht weiter betrachtet.

Generell bieten Optimierungen, die auf den nichtaktiven Betrieb während der Programmausführung abzielen, kein Potential für die Berücksichtigung in den Folgeüberlegungen.

Im aktiven Betrieb leitet die Kombination des Taktsignals CKE auf *'low'* mit einem NOP-Befehl den Power-Down Modus ein, sofern währenddessen keine Zugriffe auf den Speicher stattfinden. Je nachdem ob alle Bänke des Speichers sich im Zustand *idle* befinden, spricht man vom *precharge power-down*, andernfalls vom *active power-down*. Im *precharge power-down* werden die Ein- und Ausgabepuffer des Speichers abgeschaltet, wodurch maximale Einsparungen des Energieverbrauchs im Standby-Betrieb möglich sind. Das Verlassen dieses Zustands wird durch Kombination des Taktsignals CKE auf 'high' und einen NOP-Befehl veranlasst.

In Abb. 4.18 ist das Speichertiming für den Zustandswechsel in den *power-down state* dargestellt. Fasst man die Taktzyklen für den Wechsel in diesen Zustand und das Ver-

lassen zusammen, erhält man den in Taktzyklen ausgedrückten *overhead* T_{OH}. Offensichtlich reichen insgesamt $T_{OH} = 3$ Taktzyklen für diese Vorgänge aus. Wie in Abb. 4.18 erkennbar, befindet sich der Speicher für die Phasen des Übergangs im *standby-state*.

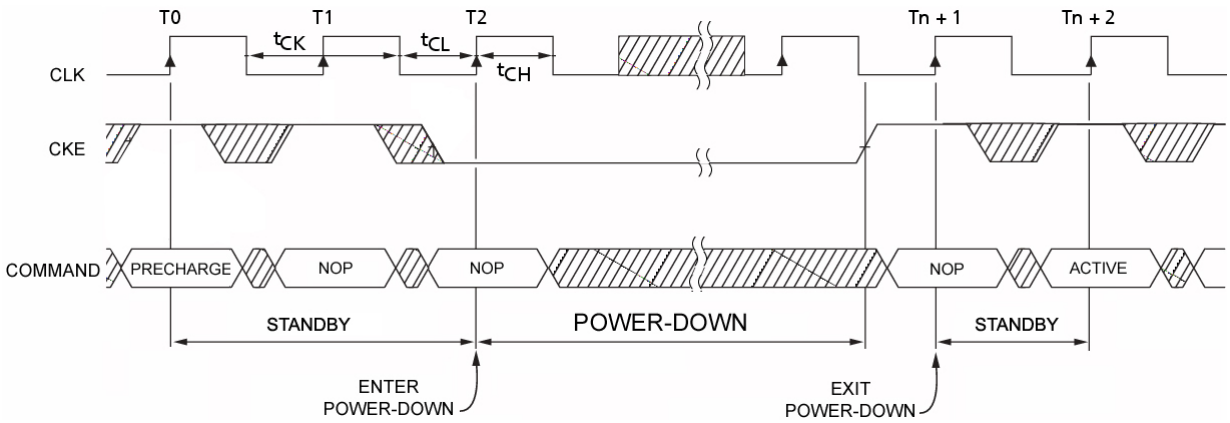


Abbildung 4.18: Power-Down Timing

4.5.2 Motivation

Die Möglichkeit für eine compilergesteuerte Optimierung hinsichtlich des *power-down state* wird im folgenden exemplarisch motiviert. In Abb. 4.19 ist der zeitliche Verlauf eines Programms bei Ausführung aller Instruktionen aus dem Hauptspeicher abgebildet.

Das Programm besteht aus den vier Basisblöcken A,B,C und D. Als vereinfachende Annahme wird davon ausgegangen, dass der instruktionsbedingte Energieverbrauch eines jeden Basisblockes identisch ist. Weiterhin hat jeder Basisblock gleiche Größe, lediglich die Ausführungshäufigkeiten unterscheiden sich.
Dem Compiler steht zur Verteilung der Basisblöcke nebst dem Hauptspeicher in SDRAM-Technologie zusätzlich ein Scratchpad-Speicher zur Verfügung, der unter obigen Annahmen drei Basisblöcke aufnehmen kann. Ohne Angabe der zeitlichen und elektrischen Kenngrößen beider Speicher wird angenommen, dass die Ausführung eines Basisblockes aus dem Scratchpad-Speicher eine kürzere Ausführungszeit und einen deutlich geringeren Energieverbrauch vorweist.

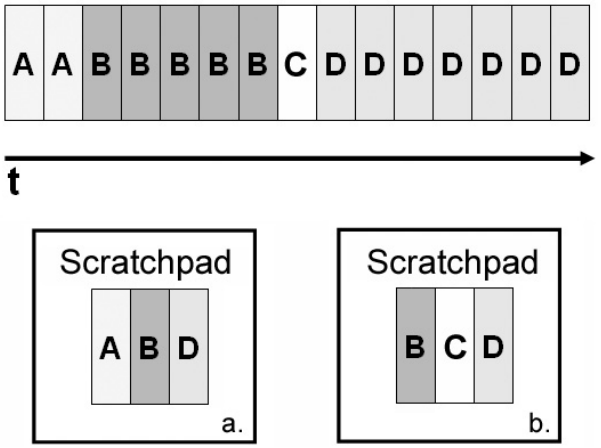


Abbildung 4.19: Verschiedene Aufteilungsmöglichkeiten

Unter ausschließlicher Berücksichtigung des zugriffsbedingten Energieverbrauchs wird durch eine Verteilung der Basisblöcke wie in Abb. 4.19a dargestellt die größte Einsparung erzielt.
Alternativ könnte der Compiler auch eine Verteilung vornehmen, wie sie in Abb. 4.19b vorgeschlagen wird. Es fällt auf, dass durch diese Wahl hinsichtlich der Ausführungszeit und des zugriffsbedingten Energieverbrauchs gegenüber der ersten Variante Nachteile in Kauf genommen werden müssten, da der Basisblock A aufgrund seiner zweifachen Ausführung einen höheren Nutzen im Scratchpad-Speicher vorweist.

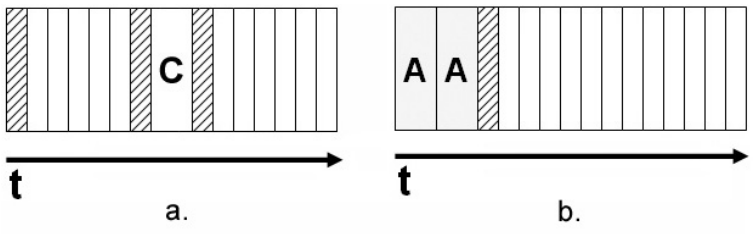


Abbildung 4.20: Power-Down Phasen

Betrachtet man aber nun den zeitlichen Programmverlauf der beiden Varianten aus Sicht des Hauptspeichers in Abb.4.20, so erkennt man unterschiedliches Potential für die Ausnutzung des *power-down state*. Geht man davon aus, dass der Hauptspeicher sich zu Beginn der Ausführung nicht im *power-down state* befindet, soll in Abb.4.20a zunächst in diesen Zustand gewechselt werden. Der schraffierte Bereich stellt jeweils

die Zeitpunkte für den Zustandswechsel dar. Für die einmalige Ausführung des Basisblockes C wird ein Verlassen dieses Zustands für die Zeilenaktivierung gemäß Abb.4.18 notwendig. Im Anschluss kann für die verbleibende Zeit der Ausführung wieder in den *power-down state* gewechselt werden.
Anhand der Abb.4.20b lässt sich der Vorteil der alternativen Variante der Verteilung erkennen. Nach Ausführung des Basisblocks A kann der Hauptspeicher in den *power-down state* versetzt werden und in diesem Zustand verbleiben.

Die im folgenden formulierte Optimierung soll entgegen bisheriger Optimierungen auf die Minimierung der nichtzugriffsbedingten Energie abzielen. Offensichtlich wird die Realisierung dieses Ziels durch eine kurze Laufzeit gefördert, da über die gesamte Zeit ein konstanter Energieverbrauch anfällt. Andererseits regen Zustandswechsel im Speicher den Energieverbrauch an, so dass es sich wiederum als sinnvoll erweisen kann, unter Akzeptanz einer höheren Laufzeit die Phase im *power-down state* auszudehnen.

4.5.3 Vorbereitungen

In den Vorüberlegungen wurde angenommen, dass während der Ausführung eines Basisblocks aus dem Scratchpad unter Ausnutzung des *power-down state* kein Zugriff auf den Hauptspeicher erfolgt. Diese Annahme stellt sich in den folgenden zwei Situationen als nicht zutreffend heraus:

- Zugriffe auf den *stack* unterbrechen den *power-down state*, falls dieser sich im Hauptspeicher befindet,
- und Zugriffe eines Basisblocks auf globale Variablen, die im Hauptspeicher liegen, unterbrechen ebenfalls den *power-down state*.

Zur Behandlung der ersten Situation wird der *stack* in einen Scratchpad-Speicher mit einer Kapazität von 128 Byte ausgelagert. Letztere Situation muss in der nachfolgenden Formulierung der Optimierungsmethode beachtet werden, da sonst möglicherweise fälschliche Annahmen über den Energieverbrauch getroffen werden.

Um die Optimierungsmethode zu validieren wird eine Simulationsumgebung verwendet wie sie in [Hel04] beschrieben wird. Das Speichersystem besteht neben dem Hauptspeicher aus Scratchpad-Speichern, deren Zugriffsenergien mit dem *CACTI*-Energiemodell [BSL+01],[BSL+02] berechnet werden.
Der Energieverbrauch des Hauptspeichers in SDRAM-Technologie wird mit dem PWC bewertet.

Bei dem betrachteten Hauptspeicher handelt es sich um einen mit einer Spannung von $3,3V$ betriebenen Mobile SDRAM mit einer Speicherkapazität von $16x8$ MBit. Die elektrischen und zeitlichen Kenngrößen wurden aus dem zugehörigen Datenblatt [Mic04a] bezogen. Der Speicher wird mit dem Systemtakt von 33 MHz betrieben, die entsprechenden Speicherzugriffszyklen sind in Tab. 4.2 aufgeführt.

Um die Vergleichbarkeit der Ergebnisse mit der Optimierungsmethode aus [Hel04] zu gewährleisten, wird der Speicherzugriff im *burst-mode* in den folgenden Betrachtungen ausgeschlossen.

4.5.4 Integer Linear Programming

An dieser Stelle wird zunächst ein kurzer Exkurs in die lineare Optimierung vorgenommen, die im folgenden zur Formulierung der Optimierungsmethode verwendet wird.

Die lineare Optimierung ist ein Spezialfall der Optimierungstheorie, der sich dadurch auszeichnet, dass sowohl die Zielfunktion als auch die Nebenbedingungen durch lineare mathematische Beziehungen ausgedrückt werden können [Weg01]. Im Rahmen der Optimierungstheorie wird dann versucht, zu einem vorgegebenen eindeutig formulierten Problem eine optimale Lösung zu finden. Dazu müssen Kriterien aufgestellt sein, die eine Entscheidung erlauben, ob eine Problemlösung im Vergleich zu allen anderen denkbaren Lösungen als optimal angesehen werden kann. Für eine solche Lösung wird gefordert, dass sie in endlich vielen Schritten ermittelt werden kann oder zumindest an eine optimale Lösung angenähert werden kann.

Sei f die zu optimierende lineare Zielfunktion, wobei o.B.d.A das Optimierungsziel durch die Minimierung dieser Funktion gegeben sei, dann lautet die Standardform dieser Problemklasse

$$f(x_1,...,x_n) = c_1x_1 + ... + c_jx_j + ... + c_nx_n \rightarrow min$$

unter den linearen Nebenbedingungen

$$a_{i1}x_1 + ... + a_{ij}x_j + a_{in}x_n \geq b_i \text{ für } 1 \leq i \leq m$$

wobei die Variablen x_j einer Nichtnegativitätsbedingung unterliegen

$$x_j \geq 0 \text{ für } 1 \leq j \leq n$$

Oftmals wird bei linearen Optimierungsaufgaben zusätzlich die Ganzzahligkeit der Entscheidungsvariablen in der Gesamtlösung gefordert *(Integer Linear Programming)* (ILP). Wird diese Bedingung auf $x_j \in \{0,1\}$ eingeschränkt, spricht man von Problemen der binären Optimierung, zu denen auch die in dieser Arbeit formulierten Optimierungsaufgaben gehören.

Das Software-Paket CPLEX [Ilo] dient unter anderem zum Lösen von Aufgaben aus dem Bereich des ILP. Verschiedene optimierte Verfahren werden angeboten, die besonders zum Lösen großer Probleme mit verhältnismässig kurzer Rechenzeit geeignet sind. Diese Verfahren werden auch als Bibliotheken angeboten und können damit auch in eigenen Programmen genutzt werden.

4.5.5 Formalisierung der Optimierungsproblematik

In den folgenden Betrachtungen seien die Mengen aller Basisblöcke $\{b_1, ..., b_\eta\}$ und globalen Variablen $\{g_1, ..., g_\mu\}$ des zu optimierenden Programms gegeben durch

$$\begin{aligned} B &:= \{b_1, ..., b_\eta\} \\ G &:= \{g_1, ..., g_\mu\}. \end{aligned}$$

Diese beiden Menge bilden dann durch Vereinigung die Menge der Programmobjekte

$$O := B \cup G = \{o_1, ..., o_n\}, \text{ mit } n = \eta + \mu. \tag{4.42}$$

Die Größe eines Programmobjektes o_i wird mit $Size(o_i)$ bezeichnet.

Die einmalige Ausführungsdauer eines Basisblocks $b_i \in B \subseteq O$ aus dem Hauptspeicher sei im folgenden mit $T_{off}(o_i)$ Taktzyklen angegeben. In dieser Ausführungsdauer ist der Zugriff auf globale Variablen enthalten. Der nichtzugriffsbedingte Energieverbrauch dieser Ausführung berechnet sich zu:

$$E_{off}(o_i) = (p_{SDR\_STBY} + p_{SDR\_REF}) \cdot t_{CK} \cdot T_{off}(o_i). \tag{4.43}$$

Offensichtlich benötigt der schnellere Scratchpad-Speicher für diese Ausführung die geringere Anzahl an Taktzyklen $T_{on}(o_i)$. Wird ein Programmobjekt o_i aus dem Scratchpad-Speicher ausgeführt, besteht für diese Dauer für den Hauptspeicher die Möglichkeit in den energiesparenden *power-down state* zu wechseln, sofern das Objekt nicht auf globale Variablen zugreift, die im Hauptspeicher abgelegt wurden. Bezogen auf den Hauptspeicher fällt ein mit dem Zustandswechsel T_{OH} verbundener Energieverbrauch an, der durch die Leistungsaufnahme im *standby-state* bestimmt wird. Für die restliche Ausführungsdauer $T_{on}(o_i) - T_{OH}$ befindet sich der Hauptspeicher im *power-down state* (s. Abb. 4.21).
Mit

$$t_{on}(o_i) = t_{CK} \cdot T_{on}(o_i) \tag{4.44}$$

$$t_{OH} = t_{CK} \cdot T_{OH} \tag{4.45}$$

erhält man im Hauptspeicher den Energieverbrauch $E_{on}(o_i)$ bei Ausführung des Objekts $b_i \in B \subseteq O$ aus dem Scratchpad-Speicher

$$\begin{aligned} E_{on}(o_i) &= E_{OH} + p_{SDR\_REF} \cdot t_{on}(o_i) + p_{SDR\_PDN} \cdot (t_{on}(o_i) - t_{OH}) \\ \text{mit} \quad E_{OH} &= p_{SDR\_STBY} \cdot t_{OH} \end{aligned}$$

unter der Bedingung, dass o_i nicht auf globale Variablen zugreift.

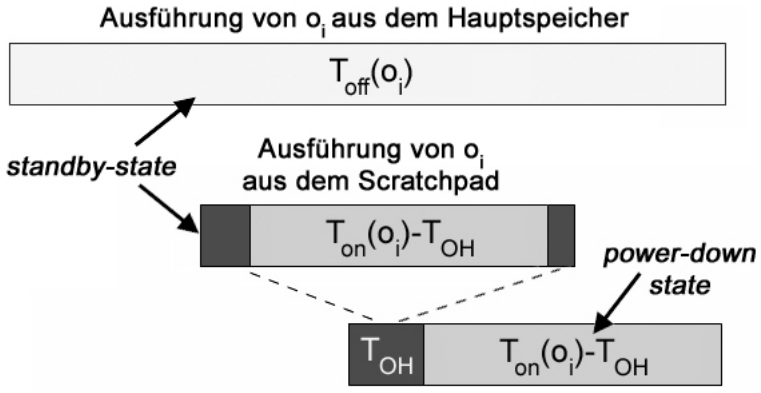


Abbildung 4.21: Zustandswechsel im SDRAM bei Scratchpad-Ausführung

Anhand des dynamischen Profilings werden dann die Zugriffshäufigkeiten auf die Programmobjekte $o_i \in B$ mit $v(o_i)$ bewertet.

Der CFG enthält Kanten zwischen zwei beliebigen Basisblöcken $b_h = o_h$ und $b_i = o_i$, die unmittelbar hintereinander ausgeführt werden können. Diese Kanten werden anhand der Ausführungshäufigkeiten mit $e(o_h, o_i)$ bewertet.

Weiterhin ist die Anzahl der Zugriffe eines Basisblocks $b_i = o_i$ auf eine Variable $v_g = o_g$ durch das Profiling bekannt. Diese Beziehung wird ebenfalls durch Kanten modelliert und entsprechend der Anzahl der Zugriffe mit $e(o_i, o_g)$ bewertet. Unter der Annahme, dass g Zugriffe auf beliebige globale Variablen (im SDRAM) bei der einmaligen Ausführung von o_i aus dem Scratchpad-Speicher stattfinden, ergeben sich bei konser-

vativer Abschätzung, g weitere Zustandswechsel für den SDRAM (s. Abb.4.22).

$$t_{DAT} = g \cdot t_{CK} \cdot (T_{DAT} + T_{OH}) \quad (4.46)$$

$$E_{DAT} = p_{SDR\_STBY} \cdot t_{DAT} \quad (4.47)$$

Abhängig vom Typ der globalen Variablen unterscheidet man T_{DAT} in

$$T_{DAT} = \begin{cases} T_{SDR\_RND16(RD)}, & \text{falls 8 oder 16-Bit Variable.} \\ T_{SDR\_RND32RD}, & \text{falls 32-Bit Variable.} \end{cases}$$

Man erhält

$$E_{on}(o_i) = E_{OH} + E_{DAT} + p_{SDR\_REF} \cdot t_{on}(o_i) + p_{SDR\_PDN} \cdot (t_{on}(o_i) - t_{OH} - t_{DAT}). \quad (4.48)$$

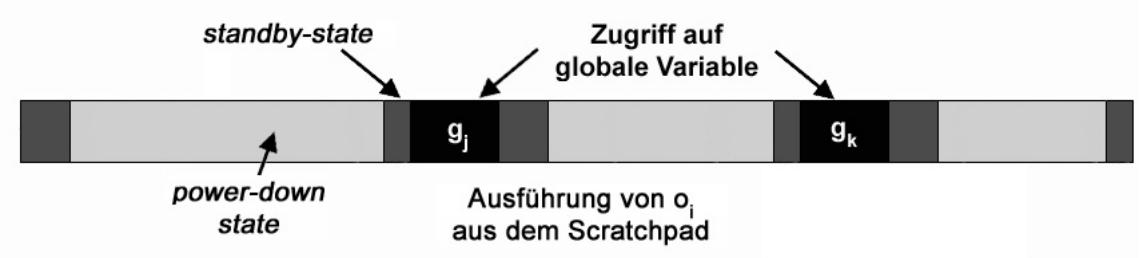


Abbildung 4.22: Basisblock mit Zugriff auf globale Variablen

Bisher wurde für jedes Programmobjekt bei dessen Ausführung aus dem Scratchpad-Speicher der Energieverbrauch E_{OH} für den Zustandswechsel berechnet. Folgt allerdings auf ein Objekt o_i die Ausführung von o_k, welches ebenfalls im Scratchpad-Speicher abgelegt wurde, kann der zuvor aufgeschlagene Energieverbrauch dem Objekt o_k gutgeschrieben werden, da sich der Hauptspeicher bereits im *power-down state* befindet (s. Abb 4.23). Diese Gutschrift sei im folgenden mit ${E_{OH}}^{+}$ bezeichnet.

$$\begin{aligned} E_{on}(o_k) &= p_{SDR\_STBY} \cdot t_{OH} + p_{SDR\_REF} \cdot t_{on}(o_k) + p_{SDR\_PDN} \cdot (t_{on}(o_k) - t_{OH}) \\ &- \underbrace{(p_{SDR\_STBY} - p_{SDR\_PDN}) \cdot t_{OH}}_{{E_{OH}}^+} \\ &= (p_{SDR\_REF} + p_{SDR\_PDN}) \cdot t_{on}(o_k) \end{aligned}$$

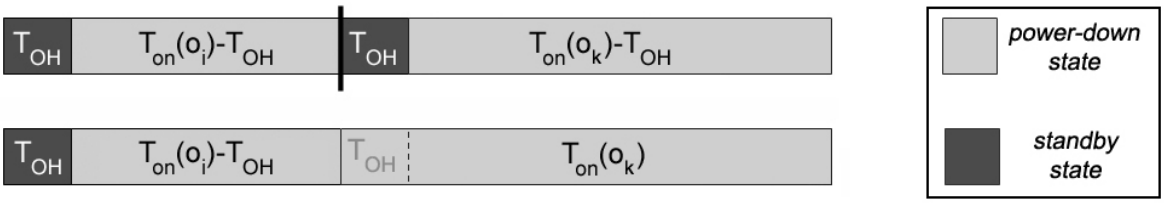


Abbildung 4.23: Gutschrift bei gemeinsamer Verschiebung in den Scratchpad

Analog erhält ein Basisblock o_i mit Zugriff auf globale Variablen bei Ausführung aus dem Scratchpad einen Energieanteil ${E_{DAT}}^+$ gutgeschrieben, falls die globalen Variablen mit dem Basisblock gemeinsam in den Scratchpad verschoben werden. Unter der Annahme, dass ein Basisblock einmalig auf eine globale Variable g_i zugreift, berechnet sich der Rückgewinn zu:

$${E_{DAT}}^+ = (p_{SDR\_STBY} - p_{SDR\_PDN}) \cdot t_{CK} \cdot (T_{DAT} + T_{OH}). \tag{4.49}$$

Entscheidungsvariablen

Für die Formulierung der Optimierungsproblematik ist noch die Festlegung von binären Entscheidungsvariablen erforderlich, die für die Generierung von erlaubten Lösungen vom *ILP-Solver* gesetzt werden können. Mit $o_i \in O$:

$$x_{on}(o_i) = \begin{cases} 1, & o_i \text{ wird in Scratchpad-Speicher abgelegt} \\ 0, & \text{sonst} \end{cases} \tag{4.50}$$

Analog gilt diese Fallunterscheidung für $x_{off}(o_i)$.
Die binären Entscheidungsvariablen, die angeben, ob der Kontrollfluss entlang einer

Kante verlaufen kann, ohne den Scratchpad-Speicher zu verlassen, sind für die Berechnung der Gutschrift E^+_{OH} und E^+_{DAT} erforderlich. Weiterhin wird durch diese Entscheidungsvariablen die Einsparung an Speicherplatz im Scratchpad-Speicher durch Wegfall des *long jumps* bewertet (vgl. [Hel04]).

$$x(o_h, o_i) = \begin{cases} 1, & o_h, o_i \text{ werden in Scratchpad-Speicher abgelegt} \\ 0, & \text{sonst} \end{cases} \tag{4.51}$$

Nebenbedingungen

Die erste Nebenbedingung fordert, dass jedes Objekt in genau einer Partition abgelegt wird:

$$\forall\, i \text{ mit } 1 \leq i \leq n : x_{on}(o_i) + x_{off}(o_i) = 1. \tag{4.52}$$

Die Entscheidungsvariablen der Kanten dürfen offensichtlich nur dann gesetzt werden, wenn auch die zugehörigen Programmobjekte in den Scratchpad-Speicher abgelegt werden. Diese Nebenbedingung wird wie folgt erreicht:

$$\forall\, h, i \text{ mit } 1 \leq i \leq n,\ 1 \leq h \leq n : x_{on}(o_h) + x_{on}(o_i) - 2\, x(o_h, o_i) \geq 0. \tag{4.53}$$

Die Kapazität der betrachteten Scratchpad-Speicher ist im Verhältnis zum Hauptspeicher sehr gering und bietet daher meist nur Platz für wenige Programmobjekte, da andernfalls die Lösung des zugrundeliegenden Problems trivial wäre. Die Einhaltung der begrenzten Kapazität $Size(on)$ des Scratchpad-Speichers stellt ebenfalls eine Nebenbedingung dar:

$$\forall\, i \text{ mit } 1 \leq i \leq n : \sum x_{on}(o_i) \cdot Size(o_i) \leq Size(on). \tag{4.54}$$

Anhand der getroffenen Vorbereitungen kann nun das Minimierungsproblem für den Fall, dass zur Lösung eine Scratchpad-Partition zur Verfügung steht, formuliert werden. Zur besseren Lesbarkeit der folgenden Gleichung wird auszugsweise die Interpretation einzelner Variablen wiederholt.

$$\begin{aligned} v(o_i) &= \text{Anzahl der Ausführungen des Basisblocks } o_i \\ e(o_h, o_i) &= \text{Anzahl der aufeinanderfolgenden Ausführungen von} \\ &\quad \text{Basisblock } o_h \text{ und } o_i \\ e(o_i, o_k) &= \text{Anzahl der Zugriffe von } o_i \text{ auf die globale Variable } o_k \end{aligned}$$

Minimiere:

$$\sum_{i=1}^{n} v(o_i) \cdot E_{on}(o_i) \cdot x_{on}(o_i) + v(o_i) \cdot E_{off}(o_i) \cdot x_{off}(o_i)$$
$$- \sum_{i=1}^{n} \left(\sum_{h \in B} e(o_h, o_i) \cdot {E_{OH}}^{+} \cdot x(o_h, o_i) - \sum_{k \in G} e(o_i, o_k) \cdot {E_{DAT}}^{+} \cdot x(o_i, o_k) \right) \tag{4.55}$$

unter zuvorgenannten Nebenbedingungen.

Nutzung partitionierter Speicher

In Anlehnung an [Hel04] wird nun das Vorhandensein mehrerer Scratchpad-Partitionen angenommen. Hierzu müssen einige Anpassungen der vorhergehenden Formulierung vorgenommen werden, die sich auf die Entscheidungsvariablen und Nebenbedingungen auswirken. Unter der Annahme von m zur Verfügung stehenden Scratchpad-Partitionen ergibt sich für die Entscheidungsvariablen die Entscheidungsmatrix

$$X = \begin{pmatrix} x(o_{1,1}) & \cdots & x(o_{1,m}) \\ \vdots & \ddots & \vdots \\ x(o_{n,1}) & \cdots & x(o_{n,m}) \end{pmatrix} \tag{4.56}$$

$$x(o_{i,j}) = \begin{cases} 1, & o_i \text{ wird in Scratchpad-Speicher } j \text{ abgelegt} \\ 0, & \text{sonst.} \end{cases} \tag{4.57}$$

Analog ergibt sich für die Entscheidungsvariablen der Kanten:

$$x(o_{h,j}, o_{i,j}) = \begin{cases} 1, & o_h, o_i \text{ werden in Scratchpad-Speicher } j \text{ abgelegt} \\ 0, & \text{sonst.} \end{cases} \tag{4.58}$$

Die Anpassungen der Nebenbedingungen ergeben sich aus den Überlegungen für einen Scratchpad-Speicher. Man erhält

- $\forall\, i$ mit $1 \leq i \leq n : \sum_{j=1}^{m} x(o_{i,j}) + x_{off}(o_i) = 1$
- $\forall\, h, i$ mit $1 \leq i \leq n,\ 1 \leq h \leq n$ und $\forall\, j$ mit $1 \leq j \leq m : x(o_{h,j}) + x(o_{i,j}) - 2x(o_{h,j}, o_{i,j}) \geq 0$
- $\forall\, j$ mit $1 \leq j \leq m : \sum_{i=1}^{n} x(o_{i,j}) \cdot Size(o_i) \leq Size(on_j)$

Minimiere:

$$\sum_{i=1}^{n}\sum_{j=1}^{m} v(o_i)\cdot E_{on}(o_i)\cdot x(o_{i,j}) + \sum_{i=1}^{n} v(o_i)\cdot E_{off}(o_i)\cdot x_{off}(o_i)$$
$$-\sum_{i=1}^{n}\sum_{h\in B} e(o_h,o_i)\cdot {E_{OH}}^{+}\cdot x(o_{h,j},o_{i,j}) + \sum_{k\in G} e(o_i,o_k)\cdot {E_{DAT}}^{+}\cdot x(o_{i,j},o_{k,j}) \qquad (4.59)$$

unter obengenannten Nebenbedingungen.

Die Darstellung der Ergebnisse und Auswertung dieser Optimierungsmethode erfolgt in Kap.6.

Kapitel 5

SDRAM- und Flash-basierte Speichertechnologien

Ein geringer Energiebedarf trotz anspruchsvoller Multimedia-Merkmale bestimmt derzeit die Entwicklung von mobilen Endgeräten. Unter diesen Rahmenbedingungen ergeben sich zusätzliche Anforderungen, die sich durch umfangreiche Applikationen und den dadurch notwendigen großen Hauptspeicher ergeben. Um diesen Anforderungen gerecht zu werden, bestehen gängige Systeme aus der Kombination verschiedener Speichertechnologien, die jeweils ihre Vorteile in das System einfließen lassen.
Flash-Speicher dienen hierbei als *boot block* und garantieren als Permanentspeicher die Erhaltung der Daten. Flüchtige Speicher in DRAM-Technologie stellen aufgrund ihrer schnellen Zugriffs- und Übertragungsgeschwindigkeiten die Hauptspeicher-Komponente. Da mittlerweile Flash-Speicher einen relativ schnellen *random access* erlauben, stellen sie neben ihren sonstigen Verwendungszwecken eine mögliche Konkurrenz als Instruktionsspeicher zum SDRAM dar.

Im folgenden wird zunächst ein Überblick über mögliche Kombinationen dieser Speicher erarbeitet, anschließend mit den Eigenschaften der betrachteten Arbeitsumgebung verglichen und eine Anpassung vorgenommen. Des Weiteren wird an dieser Stelle untersucht, inwiefern eine compilergesteuerte Nutzung der vorhandenen Speicher implementiert werden kann.

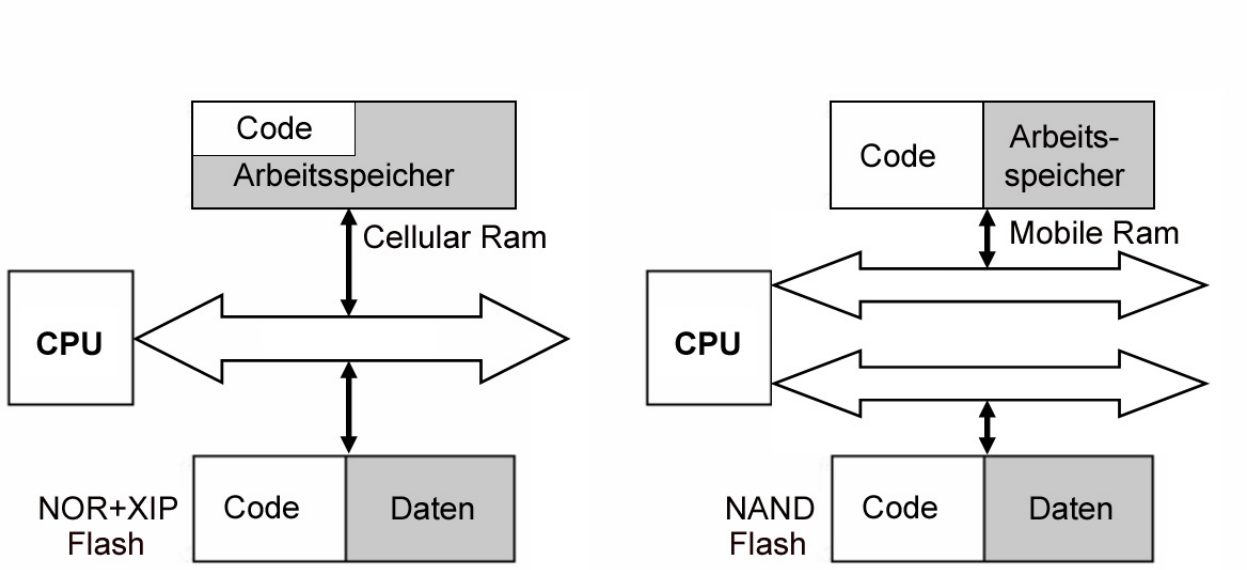


Abbildung 5.1: Alternative Speicherkonfigurationen [Der03]

5.1 Flash-Speicher - Architekturen und Konzepte

Flash-Speicher wird in eingebetteten Systemen als Daten- und Instruktionsspeicher verwendet, wobei gewisse Flash-Speicher hierzu das Konzept des *eXecute in place* (XIP) anbieten, welches dem Prozessor die Möglichkeit eröffnet, Instruktionen direkt aus dem Flash-Speicher zu laden und auszuführen. Durch dieses Konzept kann die Speicherkapazität der im System befindlichen Speicherkomponenten deutlich reduziert werden, indem beispielsweise das Betriebssystem (OS) und die Anwenderprogramme aus dem Flash-Speicher heraus abgearbeitet werden. Der für das OS und die Applikationen benötigte Hauptspeicher wird so reduziert und die Leistungsaufnahme im Vergleich zu anderen Konzepten erheblich vermindert.

Alternativ dazu werden bei *store and download*-Architekturen (SND) Codesektionen von einem nichtflüchtigen Speicher auf einen flüchtigen Speicher kopiert und dann von diesem ausgeführt. Mit Flash-Speichern in NOR-Technologie lassen sich beide Ansätze realisieren, während Flash-Speicher in NAND-Technologie lediglich das SND-Konzept

verfolgen, welches auch häufig als *code shadowing* [Der03] bezeichnet wird. Bei der Realisierung von SND-Konzepten unterscheidet [Cam04] folgende Varianten:

- *fully shadowed code* - die komplette Codesektion wird beim Booten in den Hauptspeicher kopiert, mit dem Nachteil einen Großteil des Speicherplatzes fest zu reservieren
- *demand paged code* - einzelne Codesektionen (Programme) werden auf Anfrage in den Hauptspeicher kopiert, wodurch Speicherplatz eingespart werden kann. Allerdings ist mit diesem Ansatz ein dynamischer Kopiervorgang verbunden, der einen zeitlichen Mehraufwand erzwingt.
- *application paged code* - applikationsspezifische Codesektionen (Programmteile) werden auf Anfrage in den Hauptspeicher kopiert, wodurch maximale Einsparungen bei der Reservierung von Speicherplatz erzielt werden können. Die Nachteile des *demand paged code* werden bei diesem Verfahren allerdings deutlich verstärkt und die Komplexität erheblich gesteigert.

Wie bereits in Kap.2 erläutert, unterscheiden sich die beiden Flash-Technologien durch ihre Zugriffsmöglichkeiten und im Wesentlichen durch ihre Zugriffsgeschwindigkeiten. Bei der NAND-Technologie wird die Speichermatrix durch serielle Anordnung der Transistoren organisiert, während bei der NOR-Technologie die Speicherzellen parallel angeordnet werden. Insbesondere letztere Form der Organisation ermöglicht einen schnellen *random access* (s. Tab. 5.1) und somit XIP-Fähigkeit. Ein Nachteil dieser Technologie ist allerdings ist die sehr geringe Schreib- und Löschgeschwindigkeit.

Speicher	Energie	Zugriffszeit
NOR-Flash	33,6 nJ (4 Worte)	210 ns
NAND-Flash	1,18 μJ (512 Byte)	35,8 μs

Tabelle 5.1: Lesezugriff in Flash-Architekturen [CL03]

Die NAND-Technologie hingegen bietet zwar hohe Zellendichte und Kapazität in Verbindung mit schnellem Schreiben und Löschen, der lesende Zugriff erfolgt aber nur seitenweise mit einer initialen Verzögerung von 10 bis 25 μs bevor die Speicherinhalte zunächst im *data/spare-register* verfügbar sind. Erst dann können die Daten sequentiell

aus dem Register ausgelesen werden, so dass ein effizienter *random access mode* nicht unterstützt wird [LPL+04]. Weiterhin werden zur Programmausführung spezielle *memory technology driver* (MTD) notwendig, dessen Implementierung und Integration in das System eine deutliche Steigerung der Anforderungen an Hard- und Software zur Folge hat.

Durch Verwendung von XIP besteht, wie bereits angemerkt, die Möglichkeit, ein großes Energieeinsparpotential auszuschöpfen. Einerseits entfällt der Mehraufwand, der durch das Kopieren beim *code shadowing* entsteht, andererseits wird weniger Hauptspeicher benötigt, wodurch die Größe dieser Speicher reduziert werden kann oder aber mehr Platz für die Speicherung von Daten zur Verfügung steht. Weiterhin ergibt sich durch XIP eine verbesserte Möglichkeit zur Ausnutzung von Funktionalitäten wie PASR und TCSR im Hauptspeicher.

5.2 Energieverbrauch von Flash-Speichern

Um die Integration von Flash-Speichern in die Arbeitsumgebung vornehmen zu können, wird im folgenden die Erläuterung des Energieverbrauchs anhand der Herleitung der erforderlichen Gleichungen durchgeführt.
Die ebenfalls benötigten Zugriffszeiten werden hierzu unter Verwendung der Timing-Diagramme und zeitlichen Kenngrössen aus den jeweiligen Datenblättern bestimmt. Im Gegensatz zum SDRAM ergibt sich der Energieverbrauch ausschließlich durch Speicherzugriffe, die Leistungsaufnahme im *idle-* und *power-down state* ist vernachlässigbar klein ($I_{DD} = 75\mu A$, [Mic04b]). Somit ist die Erstellung eines zustandsbasierten Leistungsmodells nicht erforderlich.

Hinsichtlich der Lesezugriffe unterscheidet man bei Flash-Speichern in

- *asynchronous random access*
- *intrapage access*
- *burst access* (im synchronen Betrieb)

Der *burst access* wird ausschließlich von Flash-Speichern, die für den synchronen Betrieb konfiguriert werden können, unterstützt. Der wesentliche Unterschied zwischen *burst* und *intrapage access* besteht darin, dass beim *burst access* der Speicherzugriff im Takt des Speicherbusses vorgenommen wird. Identisch ist bei beiden Zugriffsarten, dass der Erstzugriff als *random access* erfolgt.

Der zeitliche Aufwand für diesen Zugriff ist mit einem *initial access delay* verbunden und wird im folgenden mit der Kenngröße *adress to output delay* (t_{AA}) angegeben. Erfolgt ein nachfolgender Zugriff innerhalb der zuvor adressierten Seite (*intrapage access*), fällt der deutlich geringere zeitliche Aufwand *page address delay* (t_{APA}) für die Bestimmung der nächsten Adresse an (s. Abb. 5.2).

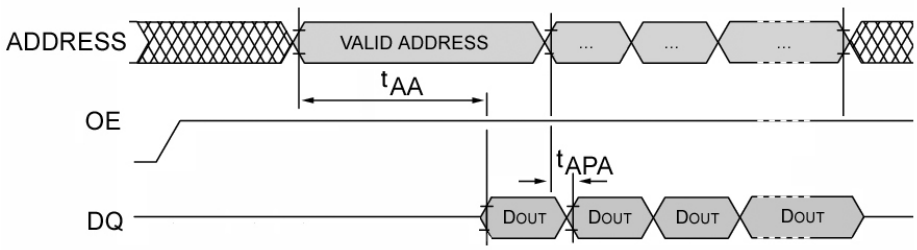


Abbildung 5.2: Page Mode Read Timing [Mic04b]

Falls der synchrone Betrieb unterstützt wird, kann auf den Speicher im *burst access* zugegriffen werden. Die Verzögerung für das Bereitstellen der ersten Adresse im synchronen Betrieb ist mit einer, dem *initial access delay* entsprechenden Anzahl an Taktzyklen, verbunden. Die Umsetzung von t_{AA} in Taktzyklen wird intern durch den *latency counter* gewährleistet. Abb. 5.3 stellt den beschriebenen Sachverhalt dar.

Die zur Berechnung benötigten Kenngrößen lassen sich prinzipiell ohne größeren Aufwand aus dem jeweiligen Datenblatt des Herstellers ablesen und unterscheiden sich in ihrer Bezeichnung nur unwesentlich. Tab. 5.2 gibt Aufschluss über die im folgenden verwendeten Größen, die in ihrer Bezeichnung der Eingabemaske aus [Mic02a] entsprechen.

Anhand der Vorbereitungen werden nun schrittweise die einzelnen Leistungen, Ener-

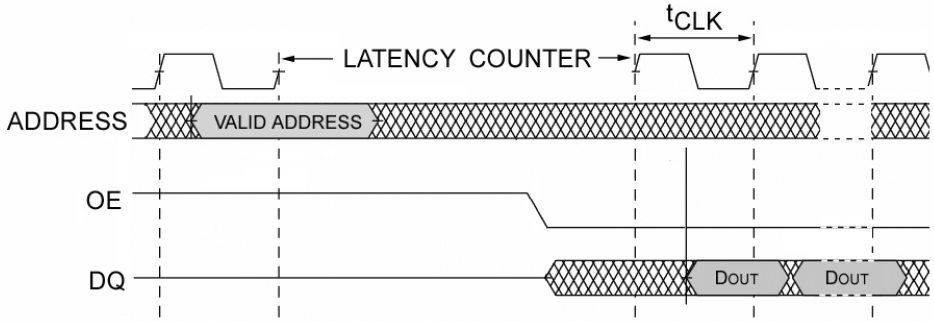


Abbildung 5.3: Burst Mode Read Timing [Mic04b]

Bezeichnung	Einheit	Abkürzung
Supply voltage	V	V_{DD}
Read async. access time	ns	t_{AA}
Read intrapage access time	ns	t_{APA}
Read burst access time	ns	t_{CLK}
Read async. access current	mA	I_{DD1}
Read intrapage access current	mA	I_{DD2}
Read burst access current	mA	I_{DD3}

Tabelle 5.2: Flash-Speicher Kenngrössen

gien und Taktzyklen für den jeweiligen Zugriff auf den Flash-Speicher mit XIP-Funktionalität berechnet. Für den asynchronen lesenden Speicherzugriff auf ein 16-Bit Wort mit T_{DOUT} Taktzyklen für die Datenausgabe, erhält man:

$$P_{XIP\_RND16(RD)} = V_{DD} \cdot I_{DD1} \tag{5.1}$$

$$E_{XIP\_RND16(RD)} = P_{XIP\_RND16(RD)} \cdot t_{AA} + E_{XIP\_DQ} \tag{5.2}$$

$$T_{XIP\_RND16(RD)} = T_{DOUT} + \left\lceil \frac{t_{AA}}{t_{CK}} \right\rceil \tag{5.3}$$

Der Zugriff auf ein 32-Bit Wort innerhalb einer Seite setzt sich analog zum SDRAM in der Berechnung aus zwei Komponenten zusammen - zum einen aus der zuvor bestimmten Leistungskomponente für den initialen *random access*, zum anderen aus der Leistungskomponente für den sequentiellen Zugriff auf das zweite 16-Bit Wort inner-

halb der zuvor adressierten Seite:

$$P_{XIP\_SEQ16(RD)} = V_{DD} \cdot I_{DD2} \tag{5.4}$$

$$E_{XIP\_RND32(RD)} = E_{XIP\_RND16(RD)} + (E_{XIP\_DQ} + P_{XIP\_SEQ16(RD)} \cdot t_{APA}) \tag{5.5}$$

Unter Berücksichtigung des *initial access delay* für den Speicherzugriff auf das erste 16-Bit Wort ergibt sich für den Zugriff im *intrapage access* die Gesamtzahl der Taktzyklen für den betrachteten Zugriff:

$$T_{XIP\_RND32(RD)} = T_{XIP\_RND16(RD)} + \left(T_{DOUT} + \left\lceil \frac{t_{APA}}{t_{CK}} \right\rceil \right) \tag{5.6}$$

Anhand der Vorüberlegungen lassen sich ohne weiteren Aufwand die Zugriffszeit und der Energieverbrauch für ein sequentiell gelesenes 16-Bit Wort berechnen.

$$T_{XIP\_SEQ16(RD)} = T_{XIP\_RND32(RD)} - T_{XIP\_RND16(RD)} \tag{5.7}$$

$$E_{XIP\_SEQ16(RD)} = E_{XIP\_RND32(RD)} - E_{XIP\_RND16(RD)} \tag{5.8}$$

Bei der Energieberechung des Zugriffs im synchronen Betrieb *(burst access)* verfährt man analog zu Gl.5.5, lediglich die Stromstärke und Zugriffszeit müssen angepasst werden.

Die Abbildungen 5.4 und 5.5 geben Aufschluss darüber, wie sich die einzelnen Zugriffsarten auf Durchsatz und Energieverbrauch auswirken. Offensichtlich ist, dass der *continuous burst access* höchsten Durchsatz bei geringstem Energieverbrauch ermöglicht.

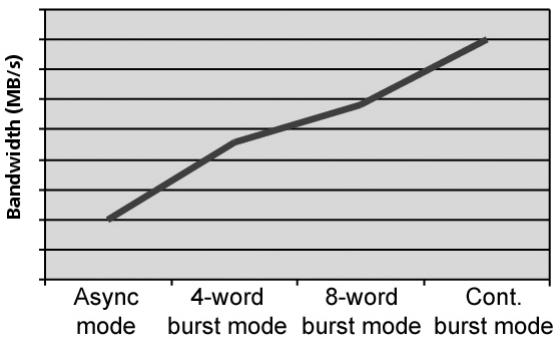


Abbildung 5.4: Durchsatz [Mic02a]

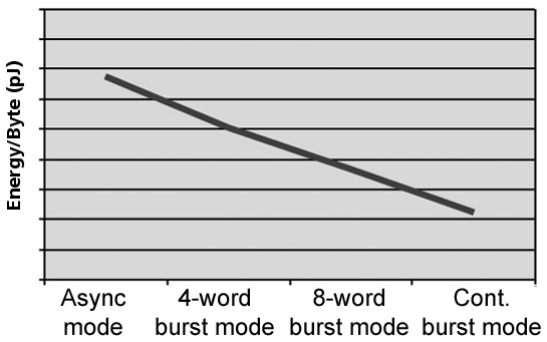


Abbildung 5.5: Energieverbrauch [Mic02a]

Energieverbrauch am Beispiel des Micron QFlash

Anhand der zuvor aufgestellten Gleichungen wird nun exemplarisch für einen Micron QFlash [Mic04b] der Energieverbrauch für den Speicherzugriff berechnet. Hierbei handelt es sich um einen *async/page* Flash-Speicher mit einer Kapazität von 64MBit, der nicht synchron betrieben werden kann und somit keinen *burst access* erlaubt. Die weiteren Kenngrößen sind in Tab.5.2 dargestellt.

Bezeichnung	Einheit	Abkürzung	Kenngrösse
Supply voltage	V	V_{DD}	3,3
Read async. access time	ns	t_{AA}	100
Read intrapage access time	ns	t_{APA}	20
Read burst access time	ns	t_{CLK}	-
Read async. access current	mA	I_{DD1}	9
Read intrapage access current	mA	I_{DD2}	8
Read burst access current	mA	I_{DD3}	-

Tabelle 5.3: *Micron QFlash*

Analog zum SDRAM soll auch in diesem Beispiel $T_{DOUT} = 1$ gelten. Die Taktzyklen, die ein solcher Zugriff dann beansprucht, ergeben sich unter der Annahme, dass der Prozessor mit 100 MHz getaktet ist ($t_{CK} = 10ns$) zu

$$T_{XIP\_RND16(RD)} = 1 + \left\lceil \frac{100ns}{10ns} \right\rceil = 11$$

Unter der Annahme, dass $E_{XIP\_DQ} = E_{SDR\_DQ}$:

$$E_{XIP\_DQ} = 77,8mW \cdot 10ns = 0,778nJ$$

Für den asynchronen Zugriff auf ein Halbwort erhält man:

$$E_{XIP\_RND16(RD)} = 3,3V \cdot 9mA \cdot 100ns + 0,778nJ = 3,748nJ$$

Weiterhin werden für die Integration in die Arbeitsumgebung die Taktzyklen für den

32Bit-Zugriff benötigt:

$$T_{XIP\_RND32(RD)} = 11 + 1 + \left\lceil \frac{20ns}{10ns} \right\rceil = 14$$

Der Energieverbrauch eines solchen Speicherzugriffs auf den betrachteten Flash-Speicher ergibt sich zu

$$E_{XIP\_RND32(RD)} = 3,748nJ + (0,778nJ + 3,3V \cdot 8mA \cdot 20ns) = 5,054nJ$$

Die Zugriffsenergien für die Speicherzugriffe mit initialem *random access* sind somit bestimmt. Für jedes weitere 16-Bit Wort, auf das im *intrapage access* zugegriffen wird, lassen sich die Zugriffszyklen und der Energieverbrauch durch Bildung der Differenz eines Zugriffes auf 32-Bit und 16-Bit Wort berechnen.

$$\begin{aligned} T_{XIP\_SEQ16(RD)} &= 14 - 11 = 3 \\ E_{XIP\_SEQ16(RD)} &= 5,054nJ - 3,748nJ = 1,306nJ \end{aligned}$$

5.3 Laden von Programmobjekten aus dem Flash-Speicher

Bei der derzeitigen Arbeitsumgebung wird ein Programm compilergesteuert in einzelne Programmobjekte zerlegt, die dann gemäß der gewählten Partitionen in mehrere separate Objektdateien übersetzt wird. Die einzelnen Objektdateien werden dann vom bereitgestellten Linker zu einem ausführbaren Programm zusammengefasst, indem die jeweilige Objektdatei mittels *scatter loading* an die Adresse der entsprechenden Speicherpartition geladen werden. Die Speicheraufteilung wird in einer Konfigurationsdatei *scat.txt* festgehalten, die nebst dem Namen der einzelnen Objekte zusätzliche Informationen über die einzelnen Partitionen, insbesondere die benötigten Startadressen bereitstellt.

Zu Beginn der Programmausführung werden dann nach dem Konzept des *code shadowing* die einzelnen Programmobjekte an die jeweiligen Adressen der Speicher kopiert.

5.3.1 Anpassung der Arbeitsumgebung

Der zuvor beschriebene Sachverhalt lässt zunächst darauf schließen, dass in der Arbeitsumgebung, wie sie in vorhergehenden Arbeiten betrachtet wurde, von einem Flash-Speicher in NAND-Technologie ausgegangen oder zumindest die XIP-Fähigkeit des Flash-Speichers nicht ausgenutzt wurde.

Als vorbereitende Maßnahmen für weitere Untersuchungen wurden an dieser Stelle folgende Anpassungen vorgenommen:

- Bei vorhergehenden Arbeiten wurde der Kopiervorgang i.a. nicht festgehalten und somit auch nicht dem Energiebedarf der Programmausführung zugerechnet. Insbesondere bei kleinen Programmen wird sich herausstellen, dass das initiale Kopieren vom Flash-Speicher einen erheblichen Einfluss auf den Energieverbrauch der Speicher haben kann.
  $\Rightarrow$ Integration des Kopiervorgangs in die Analyse des Energieverbrauchs

- Desweiteren wurde prinzipiell für alle wählbaren Speicherpartitionen in der Initialisierung die Kopierfunktion aufgerufen, unabhängig davon, ob diese Speicherpartitionen vom Compiler überhaupt gewählt wurden. Hierbei werden zwar keine unbenötigten Daten kopiert, trotzdem entsteht ein ungerechtfertigter Mehraufwand für den alleinigen Aufruf der Kopierfunktion.
  $\Rightarrow$ Dynamische Erzeugung des Initialisierungscodes

- Die betrachteten Speicher in SDRAM- und Flash-Technologie sind aufgrund ihrer Aktualität für den Betrieb in modernen Systemen mit adäquat schnellen Prozessoren konzipiert. Dementsprechend sind Anpassungen der Taktfrequenz des Prozessors notwendig, da durch gravierende Untertaktung der Speicher, die Ergebnisse nicht repräsentant wären (vgl. Tab.4.2).
  $\Rightarrow$ Anbindung der Speicher an einen Bus mit 100MHz Taktfrequenz

Der letzte Aufzählungspunkt erfordert eine Überarbeitung des Prozessorenergiemodells, da die Erhöhung der Taktfrequenz des Speicherbusses eine dementsprechende Erhöhung der CPU-Frequenz erfordert und somit eine höhere Leistungsaufnahme in der CPU verursacht. Der Einfluss von Schaltaktivität α, Ausgangslast C_L, Betriebsspan-

nung V_{DD} und Frequenz f auf die Leistungsaufnahme ist gegeben durch:

$$P = \alpha \cdot C_L \cdot (V_{DD})^2 \cdot f \tag{5.9}$$

Die Leistungsaufnahme des betrachteten ARM-Prozessors, bei einem Betrieb mit 33 MHz und einer Spannung von 3,3V, beläuft sich in der derzeitigen Arbeitsumgebung auf durchschnittlich 150mW. Durch Multiplikation der Leistung mit einem Skalierungsfaktor würde sich die Leistung, dem Takt entsprechend, verdreifachen und somit auf 450mW ansteigen.

Die Betriebsspannung moderner ARM-Prozessoren liegt, aufgrund von Fortschritten in der Fertigung, derzeit bei $1,7 - 1,9V$. Da die Spannung quadratisch in die Berechnung der Leistungsaufnahme eingeht, würde sich bei entsprechender Skalierung, der zuvor angenommene Faktor durch die Frequenzskalierung relativieren.

Eine Aufteilung der Leistungsaufnahme des ARM920T-Prozessors in 0,18μm-Technologie wird in [Seg01] dargestellt (s.Abb.5.6).

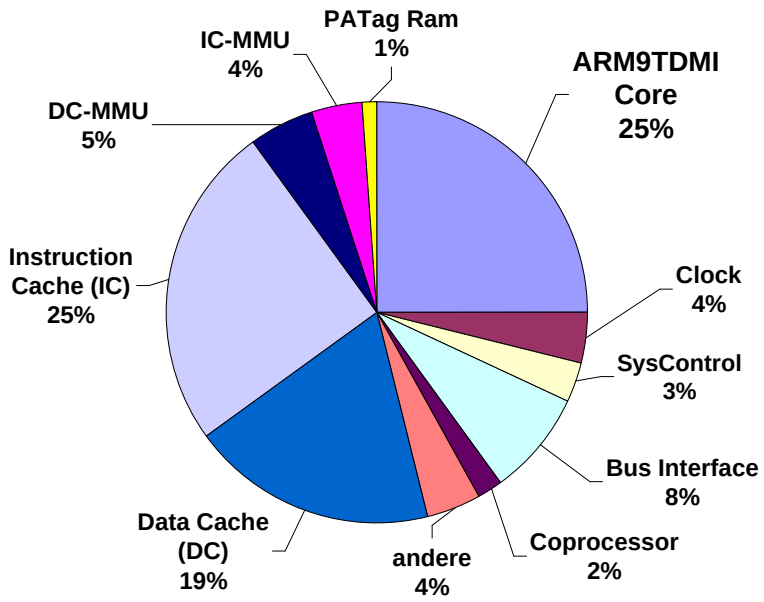


Abbildung 5.6: Analyse der Leistungsaufnahme des ARM920T

Aus dem Diagramm geht hervor, dass 54% der Leistungsaufnahme durch Caches (inkl. MMU und PATag Ram) verursacht werden. Mit einem Anteil von 25% der Gesamtleistung nimmt der ARM9TDMI-Core 54% der Leistung in einem System ohne Caches auf. Unter Verwendung von [ARM99] ergibt sich somit folgende grobe Abschätzung für die Leistungsaufnahme der betrachteten Prozessoren, unter der Annahme, dass der Core 50% der Gesamtleistung einer CPU ohne Caches aufnimmt.

Technologie [μm]	Taktfrequenz (*worst case*) [MHz]	V_{DD} [V]	Core [mW/MHz]	CPU [mW/MHz]	Leistung [mW]
0,35	33	3,3	2,07	4,14	136,6
0,25	66	2,5	0,8	1,6	105,6
0,18	84	1,8	0,25	0,5	42

Tabelle 5.4: Geschätzte Leistungsaufnahme von ARM7TDMI-basierten Prozessoren

Aus der Tab.5.4 geht hervor, dass trotz zunehmender Taktfrequenz die relative Leistungsaufnahme abnimmt, so dass durch Weiterverwendung des bestehenden Prozessorenergiemodells der Energieverbrauch der CPU bei höherer Taktfrequenz nicht unterbewertet wird.

Da im vorliegenden Modell keine Caches vorhanden sind und der betrachtete Prozessor einen Großteil der Instruktionen innerhalb eines Taktes verarbeitet, erlaubt eine Gleichtaktung von Prozessor und Speicher im *burst mode* das Einholen von einer Instruktion pro Takt. Instruktionen, die mehrere Taktzyklen zur Verarbeitung benötigen (Transferbefehle mit Zugriff auf den Hauptspeicher, Sprungbefehle), unterbrechen durch zuvor formulierte Restriktionen den *burst access*, so dass die Angleichung des CPU-Taktes mit dem Speichertakt als ausreichend betrachtet wird. In Systemen, bei denen der Speichertakt deutlich unter dem Prozessortakt liegt, werden für gewöhnlich Caches eingesetzt um diese Geschwindigkeitsdiskrepanz auszugleichen.

5.3.2 Analyse der Kopierfunktion

Die Analyse des Energieverbrauchs des Kopiervorgangs setzt eine genaue Kenntnis der eigentlichen Kopierfunktion voraus. Hierzu werden für jede Objektdatei die notwen-

digen Adressen und die Objektgröße geladen. Folgender Auszug aus dem Initialisierungscode stellt obigen Sachverhalt dar.

```
LDR     r0, =$copyloadsym
LDR     r1, =$copybasesym
MOV     r2, r1
LDR     r4, =$copylensym
ADD     r2, r2, r4
BL      copy
```

Mnemonic	Instruktions-zyklen	Zugriff (Inst)	Zugriff (Data)
LDR	3	R	R
LDR	3	R	R
MOV	1	R	-
LDR	3	S	R
ADD	1	R	-
BL	3	S	-
T_{INIT}	14		

Tabelle 5.5: Speicherzugriff in der Initialisierung

Für jede Objektdatei werden also einmalig sechs Befehle ausgeführt, von denen drei auf den Speicher zugreifen. Nach dem Laden der einzelnen Parameter wird die eigentliche Kopierroutine angesprungen und die Objektdatei wortweise (32 Bit) in einer Schleife in die entsprechende Partition kopiert. Die Anzahl der Schleifendurchläufe L für ein Speicherobjekt o_i bei gegebener Objektgröße $Size(o_i)$ (in Byte) kann nach oben abgeschätzt werden durch

$$L(o_i) \leq \left\lfloor \frac{Size(o_i)}{4} \right\rfloor . \tag{5.10}$$

In jedem Durchlauf der Kopierroutine werden vier Befehle, darunter zwei Speicherbefehle, ausgeführt.

```
copy

    CMP     r1, r2
    LDRCC   r3, [r0], #4 // bedingter load
```

```
STRCC   r3, [r1], #4 // bedingter store
BCC     copy
MOV     pc, lr          ; return from subroutine copy
```

Mnemonic	Instruktions-zyklen	Zugriff (Inst)	Zugriff (Data)
CMP	1	R	-
LDRCC	3	S	R
STRCC	2	R	R
BCC	3	R	-
T_{LOOP}	9		

Tabelle 5.6: Speicherzugriffe in der Kopierroutine

Der Energieverbrauch, der durch den Kopiervorgang im Speicher verursacht wird, kann nun durch Analyse der Speicherzugriffe berechnet werden.

Energieverbrauch des Speichers

Wesentlichen Einfluss auf den Verbrauch haben einerseits die Zugriffsgeschwindigkeiten der Speicher als auch die Zugriffsarten auf diese. Hierzu wird anhand des Simulationsprotokolls festgestellt, welche Zugriffe im *random access mode* (R) oder sequentiell (S) stattfinden. In den Tabellen 5.5 und 5.6 werden die Instruktionszyklen und die jeweiligen Speicherzugriffsarten für die weitere Analyse bereitgestellt.

In den folgenden Überlegungen wird davon ausgegangen, dass das Programmobjekt o_i vom Flash-Speicher auf eine SDRAM-Partition transferiert wird. Beim Kopiervorgang wird nicht der 16-Bit-Thumb-Mode, sondern der 32-Bit-ARM-Mode benutzt, so dass bei der Energieberechnung für eine Instruktion zwei 16-Bit Speicherzugriffe angenommen werden. Der Zugriff auf ein 32-Bit Wort, welches vollständig sequentiell gelesen werden kann, hat dementsprechend den Energieverbauch von zwei sequentiell gelesenen 16-Bit Zugriffen.

$$E_{XIP\_SEQ32(RD)} = 2 \cdot E_{XIP\_SEQ16(RD)} \tag{5.11}$$

Anhand der in Tab.5.5 aufgeführten Speicherzugriffe lässt sich die Berechnung des Energieverbrauchs im Speicher für die zuvor beschriebene Initialisierung wie folgt an-

geben:

$$E_{MEM\_INIT} = 7 \cdot E_{XIP\_RND32(RD)} + 2 \cdot E_{XIP\_SEQ32(RD)} \tag{5.12}$$

Bei der eigentlichen Kopierroutine muss berücksichtigt werden, dass der *store*-Befehl auf den SDRAM speichert (s.Tab.5.6). Abhängig von der Anzahl der Schleifendurchläufe lässt sich die Energie der Kopierroutine berechnen:

$$E_{MEM\_LOOP}(o_i) = L(o_i) \cdot \left(4 \cdot E_{XIP\_RND32(RD)} + E_{XIP\_SEQ32(RD)} + E_{SDR\_RND32(WR)}\right) \tag{5.13}$$

Energieverbrauch der CPU

Der Energieverbrauch in der CPU wird durch die ausgeführten Instruktionen und deren Ausführungsdauer hervorgerufen. Die Ausführungsdauer wiederum wird einerseits durch die Instruktionszyklen, andererseits durch die zusätzlichen Wartezyklen in den Speichern bestimmt. Die Gesamtzyklen der Initialisierungsphase berechnen sich mit T_{INIT} (s.Tab.5.5) zu:

$$T_{CPU\_INIT} = T_{INIT} + 7 \cdot T_{XIP\_RND32(RD)} + 2 \cdot T_{XIP\_SEQ32(RD)} \tag{5.14}$$

Es werden nun die Zyklen der einzelnen Schleifendurchläufe berechnet, wobei auch hier wieder beachtet werden muss, dass der *store*-Befehl auf die SDRAM-Partition speichert. Durch Multiplikation mit den Schleifendurchläufen erhält man mit T_{LOOP} (s.Tab.5.6) die Gesamtzyklen der eigentlichen Kopierfunktion.

$$T_{CPU\_LOOP}(o_i) = L(o_i) \cdot \left(T_{LOOP} + 4 \cdot T_{XIP\_RND32(RD)} + T_{XIP\_SEQ32(RD)} + T_{SDR\_RND32(WR)}\right) \tag{5.15}$$

Die CPU-Energie erhält man schließlich durch Multiplikation der Instruktions-Energie mit der Summe der zuvor berechneten Wartezyklen.
Da [The00] in seiner Arbeit lediglich die Energiewerte für den Thumb-Befehlssatz des ARM-Prozessors gemessen hat, wird an dieser Stelle ein durchschnittlich gemessener Strom I_{AVG} für 32-Bit Instruktionen angenommen.

$$E_{CPU\_INIT}(o_i) = T_{CPU\_INIT}(o_i) \cdot t_{CK} \cdot I_{AVG} \cdot V_{DD} \tag{5.16}$$

$$E_{CPU\_LOOP}(o_i) = T_{CPU\_LOOP}(o_i) \cdot t_{CK} \cdot I_{AVG} \cdot V_{DD} \tag{5.17}$$

Insgesamt erhält man für das Kopieren des Objekts o_i:

$$E_{COPY}(o_i) = E_{CPU\_LOOP}(o_i) + E_{MEM\_LOOP}(o_i) \tag{5.18}$$

Für die einmalige Initialisierung, die für jede Speicherpartition zum Laden der entsprechenden Adressen anfällt, addiert man die entstehenden Energieverbräuche in CPU und Speicher:

$$E_{INIT} = E_{MEM\_INIT} + E_{CPU\_INIT} \tag{5.19}$$

Fasst man alle Objekte o_i zusammen, die zu Beginn der Programmausführung auf die Speicherpartition mem_i transferiert werden, ergibt sich die Gesamtenergie:

$$E_{COPY}(mem_i) = E_{INIT} + \sum_{o_i \in mem_i} E_{COPY}(o_i) \tag{5.20}$$

5.4 Ausnutzung der XIP-Funktionalität von NOR-Flash

Das folgende Kapitel befasst sich mit einer compilergesteuerten Optimierung, die speziell auf SDRAM und Flash-basierte Speichertechnologien zugeschnitten ist. Ziel der Optimierung stellt die Minimierung des Gesamtenergieverbrauchs dar.

Zum einen steht dem Compiler ein Flash-Speicher [Mic04b] zur Verfügung, der mit XIP-Funktionalität ausgestattet ist, zum anderen werden zwei mobile SDRAM-Speicher [Mic04c] in das betrachtete System integriert, die als separate Instruktions- und Datenspeicher genutzt werden.

5.4.1 Partitionierung des Hauptspeichers

Die Partitionierung des Hauptspeichers wird durch die Aufteilung in Instruktions- und Datenspeicher motiviert. Diese Aufteilung hat folgende Vorteile:

- Der Datenspeicher in SDRAM-Technologie behebt das Problem der eingeschränkten Beschreibbarkeit des Flash-Speichers. Zu Beginn einer Programmausführung

werden alle Daten aus dem Flash-Speicher in den Datenspeicher kopiert und können dort beliebig oft modifiziert und überschrieben werden. Weiterhin wird der *stack* in den Datenspeicher gelegt.

- Der Instruktionsspeicher in SDRAM-Technologie steht dem Compiler als schnelle Alternative zum Flash-Speicher zur Verfügung. Wenn Instruktionen aus dem Flash-Speicher ausgeführt werden, besteht für diesen SDRAM die Möglichkeit in den Power-Down Modus zu wechseln. Weiterer Vorteil dieser Aufteilung ist die temporäre Nutzung des Instruktionsspeichers. Sofern alle Instruktionen aus diesem Speicher abgearbeitet sind, kann diese Partition nach dem letzten Speicherzugriff abgeschaltet werden (DPD).

Anhand der Partitionierung werden nun drei verschiedene Ansätze für die Programmausführung untersucht und hinsichtlich ihres Energieverbrauchs bewertet:

- *fully code shadowing* - die komplette Codesektion wird zu Beginn der Programmausführung von dem Flash-Speicher in den Instruktionsspeicher kopiert und das gesamte Programm aus dem SDRAM-Speicher ausgeführt
- *fully execute in place* - die komplette Codesektion wird aus dem Flash-Speicher ausgeführt, der Instruktionsspeicher wird abgeschaltet.
- *compiler-based memory allocation* - die compilergesteuerte Optimierung nimmt die Programmverteilung vor. Einzelne Codesektionen werden dann aus dem Flash-Speicher, andere wiederum zu Beginn der Ausführung in den Instruktionsspeicher in SDRAM-Technologie kopiert und von diesem ausgeführt. Das dynamische Kopieren unter Ausnutzung des DPD wird gemäß der in Kap.4 durchgeführten Untersuchung des Optimierungspotentials nicht betrachtet.

Die Daten werden jeweils zu Beginn der Programmausführung in den Datenspeicher in SDRAM-Technologie kopiert.

5.4.2 Vorbereitungen

Als vorbereitende Maßnahme wird die Energie für den gewählten SDRAM berechnet, die ausschließlich durch die zugriffsbedingten Leistungskomponenten bestimmt wird

(s Kap.4.2.2). Hierbei werden alle anderen Energieverbräuche in den Zugriffs- als auch in den Nichtzugriffsphasen zunächst vernachlässigt.

Während der Analyse des *trace* durch den *enProfiler* werden für die einzelnen Programmobjekte die Energieanteile für CPU und Speicher als Kosten festgehalten. Diese Anteile werden unter Berücksichtigung der sequentiellen Zugriffe für die beiden zur Verfügung stehenden Instruktionsspeicher (Flash und SDRAM) berechnet.

Energiebedarf des Objekts o_i bei Ausführung aus dem SDRAM

Für die einmalige Ausführung eines Programmobjekts o_i summiert man die Energiekosten für die Lesezugriffe auf und erhält die zugriffsbedingte Energie. Unter der Annahme, dass r Zugriffe im *random access mode* und s Zugriffe im *burst mode* stattfinden ergibt sich (s Kap.4.2.2):

$$E_{SDR\_ACC}(o_i) = r \cdot E^*_{SDR\_RND16(RD)} + s \cdot E^*_{SDR\_SEQ16(RD)} \tag{5.21}$$

Ebenso werden für die zur Verfügung stehenden Partitionen die jeweils benötigten Ausführungsdauern $T_{SDR}(o_i)$ und $T_{XIP}(o_i)$ des Programmobjekts festgehalten.

Anhand der Ausführungsdauer eines Programmobjekts werden dann für den SDRAM die zuvor vernachlässigten Energieanteile berechnet. Die mit dem Zugriff auf den Datenspeicher verbundenen Kosten können unberücksichtigt bleiben, da diese unabhängig von der Wahl des Instruktionsspeichers gleich sind.
Offensichtlich fällt bei der Ausführung des Objekts aus dem Instruktionsspeicher die konstante nichtzugriffsbedingte Leistungsaufnahme p_{SDR_STBY} an. Der Datenspeicher in SDRAM-Technologie wird für die Ausführungsdauer ebenfalls eine konstante Leistungsaufnahme vorweisen. Um diese Komponente nicht überzubewerten, wird für die Ausführungsdauer der Power-Down-Modus angenommen, da prinzipiell nicht jeder

Basisblock auf Daten zugreift.

$$\begin{aligned} E_{SDR\_LEAK}(o_i) &= (p_{SDR\_STBY} + p_{SDR\_REF}) \cdot t_{CK} \cdot T_{SDR}(o_i) \\ &+ (p_{SDR\_PDN} + p_{SDR\_REF}) \cdot t_{CK} \cdot T_{SDR}(o_i) \\ &= (p_{SDR\_PDN} + p_{SDR\_STBY} + 2 \cdot p_{SDR\_REF}) \cdot t_{CK} \cdot T_{SDR}(o_i) \end{aligned} \quad (5.22)$$

Für den Gesamtenergiebedarf bei Ausführung eines Programmobjekts o_i addiert man die zugriffsbedingten Kosten. Multipliziert mit der Anzahl der Ausführungen dieses Objekts und dem einmaligen Kostenmehraufwand, der durch das Kopieren vom Flash-Speicher anfällt (s. Gl. 5.18), erhält man den Gesamtenergiebedarf

$$E_{SDR}(o_i) = v(o_i) \cdot (E_{SDR\_LEAK}(o_i) + E_{SDR\_ACC}(o_i) + E_{SDR\_CPU}(o_i)) + E_{SDR\_COPY}(o_i) \quad (5.23)$$

Energiebedarf des Objekts o_i bei Ausführung aus dem Flash

Analog ergeben sich für den Flash-Speicher die Kosten für den Speicherzugriff, wobei hier die s Zugriffe im *intrapage access* stattfinden, da kein synchroner Flash-Speicher verwendet wird.

$$E_{XIP\_ACC}(o_i) = r \cdot E_{XIP\_RND16(RD)} + s \cdot E_{XIP\_SEQ16(RD)}$$

Der Anteil der nichtzugriffsbedingten Energiekosten der SDRAM-Speicher werden ebenfalls auf die Kosten aufgeschlagen, da auch bei Ausführung dieses Objekts aus dem Flash-Speicher im SDRAM während der Nichtzugriffsphase Energieverbräuche anfallen. Während der Ausführung aus dem Flash-Speicher wird nicht auf den SDRAM-Instruktionsspeicher zugegriffen, so dass ein Wechsel in den *power-down state* angenommen wird (s.Kap. 4.5). Der konstante Verbrauch des Datenspeichers wird während der Flash-Ausführung ebenfalls durch p_{SDR_PDN} bestimmt.

$$\begin{aligned} E_{XIP\_LEAK}(o_i) &= (p_{SDR\_STBY} + p_{SDR\_REF}) \cdot t_{CK} \cdot T_{OH} \\ &+ (p_{SDR\_PDN} + p_{SDR\_REF}) \cdot t_{CK} \cdot (T_{XIP}(o_i) - T_{OH}) \\ &+ (p_{SDR\_PDN} + p_{SDR\_REF}) \cdot t_{CK} \cdot T_{XIP}(o_i) \end{aligned}$$

Anschließend werden anhand der Ausführungen des betrachteten Programmobjekts die Gesamtkosten berechnet, wobei hier die Kopierkosten wegfallen.

$$E_{XIP}(o_i) = v(o_i) \cdot (E_{XIP\_LEAK}(o_i) + E_{XIP\_ACC}(o_i) + E_{XIP\_CPU}(o_i)) \tag{5.24}$$

5.4.3 Vorauswahl der Objekte für die Flash-Speicher Ausführung

Anhand der durch den *enProfiler* festgestellten Zugriffsabfolge wird eine Vorauswahl derjenigen Programmobjekte getroffen, die für die Flash-Ausführung unter Ausnutzung des DPD des SDRAM-Instruktionsspeichers geeignet sind.
Zum einen wird hierdurch die Problemgröße des anschließend formulierten IP-Modells reduziert. Weiterhin wird durch diese Vorauswahl festgestellt, ob unter Ausnutzung des DPD die komplette Ausführung des Programms aus dem Flash-Speicher die günstigste Variante (in Bezug auf den Energieverbrauch) darstellt.

Hierzu sei $S = \{s_1, ..., s_m\}$ die zeitliche Abfolge von Basisblocksequenzen und o_{s_j} der in der Sequenz s_j ausgeführte Basisblock (s. Abb.5.7). Mit der Anzahl der Ausführungen $v(o_{s_j})$ des Basisblocks o_{s_j} in der Sequenz s_j erhält man:

$$\forall j,\ 1 \leq j \leq m \ \text{ mit } \ o_{s_j} = o_i : \sum_j v(o_{s_j}) = v(o_i) \tag{5.25}$$

Da der SDRAM nach dem Abschalten seinen Inhalt verliert, wird zur Vorauswahl das Zugriffsschema rückwärts durchlaufen. Es wird nun der Zeitpunkt bestimmt, ab dem der Speicher abgeschaltet werden kann. Die auf diesen Zeitpunkt folgenden Objekte sind aufgrund des geringeren Energieverbrauchs für die Flash-Ausführung prädestiniert und müssen somit nicht in den SDRAM kopiert werden.

Hierbei wird der Energieverbrauch der Flash-Ausführung neu berechnet, da die Instruktionspartition bei deren Ausführung bereits abgeschaltet ist. Für den Datenspeicher wird wiederum die Leistungskomponente p_{SDR_PDN} mit Refresh hinzuaddiert.

$$E^{DPD}_{XIP\_LEAK}(o_{s_j}) \quad = \quad (p_{SDR\_PDN} + p_{SDR\_REF} + p_{SDR\_DPD}) \cdot t_{CK} \cdot T_{XIP}(o_{s_j}) \tag{5.26}$$

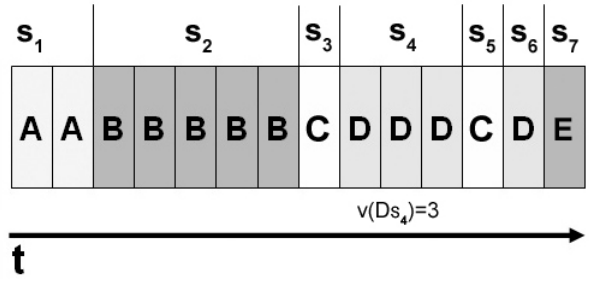


Abbildung 5.7: Zeitliche Abfolge von Basisblocksequenzen

Die zuvor angenommenen Kopierenergien werden im Rahmen der Vorauswahl dem Flash-Speicher zu geeigneten Zeitpunkten gutgeschrieben und daher aus den Kosten für die SDRAM-Ausführung extrahiert. Man erhält:

$$E_{XIP}^{DPD}(o_{s_j}) = v(o_{s_j}) \cdot (E_{XIP\_LEAK}^{DPD}(o_{s_j}) + E_{XIP\_ACC}(o_{s_j}) + E_{XIP\_CPU}(o_{s_j})) \quad (5.27)$$

$$E_{SDR}^{DPD}(o_{s_j}) = v(o_{s_j}) \cdot (E_{SDR\_LEAK}(o_{s_j}) + E_{SDR\_ACC}(o_{s_j}) + E_{SDR\_CPU}(o_i)) \quad (5.28)$$

Algorithmus zur Vorauswahl von DPD-Objekten

Zur Erläuterung der Vorauswahl werden zunächst drei Mengen eingeführt:

$$A := \{o_i |\ o_i \text{ hat zum betrachteten Zeitpunkt noch Ausführungen offen}\}$$
$$M := \{o_i |\ o_i \text{ ist für die Ausführung aus dem Flash-Speicher geeignet}\}$$
$$X := \{o_i |\ o_i \text{ wird aus dem Flash-Speicher ausgeführt}\}$$

Zunächst wird die allgemeine Funktionsweise des Algorithmus erläutert und dieser anschließend in Pseudo-Code notiert.

In Abb.5.8a wird als erstes das Objekt E besucht und festgestellt, dass das Objekt an dieser Stelle erst- und letztmalig ausgeführt wird. Durch Ausführung dieses Objekts aus dem Flash-Speicher wird durch Wegfallen der Kopierenergie ein positiver Nutzen berechnet und somit das Objekt in die Menge X aufgenommen. Der aktuelle positive Nutzwert wird mit 0 reinitialisiert. An dieser Stelle wird ebenfalls angenommen, dass der SDRAM spätestens nach Ausführung des vorhergehenden Objekts abgeschaltet werden kann (DPD).

In Abb.5.8b wurden die Objekte C und D besucht, die beide zunächst in die Menge A eingefügt werden, da sie zu früheren Zeitpunkten noch Ausführungen offen haben.

In Abb.5.8c wird die Sequenz mit der ersten Ausführung des Objekts D gefunden und somit D wird aus der Menge A entfernt. Aufgrund seiner häufigen Ausführung weist dieses Objekt im Beispiel einen höheren Nutzen im SDRAM vor, so dass dieses Objekt zunächst nur als geeignet deklariert und somit in die Menge M eingefügt wird. Der derzeitige Nutzwert ist negativ.

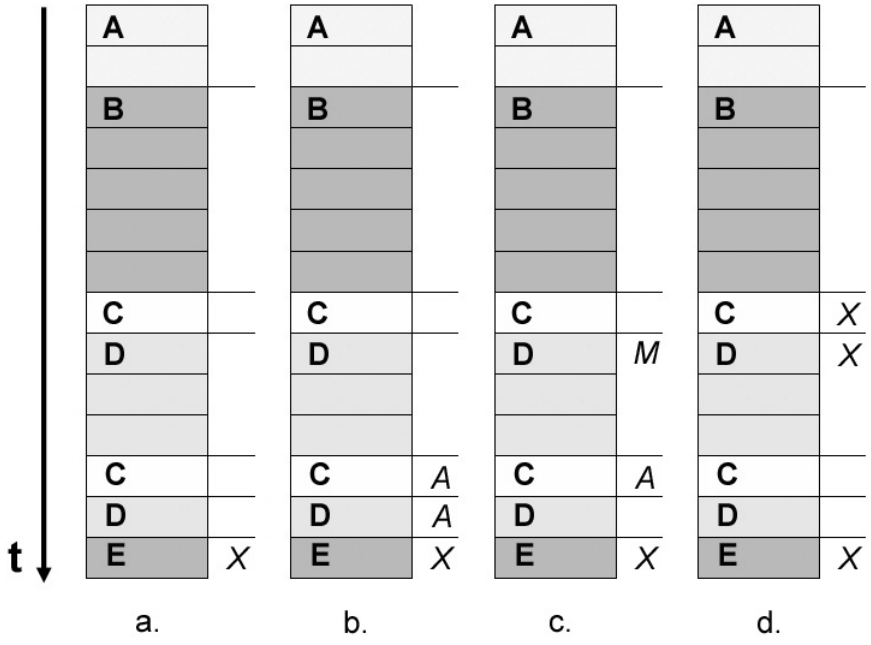


Abbildung 5.8: Algorithmus zur Vorauswahl von DPD-Objekten

In Abb.5.8d findet der Algorithmus die erste Ausführung des Objekts C. Der Nutzen im Flash-Speicher dieses Objekts sei im Beispiel dermaßen hoch, dass der bisherig negative Nutzwert durch Addition des Nutzens von Objekt C positiv wird. Somit werden C und alle geigneten Objekte aus M für die Flash-Ausführung gekennzeichnet und der aktuelle Nutzwert reinitialisiert.

Nachdem der Algorithmus bei der ersten Basisblocksequenz angelangt ist, sei im Beispiel der Nutzwert negativ. Somit wird kein weiteres Objekt für die Flash-Ausführung vorgesehen und der Algorithmus terminiert.

Formalisierung des Algorithmus

Aus dem Beispiel geht hervor, dass zu bestimmten Zeitpunkten die Kopierenergie für die Objekte $o_i \in M$ in den nachfolgenden Betrachtungen benötigt wird.
Mit $E^C(o_i) = E_{SDR\_COPY}(o_i)$ erhält man:

$$E^C(M) = \sum_{o_i \in M} E^C(o_i) \tag{5.29}$$

Die Differenz zwischen dem Energieverbrauch der SDRAM- und Flash-Ausführung einer Sequenz s_j wird bestimmt und dem aktuellen Energiegewinn E^+ (initialisiert mit 0) aufaddiert.

$$E^+ \leftarrow \left(E^+ + (E^{DPD}_{SDR}(o_{s_j}) - E^{DPD}_{XIP}(o_{s_j}))\right)$$

Sei $v^*(o_i)$ (initialisiert mit $v(o_i)$) die noch verbleibende Anzahl der Ausführungen des Objekts o_i. Bei Betrachtung der Sequenz s_j wird die Anzahl der Ausführungen des Basisblocks $v(o_{s_j})$ mit $o_{s_j} = o_i$ von $v^*(o_i)$ abgezogen.

$$v^*(o_i) \leftarrow \left(v^*(o_i) - v(o_{s_j})\right) \tag{5.30}$$

Hierbei wird nun geprüft, ob es sich in dieser Sequenz um den Erstzugriff auf das Objekt $o_{s_j} = o_i$ handelt.

- $v^*(o_i) = 0$: In dem Fall des Erstzugriffs wird o_i aus A entfernt und anschließend überprüft, ob $A = \emptyset$.
  - $A \neq \emptyset$: Das aktuelle Objekt wird in M eingefügt.
  - $A = \emptyset$: Die Kopierenergien für das betrachtete Objekt und alle Objekte $\in M$ werden dem aktuellen Energiegewinn gutgeschrieben. Es wird nun überprüft, ob der Gewinn die Bedingung der Nichtnegativität erfüllt.
    - $E^+ + E^C(M) + E^C(o_i) \geq 0$: M und das aktuell betrachtete Objekt werden in X eingefügt. $E^+ = 0$ und $M = \emptyset$.
      Fahre fort mit s_{j+1}
    - $E^+ + E^C(M) + E^C(o_i) < 0$: Verfahre wie bei $A \neq \emptyset$.
      Fahre fort mit s_{j+1}

- $v^*(o_i) > 0$: Das Objekt o_i wird lediglich in die Menge A eingefügt.
  Fahre fort mit s_{j+1}

Die Objekte der Menge X werden nach Durchlauf des Zugriffsschemas automatisch dem Flash-Speicher zugeordnet und reduzieren somit die Problemgröße für die nachfolgende Optimierung, die die weiteren Objekte auf die einzelnen Partitionen verteilt.

5.4.4 Formalisierung der Optimierungsproblematik

Anhand der Kosten der einzelnen Programmobjekte bei Ausführung aus den betrachteten Speichern wird ein Minimierungsproblem formuliert.

Die Menge aller Programmobjekte o_i mit jeweiliger Größe *Size*(o_i) wird in dieser Optimierung ausschließlich durch die Basisblöcke $b_1, ..., b_n$ gebildet, da die globalen Variablen immer in den Datenspeicher verschoben werden.

$$O := \{o_1 = b_1, ..., o_n = b_n\} \tag{5.31}$$

Die Kosten für die Programmobjekte o_i ergeben sich aus den Vorüberlegungen:

$$\begin{aligned} E_{SDR}(o_i) &= v(o_i) \cdot (E_{SDR\_LEAK}(o_i) + E_{SDR\_ACC}(o_i) + E_{SDR\_CPU}(o_i)) + E_{COPY}(o_i) \\ E_{XIP}(o_i) &= v(o_i) \cdot (E_{XIP\_LEAK}(o_i) + E_{XIP\_ACC}(o_i) + E_{XIP\_CPU}(o_i)) \end{aligned}$$

Die Berechnung der Kopierenergie entnehme man der Gl.5.18 . Die Kosten für durch die Vorauswahl bestimmten Objekte der Menge X werden für den Flash-Speicher auf 0 gesetzt.

$$\forall o_i \in X : E_{XIP}(o_i) = 0 \tag{5.32}$$

Zusätzlich wird davon ausgegangen, dass von jedem Objekt in eine andere Partition gesprungen werden muss, so dass zusätzlich multipliziert mit der Anzahl der Ausführungen des Objekts die Energie für einen *long jump* (32-Bit Befehl) addiert wird. Hierbei

wird davon ausgegangen, dass dieser Befehl sequentiell gelesen werden kann.

$$E_{SDR\_JMP} = E_{SDR\_SEQ32(RD)} + E_{SDR\_CPU}(jmp) \tag{5.33}$$

$$E_{XIP\_JMP} = E_{XIP\_SEQ32(RD)} + E_{XIP\_CPU}(jmp) \tag{5.34}$$

$$E_{SDR}(o_i) \leftarrow E_{SDR}(o_i) + v(o_i) \cdot E_{SDR\_JMP} \tag{5.35}$$

$$E_{XIP}(o_i) \leftarrow E_{XIP}(o_i) + v(o_i) \cdot E_{XIP\_JMP} \tag{5.36}$$

Weiterhin werden die Kanten für das Problem generiert, wobei eine Kante $e(o_h, o_i)$ mit der Anzahl der aufeinanderfolgenden Ausführungen von o_h und o_i bewertet wird. Durch diese Kanten kann einerseits modelliert werden, dass bei Ablegen zweier solcher Objekte in einer Partition der *long jump* und die damit verbundenen Energiekosten entfallen. Andererseits wird bei den Problemgrößen für den Flash-Speicher die Kante analog zu Kap.4 mit dem Rückgewinn E^{+}_{OH} bewertet, da der Instruktionsspeicher in SDRAM-Technologie bei Ausführung aus dem Flash-Speicher im *power-down state* bleiben kann.

Entscheidungsvariablen

Aufgrund der zwei alternativ wählbaren Instruktionsspeicher werden für jedes Programmobjekt und jede Kante zwei Entscheidungsvariablen festgelegt.

$$x(o_{i,SDR}) = \begin{cases} 1, & o_i \text{ wird im SDRAM-Instruktionsspeicher abgelegt} \\ 0, & \text{sonst} \end{cases} \tag{5.37}$$

Die positive Entscheidungsvariable einer Kante erlaubt die Bewertung des kombinierten Nutzens zweier gemeinsam verschobener Programmobjekte und somit eine Kostenminderung für die betrachtete Partition.

$$x(o_{h,SDR}, o_{i,SDR}) = \begin{cases} 1, & o_h, o_i \text{ werden im SDRAM-Instruktionsspeicher abgelegt} \\ 0, & \text{sonst} \end{cases} \tag{5.38}$$

Analog interpretiert man die Entscheidungsvariablen $x(o_{i,XIP})$ und $x(o_{h,XIP}, o_{i,XIP})$.

Nebenbedingungen

Die Formulierung der Nebenbedingungen wird durch zwei Auflagen bestimmt. Einerseits muss garantiert werden, dass jedes Objekt in genau einer Partition abgelegt wird, andererseits dürfen die Entscheidungsvariablen von Kanten nur dann gesetzt werden, wenn auch die zugehörigen Entscheidungsvariablen der Objekte gesetzt wurden.

$$\forall\, i \text{ mit } 1 \leq i \leq n : x(o_{i,SDR}) + x(o_{i,XIP}) = 1 \tag{5.39}$$

$$\forall\, h,i \text{ mit } 1 \leq i \leq n,\ 1 \leq h \leq n : x(o_{h,SDR}) + x(o_{i,SDR}) - 2 \cdot x(o_{h,SDR}, o_{i,SDR}) \geq 0 \tag{5.40}$$

Die Nebenbedinung in Gl.5.40 wird analog für den Flash-Speicher aufgestellt. Auf die Formulierung einer Größenbeschränkung kann in der Formalisierung aufgrund der betrachteten Instruktionsspeicher verzichtet werden, da beide Alternativen über eine ausreichend große Kapazität verfügen, die es in allen zur Verfügung stehenden *benchmarks* erlaubt, die komplette Codesektion aufzunehmen. Anhand der Vorbereitungen lässt sich nun das Optimierungsziel formulieren.

Minimiere:

$$\begin{aligned}&\sum_{i=1}^{n} E_{SDR}(o_i) \cdot x(o_{i,SDR}) + E_{XIP}(o_i) \cdot x(o_{i,XIP}) - \sum_{i=1}^{n}\sum_{h=1}^{n} e(o_h, o_i) \cdot {E_{OH}}^{+} \cdot x(o_{h,XIP}, o_{i,XIP}) \\ &- \sum_{i=1}^{n}\sum_{h=1}^{n} e(o_h, o_i) \cdot \left(E_{SDR\_JMP} \cdot x(o_{h,SDR}, o_{i,SDR}) + E_{XIP\_JMP} \cdot x(o_{h,XIP}, o_{i,XIP})\right)\end{aligned} \tag{5.41}$$

unter zuvorgenannten Nebenbedingungen.
Die Ergebnisse und deren Auswertungen werden in den folgenden Kapiteln vorgenommen.

Kapitel 6

Ergebnisse

Im Rahmen dieser Arbeit wurden zwei Optimierungsmethoden formuliert, die mit unterschiedlichen Optimierungskriterien auf eine Energiereduktion in heterogenen Speichersystemen abzielen. Diese werden im folgenden getrennt betrachtet und ausgewertet.

6.1 Reduktion des nichtzugriffsbedingten Energieverbrauchs

Die in Kap.4 formulierte Optimierungsmethode zielt auf die Reduktion des nichtzugriffsbedingten Energieverbrauchs eines Hauptspeichers in SDRAM-Technologie unter Ausnutzung des Power-Down-Modus ab, indem Instruktionen und Variablen auf Scratchpad-Partitionen verlagert werden.
Da der nichtzugriffsbedingte Energieverbrauch einen zustandsabhängigen, konstanten Verbrauch über die Zeit darstellt, wird in dieser Optimierungsmethode die Laufzeit konsequenterweise mitgewichtet. Ergänzt wird diese Optimierungsmethode durch die Berücksichtigung der Beziehung zwischen globalen Variablen und der Basisblöcke, die auf diese zugreifen.

Der Optimierungsmethode stehen partitionierte Scratchpad-Speicher als alternative Instruktions- und Datenspeicher zur Verfügung. Aufgrund der vielzahligen Kombina-

tionen [Hel04] an miteinander kombinierbaren Scratchpad-Partitionen würde sich eine vollständige Untersuchung aller Kombinationen mit jeweiliger Auswertung als sehr aufwendig erweisen. Daher wurden folgende Einschränkungen bei der Validierung der Optimierungsmethode vorgenommen:

- Die Anzahl der miteinander kombinierbaren Scratchpad-Partitionen wird auf maximal zwei beschränkt. Hierzu werden sukzessiv die Speicherkapazitäten um 64 bzw. 256 Byte erhöht. Als obere Grenze werden 1024 Byte angesetzt.
- Es werden nur Scratchpad-Partitionen betrachtet, deren Gesamtkapazität unter der Programmgröße liegt, da sonst die Lösung trivial wäre.
- Es werden *benchmarks* mit ausreichendem Optimierungspotential untersucht, hierzu zählen
  - Programme mit einer entsprechend großen Anzahl an Basisblöcken
  - und Programme mit Zugriffen auf globale Variable.

Bei der Auswahl der Programme werden typische Anwendungsprogramme ausgesucht, die auszugsweise in Tab. 6.1 aufgelistet sind. Zum Vergleich werden Ergebnisse mit dem *bottom-up*-Ansatz [Hel04] bei jeweils gleicher Scratchpad-Partitionierung generiert. Die jeweiligen Gesamtenergieverbräuche der beiden Ansätze werden mit *Power-Down-* und *Bottom-Up Energy* bezeichnet.

Programm	Beschreibung
multi_sort.c	mehrere Sortieralgorithmen
adpcm.c	ADPCM encoder/decoder
me_ivlin.c	Media Applikation (Integer Arithmetik)
fir_viva.c	Finite Impulse Response Filter

Tabelle 6.1: Ausgewählte Programme für die Simulation

Teilweise lassen sich geringfügige Unterschiede in den jeweiligen Ergebnissen ausmachen, die allerdings keine Auswertung erlauben. An dieser Stelle wird davon ausgegangen, dass sich diese Differenzen durch leichte Ungenauigkeiten bei der Problemformulierung ergeben und der IP-Solver abhängig davon andere Kandidaten für die Scratchpad-Partitionen auswählt.

Ergebnisse

Zur Validierung der Optimierungsmethode wurden vier *benchmarks* untersucht: *fir_viva*, *me_ivlin*, *adpcm* und *multi_sort*.

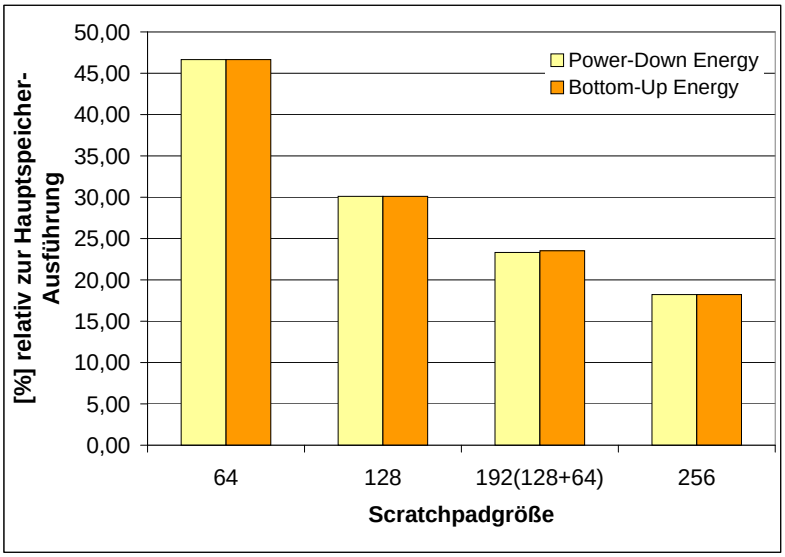


Abbildung 6.1: Energieeinsparung bei me_ivlin.c

Das Programm *me_ivlin* (s. Abb.6.1) stellt einen Sonderfall der ausgewählten Programme dar, da keine globalen Variablen deklariert sind. Somit verlagert sich das Optimierungspotential ausschließlich in die Basisblöcke und deren Ausführungshäufigkeiten. Die Ergebnisse sind dementsprechend bei beiden Optimierungsmethoden nahezu identisch. Einerseits werden die schnellen und energiesparsameren Zugriffsmöglichkeiten auf die Scratchpad-Speicher ausgenutzt, andererseits dadurch die Laufzeit gering gehalten und somit der nichtzugriffsbedingte Energieverbrauch gegenüber der kompletten Ausführung des Programms aus dem Hauptspeicher reduziert.
Bei größter Scratchpad-Kapazität erreichen beide Optimierungsmethoden eine Reduktion des Energieverbrauchs von annähernd 82%.

In den Programmen *fir_viva* und *adpcm* sind globale Variablen deklariert, die (aufgrund ihrer Größe) teilweise die Verlagerung in eine Scratchpad-Partition erlauben würden. Demgegenüber gestaltet sich das Optimierungspotential der Basisblöcke in beiden *benchmarks* gänzlich anders.

Bei *fir_viva* (s. Abb.6.2), bestehend aus sehr wenigen Basisblöcken mit stark variierenden Ausführungshäufigkeiten, ließe sich die Lösung durchaus von Hand generieren. Die Ergebnisse sind in beiden Optimierungsmethoden fast identisch und reduzieren den Energieverbrauch bei größter Scratchpad-Kapazität um 76%.

Durch Hinzunahme einer 64 Byte Partition lassen sich gegenüber dem Ergebnis bei Ausführung mit einem Scratchpad mit einer Kapazität von 128 Byte keine Verbesserungen verzeichnen, was darauf zurückzuführen ist, dass die Methoden den zusätzlichen Speicher nicht nutzen können, da die Programminstruktionen bereits vollständig im Scratchpad abgelegt werden. Erst durch Erweiterung auf eine Kapazität von 256 Byte ist ausreichend Speicherplatz vorhanden um, in beiden Ansätzen eine globale Variable in den Scratchpad zu verschieben.

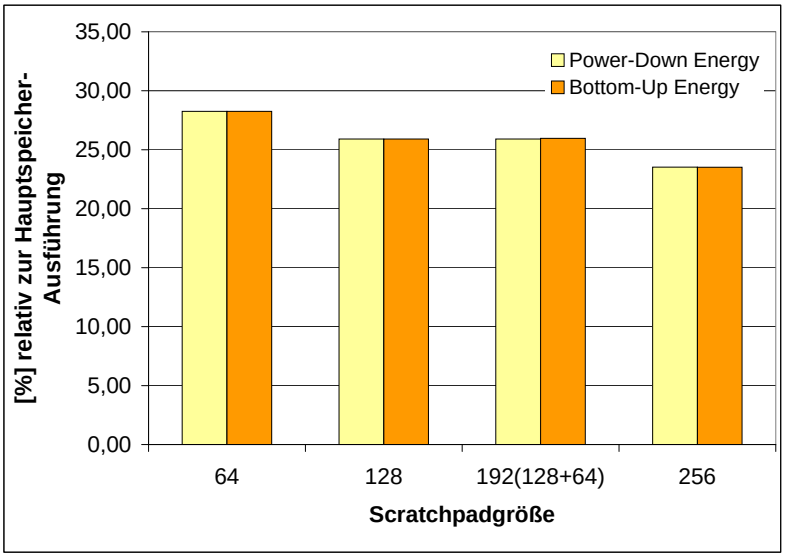


Abbildung 6.2: Energieeinsparung bei fir_viva.c

Obwohl der *benchmark adpcm* (s. Abb.6.3) über mehrere globale Variablen verfügt, verlagert sich das Optimierungspotential in die Programminstruktionen, da die Mehrzahl der Basisblöcke große Ausführungshäufigkeiten vorweist.

Bei der kleinsten Scratchpad-Partition konnte gegenüber dem *bottom-up*-Ansatz 2,8% mehr Energie eingespart werden, obgleich sich die Verteilung in nur zwei Basisblöcken unterscheidet. Diese Reduktion lässt sich zum einen auf die Verkürzung der Laufzeit um fast 3% und zusätzlicher Ausnutzung des *power-down state* von fast 9% des SDRAM-Hauptspeichers begründen.

Zurückzuführen ist das schlechtere Ergebnis beim *bottom-up*-Ansatz auf die Auswahl der Basisblöcke, bei der 4 Byte des Speicherplatzes des Scratchpads ungenutzt bleiben. Bei größter Scratchpad-Partition lässt sich allerdings bei beiden Optimierungsmethoden eine Reduktion des Energieverbrauchs von über 83% erzielen.

Die geringfügigen Abweichungen, die sich in Abb. 6.3 bei den beiden größeren Scratchpad-Kapazitäten erkennen lassen, erlauben keine Interpretation. Die beiden Ansätze schneiden jeweils bei einer Partition um wenige Prozentpunkte besser ab, wobei diese geringfügigen Abweichungen sich auf Ungenauigkeiten bei der Problemaufstellung oder beim *enProfiling* zurückführen lassen.

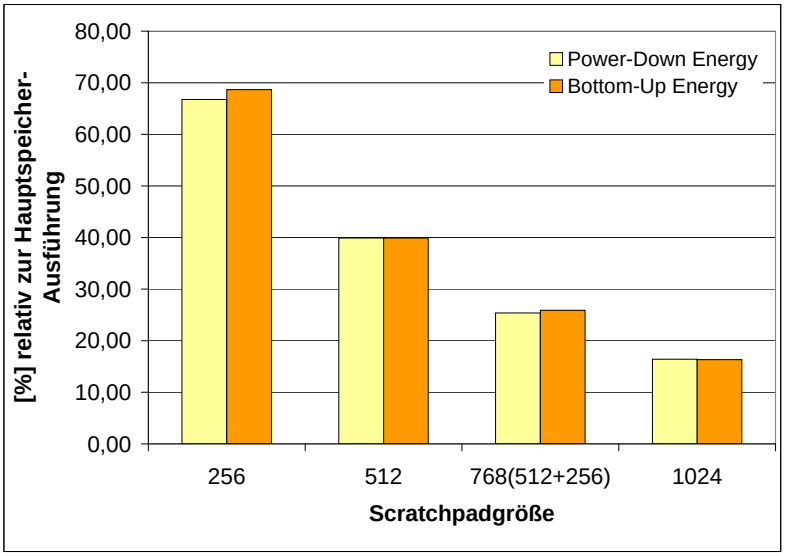


Abbildung 6.3: Energieeinsparung bei adpcm.c

Gänzlich anders gestalten sich die Ergebnisse bei dem Programm *multi_sort* (s. Abb.6.5). Bei drei der vier betrachteten Partitionierungen erzielen beide Optimierungsmethoden

gleiche Ergebnisse. Ein gravierender Unterschied ergibt sich allerdings bei der Untersuchung des *benchmarks* mit einer zusammengesetzten Scratchpad-Kapazität von 768 Byte, bei der gegenüber der Lösung des *bottom-up* Ansatzes 14% an Energie eingespart werden konnten.

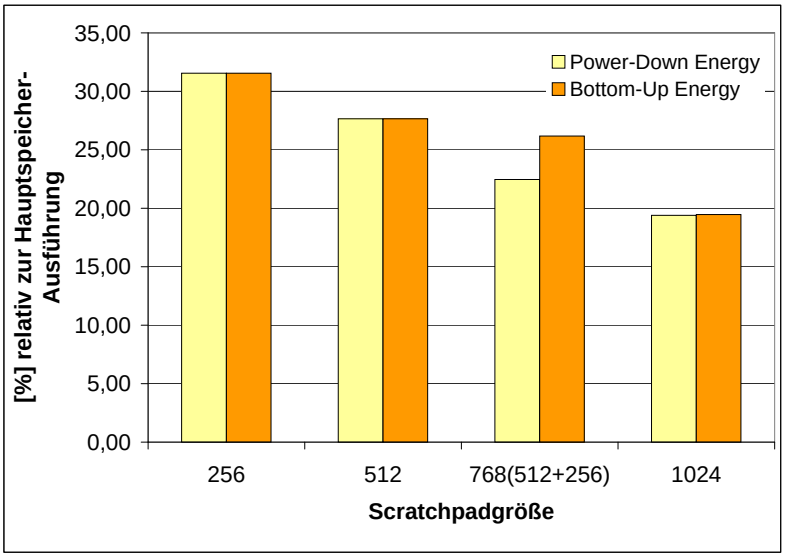


Abbildung 6.4: Energieeinsparung bei multi_sort.c

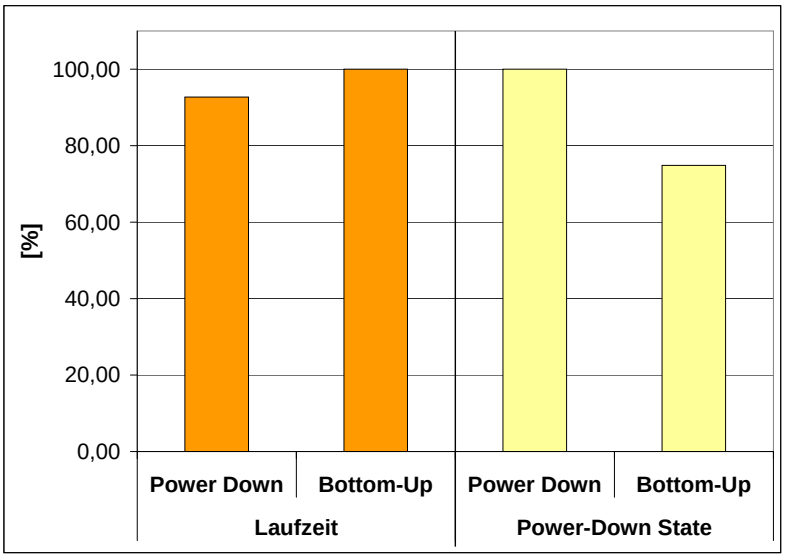


Abbildung 6.5: Vergleich von Laufzeit und *power-down state* bei beiden Ansätzen

Die Begründung für dieses beachtliche Ergebnis liegt in der Modellierung des Zusam-

menhangs zwischen Basisblöcken und deren Zugriff auf globale Variablen, was in [Hel04] vernachlässigt wurde. Während der *bottom-up*-Ansatz versucht, die Scratchpad-Speicher mit Programminstruktionen zu füllen, erkennt der Ansatz zur Reduktion des nichtzugriffsbedingten Energieverbrauchs das Potential in der gemeinsamen Verschiebung von Basisblöcken und Variablen.

In Abb. 6.5 erkennt man, dass der SDRAM bei der Lösung des *bottom-up* Ansatzes ein um 25% geringeres Potential zur Ausnutzung des *power-down state* vorweist. Dieser Sachverhalt lässt sich darauf zurückführen, dass eine häufig frequentierte globale Variable im Hauptspeicher abgelegt und durch regelmäßigen Zugriff auf diese der Zustandswechsel unterbunden wird. Durch das Ablegen der globalen Variable im Scratchpad lässt sich die Laufzeit um 8% gegenüber des *bottom-up* Ansatzes verringern. Steht den Optimierungsmethoden allerdings mehr Speicherkapazität zur Verfügung, erzielen beide Ansätze die gleichen Ergebnisse, die eine Energiereduktion von über 80% gegenüber der Ausführung aus dem Hauptspeicher ermöglichen.

6.2 Reduktion des Energieverbrauchs von SDRAM- und Flash-basierten Speichertechnolgien

Für die Simulation und Validierung der in Kap.5 formulierten Optimierungsmethodik wurden *benchmarks* ausgewählt (s. Tab.6.2), die typische Anwendungsfälle in eingebetteten Systemen darstellen und sich in vorhergehenden Arbeiten bereits als gute Kandidaten bewährt haben. Der *benchmark ref_idct* (Referenzimplementierung) wurde von den Untersuchungen ausgeschlossen, da eine Vielzahl der Speicherzugriffe durch Aufruf von Bibliotheksfunktionen der *arm library* verursacht werden. Um diese Funktionen in der Optimierung berücksichtigen zu können, müsste die *library* entpackt und die einzelnen Objektdateien in die jeweilige Partition gelinkt werden. Aufgrund bestehender Abhängigkeiten der einzelnen Objektdateien gestaltet sich diese Anpassung allerdings als sehr komplex.

Die betrachteten *benchmarks* weisen unterschiedliche Eigenschaften vor, von denen weniger die Gesamtgröße als vielmehr die Anzahl der Programmobjekte, deren Größe und

Programm	Beschreibung
multi_sort.c	mehrere Sortieralgorithmen
encode_combined.c	GSM encoder/decoder
mpegdec.c	MPG encoder
fast_idct.c	Inverse Diskrete Cosinus Transformation
adpcm.c	ADPCM encoder/decoder
me_ivlin.c	Media Applikation (Integer Arithmetik)
fir_viva.c	Finite Impulse Response Filter

Tabelle 6.2: Ausgewählte Programme für die Simulation

die jeweiligen Ausführungshäufigkeiten für die Ergebnisinterpretation interessant sind. Die Ausführungshäufigkeiten in Kombination mit der Größe des jeweiligen Programmobjekts haben entscheidenden Einfluss auf die Auswahl des ILP-Solvers, so dass große Programmobjekte, die nur selten ausgeführt werden, aufgrund der zusätzlichen Kosten, die durch den Kopiervorgang entstehen würden, für die Flash-Ausführung prädestiniert sind.

Häufig ausgeführte Programmobjekte werden demnach in den schnellen SDRAM verschoben, weil der zeitliche Gewinn der Ausführung die zusätzlichen Kosten durch den Kopiervorgang kompensiert. Weiterhin werden die Entscheidungen des IP-Solvers erheblich durch die angenommenen Zeilenaktivierungen bei Ausführung des betrachteten Basisblocks aus dem SDRAM beeinflusst. So kann es durchaus sein, dass ein Basisblock trotz häufiger Ausführung nicht in den SDRAM verschoben wird, da er bspw. aufgrund seiner geringen Größe zu geringes Potential für den sequentiellen Zugriffsmodus bietet.

Anhand der *benchmarks* wird nun das Potential der in Kap.5 formulierten Optimierungsmethode in einem Speichersystem bestehend aus einem Instruktions- und Datenspeicher in SDRAM-Technologie sowie einem Flash-Speicher in NOR-Technologie mit XIP-Funktionalität untersucht.

Da der Flash-Speicher mit einer *intrapage access time* $t_{APA} = 20ns$ gegenüber dem SDRAM einen relativ langsamen sequentiellen Zugriff erlaubt, wurde in einer zweiten Testreihe t_{APA} auf $10ns$ reduziert, um die Aussage zu überprüfen, dass diese Kenngröße maßgeblichen Einfluss auf das Optimierungsergebnis ausübt.

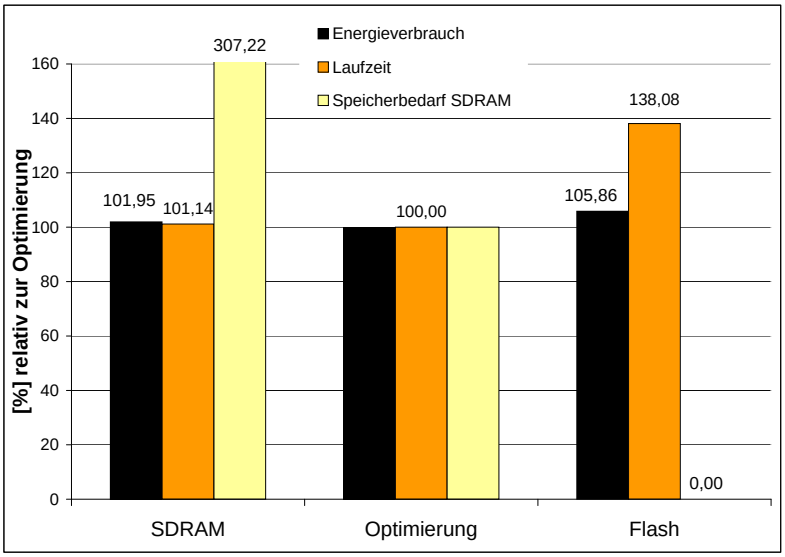


Abbildung 6.6: Auswertung von mpegdec.c mit $t_{APA} = 20ns$

In Abb. 6.6 sind die Ergebnisse bei ausschließlicher Verwendung von SDRAM bzw. Flash als Instruktionsspeicher in Prozent relativ zur optimierten Variante, die beide Speichertypen kombiniert, eingetragen. In den folgenden Abbildungen wird auf die Darstellung der Optimierung verzichtet, da diese stets als Referenzmaß dient und somit immer auf 100% skaliert wird.

Die Ergebnisse für die erste Testreihe sind teilweise ähnlich und erlauben somit eine Gruppeneinteilung und gemeinsame Berurteilung. Einzige Ausnahme bildet hier das Programm *mpegdec* (s. Abb.6.6), das im Vergleich zu den anderen ausgewählten Programmen bedeutend mehr Speicherplatz benötigt und aus wesentlich mehr Programmobjekten besteht.

Obgleich der Anteil der Programmobjekte im Flash-Speicher bei der optimierten Variante sehr groß ist, verbraucht die Ausführung des gesamten Programms aus dem Flash-Speicher fast 6% mehr Energie und hat eine um knapp 40% längere Laufzeit. Von den fast 70 Funktionen werden zwei Drittel komplett aus dem Flash-Speicher ausgeführt, da die Ausführungshäufigkeiten der jeweiligen Basisblöcke zumeist im einstelligen Bereich liegen. Gegenüber der SDRAM-Ausführung kann bei der optimierten Variante der Speicherplatzbedarf in der Instruktionspartition in SDRAM-Technologie um 67% reduziert werden.

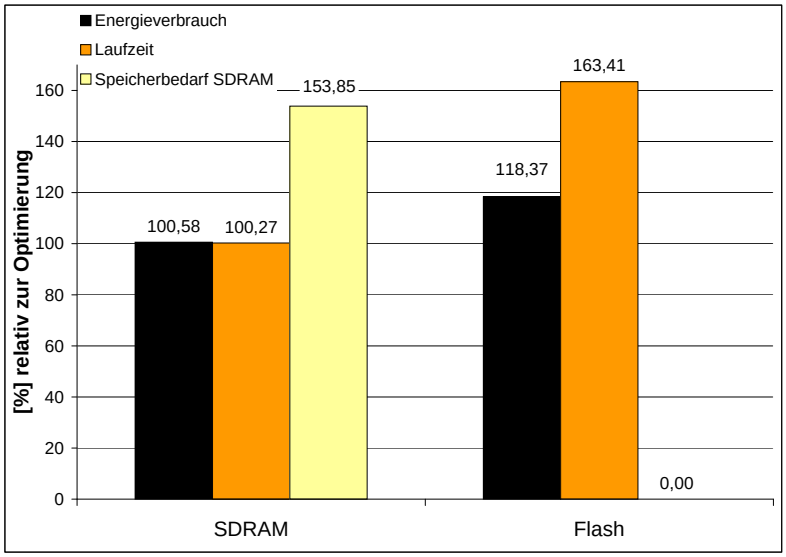


Abbildung 6.7: Auswertung von me_ivlin.c mit $t_{APA} = 20ns$

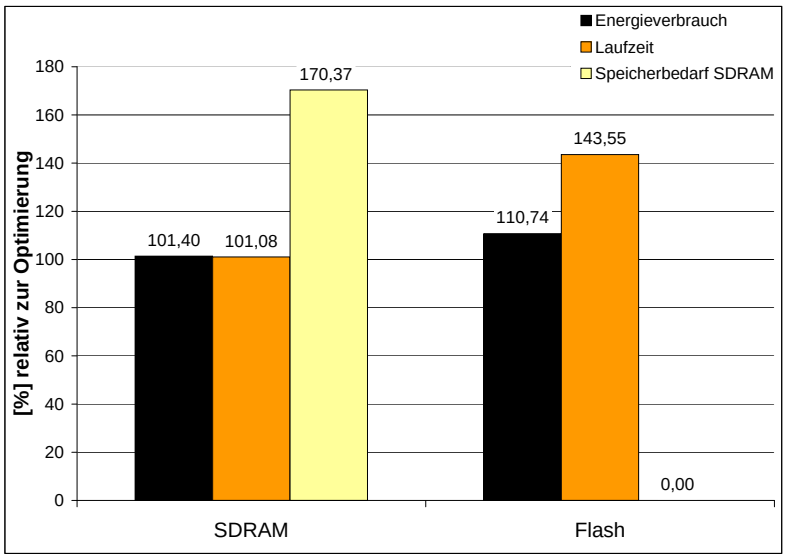


Abbildung 6.8: Auswertung von adpcm.c mit $t_{APA} = 20ns$

Durchaus gute Ergebnisse konnte die Optimierungsmethode auch bei den *benchmarks me_ivlin*, *adpcm* und *fir_viva* (s. Abb.6.7-6.9) gegenüber der Flash-Ausführung erzielen. Auf einen großen Teil der Instruktionen kann sequentiell zugegriffen werden. Da der Flash-Speicher im *intrapage access* gegenüber dem *burst access mode* einen zusätzlichen Taktzyklus für die Bereitstellung der Folgeadresse benötigt, lassen sich Einsparungen der Laufzeit von 31-38% erzielen. Obgleich der SDRAM eine deutlich höhere Leistungs-

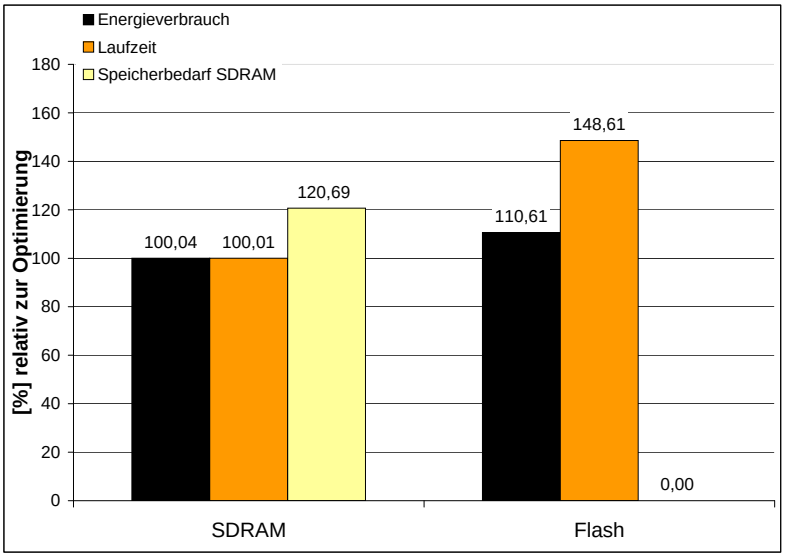


Abbildung 6.9: Auswertung von fir_viva.c mit $t_{APA} = 20ns$

aufnahme vorweist, konnten weiterhin laufzeitbedingt Energieeinsparungen von 9-15% erwirtschaftet werden. Gegenüber der SDRAM-Ausführung ließen sich weder bei Laufzeit noch Energieverbrauch nennenswerte Einsparungen ausmachen. Als großer Vorteil bei der optimierten Variante wurde bedeutend weniger Speicherplatz vom Instruktionsspeicher in SDRAM-Technologie beansprucht. Im Durchschnitt wurde ein Drittel des Speicherplatzes eingespart und somit der Bedarf auf lediglich 68% gesenkt.

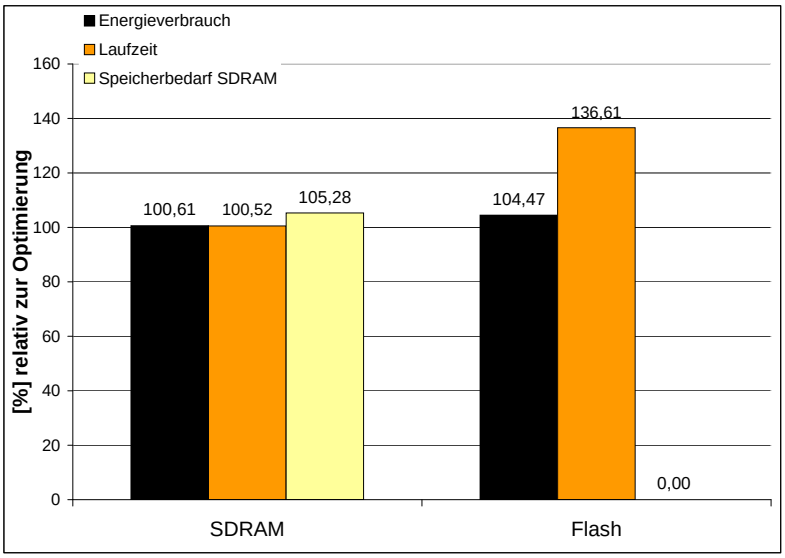


Abbildung 6.10: Auswertung von fast_idct mit $t_{APA} = 20ns$

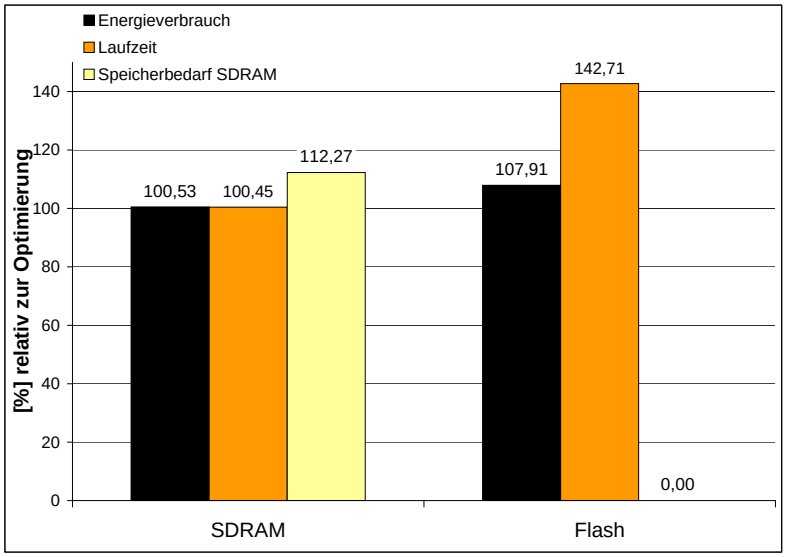


Abbildung 6.11: Auswertung von multi_sort.c mit $t_{APA} = 20ns$

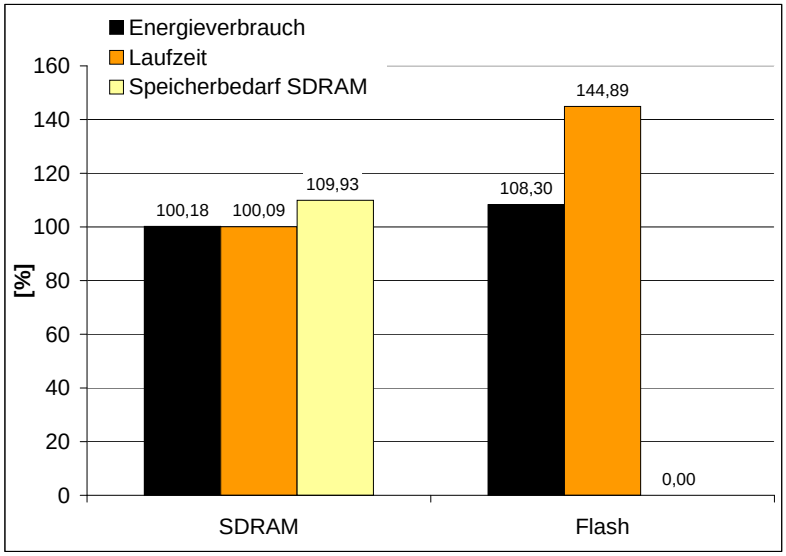


Abbildung 6.12: Auswertung von encode_combined.c mit $t_{APA} = 20ns$

Ähnliche Ergebnisse konnten bei den *benchmarks fast_idct*, *multi_sort* und *encode_combined* hinsichtlich der Laufzeit (Einsparungen von durchschnittlich 29%) gegenüber der Flash-Ausführung erzielt werden (s. Abb.6.10-6.12) Demgegenüber konnte der Energieverbrauch im Durchschnitt lediglich um 6% reduziert werden. Gegenüber der SDRAM-Ausführung konnten keine nennenswerten Verbesserungen erkannt werden.

Gänzlich anders gestalten sich die Ergebnisse bei der zweiten Testreihe mit einem Flash-Speicher, der einen deutlich schnelleren sequentiellen Zugriff im *intra-page access* ermöglicht.
Auch hier lassen sich Gruppen bilden, wobei erstere sich dadurch auszeichnet, dass die Optimierungsmethode die komplette Codesektion in den Flash-Speicher verlagert und somit kein SDRAM-Instruktionsspeicher benötigt wird.

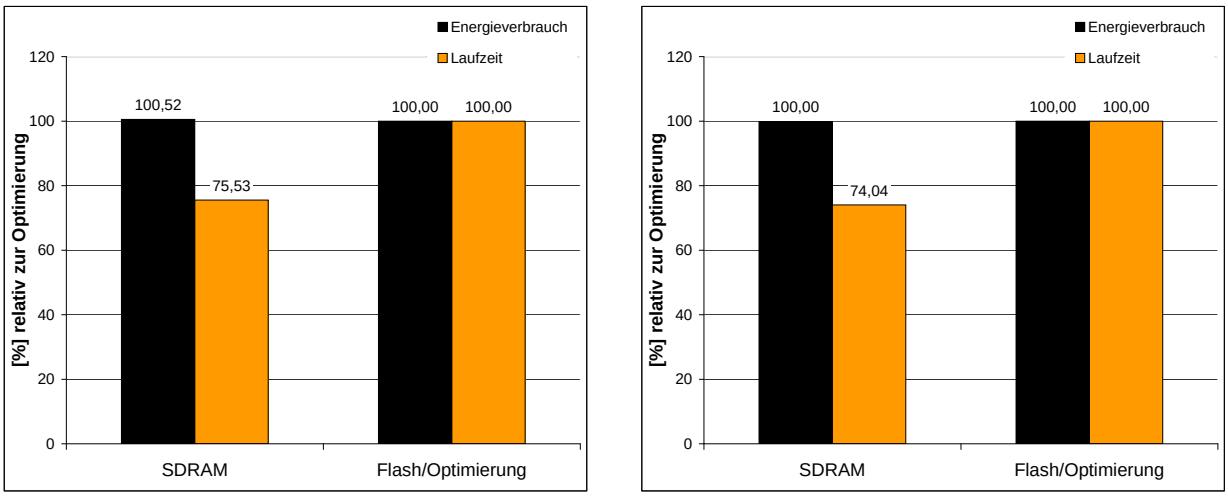


Abbildung 6.13: Auswertung von encode_combined.c und fir_viva.c mit $t_{APA} = 10ns$

Bei den benchmarks *encode_combined* (s.Abb.6.13, links) und *fir_viva* (s.Abb.6.13, rechts) entscheidet sich der IP-Solver für die Verlagerung der vollständigen Codesektion in den Flash-Speicher. Hierdurch werden unter Akzeptanz einer bedeutend höheren Laufzeit minimale bzw. vernachlässigbare Verminderungen des Energieverbrauchs gegenüber der SDRAM-Ausführung erzielt. Dieses Ergebnis lässt sich darauf zurückführen, dass in der Optimierungsmethode ausschließlich das Ziel der Energiereduktion vorgegeben wird.

Bei dem benchmark *fast_idct* (s.Abb.6.14) wird fast das gesamte Programm aus dem Flash-Speicher ausgeführt. Die Anzahl der Kandidaten, die in den SDRAM verschoben werden ist im Verhältnis zur Gesamtgröße des Programms mit 4% sehr gering. Gegenüber der SDRAM-Ausführung konnte der Energieverbrauch um 4% gesenkt werden. Diese Reduktion lässt sich auf die wenigen Basisblöcke, die in den SDRAM kopiert werden, zurückführen. Durch das Ziel der Energiereduktion wird allerdings eine um fast 18% längere Laufzeit erzwungen, gegenüber der Flash-Ausführung lässt sich

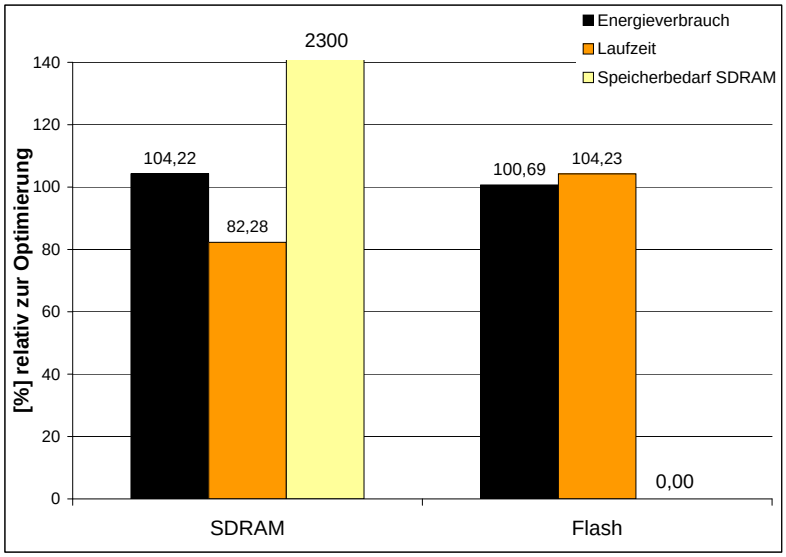


Abbildung 6.14: Auswertung von fast_idct mit $t_{APA} = 10ns$

allerdings die Laufzeit um 4% verkürzen.

Die *benchmarks multi_sort*, *me_ivlin* und *mpegdec* (s. Abb.6.15-6.17) erlauben aufgrund ihrer Ergebnisse ebenfalls eine gemeinsame Auswertung.

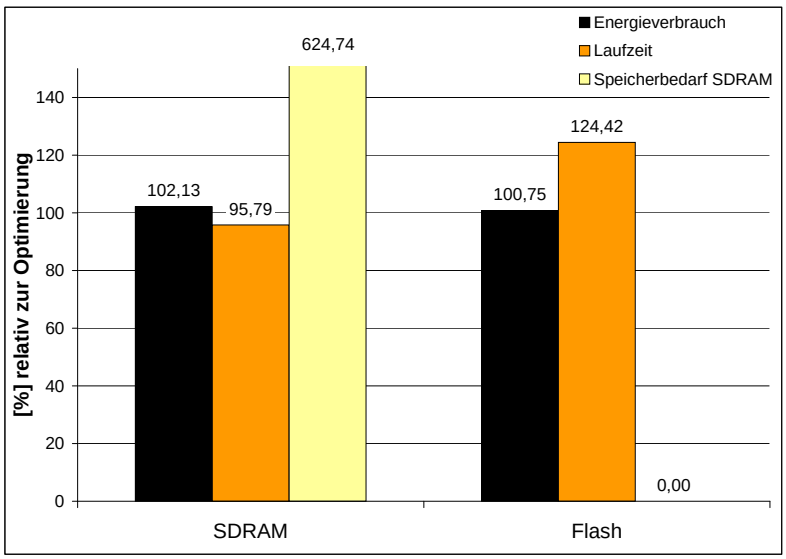


Abbildung 6.15: Auswertung von mpegdec.c mit $t_{APA} = 10ns$

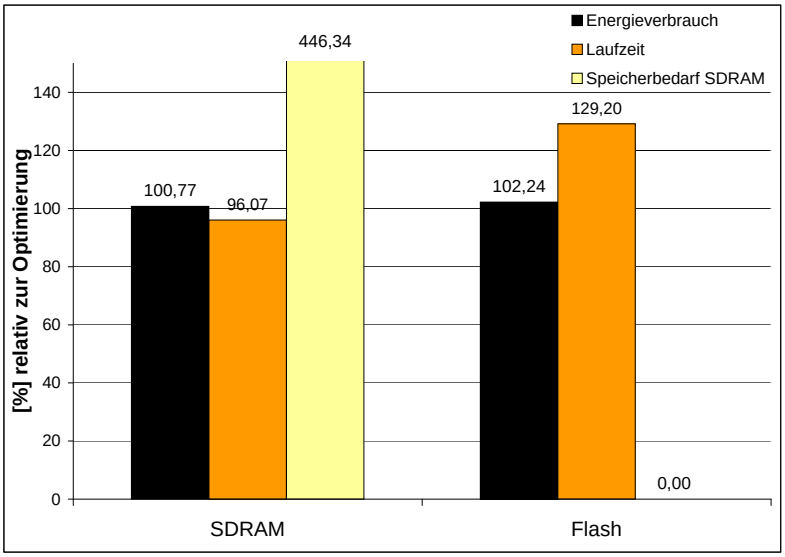


Abbildung 6.16: Auswertung von multi_sort.c mit $t_{APA} = 10ns$

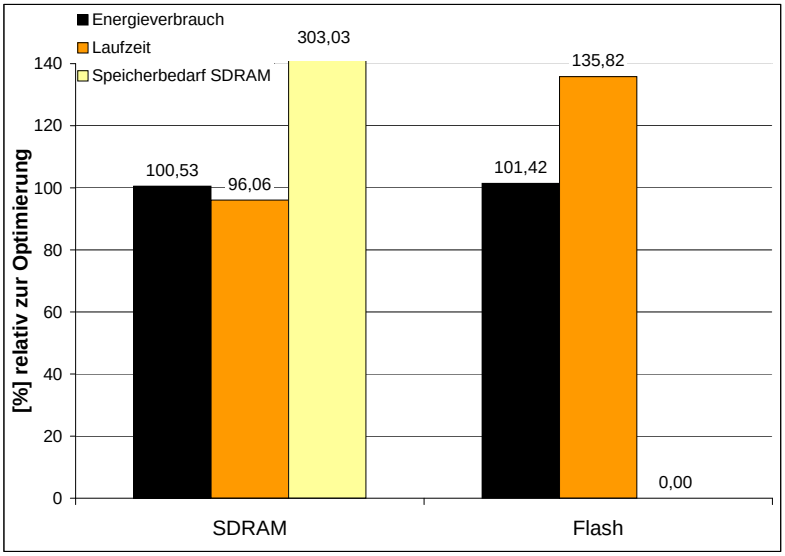


Abbildung 6.17: Auswertung von me_ivlin.c mit $t_{APA} = 10ns$

Bei den drei Programmen lassen sich bei der optimierten Variante Reduktionen des Energieverbrauchs gegenüber den beiden Standard-Lösungen erkennen. Je nach *benchmark* liegen die Ersparnisse gegenüber einer der beiden Standard-Lösungen im Bereich von 2%. Die Laufzeit der Flash-Ausführung liegt trotz schnellerem Flash-Speicher mit durchschnittlich 30% noch deutlich über der optimierten Lösung.
Gegenüber der SDRAM-Ausführung wird ein vertretbarer Anstieg der Laufzeit von

4% bei der optimierten Variante in Kauf genommen. Der Speicherbedarf im Instruktionsspeicher in SDRAM-Technologie kann in allen Fällen beachtlich gesenkt werden, so dass im besten Fall nur noch 16% des Speicherplatzes im SDRAM benötigt wird.

Der *benchmark adpcm* (s.Abb.6.18) stellt einen Sonderfall dar, da die optimierte Variante die kürzeste Laufzeit erzielt. Dieses Ergebnis lässt sich auf die durchschnittlichen Ausführungshäufigkeiten zurückführen, die gegenüber zu den anderen untersuchten Programmen gering ausfallen. Durch die geringe Laufzeit des *benchmarks* ergibt sich eine stärkere Gewichtung der Kopierenergie.

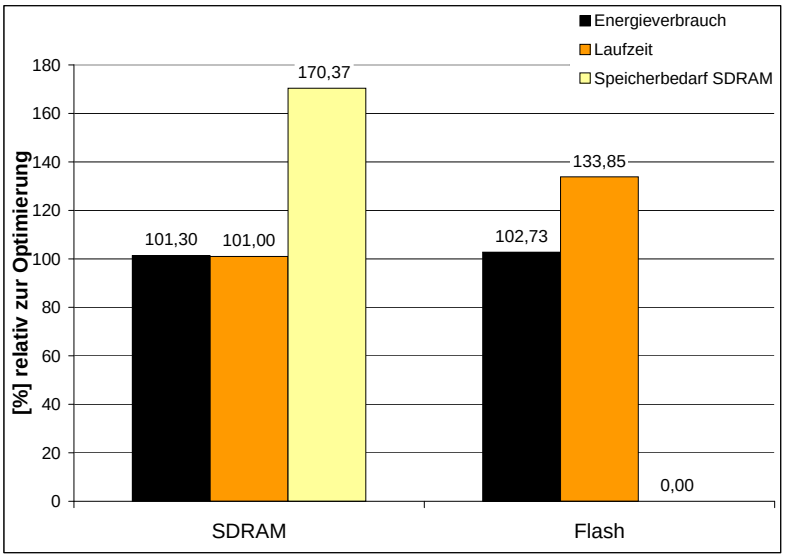


Abbildung 6.18: Auswertung von adpcm.c mit $t_{APA} = 10ns$

Um den gravierenden Einfluss von Ausführungshäufigkeiten auf die Lösung des ILP-Solvers zu verdeutlichen, wurde der *benchmark biquad_N_sections* mit einem Flash-Speicher mit $t_{APA} = 20ns$ untersucht.
Hierbei handelt es sich um eine rekursive Filter-Applikation zweiter Ordnung, die eine typische DSP-Anwendung darstellt. Um die Ausführungshäufigkeiten einzelner Basisblöcke zu erhöhen, werden Filter höherer Ordnung durch Modifikation des Parameters N erzeugt.
Die Ergebnisse in Abb. 6.19 zeigen, dass bei kleinen Ausführungshäufigkeiten der Mehraufwand durch die Kopiervorgänge maßgeblichen Einfluss auf den Energieverbrauch

ausübt, so dass für $N = 25$ die optimierte Lösung der Flash-Ausführung entspricht. Für $N = 100$ bietet der *benchmark* gegenläufiges Potential für die SDRAM- und Flash-Ausführung, so dass die optimierte Lösung deutliche Einsparungen in Energieverbrauch und Laufzeit gegenüber beiden Varianten erzielt. Für $N = 1000$ wird bei der Flash-Ausführung die erhöhte Laufzeit gegenüber der SDRAM-Ausführung deutlich und wirkt sich negativ auf den Energieverbrauch aus.

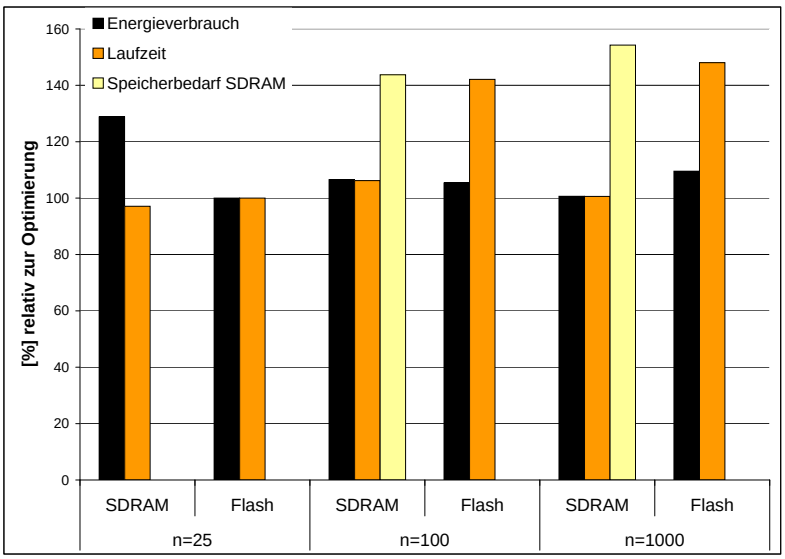


Abbildung 6.19: Auswertung von biquad_N_sections.c mit $t_{APA} = 20ns$

Eine Zusammenfassung der Ergebnisse wird im folgenden Kapitel vorgenommen.

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden moderne Speicher in SDRAM- und Flash-Technologie untersucht, entsprechende Energiemodelle zur Bewertung dieser Speicher entwickelt und anschließend in die Arbeitsumgebung integriert.

Gegenüber dem Energieverbrauch von statischen und Flash-Speichern, der ausschließlich durch Zugriffe verursacht wird, gestaltet sich das Energiemodell zur Bewertung von SDRAM-Speichern wesentlich komplexer und setzt sich aufgrund technologischer Eigenschaften aus folgenden verschiedenen Komponenten zusammen:

- Energieverbrauch durch Bank- und Zeilenaktivierung
- zugriffsbedingter Energieverbrauch
- zustandsabhängiger, konstanter, nichtzugriffsbedingter Energieverbrauch
- Energieverbrauch durch *Refresh* der Speicherinhalte

Weiterhin wurde die zur Verfügung stehende Umgebung um die Berücksichtigung sequentieller Zugriffe auf benachbarte Speicheradressen erweitert, um die Geschwindigkeitsvorteile moderner Speicher ausschöpfen zu können. Man unterscheidet:

- sequentielle Zugriffe im *burst-mode* bei synchronen Speichern
- sequentielle Zugriffe im *page-mode* bei asynchronen Speichern

Nichtflüchtige Speicher, die eine ständige Erhaltung von Daten und Programmen garantieren, waren in vorhergehenden Arbeiten zwar Bestandteil des jeweils betrachteten Systems, wurden aber von einer weitergehenden Betrachtung ausgeschlossen. Das *code shadowing*, bei dem zu jedem Programmstart zunächst Daten und Instruktionen in die entsprechenden Speicherpartitionen übertragen werden, blieb bei der Kalkulation des Energieverbrauchs i.d.R. unberücksichtigt. Hinzukommt, dass moderne nichtflüchtige Speicher in Flash-Technologie den *random access* unterstützen, so dass folgende Anpassungen notwendig waren:

- Berücksichtigung von Kopiervorgängen und Bewertung der damit verbundenen Energieverbräuche
- Nutzung der XIP-Funktionalität von Flash-Speichern in NOR-Technologie zur direkten Instruktionsausführung und Einsparung des Kopiervorgangs

Basierend auf den Erkenntnissen, die durch Programmsimulation und Auswertung der Energieverbräuche gewonnen werden konnten, wurden zwei Optimierungsmethoden mit dem Ziel der compilergesteuerten Energiereduktion unter Ausnutzung heterogener Speichersysteme entwickelt, wobei erstere auf den Ergebnissen von [Hel04] aufbaut.

- SDRAM-Speicher als Hauptspeicher und partitionierte Scratchpad-Speicher mit dem Ziel der Minimierung des Energieverbrauchs unter Ausnutzung des *power-down*-Modus von SDRAM-Speichern und Modellierung des Zusammenhangs zwischen Basisblöcken und globalen Variablen
- partitionierte SDRAM-Speicher als Instruktions- und Datenspeicher und asynchrone Flash-Speicher in NOR-Technologie mit dem Ziel der Minimierung des Energieverbrauchs unter Ausnutzung des *power-down*-Modus unter Berücksichtigung sequentieller Speicherzugriffe und Ausnutzung der XIP-Funktionalität zur Vermeidung von Kopiervorgängen.

Die erzielten Ergebnisse werden im folgenden zusammengefasst.

7.1 Fazit

Durch Berücksichtigung der konstanten Energieverbräuche der SDRAM-Speicher wurde in beiden Optimierungsmethoden der Laufzeitkomponente eine stärkere Gewichtung zugeteilt. Daraus lassen sich zwei grundsätzliche Feststellungen ableiten, die sich für die Lösung der jeweiligen Probleme ergeben:

- Durch Reduktion der Laufzeit können die konstanten, nichtzugriffsbedingten Energieverbräuche des SDRAM im *idle state* reduziert werden.
- Durch geschickte Nutzung des Flash-Speichers unter Akzeptanz einer erhöhten Laufzeit ergibt sich die Möglichkeit des Wechsels in den *power-down state* des SDRAM, wodurch der nichtzugriffsbedingte Energieverbrauch im SDRAM reduziert werden kann.

Durch Betrachtung von sequentiellen Speicherzugriffen wurden Geschwindigkeitsvorteile von SDRAM-Speichern ermöglicht. Auch hier lassen sich weitere Feststellungen für SDRAM-Speicher formulieren

- SDRAM-Speicher schöpfen ihre Geschwindigkeitsvorteile nur bei Anbindung an einen adäquat schnellen Speicherbus aus
- Durch den sequentiellen Zugriff kann die erneute Zeilenaktivierung entfallen und der damit verbundene Energieverbrauch im SDRAM drastisch gesenkt werden.

Weiterhin wurden Flash-Speicher betrachtet, die mit XIP-Funktionalität ausgestattet, durchaus eine mögliche Konkurrenz zum SDRAM als Instruktionsspeicher darstellen. Von einer Verwendung als Datenspeicher wurde aufgrund der Schreibhäufigkeiten in den betrachteten *benchmarks* abgesehen.
Folgende Auflistung stellt die prinzipiellen Feststellungen, die im Rahmen dieser Arbeit zu Flash-Speichern angestellt wurden, zusammen.

- Flash-Speicher weisen einen deutlich geringeren Energieverbrauch beim Speicherzugriff als SDRAM-Speicher vor. Insbesondere die nichtzugriffsbedingten Energieverbräuche sind vernachlässigbar klein.

- Die Möglichkeit des sequentiellen Zugriffs wird bei asynchronen Speichern im *page-mode* unterstützt und reduziert somit die Zugriffszeit auf Folgeadressen innerhalb einer Seite. Die Zugriffszeit liegt allerdings deutlich über der des sequentiellen Zugriffs im *burst-mode* bei synchronen Flash-Speichern.
- Flash-Speicher, die den synchronen Betrieb unterstützen, ermöglichen den Zugriff im *burst-mode*, wobei die maximale Taktfrequenz im synchronen Betrieb immer noch deutlich unter der von SDRAM-Speichern liegt.

In Abb. 7.1 sind die durchschnittlichen prozentualen Verteilungen der Energieanteile bei jeweiliger Programmausführung abgebildet. In diesem Zusammenhang lässt sich sehr deutlich erkennen, dass bei einem Flash-Speicher mit $t_{APA} = 10ns$ die optimierte Programmausführung einen Kompromiss zwischen Laufzeit und Speicherenergieverbrauch eingeht.

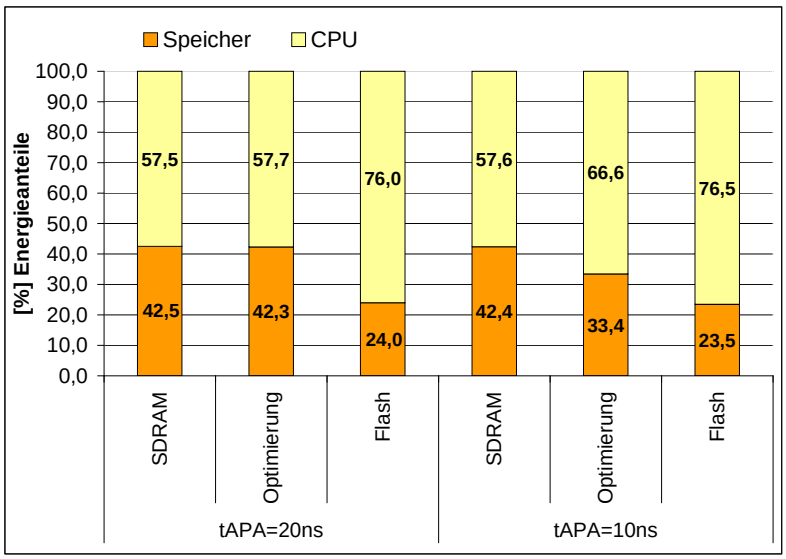


Abbildung 7.1: Prozentuale Energieanteile

Die Zusammenfassung der Ergebnisse ist in Abb. 7.2 dargestellt. Man erkennt, dass die Ausführung eines Programms aus dem Flash-Speicher mit $t_{APA} = 20ns$ im Durchschnitt 45,2% mehr Laufzeit beansprucht als die optimierte Lösung. Demgegenüber gering fällt der zusätzliche Energieverbrauch von 9,4% aus, der sich durch die erhöhte Laufzeit begründet und unter Betrachtung von Abb. 7.1 eindeutig der CPU zuschreiben lässt. Gegenüber der SDRAM-Ausführung konnten keine nennenswerten Einsparungen hinsichtlich der Laufzeit und des Energieverbrauchs erzielt werden. Einen großen Vorteil

der Optimierung stellt allerdings der geringere Speicherbedarf im SDRAM dar. Auf die Darstellung eines Mittelwerts wird hier aufgrund der sehr stark abweichenden Ergebnisse verzichtet. Der Speicherbedarf des Instruktionsspeichers in SDRAM-Technologie konnte im schlechtesten Fall um 5%, im besten Fall um 67% gesenkt werden.

Bei der zweiten Untersuchung wurde ein Flash-Speicher gewählt, der mit $t_{APA} = 10ns$ einen bedeutend schnelleren sequentiellen Zugriff ermöglicht. Durch Modifikation dieser Kenngröße wird die Zugriffsdauer im sequentiellen Modus beim Flash-Speicher um einen Taktzyklus reduziert. Die Auswertung hat ergeben, dass der Optimierer sich in zwei Fällen für die ausschließliche Ausführung des Programms aus dem Flash-Speicher entschieden hat. Durch Vorgabe des Minimierungsziels wurde zwar stets die Lösung mit dem geringsten Energieverbrauch gewählt, die Laufzeit allerdings gravierend verschlechtert. Der Energieverbrauch der SDRAM-Ausführung lag im Durchschnitt nur 1,6% und die der Flash-Ausführung nur 1,1% über dem Verbrauch der optimierten Lösung. Die Laufzeit der Flash-Ausführung lag weiterhin deutlich mit durchschnittlich 18,2% über der optimierten Lösung, demgegenüber lag die Laufzeit der SDRAM-Ausführung 11,3% unter der energieoptimierten Programmausführung. Der Speicherbedarf im SDRAM konnte allerdings drastisch reduziert werden. Sieht man von den beiden Programmen, bei denen sich der Optimierer für die Flash-Ausführung entscheidet, ab, so ergeben sich Reduktionen des Speicherbedarfs von 41% bis 96%.

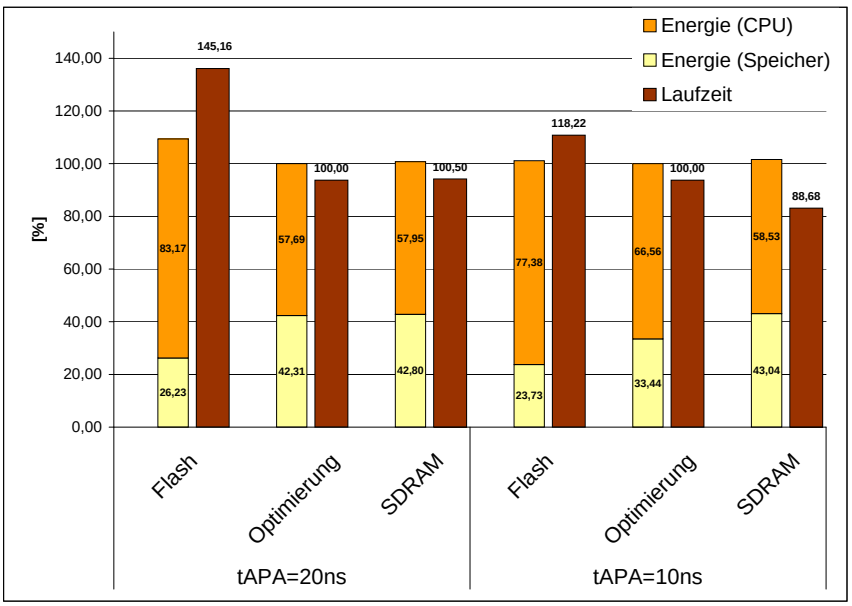


Abbildung 7.2: Ergebnisse der Optimierung

Betrachtet man nun unter Hinzunahme der Ergebnisse das Optimierungspotential für den Compiler, so ergeben sich bei gemeinsamer Betrachtung beider Speichertechnologien folgende Festellungen:

- Beide Technologien verfügen zwar über hohe Speicherkapazität, sind aber in ihren sonstigen Eigenschaften gegenläufig. SDRAM-Speicher sollten dann bevorzugt werden, wenn eine geringe Laufzeit Ziel der Optimierung ist. Demgegenüber weisen Flash-Speicher einen deutlich geringeren Energieverbrauch beim Speicherzugriff vor.
- Je höher die Leistungsaufnahme des Prozessors ist, desto negativer wirkt sich eine längere Laufzeit durch Flash-Speicher auf den Gesamtenergieverbrauch aus.
- Das Optimierungspotential durch den Compiler ist stark von den jeweiligen elektrischen und zeitlichen Kenngrößen der betrachteten Speicher abhängig. Die Ergebnisse variieren entsprechend der Gewichtung der jeweiligen Vor- und Nachteile, so dass die Modifikation einzelner Kenngrößen vorherige Ergebnisse grundsätzlich verändern kann.

Der Abschluss dieses Kapitels wird durch den folgenden Ausblick auf mögliche weiterführende Arbeiten gebildet.

7.1.1 Ausblick

Im ersten Teil der Arbeit wurde eine Optimierungsmethode implementiert, die den Zusammenhang von globalen Variablen und den darauf zugreifenden Basisblöcken berücksichtigt. Hierdurch konnte in einem Fall eine drastische Senkung des Energieverbrauchs erzielt werden. Aufgrund der beschränkten Kapazitäten der Scratchpad-Speicher und der oft über diesen Kapazitäten liegenden Größe der globalen Variablen, ergibt sich allerdings nur eine bedingte Ausnutzbarkeit für die Optimierungsmethode. Durch die Zerlegung von globalen Variablen in einzelne Sektionen und die gegenseitige Verdrängung dieser Sektionen zur Laufzeit kann ein bedeutend größeres Optimierungspotential erzeugt werden. Dies ist bereits Bestandteil der laufenden Forschungsarbeiten am LSXII.

Die Auswertung der Ergebnisse, der im zweiten Teil der Arbeit implementierten Optimierungsmethode, ergibt zwei Ansatzmöglichkeiten für weiterführende Arbeiten:

- Durch die Zielvorgabe der Energiereduktion wurden teilweise beachtlich längere Laufzeiten in Kauf genommen, die im Verhältnis nicht gerechtfertigt sind. Durch zusätzliche Berücksichtigung dieser Komponente entstehen Optimierungsprobleme mit mehreren Kostenfunktionen (multi-objektive Optimierung, [Deb04]), die sich mit evolutionären Algorithmen effizient lösen lassen.
- Durch Änderung der zeitlichen Kenngröße t_{APA} konnte im Rahmen einer Sensitivitäts-Analyse der erhebliche Einfluss dieses Parameters auf das Ergebnis der Optimierung festgestellt werden. Aus diesem Grund werden weitere Untersuchungen durch Modifikation von elektrischen und zeitlichen Kenngrößen angeregt.

Aufgrund der Betrachtung moderner Speicher wurde eine Erhöhung der Taktfrequenz des Speicherbusses (und der CPU) vorgenommen. Die Geschwindigkeit moderner ES fordert adäquat schnelle Speichertechnologien, die derzeit durch Mobile DDR-SDRAM und Flash-Speicher mit *burst-mode*-Unterstützung gestellt werden. Durch geringfügige Anpassung der Energiemodelle lassen sich auch diese Speicher untersuchen. Ebenso wird die Implementierung eines Energiemodells für pseudo-statische Speicher (*unitransistor* SRAM) angeregt, da diese Speichertechnologie aufgrund ihrer Eigenschaften häufig in mobilen Endgeräten eingesetzt werden.

Weiterhin wurden sequentielle Zugriffe und deren Auswirkungen auf den Speicherenergieverbrauch untersucht. An dieser Stelle wurden Restriktionen festgelegt, die das Potential des sequentiellen Zugriffsmodus einschränken.

- Jeder Zugriff auf die erste Instruktion eines Basisblocks wurde als *random access* bewertet (irregulärer *random access*). Diese Annahme war notwendig, um einem Basisblock ein festes Zugriffsprofil unabhängig von seiner Speicherposition zuordnen zu können. Eine Möglichkeit, die sequentielle Zugriffsrate auf einen Basisblock zu erhöhen, ergibt sich durch geschickte Positionierung der Basisblöcke durch den Linker. Stark frequentierte und/oder große Basisblöcke sollten nicht zeilenübergreifend im SDRAM abgelegt werden, um die Anzahl der Zeilenaktivierungen gering zu halten. Weiterhin kann die Berücksichtigung sequentieller

Speicherzugriffe bereits während der Codegenerierung stattfinden. Hierzu könnte untersucht werden, wieviel Einsparungen *Traces* [TY96] bringen, bei denen möglichst häufig sequentiell vom Speicher gelesen wird.

- Sofern eine Instruktion einen Speicherzugriff verursacht, wird derzeit konservativ davon ausgegangen, dass durch den Datenzugriff der sequentielle Zugriff unterbrochen wird. Die Folgeinstruktion wird dementsprechend als *random access* bewertet und somit eine erneute Zeilenaktivierung angenommen. Moderne Speicher in SDRAM-Technologie umgehen dieses Problem durch den *clock suspend mode*. Erfolgt während des Zugriffs im *burst mode* ein Zugriff auf einen anderen Speicher, wird die synchrone Logik des SDRAM 'eingefroren'. Der aktuelle Inhalt der Ausgabepuffer bleibt erhalten und die Autoinkrementierung durch den *burst counter* wird unterbunden. Wird der Zugriff auf den SDRAM fortgesetzt, erfolgt der nächste Zugriff direkt im *burst mode*, eine erneute Zeilenaktivierung entfällt.

Während des Speicherzugriffs wurde in der CPU eine konstante Leistungsaufnahme angenommen, die sich insbesondere bei den langsameren Flash-Speichern negativ auf den Gesamtenergieverbrauch ausgewirkt hat. Der Energieverbrauch, den die CPU im *idle state* verursacht, wurde als konstant angenommen.
Mit dem *dynamic voltage scaling* steht ein Ansatz zur Verfügung, der an genau dieser Stelle zur Senkung des Energieverbrauchs in der CPU ansetzt. Moderne ARM-Prozessoren verfügen hierzu analog zu den betrachteten SDRAM-Speichern über ein *power managment*.
Bei der Weiterverwendung des bestehenden Prozessorenergiemodells bei Annahme eines höheren CPU-Taktes, wurde von der Skalierung der Kenngrößen abgesehen, da sonst die Leistungsaufnahme der CPU überbewertet worden wäre. Dementsprechend wird eine Anpassung dieses Modells unter Berücksichtigung moderner Fertigungstechnologien vorgeschlagen.

Abbildungsverzeichnis

Literaturverzeichnis

[App98] A. W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[ARM99] ARM. *Cost-effective Per-Use Access to ARM Technology*. Foundry Program Flyer, www.arm.com, 1999.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bäh02] H. Bähring. *Mikrorechner-Technik*. Springer Verlag, 2002.

[Bin99] J. Binder. *Embedded DRAM*. Elektronik Industrie, 1999.

[BSL+01] R. Banakar, S. Steinke, B.S. Lee, M. Balakrishnan, and P. Marwedel. *Comparison of Cache- and Scratchpad based Memory Systems with Respect to Performance, Area and Energy Consumption*. Technical Report, IIT Delhi, Universität Dortmund, 2001.

[BSL+02] R. Banakar, S. Steinke, B.S. Lee, M. Balakrishnan, and P. Marwedel. *Scratchpad Memory - A Design Alternative for Cache Onchip Memory in Embedded Systems*. CODES2002, Colorado (U.S.A), 2002.

[Cam04] A.J. Camber. *XIP-NOR based Systems: Architectural Choice for Cellular Subsystems*. Technical Paper, Intel Corporation, 2004.

[CL03] N. Chang and H. Lee. *Energy-Aware Memory Allocation in Heterogeneous Non-Volatile Memory Systems*. ISPLED03, Seoul (Korea), 2003.

[Deb04] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley and Sons, 2004.

[Der03] P. Deriot. *Comparing XIP and Code Shadowing Architectures for 2.5G Cellular Phones*. Wireless Market Newsletter Q03/04, Micron Technology, 2003.

[Gru02] N. Grunwald. *Energieminimierung eingebetteter Systeme durch die dynamische Nutzung eines Scratchpad-Speichers*. Diplomarbeit, Universität Dortmund, Lehrstuhl XII, 2002.

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

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Archtitecture - A Quantitative Approach*. Morgan Kaufmann, 1996.

[Ilo] Ilog. *CPLEX-Optimizer*. www.ilog.com.

[IY00] T. Ishihara and H. Yasuura. *A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors*. DATE2000, Paris (Frankreich), 2000.

[Kan01] M. Kandemir. *Dynamic Management of Scratchpad Memory Space*. DAC2001, Las Vegas (U.S.A.), 2001.

[Leu01] R. Leupers. *Compiler für eingebettete Systeme*. Spezialvorlesung, Universität Dortmund, 2001.

[LKC02] H. Lee, K. Kim, and N. Chang. *Energy Exploration and Reduction of SDRAM Memory Systems*. DAC02, New Orleans (U.S.A.), 2002.

[LPL+04] J. Lee, C. Park, J. Lim, K. Kwon, and M. Sang. *Compiler Assisted Demand Paging for Embedded Systems with Flash Memory*. EMSOFT04, Pisa (Italien), 2004.

[Mar03] P. Marwedel. *Rechnerarchitektur*. Stammvorlesung, Universität Dortmund, 2003.

[Mic00] Micron. *SDRAM Write to Active Command Timing*. Technical Note 48-05, Micron Technology, 2000.

[Mic01] Micron. *Calculating Memory System Power for DDR*. Technical Note 46-03, Micron Technology, 2001.

[Mic02a] Micron. *Burst Mode Offers Significant Advantages*. Wireless Market Newsletter Q01/02, www.micron.com/flashcalc, 2002.

[Mic02b] Micron. *Mobile SDRAM Power Saving Features*. Technical Note 48-10, Micron Technology, 2002.

[Mic04a] Micron. *128Mbx16x32Mobile.pdf*. Datasheet 03/04, www.micron.com, 2004.

[Mic04b] Micron. *MT28F640J3.pdf*. Datasheet 06/04, www.micron.com, 2004.

[Mic04c] Micron. *Y25L_64MB.pdf*. Datasheet 03/04, www.micron.com, 2004.

[ML01] P. Marwedel and R. Leupers. *Retargetable Compiler Technology for Embedded Systems*. Kluwer Academic Publishers, 2001.

[Par04] C. Park. *Energy-Aware Demand Paging on NAND Flash-based Embedded Storages*. ISPLED04, Newport Beach (U.S.A.), 2004.

[PDN99] P.R. Panda, N.D. Dutt, and A. Nicolan. *Memory Issues in Embedded Systems-on-Chip*. Kluwer Academic Publishers, 1999.

[Raw04] F. Rawson. *MemPower - A Simple Memory Power Analysis Tool Set*. IBM Austin Research Labroratory, 2004.

[Sam02] Samsung. *DPD - The strongest Method to reduce SDRAM Power Consumption*. Application Note, Samsung Electronics, 2002.

[Sch04] U. Schwiegelsohn. *Halbleiterspeicher*. Technische Informatik, Universität Dortmund, 2004.

[Seg01] S. Segars. *Low Power Design Techniques for Microprocessors*. Präsentation, ISSCC2001, San Francisco (U.S.A.), 2001.

[SGW+02] S. Steinke, N. Grunwald, L. Wehmeyer, R. Balakrishnan, M. Banakar, and P. Marwedel. *Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory*. ISSS, Kyoto (Japan), 2002.

[SKWM01] S. Steinke, M. Knauer, L. Wehmeyer, and P Marwedel. *An accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations*. PATMOS01, Yverdon (Schweiz), 2001.

[SLWM02] S. Steinke, B.S. Lee, L. Wehmeyer, and P. Marwedel. *Assigning Programm and Data Objects to Scratchpad for Energy Reduction*. DATE2002, Paris (Frankreich), 2002.

[The00] M. Theokaridis. *Energiemessung von ARM7TDMI Prozessor-Instruktionen*. Diplomarbeit, Universität Dortmund, Lehrstuhl XII, 2000.

[Tiw96] V. Tiwari. *Logic and System Design for Low Power Consumption*. Dissertation, University of Princeton, 1996.

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. *Power analysis of embedded software: A first step towards software power minimization*. IEEE, Transactions on VLSI Systems, 1994.

[TY96] H. Tomiyama and H. Yasuura. *Optimal Code Placement Of Eembedded Software for Instruction Caches*. 9th European Design and Test Conference, Paris (Frankreich), 1996.

[VWM04] M. Verma, L. Wehmeyer, and P. Marwedel. *Dynamic Overlay of Scratchpad Memory for Energy Minimization*. CODES+ISSS, Stockholm (Schweden), 2004.

[Weg01] I. Wegner. *Lineare Optimierung und primal-duale Approximationsalgorithmen*. Effiziente Algorithmen, Universität Dortmund, 2001.

[Zob01] C. Zobiegala. *Energieeinsparung durch compilergesteuterte Nutzung des On-Chip-Speichers*. Diplomarbeit, Universität Dortmund, Lehrstuhl XII, 2001.