Kapitel 1

Einleitung

Tagtäglich überfluten uns Informationen über neu entwickelte, schnellere Computer, PDAs, MP3-Player und Handys mit zusätzlichen neuen Funktionen. In einer technologieorientierten Gesellschaft erscheint ein Leben ohne diese Geräte kaum mehr vorstellbar. Man kann sogar behaupten, dass unser Leben von diesen elektronischen Helfern abhängig geworden ist. Das gemeinsame Grundprinzip all dieser Geräte besteht in elektrischen Steuerungen, die es uns ermöglichen, das Leben zeitsparender, effektiver und bequemer zu gestalten. Die Entwicklung geht hin zu immer raffinierteren, komplexeren und leistungsfähigeren Systemen und ein Ende ist kaum vorhersehbar.

In den letzten Jahrzehnten haben wir eine rapide Entwicklung auch in dem Bereich der eingebetteten Systeme (EIS) erlebt. EIS werden häufig in mobilen Systembereichen eingesetzt. Da durch einen harten Konkurrenzkampf und stetig wachsende Ansprüche der Konsumenten mehr Funktionalitäten und mehr Performance für die mobilen Geräte gefordert sind, werden sie mit leistungsstärkeren Prozessoren sowie größeren und schnelleren Speicherbausteinen ausgestattet. Außerdem sollen die mobilen Geräte über eine längere Betriebsdauer als derzeitig verfügen. Auch diese Tendenz wird sich weiter fortsetzen.

Die Forderungen nach höherer Geschwindigkeit, mehr Funktionalitäten und längerer Betriebsdauer führen in der Regel zu einem höheren Energiebedarf.

Bei mobilen Geräten, die üblicherweise mit einer autarken Energiequelle arbeiten, besteht Entwicklungsbedarf an leistungsstärkeren Energieträgern. Bei dem gleichzeitigen Trend zu kleineren und leichteren Geräten sollten diese mit gleichem oder geringerem Platzbedarf und mit einem geringeren Gewicht entwickelt werden. Da die Batterieentwicklung viel langsamer als die Entwicklung der Hardware verläuft, sind wir gezwungen, weitere Lösungen zu finden. Es ist daher zu überlegen, welche Maßnahmen zur Reduzierung des Energieverbrauches in den Bereichen Hardware oder Software zu ergreifen sind.

Bei der Hardwareentwicklung gewinnen Strategien zur Reduktion des

Energieverbrauches mehr und mehr an Bedeutung. Allein um die Wärmeentwicklung zu begrenzen, wird moderne Hardware unter Berücksichtigung der Reduktion des Energieverbrauches hergestellt [S300] [AMD2].

Auch im Bereich der Software existieren mehrere Lösungsansätze. So sollten Programme energieoptimiert implementiert werden. Beispielsweise können energieaufwändige durch energiesparende Befehle (z.B. Multiplikation durch Addition) ersetzt werden.

Eine weitere Möglichkeit zur Reduktion des Energieverbrauches besteht in der softwaregesteuerten Reduzierung der Taktfrequenz und der Versorgungsspannung [HPS98] [OIY99]. Vor allem durch die Reduzierung der Versorgungsspannung kann eine sehr hohe Energieeinsparung erzielt werden, weil die Versorgungsspannung quadratisch in die Berechnung des Energieverbrauchs eingeht. So verringert sich der Energieverbrauch auf ein Viertel, wenn die Versorgungsspannung auf die Hälfte reduziert wird. Nach der Arbeit von Ishihara et al. [OIY99] kann der Energieverbrauch eines Prozessors durch die geregelte Reduzierung der Versorgungsspannung um bis zu 58 % reduziert werden. Allerdings muss die Taktfrequenz verringert werden.

1.1 Motivation

Es existieren viele Lösungsansätze für das Problem der Reduktion des Energieverbrauches bei eingebetteten Systemen. Ein weiterer, in der bisherigen Aufzählung noch nicht erwähnter Lösungsansatz, ist der Einsatz eines On-Chip-Speichers. Ein On-Chip-Speicher wurde bisher dafür eingesetzt, um den Gesamtprogrammablauf zu beschleunigen. Allerdings bringt ein On-Chip-Speicher nicht nur einen Geschwindigkeitsvorteil, sondern aufgrund seines niedrigen Energieverbrauchs pro Zugriff auch eine Reduzierung des Gesamtenergieverbrauches mit sich [Theo00]. Im Rahmen dieser Diplomarbeit wurden 2 On-Chip-Arten, ein Cache- und ein Scratch-Pad-Speicher, betrachtet. Es gibt auch einen Off-Chip-Cache-Speicher, allerdings beschränkt sich diese Arbeit auf die On-Chip-Variante.

Ein Cache-Speicher besitzt eine Hardwaresteuerung und ein Scratch-Pad wird durch Software gesteuert. Es ist interessant zu untersuchen, wie sich diese unterschiedlichen Steuerungsprinzipien auf den Energieverbrauch auswirken.

1.2 Ziel dieser Arbeit

Eines der Ziele dieser Diplomarbeit besteht in der Untersuchung des Energieverbrauchs eines ARM7-Prozessorsystems mit den verschiedenen Cacheorganisationen. Die daraus entstandenen Ergebnisse werden anschließend mit dem

Energieverbrauch des ARM7-Prozessorsystems mit einem Scratch-Pad-Speicher verglichen. Um die Auswirkung des Einsatzes der beiden On-Chip-Speicher in Bezug auf den Energieverbrauch näher zu veranschaulichen, werden die beiden Speicher mit verschiedenen Speichergrößen simuliert. Außerdem werden weitere Aspekte wie die Performance, das Zugriffsverhalten auf On- und Off-Chip-Speicher untersucht. Schließlich werden die Vor- und Nachteile des Einsatzes der Cache- und Scratch-Pad-Speicher anhand der untersuchten Aspekte aufgezeigt.

1.3 Kapitelübersicht

Im Kapitel 2 werden die Grundlagen eines Speichersystems vorgestellt. Dabei werden die allgemeine Speicherhierarchie, nähere Information über Scratch-Pad- und Cache-Speicher und die Algorithmen der beiden On-Chip-Speicher erläutert.

Das Kapitel 3 beschreibt die physikalischen Grundlagen und die Energiebetrachtung der beiden On-Chip-Speicher. Vor allem werden die Energiemodelle des Cache- und Scratch-Pad-Speichers vorgestellt.

Das Kapitel 4 beinhaltet die Beschreibung über den Simulationsablauf, die betrachtete Hardware, das Simulationstool, den Profiler „Traceanalyzer" und die Erweiterung des Traceanalyzers für den Cache-Speicher.

In dem Kapitel 5 werden die Simulationsergebnisse, ihre Analyse und Bewertung vorgestellt.

Das Kapitel 6 beschreibt die Zusammenfassung der Ergebnisbewertung und Verbesserungsvorschläge.

Anschließend werden in dem Anhang die vollständigen Simulationsergebnisse und der ARM Thumb-Befehlssatz dokumentiert.

Kapitel 2

Grundlagen von Scratch-Pad und Cache

Die beiden On-Chip-Speicher, die im Rahmen dieser Diplomarbeit untersucht wurden, speichern Programmteile und Daten ab, und stellen diese dem Prozessor zur Verfügung. Da der Energieverbrauch pro Zugriff auf einen On-Chip-Speicher viel geringer als der auf einen Off-Chip-Speicher ist, kann die Energiereduktion durch den Einsatz eines On-Chip-Speichers erreicht werden.

Ein On-Chip-Speicher ist ein Teil des ARM7-Prozessorsystems, das im Rahmen dieser Diplomarbeit simuliert und dessen Energieverbrauch berechnet wurde. In diesem Kapitel wird beschrieben, wie und warum die Speicher hierarchisch aufgebaut sind und wie On-Chip-Speicher, z.B. Scratch-Pad, und Cache funktionieren.

Ein idealer Rechner besitzt einen Speicher mit folgenden Eigenschaften:

- beliebig große Kapazität.
- minimale Zugriffszeit
- maximale Zugriffsrate
- minimale Kosten
- Nichtflüchtigkeit

Da es keinen Speicher gibt, der all diese Anforderungen erfüllt, wird versucht, eine Annäherung zu suchen. Mit einem hierarchisch aufgebauten Speichersystem hat man einen Kompromiss zwischen Geschwindigkeit und Kosten gefunden.

2.1 Speicherhierarchie

In der Realität treten einige Tradeoffs auf, z.B. zwischen Speichergröße und Zugriffszeit und zwischen Geschwindigkeit und Kosten. Wenn die Speichergröße zunimmt, nimmt in der Regel auch die Zugriffszeit zu. Dies führt zu längerer Ausführungszeit, was im Allgemeinen zu keiner funktionalen Einschränkung, aber zu geringer Performance führt. Im umgekehrten Fall tritt aber eine Beschränkung der lauffähigen Programme auf, da besonderes große Programme nicht ausgeführt werden können.

Ein anderer Tradeoff besteht in der Geschwindigkeits- und Kostenfrage. Man kann sich zum Beispiel fragen, warum man kein Speichersystem nur mit ausreichend vielen Registern baut. Es wäre ein sehr schnelles und energiemäßig günstiges System. Da die schnelleren Speicher jedoch wesentlich teurer sind und zum Teil größeren Platzbedarf haben, wie es bei SRAM der Fall ist (Faktor ≥4) [MO01], kann ein Speichersystem leider nicht nur aus schnellen Speicherbausteinen z.B. SRAM bestehen. Aus diesen Gründen wird man gezwungen, vorhandene Ressourcen möglichst effizient zu nutzen.

Das Resultat ist die Ausnutzung von Lokalitätseigenschaften [TG01]:

- Zeitliche Lokalität: wenn das Datum oder die Instruktion gerade benutzt wurde, kommt es häufig vor, dass ein Datum oder eine Instruktion bald wieder benutzt wird.

- Örtliche Lokalität: Es kommt auch häufig vor, dass, wenn auf ein Datum oder eine Instruktion zugegriffen wurde, bald auf ein benachbartes Datum oder eine benachbarte Instruktion zugegriffen wird. Zum Beispiel werden die Instruktionen, abgesehen von Adresssprüngen, nacheinander ausgeführt.

Dank dieser Eigenschaften wird mit günstigen Kosten, d.h. ohne überdimensionale teure und schnelle Speicher, die erforderliche Performance erreicht. Es handelt sich hier um mehrere Speicherstufen mit unterschiedlichen Speichergrößen und Geschwindigkeiten.

Die Speicherhierarchie versucht die Annäherung an den idealen Speicher durch reale Speichermedien und Nutzung von Lokalitätseigenschaften zu erreichen. Kleinere, schnellere und teurere Cache-Speicher werden benutzt, um Daten zwischenzu-speichern, die sich in größeren, langsameren und billigeren Speichern befinden.

Es ist interessant zu beobachten, wie die großen Datenmengen von einem langsamen Speicher auf einen kleinen schnellen Speicher gelangen. In der fünfziger und sechziger Jahren haben sich Programmierer viel Mühe geben müssen, wenn sie ein Programm in den kleinen Speicher quetschen wollten. Weil die damalige Speichergröße recht gering war, haben sie sogar zum Teil schlechtere Algorithmen verwenden müssen, nur aus dem Grund, weil diese kleiner waren.

In der *Abbildung 2.1* ist eine typische Speicherhierarchie ersichtlich.

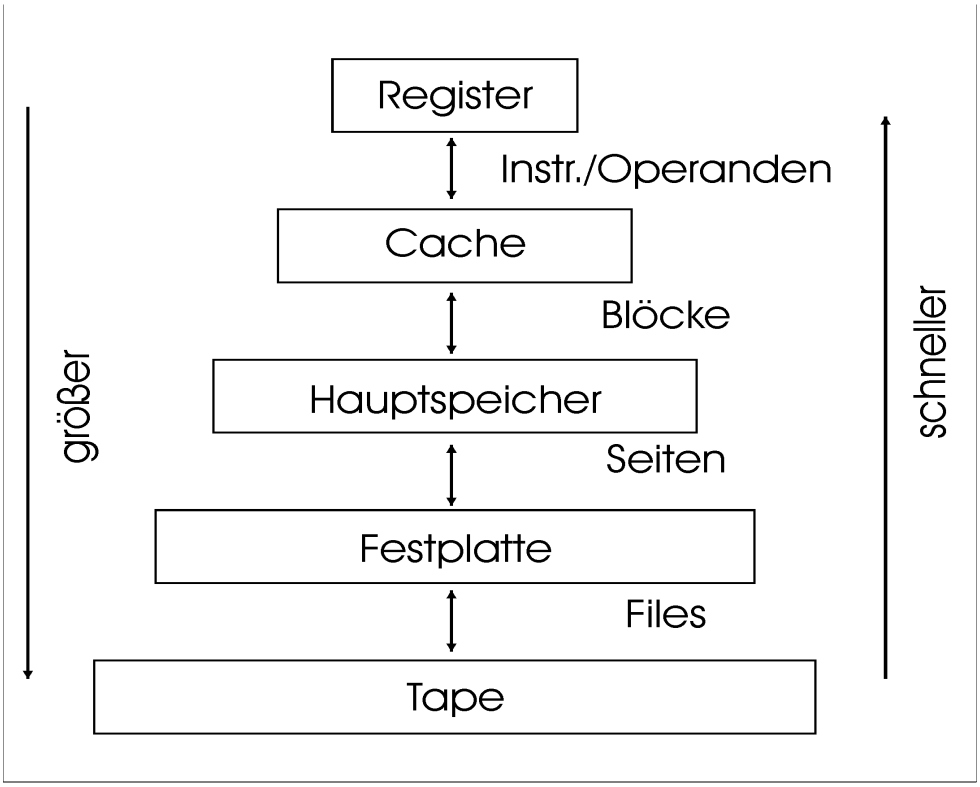


Abbildung 2.1 *Speicherhierarchie*

Wenn das Programm zu groß für den Speicher ist, wird typischerweise ein zweiter Speicher, z.B. eine Festplatte, benutzt. Der Programmierer muss in diesem Fall sein Programm in Stücke zerlegen, damit es in den Speicher passen kann. Diese sogenannten Überlagerungen wurden nacheinander ausgeführt. Allerdings mussten die Programmierer dafür sorgen, dass das Programm richtig zerlegt wurde, wo und wie diese Überlagerungen gespeichert und transportiert werden sollten.

Im Jahr 1961 haben britische Forscher in Manchester diesen Überlagerungsprozess automatisiert [TG01]. Diese Methode ist heute als virtueller Speicher bekannt. Außer den oben beschriebenen Gründen gibt es noch zwei weitere Gründe, die die Benutzung des virtuellen Speichers notwendig machen. Ein Grund ist, dass das Laden eines ganzen Programms in den Hauptspeicher zu erheblichen Startzeiten führen kann. Andererseits werden zum Startzeitpunkt nicht alle Dateien und Programmteile benötigt, wie z.B. bei Editoren. Der andere Grund ist die gleichzeitige Ausführung mehrerer Prozesse. Einige Betriebssysteme erlauben die parallele Ausführung mehrerer Prozesse, die jeweils einen eigenen Adressraum haben.

Bei der Verwendung des virtuellen Speichers müssen virtuelle Adressen eines Prozesses in physikalische Adressen des Hauptspeichers umgewandelt werden Das Konzept, den Adressraum und die Speicheradresse zu trennen, funktioniert folgendermaßen:

Wir betrachten einen Rechner mit beispielsweise 4 Kbyte Hauptspeicher. Zu einem beliebigen Zeitpunkt können 4096 Speicherwörter (hier 1 Wort = 1 Byte) direkt adressiert werden. Das bedeutet allerdings nicht, dass nur die virtuellen Speicheradressen 0 bis 4095 adressiert werden können. Die Adressen, z.B. 4096 bis 8191, werden auf die Adressen 0 bis 4095 im Hauptspeicher abgebildet.

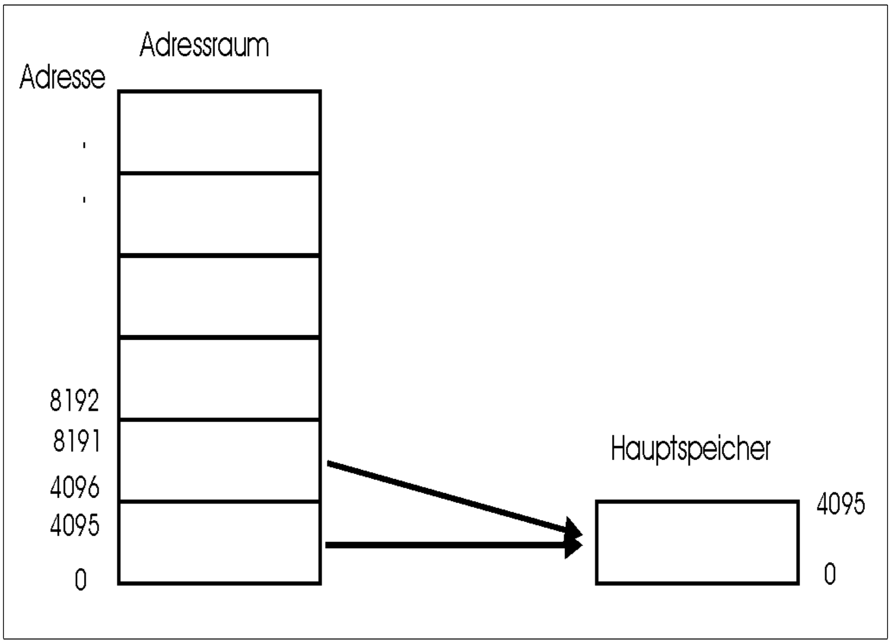


Abbildung 2.2 *Abbildung der virtuellen Adressen auf die Adressen 0 bis 4095 im Hauptspeicher [TG01]*

Ein Rechner mit virtuellem Speicher würde wie folgt vorgehen:

1. Der Speicherinhalt (0 bis 4095) wird auf die Festplatte gespeichert.
2. Die gesuchten Wörter, z.B. 4096 bis 8191, werden von der Festplatte in den Hauptspeicher geladen.
3. Die Ausführung wird fortgesetzt, als ob alle Speicherinhalte von vornherein im Arbeitsspeicher enthalten gewesen wären.

Diesen Vorgang nennt man *Paging* und die in fester Größe eingeteilten virtuellen Speicherbereiche *Seiten (Pages)*.
Für die Programmierer bedeutet diese Technik eine Sicht, als wenn sie einen Rechner mit kontinuierlichem linearen Speicher in gleicher Größe wie den des virtuellen Adressraumes hätten.

Ein Sekundärspeicher, z.B. eine Festplatte, ist eine wichtige Voraussetzung für einen virtuellen Speicher. Da sich alle Programme und Daten auf der Platte befinden und ausgeführt werden können, kann man sich vorstellen, dass das Programm auf der Platte als Original und die in den Arbeitsspeicher überführten Teile als Kopie anzusehen sind. Wenn die Daten im Arbeitsspeicher geändert werden, sind die Originale entsprechend zu aktualisieren.

Der virtuelle Adressraum wird in eine Reihe von festen Größen (Seiten) aufgeteilt. Die Größen reichen heute von 512 Byte bis 64 Kbyte pro Seite, gelegentlich werden auch Größen von 4 Mbyte benutzt [TG01]. Die Seitengröße ist typischerweise eine 2er Potenzzahl. Der physikalische Adressraum wird auch in die gleiche Größe wie die Seitengröße aufgeteilt, so dass jeder Teil des Arbeitsspeichers genau eine Seite aufnehmen kann. Der Teil dieses Arbeitsspeichers wird als Seitenrahmen (*page frame*) bezeichnet.

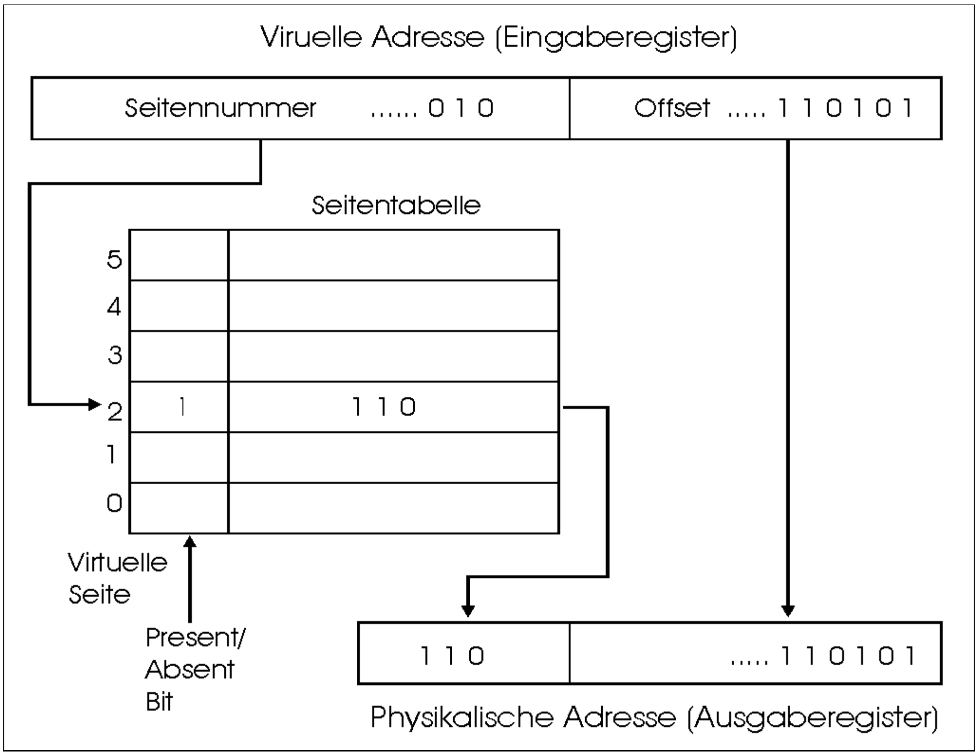


Abbildung 2.3 Bildung einer physikalischen Adresse aus einer virtuellen Adresse

Die Übersetzung einer virtuellen Adresse in eine physikalische lässt sich für das *Paging* wie in *Abbildung 2.3* darstellen. Da ein Speicher nur Arbeitsspeicheradressen kennt, also keine virtuellen Adressen, muss sie genau an ihn übermittelt werden. Jeder Computer mit einem virtuellen Speicher hat eine Funktionseinheit zur Durchführung der Abbildung von virtuellen Adressen in physikalische Adressen. Diese Funktionseinheit nennt man MMU (Memory Management Unit).

Die virtuelle Adresse besteht aus der Seitennummer und dem Offset innerhalb der Seite. Die virtuelle Seitennummer wird als Index zu einer Seitentabelle benutzt, die angibt, welche Seitenrahmen durch welche Seiten belegt sind.

Die MMU prüft als erstes den Eintrag der Seitentabelle, um zu sehen, ob sich die referenzierte Seite im Hauptspeicher befindet. Normalerweise können nicht alle virtuelle Seiten in den Hauptspeicher geladen werden, da der Hauptspeicher in der Regel kleiner als der virtuelle Speicher ist. Die MMU führt diese Überprüfung durch, indem sie das Valid-Bit (Present/Absent-Bit) im betroffenen Eintrag in der Seitentabelle prüft.

Wenn die Seite im Hauptspeicher vorhanden ist, wird der Wert des Seitenrahmens (Indexwert des Seitenrahmens) aus dem gewählten Eintrag wie in der *Abbildung 2.3* in den oberen Teil des Ausgaberegisters und der Offsetwert in den unteren Teil des Ausgaberegisters kopiert. Die physikalische Adresse besteht aus diesen beiden Teilen.

Wenn die Seite nicht im Hauptspeicher liegt, nennt man das Seitenfehler. Nach Eintritt eines Seitenfehlers muss die erforderliche Seite von der Festplatte gelesen werden. Dazu wird zuerst überprüft, ob ein Seitenrahmen im Hauptspeicher leer ist. Wenn kein Seitenrahmen leer ist, wird eine Seite aus dem Hauptspeicher verdrängt und die Seitentabelle aktualisiert. Dann wird die geforderte und von der Festplatte gelesene Seite in den Hauptspeicher geschrieben. Diese Vorgehensweise wird *als Demand-Paging* bezeichnet. Beim *Demand-Paging* werden Seiten nur überführt, wenn sie tatsächlich angefordert werden, nicht im voraus.

Eine andere Methode basiert auf dem sogenannten *Working Set.* Die Idee stammt aus der Beobachtung, dass auf den Adressraum von den meisten Programmen nicht gleichmäßig referenziert wird. Die Seiten, die zu einem Zeitpunkt *t* von den letzten *k* Speicherreferenzen benutzt wurden, werden *Working Set* genannt [TG01]. Da sich diese Seitenmenge im Verlauf der Zeit meistens nur langsam ändert, können diese Seiten vor dem Programmstart geladen werden.

Beim Seitenfehler stellt sich die Frage, welche Seite aus dem Hauptspeicher verdrängt werden soll, um die angeforderte Seite einzuladen. Dies ist eine sehr wichtige Frage, weil man möglichst viele Seiten, die häufig benutzt werden, im Hauptspeicher halten will. Um für die angeforderte Seite Platz zu schaffen, muss aber irgendeine andere Seite aus dem Hauptspeicher gelöscht werden. Folglich braucht man eine effiziente Methode, die die zu entfernende Seite effektiv aussucht. Eine Seite zufällig auszuwählen, wäre einfach zu implementieren, aber es wäre keine gute Idee. Dann wäre die Gefahr groß, dass zufällig die Seite ausgewählt würde, die das Programm vorwiegend nutzte.

Die häufig benutzten Ersetzungsstrategien sind Folgende:

- LRU (Least Recently Used):
  Eine der beliebtesten Methoden. Bei LRU wird die Seite aus dem Hauptspeicher verdrängt, auf die am längsten nicht mehr zugegriffen wurde.

- LFU (Least Frequently Used):
  Die Seite wird ersetzt, auf die seit ihrer Einlagerungszeit am seltensten zugegriffen wurde.

- FIFO (First In First Out):
  Die Seite wird ersetzt, die jeweils zuerst geladen wurde, unabhängig davon, wann diese Seite zuletzt angefordert wurde.

Diese Ersetzungsstrategien werden auch bei einem Cachespeicher benutzt, weil ein Cache-Controller entscheidet, welcher der Cache-Inhalte ersetzt werden soll. Bisher wurde die allgemeine Speicherarchitektur beschrieben. In den nächsten Unterkapiteln werden nähere Informationen über die beiden On-Chip-Speicher, Cache- und Scratch-Pad-Speicher, vorgestellt.

2.2 Scratch-Pad

Die meisten eingebetteten Systeme haben so genannte *system-on-chip (SoC)-Architekturen*, weil sie meistens in kleiner Größe gebaut werden sollen. Was bei den *SOC-Architekturen* besonders berücksichtigt werden soll, sind vor allem die richtige Speicherkonfiguration und das Datenmanagement. Cache und Scratch-Pad werden als On-Chip-Speicher in den eingebetteten Systemen eingesetzt und sind schnell und energiemäßig günstig.

Ein Cache funktioniert unabhängig von Anwendungen selbstständig und wird hardwaremäßig gesteuert. Ein Scratch-Pad funktioniert dagegen nicht selbstständig im Sinne der hardwaremäßigen Steuerung, sondern Programmierer oder Compiler übernehmen die Steuerungsaufgabe.

2.2.1 Funktionsweise eines Scratch-Pads

Ein Scratch-Pad ist ein frei adressierbarer Speicher und wird als On-Chip-Speicher zum Speichern von Teilen von Daten und Programmen eingesetzt. Die Applikation muss vorher analysiert werden, um die häufig benutzten Daten oder Programmteile in den On-Chip-Speicher laden zu können. Der Programmierer oder der Compiler muss dafür sorgen, dass die teuren Off-Chip-Zugriffe möglichst gering bleiben.

Ein Scratch-Pad befindet sich zwischen der CPU und dem Hauptspeicher und benutzt den selben Adress-Bus und Daten-Bus. Die im Scratch-Pad eingelagerten Daten stehen im statischen Fall kontinuierlich der CPU zur Verfügung und können im dynamischen Fall ersetzt werden. Bei einem statischen Scratch-Pad-Algorithmus werden die Daten und die Programmteile, die in den Scratch-Pad-Speicher verschoben wurden, nicht mehr ersetzt. Bei einem dynamischen Scratch-Pad-Algorithmus können diese Programmteile und Daten durch neue Programmteile bzw.

Daten ersetzt werden, wenn z.B. auf sie nicht mehr zugegriffen wird. In dieser Arbeit wird nur der statische Fall behandelt, und davon ausgegangen, dass der Gesamtspeicherbedarf des Programms größer als die Größe des Scratch-Pads ist.

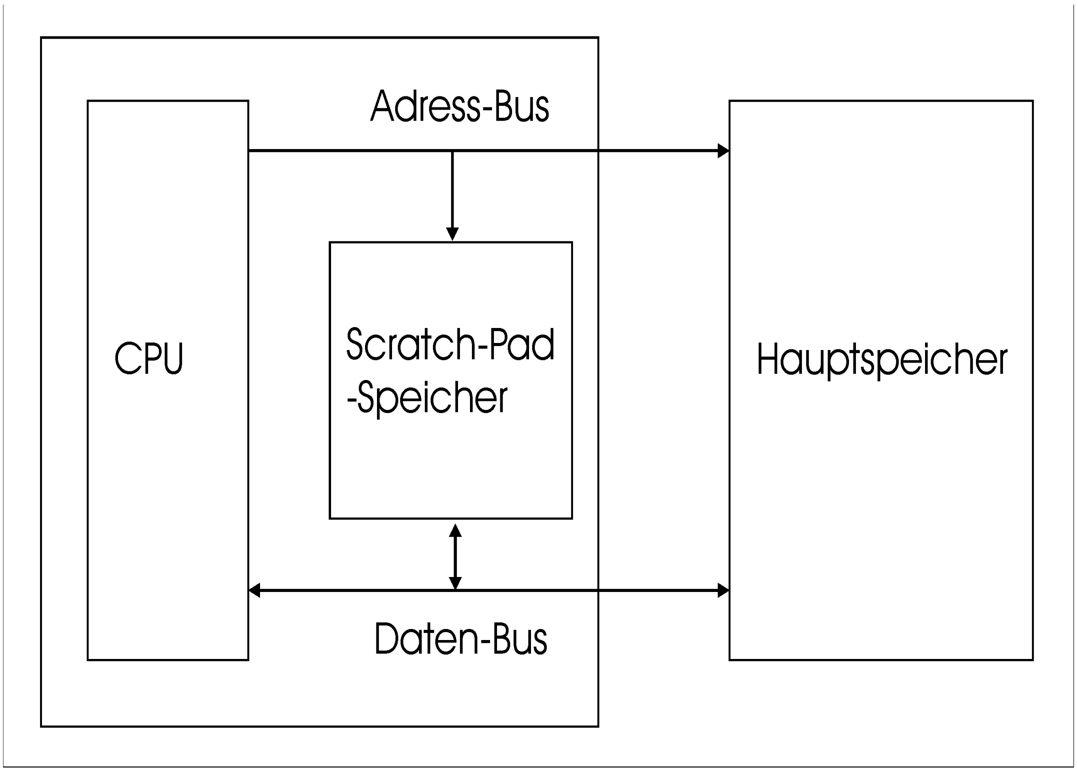


Abbildung 2.4 Aufbau eines Scratch-Pads

Um guten Code für ein Speichersystem mit Scratch-Pad zu erhalten, muss dafür gesorgt werden, dass der Compiler außer den üblichen Optimierungszielen, z.B. Optimierung für die Codegröße oder für die Geschwindigkeit, explizit den Datentransfer zwischen Off-Chip-Speicher und Scratch-Pad scheduled.

Bei der Vorbereitung für die Anwendung von Scratch-Pads müssen vor allem folgende Faktoren berücksichtigt werden:

1. das Datenlayout in dem Off-Chip-Speicher,
2. das Zugriffsverhalten der Applikation,
3. die Größe des Scratch-Pads.

Der CPU ist es transparent, wo die Daten sich befinden. Da auf den Hauptspeicher durch den Einsatz von Scratch-Pad-Speichern weniger zugegriffen wird, entstehen weniger Wartezyklen (Waitstates).

2.2.2 Algorithmen für Scratch-Pad

Die meisten Algorithmen für die Speicher-Allokation haben ähnliche Vorgehensweisen. Der für diese Diplomarbeit eingesetzte Algorithmus ist aus der Diplomarbeit von Zobiegala [Zob01] entnommen worden. Der Algorithmus verschiebt nicht nur die globalen Variablen sondern auch einen Teil des Programms in den Scratch-Pad-Speicher. Von den Instruktionen sind die Befehle, Load, Store, Push und Pop zu beachten, weil nur diese Befehle auf den Hauptspeicher zugreifen.

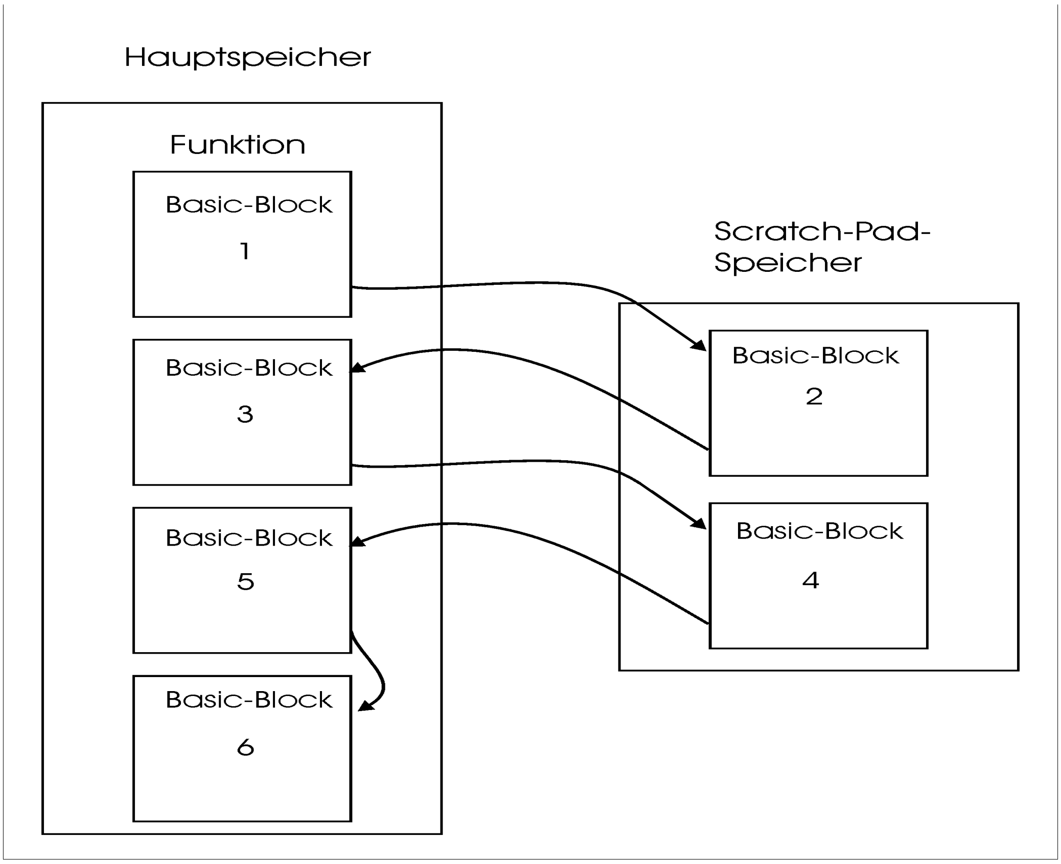


Abbildung 2.5 *Die Einlagerung der ausgewählten Basic-Blöcke in den Scratch-Pad-Speicher*

Der Algorithmus läuft in vier Schritten:

1. **Erkennen und Berechnen von globalen Variablen**

   Der Code wird nach den globalen Variablen durchsucht und die Anzahl der Zugriffe werden ermittelt. Wie bei Steinke et al. [St01] beschrieben, befinden sich im ASM-Code Load und Store-Anweisungen. Die mit den Anweisungen assoziierten Variablen werden erkannt und die Anzahl der Zugriffe berechnet. So ergibt sich aus dem Kostenunterschied (Off-Chip versus On-Chip) eine

Wertigkeit für die betreffende Variable usw. Dies geschieht mit allen Objekten. Mögliche globale Variablen sind sowohl skalare als auch nicht skalare Variablen.

2. **Erkennen und Berechnen von Funktionen und Basic-Blöcken**

   Das gleiche geschieht mit Funktionen und Basic-Blöcken. Der Code wird nach den Basic-Blöcken durchsucht und die Anzahl der Zugriffe ermittelt. Das ergibt eine Liste von „Memory Objekten“, die verschoben werden können. Wenn die ganze Funktion in den Scratch-Pad verschoben werden soll, sind die Berechnung für die benötige Speichergröße und der Aufwand für die Compilermodifizierung einfacher, da der Assembler-Code nicht geändert werden muss. Wird nur ein Teil einer Funktion mehrfach aufgerufen, sollte nur der Teil in den Scratch-Pad verschoben werden, weil einerseits die Funktion zu groß sein kann, und andererseits sie selbst nur einmal oder selten aufgerufen wird.

3. **Lösen des Knapsack-Problems**

   Das Problem ist ein Knapsack-Problem [Zob01], d.h. welche der Objekte mit unterschiedlichen Größen und Zugriffszahlen in den Scratch-Pad mit einer begrenzten Kapazität verschoben werden sollen. Daher wird die Liste mit den Memory-Objekten nach ihrer Wertigkeit sortiert. Nun wird ein Branch und Bound aufgespannt. Der Branch und Bound-Algorithmus basiert auf einem Binärbaum, wo der oberste Knoten alle Probleme darstellt. Der linke Zweig bedeutet, dass die Objekte nicht in den Scratch-Pad verschoben werden, und der rechte Zweig bedeutet, dass die Objekte in den Scratch-Pad verschoben werden. Da man bei der Entscheidung bestimmte Bedingungen zur Hilfe nehmen kann, beispielsweise werden die Knoten, deren Objektgröße größer als die Größe des Scratch-Pad-Speichers ist, nicht mehr berücksichtigt, werden einige Teilbäume nicht mehr betrachtet. Dies führt dazu, dass die Laufzeit nicht exponentiell verlaufen muss.

4. **Verteilen der berechneten Objekte in die verschiedenen Speicher**

   Nach der Berechnung der Wertigkeiten der Memory-Objekte, die in den On-Chip-Speicher passen, wird der ASM-Code, der von dem ARM-Compiler erzeugt wurde, auf zwei Dateien (**onchip.asm* und **offchip.asm*) verteilt. Dann geschieht die Zuweisung von festen Speicheradressen, was die Aufgabe des Linkers ist. (Auflösung von symbolischen Links und Zuweisung von festen Adressen)

Ein anderer aber ähnlicher Algorithmus für die Allokation von globalen Objekten ist von Sjödin [Sj98] beschrieben. Dieser Algorithmus läuft auch in vier Schritten:

1. „Point-to“ Analyse
   Durch die „Point-to“ Analyse wird ermittelt, welche Objekte durch „load“- und „store“-Anweisungen geladen bzw. gespeichert werden sollen.

2. Bildung des „Callgraph“
   Die Informationen aus der „Point-to“ Analyse werden gebraucht, um den „Callgraph“ zu bilden.

3. Static Profiling
   Hier wird die Anzahl der Ausführungen von Basic -Blöcken ermittelt.

4. Memory Allokation
   Um möglichst viele Zugriffe auf Scratch-Pad-Speicher zu erreichen, werden die Objekte, auf die am meisten zugegriffen wird, in den Scratch-Pad-Speicher eingelagert.

Das Problem bei der Allokation ist ebenfalls das „Knapsack"-Problem. Allerdings haben Sjödin et al. einen einfacheren Algorithmus von Cormen et al. („simple greedychoise approximation algorithm") [CLR90] implementiert. Dieser Algorithmus arbeitet in zwei Schritten.

1. Jedem globalen Objekt wird eine Priorität zugewiesen. Diese Priorität ist definiert wie folgt:

$$\frac{N_i}{S_i} \tag{2.2}$$

   N_i = Die Anzahl der Zugriffe, S_i = Die Größe des Objektes

2. Objekte werden nach den Prioritäten in den Scratch-Pad eingelagert, bis kein Objekt mehr hineinpasst.

2.3 Cache

Ein Cache befindet sich in der Regel zwischen der CPU und dem Hauptspeicher, und wird zur Beschleunigung der Programmausführung eingesetzt. Der Cacheeinsatz ist nicht nur wegen seiner schnellen Zugriffszeit interessant, sondern auch wegen seines sehr günstigen Energieverbrauchs vor allem in dem Systembereich, wo nur eine autarke Energieversorgung in Frage kommen kann.

Im Gegensatz zum Scratch-Pad besitzt ein Cache eine Kontroll-Logik, durch die der Datentransfer geregelt wird. Eine explizite Softwaresteuerung ist nicht nötig. Ein Cache lebt von den Lokalitätseigenschaften der Programme, und behält die Daten, auf die zuletzt zugegriffen wurde, in seinen Cachelines.

Im Rahmen dieser Diplomarbeit wurde der Energieverbrauch des Caches in Abhängigkeit von der Cachegröße und der Cacheorganisation untersucht. In diesem Kapitel werden die Arten, Eigenschaften, Organisationen und Algorithmen eines Caches vorgestellt.

2.3.1 Cachearten

Es gibt mehrere Unterscheidungskriterien für Cachearten. Das erste Kriterium ist, wieviele Caches man in seinem System hat und wo sich die Caches befinden. Nach diesem Kriterium können 3 Cachearten unterschieden werden [AMD1]:

- **First-Level-Cache:**
  Das ist der schnellste Speicher, der im Prozessor eingebaut ist. Dort werden Befehle und Daten zwischengespeichert. Im Rahmen dieser Diplomarbeit wird diese Art des Caches betrachtet. Die Bedeutung des On-Chip-Caches wächst mit der höheren Geschwindigkeit der CPU. Denn dieser Cache vermeidet entsprechende Verzögerungen in der Datenübermittlung (keine Wait States) und hilft, eine CPU optimal auszulasten.

- **Second-Level-Cache:**
  Das ist die bekannteste Form eines Caches, der in der Regel außerhalb des Prozessors liegt. In ihm werden die Daten des Arbeitsspeichers (RAM) zwischengespeichert.

- **Third-Level-Cache:**
  Diese Art von Caches verwendet z.B. der Chip-Hersteller „AMD“ bei seinem Prozessor K6-3. Er liegt außerhalb des Prozessors, und ist bis zu 2 MB groß. Allerdings liegen hier die anderen beiden L1- und L2-Caches im Prozessor.

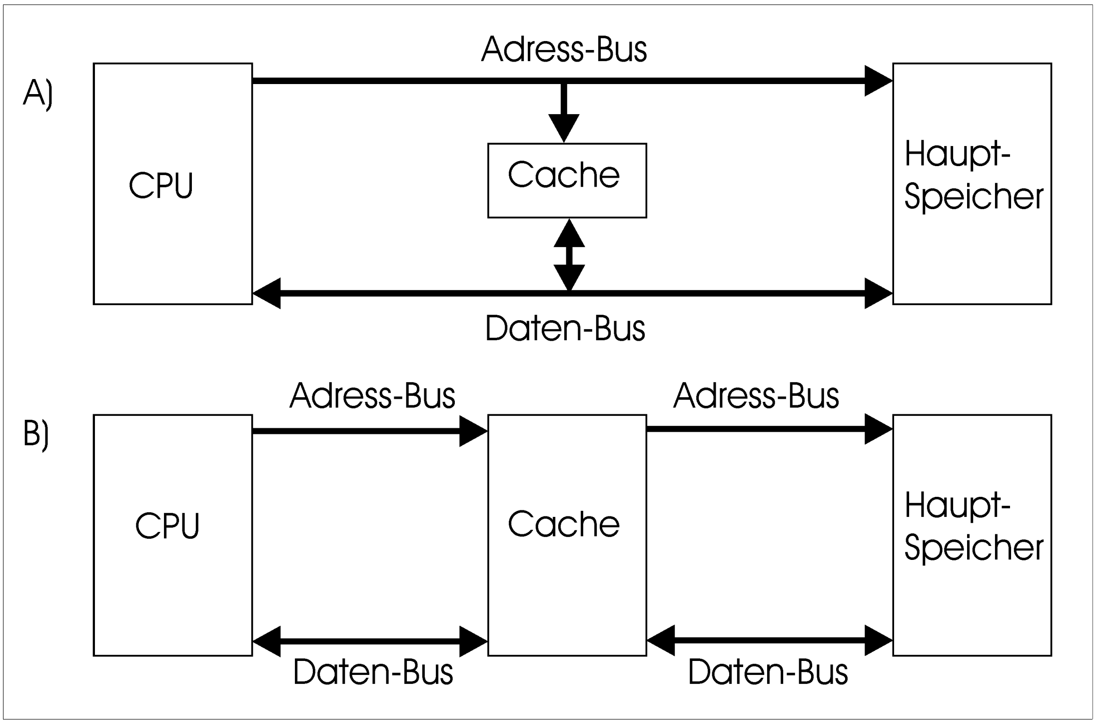


Abbildung 2.6 Lookaside-Cache (A) und hierarchischer Cache (B)

Ein anderes Kriterium ist die Trennung des Systembusses durch einen Cache [Lin01].

- **Lookaside-Cache:**
  Der Lookaside-Cache wird in der Regel bei einem Ein-Prozessor-System eingesetzt. Er wird parallel zum Hauptspeicher eingebaut. Das Datum wird im Cache gesucht, und parallel dazu wird auf den Hauptspeicher zugegriffen. Wenn das gesuchte Datum sich im Cache befindet, erzeugt der Cachecontroller ein Steuersignal. Dadurch kann die Suche im Hauptspeicher abgebrochen werden. Da der Hauptspeicher auch parallel zum Cache nach einem Datum durchsucht wird, ist die Zeit für „miss penalty" kürzer als die bei den anderen Cachearten.

- **Hierarchischer Cache:**
  Durch einen hierarchischen Cache wird der Prozessor vom Bussystem getrennt. Der Cachecontroller ist in diesem Fall zuständig für alle Datentransfers und die Kommunikation. Wenn sich das gesuchte Datum nicht im Cache befindet, wird erst dann die Suche im Hauptspeicher ausgeführt. Dieser Cache ist für Multiprozessorsysteme geeignet, da die meisten Aktionen nur zwischen dem Prozessor und Cache stattfinden.

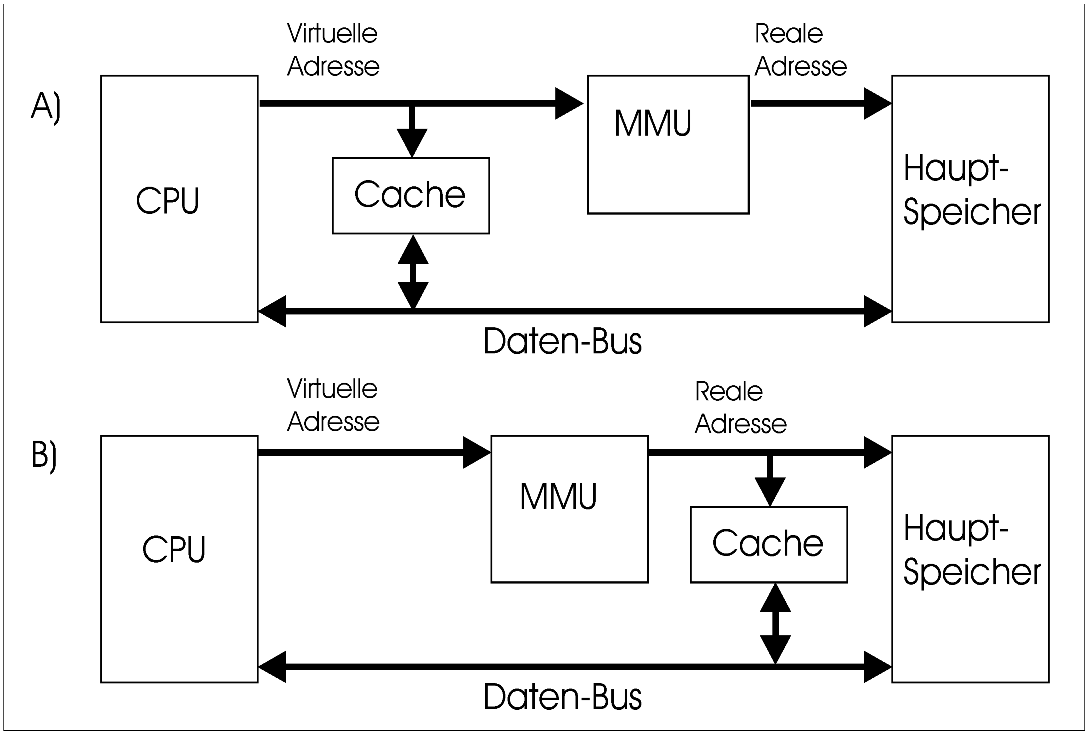


Abbildung 2.7 Virtueller Cache (A) und realer Cache (B) [Ma00]

Nach der Anordnung unterscheidet sich ein virtueller Cache von einem realen Cache [Ma00]:

- **Virtueller Cache:**
  Ein virtueller Cache befindet sich zwischen der CPU und der MMU und arbeitet mit virtuellen Adressen. Da der Zugriff auf die MMU und den Cache parallel läuft (Die beiden Zugriffe sind voneinander nicht kausal abhängig!), arbeitet ein virtueller Cache in der Regel schneller als ein Cache, der mit realen Adressen arbeitet (Realer Cache).

- **Realer Cache:**
  Ein realer Cache liegt zwischen MMU und Hauptspeicher und arbeitet daher mit den physikalischen Adressen. Im Gegensatz zu dem virtuellen Cache muss beim realen Cache die Adressumwandlung in der MMU zuerst ausgeführt werden. Dann wird anhand dieser realen Adresse das Datum im Cache bzw. im Hauptspeicher gesucht.

Das letzte Kriterium ist die Datenverwaltung im Cache. Die Frage ist, ob ein Cache die Daten und Instruktionen getrennt verwaltet. Hier unterscheidet sich der Unified-Cache von dem Split-Cache [TG01].

- **Unified-Cache:**
  Beim Unified-Cache werden Daten und Instruktionen in demselben Cache gehalten. Diese Art von Cache hat den Vorteil, dass er ein einfaches Design besitzt, und der Verwaltungsaufwand geringer als der beim Cache ist, der separate Speicher für Daten und Instruktionen besitzt. Dies hat allerdings den Nachteil, dass ein Datum und eine Instruktion nicht gleichzeitig vom Cache geholt werden können.

- **Split-Cache**
  Ein Split-Cache hat einen Instruction-Cache und einen Data-Cache, sodass die Daten und Instruktionen getrennt abgespeichert werden. Ein Split-Cache wird auch als „Havard-Architektur“ bezeichnet, weil der „Howard Aikens Mark-III-Computer unterschiedliche Speicher für Instruktionen und Daten hatte. Da die Pipeline-CPUs immer mehr verwendet werden, wo der Prozessor auf eine Instruktion und ein Datum gelegentlich gleichzeitig zugreift, wird diese Art des Caches immer stärker verwendet.

2.3.2 Aufbau eines Caches

Ein Cache besteht aus den folgenden 3 Teilen: Tag-Speicher, Daten-Speicher und Cache-Controller.

- **Tag-Speicher:**
  In dem Tag-Speicher werden Tags gespeichert. Ein Tag ist der vordere Teil der Adresse eines Datums. Das Tag gibt die Information an, in welcher Cacheline sich das Datum befindet.

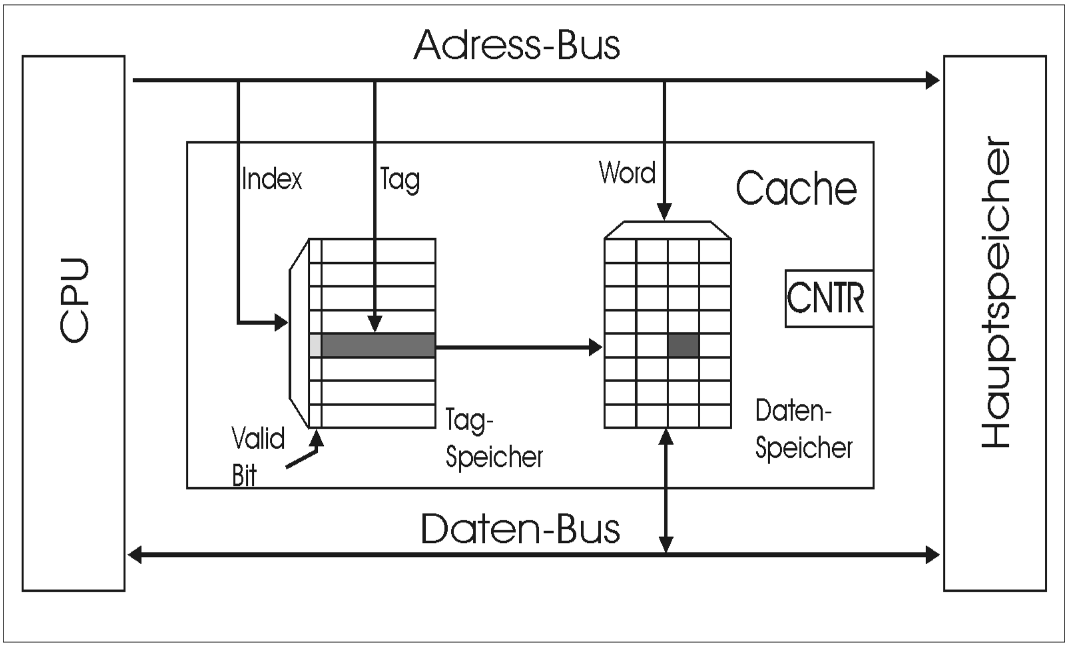


Abbildung 2.8 Der Aufbau eines Caches

- **Daten-Speicher:**
  Im Daten-Speicher wird eine Kopie der Daten oder Instruktionen aus dem Hauptspeicher gespeichert. Der Daten-Speicher wird in Blöcke mit fester Größe (Cachelines) aufgeteilt.

- **Cache-Controller:**
  Der Cache-Controller ist für die Kommunikation mit den anderen Systemkomponenten und für den Datentransfer zuständig. Er verwaltet auch die Cachelines und legt fest, nach welchen Strategien die Cacheeinträge ersetzt und aktualisiert werden sollen.

Der Hauptspeicher wird in Blöcke mit fester Größe namens Cacheblock oder Cacheline aufgeteilt. Ein Cacheblock besteht normalerweise aus 4 bis 64 aufeinanderfolgenden Byte. Die Blöcke sind ab 0 durchnumeriert. Die Abbildung 2.9 ist ein Beispiel für einen Direct-Mapped Cache. Dieser wird im Kapitel 2.3.3 näher erläutert. Der Cache ist 128 Byte groß und die Größe jeder Cacheline (Cacheblock) beträgt 16 Byte. Die Adresse 0 bis 15, 128 bis 143, 256 bis 271 usw. beanspruchen den gleichen Cacheblock 0 (in der *Abbildung 2.9 (B)*). Die Adressen auf derselben Zeile beanspruchen jeweils den gleichen Cacheblock, der auf der gleichen Zeile liegt. Die Daten der Adressen sind also die potenziellen Kandidaten für den jeweiligen Cacheblock, wenn sie von der CPU angefordert werden.

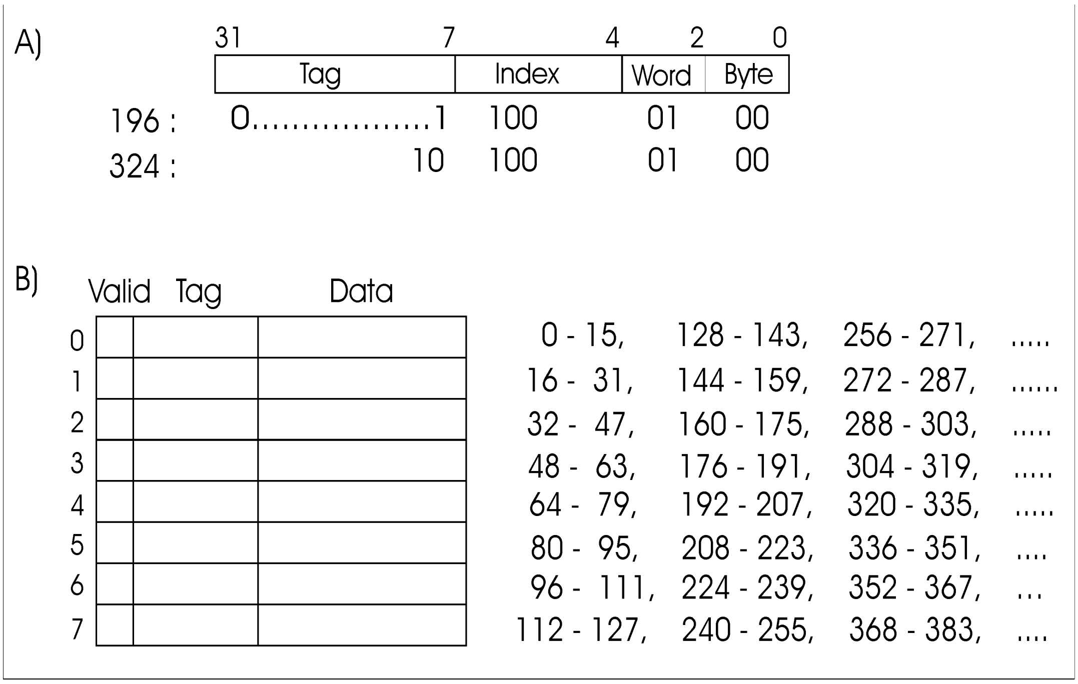


Abbildung 2.9 *eine virtuelle Adresse(A) und ein Beispiel für einen „Direct-Mapped Cache“(A)*

Ein Cache funktioniert folgendermaßen:

Wird auf den Speicher zugegriffen, prüft der Cachecontroller, ob das angeforderte Datum im Cache enthalten ist. Wenn das Datum im Cache ist, kann der Weg zum Hauptspeicher eingespart werden. Wenn nicht, das nennt man Cachemiss, wird das Datum vom Hauptspeicher geholt.

Produziert die CPU eine Adresse, wird die Adresse so aufgeteilt: Tag, Index, Word und Byte (*Abbildung 2.9 (A)*) [ARMcm98], [TG01].

- **Tag:**
  Durch das Vergleichen der Tag-Bits des Tag-Speichers mit den Bits der Adresse des gesuchten Datums kann herausgefunden werden, ob sich das Datum tatsächlich im Cache befindet.

- **Index:**
  Der Index gibt an, in welchem Cacheblock (oder Set beim mehrfach assoziativen Cache) sich das gesuchte Datum befindet, falls es dort ist.

- **Word:**
  Das Feld Word gibt an, welches Wort innerhalb des Blocks angefordert wird.

- **Byte:**

Das Feld Byte wird normalerweise nicht benutzt. Es spricht ein einzelnes Byte innerhalb des Cacheblocks an. Bei einem Cache mit einer 32 Bit Datenbreite, der nur 32 Bit-Wörter liefert, ist dieses Feld immer 0 [TG01].

2.3.3 Cacheorganisationen

In einem Cache werden Daten zwischengelagert. Hier stellt sich die Frage, in welchem Cacheblock diese Daten abgespeichert werden können. Dafür gibt es 3 Cacheorganisationen [TG01]:

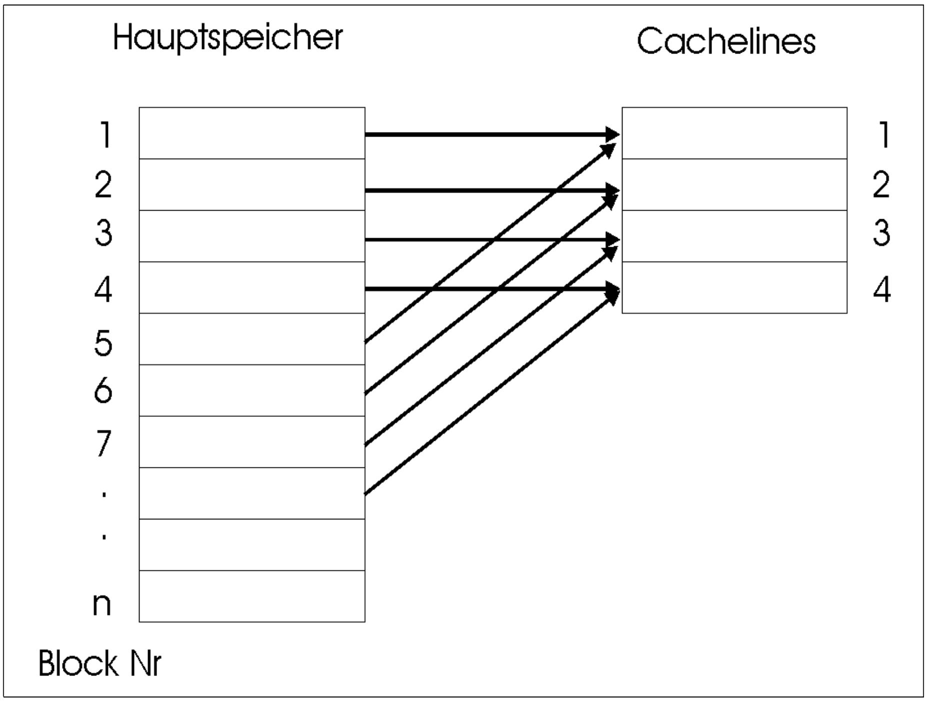


Abbildung 2.10 Ein „Direct-Mapped Cache"

- **Direct-Mapped Cache:**
  Die einfachste Cacheorganisation ist der „Direct-Mapped Cache". Beim „Direct-Mapped Cache" kann jeder Speicher-Block im Hauptspeicher nur in einen bestimmten Cacheblock geladen werden (*Abbildung 2.9* und *2.10*). Da jeder Block im Hauptspeicher nur in einem bestimmten Cacheblock abgespeichert werden kann, ist eine Ersetzungsstrategie gar nicht notwendig. Allerdings kann es in einem bestimmten Fall vorkommen, dass ein System mit einem „Direct-Mapped Cache" noch langsamer als ein System ohne Cache ist. Wenn die Adressen, die für einen bestimmten Cacheblock benutzt werden, nacheinander in den Cache geladen werden sollen, hat man ein großes Problem, weil der Cacheeintrag nach jedem Laden ersetzt werden muss. In diesem Fall produziert der Cache jedes mal einen Cachemiss, was zusätzliche Zeit und Energie kostet.

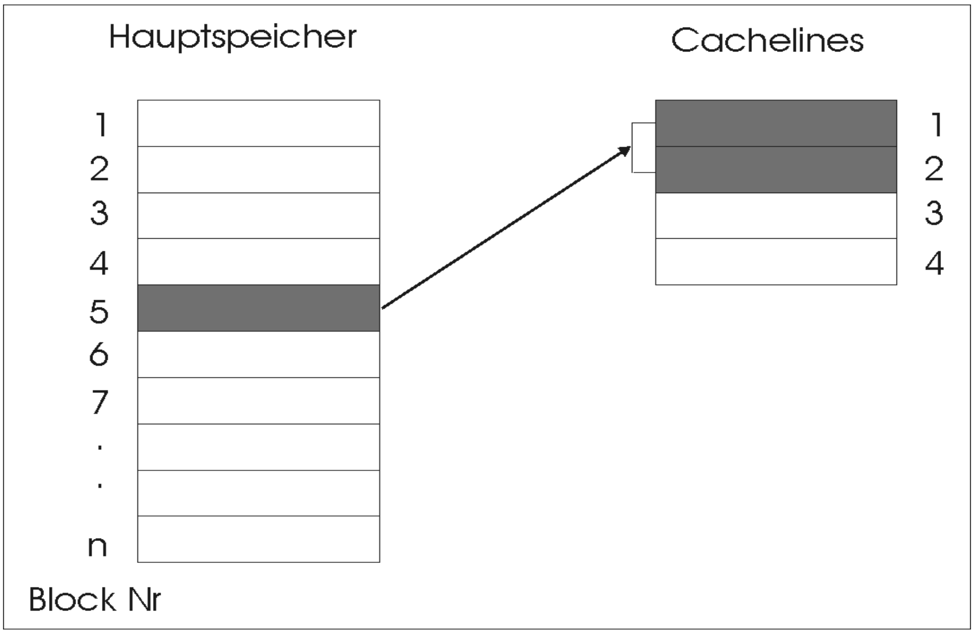


Abbildung 2.11 Ein 2-fach assoziativer Cache

- **set-assoziativer Cache (teil-assoziativer Cache):**
  Beim teil-assoziativen Cache kann ein Speicherblock im Hauptspeicher in einen Block in einem bestimmten Set geladen werden. Dies wird n-way set-assoziativer Cache genannt. Das n gibt die Anzahl der Sets des Caches an. Das hat den Vorteil, dass der oben beschriebene Konflikt vermieden werden kann. Da für einen Block des Hauptspeichers mehrere Blöcke in dem bestimmten Set des Caches in Frage kommen können, muss dieses entsprechend geregelt werden. Diese Ersetzungsstrategien werden im Kapitel 2.3.4 näher erläutert.

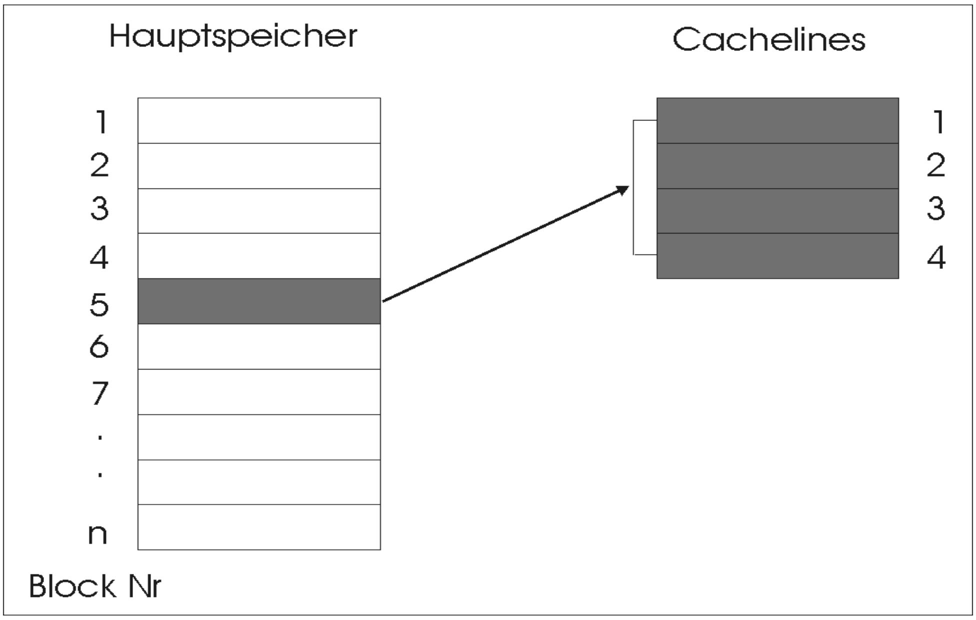


Abbildung 2.12 Ein voll-assoziativer Cache

- **Voll-assoziativer Cache:**

Im seltensten Fall gibt es auch einen voll-assoziativen Cache. Beim voll-assoziativen Cache kann ein Speicherblock in jeden beliebigen Cacheblock geladen werden. Da die Steuerung mit dem steigenden n erheblich erschwert wird, wird ein Cache mit der mehr als 4-fachen Assoziativität kaum angewendet.

2.3.4 Cachealgorithmen

Ein Cache lebt von zeitlichen und räumlichen Lokalitäten. Das heißt, ein Cache sollte möglichst viele Daten, die häufig benutzt werden, im Cache bereit stellen. Aber ein Cache hat eine begrenzte Speicherkapazität, sodass bei jedem Cachemiss irgendein Cacheblock ersetzt werden muß. Hier stellt sich die Frage, welcher Block ersetzt werden soll.

Es gibt drei Ersetzungsstrategien beim Lesen, die häufig benutzt werden. Und zwar Random, Round-Robin und LRU [TG01] [HP96].

- **Random:**
  Der Cache-Block, der ersetzt werden soll, wird zufällig ausgewählt. Dies ist einfach zu realisieren. Allerdings kann es vorkommen, dass der Cacheinhalt zufällig ausgewählt wird, auf den häufig zugegriffen wird.

- **Round-Robin:**
  Die Cache-Blöcke werden nacheinander ersetzt. Dabei wird die Häufigkeit des Zugriffs auf den jeweiligen Block nicht berücksichtigt. Es wird der Block ersetzt, der am längsten im Cache liegt.

- **LRU (Least Recently Used):**
  Der Block, der am längsten nicht mehr gelesen wurde, wird ersetzt. Diese Strategie baut auf dem Lokalitätsprinzip auf. Da die Wahrscheinlichkeit dafür, dass der kürzlich verwendete Block wieder verwendet wird, sehr hoch ist, löscht man am besten den am wenigsten benutzten Block. Nach dem zeitlichen Lokalitätsprinzip ist das Ergebnis von *LRU* etwas günstiger, dafür ist der Aufwand am größten. Vor allem ist der Aufwand für *LRU* beim mehr als 4-fachen set-assoziativen Cache zu groß, so dass solche Caches kaum in der Praxis zu finden sind.

Bisher wurde das Verhalten eines Caches beim Lesen beschrieben. Da die Daten nicht nur vom Cache gelesen sondern auch geschrieben werden, muss dieses Schreiben auch geregelt werden.

Schreibt ein Prozessor ein Wort, welches im Cache enthalten ist, muss er entweder das Wort im Cache aktualisieren oder verwerfen. Bei fast allen Designs wird der Cache-Eintrag aktualisiert. Aber was geschieht mit dem Wort im Hauptspeicher? Diese Aktualisierungsoperation kann sofort geschehen oder später ausgeführt werden, wenn der Cacheblock ersetzt werden muss. Ersteres nennt man *Write-Through* und das letztere *Write-Back*.

- **Write-Through:**
  Das ist das Verfahren, bei dem der Cache die Daten sofort in den Arbeitsspeicher schreibt. Die Steuerung für den Schreibvorgang wird vom Cache übernommen. Der Prozessor kann in dieser Zeit weiterarbeiten.

- **Write-Back:**
  Die Daten im Cache werden aktualisiert, aber nicht die im Hauptspeicher. Die Daten im Hauptspeicher werden erst aktualisiert, wenn der Cache-Block, der die Daten enthält, ersetzt werden muß.

Was passiert, wenn das Wort nicht im Cache vorhanden ist? Entweder wird das Wort in den Cache überführt (*Write-Allocate*) oder nur in den Speicher geschrieben (*No-Write-Allocate*). Die meisten Designs, die auf *Write-Through* basieren, nutzen *No-Write-Allocate*, weil sonst das einfache Design, welches der Vorteil des *Write-Through* Verfahrens ist, erschwert wird.

In dem nächsten Kapitel wird vor allem das Energiemodell des Scratch-Pad-Speichers und des Caches beschrieben. Es wird näher erläutert, wie der einzelne Energieverbrauchswert in Abhängigkeit von der On-Chip-Größe und den Cacheorganisationen berechnet wird und welches Werkzeug dafür eingesetzt wurde.

Kapitel 3

Energiebetrachtung von Scratch-Pad und Cache

Ein niedriger Energieverbrauch ist im Bereich EIS (eingebettete informationsverarbeitende Systeme) ein wichtiges Optimierungsziel neben Geschwindigkeits- und Codegrößenoptimierung. Theokharidis hat im Rahmen seiner Diplomarbeit [Theo00] gezeigt, dass die Zugriffe auf den Hauptspeicher neben den zusätzlichen Wartezyklen auch sehr viel Energie verbrauchen (ca. Faktor 10 gegenüber On-Chip).

Allerdings hat er eine Konstante für den Energieverbrauch des Scratch-Pad-Speichers angenommen, sodass der Energieverbrauch des On-Chip-Speichers unabhängig von der Speicherkapazität war. Im Rahmen dieser Diplomarbeit wurde der Energieverbrauch durch den Einsatz von zwei On-Chip-Speicherarten, Cache und Scratch-Pad, untersucht, und unterschiedliche Werte für den Energieverbrauch der beiden On-Chip-Speicher in Abhängigkeit von der Speicherkapazität (Scratch-Pad und Cache) und Organisation (Cache) berücksichtigt.

Wenn man die Wahl hat, zwischen Cache und Scratch-Pad zu entscheiden, benötigt man eine Entscheidungshilfe, damit man den passenden On-Chip-Speicher für jeden Anwendungszweck effektiv einsetzen kann. Daher ist es notwendig, den Energieverbrauch der zwei On-Chip-Speicher, Cache und Scratch-Pad, zu ermitteln.

Da existierende Compiler, Simulatoren und Profiler keinen Energieverbrauch berücksichtigen, muss zuerst ein grundlegendes Energiemodell ausgewählt werden.

In diesem Kapitel werden wichtige Begriffe für die Energiebetrachtung erläutert und die Energiemodelle für die beiden On-Chip-Speicher vorgestellt.

3.1 Physikalische Grundlagen

In diesem Unterkapitel werden die grundlegenden Begriffe „Leistung“ und „Energie“ definiert. Diese bei der „Low Power“-Forschung sehr häufig verwendeten Begriffsdefinitionen sind aus der Literatur [Pre86] entnommen worden.

Die Leistung P ist definiert als das Produkt aus der Spannung U und dem Strom I .

$$P = U * I \tag{3.1}$$

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

Die Energie E ist definiert durch das Produkt aus der Leistung P und der Zeit t.

$$E = P * t \tag{3.2}$$

Die Einheit von Energie E ist $J\ (Joule) = W\ (Watt) * s\ (Sekunde) = V * A * s$.

Mit den beiden Gleichungen 3.1 und 3.2 ergibt sich die Energie E:

$$E = P * t = U * I * t \tag{3.3}$$

3.2 Die Ursachen des Energieverbrauchs

Da CMOS-Schaltungen für die meisten Speicher und Prozessoren eingesetzt werden, sollen die Ursachen des Energieverbrauchs der CMOS-Schaltungen näher erläutert werden. Ein elektrischer Strom fließt nicht konstant bei einer Befehlsausführung, sondern durch kurze Stromsteigerung beim Umschalten einer CMOS-Schaltung. Daher wird ein durchschnittlicher Wert für den Energieverbrauch einer CMOS-Schaltung ermittelt. Die Formel (3.4) gibt die wichtigen Faktoren für den Energieverbrauch an [Low00].

$$\boldsymbol{E_{summe} = E_{sw} + E_{short} + E_{leakage}} \tag{3.4}$$

Die vier Komponenten sind E_{sw}: Energie durch die Schaltströme, E_{short}: Energie durch die Kurzschlußströme, $E_{leakage}$: Energie durch die Leckströme.

Die Ursachen für den Energieverbrauch einer CMOS-Schaltung sind folgende:

1. **Schaltstrom:**

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

$$P_{sw} = \frac{1}{2} V_{dd}^2 \sum_i (C_{Load_i} * TR_i) \qquad (3.5)$$

Mit V_{dd}: Versorgungsspannung, C_{loadi}: Lastkapazität einer Zelle und TR_i: Wechselrate beim Laden und Entladen der Kapazität.

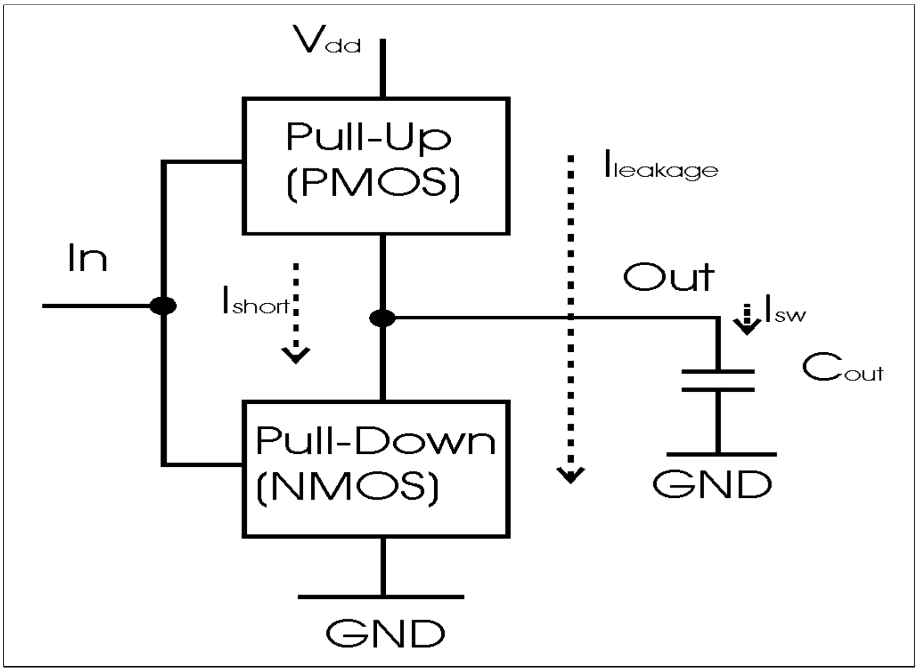


Abbildung 3.1 Eine CMOS-Schaltung
($I_{leakage}$: Leckstrom, I_{short} : Kurzschlußstrom, I_{sw} : Schaltstrom *)*

2. Leckstrom:

Leckströme sind unerwünschte, sehr kleine Ströme, die bei angelegter Spannung durch ein isolierendes Material hindurchfließen. Das tun sie deshalb, weil es in der Praxis keine absolut perfekten Isolatoren gibt. Bei Halbleitern treten sie an verschiedensten Stellen auf.

Beispielsweise nehmen wir eine Diode, die angeblich den Strom nur in einer Richtung fließen lässt. Richtig ist, dass die Diode in einer Richtung den Strom sehr gut fließen lässt und in der anderen Richtung nur ganz schlecht. Bei Dioden nennt man diesen Leckstrom auch Sperrstrom.

Mit 1 % hat dieser Leckstrom einen geringen Anteil an dem Gesamt-Energieverbrauch bei einer aktiven Schaltung und dieser tritt auch bei einer inaktiven Schaltung auf. Allerdings wird dieser geringen Anteil (1 %) mit abnehmender Versorgungsspannung größer, sodass er in der Zukunft mehr Bedeutung bekommen wird.

$$P_{Leak} = \sum_i P_{CellLeakage_i} \tag{3.6}$$

$P_{CellLeakagei}$ ist der Energieverbrauch einer Zelle *i* in einem inaktiven Zustand.

3. Kurzschlußstrom:

Dieser Strom fließt beim Schalten von Transistoren kurzzeitig. Je nach der Übergangszeit kann der Anteil am Gesamtenergieverbrauch bis zu 30 % liegen.

Die Formel für diesen Energieverbrauch ist:

$$P_{short} = \sum_j f(C_{Load_j}, Tr_{input}) * TR_j \tag{3.7}$$

Wobei Tr_{input} der Schaltzeit eines Transistors entspricht.

3.3 Energiemodell

Die Energiemodelle für die beiden On-Chip-Speicher werden benötigt, um den Energieverbrauch durch den Einsatz des Caches und des Scratch-Pads anhand der Simulationsausgabe zu berechnen und zu vergleichen.

Tiwari hat in seinen Arbeiten [TMW94] [TIW96] ein allgemeines Energiemodell für Prozessor-Instruktions-Kosten vorgestellt. Dieses Modell dient auch als Grundlage des Energiemodells für die beiden On-Chip-Speicher. Da das Modell keine Speicherkosten beinhaltet, wurde das Modell für On-Chip- und Off-Chip-Speicher erweitert.

Das Energiemodell besteht aus Prozessorkosten, On-Chip-Speicherkosten und On-Chip-Speicherkosten. Die einzelnen Kosten werden in den folgenden Unterkapiteln näher beschrieben.

Der direkte Vergleich des Energieverbrauchs von beiden On-Chip-Speichern wäre nicht fair, weil auf den Cache und auf den Scratch-Pad unterschiedlich zugegriffen wird. Da die Zugriffe auf den Hauptspeicher bei den beiden Konfigurationen unterschiedlich sind, muss dessen Energieverbrauch fairerweise auch mitberücksichtigt werden.

Da es keine Möglichkeiten gegeben hat, explizit den Energieverbrauch des Caches und des Scratch-Pads zu messen, wurde der Energieverbrauch der beiden On-Chip-Speicher durch das Programm *„CACTI“* (Cache Access and Cycle Time Information) [WJ94], [WJ96] berechnet.

Abbildung 3.2 Energieverbrauch eines Prozessorsystems

Der Energieverbrauch eines Scratch-Pad hängt von der Größe (Speicherkapazität) ab, und der Energieverbrauch eines Caches hängt sowohl von der Größe als auch von der Cacheorganisation stark ab. Somit wurden unterschiedliche Werte für unterschiedliche On-Chip-Kapazitäten und unterschiedliche Cacheorganisation bei der Simulation berücksichtigt.

3.3.1 Prozessorkosten

Der Prozessorkosten besteht aus *Basic Costs, Inter-Instruction Costs* und *Pipeline & Cachemiss Costs* [TIW96].

1. **Basic Costs:**
   Jede Befehlsausführung innerhalb eines Programms bewirkt eine Zustandsänderung am Prozessor. Diese Zustandsänderungen (eine erhöhte Schaltkreisaktivität) sind die Energiegrundkosten einer Instruktion.

   $$E_{Base} = \sum_i (B_i * N_i) \qquad (3.8)$$

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

2. **Kosten für Pipeline-Stalls und Cachemisses:**
   Die meisten modernen Prozessoren haben Pipelining zur Erhöhung ihrer Leistung. Pipelining ist eine Methode der parallelen Datenverarbeitung. Es gibt mehrere Phasen bei einer Ausführung einer Instruktionen. Die Instruktionen

werden nacheinander, sobald eine Phase der vorherigen Instruktion beendet ist, ausgeführt, so dass ein großer Teil ihrer Bearbeitung parallel laufen kann. Allerdings können nicht alle Phasen parallel laufen. Bei einem Pipeline-Stall wird die Weiterausführung der Befehle in der Pipeline durch eingefügte interne Wartebefehle vorerst angehalten, bis der Pipeline-Stall aufgehoben ist. Bei einem Cachemiss wird nach einem misslungenen Cacheread/-write auf den Hauptspeicher zugegriffen. All das bedeutet zusätzliche Energiekosten, die berücksichtigt werden müssen. Für den Pipeline-Stall sind drei Ursachen bekannt:

1) **Resource Hazard:**
   Dieser entsteht, wenn von zwei Instruktionen auf gleiche Systemressourcen zugegriffen wird. In diesem Fall wird in der Pipeline ein interner Wartezyklus eingefügt.

2) **Data Hazard:**
   Dieser entsteht, wenn eine Instruktion einen Eingabewert benötigt, der von einer anderen Instruktion erzeugt wird. Diese beiden Instruktionen können nicht parallel ausgeführt werden.

3) **Control Hazard:**
   Durch konditionale oder unkonditionale Sprüngen entsteht ein Control Hazard. Diese Veränderung des linearen Kontrollflusses macht die überlappende Ausführung der betroffenen Instruktionen unmöglich, weil die Adresse einer Instruktion von der vorhergehenden Instruktion bestimmt wird.

Im Rahmen dieser Diplomarbeit wurde das ARM7-Prozessorsystem mit einem Cache- bzw. mit einem Scratch-Pad-Speicher untersucht. Die Kosten des Pipeline-Stalls und des Cachemisses sind als „*MI*“ (Memory Idle) in der ARMulator-Ausgabe zu sehen. Allerdings gibt es keinen separaten Eintrag für Pipeline-Stalls.

```
IT 0050002A aa15 ADD      r2,r13,#0x54
MSW4____ 0057FFEC 00000000
MSW4____ 0057FFF0 0050008C
MSW4____ 0057FFF4 0040000D
E 00500030 004000B0 10002       /Event für Cachemiss
MI                              /Memory Idle
```

Abbildung 3.3 Cachemiss-Event in einer ARMulator-Ausgabe

Die Kosten für Pipelinestalls und Cachemisses werden wie folgt berechnet:

$$E_{PipelineCachemiss} = \sum_k (E_k * N_k) \tag{3.9}$$

Die Energiekosten E_k werden bei jedem Auftreten (N_k) von den Pipeline-Stalls und Cachemisses anhand der Berücksichtigung der „*MI's*“ berechnet.

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

Die Inter-Instruktionskosten fallen sowohl für den Prozessor als auch für den Speicher an. Durch die Änderung der On-Chip-Größe werden diese Kosten nicht in einer bestimmte Richtung beeinflusst, und beim Vergleich der beiden On-Chip-Speicher können diese Kosten die Berechnung der On-Chip-Speicher unnötig erschweren. Deshalb wurden diese Kosten als Durchschnittswert auf die Basis-Kosten aufgeschlagen.

Die Prozessorkosten E_{cpu} sind die Summe der Basiskosten E_{Base} und der Pipeline Stall und Cachemiss-Kosten $E_{PipilineCachemiss}$.

$$E_{cpu} = E_{Base} + E_{PipelineCachemiss} \tag{3.10}$$

3.3.2 On-Chip-Speicherkosten (Cache)

Um die Kosten, die bei dem On-Chip-Cache anfallen, zu berechnen, wurde zuerst der Energieverbrauch des Caches pro Zugriff berechnet. Dazu wurde das „CACTI-Modell“ [WJ94], [RJ99] verwendet. Dieses Modell kann den Energieverbrauch für eine 0,5µm Chiptechnologie (wie bei ARM7T) berechnen.

Der Aufruf auf der Kommandozeile sieht wie folgt aus:

cacti **C B A TECH**

C : Cachegröße in Byte
B : Blockgröße in Byte
A : Set-Assoziativität
TECH : Chip-Technologie

Es gibt weitere Eingabeparameter, die in der Datei „*def.h*“ eingetragen werden müssen. Diese Parameter sind die Datenbusbreite in Bit und die Adressbusbreite (*Abbildung 3.4*).

```
// Address bits in a word, and number of output bits
from the cache

#define ADDRESS_BITS 24
#define BITOUT 32
```

Abbildung 3.4 Eingabeparameter in der „def.h“

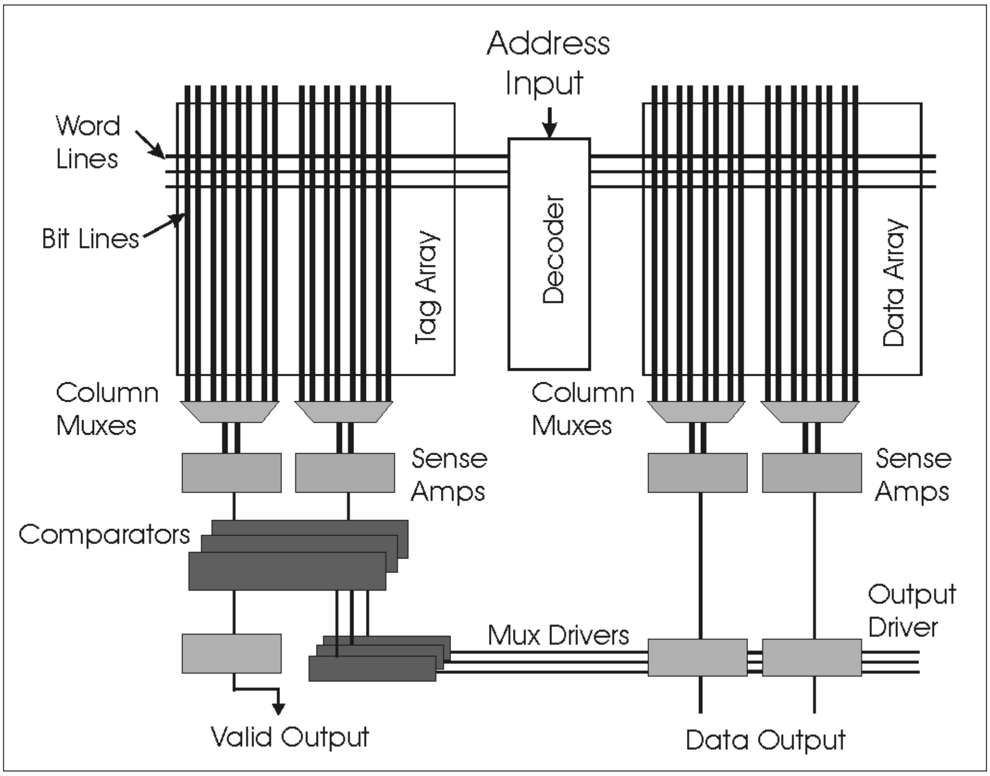


Abbildung 3.5 Cache-Organisation

Das CACTI-Modell erzeugt detaillierte Informationen über den Energieverbrauch jeder einzelnen Cachekomponente. In der *Abbildung 3.5* sind die Cachekomponenten, die bei dem CACTI-Modell für die Berechnung des Energieverbrauchs berücksichtigt wurden, dargestellt.

Der Energieverbrauch des Caches pro Zugriff wird wie folgt berechnet:

$$E_{Cache,i,j} = \sum (E_{Dataarray} + E_{Tagarray} + E_{Decoder} + E_{Wordline} + E_{Bitline} + E_{Senseamp} + E_{Columnmuxes} + E_{Compare} + E_{Valid} + E_{outputdriver}) \tag{3.11}$$

i	: Cache-Größe
j	: Set-Assoziativität
$E_{Dataarray}$	: Energieverbrauch des Data-Arrays
$E_{Tagarray}$	: Energieverbrauch des Tag-Arrays
$E_{Decoder}$	: Energieverbrauch des Decoders (für Tag und Data)
$E_{Wordline}$	: Energieverbrauch der Wordlines
$E_{Bitline}$	: Energieverbrauch der Bitlines
$E_{Columnmux}$	: Energieverbrauch der Column Muxes (für Tag und Data)
$E_{Senseamp}$	: Energieverbrauch der Sense Amps (für Tag und Data)
$E_{Compare}$	: Energieverbrauch der Comparators
E_{Valid}	: Energieverbrauch des Valid-Output-Drivers
$E_{Outputdriver}$	: Energieverbrauch des Output-Drivers

In der *Tabelle 3.1* sind die Werte des jeweiligen Energieverbrauchs in Abhängigkeit von den Cache-Größen und Cache-Organisationen eingetragen. Die Werte in den markierten Zellen wurden approximiert, weil die Speicher-Größen bei den Cache-Organisationen für das CACTI-Modell zu klein waren. Zuerst wurde die durchschnittliche Steigung der simulierten Werten gebildet. Mit dieser Steigung wurde die restlichen Werte, die das CACTI-Modell nicht berechnen konnte, berechnet. Z.B. wurden bei der 2-fach assoziativen Cache die jeweilige Steigung zwischen den Speichergrößen von 128 Byte bis 4096 Byte berechnet und anschließend ein Mittelwert (9 %) gebildet. Mit diesem Mittelwert wurde der Energieverbrauch des Cache-Speichers mit der Speichergröße von 64 Byte berechnet. (1,79nJ * 0,09 = 0,16 nJ) Der Wert bei der Speichergröße von 8192 Byte wurde wegen der zu hohen Abweichung ausgenommen.

Tabelle 3.1 Energieverbrauch des Caches

Speichergröße	DA	2X	4X	8X	FA
64 Byte	0,71 nJ	1,63 nJ	2,71 nJ	5,52 nJ	1,20 nJ
128 Byte	0,76 nJ	1,79 nJ	3,05 nJ	5,87 nJ	1,36 nJ
256 Byte	0,86 nJ	1,90 nJ	3,32 nJ	6,24 nJ	2,03 nJ
512 Byte	0,98 nJ	2,05 nJ	3,48 nJ	6,63 nJ	2,47 nJ
1024 Byte	1,15 nJ	2,23 nJ	3,75 nJ	6,92 nJ	3,33 nJ
2048 Byte	1,47 nJ	2,55 nJ	4,04 nJ	7,37 nJ	6,00 nJ
4096 Byte	1,69 nJ	2,88 nJ	4,71 nJ	7,95 nJ	9,18 nJ
8192 Byte	2,67 nJ	3,57 nJ	5,39 nJ	8,89 nJ	17,24 nJ

DA : Direct-Mapped Cache
2 – 8X : 2 bis 8 fach assoziativer Cache
FA : Voll assoziativer Cache

Da der Simulator (ARMulator) keine detaillierten Informationen über das Cacheverhalten ausgibt, wurde ein Cacheverhaltensmodell erstellt. Mit diesem Verhaltensmodell wird die ARMulatorausgabe analysiert.

Der zur Simulation eingesetzte On-Chip-Cache des ARM710Ts arbeitet mit „write-through" und „no write allocate". Die Cachelinegröße (Cacheblockgröße) wurde bei allen Simulationsversuchen auf 8 Byte eingestellt.

In der *Tabelle 3.2* sind das Cacheverhaltensmodell mit der Anzahl der Cachezugriffe und die erforderlichen Prozessorzyklen dargestellt.

Tabelle 3.2 a) Anzahl der Cachezugriffe beim ARM710T

Access Type		Cache Read	Cache Write	Main Memory Read	Main Memory Write
Instruction	Read Hit	1	0	0	0
	Read Miss	1	N[1)]	N	0
Data	Read Hit	1	0	0	0
	Read Miss	1	N	N	0
	Write Hit	0	1	0	1
	Write Miss	1	0	0	1

[1)] N = Anzahl der Worte in einem Cacheblock. Bei der Simulation wurde ausschließlich N=2 eingestellt.

Tabelle3.2 b) CPU-Zyklen bei den Cachezugriffen

Zugriffsarten		Cache	Hauptspeicher	Summe
Instruktion	Read Hit	1 Zyklus	0 Zyklus	1 Zyklus
	Read Miss	1 Zyklus	2 * (1 Zyklus + 3 Wartezyklen)	9 Zyklen
Daten	Read Hit	1 Zyklus	0 Zyklus	1 Zyklus
	Read Miss	1 Zyklus	2 * (1 Zyklus + 3 Wartezyklen)	9 Zyklen
	Write Hit	1 Zyklus	1 Zyklus + 3 Wartezyklen	4 Zyklen [2)]
	Write Miss	1 Zyklus	1 Zyklus + 3 Wartezyklen	5 Zyklen

[2)] In der Simulatorausgabe konnte nur der Hauptspeicherzugriff festgestellt werden, daher müssen die beiden Schreibvorgänge parallel ausgeführt worden sein. (*Abbildung 3.6*)

Der ARMulator kann entweder die Information über die Kommunikation zwischen dem Prozessor und dem Cache oder über die Kommunikation zwischen dem Prozessor und dem Hauptspeicher ausgeben. In dem ersten Fall sind die Datentransfers zwischen dem Prozessor bzw. dem Cache und dem Hauptspeicher unsichtbar, und im zweiten Fall sind die Datentransfers zwischen dem Prozessor und dem Cache unsichtbar. Zur Berechnung des Energieverbrauchs wurde der zweite Fall betrachtet. Der erste Fall wurde auch betrachtet, um die gesamten Prozessorzyklen und Cachezugriffe zu überprüfen.

```
IT 0050001E 600b STR          r3,[r1,#0]     /Write Hit
MSW4___ 0057FCE4 E800E806
MNR2O__ 00500024 6011
IT 00500020 44a5 ADD          r2,#0xd8       /Read Hit
MNR2O__ 00500026 4C01
IT 00500022 2108 MOV          r1,#8
```

Abbildung 3.6 a) Datentransfer zwischen der CPU und dem Cache

```
IT 0050001E 600b STR          r3,[r1,#0]     /Write Hit
MI
MSW4___ 0057FCE4 E800E806
IT 00500020 44a5 ADD          r2,#0xd8       /Read Hit
MI
IT 00500022 2108 MOV          r1,#8
```

Abbildung 3.6 b) Datentransfer zwischen der CPU und dem Hauptspeicher

Der Energieverbrauch des Caches lässt sich durch die Summe der einzelnen Energieverbräuche des Caches pro Zugriff berechnen.

$$E_{Cache} = E_{cache,i,j} * N_k \tag{3.12}$$

N_k (die Anzahl der Cachezugriffe) ist die Summe der Anzahl der Lesezugriffe und der Schreibzugriffe des Caches.

$$N_k = N_{Read} + N_{Write} \tag{3.13}$$

3.3.3 On-Chip-Speicherkosten (Scratch-Pad)

Durch den Loader werden Daten und Programmteile in den Scratch-Pad-Speicher verschoben. D.h. ein Scratch-Pad-Speicher braucht keine hardwaremäßige Kontrolleinheit. Um den Energieverbrauch eines Scratch-Pad-Speichers zu berechnen, hat man fast die gleiche Vorgehensweise wie bei der Energieverbrauchs-berechnung eines Caches.

Zuerst wurde der Energieverbrauch pro Scratch-Pad-Zugriff berechnet und danach der gesamte Energieverbrauch des Scratch-Pad-Speichers für einen Durchlauf eines

Programms berechnet. Zu diesem Zweck wurde auch das „CACTI-Modell“ eingesetzt.

Tabelle 3.3 Energieverbrauch eines Scratch-Pads pro Zugriff

Kapazität	Energieverbrauch pro Zugriff
64 Byte	0,49 nJ
128 Byte	0,53 nJ
256 Byte	0,61 nJ
512 Byte	0,69 nJ
1024 Byte	0,82 nJ
2048 Byte	1,07 nJ
4096 Byte	1,21 nJ
8192 Byte	2,07 nJ

Obwohl dieses „CACTI-Modell“ ursprünglich für einen Cache entwickelt worden ist, kann es auch für einen Scratch-Pad-Speicher verwendet werden, indem man die für den Scratch-Pad unnötigen Cachekomponenten nicht berücksichtigt. Die Eingabeparameter bleiben bis auf die Cacheorganisation gleich, damit der Vergleich fair sein kann. Als Eingabeparameter für die Cacheorganisation wurde der Direct-Mapped Cache ausgewählt.

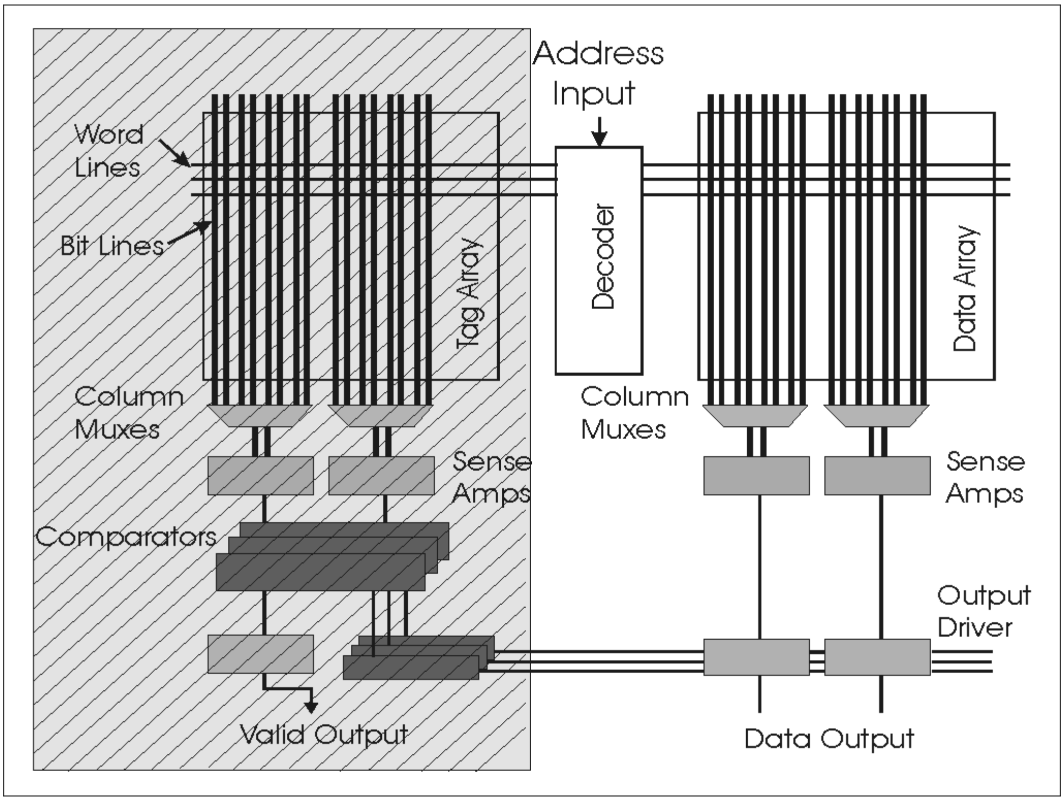


Abbildung 3.7 Berechnung des Energieverbrauchs für Scratch-Pad mit „CACTI-Modell“

Die berechneten Werte des Energieverbrauchs pro Zugriff sind in der *Tabelle 3.3* dargestellt. Ein Cache-Speicher verbraucht zwischen 145 % („Direct-Mapped Cache“ 64 Byte) bis 833 % (Voll assoziativer Cache 8192 Byte) mehr Energie als ein gleich großer Scratch-Pad-Speicher.

Die Komponenten in dem markierten Viereck in der *Abbildung 3.7* werden bei der Berechnung nicht berücksichtigt. Ein Zugriff auf den Scratch-Pad-Speicher kostet unabhängig von der Datenbreite einen Prozessorzyklus. Der gesamte Energieverbrauch eines Scratch-Pad-Speichers ist das Produkt des Energieverbrauchs des Scratch-Pad pro Zugriff mit der Anzahl der Zugriffe.

$$E_{Scratchpad} = \sum_{l} E_{Scratchpad,i} * N_l \qquad (3.14)$$

$E_{Scratchpad,i}$ ist der Energieverbrauch pro Zugriff in Abhängigkeit von der Größe des Scratch-Pad-Speichers, und N_l entspricht der Anzahl der Zugriffe auf den Scratch-Pad-Speicher.

3.3.4 Off-Chip-Speicherkosten

Die Kosten der Zugriffe auf den Hauptspeicher wurden von Theokharidis im Rahmen seiner Diplomarbeit [Theo00] berechnet. Diese Werte wurden für die Simulation weiter benutzt. Da die Anzahl der Zugriffe auf den On-Chip-Speicher und auf den Off-Chip-Speicher sehr eng mit der Größe des On-Chip-Speichers verbunden sind, ist dieser Kostenfaktor nicht zu vernachlässigen. Es gibt zwei Arten der Off-Chip-Speicherkosten.

1. Zugriffskosten beim Holen einer Instruktion:
   Diese Kosten entstehen beim Holen einer Instruktion aus dem Hauptspeicher.

2. Schreib- und Lesekosten für ein Datum:
   Wenn eine Instruktion auf ein Datum im Hauptspeicher zugreift, entstehen zusätzliche Kosten, weil das Datum beim Load in den Prozessor geschickt und beim Store in den Hauptspeicher zurückgeschrieben werden muss.

Tabelle 3.4 Prozessorzyklen des Speichersystems

Zugriffsart	Prozessorzyklen
On-Chip-Speicher 16 und 32 Bit	1 Zyklus
Hauptspeicher 16 Bit	1 Zyklus + 1 Wartezyklus
Hauptspeicher 32 Bit	1 Zyklus + 3 Wartezyklen

Da der ARM7T-Prozessor, der für die Simulation ausgewählt wurde, für das Holen eines Befehls den 16 Bit-Zugriff erlaubt, und bei der Simulation mit den eingesetzten Benchmarks für das Holen der Daten 32 Bit-Zugriff benutzt, sind die davon abhängigen Wartezyklen zu berücksichtigen. Diese Zyklen sind in der *Tabelle 3.4* zu sehen.

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

$$E_{Mainmemory} = \sum_i ((D_i + Z_i) * N_i) \tag{3.15}$$

D_i gibt die Kosten des Holens einer Instruktion aus dem Hauptspeicher (Instruction Fetch) an. Z_i ist der Energieverbrauch beim Lesen oder Schreiben eines Datums. N_i gibt die Anzahl der Zugriffe auf den Hauptspeicher an.

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

$$E_{Gesamt} = E_{cpu} + E_{Onchip} + E_{Mainmemory} \tag{3.16}$$

E_{Onchip} ist entweder der Energieverbrauch des Caches E_{Cache} oder der des Scratch-Pad-Speichers $E_{Scratchpad}$. Der Energieverbrauch des On-Chip- und Off-Chip-Speichers wird bei der Analyse zusammengesetzt und als Speicherkosten ausgegeben. In dem nächsten Kapitel werden die durchgeführten Simulationsversuche näher erläutert.

Kapitel 4

Simulationsumgebung

Im Rahmen der Diplomarbeit wird die Reduzierung des Energieverbrauchs durch den Einsatz eines On-Chip-Speichers untersucht. Um Cache- und Scratch-Pad-Speicher zu vergleichen, wurde der am Informatik-Lehrstuhl 12 der Universität Dortmund entwickelte Compiler namens „enCC" eingesetzt [EN01]. Der enCC besteht aus LANCE2 als Frontend und den Standardoptimierungen, prozessorspezifischen Optimierungen und der Befehls- und Registerausgabe als Backend. In dem Backend hat der enCC neben den üblichen Codeoptimierungen die Optimierungsoption für Scratch-Pad-Speicher. D.h. der enCC ist in der Lage, den Code zu analysieren und die globalen Variablen und häufig zugegriffene Basicblöcke oder Funktionen in den Scratch-Pad einzulagern. Damit die Simulation möglichst realistisch laufen kann, wurden zwei Zielarchitekturen, „ATMEL AT91M40400" und „CMA 222 ARM710T", betrachtet. Der Microcontroller „ATMEL AT91M40400" hat neben dem ARM7-Core einen Scratch-Pad-Speicher und „CMA 222 ARM710T" hat 8 Kbyte-Cache als On-Chip-Speicher. Als Simulationstool wurde der ARMulator [ARMu99] von ARM eingesetzt und für die Analyse der Tracedatei des ARMulator ein am Lehrstuhl vorhandenes Analyseprogramm „Traceanalyzer" erweitert und eingesetzt.

4.1 Simulationsablauf

Um zu einseitige Simulationsergebnisse zu vermeiden, wurden insgesamt neun Benchmarks aus unterschiedlichen Domänen z.B. Sortieralgorithmen, Matrizenberechnung, DSP-Benchmark oder ein Benchmark mit vielen For-Schleifen und komplexen arithmetischen Operationen eingesetzt. Die vollständige Simulation lässt sich in zwei Vorgänge und zwar Simulation mit Scratch-Pad und Simulation mit Cache aufteilen (*Abbildung 4.1*).

- **Simulationsablauf für Scratch-Pad-Modell:**

1. Der C-Code wird mit dem Compiler „enCC“ für den Prozessor „ARM7-TDMI“ mit unterschiedlichen Scratch-Pad-Größen compiliert.
2. Der „enCC“ erzeugt zwei Dateien *Dateiname_onchip.asm* und *Dateiname_-offchip-.asm*. Der On-Chip-Anteil wird in den Scratch-Pad und der Off-Chip-Anteil in den Hauptspeicher eingelagert. Dies ist die Aufgabe des Loaders.
3. Der Binärcode wird durch den „ARMulator“ simuliert.
4. Der „ARMulator“ erzeugt eine Tracedatei *Dateiname.trc*.
5. Diese Tracedatei wird mit dem Traceanalyzer analysiert.
6. Traceanalyzerausgaben

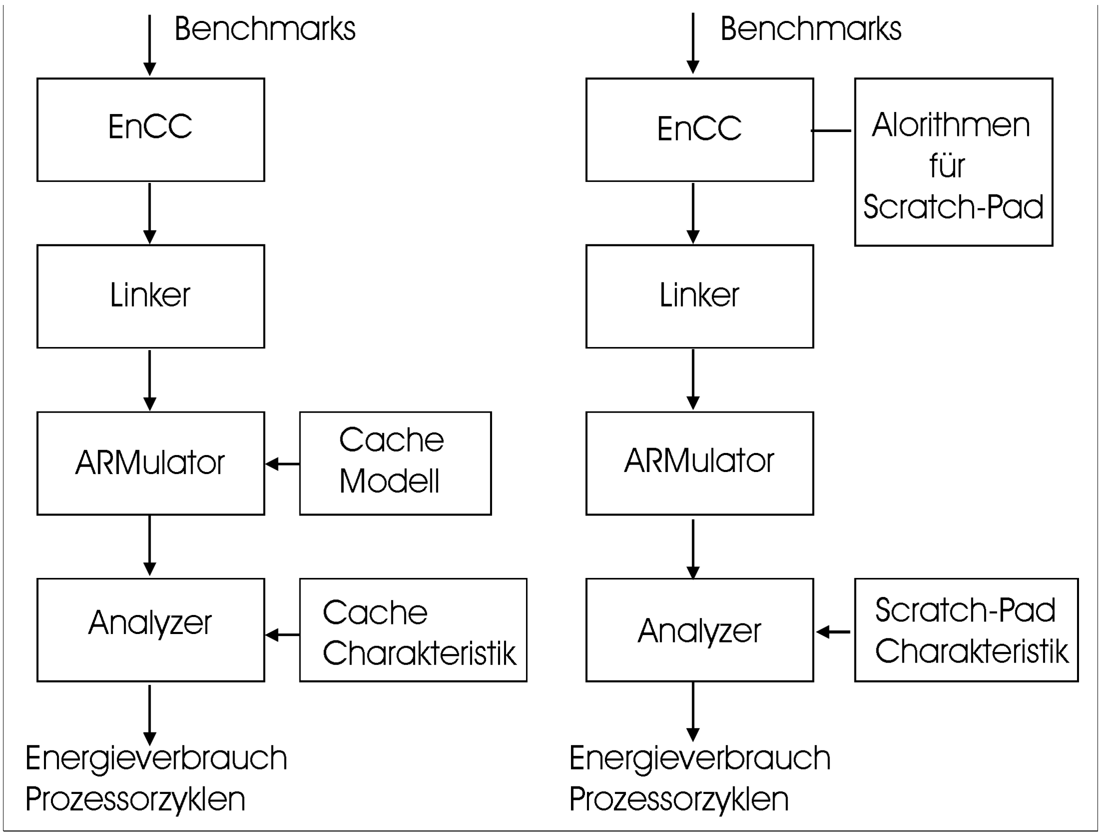


Abbildung 4.1 Der Simulationsablauf

- **Simulation für Cachemodell:**

1. Der C-Code wird mit dem Compiler „enCC“ für den Prozessor „ARM710T“

mit unterschiedlichen Cacheeinstellungen und Cachegrößen compiliert.

2. Der „enCC" erzeugt zwei Dateien *Dateiname_onchip.asm* und *Dateiname_-offchip-.asm*. In diesem Fall ist die Datei „*Dateiname_onchip.asm*" leer, weil die Compileroption „multimemory", die die Benutzung des Scratch-Pad-Speichers ermöglicht, deaktiviert bleibt.
3. Der Binärcode wird durch den „ARMulator" simuliert.
4. Der „ARMulator" erzeugt eine Tracedatei „*Dateiname.trc*".
5. Diese Tracedatei wird mit dem Traceanalyzer analysiert.
6. Traceanalyzerausgaben.

4.2 Betrachtete Hardware

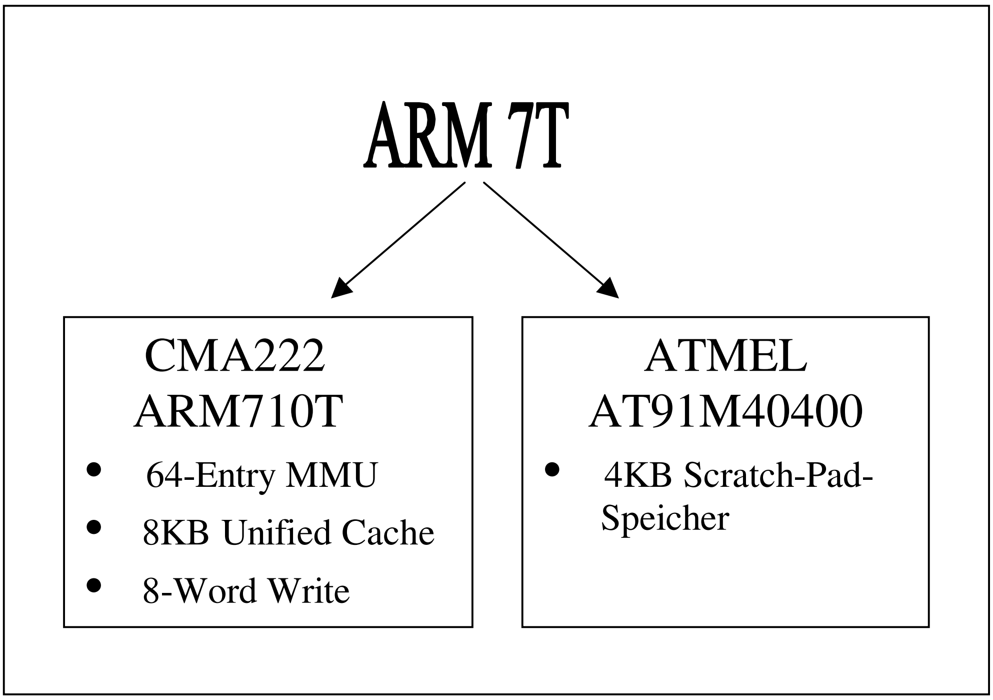


Abbildung 4.2 Die betrachtete Architektur

Um den Energieverbrauch von Cache und Scratch-Pad zu vergleichen, wurden zwei Prozessoren, „ARM7TDMI" und „ARM710T", betrachtet. Die beiden Prozessoren haben den selben ARM7-Prozessor-Core. Der „ARM7TDMI" selbst hat keinen On-Chip- Speicher, der „ARM710T" hingegen einen 4-fach assoziativen Cache.

Da zwei identische Prozessoren mit unterschiedlichen On-Chip-Speichern nicht von einem Hersteller zu erhalten waren, wurden zwei Microcontroller „ATMEL AT91M40400“ [ATM99a] und „CMA 222 ARM 710T“ [CMA01] von unterschiedlichen Herstellern ausgewählt. Die beiden Controller besitzen den „ARM7-Core“ mit den unterschiedlichen On-Chip-Speichern.

4.2.1 ATMEL AT91M40400

Der ATMEL AT91M40400 ist ein Low-Power-Microcontroller mit ARM7TDMI-Prozessor. Der ARM7TDMI-Prozessor ist ein 32bit RISC-Prozessor [ARM95b]. Der Prozessor ist bekannt durch seinen niedrigen Energiebedarf und wird daher sehr oft in dem Bereich EIS eingesetzt.

Der Prozessor besitzt neben dem 32 Bit ARM-Befehlssatz zusätzlich den 16 Bit Thumb-Befehlssatz. Der ARM-Befehlssatz zeichnet sich durch seinen großen Befehlssatz aus. Der Thumb-Befehlssatz besitzt eine hohe Codedichte und ist der Hauptgrund für den niedrigen Energiebedarf. Der Thumb-Befehlssatz deckt über 65% der ARM-Code-Größe ab und besitzt 160 % der Schnelligkeit eines vergleichbaren ARM-Prozessors mit einem 16 bit-Memorysystem [ARM95a]. Der Thumb-Befehlssatz und ARM-Befehlssatz können zusammen in einem Programm benutzt werden. Der Thumb-Befehlssatz beinhaltet insgesamt 36 Kerninstruktionen, und dem ARM-Befehlssatz stehen insgesamt 80 Kerninstruktionen zur Verfügung.

Der ARM7TDMI hat folgende Eigenschaften:

- 32bit RISC-Prozessor
- ARM-Befehlssatz und Thumb-Befehlssatz
- 31 x 32bit-Register und 6 x Statusregister
- 32 x 8 Multiplier
- Barrelshifter
- 32bit ALU
- 32bit-Adress- und Datenbus
- dreistufige Pipeline

In der *Abbildung 4.3* ist der ARM7TDMI-Prozesssor-Core dargestellt. Der für die Simulation notwendige Energieverbrauch des Prozessors und des Off-Chip-Speichers wurde von Theokharidis [Theo00] im Rahmen seiner Diplomarbeit gemessen. Bei der Messung wurde das ATMEL-Evaluationboard „AT91EB01“ mit dem Microcontroller „ATMEL AT91M40400“ eingesetzt [ATM98], [ATM99a].

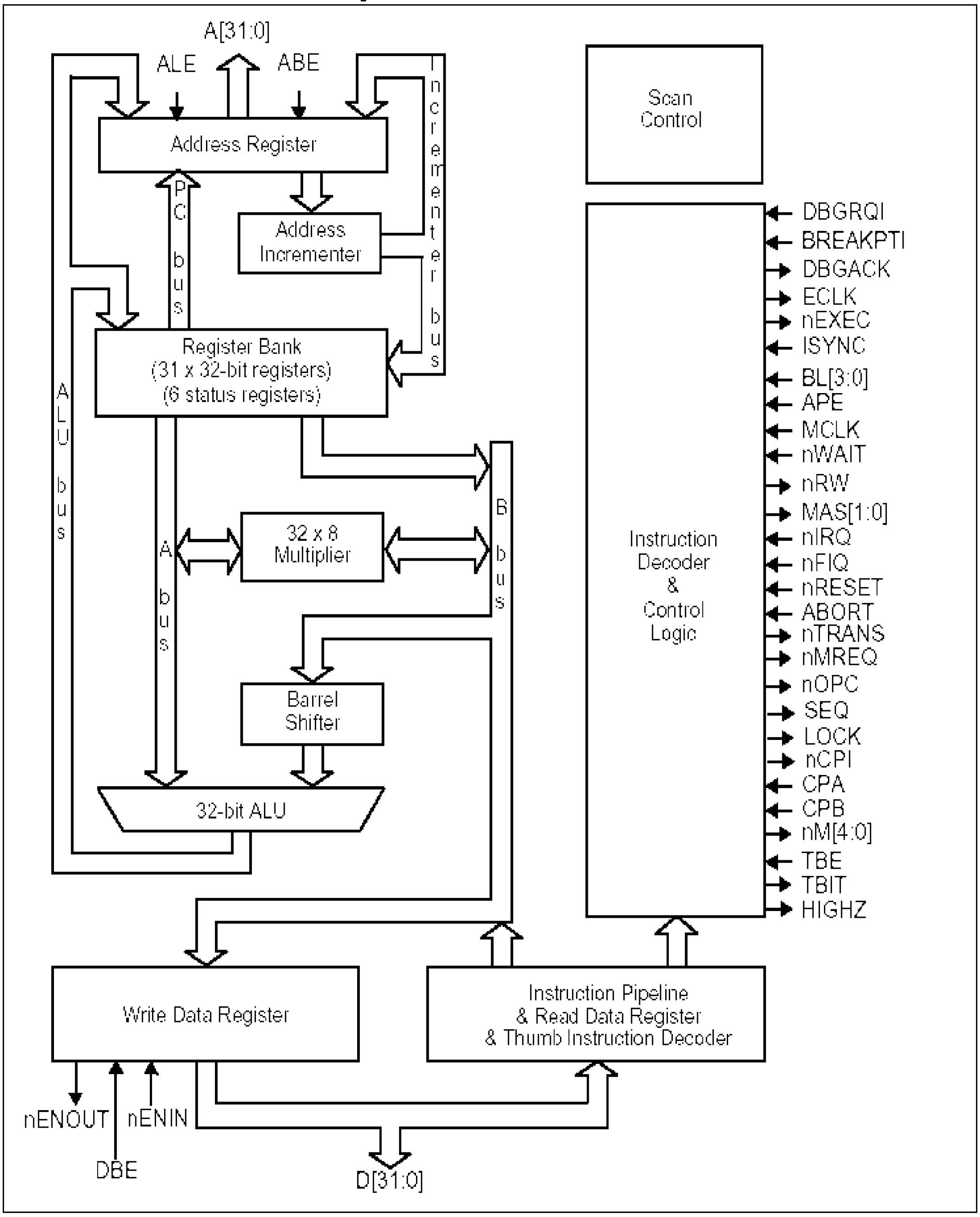


Abbildung 4.3 ARM7TDMI-Core [ARM95b]

Der auf dem Evaluationboard befindliche Microcontroller AT91M40400 unterstützt die beiden 32bit-ARM- und 16bit-Thumb-Befehlssätze und bietet die Möglichkeiten, gute Codedichte (ARM-Befehlssätze) vs. gute Performance und geringer Energieverbrauch (Thumb-Befehlssätze) an. In der *Abbildung 4.4* ist die Detailansicht vom AT91M40400 zu sehen.

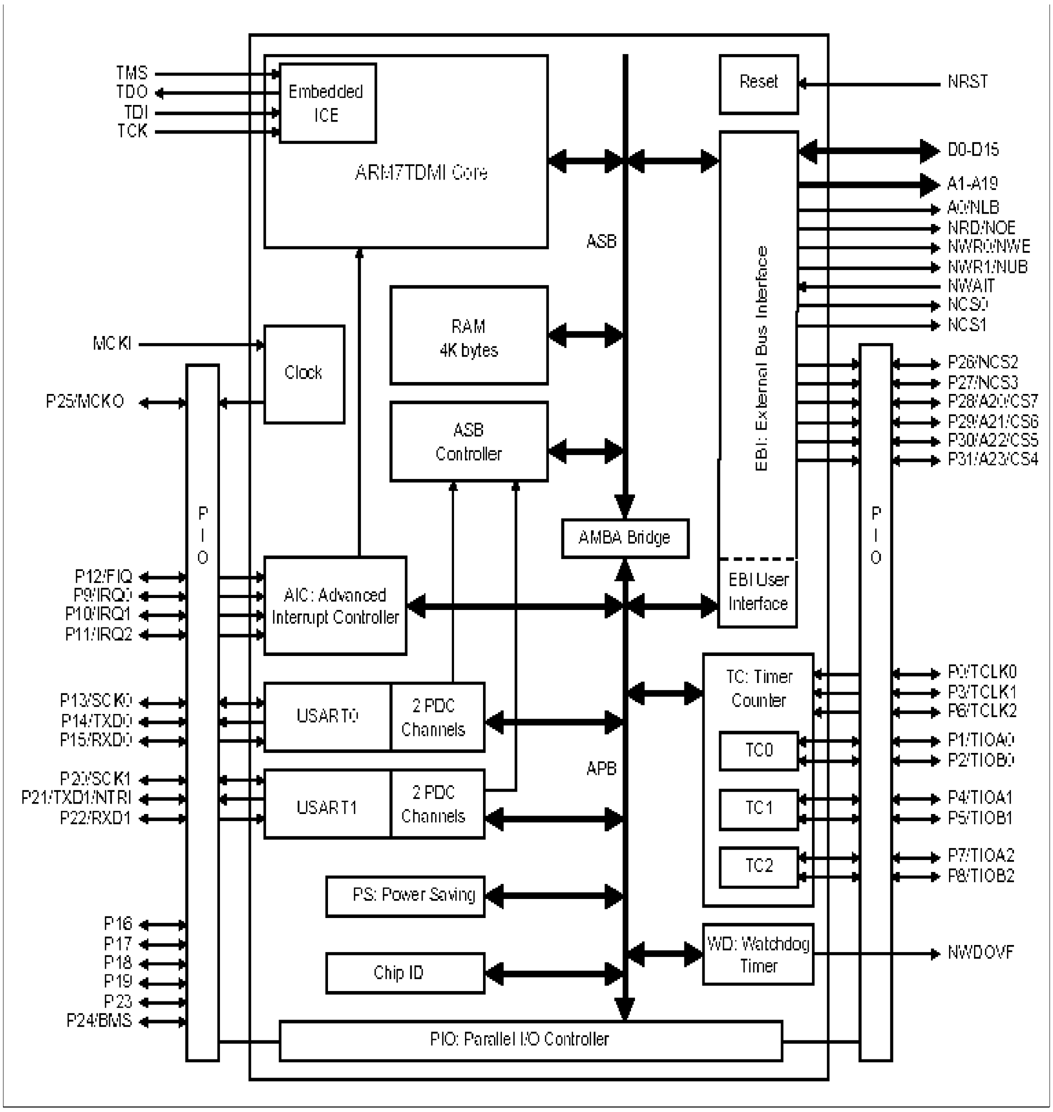


Abbildung 4.4 Detailansicht des AT91M40400

Der Microcontroller AT91M40400 [ATM99a] zeichnet sich durch folgende Eigenschaften aus:

- Eingebettete ICE-Schnittstelle für Multi-ICE
- 4KByte On-Chip-Speicher (Scratch-Pad)
- Interrupt-Controller mit 8-stufigen Prioritäten
- 32 programmierbare I/O-Leitungen
- Vollprogrammierbare External-Bus-Interface
- x 16-bit Counter/Timer
- Watchdog-Timer

- 2 x USARTs
- Low-Power-Optionen: „Low-Power-Idle“ und „Power-Down“ Modus

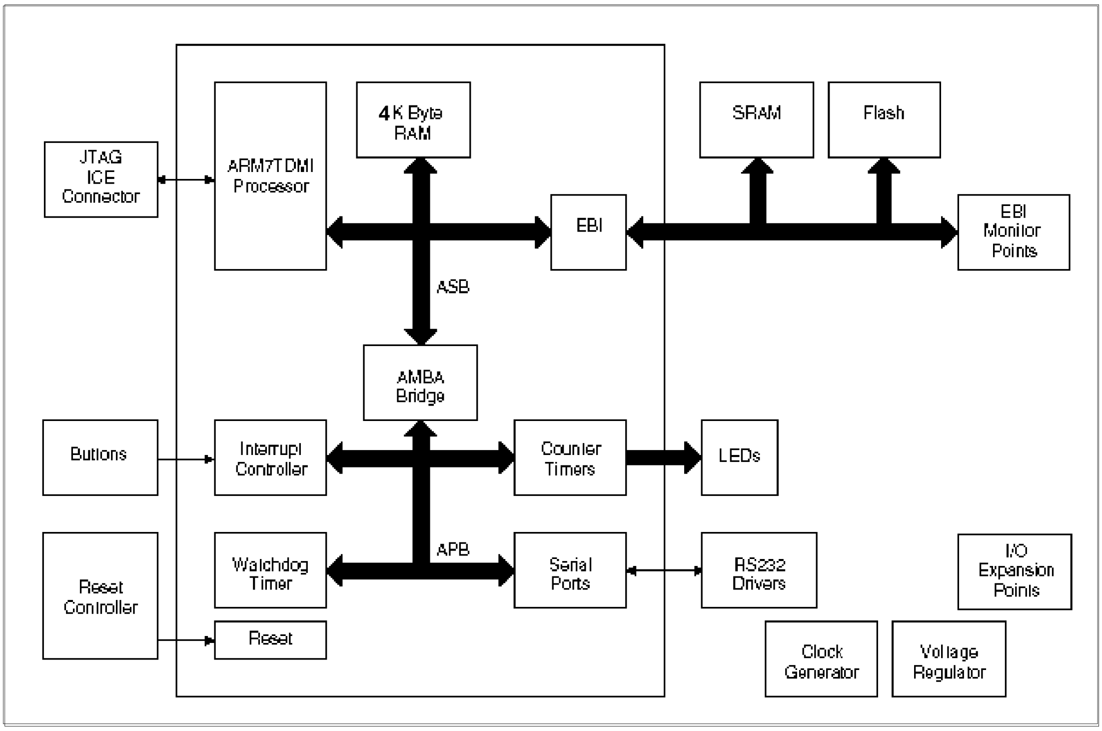


Abbildung 4.5 Blockschaltbild des AT91EB01 [ATM98]

Das Evaluationboard hat folgende Eigenschaften:

- AT91M40400 Microcontroller mit ARM7TDMI-Core
- 512K Byte 16 bit-Flash, davon sind 64 Kbyte frei verfügbar
- Die Bandbreite des Datenbusses beträgt 16 bit und die des Adressbusses 24 bit
- Das Board stellt 2 serielle Schnittstellen zur Verfügung
- JTAG ICE Debug Interface
- Angel Debug Monitor
- 3.3 V Versorgungsspannung
- Systemfrequenz beträgt 33MHz (Je nach Bedarf reduzierbar)

4.2.2 ARM710T

Für die Bildung des Cachemodells zur Simulation wurde das Prozessormodul „CMA 222 ARM710T“ von der Firma „Cogent Computer Systems“ ausgewählt. Der ARM710T wurde ausgewählt, weil der Prozessor den ARM7TDMI als Prozessor-Core und einen On-Chip-Cache-Speicher besitzt.

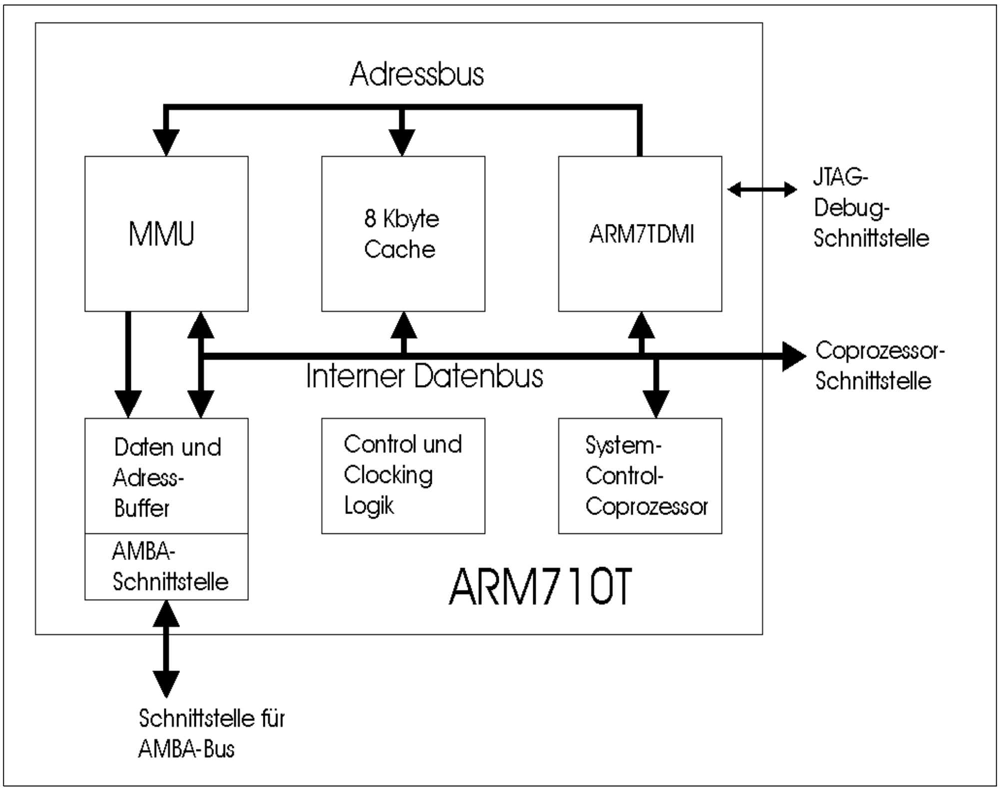


Abbildung 4.6 ARM710T Blockdiagramm

Der ARM710T-Prozessor ist ein 32bit RISC-Prozessor mit 8KB „unified“ Cache, Writebuffer und Memory Management Unit (MMU) in einem Chip. Der ARM710T ist „softwarekompatibel“ mit der ARM-Prozessor-Familie [ARM7T98]. Der Aufbau des ARM 710T-Prozessors ist in der *Abbildung 4.6* dargestellt. Für die Bildung des Cachemodells des ARMulators wurden die Eigenschaften des ARM710Ts vom „CMA 222 CPU Module“ betrachtet [CMA01].

Die Eigenschaften des ARM710T-Prozessors sind folgende:

- Industrie Standard ARM7TDMI-Core mit ARM- und Thumb-Befehlssatz
- 8 Kbyte unified Cache mit 4- fach Set-Assoziativität
- Der Cache hat 512 x 4 Words Cachelines
- Virtueller Cache

- „Write Through“ und „No Write Allocation“ Strategie („No Write Allocation“ wurde erst bei der Ergebnisanalyse der Simulation festgestellt.)
- 64-Entry TLB Memory Management Unit
- 3,3 V Versorgungsspannung
- 8 Words Write Buffer mit 0-Waitstate beim Schreiben

4.3 ARMulator

Der ARMulator ist eine Programmfamilie, die die Befehlssätze der ARM-Prozessoren simuliert. Der „ARM-Instruction-Level Simulator“ („ARMulator“) ermöglicht es, ein Hardwaresystem in Software zu simulieren. Dies schließt die Simulation des Prozessors, der RAM/ROM-Bänke, der Adreßbereiche, der Wartezyklen, der Geschwindigkeit der Peripherie etc. mit ein.

In diesem Unterkapitel werden die Eigenschaften, der Aufbau und die Konfiguration des ARMulators dargestellt.

4.3.1 Eigenschaften des ARMulators

Der ARMulator simuliert die Befehlsausführungen der Programme, und gibt genaue Angaben über Taktzyklen aus. Seine Anwendung findet man häufig auch bei Simulationen für Prototypen.

Der ARMulator besteht aus 4 Elementen [ARMu99].

1. Prozessor-Core Modell: Dieses Modell bedient die Kommunikation mit dem Debugger.
2. Das Memorysystem: Es handelt sich um den Datentransfer zwischen dem ARM-Modell und dem Speicher-Modell. (oder dem MMU-Modell) Dieses Modell kann auch ein Cachemodell beinhalten. Der ARMulator gibt die Information über die Datentransfer entweder zwischen dem Prozessor und Cache oder dem Prozessor und Hauptspeicher aus.
3. Coprocessor Interface: Für den Fall, dass die Koprozessorbefehle ausgeführt werden.
4. OS Interface: Schnittstelle für verschiedene Betriebssysteme (Unix, Linux und Windows etc.)

Wie vorher erwähnt, ist der ARMulator ein flexibles Programm zur Emulation der auf ARM basierenden Systeme. Die jeweiligen Konfigurationen erfolgen durch die

Datei „armul.cnf".

Die Datei „armul.cnf" besteht aus den 6 Bereichen [ARMco98]:

- Header
- OS Modell
- Prozessoren
- Speicher
- Koprozessoren
- Modelle

Zum Beispiel sehen die **Prozessoreinträge** in der armul.cnf folgendermaßen aus:

```
{ ARM7TDM
;; Features
Processor=ARM7TDM
Core=ARM7
ARMulator=BASIC
Architecture=4T
Nexec
LateAborts
Debug

ARM7TDMI:Processor=ARM7TDMI
ARM7TM:Processor=ARM7TM

;; Cached variants
ARM710T:Processor=ARM710T
ARM710T:Memory=ARM710T
}
```

Der Eintrag „ARM7TDM: Processor=ARM7TDMI" deklariert das „Kind" des ARM7-TDMs namens ARM7TDMI. Man kann hier ein neues Prozessormodell z.B. für einen neuen Microcontroller einfügen. Dies würde zusätzlich zu dem oberen Beispiel wie folgt aussehen:

```
ARM7Txx:Processor=ARM7Txx
ARM7Txx:Memory=NeuesSpeicherModellxx
ARM7Txx=ARM7TM
```

Mit der Deklaration von neuen Prozessoren und einem Memorymodell kann man z.B. ein neues Prozessormodell und ein neues Memorymodell simulieren, das das Prozessormodell benutzt.

4.3.2 Cachemodell

Der ARMulator bietet auch die Möglichkeit an, ein Prozessormodell mit verschiedenen Cacheeinstellungen zu simulieren. Der ARMulator unterstützt zwei Cachemodelle, MMUlator und StrongMMU, wobei der MMUlator für die Prozessoren wie ARM600, ARM610, ARM700, ARM710, ARM710a und ARM810 und die StrongMMU für SA-110 angewendet werden [ARMcm98]. Die Konfiguration erfolgt auch in der Datei „armul.cnf". Ein Abschnitt der armul.cnf sieht wie folgt aus:

```
{ MMUlator
{ ARM700

TLBSize=64
RNG=7
WriteBufferAddrs=4
WriteBufferWords=8
CacheReplaceTicks=1
CacheWrite=WriteThrough
HasRFlag
HasUpdateable=FALSE
BufferedSwap=FALSE
Architecture=3
CacheWriteBackInterlock
sNa=Yes
Replacement=Random
Has26BitConfig
HasWriteBuffer
CheckCacheWhenDisabled

ARM710T:CacheWords=4
ARM710T:CacheAssociativity=4
ARM710T:CacheBlocks=64
ARM710T:Architecture=4T
ARM710T:ChipNumber=0x710
ARM710T:Revision=0
ARM710T:ThumbAware=1
ARM710T:ProcessId=0
; Set core/memory clock ratio
MCCFG=2
}

ARM710T=ARM700
}
```

Die Einstellung erfolgt durch die Parameteränderung (die fett geschriebenen Einträge). In der Tabelle 4.1 ist die Beschreibung für die einzelnen Parameter dargestellt. Die oben beschriebene Einstellung bedeutet:

4 (Byte/Word) x 4 (Words/Line) x 4 (Lines/Set) x 64 (sets) = 4096 Byte (Cachegröße)

Durch diese Änderung können verschiedene Cacheorganisationen mit unterschiedlichen Speichergrößen simuliert werden. Im Rahmen dieser Diplomarbeit wurden insgesamt 5 Cacheorganisationen, „Direct-Mapped", 2- bis 8-fach und voll assoziativer Cache mit den Speichergrößen zwischen 64 Byte bis 8192 Byte simuliert. Alle Cacheorganisationen hatten eine Cacheblockgröße von 8 Byte (Cachewords=2).

Tabelle 4.1 Cacheparameter

Tag	Beschreibung
CacheWords	Anzahl der Worte in einer Cacheline
CacheAssociativity	Anzahl der Cachelines in einem Set
CacheBlocks	Anzahl der Sets im Cache

Außerdem kann man die Schreibstrategien wie „Write Through“ oder „Write Back“ auswählen, und auch den „Write Buffer“ einstellen. Dabei lässt sich für den Writebuffer nicht nur die Größe einstellen, sondern er lässt sich auch vollständig aktivieren oder deaktivieren. Allerdings kann der ARMulator nur die beiden Cacheersetzungsstrategien, Zufall und Round Robin, simulieren. Er ist leider nicht in der Lage, die häufig verwendete Cacheersetzungsstrategie „LRU“ (least recently used) zu simulieren.

4.4 Traceanalyzer

Der Traceanalyzer wurde von Schwarz [Sw00] im Rahmen seiner Diplomarbeit implementiert. Der Traceanalyzer analysiert die Traceausgabe des ARMulators und ermittelt die Eigenschaften eines simulierten Programms.

Der ursprüngliche Traceanalyzer war nicht in der Lage, die Cacheeigenschaften zu ermitteln. Außerdem haben die beiden On-Chip-Speicher in Abhängigkeit von der On-Chip-Speichergröße und der Cache-Speicher auch von den Set-Assoziativitäten einen stark unterschiedlichen Energieverbrauch, dessen Unterschied bei der früheren Simulation nicht berücksichtigt wurde. Daher wurde der Traceanalyzer entsprechend ausgeweitet, sodass zwei separate Traceanalyzer für Scratch-Pad- und Cache-Speicher entstanden.

Der Aufruf des Traceanalyzers für Scrach-Pad erfolgt über die Kommandozeile mit folgenden Parametern:

trace_scp *<Tracedatei> <Simulatorausgabe> <Linkerausgabe> <Instruktionsdatei> <Speicherdaten>*

Der Aufruf des Traceanalyzers für Cache:

trace_cache <Tracedatei> <Simulatorausgabe> <Linkerausgabe> <Instruktionsdatei> <Speicherdaten> <Energiedaten>

Der Parameter „Energiedaten“ beinhaltet den unterschiedlichen Energieverbrauch des Caches in Bezug auf die Größe und Set-Assoziativität.

Eine Ausgabe des Traceanalyzers sieht folgendermaßen aus:

```
Memory Size:
     Memoryunit                Prog Data
     offchip                   188  160

  executed Instructions :      533
  access to datamemory  :      1076 Byte
  total offchip access  :      269
     Memoryunit                Inst Data Energy/10^9
     ROM read  4 Byte          0    3    49.3
     offchip read  4 Byte      0    185  49.3
     offchip write 4 Byte      0    81   41.1

     Number of Cacheaccess       343        3.3
     Number of Cacheread         209        3.3
     Number of Cachewrite        134        3.3

CPU-Cycles : 2133
Energy : 23.938/10^6 Ws = 10.191/10^6 Ws (Instruction)
+13.747/10^6 Ws (Memory)
Power : 370.3 mW = 157.7 mW (Instruction) +212.7
mW(Memory)
```

Die Werte aus der linken Spalte im Kästchen geben die Anzahl der Cachezugriffe, die Werte aus der rechten Spalte den Energieverbrauch der jeweiligen Cacheeinstellungen aus.

Außerdem gibt der Traceanalyzer folgende Informationen aus:

- Programm- und Datengröße
- Prozessorzyklen
- Leistung und Energieverbrauch
- Anzahl der ausgeführten Instruktionen
- Anzahl der Speicherzugriffe

Insgesamt wurden bei der Simulationen für die beiden On-Chip-Speicher 9 Benchmarks eingesetzt. Für jede Cacheorganisation und für jede Größe der beiden On-Chip-Speicher wurde der ARMulator neu konfiguriert. Der erweitete Traceanalyzer ist in der Lage, den unterschiedlichen Energieverbrauch pro Zugriff auf einen On-Chip-Speicher in Bezug auf die On-Chip-Speichergröße zu berücksichtigen. Die Simulationsergebnisse, die Ausgabe des Traceanalyzers, und die Auswertung über den Energieverbrauch und die Performance des ARM7-Prozessorsystems mit den beiden On-Chip-Speichern werden im nächsten Kapitel vorgestellt.

Kapitel 5

Ergebnisse und Auswertung

Im Rahmen dieser Diplomarbeit wurden die verschiedenen Cacheorganisationen und der Scratch-Pad-Speicher mit unterschiedlichen Größen in Bezug auf Energieverbrauch und Performance untersucht. Zuerst wurden verschiedene Cacheorganisationen untersucht, um herauszufinden, welche der Cache-organisationen energiemäßig günstiger ist und welche einen Performancevorteil bringt. Dann wurden weitere Merkmale, z.B. Zugriffe auf den Cache und Cachemisses untersucht, um die Gründe für unterschiedliche Verläufe der Simulationsergebnisse der Cacheorganisationen in Bezug auf die Energie und Performance zu verstehen. Für den Scratch-Pad-Speicher wurde aus dem gleichen Grund neben der Energie und Performance auch die Anzahl der Scratch-Pad-Zugriffe betrachtet.

Zum Vergleichen der beiden On-Chip-Speicher wurden insgesamt 9 Benchmarks eingesetzt. Diese Benchmarks sind vor allem Sortierprogramme und Filter-applikationen. In diesem Kapitel werden die Simulationsergebnisse vorgestellt und die durch Simulation gewonnenen Informationen näher erläutert.

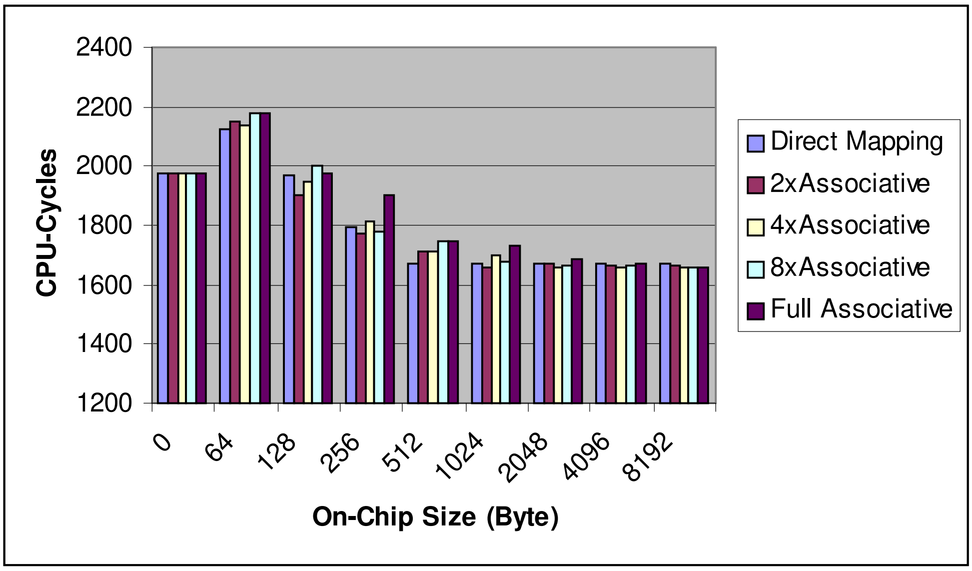
5.1 Prozessorzyklen

Wir wissen, dass die Energieoptimierungen nicht unbedingt auch zu einer besseren Performance führen. Ein Beispiel dafür ist das Registerpipelining [SWM00]. Durch die zusätzlich eingebauten Optimierungen wird der Programmcode größer und langsamer. Die beiden Optimierungen, Energie- und Performanceoptimierung, haben andererseits ein gemeinsames Streben. Dies ist die Reduzierung der Off-Chip-Speicherzugriffe. Die Reduzierung der Off-Chip-Speicherzugriffe kann sowohl zu einer Reduzierung des Energieverbrauchs als auch zu einer Verbesserung der Performance führen. Die Zugriffe auf den Hauptspeicher sind energiemäßig teuer und bedingt durch die zusätzlichen Wartezyklen langsamer. Dieses Problem kann durch den Einsatz der beiden On-Chip-Speicher gelöst werden.

Wir betrachten zuerst die Prozessorzyklen, die die eingesetzten Benchmarks für ihren Ablauf benötigen.

5.1.1 CPU-Cycles der verschiedenen Cacheorganisationen

Im Rahmen dieser Diplomarbeit wurden insgesamt 5 Cacheorganisationen (Direct-Mapped, 2- bis 8-fach assoziativer und voll assoziativer Cache) simuliert, um die Performance und den Energieverbrauch der verschiedenen Cacheorganisationen zu untersuchen.



CacheSize(Byte)	0	64	128	256	512	1024	2048	4096	8192
Direct Mapping	1977	2125	1967	1791	1670	1670	1670	1670	1670
2xAssociative	1977	2153	1904	1774	1713	1661	1670	1665	1665
4xAssociative	1977	2136	1948	1816	1711	1701	1656	1656	1656
8xAssociative	1977	2178	2001	1781	1748	1678	1665	1665	1656
Full Associative	1977	2178	1977	1904	1749	1730	1687	1670	1656

Abbildung 5.1 Prozessorzyklen bei biquad_N_section

Bei dem DSP-Bechmark „biquad_N_section" steigen zuerst die Prozessorzyklen bei allen Cacheorganisation. Bei der Cachegröße von 128 Byte benötigt ein Prozessor mit einem 2-fach assoziativen Cache erst weniger Zyklen als ein Prozessor ohne Cache. Die restlichen Cacheorganisationen benötigen erst ab der Cachegröße von 256 Byte weniger Prozessorzyklen. Dies ist darauf zurückzuführen, dass der 2- fach assoziative Cache bei den Cachegrößen von 128 und 256 Byte die wenigsten Cachemisses hat (*Abbildung 5.2*).

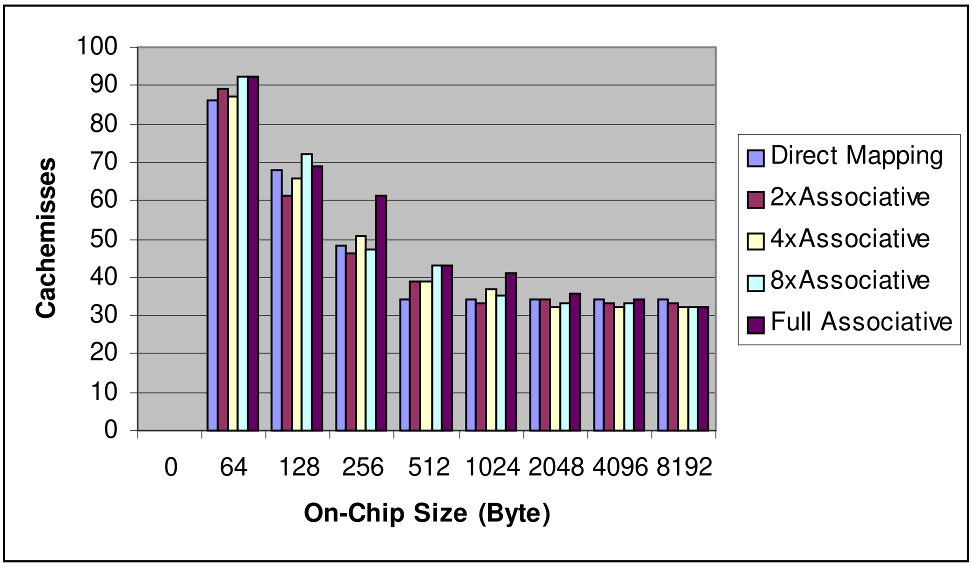


Abbildung 5.2 Anzahl der Cachemisses bei biquad_N_section

Bei jedem Cachemiss entsteht ein sogenanntes „Memory Idle“, bei dem sich der Hauptspeicher im Leerlaufzustand befindet und weitere Wartezyklen bedingt durch den Zugriff auf den Hauptspeicher eingefügt werden. Die Cachegröße von 64 Byte bzw. 128 Byte ist viel zu klein, sodass das zu häufige Ersetzen der Cacheinhalte zu negativen Auswirkung auf die Performance führt. Ab der Cachegröße 512 Byte ändern sich die Prozessorzyklen nicht mehr viel. Die meisten Daten und Befehlen liegen im Cache. Die noch übrig gebliebenen Cachemisses sind allein dadurch entstanden, dass der Cache zuerst leer war, und dann gefüllt werden musste.

Tabelle 5.1 Cachemissrate des 2-fach assoziativen Caches

CacheSize (Byte)	64	128	256	512	1024	2048	4096	8192
biquad_N_sections	16,2%	11,3%	8,6%	7,3%	6,2%	6,3%	6,2%	6,2%
bubble_sort	30,7%	8,3%	3,6%	0,3%	0,1%	0,1%	0,1%	0,1%
heap_sort	41,5%	24,9%	9,9%	1,7%	0,8%	0,7%	0,6%	0,6%
insertion_sort	19,7%	9,6%	3,5%	0,7%	0,4%	0,3%	0,3%	0,3%
lattice	31,2%	16,2%	7,3%	4,8%	2,7%	2,3%	2,2%	2,1%
matrix-mult	34,0%	17,3%	8,5%	5,4%	4,9%	4,7%	4,7%	4,7%
me_ivlin	12,0%	5,4%	2,6%	0,2%	0,1%	0,0%	0,0%	0,0%
quick_sort	40,7%	32,4%	14,1%	5,4%	4,0%	2,6%	2,6%	2,6%
selection_sort	12,6%	6,1%	3,4%	0,3%	0,2%	0,1%	0,1%	0,1%

Bei der Mediaapplikation „me-ivlin“ sieht der Verlauf der Prozessorzyklen etwas anders aus. Bei der Cachegröße von 64 Byte gibt es keine hohe Spitze wie beim biquad_N_section. Die Prozessorzyklen verringern sich langsamer aber deutlich, sobald der Prozessor einen Cache hat (*Abbildung 5.3*). Einen solchen Verlauf gab es noch einmal bei dem Benchmark „selection sort". Das liegt daran, dass die beiden Benchmarks eine niedrigere Cachemissrate als die restlichen Benchmarks besitzen. Das führt dazu, dass die Cachemisses, die bei der kleinen Cachegröße für die hohen Prozessorzyklen verantwortlich sind, auf die Gesamtzahl der Prozessorzyklen der beiden Benchmarks weniger Einfluss haben.

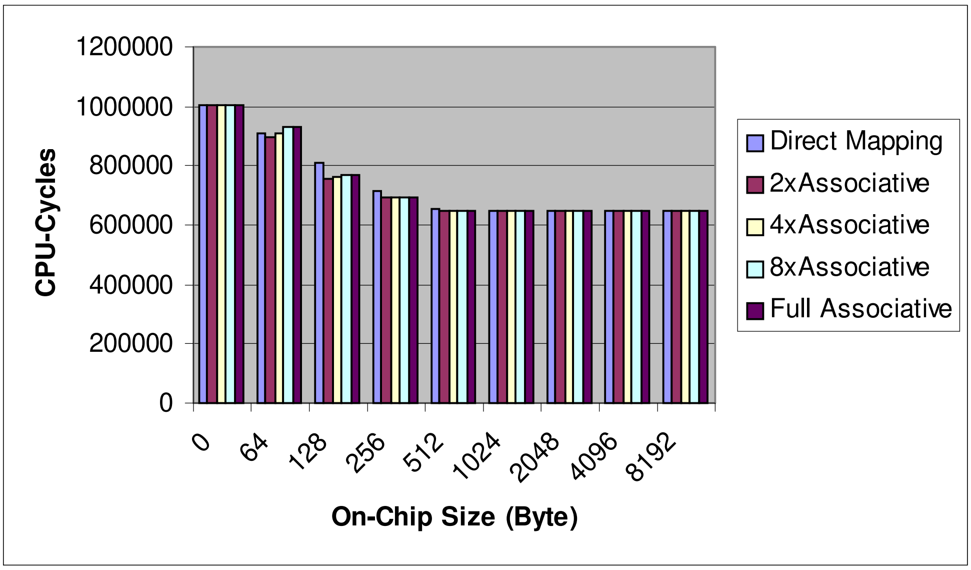


Abbildung 5.3 Prozessorzyklen bei „me-ivlin"

Bei dem Benchmark „me-ivlin" ist der 2-fach assoziative Cache für die Cachegrößen von 64 und 128 Byte am schnellsten und bei der Größe von 256 Byte am zweitschnellsten. Bei der Betrachtung der Cachegrößen zwischen 64 und 512 Byte (da in den größeren Cache alle Befehle und Daten passen würden) sind zwei bzw. 4-fach assoziative Caches bei den meisten eingesetzten Benchmarks schneller als andere Cacheorganisationen.

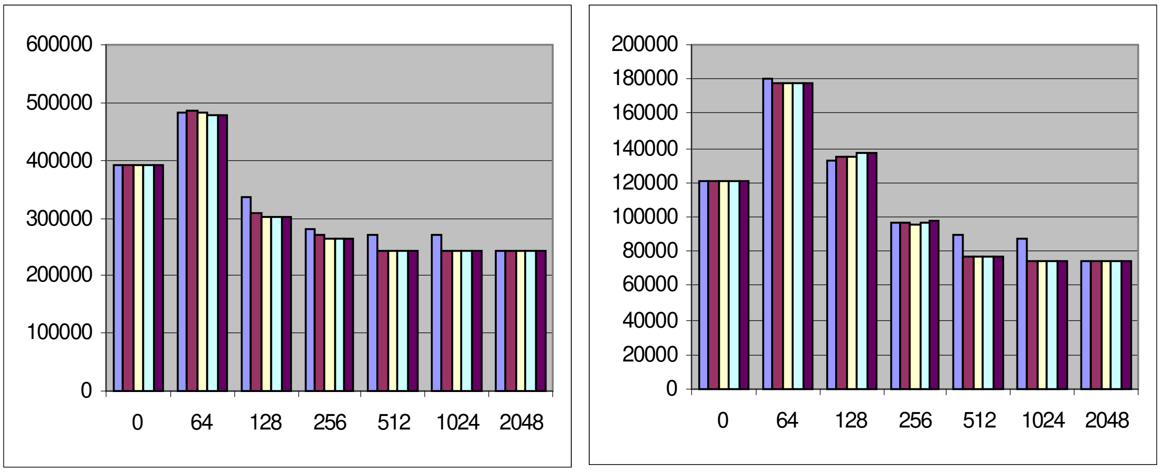


Abbildung 5.4 Prozessorzyklen bei bubble sort (l) und heap sort (r)

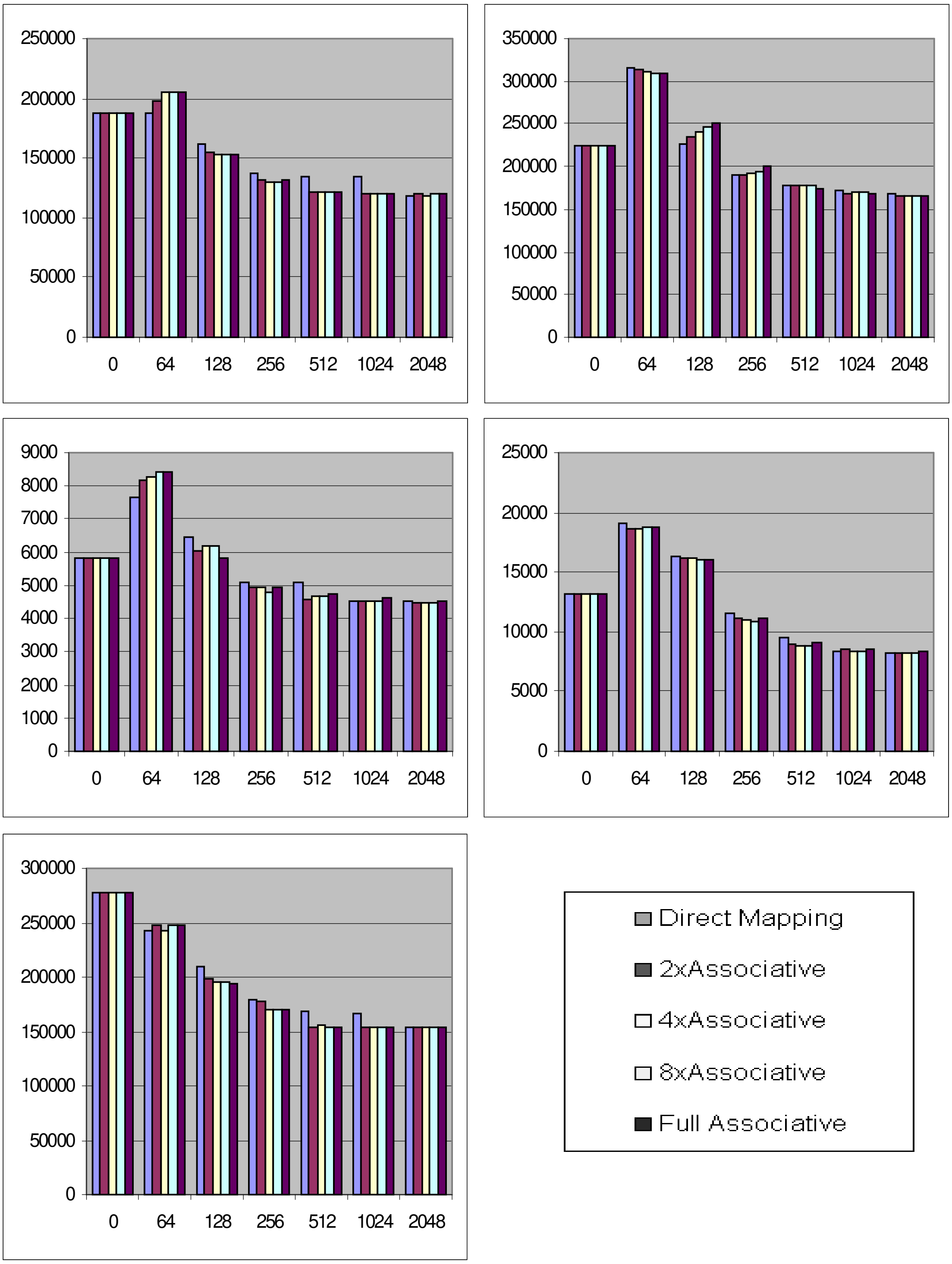


Abbildung 5.5 *Prozessorzyklen bei insertion sort (Oben links), lattice (Oben rechts), matrix_mult (Mitte links), quick sort (Mitte rechts) und selection sort (Unten links)*

Auffallend ist, dass die Anzahl der Prozessorzyklen des ARM7-Prozessors mit dem „Direct-Mapped Cache“ in dem Bereich zwischen 512 und 1024 Byte bei allen Sortierprogrammen immer noch viel höher als bei den anderen Cacheorganisationen ist. Dies ist darauf zurückzuführen, dass die Anzahl der Cachemisses bei den Sortierprogrammen bis zu 24 fach (bei „bubble sort“ mit der Cachegröße von 1024 Byte) höher als bei den anderen Benchmarks liegt.

Die Simulationsergebnisse zeigen, dass der „Direct-Mapped Cache“ in der Regel eine höhere Cachemissrate hat, und mehr als 4-fach Assoziativität wegen der aufwendigen Datenverwaltung innerhalb des Caches (z.B. nach einem Datum müssen noch mehr Cacheblöcke durchsucht werden.) negative Auswirkung auf die Performance hat. In den *Abbildungen 5.4* und *5.5* sind die Prozessorzyklen der eingesetzten Benchmarks bis zur Cachegröße von 2048 Byte dargestellt, weil größere Caches als 2048 Byte keinen Einfluss mehr auf die Performance gezeigt haben.

5.1.2 Performancevergleich von Cache und Scratch-Pad

In diesem Unterkapitel werden die Vergleichsergebnisse der Prozessorzyklen von Cache- und Scratch-Pad-Speichern vorgestellt. Verglichen wurden ein 2-fach assoziativer Cache und ein Scratch-Pad mit den Speichergrößen von 64 bis 8192 Byte. Für den Scratch-Pad-Speicher wurde die Energieoptimierung durchgeführt.

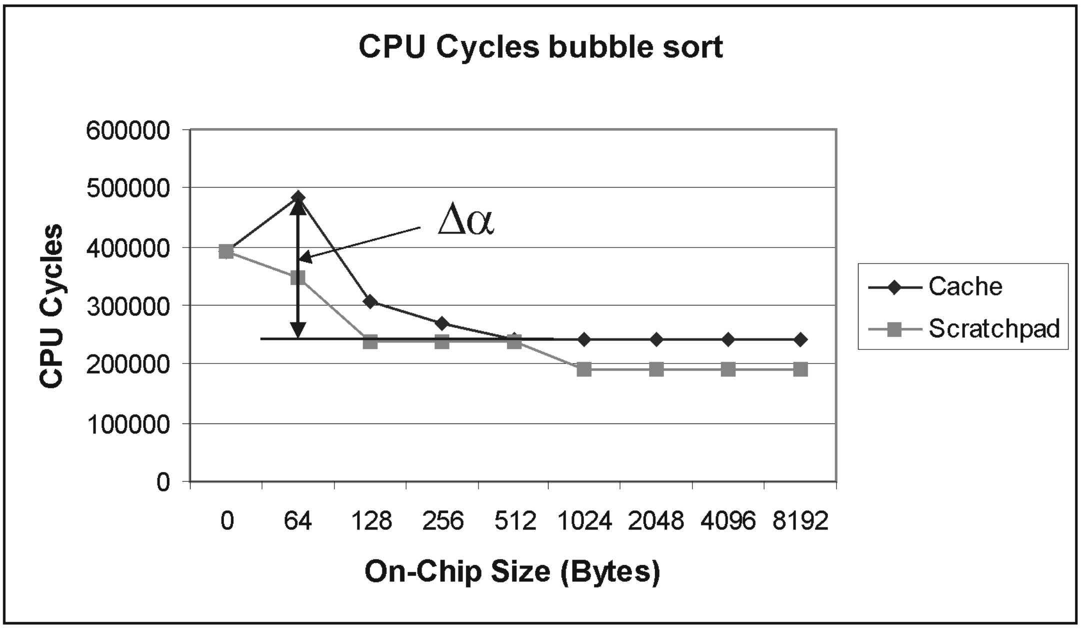


Abbildung 5.6 Prozessorzyklen bei bubble sort

Wir schauen uns zuerst den Vergleich bei „bubble sort“ an. Bei der On-Chip-Größe von 64 Byte ist der Scratch-Pad 27,9 %, bei 512 Byte 11,7 % und ab 512 Byte, bei der alle Daten und Programmteile im On-Chip sind, 20,5 % schneller als der Cache. Bei den kleinen On-Chip-Größen ist der Cache wegen der hohen Cachemissrate besonders langsam. Das kann man bei dem Vergleich von On-Chip-Zugriffen gut erkennen (*Abbildung 5.7*). Die Prozessorzyklen des Scratch-Pad-Speichers verringern sich ständig bis alle Daten und Programmteile im On-Chip liegen. Ab der Cachegröße von 2048 liegen alle Daten und Befehle wie beim Scratch-Pad im Cache (On-Chip-Speicherbelegung des Programms : 192 Byte, der Daten : 408 Byte). Der Speicherplatzbedarf der restlichen Benchmarks ist im Anhang (B. Simulationsergebnisse) aufgeführt. Das $\Delta\alpha$ gibt die zusätzlichen Prozessorzyklen an, die bedingt durch die Cachemisses außer den Cachemisses bei der Initialisierung (Beim Programmstart ist ein Cache immer leer!) entstehen.

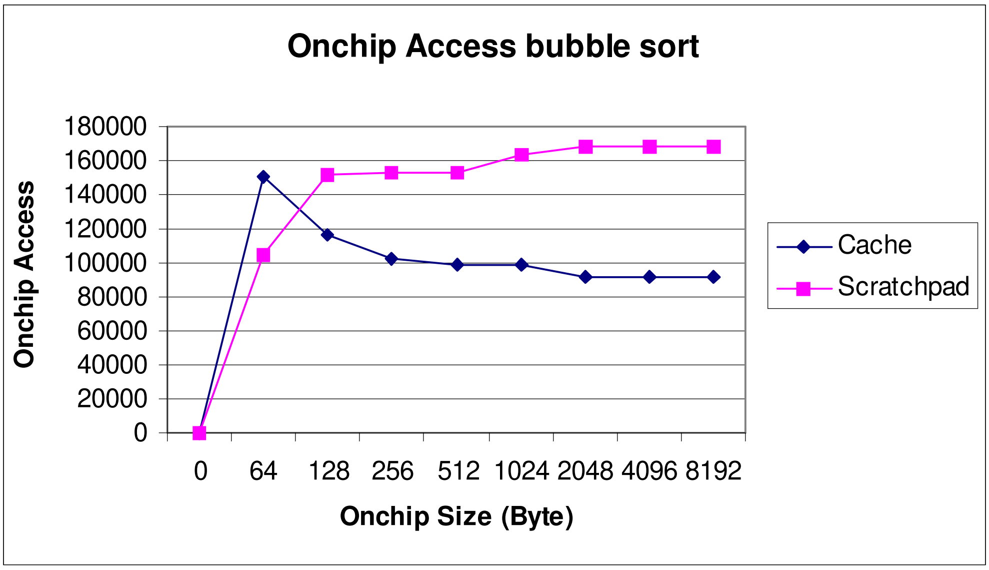


Abbildung 5.7 On-Chip-Zugriffe bei bubble sort

Die Anzahl der Zugriffe auf den Cache steigt bei der Größe von 64 Byte sprunghaft an, und nimmt danach langsam ab. Die Gründe dafür sind die fallende Anzahl der Cachemisses und die damit verbundenen zusätzlichen Cachezugriffe mit der steigenden Cachegröße bis zur Cachegröße von 2048 Byte. Zum Beispiel kostet ein Readmiss bei einem Cache, in dessen Cacheblock mit 2 Worten, 9 Zyklen und insgesamt 3 Cachezugriffe (Kapitel 3.3.2).

Die Prozessorzyklen bzw. Zugriffe auf den Scratch-Pad-Speicher haben sich in dem Bereich zwischen 128 bis 512 Byte kaum geändert. Hier wurden 200 Byte von Daten und Programmteilen in den Scratch-Pad-Speicher verschoben. Die Anzahl der Zugriffe auf den Scratch-Pad-Speicher bleiben gleich. Für die Größe des Scratch-Pad-Speichers von 256 und 512 Byte hat der Scratch-Pad-Algorithmus keine weiteren Basisblöcke für den noch vorhandenen Speicherplatz gefunden.

Wenn bei einem Programmablauf ziemlich oft abwechselnd auf den Scratch-Pad- und den Hauptspeicher zugegriffen wird, kann sich die Benutzung eines Scratch-Pad-Speichers auf die Performance negativ auswirken, weil jeder zusätzliche Sprungbefehl für den Sprung vom Scratch-Pad-Speicher auf den Hauptspeicher und umgekehrt mehrere Zyklen braucht (Zyklen für die Ausführung des Befehls, das Füllen der Pipeline und die Wartezyklen beim Hauptspeicherzugriff).

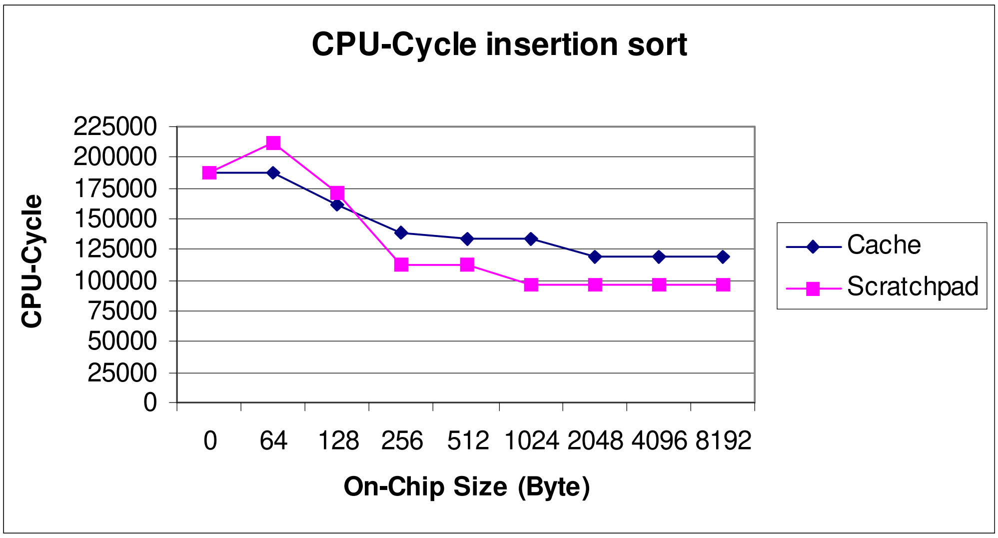


Abbildung 5.8 Prozessorzyklen bei insertion sort

Bei dem Benchmark „insertion sort" kann man genau diese negative Auswirkung erkennen. Bei der On-Chip-Größe von 64 Byte benötigt der Scratch-Pad-Speicher mehr Prozessorzyklen als der Cache-Speicher. Bei allen anderen Benchmarks tritt diese sprungartige Zunahme der Prozessorzyklen nicht auf. Die Prozessorzyklen des Scratch-Pad-Speichers verringern sich schon bei der Speichergröße von 64 Byte.

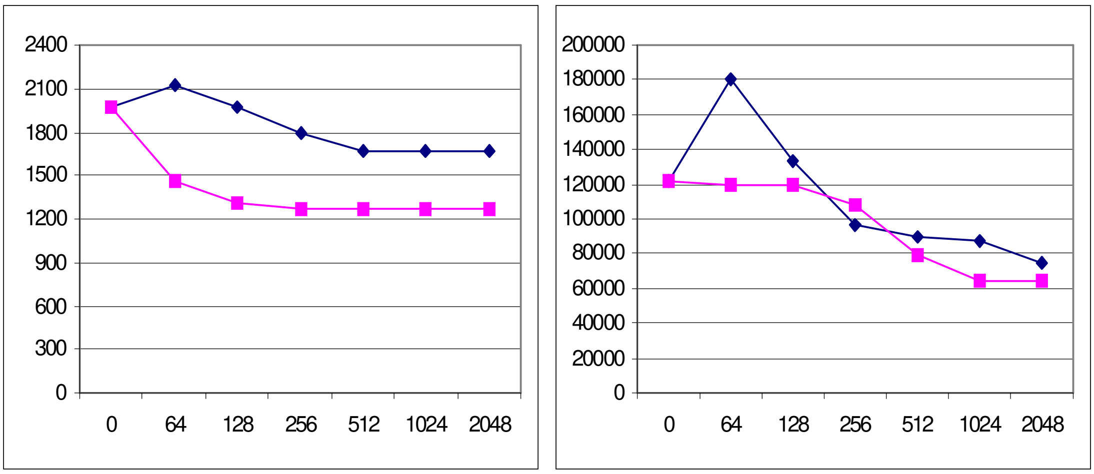


Abbildung 5.9 Prozessorzyklen bei biquad_N_section (l) und heap sort (r)

Während der Einsatz des Scratch-Pad-Speichers bei dem „biquad_N_section" schon bei der Speichergröße 64 Byte einen richtigen Performancevorteil (25,8%) gegenüber dem Prozessor ohne On-Chip-Speicher hatte, konnte der Scratch-Pad bei dem „heap sort" erst ab der Größe von 512 Byte Performanceverbesserungen bringen. So war der Prozessor mit dem Cache-Speicher bei der On-Chip-Größe von

256 Byte sogar schneller als der mit dem Scratch-Pad-Speicher.

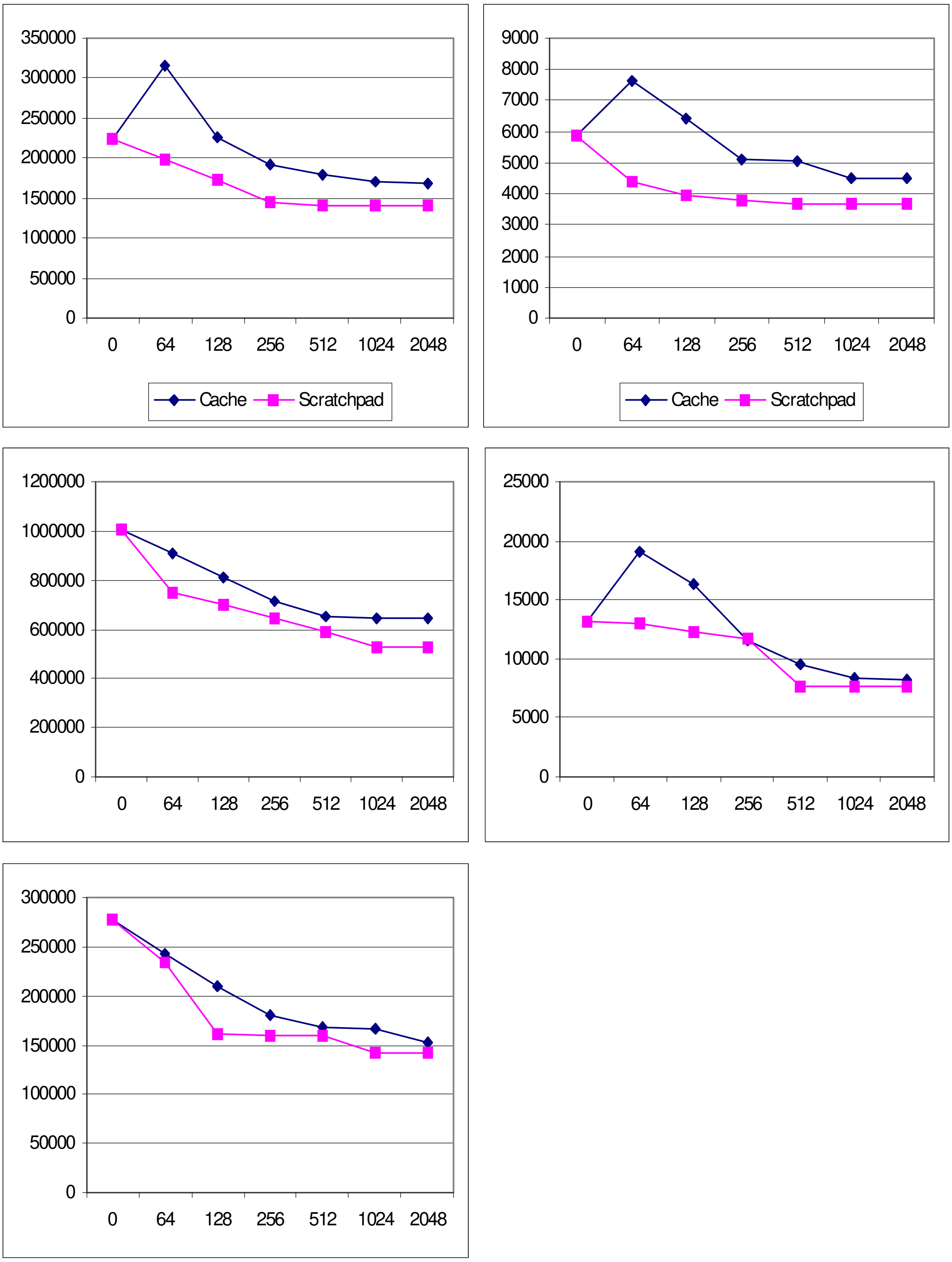


Abbildung 5.10 *Prozessorzyklen bei lattice (Oben links), matrix_mult (Oben rechts), me-ivlin (Mitte links), quick sort (Mitte rechts) und selection sort (Unten)*

Der Einsatz des Scratch-Pad-Speichers bringt trotz der zusätzlichen Sprungbefehle bei den eingesetzten Benchmarks bis auf den „heap sort“ mit der Speichergröße von 265 Byte eindeutige Performancevorteile. Vor allem bei den relativ kleinen On-Chip-Größen, bei denen nur ein Teil von den Befehlen und den Daten in den Scratch-Pad- bzw. in den Cache-Speicher geladen werden kann, hat der Einsatz eines Scratch-Pad-Speichers einen großen Performancevorteil gebracht.

5.2 Energieverbrauch

Bisher haben wir beobachtet, dass der Einsatz eines Scratch-Pad-Speichers trotz seines statischen Algorithmus und der zusätzlichen Sprungbefehle eine ziemlich gute Performance gegenüber der des Caches bringen kann. Das eigentliche Ziel der eingesetzten Optimierung für den Scratch-Pad-Speicher ist die Reduzierung des Energieverbrauchs. Bei den EIS ist die Reduzierung des Energieverbrauchs besonders wichtig. Da der Einsatz von On-Chip-Speichern für die Reduzierung des Energieverbrauchs eine Lösung sein kann, wurde der Energieverbrauch des ARM-Prozessorsystems mit einem Cache-Speicher und mit einem Scratch-Pad-Speicher untersucht.

In diesem Kapitel werden die Simulationsergebnisse in Bezug auf den Energieverbrauch der verschiedenen Cacheorganisationen und des Scratch-Pad-Speichers vorgestellt.

5.2.1 Energieverbrauch der Cacheorganisationen

Cache-Speicher mit unterschiedlichen Cacheorganisationen finden heute einen breiten Einsatz im Bereich eingebetteter Systeme. Vor dem Vergleich mit dem Scratch-Pad-Speicher sollte deshalb untersucht werden, welche der verschiedenen Cacheorganisationen energiemäßig günstiger sind.

Zuerst betrachten wir den Energieverbrauch der verschiedenen Cacheorganisationen. Wie bei den Untersuchungen für die Prozessorzyklen wurden insgesamt 5 Cacheorganisationen (Direct-Mapped, 2x, 4x, 8x und voll assoziativer Cache mit der Cachegröße von 64 bis 8192 Byte) simuliert und ihr Energieverbrauch untersucht.

Diese Untersuchungen dienen zuerst dazu, eine energieverbrauchsarme Cache-organisation unter Berücksichtigung der Cachemissrate zu finden. Die Verläufe der Energieverbräuche wurden analysiert, um die Ursache für den Energieverbrauch zu finden.

Für die Untersuchungen wurden auch die gleichen Benchmarks wie bei den Untersuchungen für die Prozessorzyklen eingesetzt. Die Energieverbräuche des Prozessors, des Cache-Speichers und des Hauptspeichers wurden gesondert berechnet, um genau herauszufinden, welche Komponente die Hauptursache des

Energieverbrauchs ist und den gegenseitigen Einfluss innerhalb des ARM7-Prozessorsystems zu analysieren.

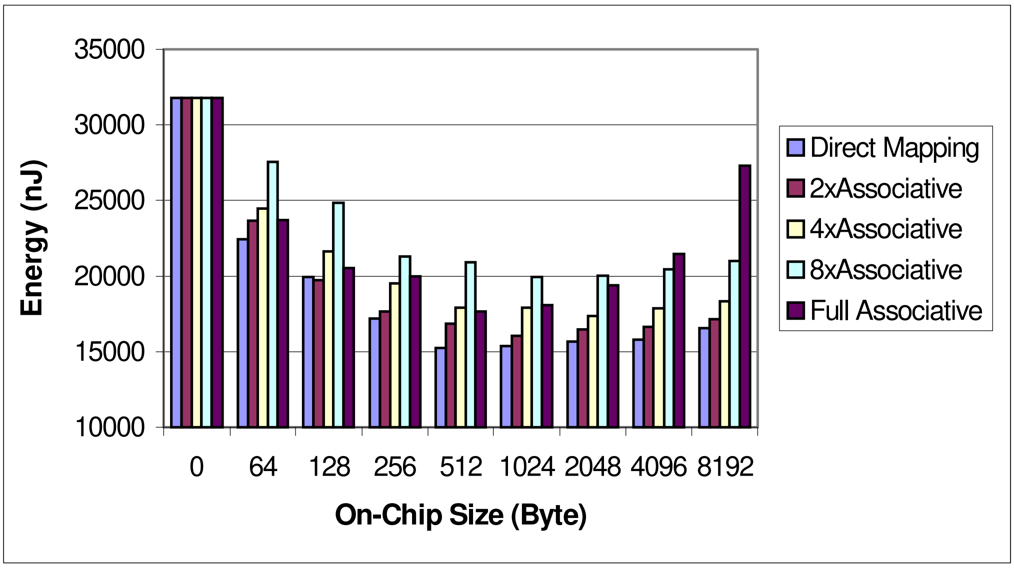


Abbildung 5.11 *Energieverbrauch bei biquad_N_section*

In der *Abbildung 5.11* sind die Energieverbräuche des ARM7-Prozessorsystems mit 5 Cacheorganisationen bei dem Benchmark „biquad_N_section" dargestellt. Sie zeigt, dass der Energieverbrauch beim „Direct-Mapped Cache" am niedrigsten ist. Diese Cacheorganisation hatte allerdings nicht die geringste Anzahl der Prozessorzyklen (*Abbildung 5.1*) und der Cachemisses (*Abbildung 5.2*). Außerdem wurde keine Optimierung für den Cache-Speicher eingesetzt. Dass diese Cacheorganisation trotzdem den niedrigsten Energieverbrauch hat, liegt daran, dass der Energieverbrauch pro Zugriff auf diese Cacheorganisation viel niedriger als bei den restlichen Cacheorganisationen ist, und der Anteil der Cachezugriffe in den gesamten Prozessorzyklen über 30% liegt. In der *Tabelle 5.2* (aus Kapitel 3.3.2) ist noch mal der Energieverbrauch der 5 Cacheorganisation dargestellt. Der Energieverbrauch des „Direct-Mapped Caches" liegt von 40,8 % bis zu 84,5 % niedriger als der anderer Cacheorganisationen.

Tabelle 5.2 *Energieverbrauch der Cacheorganisationen*

Speichergröße	DA	2X	4X	8X	FA
64 Byte	0,71 nJ	1,63 nJ	2,71 nJ	5,52 nJ	1,20 nJ
128 Byte	0,76 nJ	1,79 nJ	3,05 nJ	5,87 nJ	1,36 nJ
256 Byte	0,86 nJ	1,90 nJ	3,32 nJ	6,24 nJ	2,03 nJ
512 Byte	0,98 nJ	2,05 nJ	3,48 nJ	6,63 nJ	2,47 nJ
1024 Byte	1,15 nJ	2,23 nJ	3,75 nJ	6,92 nJ	3,33 nJ
2048 Byte	1,47 nJ	2,55 nJ	4,04 nJ	7,37 nJ	6,00 nJ
4096 Byte	1,69 nJ	2,88 nJ	4,71 nJ	7,95 nJ	9,18 nJ
8192 Byte	2,67 nJ	3,57 nJ	5,39 nJ	8,89 nJ	17,24 nJ

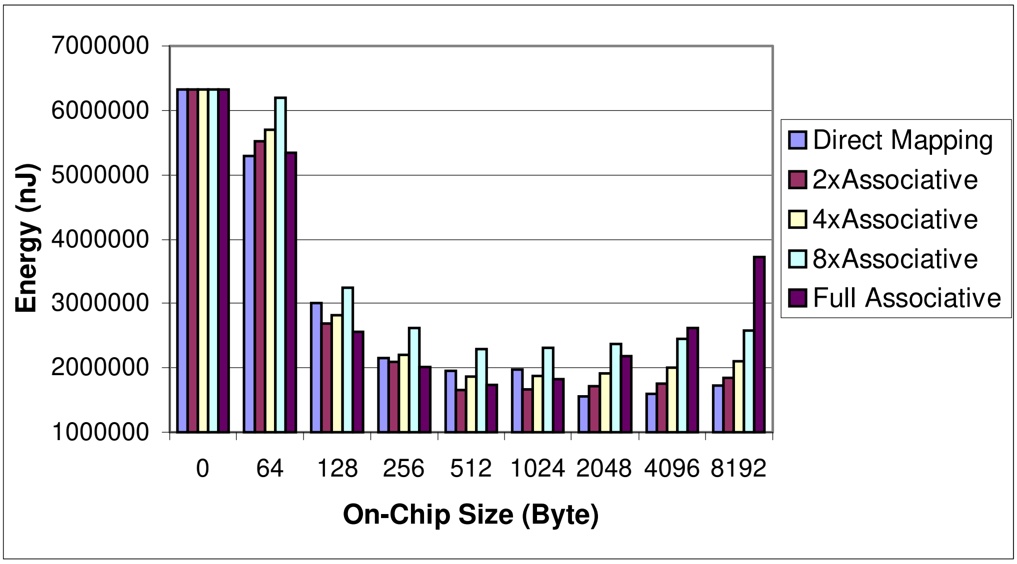


Abbildung 5.12 Energieverbrauch bei bubble sort

Bei den Sortierprogrammen ist der "Direct-Mapped Cache" nicht mehr die energiesparsamste Variante. In der *Abbildung 5.12* ist der Energieverbrauch bei „bubble sort" dargestellt. Bei der Cachegröße von 64 Byte zeigen ARM7-Prozessorsysteme mit dem „Direct-Mapped Cache" und dem voll assoziativen Cache aufgrund des geringeren Energieverbrauchs der beiden Cacheorganisationen (*Tabelle 5.2*) einen relativ niedrigen Gesamtenergieverbrauch. Der Energieverbrauch der 4-, 8-fach und voll assoziativen Cacheorganisation nehmen mit der Cachegröße aufgrund der aufwendigen Datenverwaltung im Cache rapide zu, so dass der Gesamtenergieverbrauch vor allem mit der größeren Cachegröße sehr negativ verläuft. Allerdings liegen ab der Cachegröße von 512 Byte alle Daten und Befehle im Cache. Das bedeutet, dass die noch größeren Caches nicht nur mehr Chipfläche benötigen, sondern auch keinen energiemäßigen Vorteil mit sich bringen.

Obwohl ein 2-fach assoziativer Cache selbst pro Zugriff mehr Energie verbraucht als ein „Direct-Mapped Cache", ist der Gesamtenergieverbrauch mit dem 2-fach assoziativen Cache niedriger. Die Ursache dafür ist die sehr hohe Anzahl von Cachemisses beim „Direct-Mapped Cache" (*Tabelle 5.3*). Die zusätzlichen Zugriffe auf den Hauptspeicher durch die Cachemisses haben mehr Einfluss auf den Gesamtenergieverbrauch als der niedrige Eigenenergieverbrauch des „Direct-Mapped Caches".

Tabelle 5.3 Cachemisses bei bubble sort

CacheSize (Byte)	64	128	256	512	1024	2048	4096	8192
Direct Mapping	27533	11083	4837	3357	3311	112	112	112
2xAssociative	28074	7720	3367	241	136	133	111	111
4xAssociative	27615	7199	2737	256	143	118	119	111
8xAssociative	27342	7155	2695	236	142	130	116	114
Full Associative	27342	7270	2695	335	183	136	131	121

Abbildung 5.13 *Energieverbrauch in nJ bei heap sort (Oben links), insertion sort (Oben rechts), lattice (Mitte links), matrix_mult (Mitte rechts), me-ivlin (Unten links) und quick sort (Unten rechts)*

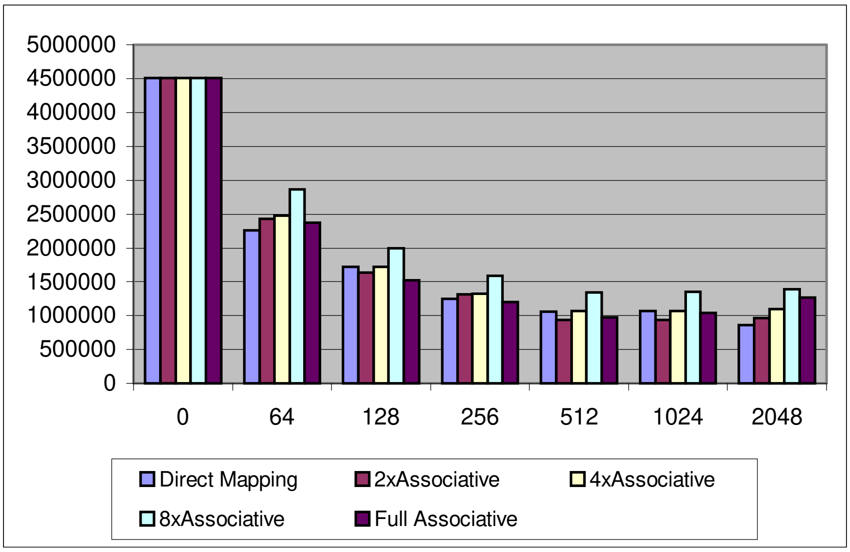


Abbildung 5.14 Energieverbrauch in nJ bei selection sort

In den *Abbildungen 5.13* und *5.14* sind die restlichen Simulationsergebnisse (Energieverbrauch der restlichen Benchmarks) dargestellt. Bei den Benchmarks „biquad_N_section", „insertion sort", „me-ivlin" und „selection sort" wurde der Energieverbrauch durch den Einsatz des Caches selbst bei der kleinen Cachegröße von 64 Byte sofort geringer. Das liegt daran, dass die vier Benchmarks niedrigere Cachemissraten im Vergleich zu den restlichen Benchmarks besitzen (*Tabelle 5.1*). Das hat dazu geführt, dass der Einfluss der Cachemisses auf den Gesamtenergieverbrauch bei den vier Benchmarks geringer wurde.

Es hat sich insgesamt bei den eingesetzten Benchmarks gezeigt, dass die Cacheorganisationen, der „Direct-Mapped Cache und der 2-fach assoziative Cache, einen niedrigeren Energieverbrauch als die anderen Cacheorganisationen haben. Die Gründe dafür sind, dass der „Direct-Mapped Cache" den geringsten Eigenenergieverbrauch hat und der 2-fach assoziative Cache den geringen Eigenenergieverbrauch und wenige Cachemisses hat. Die Cacheorganisationen mit mehr als 2-fach Assoziativität produzieren zwar wenigere Cachemisses als der 2-fach assoziative Cache, verbrauchen aber selbst mehr Energie (*Tabelle 5.2*).

5.2.2 Vergleich des Energieverbrauchs von Cache und Scratch-Pad

Der Gesamtenergieverbrauch eines ARM7-Prozessorsystems ergibt sich aus der Summe der Prozessorkosten, der On-Chip-Speicherkosten und der Off-Chip-Speicherkosten. Dabei spielen die On-Chip- und Off-Chip-Speicherkosten für den Energieverbrauch des Gesamtsystems eine entscheidende Rolle. Deshalb ist es wichtig, zu beobachten, wieviel Off-Chip-Zugriffe durch die On-Chip-Zugriffe ersetzt werden können. Wir wissen auch aus den Simulationsergebnissen (*Tabelle*

3.1 und Tabelle 3.3) des „CACTI-Modells“, dass der Energieverbrauch pro Zugriff auf den Scratch-Pad-Speicher viel niedriger als der des Cache-Speichers ist. Es musste untersucht werden, wie positiv sich dieser Vorteil des Scratch-Pad-Speichers gegen den Cache-Speicher auswirken kann.

Als ein Lösungsansatz für die Energiereduktion in dem Bereich EIS wurden im Rahmen dieser Arbeit der Energieverbrauch des Cache- und des Scratch-Pad-Speichers untersucht. Da ein dynamischer Algorithmus für den Scratch-Pad-Speicher für die Untersuchung noch nicht zur Verfügung stand, wurde für die Simulation nur der statische Algorithmus von Zobiegala [Zob01] eingesetzt.

Der Energieverbrauch des Scratch-Pad-Speichers wurde mit dem Energieverbrauch des ARM7-Prozessorsystems mit dem energiemäßig günstigen 2-fach assoziativen Cache verglichen.

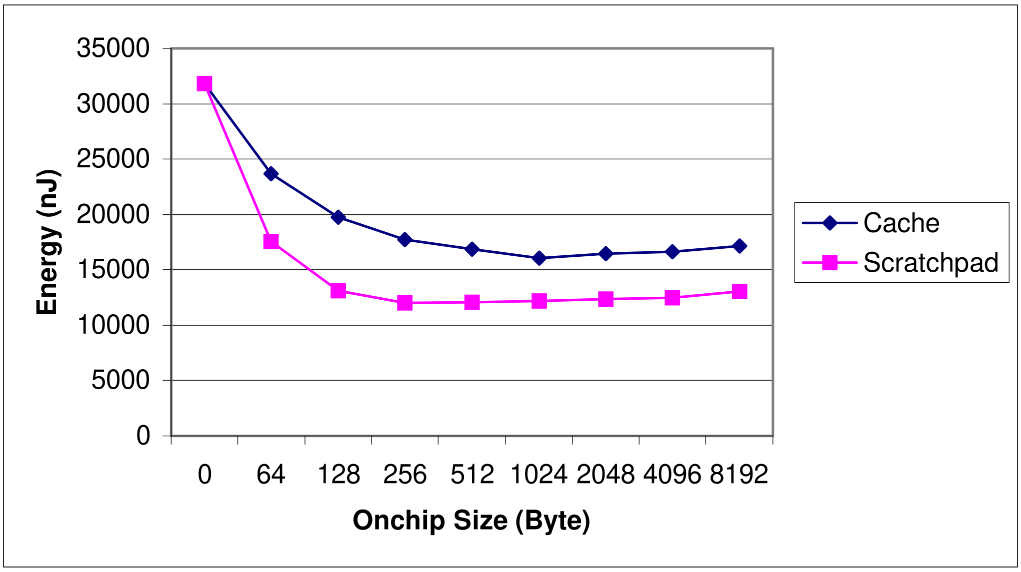


Abbildung 5.15 *Energieverbrauch bei biquad_N_section*

In der *Abbildung 5.15* ist der Energieverbrauch bei „biquad_N_section“ zu sehen. Bei dem Benchmark ist die Energieeinsparung durch den Einsatz des Scratch-Pad-Speichers sehr eindeutig. Mit dem Scratch-Pad-Speicher wurde bis zu 47 % (bei der On-Chip-Speichergröße von 256 Byte) weniger Energie als mit dem Cache-Speicher verbraucht. Die leichte Steigerung des Energieverbrauchs ab der On-Chip-Speichergröße von 2048 Byte liegt daran, dass der Energieverbrauch pro Zugriff auf den Cache- und den Scratch-Pad-Speicher mit der On-Chip-Speichergröße zunimmt.

Die On-Chip-Speichergröße von 2048 Byte ist für alle eingesetzten Benchmarks ausreichend groß. Daher werden weitere Ergebnisse nur bis zu der Speichergröße von 2048 Byte betrachtet.

In der *Abbildung 5.16* sind die Energieverbräuche mit den 3 Energiekosten (Prozessorkosten, On- und Off-Chip-Kosten) dargestellt.

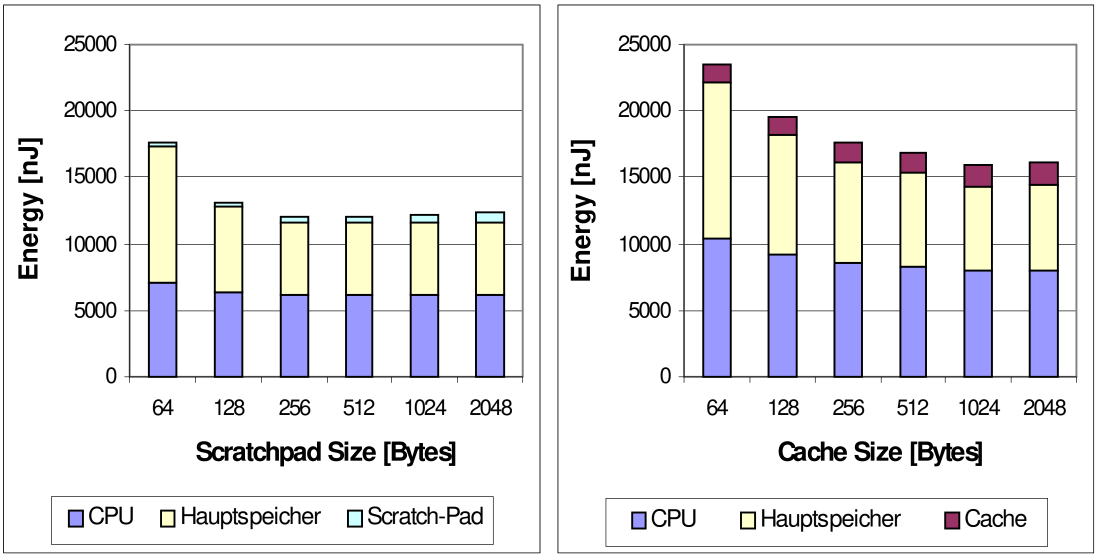


Abbildung 5.16 Energieverbrauch mit 3 Kostenarten bei biquad_N_section
(*Diese Legendenbezeichnungen und Teilungen sind auch für die weiteren Abbildungen dieser Art gleich. Daher werden diese Legendenbezeichnungen für die weiteren Abbildungen nicht nochmals aufgeführt*.)

In der *Abbildung 5.16* ist gut zu erkennen, dass das Prozessorsystem mit dem Scratch-Pad-Speicher in allen 3 Bereichen einen geringeren Energieverbrauch als das System mit dem Cache-Speicher aufweist. Das System mit dem Cachespeicher verbraucht bei der On-Chip-Speichergröße von 128 Byte 31%, 28,4% und 75,4% mehr Energie im Prozessor, dem Hauptspeicher und dem On-Chip-Speicher als das System mit Scratch-Pad-Speicher.

Dies zeigt uns, dass der Energieverbrauch sehr stark reduziert werden kann, wenn man die Daten und Programmteile in Bezug auf ihre Zugriffshäufigkeit und ihre Größe vorher kennt, analysiert und in den Scratch-Pad-Speicher verschiebt.

Dass ein Einsatz eines Scratch-Pad-Speichers nicht immer energiemäßig günstiger als ein Einsatz eines Cache-Speichers ist, zeigt uns die *Abbildung 5.17*. In der *Abbildung 5.17* ist der Energieverbrauch bei „insertion sort“ dargestellt. Bei den On-Chip-Speichergrößen von 64 Byte und 128 Byte, bei denen normalerweise ein Cache-Speicher wegen einer hohen Anzahl von Cachemisses nicht effektiv eingesetzt werden konnte, sind die Energieverbräuche des Prozessorsystems mit dem Scratch-Pad-Speicher deutlich höher als der des Systems mit Cache-Speicher.

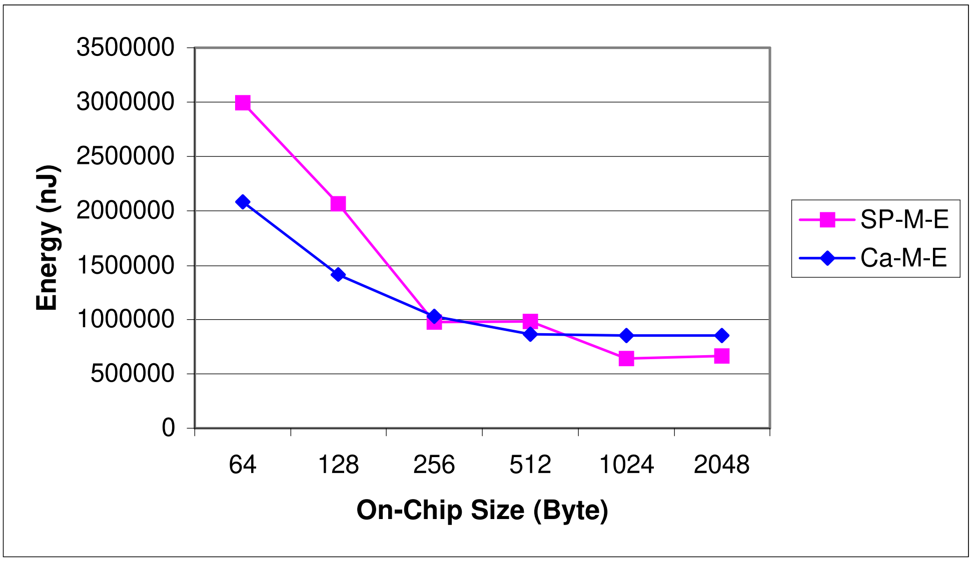


Abbildung 5.17 *Energieverbrauch bei insertion sort*

Die häufigen Sprüngen zwischen dem Scratch-Pad- und dem Hauptspeicher produzieren viele zusätzliche Prozessorzyklen (*Abbildung 5.8*). Dadurch wurde die Energiereduktion des Hauptspeichers bei dem Benchmark „insertion sort" deutlich geringer (6,7 %) und der Energieverbrauch der CPU (-12,5 %) wurde sogar größer als der des Prozessorsystems ohne On-Chip-Speicher (*Abbildung 5.18*).

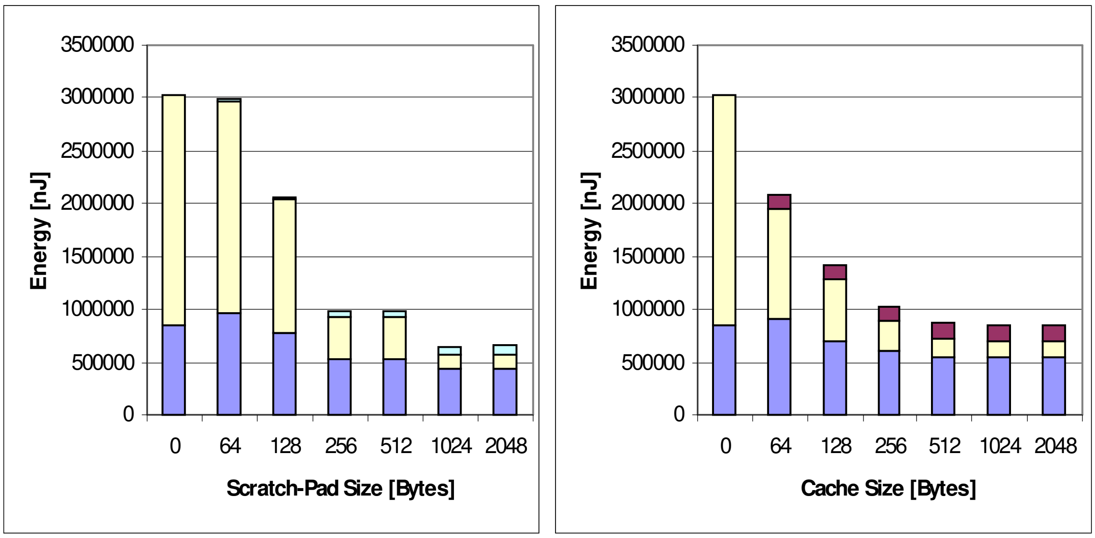


Abbildung 5.18 *Energieverbrauch mit 3 Kostenarten bei insertion sort*

Der Energieverbrauch des ARM7-Prozessorsystems mit Cachespeicher hat sich dagegen von Anfang an deutlich verringert. Da die Cachemissrate bei dem Benchmark auch bei der kleinen Cachegröße (z.B. 64 Byte) sehr niedrig (19,7 %) ist,

ist die Energiereduktion in diesem Fall gegenüber dem Scratch-Pad-Speicher viel größer.

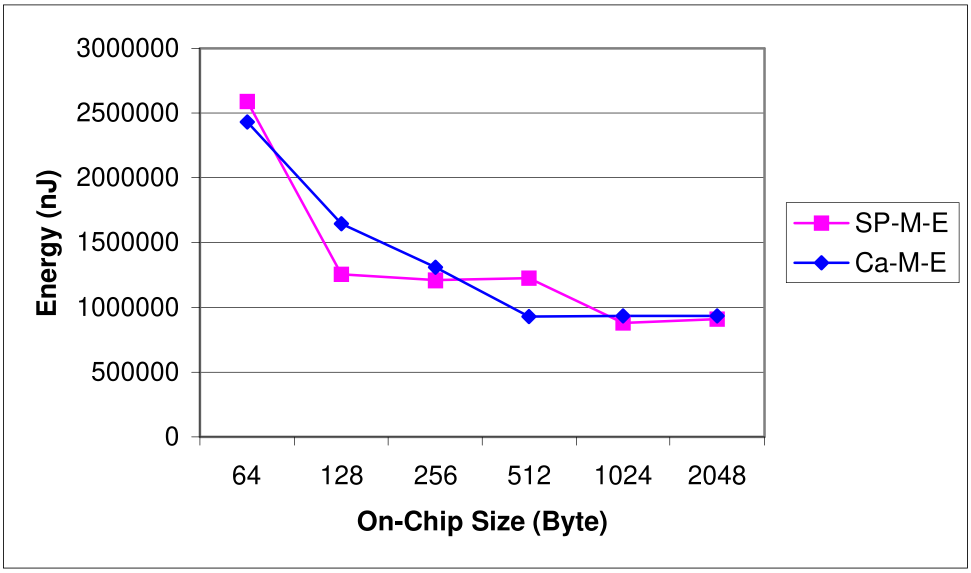


Abbildung 5.19 Energieverbrauch bei selection sort

Bei dem Benchmark „selection sort“ sehen die Verläufe der beiden Energieverbräuche ähnlich wie die Verläufe der Prozessorzyklen (*Abbildung 5.10*) aus. Der Verlauf mit dem Scratch-Pad-Speicher ändert sich zwischen der On-Chip-Speichergröße von 128 Byte und 512 Byte kaum. Der Scratch-Pad-Speicher hat für die Speichergröße von 256 Byte nur wenige (52 Byte) und bei 512 Byte keine Basisblöcke oder Daten mehr in den Scratch-Pad-Speicher verschieben können. Die zu verschiebenden Basisblöcke waren größer als der zusätzliche Platz im Scratch-Pad-Speicher.

Tabelle 5.4 Ausnutzung des Scratch-Pad-Speichers bei quick sort

Speichergröße (Byte)	64	128	256	512
Belegte Speicher (Byte)	44	72	156	392
Nutzungsrate	68,8 %	56,3 %	60,9 %	76,6 %

In der Tabelle 5.4 ist die Ausnutzung des Scratch-Pad-Speichers dargestellt. Bei dem Benchmark „quick sort“ wurde der Scratch-Pad-Speicher nicht vollständig ausgenutzt. Die Folge war, dass der Energieverbrauch des Prozessorsystems mit dem Scratch-Pad-Speicher bei der On-Chip-Speichergröße von 256 Byte höher als der Verbrauch des Systems mit dem Cache-Speicher lag (*Abbildung 5.20*).

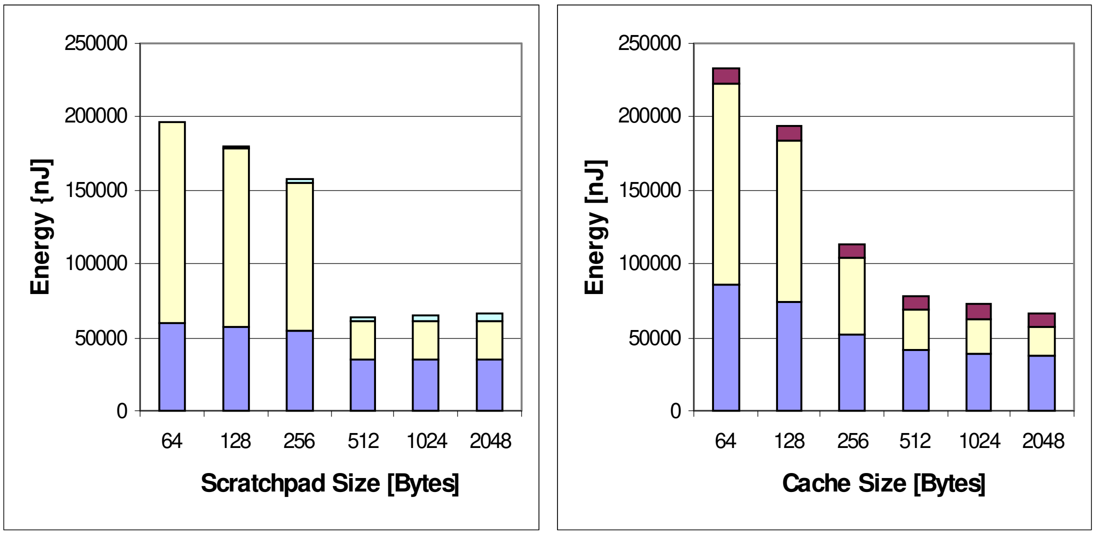


Abbildung 5.20 Energieverbrauch mit 3 Kostenarten bei quick sort

Durch diese niedrige Ausnutzung des Scratch-Pad-Speichers hat sich der Energieverbrauch des Hauptspeichers bis zu der On-Chip-Speichergröße von 256 Byte wenig verringert.

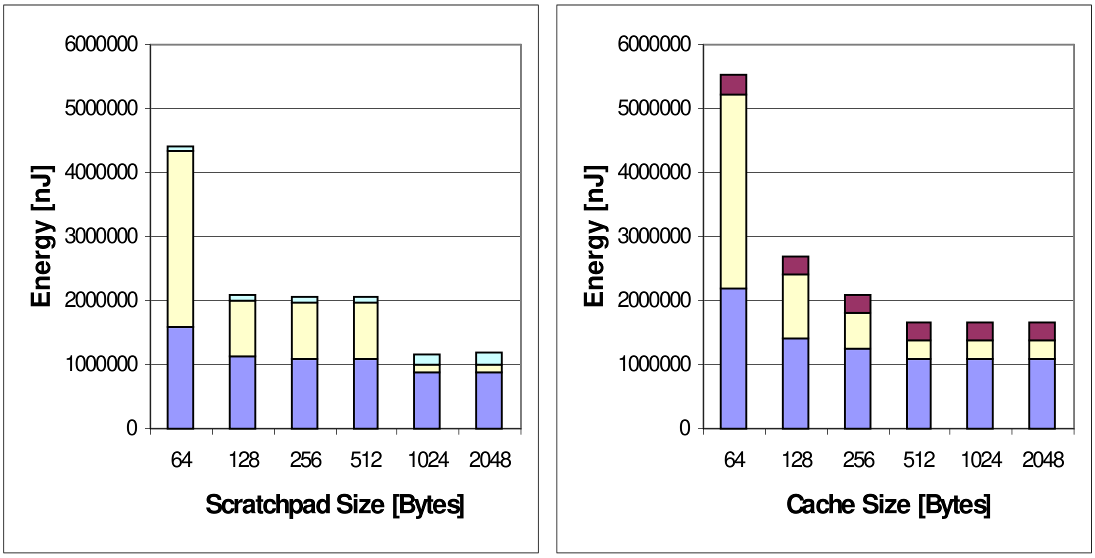


Abbildung 5.21 Energieverbrauch mit 3 Kostenarten bei bubble sort

Bei dem Benchmark „bubble sort" ist dieses Problem auch gut zu sehen (*Abbildung 5.21*). Die Programmteile bzw. Daten sind zu groß, so dass sie erst bei der Speichergröße von 1024 Byte vollständig verschoben werden.

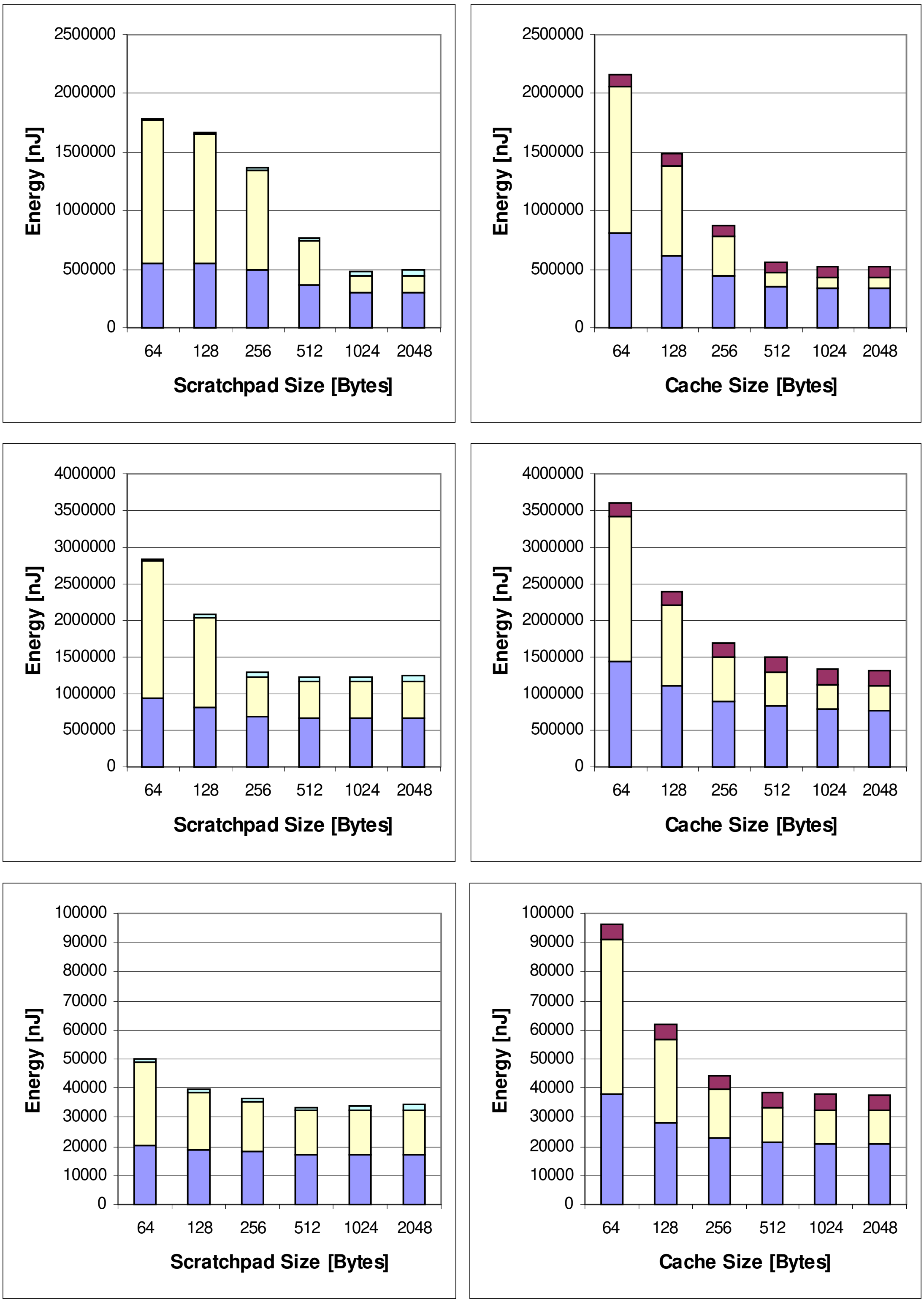


Abbildung 5.22 *Energieverbrauch bei heap sort (o), lattice (m) und matrix_mult (u)*

In der *Abbildung 5.22* sind die restlichen Vergleichsergebnisse dargestellt. Während bei dem „heap sort" das vorherige Problem gut zu erkennen ist, wurden bei den Benchmarks „lattice" und „matrix_mult" viel mehr Energiereinsparungen durch den Einsatz des Scratch-Pad-Speichers als mit dem Cache-Speicher erzielt.

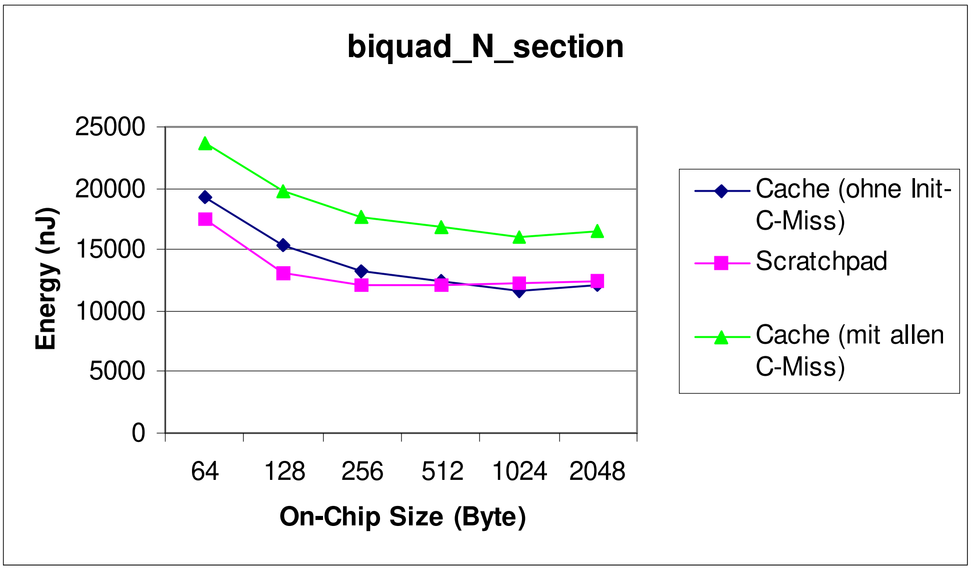


Abbildung 5.23 Energieverbrauch mit und ohne Initialisierungs-Cachemiss

Bei den bisherigen Untersuchungen, die im Rahmen dieser Diplomarbeit durchgeführt worden sind, wurden die Initialisierungs-Cachemisses immer berücksichtigt, weil ein Cache beim Programmstart leer ist und zuerst gefüllt werden muss. Um den Anteil dieser Initialisierungs-Cachemisses abschätzen zu können und den Einfluss auf den Gesamtenergieverbrauch zu untersuchen, wurde der Energieverbrauchsanteil der Initialisierungs-Cachemisses vom Gesamtenergieverbrauch subtrahiert.

In der *Abbildung 5.23* sind die Ergebnisse beim Benchmark „biquad_N_section" dargestellt. Um den Unterschied zwischen den Energieverbräuchen mit und ohne Initialisierungs-Cachemisses zu veranschaulichen, wurde auch der Energieverbrauch mit allen Cachemisses aufgeführt.

Da der Anteil der Initialisierungs-Cachemisses an den Gesamtcachemisses bei dem Benchmark „biquad_N_section" über 54 % (bei der Cachegröße von 128 Byte) liegt, beträgt der Energieverbrauch ohne Initialisierungs-Cachemisses bei der Cachegröße von 128 Byte um 22,4 % weniger als der mit allen Cachemisses.

Der Energieverbrauch des ARM7-Prozessorsystems mit einem Scratch-Pad-Speicher ist allerdings zwischen den On-Chip-Speichergrößen von 64 bis 512 Byte niedriger und ab der Speichergröße von 1024 Byte höher. Dies liegt daran, dass der Stack noch auf dem Hauptspeicher liegt und somit auf den Hauptspeicher zugegriffen werden muss. Seit kurzem ist der Scratch-Pad-Algorithmus von Steinke et al. [ST01], [Zob01] weiterentwickelt worden. Der Stack kann in den Scratch-Pad-Speicher verschoben werden, falls kein rekursiver Algorithmus im Programmcode verwendet wird. Ab der Scratch-Pad-Speichergröße von 256 Byte würde es beim Benchmark „biquad_N_section“ keinen Zugriff mehr auf den Hauptspeicher geben und der Energieverbrauch des Prozessorsystems sich mit dem Scratch-Pad-Speicher um 63,1 % (bei der On-Chip-Speichergröße von 512 Byte, *Tabelle 5.5*) verringern.

Tabelle 5.5 Energieverbrauch (in nJ) bei biquad_N_section

On-Chip-Speichergröße	64	128	256	512	1024	2048
mit Stackverschiebung	17088	13091	11771	4344	4344	4344
ohne Stackverschiebung	17088	13091	11771	11771	11771	11771

5.3 Vor- und Nachteile der beiden On-Chip-Speicher

Der Einsatz eines Cache- oder Scratch-Pad-Speichers hat gewisse Vor- und Nachteile. Durch die Beachtung der Vor- und Nachteile können die beiden On-Chip-Speicher effektiver eingesetzt werden. Die Nachteile eines Cache-Speichers sind die Vorteile eines Scratch-Pad-Speichers und umgekehrt. Daher werden nur die Vorteile der beiden Speicher erwähnt.

Ein Cache hat gegenüber dem Scratch-Pad folgende Vorteile:

- **Anwendungsunabhängig:**
  Ein Cacheeinsatz ist von den Anwendungen unabhängig, da ein Cache eine hardwaremäßige Steuerlogik besitzt, durch die der Cache gesteuert wird. D.h. ein Cache speichert automatisch das gerade gelesene Datum oder den Befehl für den nächsten, möglichen Zugriff auf das Datum oder den Befehl ab. Es ist kein Wissen über die Applikation notwendig.

- **Keine Modifikation des Compilers und Analyse der Anwendung:**
  Man braucht keine aufwendige Modifikation des Compilers und Analyse des einzusetzenden Programms.

- **Breitere Einsatzbereiche:**
  Da ein Cache anwendungsunabhängig eingesetzt werden kann, kann ein Cache überall zum Einsatz kommen, wo ein Scratch-Pad wegen der Vielfalt der Anwendungen, die im System laufen, nicht effektiv eingesetzt werden könnte.

Ein Einsatz eines Scratch-Pad-Speichers hat folgende Vorteile

- **Höhere Effektivität als Cache:**
  Ein Einsatz des Scratch-Pads kann höhere Effektivität als der Einsatz des Caches haben.

Die Voraussetzungen dafür sind:

1. Die Anwendung muss vorher analysiert werden, und/oder
2. der Compiler muss in der Lage sein, den Programmcode unter Berücksichtigung seiner Speicherkapazität so zu analysieren, dass nur die Daten, auf die am häufigsten zugegriffen wird und von der Größe her am besten in den Scratch-Pad passen, dorthin zu verschieben.

- **Kein Cachemiss:**
  Da ein Cache zuerst aufgefüllt werden muss, und die Cacheeinträge wegen der begrenzten Kapazität ständig ersetzt werden, ist ein Cachemiss unvermeidbar. Wegen des doppelten Aufsuchens des gesuchten Datums im Cache und dem Hauptspeicher entsteht unnötig mehr Energiebedarf und Programmausführungszeit. Im Gegensatz zu einem Cache werden in einem Scratch-Pad-Speicher nur die Daten und Programmteile gespeichert, die häufig zum Einsatz kommen, und die analog zum Hauptspeicher benutzt werden. Daher produziert ein Scratch-Pad niemals einen Fehlzugriff wie ein Cache.

- **Weniger Platzbedarf:**
  Da ein Scratch-Pad keine hardwaremäßige Steuereinheit und keinen Tag-Speicher benötigt, ist der Platzbedarf eines Scratch-Pad kleiner als der eines Caches. Daher könnte der Vergleich auch zwischen zwei flächenmäßig gleichgroßen Speichern erfolgen, was sich zugunsten des Scratch-Pad-Speichers auswirken würde.

- **Genauere WCET-Abschätzung:**
  Die WCET bei einem Cache wird pessimistisch abgeschätzt (ständige Cachemisses). Der Unterschied gegenüber einem normalen Ablauf eines Programms würde viel zu hoch sein. Beim Scratch-Pad ist es von vornherein klar, wo die Daten liegen. Daher ist eine genauere Abschätzung der WCET möglich.

5.4 Auswertung

Unter den untersuchten Cacheorganisationen ist der 2-fach assoziative Cache meistens am schnellsten und energiesparsamsten. Bei dem Benchmark „lattice“ hat der „Direct-Mapped Cache“ allerdings den niedrigsten Energieverbrauch. Trotzdem scheint der 2-fach assoziative Cache die bessere Wahl zu sein, weil der „Direct-Mapped Cache“ in der Regel eine höhere Cachemissrate hat, die man gerne vermeiden will.

Tabelle 5.6 Prozessorzyklen bei biquad_N_section (Cache)

Cache Size	0	64	128	256	512	1024	2048	4096	8192
Direct Mapping	1977	2125	1967	1791	1670	1670	1670	1670	1670
2xAssociative	1977	2153	1904	1774	1713	1661	1670	1665	1665
4xAssociative	1977	2136	1948	1816	1711	1701	1656	1656	1656
8xAssociative	1977	2178	2001	1781	1748	1678	1665	1665	1656
Full Associative	1977	2178	1977	1904	1749	1730	1687	1670	1656

Tabelle 5.7 Energieverbrauch bei bubble sort (Cache)

Cache Size	0	64	128	256	512	1024	2048
Direct Mapping	6324270	5292314	3007636	2145594	1951013	1968953	1560099
2xAssociative	6324270	5523332	2688740	2086739	1658186	1667716	1711183
4xAssociative	6324270	5701190	2818859	2199881	1858202	1877257	1913296
8xAssociative	6324270	6192108	3238730	2616093	2286536	2311935	2371737
Full Associative	6324270	5341701	2556863	2008363	1729673	1825523	2184866

Für die effiziente Nutzung eines Scratch-Pad-Speichers ist die Wahl der Programmteile und Daten, die in den Scratch-Pad verschoben werden, sehr wichtig. Hier spielt die Ausnutzung des Speicherplatzes eine große Rolle.

Tabelle 5.8 Ausnutzungsrate des Scratch-Pad-Speicherplatzes

On-Chip Size	64	128	256	512
biquad_N_section	62,5 %	93,8 %	75 %	1)
bubble sort	81,2 %	100 %	78,1 %	39,1 %
heap sort	87,5 %	87,5 %	82,8 %	100 %
insertion sort	100 %	100 %	96,9 %	48,4 %
lattice	87,5 %	96,9 %	92,2 %	64,8 %
matrix_mult	81,2 %	81,2 %	46,9 %	69,5 %
quick sort	68,8 %	56,3 %	60,9 %	76,6 %
selection sort	75 %	100 %	70,3 %	35,2 %

1) Bei der Speichergröße von 256 Byte liegen alle Programmteile im Scratch-Pad.

In der *Tabelle 5.8* ist dargestellt, wie effizient der Scratch-Pad-Speicher genutzt worden ist. Bei manchen Benchmarks wurden die Speicherplätze sogar weniger als 50 % ausgenutzt. Sicherlich können nicht immer und nicht alle Speicherplätze belegt werden. Allerdings können, z.B. durch die Teilung der Array-Größe, die vorhandenen Speicherplätze effizienter ausgenutzt werden.

Tabelle 5.9 Performanceverbesserung durch Cache vs. ohne On-Chip-Speicher

On-Chip Size	64	128	256	512	1024
biquad_N_section	- 8,9 %	3,7 %	10,3 %	13,4 %	13,4 %
bubble sort	- 23,8 %	21,6 %	31,3 %	38,1 %	38,3 %
heap sort	- 46,5 %	- 10,6 %	20,4 %	36,7 %	38,5 %
insertion sort	- 5,4 %	17,5 %	30,0 %	35,6 %	36,3 %
lattice	- 39,6 %	- 4,9 %	14,8 %	20,4 %	25 %
matrix_mult	- 39,5 %	- 2,9 %	15,5 %	21,9 %	22,9 %
quick sort	- 41,6 %	- 22,8 %	15,5 %	32,5 %	35,4 %
selection sort	10,9 %	28,9 %	36,2 %	44,5 %	44,8 %

Tabelle 5.10 Energiereduktion durch Cache vs. ohne On-Chip-Speicher

On-Chip Size	64	128	256	512	1024
biquad_N_section	25,6 %	37,8 %	44,4 %	47,0 %	49,5 %
bubble sort	12,7 %	57,5 %	67,0 %	73,8 %	73,6 %
heap sort	- 11,1 %	23,9 %	54,9 %	71,4 %	73,0 %
insertion sort	31,3 %	53,4 %	66,1 %	71,6 %	72,0 %
lattice	- 2,7 %	32,0 %	52,0 %	57,6 %	62,0 %
matrix_mult	- 2,8 %	33,8 %	52,4 %	58,9 %	59,4 %
quick sort	- 9,4 %	8,8 %	46,5 %	63,4 %	66,0 %
selection sort	46,0 %	63,5 %	71,0 %	79,3 %	79,3 %

Die *Tabelle 5.9* gibt die Performanceverbesserung durch den Cache-Speicher gegenüber dem System ohne On-Chip-Speicher an. Bis auf die Ausnahme „selection sort" wirkte sich die kleine Cachegröße auf die Performance negativ aus. Ein Cache mit kleiner Cachegröße produziert viele Cachemisses, da die Cacheinhalte ständig durch neu gelesene Daten oder Befehle ersetzt werden müssen. Es gibt nicht genügend Platz im Cache, in dem die Daten oder Befehle für den nächsten Zugriff bereit gestellt werden können. Ein Cacheeinsatz würde sich besonders lohnen, wenn z.B. eine sich oft wiederholende Schleife komplett in den Cache passen würde. Dadurch würden beim Schleifendurchlauf kaum Cachemisses entstehen. Die Simulationsergebnisse zeigen, dass die Performancesteigerung von bis zu 44,8 % durch den Einsatz des Cachespeichers erreicht wurde.

In der *Tabelle 5.10* ist die erreichte Energiereduktion durch den Cachespeicher dargestellt. Die kleine Cachegröße wirkte sich nicht so extrem negativ wie bei der Performance aus, da trotz der Cachemisses und der dadurch resultierenden zusätzlichen Zugriffe auf den Hauptspeicher doch mehr energieintensive Zugriffe auf den Hauptspeicher vermieden werden konnten. Da für den Cache-Speicher keine Energieoptimierung durchgeführt worden ist, und die Energieeinsparung nur durch die Minimierung der Hauptspeicherzugriffe durch die Cachezugriffe erreichbar ist, stehen die Performance und der Energieverbrauch sehr eng miteinander in Zusammenhang.

Tabelle 5.11 Performanceverbesserung durch Scratch-Pad vs. ohne On-Chip-Speicher

On-Chip Size	64	128	256	512	1024
biquad_N_section	25,8 %	33,5 %	35,9 %	35,9 %	35,9 %
bubble sort	11,3 %	38,8 %	38,8 %	38,8 %	51,0 %
heap sort	1,6 %	2,0 %	11,2 %	34,9 %	46,5 %
insertion sort	- 12,7 %	8,9 %	39,6 %	39,6 %	48,9 %
lattice	11,1 %	22,8 %	35,4 %	36,9 %	36,9 %
matrix_mult	25,4 %	32,2 %	34,9 %	37,0 %	37,0 %
quick sort	1,9 %	6,6 %	11,8 %	42,3 %	42,3 %
selection sort	15,7 %	41,8 %	42,7 %	42,7 %	48,8 %

In der *Tabelle 5.11* wird gezeigt, wieviel Performanceverbesserung gegenüber dem System ohne On-Chip-Speicher durch den Einsatz eines Scratch-Pad-Speichers erreicht wurde. Die größte Verbesserung wurde bei dem Benchmark „bubble sort" erzielt. Bei einer Scratch-Pad-Speichergröße von 1024 Byte waren 51 % weniger Prozessorzyklen nötig als bei einem Prozessorsystem ohne On-Chip-Speicher.

Der Benchmark „insertion sort" ist das einzige Beispiel dafür, dass durch die Sprungbefehle zwischen On- und Off-Chip-Speicher sogar mehr Prozessorzyklen als bei einem Prozessorsystem ohne On-Chip-Speicher produziert wurden. Bei den anderen Benchmarks haben sich die Prozessorzyklen verringert, weil die On-Chip-Speicherzugriffe keine Wartezyklen benötigen und durch die Ersetzung der vielen Off-Chip-Zugriffe durch die On-Chip-Zugriffe insgesamt weniger Prozessorzyklen benötigt wurden.

Tabelle 5.12 Energiereduktion durch Scratch-Pad vs. ohne On-Chip-Speicher

On-Chip Size	64	128	256	512	1024
biquad_N_section	44,8 %	58,7 %	62,2 %	62,0 %	61,7 %
bubble sort	30,6 %	66,9 %	67,5 %	67,4 %	81,8 %
heap sort	8,6 %	14,4 %	29,8 %	60,6 %	75,3 %
insertion sort	1,4 %	31,7 %	67,8 %	67,6 %	78,8 %
Lattice	19,7 %	41,1 %	63,4 %	65,3 %	65,0 %
matrix_mult	46,8 %	58,0 %	60,8 %	64,2 %	63,9 %
quick sort	7,8 %	15,9 %	26,4 %	70,0 %	69,6 %
selection sort	42,6 %	72,2 %	73,1 %	72,9 %	80,5 %

Es wurde auch untersucht, wie viel Energiereduktion durch den Einsatz eines Scratch-Pad-Speichers gegenüber dem System ohne On-Chip-Speicher erzielt werden kann (*Tabelle 5.12*). Durch den Scratch-Pad-Speicher konnte z.B. bei dem Benchmark „bubble sort" bis zu 81,8 % an Energie gespart werden. Allerdings waren in diesem Fall alle Programmteile und Daten im On-Chip-Speicher und das würde bei dem realen Einsatz in der Regel nicht der Fall sein. Aber selbst bei den kleinen

On-Chip-Speichergrößen wie 64 oder 128 Byte war eine große Energiereduktion zu beobachten.

Eindeutig zu sehen ist die Relation zwischen der Performance und dem Energieverbrauch. Als Beispiel sehen wir uns den Benchmark „heap sort“ an. Bei der On-Chip-Speichergröße von 64 Byte haben sich die Prozessorzyklen 1,6 % verringert, und der Energieverbrauch 8,6 %. Bei der Größe von 128 Byte sind es 2,0 % und 14,4 %, bei der Größe von 256 Byte sind es 11,2 % und 29,8 %, bei der Größe von 1024 Byte sind es 46,5 % und sogar 75,3 %. Wie bei dem Cache-Speicher hat sich der Energieverbrauch insgesamt viel stärker verringert als die Anzahl der Prozessorzyklen.

Tabelle 5.13 Performanceverbesserung durch Scratch-Pad vs. durch Cache

On-Chip Size	64	128	256	512	1024
biquad_N_section	31,9 %	31,0 %	28,5 %	26,0 %	23,7 %
bubble sort	28,3 %	21,9 %	11,7 %	1,9 %	20,6 %
heap sort	32,8 %	11,4 %	- 11,6 %	- 2,9 %	13,1 %
insertion sort	- 6,9 %	- 10,5 %	13,7 %	6,2 %	19,8 %
lattice	36,3 %	26,4 %	24,2 %	20,7 %	15,8 %
matrix_mult	46,5 %	34,2 %	23,0 %	19,3 %	18,3 %
quick sort	30,7 %	23,9 %	- 4,4 %	14,5 %	10,7 %
selection sort	5,4 %	18,3 %	10,1 %	- 3,2 %	7,3 %

Als nächstes wurden die erzielte Performanceverbesserung und die Energiereduktion der beiden On-Chip-Speicher miteinander verglichen. In der *Tabelle 5.13* ist dargestellt, wie viel mehr Performanceverbesserungen durch den Scratch-Pad-Speicher gegenüber dem Cache-Speicher erreicht worden sind.

Die Ergebnisse sprechen fast eindeutig für den Scratch-Pad-Speicher. Das liegt an den unvermeidbaren Cachemisses und den damit verbundenen zusätzlichen Zugriffen auf den Hauptspeicher. Diese sind, von der Anzahl her, viel mehr als die zusätzlichen Zyklen, die aufgrund der eingefügten Sprungbefehle für den Scratch-Pad-Speicher entstanden sind.

An manchen Stellen, wo negative Zahlen zu sehen sind, hat das System mit einem Cache-Speicher weniger Prozessorzyklen. Bei den Benchmarks „heap sort“, quick sort“ und „selection sort“ war das ARM7-Prozessorsystem mit einem Scratch-Pad-Speicher zuerst schneller, dann langsamer und dann wieder schneller als das System mit einem Cache-Speicher. Das ist darauf zurückzuführen, dass die Daten bzw. die Basisblöcke zu groß für den jeweiligen freien Scratch-Pad-Speicherplatz waren, und dass der Scratch-Pad-Speicher somit nicht voll genutzt werden konnte.

Tabelle 5.14 Energiereduktion durch Scratch-Pad vs. durch Cache

On-Chip Size	64	128	256	512	1024
biquad_N_section	25,9 %	33,6 %	47,1 %	28,3 %	24,2 %
bubble sort	17,0 %	30,4 %	4,3 %	- 5,8 %	41,6 %
heap sort	21,6 %	- 11,1 %	- 35,7 %	- 27,5 %	9,6 %
insertion sort	- 43,5 %	- 46,6 %	5,1 %	-14,0 %	24,5 %
lattice	21,8 %	13,3 %	23,7 %	18,2 %	7,8 %
matrix_mult	48,3 %	36,6 %	17,7 %	12,9 %	11,0 %
quick sort	15,8 %	7,8 %	- 37,7 %	17,9 %	10,7 %
selection sort	- 6,4 %	23,8 %	7,4 %	- 31,0 %	5,7 %

Die *Tabelle 5.14* gibt die Vergleichsergebnisse zwischen dem Scratch-Pad- und dem Cache-Speicher für die Energiereduktion an. Die positiven Zahlen bedeuten, dass eine größere Energieeinsparung durch den Scratch-Pad-Speicher erzielt worden ist. Hier ist auch gut zu erkennen, dass die negative Performance gegenüber dem Cache-Speicher mehr Energieverbrauch bedeutet. Bei dem Benchmark „insertion sort" war der Energieverbrauch des Systems mit dem Cachespeicher bei der On-Chip-Speichergröße von 64, 128 und 512 Byte niedriger. Bei dem Benchmark ist die Ausnutzungsrate des Speicherplatzes am Anfang ziemlich hoch, aber die Zugriffe auf den Scratch-Pad-Speicher waren gering, so dass der Energieverbrauch am Hauptspeicher vor allem bei den Scratch-Pad-Speichergrößen von 64 und 128 Byte viel höher als der des Systems mit dem Cachespeicher war. Dieses Beispiel zeigt, wie entscheidend die Wahl der Programmteile und Daten sein kann, die in den Scratch-Pad-Speicher verschoben werden sollen. Allerdings arbeitet der für die Untersuchung eingesetzte Scratch-Pad-Algorithmus statisch und dieser befindet sich noch in der Entwicklung. Trotzdem hat der Algorithmus große Energiereduktionsmöglichkeiten gezeigt.

Kapitel 6

Zusammenfassung und Ausblick

In der Vergangenheit hat sich die Geschwindigkeit zum fast alleinigen Maßstab in der Computerwelt entwickelt. Das wachsende Interesse der Konsumenten und der harte Konkurrenzkampf unter den Herstellern führten zu einer rapiden Entwicklung der Hardware im Hinblick auf erhöhte Funktionalität der Geräte. Es ist zu erwarten, dass die Ansprüche der Konsumenten hinsichtlich höherer Geschwindigkeit und zusätzlicher Funktionalität steigen und diese Entwicklung weiter fördern.

Allerdings geht entsprechend schnellere Hardware in der Regel mit einem höheren Energiebedarf einher. Dieser wachsende Energiebedarf wird vor allem bei den mobilen Geräten immer problematischer, weil sie auf eine autarke Energiequelle begrenzter Kapazität angewiesen sind. Hard-, Software-Industrie und auch die Batterie-Hersteller arbeiten mit Hochdruck an Lösungen, den Energieverbrauch zu senken und die Kapazität der Batterien zu erhöhen.

Ein vielversprechender Lösungsansatz für die Reduzierung des Energieverbrauchs besteht im Einsatz der Cache- und Scratch-Pad-Speicher. Da ein Zugriff auf einen Hauptspeicher viel mehr Energie als der Zugriff auf einen On-Chip-Speicher verbraucht, kann der Einsatz eines Cache- oder eines Scratch-Pad-Speichers den Energieverbrauch eines Prozessorsystems stark reduzieren.

Ein Cache-Speicher ist aufgrund des Hardware-Controllers weitgehend anwendungsunabhängig einsetzbar. Ein Scratch-Pad-Speicher hingegen wird durch Software gesteuert. Über die Zwischenspeicherung im Scratch-Pad-Speicher entscheidet ein spezifischer Algorithmus.

Platz und Energieverbrauch des Hardware-Controllers beim Cache-Speicher entfallen bei einem Scratch-Pad-Speicher. Er benötigt so eine geringere Chip-Fläche und der Energieverbrauch pro Zugriff ist niedriger als bei einem Cache-Speicher mit der gleichen Speicherkapazität.

Im Rahmen dieser Diplomarbeit wurden die beiden unterschiedlichen On-Chip-Speicher-Varianten „Scratch-Pad“ und „Cache“ in Bezug auf die Performance und den Energieverbrauch miteinander verglichen. Dabei wurden verschiedene Cacheorganisationen berücksichtigt.

Unter den untersuchten Cacheorganisationen ist der 2-fach assoziative Cache meistens am schnellsten und energiesparsamsten. Ein „Direct-Mapped Cache“ hat den niedrigsten Energieverbrauch pro Zugriff und z.B. bei dem Benchmark „biquad_N_section“ den geringsten Energieverbrauch. Allerdings hat der Direct-Mapped Cache“ in der Regel eine höhere Cachemissrate, so dass der 2-fach assoziative Cache insgesamt bei den eingesetzten Benchmarks sowohl für die Geschwindigkeit als auch für den Energieverbrauch eine günstigere Wahl ist. Daher wurde zum Vergleich mit dem Scratch-Pad-Speicher diese Cacheorganisation gewählt.

Beim Einsatz des Scratch-Pad-Speichers konnte eine bis zu 46,5 %-ige Reduktion der Prozessorzyklen im Vergleich zum ARM7-Prozessorsystem mit einem Cachespeicher gemessen werden (Benchmark „matrix_mult“). Weniger deutlich treten die Vorteile der On-Chip-Speicher bei den kleinen Speichergrößen von 64 Byte auf. Ein Cachespeicher mit der Speichergröße 64 Byte produziert viele Cachemisses, da die Cacheinhalte aufgrund der zu kleinen Speicherkapazität ständig ersetzt werden.

Ein Scratch-Pad-Speicher produziert zwar keine Fehlzugriffe, aber die häufigen Sprünge zwischen dem On-Chip- und dem Off-Chip-Speicher verursachen mehr Prozessorzyklen. Ein typisches Beispiel hierfür ist der Benchmark „insertion sort“. Durch den Einsatz des Scratch-Pad-Speichers mit der Speichergröße von 64 Byte stiegen die Prozessorzyklen um 12,7 % gegenüber dem Prozessorsystem ohne On-Chip-Speicher an. Die Senkung des Energieverbrauchs fiel in diesem Fall nur äußerst geringfügig aus, sie lag bei 1,4 % im Vergleich zu einem Prozessorsystem ohne On-Chip-Speicher.

Das ARM7-Prozessorsystem mit einem Scratch-Pad-Speicher hat bei den eingesetzten Benchmarks 31-mal (Cache-Speicher : 9-mal, *Tabelle 5.14*) einen niedrigeren Energieverbrauch als das System mit einem Cache-Speicher. Das liegt an den unvermeidbaren Cachemisses und den damit verbundenen zusätzlichen Zugriffen auf den Hauptspeicher beim System mit Cache-Speicher. Die Anzahl der zusätzlichen Zyklen durch die Cachemisses ist größer als die Anzahl der zusätzlichen Zyklen, die aufgrund der eingefügten Sprungbefehle für den Scratch-Pad-Speicher und für das neue Füllen der Pipeline notwendig sind.

Bei den Benchmarks „heap sort“, quick sort“ und „selection sort“ hat das ARM7-Prozessorsystem mit einem Scratch-Pad-Speicher an einigen Stellen (*Tabelle 5.14*) einen höheren Energieverbrauch als ein System mit einem Cache-Speicher. Das ist darauf zurückzuführen, dass die Daten bzw. die Basisblöcke zu groß für den jeweiligen freien Scratch-Pad-Speicherplatz waren, und der Scratch-Pad-Speicher somit nicht voll genutzt werden konnte. Bei manchen Benchmarks wurden die Speicherplätze sogar zu weniger als 50 % ausgenutzt. Sicherlich können nicht immer und nicht alle Speicherplätze belegt werden. Allerdings können, z.B. durch die Teilung der Array-Größe, die vorhandenen Speicherplätze effizienter ausgenutzt werden. Außerdem arbeitet der eingesetzte Scratch-Pad-Algorithmus statisch und seine Entwicklung ist noch nicht zu Ende. Ein neuer Algorithmus, der dynamisch arbeitet, befindet sich momentan in der Entwicklung. Somit können die

Speicherinhalte, auf die nicht mehr zugegriffen wird, durch andere Basisblöcke bzw. Daten ersetzt werden.

Bei der Untersuchung wurden keine Optimierungen für den Cache-Speicher durchgeführt. Für einen Cache-Speicher existieren jedoch mehrere Optimierungen. Man kann z.B. Schleifen in dem Quellcode vertauschen [HG98], Arraygrößen anpassen, damit sie in den Cacheblock passen. Diese dienen in der Regel zur Geschwindigkeitssteigerung. Ein Cache-Einsatz als Lösungsansatz für die Energiereduktion erfolgt ausschließlich durch die Reduzierung bzw. die Ersetzung der energiemäßig teuren Off-Chip-Zugriffe durch die Cache-Zugriffe. Daher sind die Energie- und Geschwindigkeitsoptimierung bei dem Cache-Speicher proportional zueinander.

Ein weiterer Vorteil eines Scratch-Pad-Speichers besteht in der geringeren Beanspruch von Chip-Fläche im Vergleich zu einem Cache-Speicher. Ein weiterer Untersuchungsansatz wäre daher der Vergleich des Energieverbrauchs der beiden On-Chip-Speicher mit einer gleich großen Chip-Fläche.

Die Untersuchungen haben gezeigt, dass die Verwendung von On-Chip-Speicher zu einer deutlichen Reduktion des Energieverbrauchs der Prozessorsysteme führt. Ähnliche Untersuchungen wurden von Banakar et al. [BS01] durchgeführt. Bei den Untersuchungen wurden außer dem Energieverbrauch und der Performance der Chip-Flächenbedarf der beiden On-Chip-Speicher betrachtet.

Im Vergleich von Scratch-Pad und Cache-Speicher sind im Hinblick auf Geschwindigkeit und Energieverbrauch die Vorteile bei Scratch-Pad-Speicher zu sehen. Nachteile durch den Einsatz des Scratch-Pad-Speichers ergeben sich bei der aufwendigen Analyse der Anwendungen und damit verbundenen Abhängigkeit.

Ein Einsatz der On-Chip-Speicher ist nur einer von mehreren Lösungsansätzen für die Reduzierung des Energieverbrauchs. Energiereduktion muss auf allen Ebenen erfolgen. Obwohl die Hardware-Industrie in der letzten Zeit schon allein aufgrund der enormen Wärmeentstehung viel für die Energiereduktion getan hat [S300] [AMD2], können sie allein dieses Problem nicht lösen. Schließlich werden die Anwendungen auch immer umfangreicher. Um diese Anwendungen in einer angemessenen anwendergerechten Zeit auszuführen, muss eine leistungsfähigere Hardware entwickelt werden, und somit haben die Anwendungen einen höheren Energiebedarf. Schon bei der Implementierungsphase sollte die Energiereduktion berücksichtigt werden, und/oder nach der Implementierung können Energieoptimierungen, z.B. „DTSE optimizations“ [DCM99], für den erstellten Quellcode ausgeführt werden. Es existieren auch viele energiesparende Compileroptimierungen, z.B. „loop permutations“, „loop tiling“ und „loop fusion“ [KV00]. Das Problem, die Reduzierung des Energieverbrauchs, kann nur dann gelöst werden, wenn die Bemühungen von allen Seiten kommen.

Literaturverzeichnis

[AMD1] *AMD-K6®-III Processor Technical Features and Innovations*, http://www.amd.com/products/cpg/k6iii/techfeatures.html.

[AMD2] G. Davis and D. Paulson: *Am186CC Microcontroller Power Management Circuit Application Note*, http://www.amd.com/epd/desiging/codekits/3.series2/23.ck0023pow/51.ck002300/23111.pdf, November 1999.

[ARM7T98] *ARM710T Datasheet*, ARM Ltd., http://www.arm.com, 1998.

[ARM95a] *An Introduction to Thumb*, Advanced RISC Machines Ltd., 1995.

[ARM95b] *ARM7TDMI Data Sheet*, Advanced RISC Machines Ltd., 1995.

[ARMc98] *Application Note 53 Configuring ARM Caches*, ARM Ltd., http://www.arm.com, 1998.

[ARMcm98] *Application Note 51 ARMulator Cache Models*, ARM Ltd., http://www.arm.com, 1998.

[ARMco98] *Application Note 52 The ARMulator Configuration File*, ARM Ltd., http://www.arm.com, 1998.

[ARMu99] *Application Note 32 ARMulator*, ARM Ltd., http://www.arm.com, 1999.

[ATM98] *AT91EB01 Evaluation Board*, User Manual, Atmel Corporation, 1998.

[ATM99a] *AT91 ARM Thumb Microcontroller*, AT91M40400, Atmel Corporation, 1999.

[ATM99b] *Embedded RISC Microcontroller Core*, ARM7TDMI, Atmel Corporation, 1999.

[BS01] R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan and P. Marwedel: *Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption*, Technical report number 762, Universität Dortmund, Sep. 2001

[CLR90] T. Cormen, C. Leiserson and R. Rivest: *Instruction to Algorithms*, The MIT Press, 1990

[CMA01] *CMA222 ARM710T CPU Module*, http://www.cogcomp.com/data_sheets/cma222ds.htm.

[DCM99] K. Danckaert, F. Catthoor and H. De Man: *Platform independent data transfer and storage exploration illustrated on a parallel cavity detection algorithm*, PDPTA (CSREA Conf. on Parallel and Distributed Processing Techniques and Applications), June 1999

[EN01] *"enCC" Homepage* am Informatik-Lehrstuhl 12 der Universität Dortmund, http://ls12-www.cs.uni-dortmund.de/research/encc/Welcome.html, 2001

[GMV00] C. Ghez, M. Miranda, A. Vandecappelle, F. Catthoor and D. Verkest: *Systemmatic high-level Address Code Transformations for Piece-wise Linear Indexing: Illustration on a Medical Imaging Algorithm*, IMEC Lab. Kapeldreef 75, Belgium, 2000.

[HG98] H. Hopp, D. Genius und S. Braun: *Cacheoptimierungen*, Interner Bericht, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 1998.

[HP96] J. Hennessy and D. Patterson: *Computerarchitektur*, Morgan Kaufmann Publishers Inc., 1996.

[HPS98] I. Hong, M. Potkonjak and M. Srivastava: *On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor*, In Proc. of ICCAD, 1998.

[Lin01] V. Lindenstruth: *Cache*, web.kip.uni-heidelberg.de/Hardwinf/I2WS00/folien/l11_cache.pdf, 2001.

[KV00] M. Kandemir, N. Vijaykrishnan, M. Irwin and W. Ye: *Influence of Compiler Optimizations on System Power*, DAC, 2000.

[Low00] *Low Power Design Guide*, http://www.lowpower.de, 2000.

[Ma00] P. Marwedel, Rechnerarchitektur, Vorlesungsskript am Informatik-Lehrstuhl 12 der Universität Dortmund, 2000.

[OIY99] T. Okuma, T. Ishihara and H. Yasuura, Real-Time Task Scheduling for a Variable Voltage Processor, ISSS, 1999.

[RJ99] G. Reinman, N. Jouppi: *An integrated cache timing and power modell*, Report – COMPAQ Western Research Laboratory, Palo Alto, 1999.

[S300] *Simply Revolutionary*, http://www.transmeta.com/technology/index.html, Jan. 2000.

[Sj98] J. Sjödin, B. Fröderberg and T. Lindgren: *Allocation of Global Data Objeckts in On-Chip RAM*, Proc. Of the ACM CASES Workshop on Compiler and Architectural Support for Embedded Computer Systems, December 1998.

[ST01] S. Steinke, C. Zobiegala, L. Wehmeyer and P. Marwedel: *Moving programm objects to scratch-pad memory for energy reduction*, Technical report number 756, Universität Dortmund, Apr. 2001.

[Sw00] R. Schwarz: *Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining,* Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2000.

[Syn96] *Power Products Reference Manual*, Version 3.5, Synopsys, 1996.

[Theo00] M. Theokharidis: *Energiemessung von ARM7TDMI Prozessor-Instruktionen*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2000.

[TG01] A. Tanenbaum and J. Goodman: *Computerarchitektur*, Pearson Studium, 2001.

[WJ94] Steven J.L. Wilton and Norman P. Jouppi: *An Enhanced Access and Cycle Time Model for On-Chip Caches*, WRL Research Report 93/5, 1994.

[Zob01] C. Zobiegala: *Energieeinsparung durch compilergestützte Nutzung des On-Chip-Speichers*, Diplomarbeit am Informatik-Lehrstuhl 12 der Universität Dortmund, 2001.

Anhang

A. Thumb-Kernbefehlssatz

Instruktion	Instruktionsart	Zyklen	Aktion
ADC (Op1, Op2)	Add with Carry	1	Op:=Op1 + Op2 + C-bit
ADD (Op1, Op2)	Add	1	Op:=Op1 + Op2
AND (Op1, Op2)	AND	2	Op:=Op1 AND Op2
ASR (Op1, Op2)	Arithmetic Shift Right	1	Op:=Op1 ASR Op2
BIC (Op1, Op2)	Bit Clear	1	Op:=Op1 AND NOT Op2
CMN (Op1, Op2)	Compare Negative	1	CPSR Flags:= Op1 + Op2
CMP (Op1, Op2)	Compare	1	CPSR Flags:= Op1 - Op2
EOR (Op1, Op2)	EOR	1	Op:=Op1 EOR Op2
LSL (Op1, Op2)	Logical Shift Left	2	Op:=Op1 « Op2
LSR (Op1, Op2)	Logical Shift Right	2	Op:=Op1 » Op2
MUL (Op1, Op2)	Multiply	2-5	Op:=Op1 · Op2
NEG (Op1, Op2)	Negate	1	Op:= - Op2
ORR (Op1, Op2)	OR	1	Op:=Op1 OR Op2
ROR (Op1, Op2)	Rotate Right	1	Op:=Op1 ROR Op2
SBC (Op1, Op2)	Subtract with Carry	1	Op:=Op1 - Op2 - Not C-Bit
SUB (Op1, Op2)	Subtract	1	Op:=Op1 - Op2
B (Op1)	Unconditional Branch	3	PC:= Op1
BCC (Op1)	Conditional Branch	3	PC:= Op1
BL (Op1)	Branch and Link	3	LR:= PC - 2; PC:= Op1
BX (Op1)	Branch and Exchange	3	PC:= Op1
LDMIA (Op1, OpN)	Load Multiple	3-10	OpN:= [Op1]; Op1:= Op1 + 4 · N
LDR (Op1, Op2)	Load Word	3	Op1:= [Op2]
LDRB (Op1, Op2)	Load Byte	3	Op1:= [Op2]
LDRH (Op1, Op2)	Load Halfword	3	Op1:= [Op2]
LDRSB (Op1, Op2)	Load signed Byte	3	Op1:= [Op2]
LDRSH (Op1, Op2)	Load signed Halfword	3	Op1:= [Op2]
MOV (Op1, Op2)	Move Register	1	Op1:= Op2
STMIA (Op1, OpN)	Store Multiple	2-9	[OpN]:= Op1; Op1:= Op1 + 4 · N
STR (Op1, Op2)	Store Word	2	[Op2]:= Op1
STRB (Op1, Op2)	Store Byte	2	[Op2]:= Op1
STRH (Op1, Op2)	Store Halfword	2	[Op2]:= Op1
POP (OpN)	Pop Multiple	3-10	OpN:= [SP]; SP:= SP - 4 · N
PUSH (OpN)	Push Multiple	2-9	[SP]:= OpN; SP:= SP - 4 · N
SWI (Op1)	Software Interrupt	2	LR:= PC - 2; PC:= Op1

B. Simulationsergebnisse

Scratch-Pad

Energie(nJ)

Onchip Size	0	64	128	256	512	1024	2048	4096	8192
biquad	31812	17553	13131	12035	12092	12184	12362	12468	13071
bubble_sort	6324270	4390280	2093590	2052324	2064602	1149391	1191592	1216912	1360393
heap_sort	1944976	1777473	1664339	1364686	765626	480044	491915	499038	539400
insertion_sort	3034858	2993083	2072298	977425	983320	641945	661827	673757	741358
lattice	3518066	2823399	2073188	1288592	1221489	1232226	1252875	1265264	1335470
matrix-mult	93711	49829	39366	36709	33556	33837	34379	34703	36544
me_ivlin	16256297	8705516	7415238	4825500	4054769	3573293	3661080	3786103	3994476
quick_sort	213308	196593	179293	157067	64143	64799	66060	66816	71103
selection_sort	4506018	2587806	1253228	1211629	1221105	880717	911739	930351	1035825

CPU-Cycle

Onchip Size	0	64	128	256	512	1024	2048	4096	8192
biquad	1977	1467	1314	1268	1268	1268	1268	1268	1268
bubble_sort	391820	347549	239947	237949	237949	191953	191953	191953	191953
heap_sort	121427	119540	119042	107833	79057	64918	64918	64918	64918
insertion_sort	187602	211435	170936	113312	113312	95783	95783	95783	95783
lattice	224001	199217	172988	144686	141402	141402	141402	141402	141402
matrix-mult	5852	4364	3966	3808	3687	3687	3687	3687	3687
me_ivlin	1003666	746770	702909	644717	589539	530557	530550	530550	530550
quick_sort	13208	12958	12334	11652	7625	7625	7625	7625	7625
selection_sort	278274	234552	161826	159430	159430	142498	142498	142498	142498

On-Chip Access

Onchip Size	0	64	128	256	512	1024	2048	4096	8192
biquad	0	510	675	709	709	709	709	709	709
bubble_sort	0	104691	151673	153469	153469	163547	168801	168801	168801
heap_sort	0	11891	21089	31358	42764	46552	47485	47485	47485
insertion_sort	0	37595	60744	73688	73688	79531	79531	79531	79531
lattice	0	24892	52549	80863	82595	82595	82595	82595	82595
matrix-mult	0	1680	2030	2050	2165	2165	2165	2165	2165
me_ivlin	0	304512	357065	508249	501151	463043	463050	464202	464202
quick_sort	0	1098	1722	2698	4994	5043	5043	5043	5043
selection_sort	0	104832	117346	118442	118442	124086	124086	124086	124086

Cache

DM = Direct Mapping, FA = Full Associative, 2 - 8 x = 2 - 8 x Associative

Biquad Energie(nJ)

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	31812	22472	19952	17193	15276	15405	15647	15814	16556
2x	31812	23695	19772	17703	16865	16074	16465	16637	17159
4x	31812	24585	21692	19547	17901	17897	17363	17868	18381
8x	31812	27687	24909	21333	20927	19921	20032	20470	21020
FA	31812	23723	20585	19963	17698	18075	19379	21484	27315

CPU-Cycle

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	1977	2125	1967	1791	1670	1670	1670	1670	1670
2x	1977	2153	1904	1774	1713	1661	1670	1665	1665
4x	1977	2136	1948	1816	1711	1701	1656	1656	1656
8x	1977	2178	2001	1781	1748	1678	1665	1665	1656
FA	1977	2178	1977	1904	1749	1730	1687	1670	1656

Cacheaccess

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	0	720	681	636	604	604	604	604	604
2x	0	727	661	630	613	601	604	602	602
4x	0	717	673	639	612	612	599	599	599
8x	0	726	683	630	622	605	601	602	599
FA	0	726	680	659	621	617	606	604	599

bubble_sort Energie(nJ)

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	6324270	5292314	3007636	2145594	1951013	1968953	1560099	1590260	1724611
2x	6324270	5526286	2688740	2086739	1658186	1667716	1711183	1753323	1847917
4x	6324270	5703527	2829565	2199881	1858202	1877257	1913296	2005315	2097424
8x	6324270	6209628	3249427	2621963	2286536	2311935	2371737	2449152	2577710
FA	6324270	5341701	2556863	2008363	1729673	1825523	2184866	2620151	3723621

CPU-Cycle

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	391820	481857	336047	282303	269487	269118	241407	241407	241407
2x	391820	485053	307068	269351	242497	241610	241586	241402	241402
4x	391820	481920	302452	263967	242636	241672	241458	241467	241402
8x	391820	479793	302294	263579	242474	241666	241561	241444	241426
FA	391820	479793	303298	263597	243337	242012	241618	241573	241485

Cacheaccess

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	0	150663	116210	102677	99104	99011	91911	91911	91911
2x	0	147711	108593	99423	92193	91960	91954	91909	91909
4x	0	146611	107067	97991	92241	91983	91922	91925	91909
8x	0	146002	106972	97848	92200	91980	91952	91920	91914
FA	0	146002	107079	97805	92443	92076	91971	91951	91933

heap_sort Energie(nJ)

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	1944976	2139638	1398671	818691	721262	685317	485373	494817	536883
2x	1944976	2161753	1479507	877046	555375	526030	536272	545898	575516
4x	1944976	2244105	1569265	927916	625047	590727	598346	625564	653405
8x	1944976	2440811	1763894	1090914	759398	732744	742296	765085	804273
FA	1944976	2139019	1504342	897155	574401	579054	687539	820510	1163760

CPU-Cycle

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	121427	180080	132951	95985	89724	87147	74190	74190	74190
2x	121427	178075	134346	96639	76842	74663	74460	74190	74190
4x	121427	177458	134975	95707	77272	74620	74343	74251	74176
8x	121427	178314	137098	96806	77104	74931	74389	74276	74212
FA	121427	178314	137617	97707	76898	74994	74614	74414	74260

Cacheaccess

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	0	53556	42197	33765	32237	30961	27628	27628	27628
2x	0	52552	42524	33649	28326	27717	27691	27626	27626
4x	0	52705	42100	32960	28416	27792	27669	27647	27626
8x	0	52985	42721	33051	28353	27750	27668	27646	27633
FA	0	52985	42828	33296	28332	27827	27748	27692	27645

insertion_sort Energie(nJ)

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	3034858	1858740	1448019	1064694	1016022	1020358	792401	806958	871803
2x	3034858	2087454	1413743	1029697	862268	849994	869749	885699	931355
4x	3034858	2329923	1489057	1105210	956827	954339	965224	1008784	1051703
8x	3034858	2549465	1699480	1310451	1170263	1164666	1187393	1223091	1284232
FA	3034858	2160445	1353362	1030048	896259	929979	1097499	1305785	1837264

CPU-Cycle

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	187602	187396	161670	137839	134473	134051	119038	119038	119038
2x	187602	197710	154758	131241	120860	119459	119381	119038	119038
4x	187602	205828	153005	129890	120774	119658	119191	119149	119033
8x	187602	205090	153208	129790	121011	119665	119281	119137	119081
FA	187602	205090	153336	130792	121190	119869	119328	119207	119103

Cacheaccess

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	0	62064	55810	49950	49126	49028	45400	45400	45400
2x	0	64675	53989	48357	45856	45504	45482	45400	45400
4x	0	66661	53537	48032	45828	45555	45436	45426	45398
8x	0	66236	53608	48007	45880	45547	45459	45421	45410
FA	0	66236	53608	48238	45921	45602	45472	45443	45415

lattice Energie(nJ)

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	3518066	3552044	2140182	1591060	1392036	1291399	1266206	1258516	1335035
2x	3518066	3614302	2391716	1688180	1492542	1336109	1336119	1353568	1411425
4x	3518066	3749987	2642201	1854826	1617984	1499244	1467648	1518176	1572128
8x	3518066	4068717	3053968	2171168	1922726	1796077	1777852	1812304	1891826
FA	3518066	3506140	2585729	1837196	1462973	1432170	1652226	1931029	2659547

CPU-Cycle

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	224001	315967	225825	191145	178377	171267	167939	166243	165440
2x	224001	312654	234891	190929	178204	167941	166177	165406	165095
4x	224001	310674	241045	192618	177581	169225	165886	165250	164770
8x	224001	309435	247361	193905	178113	169452	166148	165135	164775
FA	224001	309435	250453	199557	174305	167662	166169	165477	165028

Cacheaccess

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	0	94350	73794	65920	62710	60988	60183	59765	59563
2x	0	93587	75483	65757	62774	60160	59752	59558	59481
4x	0	92896	76931	66048	62558	60493	59667	59519	59407
8x	0	92559	78229	66355	62665	60538	59744	59488	59406
FA	0	92559	78849	67452	61648	60069	59722	59564	59463

matrix-mult Energie(nJ)

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	93711	85298	66117	44662	44263	35213	35988	36521	38895
2x	93711	96362	62074	44585	38539	38036	38527	39326	40997
4x	93711	102398	68676	48167	44083	41806	41983	43603	45248
8x	93711	114051	76272	53234	51368	49409	50120	51762	53715
FA	93711	98998	57504	44802	42700	42667	47254	54735	74253

CPU-Cycle

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	5852	7645	6426	5116	5075	4501	4501	4501	4501
2x	5852	8161	6023	4945	4568	4513	4496	4496	4496
4x	5852	8278	6180	4950	4685	4518	4487	4487	4487
8x	5852	8433	6169	4809	4657	4513	4492	4506	4487
FA	5852	8433	5825	4953	4756	4628	4518	4504	4504

Cacheaccess

Cache Size	0	64	128	256	512	1024	2048	4096	8192
DM	0	2320	2012	1689	1678	1534	1534	1534	1534
2x	0	2416	1908	1647	1551	1535	1532	1532	1532
4x	0	2439	1945	1641	1580	1539	1529	1529	1529
8x	0	2461	1936	1612	1573	1536	1531	1535	1529
FA	0	2461	1857	1648	1596	1565	1537	1534	1532

me_ivlin Energie(nJ)

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	16256297	7367642	5876439	4421132	3385648	3379507	3510032	3599768	3999502
2x	16256297	7556714	5438347	4513630	3809342	3822304	3950555	4085159	4366603
4x	16256297	8457108	6116558	5064681	4400166	4443952	4558903	4831520	5108885
8x	16256297	10062742	7426559	6364178	5667529	5739092	5918015	6152924	6536339
FA	16256297	7929016	5446748	4557622	3976189	4277665	5359732	6655866	9942899

CPU-Cycle

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	1003666	907057	811121	716915	650623	645986	645986	645986	645986
2x	1003666	894404	757496	696734	649816	646123	645986	645986	645986
4x	1003666	912107	761324	694197	649969	646224	646022	645981	645981
8x	1003666	930991	766436	696574	649053	646331	646065	645972	645972
FA	1003666	930991	769793	696563	649578	646532	646099	646040	646005

Cacheaccess

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	0	298067	272308	250214	235101	233789	233789	233789	233789
2x	0	292219	260055	245532	234697	233825	233789	233789	233789
4x	0	296222	260688	244871	234698	233847	233800	233787	233787
8x	0	300435	261882	245331	234466	233867	233803	233784	233784
FA	0	300435	262584	245331	234594	233915	233814	233799	233792

quick_sort Energie(nJ)

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	213308	233201	189990	115356	85020	65237	63443	64213	68294
2x	213308	233501	194531	114058	78151	72599	68289	69169	72042
4x	213308	241046	203007	118800	81622	75086	75156	76860	79540
8x	213308	260369	216906	130675	95860	88882	89006	90350	94890
FA	213308	230609	189898	114514	82450	77747	86850	98401	131722

CPU-Cycle

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	13208	19052	16318	11552	9607	8365	8184	8175	8175
2x	13208	18706	16218	11159	8918	8534	8201	8175	8175
4x	13208	18620	16214	11029	8764	8311	8240	8178	8170
8x	13208	18714	16040	10875	8805	8335	8239	8178	8209
FA	13208	18714	16094	11171	9081	8581	8451	8345	8314

Cacheaccess

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	0	5735	5084	3987	3460	3145	3096	3093	3093
2x	0	5645	5012	3817	3281	3178	3098	3093	3093
4x	0	5597	5008	3777	3241	3129	3106	3093	3091
8x	0	5624	4960	3736	3247	3130	3107	3092	3100
FA	0	5624	4967	3811	3307	3188	3164	3136	3124

selection_sort Energie(nJ)

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	4506018	2263487	1726971	1243854	1054680	1063746	863118	882487	968768
2x	4506018	2434131	1644175	1308404	932080	933630	960819	987107	1047854
4x	4506018	2497127	1737084	1323054	1064615	1069068	1090428	1148888	1208005
8x	4506018	2877459	2006570	1586608	1336588	1349548	1384322	1433696	1516769
FA	4506018	2376548	1520837	1201694	973375	1036812	1265904	1545249	2253453

CPU-Cycle

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	278274	242696	209252	179503	167583	167214	153452	153452	153452
2x	278274	248004	197986	177429	154492	153666	153608	153443	153443
4x	278274	242858	195051	170302	154810	153751	153514	153481	153438
8x	278274	248033	194804	169432	154499	153818	153547	153455	153473
FA	278274	248033	193290	170243	154735	154022	153672	153632	153551

Cacheaccess

CacheSize	0	64	128	256	512	1024	2048	4096	8192
DM	0	108973	101223	94211	91381	91289	88041	88041	88041
2x	0	109973	98579	93729	88291	88091	88077	88039	88039
4x	0	108725	97897	92103	88365	88112	88055	88048	88038
8x	0	109849	97815	91901	88293	88127	88064	88042	88046
FA	0	109849	97465	92083	88349	88177	88093	88083	88063

Cachemisses

biquad_N_section

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	86	68	48	34	34	34	34	34
2xAssociative	89	61	46	39	33	34	33	33
4xAssociative	87	66	51	39	37	32	32	32
8xAssociative	92	72	47	43	35	33	33	32
Full Associative	92	69	61	43	41	36	34	32

bubble sort

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	27533	11083	4837	3357	3311	112	112	112
2xAssociative	28074	7720	3367	241	136	133	111	111
4xAssociative	27615	7199	2737	256	143	118	119	111
8xAssociative	27342	7155	2695	236	142	130	116	114
Full Associative	27342	7270	2695	335	183	136	131	121

heap sort

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	12183	6883	2710	1980	1667	155	155	155
2xAssociative	11925	7061	2776	470	211	186	155	155
4xAssociative	11859	7122	2656	518	206	173	162	153
8xAssociative	11962	7348	2781	500	243	179	165	157
Full Associative	11962	7411	2879	476	250	206	181	163

insertion sort

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	8070	5112	2345	1939	1885	116	116	116
2xAssociative	9152	4334	1581	336	167	157	116	116
4xAssociative	10038	4122	1422	327	191	135	129	115
8xAssociative	9933	4128	1409	356	192	146	128	121
Full Associative	9933	4136	1523	378	216	151	136	124

latice_init

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	18327	8219	4227	2734	1908	1521	1325	1233
2xAssociative	17965	9230	4201	2728	1521	1316	1228	1192
4xAssociative	17734	9897	4391	2658	1680	1282	1209	1154
8xAssociative	17587	10602	4536	2714	1706	1316	1195	1154
Full Associative	17587	10942	5164	2250	1486	1316	1235	1183

matrix_mult

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	431	294	139	134	67	67	67	67
2xAssociative	489	245	120	75	68	66	66	66
4xAssociative	500	263	120	89	69	65	65	65
8xAssociative	518	261	103	86	68	66	67	65
Full Associative	518	221	119	97	82	69	67	67

me_ivlin

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	29971	19164	8465	644	110	110	110	110
2xAssociative	28356	12779	5961	552	126	110	110	110
4xAssociative	30240	13224	5611	568	137	114	109	109
8xAssociative	32389	13796	5884	460	150	119	108	108
Full Associative	32389	14175	5873	519	172	123	116	112

qick sort

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	1307	994	457	244	99	77	76	76
2xAssociative	1267	986	419	161	119	79	76	76
4xAssociative	1258	986	404	144	91	84	76	75
8xAssociative	1268	967	389	150	95	83	76	80
Full Associative	1268	973	419	180	122	107	95	92

selection sort

CacheSize	64	128	256	512	1024	2048	4096	8192
Direct Mapping	10575	6700	3194	1779	1733	109	109	109
2xAssociative	11075	5378	2953	234	134	127	108	108
4xAssociative	10451	5037	2140	271	144	116	112	107
8xAssociative	11013	4996	2039	235	152	120	109	111
Full Associative	11013	4821	2130	263	177	135	130	120

Speicherbedarf der Benchmarks

Scratch-Pad (alle Angaben in Byte)

Scratch-Pad Size			0	64	128	256	512	1024	2048
biquad_N_section	Programm	On-Chip	0	40	120	192	192	192	192
		Off-Chip	192	168	80	16	16	16	16
	Daten	On-Chip	0	0	0	0	0	0	0
		Off-Chip	156	156	156	156	156	156	156
bubble sort	Programm	On-Chip	0	52	132	200	200	192	192
		Off-Chip	196	168	64	4	4	4	4
	Daten	On-Chip	0	0	0	0	0	408	408
		Off-Chip	436	436	436	428	428	28	28
heap sort	Programm	On-Chip	0	56	112	212	512	540	540
		Off-Chip	544	524	508	452	48	4	4
	Daten	On-Chip	0	0	0	0	0	408	408
		Off-Chip	436	436	436	436	436	28	28
insertion sort	Programm	On-Chip	0	68	128	248	248	240	240
		Off-Chip	244	200	180	4	4	4	4
	Daten	On-Chip	0	0	0	0	0	408	408
		Off-Chip	440	440	440	432	432	32	32
lattice	Programm	On-Chip	0	56	124	236	332	332	332
		Off-Chip	340	300	236	140	8	8	8
	Daten	On-Chip	0	0	0	0	0	0	0
		Off-Chip	9670	9670	9670	9670	9670	9670	9670

matrix_mult	Programm	On-Chip	0	52	104	120	356	356	356
		Off-Chip	356	312	292	296	0	0	0
	Daten	On-Chip	0	0	0	0	0	0	0
		Off-Chip	216	216	216	216	216	216	216
Me_ivlin	Programm	On-Chip	0	64	128	256	512	820	836
		Off-Chip	836	796	756	696	536	16	0
	Daten	On-Chip	0	0	0	0	0	0	0
		Off-Chip	84	84	84	84	84	84	84
quick sort	Programm	On-Chip	0	44	72	156	392	392	392
		Off-Chip	404	392	392	360	12	12	12
	Daten	On-Chip	0	0	0	0	108	108	108
		Off-Chip	312	312	312	312	204	204	204
selection sort	Programm	On-Chip	0	48	128	180	180	172	172
		Off-Chip	176	168	60	4	4	4	4
	Daten	On-Chip	0	0	0	0	0	28	28
		Off-Chip	436	436	436	428	428	408	408

Cache

Benchmarks	biquad	bubble sort	heap sort	insertion sort	lattice	matrix _mult	me_ivlin	quick sort	selection sort
Programm	164	196	544	244	340	356	836	404	176
Daten	144	440	440	440	9676	216	88	316	440