technische universität
dortmund

Bachelorarbeit

Evaluierung des Energieverbrauchs von modernen Multi-Core-Systemen in Bezug auf das Speicherzugriffsmuster

Nikolaus Madalinski
Mai 2015

Gutachter:
Prof.Dr. Teubner
Dipl.-Inf. Breß

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl VI
http://dbis.cs.tu-dortmund.de

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

2.1 Hauptkomponenten des Computersystems 3

2.1.1 Prozessor 4

2.1.2 Speicher 6

2.2 Architektur moderner Multiprozessorsysteme 12

2.2.1 Uniform Memory Access - Architektur 13

2.2.2 Non Uniform Memory Access - Architektur 13

2.3 Multithreading 15

3 Programm zur Durchführung der Speicherzugriffe **17**

3.1 Grundidee des Programms 17

3.2 Konfiguration der Speicherzugriffe 18

3.3 Bestandteile des Programms 20

3.4 Einbinden der 'Intel-PCM' Bibliothek 21

4 Analyse der Messergebnisse **23**

4.1 Experimentelle Systemumgebung 23

4.2 Energiebedarf des Prozessors 23

4.2.1 Auswirkung unterschiedlicher Stride-Werte 25

4.2.2 Auswirkung des zufälligen Zugriffsmusters 27

4.2.3 Zugriff auf lokalen und entfernten Speicher 29

4.2.4 Zusammenfassung der Ergebnisse 32

4.3 Energiebedarf des Hauptspeichers 33

4.3.1 Auswirkung der Anzahl an Hauptspeicherzugriffen 34

4.3.2 Untersuchung der durchgehenden Anforderung des Hauptspeichers . 36

4.3.3 Zugriff auf lokalen und entfernten Speicher 37

4.3.4 Zusammenfassung der Ergebnisse 40

4.4 Auswertung von Multithreading 40

4.4.1 Einfluss auf den Energieverbrauch des Prozessors 40

4.4.2 Einfluss auf den Energieverbrauch des Hauptspeichers 42

5 Zusammenfassung und Ausblick **45**

Abbildungsverzeichnis **48**

Literaturverzeichnis **50**

Erklärung **50**

Kapitel 1

Einleitung

Seit Beginn des Computerzeitalters begleitet uns die permanente Steigerung der Rechenleistung. Die zentralen Schaltelemente der ersten elektronischen Rechner waren die Elektronenröhren, welche einen hohen Energiefluss verursachten. Mit der Einführung des Transistors wurde der Energieverbrauch stark reduziert, was zur Folge hatte, dass die Computerindustrie das Interesse an Energiesparmaßnahmen verlor [1].
In der heutigen Zeit nimmt das Thema Energie wieder eine wichtige Rolle ein. Auch im Bereich der Informations- und Kommunikationstechnik, welcher nach einer Studie im Jahr 2007 etwa 10,5% des gesamten Stromverbrauchs in Deutschland ausmachte, werden energieeffiziente Optimierungen angestrebt. In einem Computersystem ist der Energieverbrauch von mehreren Komponenten abhängig. Dabei machen der Prozessor und der Speicher in modernen Multi-Prozessor-Systemen 41% des Energieverbrauchs aus und stellen zugleich die Hauptkomponenten eines Computersystems dar [2].
Das Ziel dieser Bachelor-Thesis ist es, den Energieverbrauch unterschiedlicher Speicherzugriffe in modernen Multi-Prozessor-Systemen zu messen und zu analysieren. Durch die Evaluierung sollen energieeffiziente Zugriffsmuster erkannt werden, damit diese zukünftig bei der Programmierung von Anwendungen genutzt werden können.

Beginnend mit den Grundlagen werden in Kapitel 2 der Aufbau und die Funktionsweise des Prozessors sowie des Speichers erläutert. Des Weiteren wird die Speicherverwaltung vorgestellt und die Charakteristika moderner Multi-Prozessor-Systeme aufgeführt.
Kapitel 3 beschäftigt sich mit dem Programm, welches für die Ausführung der unterschiedlichen Speicherzugriffe eingesetzt wurde, und beschreibt das Einbinden weiterer Programmfunktionen, durch die das Messen der Energie ermöglicht wurde.
In Kapitel 4 werden die Messergebnisse von verschiedenen Speicherzugriffsmustern vorgestellt und der Energiebedarf des Prozessors und des Speichers analysiert.
Abschließend werden die Ergebnisse im letzten Kapitel zusammengefasst und Möglichkeiten zur Energieeinsparung benannt.

Kapitel 2

Grundlagen

In diesem Kapitel werden die wichtigsten Grundlagen eines Computersystems erläutert. Die Beschreibung der wesentlichen Komponenten eines Computers und dessen Arbeitsweise soll zum besseren Grundverständnis beitragen. Weiterhin werden die Charakteristika moderner Multi-Prozessor-Systeme sowie Bezüge zum Thema Energie im Computersystem aufgezeigt.

2.1 Hauptkomponenten des Computersystems

Das Grundkonzept der Rechnerarchitektur wurde bereits in den 1940er Jahren von dem Mathematiker John von Neumann entwickelt und dient heute noch unter der Bezeichnung Von-Neumann-Architektur als Referenzmodell für Computer. Ein Von-Neumann-Rechner setzt sich aus den in Abbildung 2.1 dargestellten Grundbestandteilen zusammen.

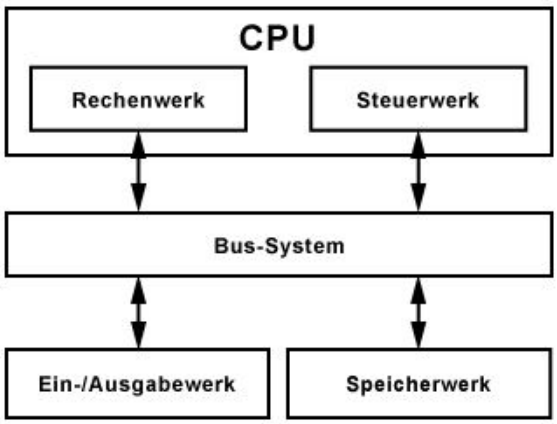


Abbildung 2.1: Modell einer Von-Neumann-Architektur [3]

Das Rechenwerk und das Steuerwerk bilden gemeinsam die zentrale Recheneinheit (engl. *Central Processing Unit*, kurz CPU) und sind über ein Bussystem mit dem Speicherwerk und dem Ein-/Ausgabewerk verbunden [3].

Das Ein- und Ausgabewerk steht für die Logik der Schnittstellen, die für eine Kommunikation mit dem Benutzer notwendig sind. Im weiteren Verlauf wird nicht weiter auf das Ein-/Ausgabewerk eingegangen. Stattdessen wird die Funktionsweise und der Aufbau der CPU und des Speichers näher erläutert.

2.1.1 Prozessor

Im Mittelpunkt eines jeden Computersystems steht die CPU. Die primäre Aufgabe dieser Recheneinheit ist die Ausführung von Befehlen und der damit verbundenen Ablaufsteuerung.

Aufbau der CPU

Für die Durchführung dieser beiden Arbeitsabläufe ist die CPU in zwei Einheiten organisiert. Abbildung 2.2 veranschaulicht die dafür eingesetzten Bestandteile. Der Datenprozessor fungiert dabei als Recheneinheit. Die Aufgabe des Datenprozessors ist die Verarbeitung eingehender Daten. Dies wird unter anderem mit Hilfe der Arithmetisch-Logischen Einheit (*Arithmetic Logical Unit*, kurz ALU) realisiert. Die ALU ist in der Lage arithmetische

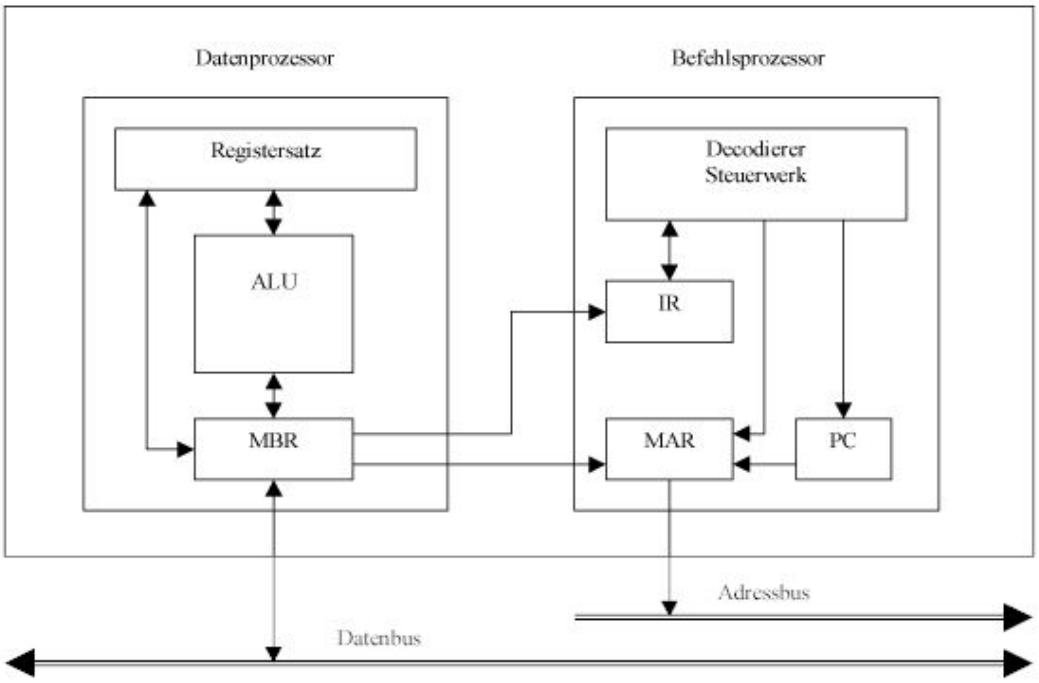


Abbildung 2.2: Arbeitsweise der CPU [4]

und logische Funktionen zu berechnen. Zur Aufnahme der dafür benötigten Operanden bedient sie sich an der unmittelbar zur Verfügung stehenden Registern. Zusätzlich steht dem Datenprozessor das Memory Buffer Register (MBR) zur Verfügung, welches der Kommunikation mit dem Speicher dient.

Des Weiteren besteht die CPU aus einem Befehlsprozessor, der für die Ablaufsteuerung zuständig ist. Ebenso wie der Datenprozessor bedient sich der Befehlsprozessor an zusätzlichen Registern. In diesem Fall sind es zum einen das Befehlsregister (IR), zum Festhalten des aktuell bearbeiteten Befehls, und das Speicheradressregister (MAR) zur Speicherung der als nächstes anzusprechenden Adresse. Außerdem kommt der Befehlszähler (PC), der die Adresse des nächsten auszuführenden Befehls enthält, zum Einsatz. Mit Hilfe dieser Register ist es dem Dekodierer und dem Steuerwerk möglich, die zu verarbeitenden Befehle zu entschlüsseln beziehungsweise die Befehlsausführungen zu steuern [5].

Ausführung von Befehlen

Wie bereits mit der Struktur einer CPU beschrieben, ist die CPU für die Verarbeitung von Befehlen und Daten zuständig. Da innerhalb des Computersystems eine große Anzahl an Daten und Befehlen gespeichert wird, muss dem Prozessor mitgeteilt werden, welche dieser Daten und Befehle verarbeitet werden sollen. Realisiert wird dies mit der Ausführung von Programmen. Ein Programm enthält eine Folge von Befehlen, die von dem Prozessor zu verarbeiten sind. Die Ausführung dieser Befehle erfolgt in der Reihenfolge, in der sie im Speicher hintereinander abgelegt sind beziehungsweise in der Reihenfolge, die durch den Programmierer bestimmt wird. Dazu wird der Inhalt des Befehlszählers um die entsprechende Anzahl der Bytes erhöht, die für die nächste Instruktion erforderlich sind [5].
In Abbildung 2.3 wird die Kommunikation zwischen dem Prozessor und den Speicherdaten veranschaulicht. Zunächst teilt das ausgeführte Programm der CPU mit, welches Datum

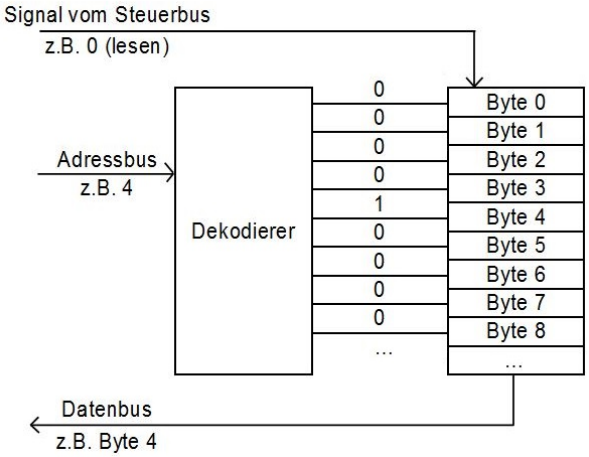


Abbildung 2.3: Adressierung von Speicherzellen

für den nächsten Befehl benötigt wird, und gibt die Adresse des Datums bekannt. Die CPU legt diese Speicheradresse dann auf den Adressbus. Der Dekodierer, welcher zwischen dem Prozessor und dem Speicher platziert ist, wandelt die eingehende Speicheradresse um und zeigt auf die gewünschten Daten. Gleichzeitig übermittelt die CPU über den Steuerbus, ob

der Inhalt der Speicheradresse lediglich ausgelesen oder ob ein Inhalt in die Speicheradresse geschrieben werden soll. Anschließend stellt der Datenbus die Verbindung zwischen der CPU und der gewünschten Speicherzelle her und die entsprechenden Daten werden zur zentralen Recheneinheit transportiert und dort weiterverarbeitet.

2.1.2 Speicher

Der Speicher repräsentiert die Funktionseinheit, die Daten und Befehle aufnimmt, aufbewahrt und bereitstellt. Insbesondere stellt der Speicher Daten und Befehle zur Verfügung, die für die Ausführung von Programmen benötigt werden. Folglich nimmt der Speicher eine zentrale Stellung ein.
In der Anfangszeit der Computerentwicklung wurden die verschiedenen Komponenten eines Computers, wie zum Beispiel die CPU und der Speicher, gemeinsam entwickelt. Dies wirkte sich auf eine ausgeglichene Leistung der einzelnen Bauteile aus. Mit dem Erreichen eines festen Konzepts des Von-Neumann-Rechners änderte sich dieses Vorgehen und die Bauteile wurden seitdem getrennt voneinander weiterentwickelt. Im Gegensatz zum Speicher stieg die Rechenleistung des Prozessors besonders schnell an, sodass dies zu immer höheren Latenzzeiten des Speichers führte [6].
In heutigen Computersystemen wird das Speichersystem in verschiedene Ebenen aufgeteilt, um die Antwortzeiten des Speichers zu reduzieren.

Speicherhierarchie

Der Datenspeicher sollte idealerweise sowohl die Eigenschaft der schnellen Zugriffsgeschwindigkeit als auch die der großen Kapazität erfüllen. Ein solcher Speicher existiert jedoch nicht [1]. Um dennoch der hohen Geschwindigkeit des Prozessors gerecht zu werden und die Daten für den Prozessor so effizient wie möglich zur Verfügung zu stellen, wird das Speichersystem hierarchisch in Speicherebenen aufgebaut.

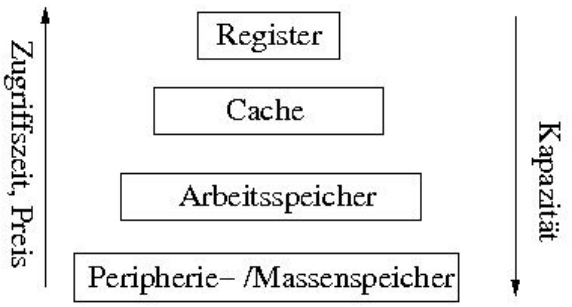


Abbildung 2.4: Speicherhierarchie [7]

Eine typische Speicherhierarchie ist strukturiert in Register, Cache, Arbeitsspeicher und dem Peripherie-/Massenspeicher. Mit wachsender Entfernung von der CPU sinkt die Zugriffsgeschwindigkeit der, in Abbildung 2.4 dargestellten, Speicherschichten und die Speicherkapazität nimmt zu. Demnach können je nach Einsatzbereich die unterschiedlichen Anforderungen besser erfüllt werden.
Die Register, welche bereits in Abbildung 2.2 vorgestellt wurden, sind der CPU unmittelbar zugeordnet und stellen die oberste Speicherebene dar. Register verfügen zwar über die geringste Speicherkapazität innerhalb des Computersystems, arbeiten dafür jedoch in der gleichen Geschwindigkeit wie die CPU. Der Cache, auch Cache-Speicher genannt, dient als Puffer-Speicher und arbeitet zwischen den schnellen Registern und dem relativ langsamen Arbeitsspeicher. Die genaue Arbeitsweise des Caches wird nachfolgend näher betrachtet.
Der Arbeitsspeicher stellt den Haupt-Systemspeicher dar und wird auch Hauptspeicher oder RAM genannt. RAM steht für den Speichertyp *Random Access Memory*, mit dem der Hauptspeicher realisiert wird. Der Hauptspeicher hat die Aufgabe, sowohl alle Daten als auch alle Befehle zu speichern, die für den Ablauf aller auszuführenden Programme benötigt werden. Auf der untersten Speicherebene, wie in Abbildung 2.4 dargestellt, befindet sich der Massenspeicher, der zur Aufbewahrung von Daten zur späteren Bearbeitung dient. Programmbefehle, die auf diesem Datenspeicher abgelegt sind, können nicht direkt ausgeführt werden. Stattdessen müssen die Daten aus dem Massenspeicher zunächst in den Hauptspeicher transportiert werden und können erst dann von dem Prozessor verarbeitet werden.

Aufbau des Speichers

Ein Speicher besteht aus vielen Speicherzellen, die in einer Matrix aus Zeilen und Spalten angeordnet sind. Jede einzelne Speicherzelle kann mit Hilfe eines Bits die Information *0* oder *1* speichern. Dabei werden mehrere Bits zusammengefasst und stellen gemeinsam ein Speicherwort dar. In modernen Computerarchitekturen setzt sich das Speicherwort meist aus 64 aufeinanderfolgenden Bits zusammen. Dies spiegelt sich in der Größe der CPU-Register wider, da der Prozessor keine einzelnen Bits, sondern stets ganze Speicherwörter verarbeitet.
Die hauptsächliche Ursache für unterschiedliche Zugriffsgeschwindigkeiten der verschiedenen Speicherebenen ist der Einsatz unterschiedlicher Speichertechnologien. Sowohl der Hauptspeicher als auch der Cache werden üblicherweise mit RAM, einem Datenspeicher mit “wahlfreiem Zugriff“, realisiert. Dies bedeutet, dass jede Speicherzelle einzeln adressierbar ist. Unterschiedlich ist jedoch die Art der Speicherzelle, die im Hauptspeicher und im Cache-Speicher eingesetzt wird.
Für den Hauptspeicher wird zumeist die in Abbildung 2.5 dargestellte Speicherzelle verwendet. Die Realisierung eines Speichers mit dieser Art von Speicherzellen nennt sich *Dynamic*

Random Access Memory, kurz DRAM.

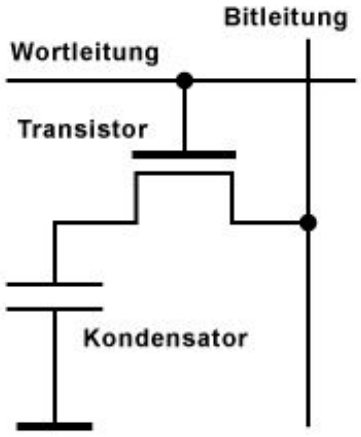


Abbildung 2.5: Aufbau einer dynamischen Speicherzelle [3]

Die DRAM-Speicherzelle besteht aus einem Transistor und einem Kondensator. Der Kondensator ist das eigentliche Speicherelement. Dabei spiegelt ein geladener Kondensator den Inhalt "1" und ein entladener Kondensator die Information "0" wider. Der Nachteil dieser Speicherzelle ist, dass sowohl der Lese-, als auch der Speichervorgang eine gewisse Zeit beanspruchen. Grund dafür ist die damit zusammenhängende Messung der Spannung am Speicherkondensator beziehungsweise die Aufladung der Spannung für den Speichervorgang. Weiterhin kommt es gelegentlich durch Leckströme zur Entladung des Kondensators, wodurch die DRAM-Speicherzellen in regelmäßigen Abständen aufgefrischt werden müssen. Auch dieser Vorgang führt zu einer zeitlichen Verzögerung [6].
Die im Cache eingesetzten Speicherzellen bestehen aus mehreren Transistoren.

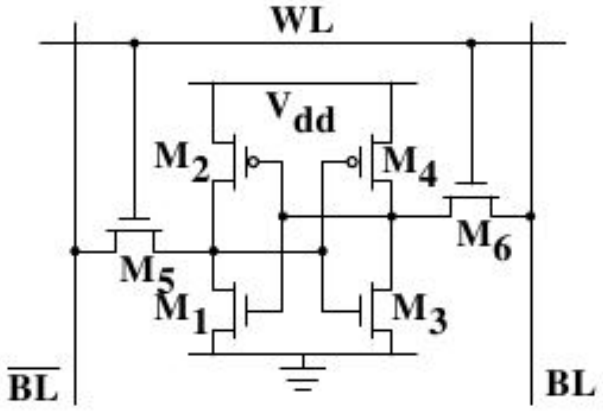


Abbildung 2.6: Aufbau einer statischen Speicherzelle [6]

In Abbildung 2.6 wird der Aufbau einer statischen Speicherzelle veranschaulicht. Dabei sind die Transistoren in der Abbildung mit "M" gekennzeichnet. Das besondere Merkmal dieser Schaltung ist, dass der Speicherinhalt auch nach Abruf des Speicherwortes erhalten bleibt.

Diese Speicherart wird daher *Static Random Access Memory*, kurz SRAM, genannt. Speicher mit dieser Eigenschaft bieten eine weitaus schnellere Zugriffszeit als Speicher, die aus DRAM-Speicherzellen aufgebaut sind. Der Nachteil liegt allerdings in der aufwendigeren Herstellung und dem größeren Strom- und Platzbedarf, da in der Regel sechs Transistoren verwendet werden.

Funktion des Cache-Speichers

Wie aus der Speicherhierarchie hervorgeht, ist der Cache-Speicher ein schneller Puffer-Speicher, der zwischen dem Prozessor und dem Hauptspeicher angeordnet ist. Der Cache hat die Aufgabe, bereits abgerufene Daten zwischenzuspeichern, sodass eine Wiederverwendung dieser Daten die Anzahl der langsamen Zugriffe auf den Hauptspeicher reduziert. Die Funktionsweise des Cache-Speicher ist in Abbildung 2.7 dargestellt.

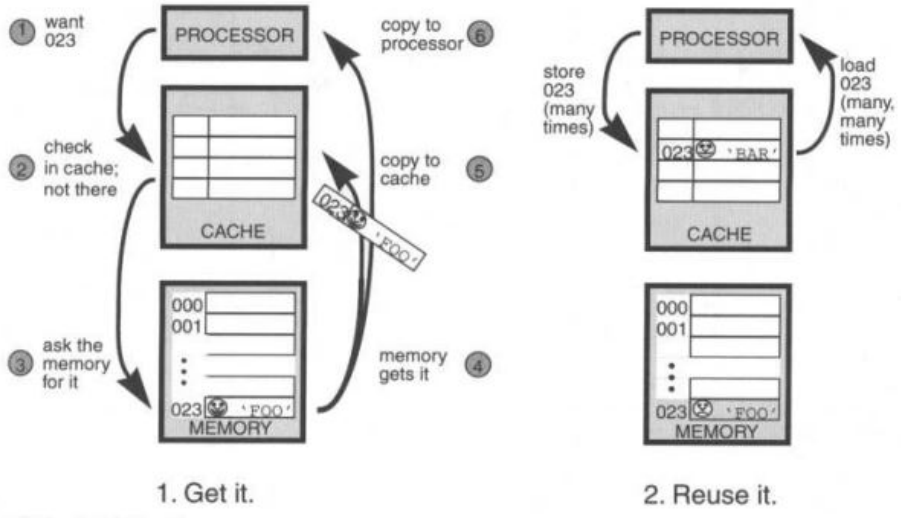


Abbildung 2.7: Das Zwischenspeichern und Wiederverwenden eines Datums im Cache [8]

Der Zugriff auf benötigte Daten aus dem Speicher beginnt, wie zuvor erläutert, indem die CPU die betreffende Adresse auf den Adressbus legt. Die Steuerlogik des Caches fängt diese Anfrage ab und überprüft anhand der Adresse, ob das angeforderte Speicherwort im Cache vorhanden ist. Ist dies der Fall, so liegt ein Cache-Hit vor und der Prozessor kann direkt mit dem Speicherwort aus dem Cache weiterarbeiten. Ist die gewünschte Information nicht im Cache vorhanden ist, so wird dies als Cache-Miss bezeichnet. Bei jedem Cache-Miss wird der ganze Speicherblock, der die entsprechende Speicheradresse enthält, in den Cache gespeichert.
Der Speicherblock entspricht dabei der Größe einer sogenannten Cache-Line, der kleinsten Verwaltungseinheit im Cache. Eine Cache-Line besteht aus mehreren Speicherzellen, sodass stets eine bestimmte Anzahl an Speicherzellen in den Cache zwischengespeichert wird.

Da der Cache eine geringere Speicherkapazität als der Hauptspeicher aufweist, wird beim Erreichen der Speicherkapazität eine Cache-Line entfernt und durch einen neuen Block aus dem Hauptspeicher ersetzt. Die Ersetzungsstrategie ist von System zu System verschieden und wird aus diesem Grund nicht weiter beschrieben.
Obwohl der Cache eine geringere Speicherkapazität als der Hauptspeicher aufweist, ist die Wahrscheinlichkeit auf einen Cache-Hit hoch. Der Cache arbeitet nach den Prinzipien der räumlichen und zeitlichen Lokalität. Weiter verbreitet sind dafür die Begriffe *Spatial Locality* und *Temporal Locality* [9]. Es wird davon ausgegangen, dass nach einem Zugriff auf ein Speicherwort mit großer Wahrscheinlichkeit auch auf die direkt darauffolgenden Speicheradressen zugegriffen wird. Dies liegt an der Art und Weise wie Programme entwickelt werden. Meist setzen sie sich aus vielen Schleifen zusammen und ein Großteil der Instruktionen wird in aufeinanderfolgenden Speicherzellen gespeichert.
Das *Prefetching* ist eine weitere Technik, mögliche Wartezyklen des Prozessors zu reduzieren. Prefetching bezeichnet das vorausgreifende Laden von Daten in den Cache. Die einfachste und in den meisten Computersystemen implementierte Methode des Prefetching ist das *Sequential read-ahead*. Dabei wird zu dem aktuell angeforderten Speicherblock zusätzlich auch der nachfolgende Speicherblock aus dem Hauptspeicher in den Cache zwischengespeichert. Das Ziel des Prefetching ist, dass zusammenhängende Daten von bereits benötigten Speicheradressen im Cache präsent sind und dadurch der Zugriff auf den langsameren Hauptspeicher nicht nötig ist. Neben der Methode des *Sequential read-ahead*, haben Entwickler die Möglichkeit, Stellen manuell für das vorausgreifende Laden von Daten zu kennzeichnen [10].

Aufteilung in mehrere Cache-Level

Moderne Computersysteme arbeiten in der Regel mit mehr als einem Cache-Speicher. Der Cache wird in sogenannte Cache-Level aufgeteilt. Üblich sind drei dieser Cache-Schichten. Der L1-Cache und meist auch der L2-Cache sind im Prozessorchip integriert und arbeiten mit dessen Taktgeschwindigkeit. Aufgrund des Platzmangels fallen die Speicherkapazitäten des L1- und L2-Caches sehr gering aus. Der L3-Cache befindet sich zwischen dem L2-Cache und dem Hauptspeicher. Dieser hat eine etwas größere Speicherkapazität als die der L1- und L2-Caches und weist zugleich eine minimal höhere Zugriffszeit auf.

Cache Organisationsarten

Cache-Lines können in sogenannte *Sätze* zusammengefasst werden. Je nach Organisationsform unterscheiden sich die Cache-Speicher in der Anzahl dieser Sätze. Für jede Speicheradresse aus dem Hauptspeicher ist ein bestimmter Satz innerhalb des Caches zuständig. Caches können auf drei verschiedene Weisen organisiert sein.

direkt abgebildet

Bei dieser Organisationsform repräsentiert jede Cache-Line einen eigenen Satz und jedem Speicherblock aus dem Hauptspeicher ist exakt eine Cache-Line zugewiesen. Wie in Abbildung 2.8 veranschaulicht, stehen bei einer Cache-Kapazität von n Cache-Lines und einem Hauptspeicher mit b Speicherblöcken jedem Speicherblock die Cache-Line $b \ mod \ n$ zur Verfügung.

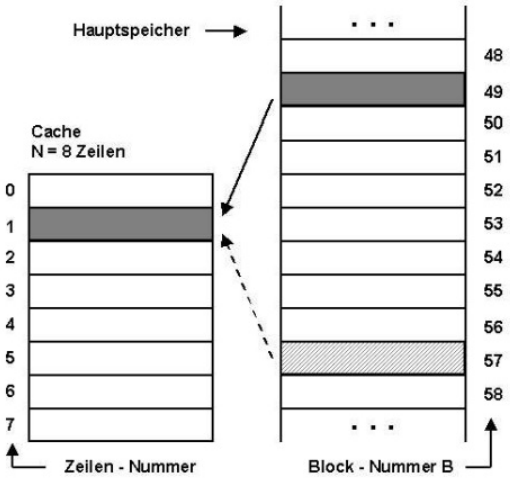


Abbildung 2.8: Direkt-abgebildeter Cache [11]

vollassoziativ

Der vollassoziative Cache spiegelt die gegensätzliche Orgianisationsform wider. Hierbei repräsentieren alle Cache-Lines einen gemeinsamen Satz. Demnach kann jede Speicheradresse aus dem Hauptspeicher in jeder Cache-Line gespeichert werden. Vorteil dieser Methode ist, dass Cache-Lines erst dann ersetzt werden müssen, wenn alle Blöcke des Caches belegt sind. Der Nachteil besteht darin, dass bei einer Anfrage der gesamte Cache durchsucht werden muss.

satzassoziativ

Der satzassoziative Cache stellt eine Mischform der beiden vorangehenden Organisationsformen dar. Diese Art von Cache-Speicher wird auch als n-fach assoziativer Cache bezeichnet, weil je n Cache-Lines zu einem Satz zusammengefasst werden, siehe Abbildung 2.9.

Zugriffsmuster

Der Zugriff auf ein Speicherwort wird differenziert in Lese- oder Schreibzugriff. Bei einem Lesezugriff werden die Daten aus dem Speicher angefordert und in der Recheneinheit verarbeitet. Für den Vorgang des Schreibzugriffs wird das entsprechende Speicherwort auf

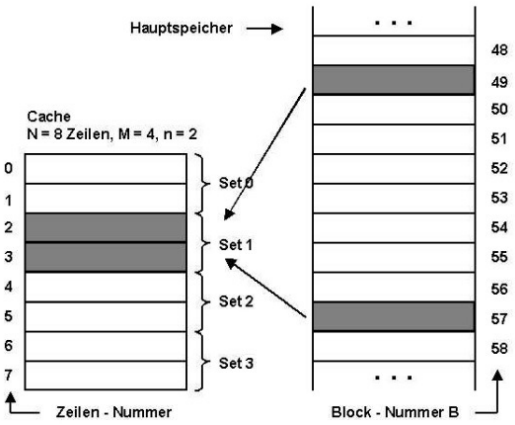


Abbildung 2.9: 2-fach assoziativer Cache [11]

gleiche Weise angesprochen. Zusätzlich wird die zu schreibende Information über den Datenbus übermittelt. Das Schreiben führt zeitweise zu einer Inkonsistenz der Daten, da die Inhalte des Caches und des Hauptspeichers nicht gleichzeitig aktualisiert werden können. Je nach Schreibstrategie werden die aktualisierten Informationen entweder direkt von einer zur nächsten Speicherschicht weitergegeben oder erst beim erneuten Zugriff auf ein veraltetes Datum korrigiert. Beide Schreibstrategien sorgen dafür, dass der Prozessor stets mit den neu geschriebenen Inhalten weiterarbeitet [12].
Eine weitere Eigenschaft des Speicherzugriffs ist die Reihenfolge, in der auf die Daten zugegriffen wird. Hier wird unterschieden in sequentiellen und wahlfreien Zugriff. Der sequentielle Speicherzugriff bezeichnet die Zugriffsart, bei der auf die Daten aufeinanderfolgend zugegriffen wird. Im Gegensatz dazu erfolgen die Zugriffe bei dem wahlfreien Zugriffsmuster in einer beliebigen Reihenfolge.

2.2 Architektur moderner Multiprozessorsysteme

Neben dem Ansatz der zuvor beschriebenen Speicherhierarchie, kann die Leistungsfähigkeit von Computersystemen besonders durch eine parallele Ausführung von Programmen gesteigert werden. Der Prozessorkern ist physikalisch begrenzt, sodass eine echte Hardware-Parallelisierung ein wesentliches Konzept zur Leistungssteigerung darstellt. Multiprozessorsysteme bestehen nicht nur aus einem, sondern aus mehreren Prozessoren. Zudem unterscheiden sich Multiprozessorsysteme in der Art, wie sie auf den Speicher zugreifen. Am meisten verbreitet ist das Speicherkonzept mit einem gemeinsamen Speicher, welches anhand der UMA (*uniform memory access*)- und NUMA (*non-uniform memory access*)-Architektur näher erläutert wird.

2.2.1 Uniform Memory Access - Architektur

Das einfachste Konzept der Multiprozessorsysteme, bei dem sich mehrere Prozessoren den Zugriff auf ein gemeinsames RAM teilen, basiert auf dem Modell der UMA-Architektur. UMA-Multiprozessoren haben die Eigenschaft eines gleichmäßigen Speicherzugriffs. Wie in Abbildung 2.10 dargestellt, benutzen mehrere Prozessoren einen einzigen Bus, um mit dem Speicher zu kommunizieren. Daraus resultiert für alle Prozessoren eine einheitliche Zugriffszeit.

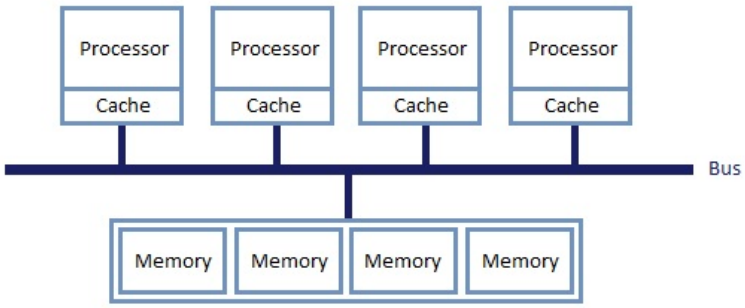


Abbildung 2.10: UMA-Multiprozessormodell [13]

Führt ein Prozessor einen Speicherzugriff durch, so ist die Busleitung zunächst von diesem Prozessor belegt und die weiteren CPUs müssen mit dem Zugriff auf den Speicher warten, bis die Busleitung wieder frei ist. Mit einer Erhöhung der Busbandbreite kann die gleichzeitige Nutzung der Busleitung auf mehr als einen Prozessorkern gesteigert werden. Die maximale Bandbreite einer Busleitung ist jedoch mit einer gewissen Anzahl an Prozessorkernen erreicht, sodass die UMA-Architektur nicht mit beliebig vielen Prozessorkernen realisierbar ist.

2.2.2 Non Uniform Memory Access - Architektur

Eine skalierbare Lösung für dieses Problem wird mit der NUMA-Architektur erreicht. Ebenso wie bei einer UMA-Architektur verfügen NUMA-Systeme über einen großen Adressraum, der von allen Prozessorkernen gemeinsam genutzt wird.
Wie die Abbildung 2.11 zeigt, setzt sich ein NUMA-Multiprozessorsystem aus mehreren UMA-Systemen zusammen. Mit diesem Konzept stehen jedem Prozessor sowohl ein lokaler Speicher als auch ein oder mehrere entfernte Speicher zur Verfügung. Der Nebeneffekt eines solchen Computersystems ist, dass im Vergleich zu dem UMA-Systemen die Speicherzugriffsgeschwindigkeit von der Lage des auszuführenden Prozessorkerns abhängt. Der Zugriff auf den lokalen Speicher eines Prozessorkerns ist wesentlich schneller als der Zugriff auf einen entfernten Speicher, da unter anderem eine zusätzliche Datenleitung für die Kommunikation genutzt wird. Für diese Datenleitung wird ein Verbindungskonzept verwendet, welches sich von der normalen Busleitung unterscheidet, sodass statt dem Be-

griff der Busleitung auch die Bezeichnung Interconnect verwendet werden kann. Je nach Prozessor-Hersteller variiert der Name dieser Datenleitung; Intel beispielsweise benutzt in diesem Zusammenhang den Begriff *QuickPath Interconnect*. Die einzelnen UMA-Systeme,

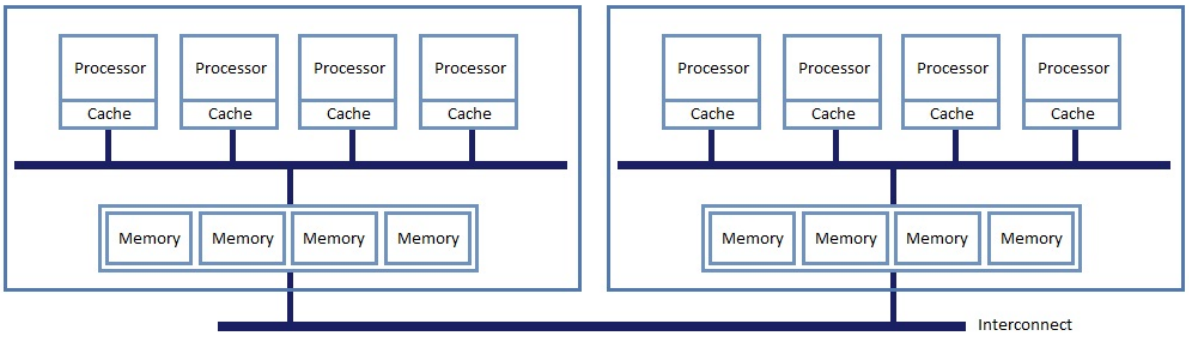


Abbildung 2.11: Beispiel eines NUMA-Systems mit zwei Knoten[13]

welche über die Interconnect-Leitung miteinander verbunden sind, werden auch als *Knoten* (oder engl. *nodes*) bezeichnet. Das in Abbildung 2.11 veranschaulichte NUMA-Modell besteht aus zwei Knoten mit jeweils vier CPUs und vier Speichermodulen. Unter der beispielhaften Annahme, dass der Datenspeicher des einen Knotens die Speicheradressen 1 bis 100.000 abbildet und der Datenspeicher des anderen Knotens die Speicheradressen 100.001 bis 200.000 enthält, kann jede der CPUs auf den gemeinsamen Adressraum von 1 bis 200.000 zugreifen.
Benötigt der Prozessor Daten, die im lokalen Speicher abgelegt sind, greift er über die direkte Verbindung mit Hilfe des Adress- und Datenbusses auf die entsprechende Speicheradresse zu. Ist die benötigte Speicheradresse auf dem entfernten Speicher hinterlegt, so muss der ausführende Prozessor die entsprechende Speicheradresse zunächst an einen Prozessor aus dem entferntem Knoten übergeben. Dieser greift dann über den Adress- und Datenbus auf das entsprechende Speicherwort zu, welches anschließend über die Interconnect-Verbindung an den ausführenden Prozessor zurückgegeben wird [14].
Wie zuvor anhand der Speicherhierarchie erläutert, werden auch bei Multiprozessorsystemen in jedem Prozessorkern meist der L1- und der L2-Cache integriert. Der L3-Cache wird innerhalb jedes Knotens von den jeweiligen Prozessorkernen gemeinsam genutzt. Auch die Sicherstellung der Cache-Kohärenz ist bei Multiprozessorsystemen für eine fehlerfreie Verwaltung der Speicherdaten entscheidend. Diese wird bei der NUMA-Architektur mit Hilfe eines verzeichnisbasierten Protokolls umgesetzt. Jeder Knoten besitzt ein sogenanntes Verzeichnis, in dem festgehalten wird, wo sich Kopien von lokalen Speicherblöcken befinden. Der Cache-Controller kann auf diese Verzeichnisse zugreifen und somit stets den aktuellen Inhalt eines jeden lokalen Speicherblocks entnehmen [15].

2.3 Multithreading

Neben der Parallelität der Hardware unterstützen die meisten moderne Computersysteme das Multithreading-Konzept. Dabei handelt es sich um eine Technik, die Wartezeiten des Prozessors mit der Verarbeitung anderer Prozesse überbrückt. Dementsprechend werden Programme in sogenannte *Threads* aufgeteilt, die jeweils einen eigenständigen Prozess darstellen. Führt die CPU solche Programme aus, so können die einzelnen Threads parallel von unterschiedlichen Prozessorkernen verarbeitet werden. Dies ermöglicht eine schnellere Abarbeitung der Programmbefehle. Zusätzlich wird die Leistungssteigerung durch das sogenannte *softwareseitige Multithreading* erhöht. Wird die Ausführung eines Programms, zum Beispiel aufgrund einer gleichzeitigen Ausführung mehrerer Programme, nur von einem einzigen Prozessorkern durchgeführt, so ist es auch einem einzelnen Prozessorkern möglich, mehrere Prozesse gleichzeitig zu verarbeiten. In diesem Fall führt der Prozessor die einzelne Befehle unterschiedlicher Threads im Wechsel aus. Dafür schaltet der Prozessor mit hoher Geschwindigkeit zwischen mehreren Threads hin und her. Dies hat den Vorteil, verursachte Wartezyklen eines Prozesses zu überbrücken [1].

Kapitel 3

Programm zur Durchführung der Speicherzugriffe

In diesem Kapitel wird die Anwendung, die zur Untersuchung unterschiedlicher Speicherzugriffe dient, näher erläutert. Es handelt sich dabei um ein Programm, das von Herrn Prof. Dr. Jens Teubner entwickelt und zur Verfügung gestellt wurde. Das Programm ist in der Programmiersprache C geschrieben und für die Leistungsbeurteilung moderner Multiprozessorsysteme konzipiert.
Implementiert ist bereits die Möglichkeit, unterschiedliche Speicherzugriffsmuster zu definieren und diese auszuführen. Des Weiteren kann die Zeit der durchgeführten Speicherzugriffe gemessen und anschließend ausgegeben werden.
Im Rahmen dieser Bachelor-Thesis ist die Messung des Energieverbrauchs von zentraler Bedeutung. Realisiert wird dies durch das Einbinden weiterer Methoden, welche durch die *Intel Performance Counter Monitor* Bibliothek frei zur Verfügung stehen [16]. Dazu zählen sowohl Methoden für die Messung der verbrauchten Energie, als auch Methoden, die das Verhalten des Caches aufzeichnen. Im Gegensatz zu dem Hauptprogramm, welches in C geschrieben ist, werden die Methoden der Intel-PCM Bibliothek in der Programmiersprache C++ bereit gestellt.
Zunächst wird der Aufbau des Programms und die Möglichkeit der Konfiguration verschiedener Speicherzugriffe erläutert. Anschließend wird beschrieben, welche Methoden für die Performancebeurteilung genutzt und wie diese eingebunden werden.

3.1 Grundidee des Programms

Das Konzept des zugrundeliegenden Programms basiert auf der Durchführung von kontrollierten Speicherzugriffen. Der Programmablauf sieht vor, dass in einer Konfigurationsdatei festgelegt wird, wie die Speicherzugriffe auszuführen sind. In dieser Datei wird unter anderem die Speichergröße, der Speicherort und das Zugriffsmuster angegeben. Die genauen

Konfigurationsmöglichkeiten werden im nächsten Abschnitt behandelt.
Die Ausführung des Programms beginnt mit dem Einlesen der Konfigurationsdatei. Entsprechend dieser Datei wird ein Speicherbereich angelegt und mit Inhalten, die das gewünschte Zugriffsmuster auslösen, gefüllt. Anschließend erfolgt die eigentliche Durchführung der Speicherzugriffe. Dafür iterieren die Prozessoren über das angelegte Speicherarray. Mit Hilfe der implementierten Funktionen wird das Verhalten des Computersystems auf die Speicherzugriffe aufgezeichnet und abschließend ausgegeben.

3.2 Konfiguration der Speicherzugriffe

Die XML-basierte Konfigurationsdatei ermöglicht es, mit unterschiedlichen Parametern, Speicherzugriffe zu definieren. Folgende Parameter können in der Konfigurationsdatei beliebig festgelegt werden und sind für die Durchführung des Programms erforderlich:

Speichergröße

```
<parameter name='size' values='1024'>
```

Der Parameter 'size' legt die Größe des Speicherblocks fest, über die im Laufe des Tests iteriert werden soll. Die Speicherkapazität wird in der Einheit Byte angegeben. Das Programm erlaubt hierbei auch die Eingabe mehrerer Speichergrößen, sodass mehrere Tests gleichzeitig definiert und vom Computersystem ausgeführt werden können. Beispielweise hat die Eingabe von *<parameter name='size' values='1024 131072 16777216'>* zur Folge, dass unabhängig voneinander die Speicherzugriffe für die Speicherarrays der Größe 1 kB, 128 kB und 16 MB ausgeführt und gemessen werden.

Abstand der Speicherdaten, auf die zugegriffen wird

```
<parameter name='stride' values='16 128 1024'>
```

Mit dem Parameter 'stride' wird definiert, auf welche Speicherelemente innerhalb der Speicherarrays zugegriffen wird. Beginnend bei der Stelle 0 im Speicherarray werden alle Speicherstellen, im Abstand von 'stride', mit Inhalt gefüllt und fungieren als Speicherzugriffselemente. Am Beispiel der Speichergröße (size) von 1024 Bytes und dem Abstand (stride) von 16 Bytes werden die Elemente an den Stellen 0, 16, 32, ... , 992 und 1008 mit Inhalten beschrieben und der Prozessor iteriert über diese Speicherdaten. Um das Cache-Verhalten der Speicherdaten nachvollziehen zu können, sollte darauf geachtet werden, dass innerhalb eines Durchlaufs stets auf dieselben Speicherelemente zugegriffen wird. Dies wird sichergestellt, indem als Wert für 'stride' maximal die Bytegröße des kleinsten Speicherarrays gewählt wird.
Auch bei diesem Parameter ist sowohl die Eingabe eines einzigen Wertes als auch die

Eingabe mehrerer Werte gleichzeitig möglich und entspricht jeweils einem separaten Test.

Ort und Zugriffsmuster des Speichers

```
<memory-regions>
    <region id='region0' numa-nodes='0' size='size' stride='
        stride' pattern='sequential' />
    <region id='region1' numa-nodes='1' size='size' stride='
        stride' pattern='random' />
</memory-regions>
```

Im Bereich *<memory-regions>...</memory-regions>* sind genauere Informationen zu den Speicherarrays anzugeben, über die im Anschluss iteriert wird. Diese Angaben werden von dem Programm eingelesen, um die gewünschten Speicherarrays mit den entsprechenden Eigenschaften anzulegen. Neben dem Namen *id*, welcher für die Referenzierung durch die Threads benötigt wird, werden die zuvor definierten Speichergrößen und der Wert für 'stride' mitgegeben. Wie im Abschnitt 2.2.2 erläutert, bestehen NUMA-Computersysteme aus mindestens zwei Knoten. Insbesondere spielt der Speicherort eine entscheidende Rolle. Dieser wird mit dem Attribut 'numa-nodes' angegeben. Darüber hinaus wird ebenso das Zugriffsmuster für jeden einzelnen Speicher definiert. Das Attribut 'pattern' akzeptiert die Werte *sequential* und *random*, welche das sequentielle und zufällige Zugriffsmuster repräsentieren.

Konfiguration der Threads

```
<threads>
  <thread id='thread0' numa-node='0' region='region1' mode=
      'read-only' />
  <thread id='thread1' numa-node='1' region='region0' mode=
      'read-only' />
</threads>
```

An dieser Stelle der Konfigurationsdatei werden abschließend die Threads spezifiziert. Dabei ist neben der Benennung der Threads anzugeben, auf welchem Knoten welche Threads auszuführen sind und auf welche entsprechenden Speicherbereiche die Zugriffe erfolgen sollen. So kann festgelegt werden, ob ein Thread auf seinen lokalen oder auf einen entfernten Speicher zugreifen soll.

Gesamt betrachtet stellt die Konfigurationsdatei das wesentliche Werkzeug für die Evaluierung des Energieverbrauchs zur Verfügung, da diese eine Simulation verschiedener Szenarien im Bezug auf den Speicherzugriff ermöglicht.

3.3 Bestandteile des Programms

In diesem Abschnitt werden die wesentlichen Funktionen des Programms kurz erläutert. Der Programmcode 3.1 zeigt die einzelnen Funktionen, die zusammen die Durchführung einer Zugriffs-Simulation einleiten.

```
static void run_experiment (setup_t *setup) {
    setup_worksets (setup);
    prepare_threads (setup);
    run_threads (setup);
        print_results (setup, stdout);
}
```

Listing 3.1: Ausführung der konfigurierten Speicherzugriffe

Die hier vorgestellte Methode *run_experiment* bekommt als Parameter den Inhalt der gesamten Konfigurationsdatei übergeben. Diese Methode wird innerhalb des Programms rekursiv aufgerufen, sodass für jede Kombination der festgelegten Speicherzugriffs-Parameter die darauf folgenden Funktionen separat angewendet werden. Somit beinhaltet der Parameter *setup* jeweils eine der festgelegten Speicherzugriffskonfigurationen. Anhand der jeweiligen Konfiguration werden die gewünschten Speicherarrays angelegt und durchlaufen. Zunächst legt die Funktion *setup_worksets(setup)* einen Speicherblock an. Der Parameter *setup* gibt der Funktion die nötige Information, auf welchem Knoten das Speicherarray angelegt werden soll. Anhand des ebenfalls übergebenen Zugriffsmusters werden in dem angelegten Speicherblock die Elemente, im Abstand von *stride*, mit Inhalten gefüllt. Als Inhalt wird dafür, wie in Abschnitt 2.1.2 erläutert, jeweils die Speicheradresse des Elementes geschrieben, welches von dem Prozessor als nächstes gelesen werden soll.
Für die weitere Vorbereitung der auszuführenden Speicherzugriffe sorgt die Funktion *prepare_threads(setup)*. Mit dieser Funktion werden die, in der Konfigurationsdatei festgelegten Threads, den entsprechenden Prozessorkernen zugewiesen.
Anschließend erfolgt der entscheidende Prozess mit der Funktion *run_threads(setup)*, bei dem die Prozessorkerne die zugewiesenen Threads ausführen. Die Prozessoren greifen dabei auf die Speicherdaten des angelegten Speicherarrays zu. Zusätzlich wird an einer anderen Stelle innerhalb des Programmcodes die Anzahl der Iterationen festgelegt. Der Zugriff auf den Speicher wird nach dem Erreichen der gewünschten Anzahl von Speicherzugriffen seitens des Programms beendet.

3.4 Einbinden der 'Intel-PCM' Bibliothek

Für das Auswerten der Speicherzugriffe sind weitere Methoden notwendig, die in das Programm eingebunden werden. Für die Evaluierung des Energieverbrauchs eignen sich besonders, wie bereits vorgestellt, die Funktionen aus der *Intel Performance Counter Monitor* - Bibliothek (kurz: Intel-PCM). In modernen Prozessoren sind sogenannte Counter integriert, die Informationen über Zustände verschiedener System-Komponenten enthalten. Mit Hilfe der Funktionen aus der Intel-PCM Bibliothek kann auf diese Counter zugegriffen und die Information ausgelesen werden.
Die Integration dieser Bibliothek in das hauptsächliche Programm bedarf einer Reihe von Anpassungen. Grund dafür ist die Programmiersprache, in der sich die Anwendungen unterscheiden. Die Intel-PCM Bibliothek wird in der Programmiersprache C++ bereitgestellt. Mit dem Einbinden dieser Bibliothek und der Integration der daraus benötigten Funktionen sind zusätzlich Änderungen am Build System erforderlich und die nicht kompatiblen Programmstellen von C müssen in C++ Syntax umgeschrieben werden. Abschließend wird das Programm mit der eingebundenen Intel-PCM Bibliothek mittels C++ Compiler umgewandelt und kann ausgeführt werden.

```
...
#include "intel/cpucounters.h"
PCM *m;
SocketCounterState *BeforeCounters, *AfterCounters;
SystemCounterState BeforeStateSystem, AfterStateSystem;
...
static void run_threads (setup_t *setup) {
        ...
        for(int i=0; i<m->getNumSockets(); ++i)
        BeforeCounters[i] = m->getSocketCounterState(i);
    BeforeStateSystem = m->getSystemCounterState();
        ...
        \\ Ausfuehrung der Threads
        ...
        for(int i=0; i<m->getNumSockets(); ++i)
        AfterCounters[i] = m->getSocketCounterState(i);
    AfterStateSystem = m->getSystemCounterState();
}
...
```

Listing 3.2: Auslesen der Counter

Die Verwendung der Methoden aus der vorgestellten Bibliothek wird durch das Aufnehmen der *cpucounter.h* Header-Datei ermöglicht. Anhand des kurzen Auszugs 3.2 des Programmcodes ist zu erkennen, dass zwei verschiedene Counter-Typen genutzt werden. Die deklarierten Variablen vom Typ *SocketCounterState* werden eingesetzt, um Informationen zu den einzelnen Knoten zu speichern. Zusätzlich werden zwei weitere Variablen genutzt, um den Zustand des gesamten Prozessorsystems festzuhalten. Diese sind vom Typ *SystemCounterState*. Um die entsprechenden Counter abfragen zu können, wird zunächst mit *PCM *m* ein Zeiger auf das Objekt der Klasse PCM definiert und an einer weiteren Stelle im Programm (main-Funktion) das entsprechende Objekt erzeugt.
In der Methode *run_threads* wird über die im System existierenden Knoten iteriert und deren Zustände, sowie der Zustand des gesamten Prozessorsystems, in den entsprechenden Variablen festgehalten. An dieser Stelle erfolgt die eigentliche Durchführung der zu messenden Speicherzugriffe. Wie zuvor beschrieben, werden in dieser Methode alle konfigurierten Threads ausgeführt. Nachdem alle Speicherzugriffe durchgeführt sind, werden die Zustände erneut ausgelesen und in den weiteren Variablen abgespeichert.
Im Anschluss an die vollständige Durchführung der Speicherzugriffe werden an einer weiteren Programmstelle die abgespeicherten Zustände mittels verschiedener Funktionen ausgewertet. Der Energieverbrauch des Hauptspeichers wird mit der Funktion *getDRAMConsumedJoules(BeforeStateSystem,AfterStateSystem)* ausgegeben und die verbrauchte Energie des Prozessorsystems mit der Funktion *getConsumedJoules(BeforeStateSystem,AfterState System)* (ohne Berücksichtigung der Energie des RAMs). Außerdem wird mittels weiterer Funktionen die Anzahl an L2- und L3-Cache Misses und Cache Hits ausgegeben, die auf den einzelnen Konten gemessen werden.

Kapitel 4

Analyse der Messergebnisse

4.1 Experimentelle Systemumgebung

Die Ausführung des vorgestellten Programms erfolgte auf einem Computersystem, welches die Eigenschaften eines modernen Rechensystems widerspiegelt. Die Verwendung der Intel-PCM Bibliothek ist auf Intel-Prozessoren beschränkt, sodass für diese Tests ein Intel-Prozessorsystem genutzt wird. Das zugrunde liegende System setzt sich aus zwei NUMA-Knoten zusammen, welche jeweils 8 physikalische Prozessorkerne beinhalten. Alle 16 Prozessorkerne sind identisch und tragen die Bezeichnung *Intel® Xeon® Processor E5-2690*. Bei den Prozessoren handelt es sich um eine 64-Bit-Architektur. Demzufolge sind der Datenbus, der Adressbus und die Register auf eine Breite von 64 Bit ausgelegt und der Prozessor kann 64 Bit gleichzeitig verarbeiten. Weiterhin ist der Cache in L1-, L2- und L3-Caches aufgeteilt, deren Cache-Line Größen jeweils 64 Bytes betragen. Die L1- und L2-Caches sind bei allen Prozessorkernen jeweils auf dem Prozessorchip integriert und 8-fach assoziativ organisiert. Der L3-Cache ist dahingegen 20-fach assoziativ organisiert und wird innerhalb jedes Knotens von den je 8 Prozessorkernen gemeinsam genutzt. Dabei sind die L1-Caches jeweils 32 Kb, die L2-Caches jeweils 256 Kb und die beiden L3-Caches jeweils 20 MB groß [17].

4.2 Energiebedarf des Prozessors

Für den Erhalt der ersten Messwerte konfigurierten wir das Programm mit 100 Millionen Speicherzugriffen und sequentiell lesendem Zugriffsmuster. Sowohl der Speicher als auch der auszuführende Thread ist auf numa-node = 1 gesetzt, sodass der Zugriff auf einen lokalen Speicher erfolgt. In Abbildung 4.1 ist die verbrauchte Energie der CPU für die ausgeführten Speicherzugriffe auf verschiedene Speichergrößen und mit einer Stride-Größe von 1024 Byte dargestellt.
Die erste Beobachtung ist der, mit zunehmender Speichergröße, steigende Energiever-

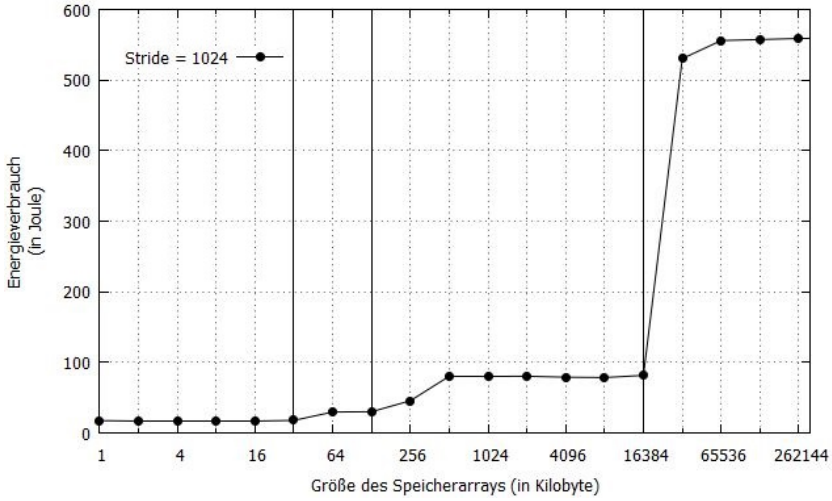


Abbildung 4.1: Energieverbrauch der CPU (Thread:1, Speicher:1, sequentieller Zugriff)

brauch. Die Energie steigt jedoch nicht proportional zur Erhöhung der Speicherkapazität, sondern weist eine stufenartige Steigerung auf. Bei einer geringen Speichergröße von 1 bis 32 Kb ist ein Energieverbrauch von etwa 17,51 Joule zu messen. Dieser steigt beim Erreichen der Speichergröße von 64 Kb auf etwa 29,71 Joule an. Erfolgt der Zugriff auf Speichergrößen zwischen 512 Kb und 16 MB, können bereits circa 80,24 Joule gemessen werden. Ein genauer Blick auf den Kurvenverlauf zeigt, dass sich die Energie genau in den Bereichen erhöht, bei denen die Speichergröße jeweils die Kapazität des L1-, L2- und L3-Caches übersteigt. Besonders auffällig ist der Anstieg des Energieverbrauchs auf circa 557 Joule bei Speichergrößen, welche die L3-Cache Kapazität überschreiten.
Verglichen mit den dazugehörigen Zugriffszeiten aus Abbildung 4.2, lässt sich für die Zugriffszeit ein identischer stufenartige Kurvenverlauf feststellen.

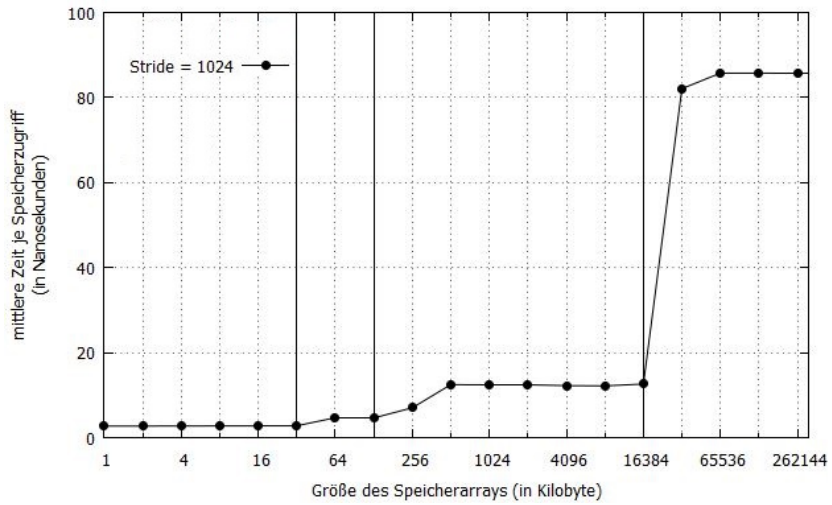


Abbildung 4.2: Zugriffszeit (Thread:1, Speicher:1, sequentieller Zugriff)

Als erste Erkenntnis ist somit festzuhalten, dass der Energieverbrauch der CPU von der Ausführungszeit des Threads abhängt.
Der stufenartige Kurvenverlauf der Zugriffszeit lässt sich, wie bereits angedeutet, mit der zugrunde liegenden Speicherorganisation erklären. Wie in Abschnitt 2.1.2 erläutert, ergeben sich mit der Verwendung tieferer Speicherschichten erhöhte Zugriffszeiten. Dazu weisen wir mit der gemessenen Anzahl an Cache-Hits nach, dass bei den jeweiligen Speichergrößen die Daten auf unterschiedlichen Speicherebenen zwischengespeichert werden.

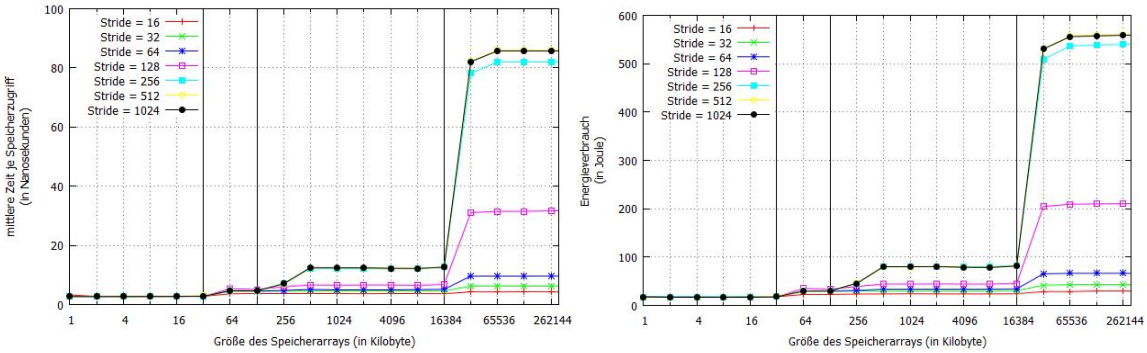
Speichergröße (in kB)	L2 Misses	L2 Hits	L3 Misses	L3 Hits
1	26.321	3.152	9.075	17.246
64	27.999	100.004.014	9.891	18.108
1.024	100.057.005	20.849	15.111	100.041.894
131.072	100.442.384	147.981	100.253.804	188.580

Tabelle 4.1: Cache-Verhalten für Stride = 1024 Byte (Thread:1, Speicher:1, sequentieller Zugriff)

Wie in Tabelle 4.1 zu erkennen, wurden die Daten bei der Ausführung der Speicherzugriffe im Falle des 1 Kb großen Speicherarrays stets im L1-Cache zwischengespeichert und der Zugriff auf die langsameren Speicherebenen verhindert. Zwar ist es mit den Methoden aus der Intel-PCM Bibliothek nicht möglich die L1-Cache-Hits zu messen, aber aus der geringen Anzahl der Cache-Misses sowie Cache-Hits im L2-Cache ist darauf zu schließen, dass die restlichen fast 100 Millionen Zugriffe einen L1-Cache-Hit verursachen. Des Weiteren ist aus der Tabelle zu entnehmen, dass beim Zugriff auf die Speicherarrays der Größe von 64 Kb sowie 1.024 Kb etwa 100 Millionen Zugriffe aus dem jeweiligen L2-Cache beziehungsweise L3-Cache bedient werden können. Auch die deutliche Erhöhung der Zugriffszeit für Speicherarrays, welche die L3-Kapazität überschreiten, resultiert aus den Cache-Misses, die in der Tabelle zu registrieren sind. Folglich wurden beim Zugriff auf das Speicherarray der Größe von 32 MB alle Daten aus dem langsamen Hauptspeicher bereitgestellt.

4.2.1 Auswirkung unterschiedlicher Stride-Werte

Die vorangehenden Messungen basieren auf Speicherzugriffen, bei denen der Abstand der Zugriffselemente 1024 Byte beträgt. Um zu überprüfen, inwiefern sich die Eigenschaft des Stride-Wertes auf die Zeit und den Energieverbrauch der CPU auswirkt, haben wir im weiteren Verlauf noch andere Abstandsgrößen untersucht. Die Abbildungen 4.3 (a) und 4.3 (b) stellen den Effekt unterschiedlicher Stride-Werte im Bezug auf die Zugriffszeit und den Energieverbrauch dar.
Zunächst ist festzustellen, dass die Kurvenverläufe des Energieverbrauchs den Kurvenverläufen der Zugriffszeit ähnlich sind. Dies verstärkt die Annahme, dass der Energieverbrauch der CPU hauptsächlich von der Zugriffszeit abhängt. Der Grund für die unterschiedlichen Zugriffszeiten bei verschiedenen Stride-Werten ist ebenfalls der Einsatz des Caches. So



(a) mittlere Zugriffszeiten (b) Energieverbrauch der CPU

Abbildung 4.3: Zugriffszeiten und Energieverbrauch der CPU (Thread:1, Speicher:1, sequentieller Zugriff)

ergeben sich für kleine Stride-Werte auch bei Speichergrößen, welche die Kapazität des L3-Caches überschreiten, geringe Zugriffszeiten. Demzufolge wird im Gegensatz zu größeren Abständen weniger Energie verbraucht. Je kleiner der Abstand der Zugriffselemente ist, desto häufiger wird ein Cache-Hit erzielt. Dies liegt zum einen an dem Zwischenspeichern einer ganzen Cache-Line, sodass die nächsten Elemente, bei kleinem Abstand, bereits im Cache präsent sind. Zum anderen wird dieser Effekt, wie in Kapitel 2 erläutert, durch das Prefetching verstärkt.
Der direkte Vergleich der Stride-Werte im Bezug auf die Leistung, die für die jeweilige Ausführung der Speicherzugriffe aufgebracht wird, zeigt einen weiteren Zusammenhang zwischen dem Energieverbrauch und der Cache-Ebenen. In Abbildung 4.4 wird die benötigte Zugriffszeit dem Energieverbrauch gegenüber gestellt, sodass die Leistungsaufnahme der einzelnen Speicherzugriffe veranschaulicht wird.

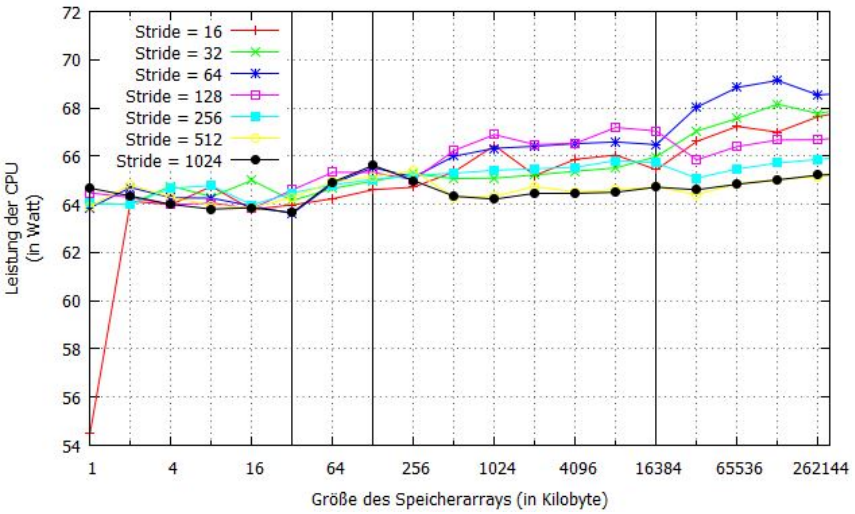


Abbildung 4.4: Leistungsaufnahme der CPU (Thread:1, Speicher:1, sequentieller Zugriff)

Vor allem im Speichergrößenbereich des Hauptspeichers sind auf den ersten Blick unerwartete Leistungsunterschiede messbar. Ausgehend von den Messwerten aus Abbildung 4.3 wäre zu erwarten, dass die Leistung entweder für alle Stride-Werte ähnlich ausfällt oder sich an der Größenordnung der Stride-Werte orientiert. Auffällig ist jedoch, dass die höchste gemessene Leistung weder die Ausführung der Speicherzugriffe mit dem höchsten noch mit dem niedrigsten Stride-Wert widerspiegelt. Stattdessen weist die Ausführung der Speicherzugriffe mit dem Zugriffsabstand in Höhe von 64 Byte die höchste Leistung auf. Auch an dieser Stelle betrachteten wir die Anzahl der Cache-Hits und untersuchten das Cache-Verhalten für die Speichergröße von 128 MB:

Stride	L1-Cache Hits	L2-Cache Hits	L3-Cache Hits
16	75 Mio	21 Mio	3 Mio
32	50 Mio	39 Mio	9 Mio
64	$\approx$ *0*	75 Mio	19 Mio
128	$\approx$ *0*	21 Mio	$\approx$ *0*
256	$\approx$ *0*	3 Mio	$\approx$ *0*
512	$\approx$ *0*	$\approx$ *0*	$\approx$ *0*
1024	$\approx$ *0*	$\approx$ *0*	$\approx$ *0*

Tabelle 4.2: Cache-Verhalten für Speicherarray = 128 Mb (Thread:1, Speicher:1, sequentieller Zugriff)

Im Zusammenhang mit den Leistungskurven lässt sich aus Tabelle 4.2 schlussfolgern, dass je mehr Speicherzugriffe auf den untersten Cache-Level Treffer erzielen, desto höher die Leistungsaufnahme. Anhand der gemessenen Anzahl von L3-Cache-Hits ist demnach für die Durchführung der Zugriffe mit Stride = 64 Byte die größte Leistungsaufnahme zuzuordnen, gefolgt von Stride = 32 Byte und Stride = 16 Byte. Die Schlussfolgerung stimmt auch für Stride = 128, 256, 512 und 1024 Byte überein. Da bei diesen Messungen keine L3-Cache-Hits erfolgen, ist als nächstes der L2-Cache hinsichtlich der meisten Cache-Hits zu überprüfen. Dies liegt bei Stride = 128 Byte vor, gefolgt von 256 Byte. Die niedrigsten und gleichzeitig nahezu identischen Energiewerte sind für Stride = 512 und 1024 Byte zu beobachten, was sich ebenfalls in den Leistungskurven aus der Abbildung 4.4 widerspiegelt.

4.2.2 Auswirkung des zufälligen Zugriffsmusters

Wie in Abschnitt 2.1.2 erläutert, kann bei der Entwicklung eines Programms festgelegt werden, in welcher Reihenfolge auf die Speicherdaten zugegriffen werden soll. Im Vergleich zu den bisherigen Messungen, bei denen sequentiell auf die Elemente des Speicherarrays zugegriffen wurde, betrachten wir an dieser Stelle den zufälligen Zugriff, dessen Messergebnisse in Abbildung 4.5 dargestellt sind.

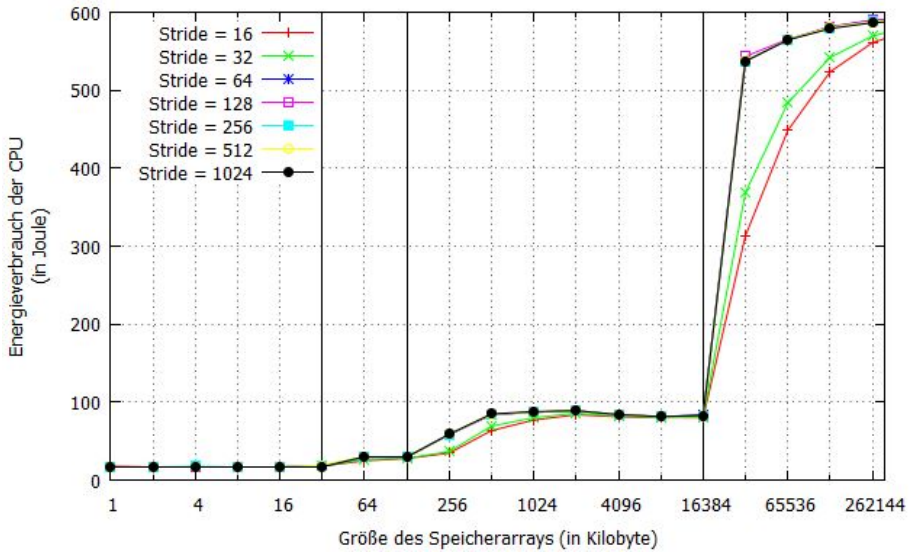


Abbildung 4.5: Energieverbrauch der CPU
(Thread:1, Speicher:1, zufälliger Zugriff)

Auffällig ist, dass die Funktion des Caches bei dem zufälligen Speicherzugriffsmuster an Effizienz verliert, wodurch der Energieverbrauch für unterschiedlicher Stride-Werte je Speichergröße nahezu identisch ist. Der Grund dafür ist, dass die Möglichkeit verringert wird, nachfolgende Speicherelemente im Cache zu haben und diese von dort aus lesen zu können. Eine Ausnahme bilden die Zugriffe mit Stride-Werten = 16 Byte und 32 Byte. Der Energieverbrauch dieser Speicherzugriffe steigt zwar im Vergleich zur sequentiellen Zugriffsreihenfolge um ein vielfaches an, allerdings ist der gemessene Energieverbrauch geringer als für die restlichen Stride-Werte.
Erklären lässt sich dies mit der zusammenhängenden Cache-Line Größe von 64 Byte. Für Stride-Werte, die kleiner sind als die Größe einer Cache-Line, besteht trotz des zufälligen Zugriffs eine höhere Wahrscheinlichkeit, benötigte Speicherdaten bereits im Cache zu haben. Daraus resultiert, dass die Daten teilweise aus dem schnellen Cache-Speicher geladen wurden und somit einen geringeren Energieverbrauch verursachten.
Da diese Wahrscheinlichkeit bei Verwendung von Stride-Werten, welche die Größe der Cache-Line überschreiten, nicht gegeben ist, ergeben sich für diese Stride-Werte ähnliche Messwerte. Diese Eigenschaft äußert sich ebenfalls in der Leistungsaufnahme der CPU. Wie in Abbildung 4.6 zu erkennen, ist die Leistung bei einem zufälligen Zugriffsmuster, im Vergleich zur gemessenen Leistung des sequentiellen Zugriffs in Abbildung 4.4, für alle Stride-Werte sehr ähnlich.
Einen weiteren Nachweis für die Abhängigkeit des Energieverbrauchs zur Zugriffszeit erhielten wir mit der Auswertung der Messergebnisse bezüglich der Lokalität von Thread und Speicher.

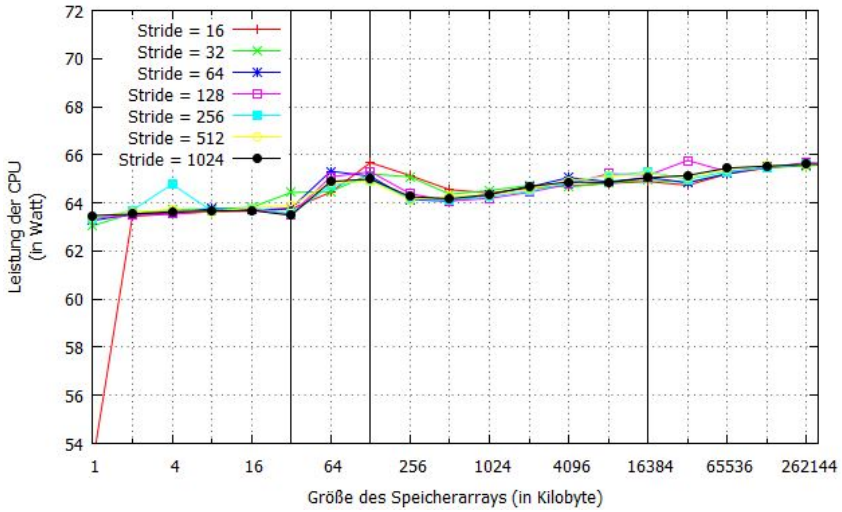


Abbildung 4.6: Leistungsaufnahme der CPU
(Thread:1, Speicher:1, zufälliger Zugriff)

4.2.3 Zugriff auf lokalen und entfernten Speicher

In diesem Abschnitt wird der Einsatz von zwei gleichzeitig genutzten Prozessorsockeln analysiert. Bei den bisherigen Messungen sind der Thread und der Speicher auf demselben Knoten konfiguriert gewesen. Nachfolgend werden die Messergebnisse dargestellt, bei denen der Thread und der Speicher auf unterschiedlichen Knoten festgelegt sind.

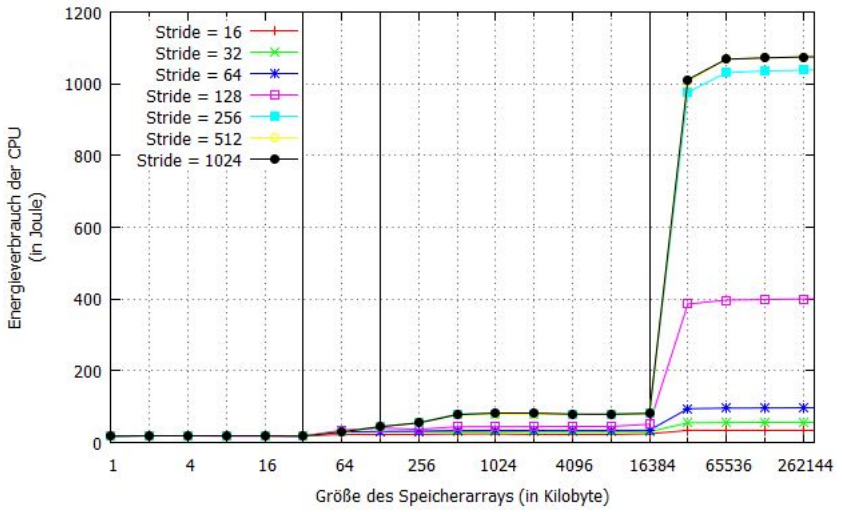


Abbildung 4.7: Energieverbrauch der CPU
(Thread:1, Speicher:0, sequentieller Zugriff)

Die Messergebnisse zeigen, dass die Ausführung der Zugriffe auf Speicherarrays, welche die L3-Cache Kapazität nicht überschreiten, ähnlich viel Energie verbraucht, wie die Ausführung von Zugriffen, bei denen sich der Thread und der Speicher auf gleichem Knoten befinden.

Deutliche Abweichungen sind für die Zugriffe auf Speichergrößen von über 16 MB festzustellen. Für Speicherarrays dieser Größenordnung benötigt die CPU ungefähr doppelt soviel Energie. Statt einem Verbrauch von etwa 557 Joule, welcher beim Zugriff auf den lokalen Speicher für Stride = 1024 Byte gemessen wurde, ist beim Zugriff auf den entfernten Speicher ein Verbrauch von circa 1077 Joule festzustellen.
Der erhöhte Energieverbrauch beim Zugriff auf den entfernten Speicher kann durch den, in Kapitel 2 erläuterten Aufbau eines NUMA-Systems erklärt werden. Dabei durchlaufen die Daten beim Zugriff auf entfernte Speicher, über die Interconnect-Datenleitung, einen längeren Weg, bis sie vom Prozessor verarbeitet werden. Dies kostet mehr Zeit und äußert sich dementsprechend in einem erhöhten Energieverbrauch.
Die Wartezyklen, die beim Zugriff auf den entfernten Speicher entstehen, spiegeln sich in den gemessenen Ausführungszeiten aus Abbildung 4.8 wider. Statt der bisher benötigten 85 Nanosekunden pro Speicherzugriff und Stride = 1024 Byte (vergleiche Abbildung 4.3a) sind nun fast 164 Nanosekunden zu messen.

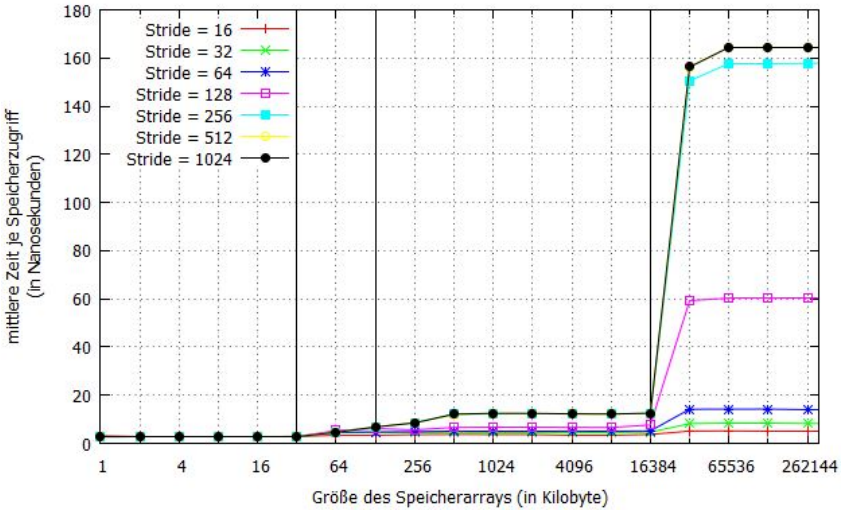


Abbildung 4.8: Zugriffszeiten
(Thread:1, Speicher:0, sequentieller Zugriff)

Aus diesen Beobachtungen kann entnommen werden, dass die CPU auch während der Wartezyklen Strom verbraucht. Wird die verbrauchte Energie der benötigten Zeit gegenüber gestellt, erhalten wir die Leistungsaufnahme der CPU, welche in Abbildung 4.9 veranschaulicht wird.
Trotz Verwendung der zusätzlichen Interconnect-Leitung und der Übergabe der Speicheranforderung an den Prozessor des entfernten Knotens ist eine nahezu identische Leistung des Prozessors zu messen, wie bei den Speicherzugriffsmessungen mit Konfiguration des Threads und Speichers auf gleichem Knoten (vergleiche Abbildung 4.4). Die unterschiedliche Art des Bussystems wirkt sich demnach nicht auf die Leistungsaufnahme der CPU aus.

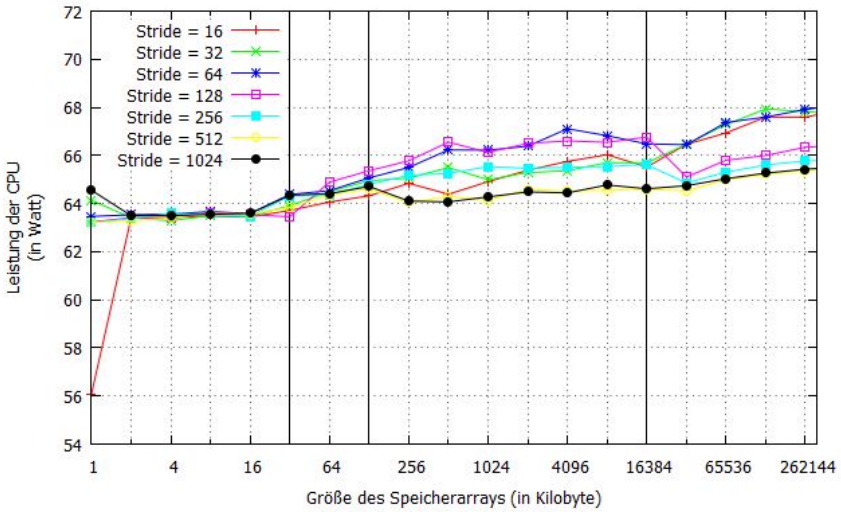
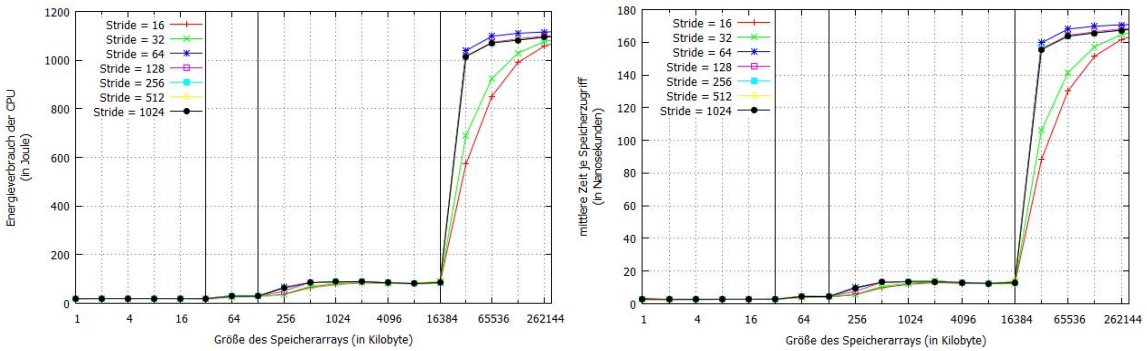


Abbildung 4.9: Leistung der CPU
(Thread:1, Speicher:0, sequentieller Zugriff)

Für das zufällige Zugriffsmuster sind, bei dem Zugriff auf einen entfernten Speicher, ähnliche Auswirkungen zu beobachten. In Abbildung 4.10 werden die entsprechenden Zugriffszeiten und der gemessene Energieverbrauch der CPU dargestellt.



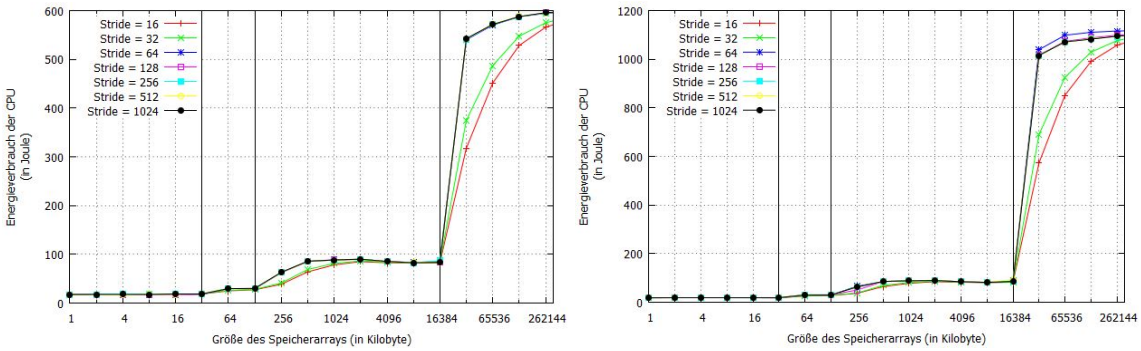
(a) Energieverbrauch der CPU

(b) mittlere Zugriffszeiten

Abbildung 4.10: Zugriffszeiten und Energieverbrauch der CPU (Thread:1, Speicher:0, zufälliger Zugriff)

Statt zuvor, für Stride = 1024 Byte und einer Speichergröße von 256 MB, gemessenen 590 Joule wurden beim Zugriff auf den entfernten Speicher fast 1095 Joule verbraucht und damit doppelt so viel.
Des Weiteren ist zu beobachten, dass die Speicherzugriffe für den Energieverbrauch identische Kurvenverläufe aufweisen wie für die entsprechende Zugriffszeit. Dies stellten wir ebenfalls, am Beispiel des sequentiellen Zugriffsmusters für den Zugriff auf den lokalen Speicher, fest (vergleiche Abbildung 4.7 und Abbildung 4.8).

Hinsichtlich der Lokalität von Thread und Speicher kann die Ausführung der Speicherzugriffe bei dem zugrundeliegenden NUMA-Prozessorsystem mit zwei weiteren Belegungskombinationen erfolgen. Zum einen können sowohl der Thread als auch der Speicher zusammen auf dem NUMA-Knoten = 0 definiert werden und zum anderen kann der Thread auf NUMA-Knoten = 0 ausgeführt werden und auf Speicherdaten auf NUMA-Knoten = 1 zugreifen.
Zur Vollständigkeit der unterschiedlichen Lokalitäten wird in Abbildung 4.11 der Energieverbrauch dieser beiden Konfigurationen, am Beispiel des zufälligen Speicherzugriffs, veranschaulicht. Wie erwartet, zeigen die Messungen identische Energiewerte, wie wir sie bereits in Abbildung 4.5 und Abbildung 4.10 (a) für den lokalen Speicherzugriff beobachteten.



(a) Thread:0, Speicher:0 (b) Thread:0, Speicher:1

Abbildung 4.11: Energieverbrauch der CPU mit zufälligem Zugriffsmuster

4.2.4 Zusammenfassung der Ergebnisse

Aus der Analyse der dargestellten Messwerte können wir folgende Ergebnisse festhalten.
Zum einen spielt die Organisation der Speicherverwaltung für den Energieverbrauch des Prozessors eine wesentliche Rolle. Besonders energieeffizient sind Programmausführungen, bei denen hauptsächlich auf die Speicherdaten des L1-Caches zugriffen wird. Der Zugriff auf Speichergrößen, welche die L2-Cache und L3-Cache Kapazitäten nicht überschreiten, verursacht einen nur etwas größeren Energieverbrauch.
Es hat sich weiterhin herausgestellt, dass vor allem die Speicherzugriffe, die nicht aus dem Cache bedient werden können, einen deutlich höheren Energieverbrauch aufweisen. Wie aus den Auswertungen hervorgeht, liegt dies an der längeren Ausführungszeit, die durch den Zugriff auf den Hauptspeicher verursacht wird. Des Weiteren hat sich aus der gemessenen Leistung ergeben, dass diese für Zugriffe auf unterschiedliche Speichergrößen nahezu identisch ist. Daraus folgt, dass der Energieverbrauch des Prozessors hauptsächlich von der benötigten Gesamtzeit der Programmausführung abhängig ist.

4.3 Energiebedarf des Hauptspeichers

In diesem Abschnitt analysieren wir den Energieverbrauch des Hauptspeichers. Die nachfolgenden Messergebnisse resultieren aus den Speicherzugriffsmessungen, die bereits für die Auswertung des Energieverbrauchs des Prozessors genutzt wurden. Allerdings wurde für die Messung des Energieverbrauchs des Hauptspeichers die entsprechende Funktion *getDRAMConsumedJoules(BeforeStateSystem,AfterStateSystem)* genutzt.
In Abbildung 4.12 wird die verbrauchte Energie des Hauptspeichers dargestellt, bei dem der Speicherzugriff mit zufälligem Zugriffsmuster auf einen lokalen Speicher erfolgte.
Erwartungsgemäß wurden bei dem Zugriff auf kleine Speicherbereiche sehr geringe Energiewerte gemessen. Dies liegt daran, dass die Speicherdaten nach dem ersten Zugriff aus dem Hauptspeicher in den Cache zwischengespeichert werden und der Hauptspeicher für die weiteren Zugriffe nicht mehr angesprochen werden muss. Wie zuvor erläutert, trifft dieses Cache-Verhalten nur auf Speichergrößen zu, welche die L3-Cache Kapazität nicht überschreiten.

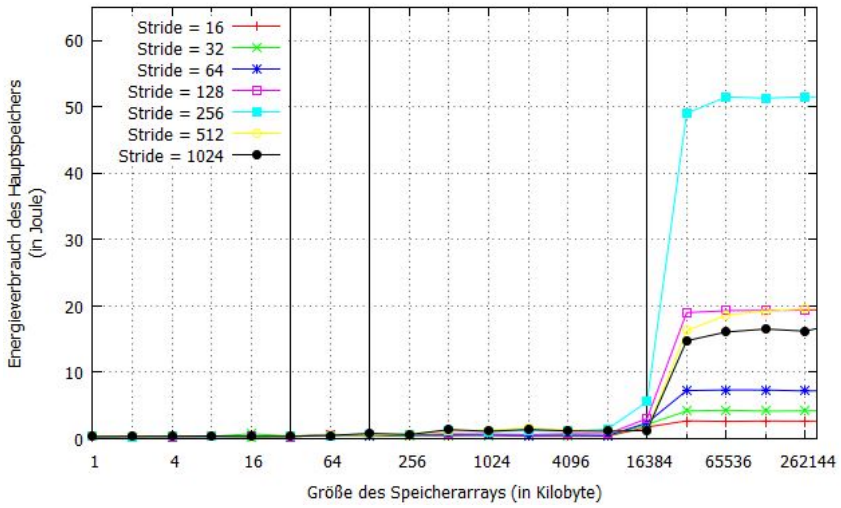


Abbildung 4.12: Energieverbrauch des Hauptspeichers (Thread:1, Speicher:1, sequentieller Zugriff)

Außerdem zeigen die Messwerte, dass der Hauptspeicher wesentlich weniger Energie verbraucht als der Prozessor. Am Beispiel der durchgeführten Speicherzugriffe mit Stride = 1024 Byte und der Speichergröße von 128 MB sind 16,5 Joule gemessen worden. Dies entspricht etwa 3% der Energie, die bei gleicher Speicherkonfiguration von dem Prozessor verbraucht wurde (557,5 Joule).
Erklären lässt sich dies mit dem, in Abschnitt 2.1.2 erläuterten, einfacherem Aufbau der DRAM-Speicherzelle. Die im Hauptspeicher verwendeten DRAM-Speicherzellen bestehen jeweils nur aus einem Transistor und einem Kondensator. Im Betrieb verursachen diese einen geringeren Energieverbrauch als die aufwendiger geschalteten SRAM-Speicherzellen,

die unter anderem im Prozessorchip integriert und für die Verarbeitung der Speicherdaten genutzt werden.
Besonders auffällig und unerwartet ist der hohe Energieverbrauch im Fall von Stride = 256 Byte. Vergleichen wir dazu die bereits veranschaulichten Zugriffszeiten aus Abbildung 4.3 (a), lässt sich vorerst kein Zusammenhang zwischen der Ausführungszeit und dem Energieverbrauch des Hauptspeichers feststellen. Um auszuschließen, dass es sich dabei um Messfehler handelt, wurde die Ausführung dieser Speicherzugriffskonfiguration wiederholt. Dies bestätigte jedoch die Korrektheit der gemessenen Werte.

4.3.1 Auswirkung der Anzahl an Hauptspeicherzugriffen

Angesichts der undurchschaubaren Beobachtungen bezüglich des Energieverbrauchs bei Speicherzugriffen unterschiedlicher Stride-Werte, wird überprüft, inwiefern sich die Anzahl der Hauptspeicherzugriffe auf dessen Energieverbrauch auswirkt. Dazu vergleichen wir die verbrauchte Energie mit den gemessenen Cache-Hits anhand von Speicherzugriffen, die mit unterschiedlichen Stride-Werten durchgeführt wurden. Der Energieverbrauch ist in Abbildung 4.13 abgebildet. Die Messerwerte ergeben sich aus Speicherzugriffen, bei denen der Thread auf Knoten 1 ausgeführt und auf den entfernten Speicher zugegriffen wurde. Die Beobachtungen bezüglich der veränderten Lokalität von Thread und Speicher werden im nachfolgenden Abschnitt ausgewertet.

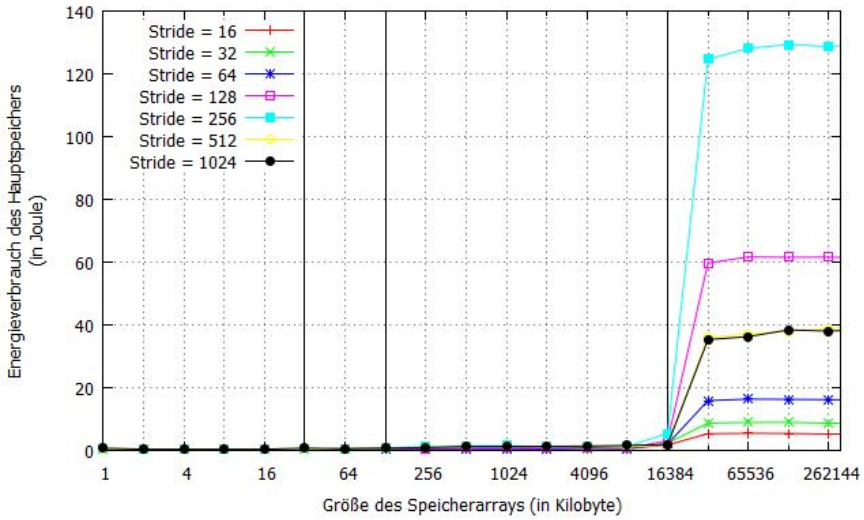


Abbildung 4.13: Energieverbrauch des Hauptspeichers (Thread:1, Speicher:0, sequentieller Zugriff)

Da die Messwerte bei einer Speichergröße ab 32 MB nahezu identisch sind, beschränken wir uns bei der Analyse auf die Messung der Speicherarrays der Größe von 128 MB.
Für die Speicherzugriffe mit den Stride-Werten = 16 Byte, 32 Byte und 64 Byte ist ein vergleichsweise geringer Energieverbrauch festzustellen.

Vergleichen wir die zugehörige Anzahl der Hauptspeicherzugriffe aus Tabelle 4.3, so ist zu erkennen, dass nur wenige Zugriffe auf den Hauptspeicher erfolgten. Im Gegensatz dazu können wir bei den restlichen Stride-Werten eine größere Anzahl an Hauptspeicherzugriffen registrieren, welche sich in einem erhöhten Energieverbrauch widerspiegelt.

Stride	L1-Cache Hits	L2-Cache Hits	L3-Cache Hits	Hauptspeicherzugriffe
16	76 Mio	19 Mio	4 Mio	1 Mio
32	51 Mio	35 Mio	9 Mio	5 Mio
64	$\approx$ *0*	64 Mio	22 Mio	14 Mio
128	$\approx$ *0*	29 Mio	$\approx$ *0*	71 Mio
256	$\approx$ *0*	$\approx$ *0*	$\approx$ *0*	100 Mio
512	$\approx$ *0*	$\approx$ *0*	$\approx$ *0*	100 Mio
1024	$\approx$ *0*	$\approx$ *0*	$\approx$ *0*	100 Mio

Tabelle 4.3: Cache-Verhalten für Speicherarray = 128 Mb (Thread:1, Speicher:0, sequentieller Zugriff)

Aufgrund der vermuteten Abhängigkeit zwischen dem Energieverbrauch und der Anzahl an Hauptspeicherzugriffen, wäre zu erwarten, dass der Energieverbrauch von Speicherzugriffen, die dieselbe Anzahl an Hauptspeicherzugriffen aufweisen, einen entsprechend ähnlichen Energieverbrauch aufweisen. Jedoch ist der gegenteilige Fall zu beobachten. Aus Tabelle 4.3 geht hervor, dass für Stride = 256 Byte, 512 Byte und 1024 Byte keine Cache-Hits gemessen worden sind, sodass bei der Ausführung dieser Speicherzugriffe alle Speicherdaten aus dem Hauptspeicher geladen wurden. Allerdings stellten wir für Stride = 256 Byte einen deutlich höheren Energieverbrauch fest als für die beiden anderen Stride-Werte. Dies führt zur Annahme, dass der Energieverbrauch des Hauptspeicher von einem weiteren Faktor innerhalb des Computersystems beeinflusst wird.
Des Weiteren ist für Stride = 128 Byte ein höherer Energieverbrauch als für Stride = 1024 Byte festzustellen. Dies spricht nicht für eine proportionale Veränderung des Energieverbrauch zur Anzahl der Hauptspeicherzugriffe, da für Stride = 128 weniger Zugriffe auf den Hauptspeicher (etwa 71 Millionen) erfolgten, als für Stride = 1024 Byte (100 Millionen).
Eine mögliche Erklärung für diese Beobachtung ist, dass es entscheidend sein könnte, ob der Hauptspeicher dauerhaft beansprucht wird oder ob Speicherdaten zwischenzeitlich aus dem Cache geladen werden. Möglicherweise benötigt der Hauptspeicher mehr Energie, wenn dieser mehrfach erneut angesprochen wird, anstatt der fortlaufenden Kommunikation mit dem Prozessor.

4.3.2 Untersuchung der durchgehenden Anforderung des Hauptspeichers

Die Überprüfung der Annahme, dass Speicherzugriffe, welche den Hauptspeicher durchgehend beanspruchen, energieeffizienter sind, als Speicherzugriffe, bei denen die Speicherdaten zum Teil aus dem Cache geladen werden, führten wir anhand der Auswertung von Speicherzugriffen mit zufälliger Zugriffsreihenfolge durch.
Abbildung 4.14 stellt den Energieverbrauch des Hauptspeichers, bei Ausführung der Speicherzugriffe mit zufälligem Zugriffsmuster und verschiedenen Stride-Werten, dar.

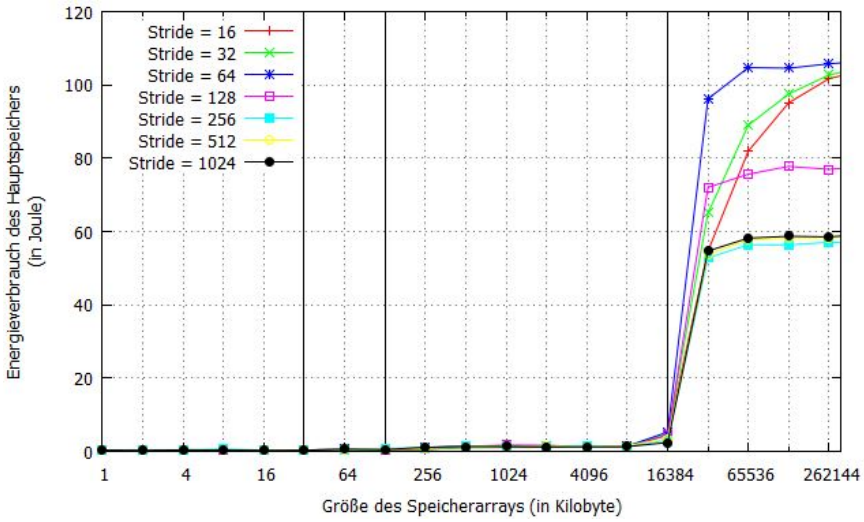


Abbildung 4.14: Energieverbrauch des Hauptspeichers
(Thread:1, Speicher:0, zufälliger Zugriff)

Auch an dieser Stelle werteten wir die Messergebnisse am Beispiel des Speicherarrays der Größe von 128 MB aus und verglichen diese mit der Anzahl der Hauptspeicherzugriffe aus Tabelle 4.4.
Die jeweilige Gegenüberstellung des Energieverbrauchs mit den Zugriffen auf den Hauptspeicher ergibt, dass bei dieser Speicherkonfiguration sowohl für Stride = 64 Byte als auch für Stride = 128 Byte, 256 Byte, 512 Byte und 1024 Byte alle Speicherdaten aus dem Hauptspeicher angefordert wurden. Die entsprechenden Werte für den Energieverbrauch des Hauptspeichers fallen allerdings sehr unterschiedlich aus, sodass sich kein Zusammenhang zwischen dem Energieverbrauch und dem dauerhaften Anfordern der Speicherdaten aus dem Hauptspeicher feststellen lässt.
Für die Stride-Werte 16 Byte und 32 Byte sind, neben der großen Anzahl an Hauptspeicherzugriffen, Zugriffe auf den L3-Cache festzustellen. In diesen Fällen ist somit davon auszugehen, dass der Hauptspeicher in unregelmäßigen Abständen immer wieder erneut angesprochen wird. Jedoch ergibt sich auch aus diesem Effekt kein eindeutiges Verhalten für den Energieverbrauch des Hauptspeichers, wie in Abbildung 4.14 zu sehen ist.
Wir erhielten mit den zur Verfügung stehenden Funktionen der Intel-PCM Bibliothek keine

Stride	L1-Cache Hits	L2-Cache Hits	L3-Cache Hits	Hauptspeicherzugriffe
16	≈ 0	≈ 0	12 Mio	88 Mio
32	≈ 0	≈ 0	8 Mio	92 Mio
64	≈ 0	≈ 0	≈ 0	100 Mio
128	≈ 0	≈ 0	≈ 0	100 Mio
256	≈ 0	≈ 0	≈ 0	100 Mio
512	≈ 0	≈ 0	≈ 0	100 Mio
1024	≈ 0	≈ 0	≈ 0	100 Mio

Tabelle 4.4: Cache-Verhalten für Speicherarray = 128 Mb (Thread:1, Speicher:0, zufälliger Zugriff)

weiteren Informationen zum genauen Ablauf des Cache-Verhaltens, sodass die Häufigkeit des Wechsels der Zugriffe auf Cache und Hauptspeicher nicht nachvollzogen werden kann. Außerdem führen die widersprüchlichen Ergebnisse aus der Messungen des sequentiellen und des zufälligen Zugriffsmusters zu keinem eindeutigen Verhalten des Energieverbrauchs vom Hauptspeicher. Folglich können wir nicht erschließen, ob sich die durchgehende Beanspruchung des Hauptspeicher, positiv oder negativ auf dessen Energieverbrauch auswirkt.

4.3.3 Zugriff auf lokalen und entfernten Speicher

Wie auch bei dem Energieverbrauch des Prozessors, analysieren wir in diesem Abschnitt, wie sich die Lokalität des Prozessors und Hauptspeichers auf den Energieverbrauch auswirkt.

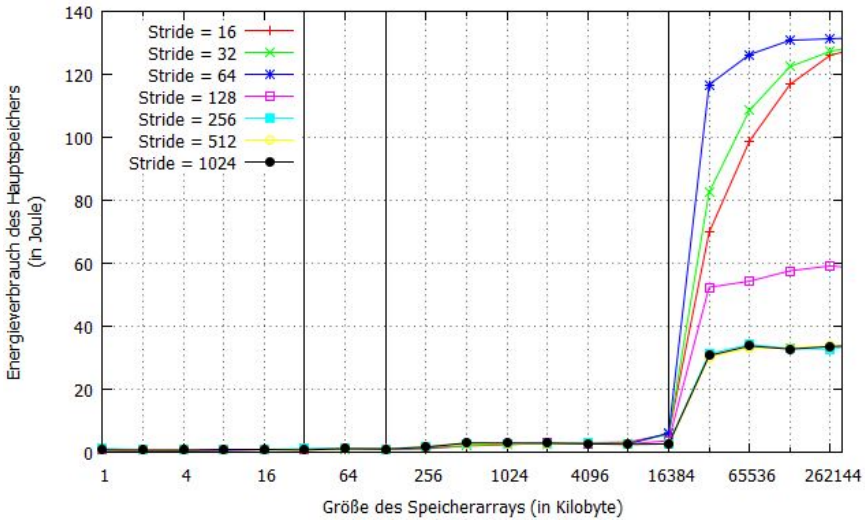


Abbildung 4.15: Energieverbrauch des Hauptspeichers (Thread:0, Speicher:1, zufälliger Zugriff)

Die Abbildung 4.15 veranschaulicht die Messergebnisse von Speicherzugriffen, bei denen wir den Thread, im Vergleich zu den vorangehenden Messungen, von Knoten 1 auf Knoten 0 veränderten und den Speicher statt auf Knoten 0 auf den Knoten 1 definierten.
Auffällig ist, dass die Veränderung der Lokalität von Speicher und Prozessor den Energieverbrauch, bei Speichergrößen, welche die L3-Cache Kapazität überschreiten, je nach Stridegröße unterschiedlich beeinflusst. Im Fall von Stride = 64 Byte ist eine Erhöhung des jeweils maximal gemessenen Energieverbrauchs von 23% festzustellen (Steigerung von 106,63 Joule auf 131,82 Joule für Speichergröße = 512 MB). Ähnlich verhält sich die Erhöhung des Verbrauchs für Stride = 32 Byte und 64 Byte. Im Gegensatz dazu wirkt sich die Veränderung der Lokalität für Stride = 128 Byte, 256 Byte, 512 Byte und 1024 Byte positiv auf den Energieverbrauch aus. Bei gleicher Speichergröße von 512 MB ist für Stride = 128 Byte eine Reduzierung des Energieverbrauch um etwa 26% festzustellen. Für Stride = 1024 Byte geht der Energieverbrauch von 59,31 Joule auf 34,14 Joule zurück, was einer Senkung des Energieverbrauchs um circa 42% entspricht.
Mit dem Vergleich der Messergebnisse aus Abbildung 4.13 und den Messergebnissen aus Abbildung 4.16 überprüften wir, ob sich die Veränderung der Lokalität, für das sequentielle Zugriffsmuster gleichermaßen auswirkt.

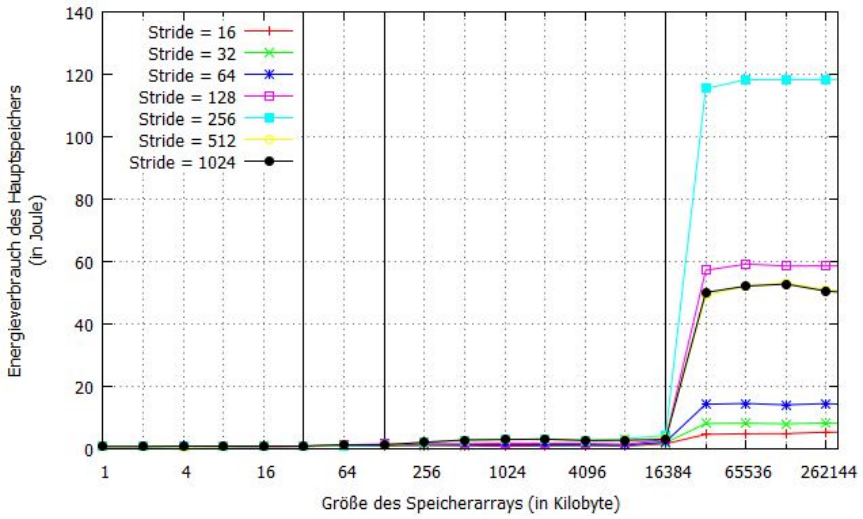
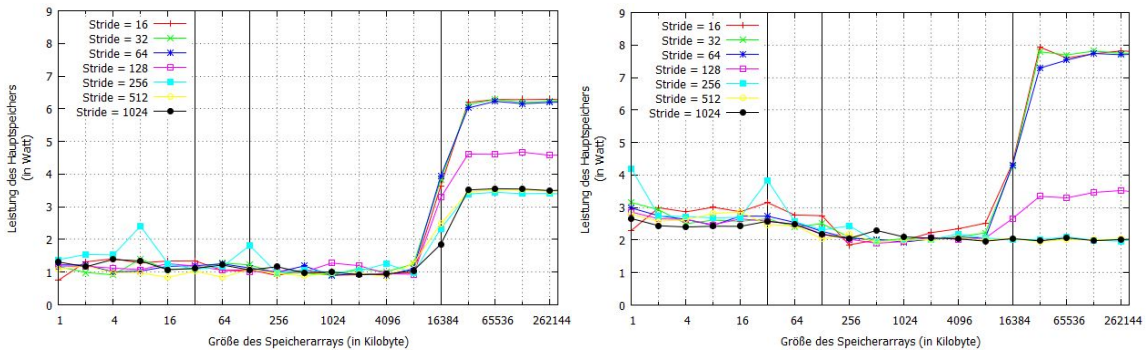


Abbildung 4.16: Energieverbrauch des Hauptspeichers (Thread:0, Speicher:1, sequentieller Zugriff)

Wie beispielsweise an Stride = 1024 Byte zu kennen, ist dies nicht der Fall. Anstatt einer Reduzierung des Energieverbrauchs, wie beim zufälligen Speicherzugriff zu beobachten war, erhöhte sich der maximal gemessene Energiewert von 38,45 Joule auf 52,83 Joule.
Aufgrund der uneinheitlichen Veränderungen der Messwerte bei Ausführung von Speicherzugriffen, die sich sowohl in der Lokalität als auch in dem verwendeten Zugriffsmuster unterscheiden, lässt sich bezüglich des Energieverbrauch des Hauptspeichers weiterhin keine eindeutige Aussage treffen.

Leistungsaufnahme im Bezug auf kleine Speicherarrays

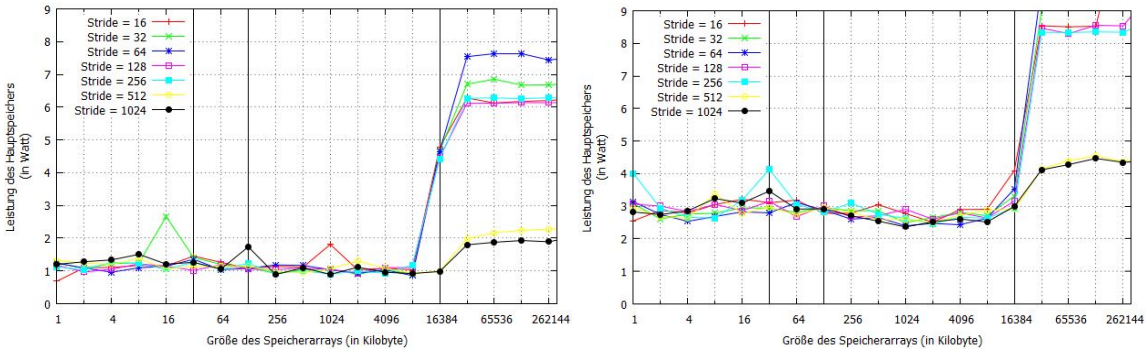
Im Bezug auf die Leistungsaufnahme des Hauptspeichers ist sowohl für das sequentielle als auch für das zufällige Zugriffsmuster, bei Zugriffen auf Speicherarrays, welche die Kapazität des L3-Caches nicht überschreiten, eine Abhängigkeit zur Lokalität des Threads festzustellen. Die Abbildungen 4.17 (a) und (b) veranschaulichen die gemessene Leistung des Hauptspeichers für den Zugriff auf den entfernten Speicher. Allerdings unterscheiden sich die Messwerte in der Lokalität des Threads und des Speichers.



(a) Thread:1, Speicher:0 (b) Thread:0, Speicher:1

Abbildung 4.17: Leistungsaufnahme des Hauptspeichers (zufälliger Zugriff)

Die Ausführung von Speicherzugriffen, bei denen der Thread auf NUMA-Knoten = 1 ausgeführt wurde, benötigte für Speicherarrays, mit einer Größe bis zu 8 MB, eine durchschnittliche Leistung von etwa 1 bis 1,5 Watt. Bei der Ausführung des Threads auf NUMA-Knoten = 0 wurde eine höhere Leistung gemessen. Diese betragen, wie in Abbildung 4.17 (b) dargestellt, eine durchschnittliche Leistung von etwa 2 bis 3 Watt.
Ein ähnliches Leistungsverhältnis ist aus den Messwerten in Abbildung 4.18 zu entnehmen, bei denen der Thread und der Speicher auf gleichem NUMA-Knoten konfiguriert sind.



(a) Thread:1, Speicher:1 (b) Thread:0, Speicher:0

Abbildung 4.18: Leistungsaufnahme des Hauptspeichers für gleiche Lokalität des Threads und des Speichers (sequentieller Zugriff)

Deutlich ist, dass auch bei diesen Messungen die Leistung des Hauptspeichers, trotz sequentiellen Zugriffsmuster, von der Lokalität des auszuführenden Threads beeinflusst wird.

4.3.4 Zusammenfassung der Ergebnisse

Die Analyse der in diesem Abschnitt durchgeführten Messungen ergibt, dass der Energieverbrauch des Hauptspeichers von der Größe des Speicherarrays abhängt. Damit zusammenhängend ist die Anzahl der Hauptspeicherzugriffe, die bei Zugriffen auf Speicherarrays, welche die Größe der L3-Cache Kapazität überschreiten, deutlich höher ist.
Ein anderer Aspekt ist, dass der Energieverbrauch des Hauptspeicher, für Zugriffe auf Speicherarrays, welche die Größe der L3-Cache Kapazität nicht überschreiten, von der Lokalität des auszuführenden Threads abhängt. Für die Messungen auf dem von uns genutzten NUMA-System stellte sich diesbezüglich die Ausführung des Threads auf NUMA-Knoten 1 als energieeffizienter heraus.
Weitere Erkenntnisse, die konsequent für alle durchgeführten Messungen nachvollziehbar sind, haben sich nicht ergeben. Somit bleibt festzuhalten, dass der Energieverbrauch des Hauptspeicher nicht allgemein vorhersehbar ist.

4.4 Auswertung von Multithreading

In diesem Abschnitt wird der Energieverbrauch des Prozessors und des Hauptspeichers im Bezug auf Multithreading untersucht. Wie in Kapitel 2 beschrieben, bezeichnet Multithreading die gleichzeitige Ausführung mehrerer unabhängiger Prozesse. Die nachfolgenden Messwerte basieren auf der gleichzeitigen Ausführung von vier Threads. Jedem auszuführenden Thread wurde jeweils ein Speicherarray zugeordnet, auf dem die Speicherzugriffe erfolgten.

4.4.1 Einfluss auf den Energieverbrauch des Prozessors

Für die in Abbildung 4.19 dargestellten Messwerte wurden sowohl die Threads als auch die Speicherarrays auf NUMA-Knoten = 0 konfiguriert und das sequentielles Zugriffsmuster vorgegeben.
Zunächst ist festzustellen, dass für unterschiedliche Stride-Werte ein unterschiedlich hoher Energieverbrauch der CPU gemessen wurde. Aufgrund der vier angelegten Speicherarrays erfolgten die Zugriffe der einzelnen Threads auf vier voneinander unabhängigen Speicherdaten. Da der verwendete NUMA-Knoten acht Prozessorkerne besitzt, ist davon auszugehen, dass die Threads sich auf die zur Verfügung stehenden Prozessorkerne aufgeteilt haben. Dies erklärt die auch hier gemessenen Abstufungen unterschiedlicher Stride-Werte.
Mit der gemessenen Anzahl an Speicher-Treffern, die in Tabelle 4.5 dargestellt ist, können

wir unsere Annahme bestätigen, dass die Threads auf unterschiedlichen Prozessorkernen ausgeführt werden.

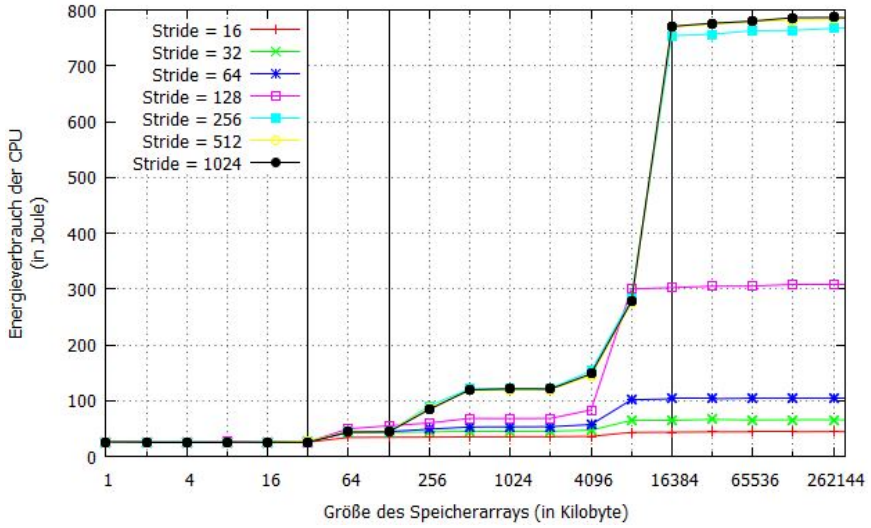


Abbildung 4.19: Energieverbrauch der CPU bei Multithreading (Thread:0, Speicher:0, sequentieller Zugriff)

Stride	L1-Cache Hits		L2-Cache Hits		L3- Cache Hits		Hauptspeicher	
	1 T	4 T	1 T	4 T	1 T	4 T	1 T	4 T
16	75 Mio	300 Mio	21 Mio	83 Mio	3 Mio	13 Mio	1 Mio	4 Mio
32	50 Mio	199 Mio	39 Mio	157 Mio	9 Mio	34 Mio	2 Mio	8 Mio
64	1 Mio	4 Mio	74 Mio	296 Mio	19 Mio	74 Mio	6 Mio	26 Mio
128	≈ 0	≈ 0	21 Mio	98 Mio	≈ 0	≈ 0	79 Mio	302 Mio
256	≈ 0	≈ 0	3 Mio	15 Mio	≈ 0	≈ 0	97 Mio	385 Mio
512	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	100 Mio	400 Mio
1024	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	100 Mio	400 Mio

Tabelle 4.5: Cache-Verhalten für Speicherarray = 128 Mb (Thread:0, Speicher:0, sequentieller Zugriff)

Aus der Tabelle ist zu entnehmen, dass im Vergleich zur Ausführung eines einzelnen Threads, bei vier ausgeführten Threads insgesamt jeweils entsprechend viermal soviel Cache-Hits gemessen wurden.
Unter Berücksichtigung der bisherigen Ergebnisse wäre zu erwarten, dass sich der Energieverbrauch bei vier gleichzeitig genutzten Prozessorkernen vervierfacht. Allerdings ist aus Abbildung 4.19 zu entnehmen, dass der Energieverbrauch des Prozessors sich nur gering erhöht. Beispielweise ist anhand der Messung mit Stride = 1024 Byte und der Speichergröße von 128 MB eine Erhöhung von 569 Joule (gemessen bei Ausführung eines einzigen Threads) auf 786 Joule zu messen.

Des Weiteren ist aus Abbildung 4.19 zu entnehmen, dass der deutliche Anstieg des Energieverbrauchs, welcher zuvor bei Überschreitung der L3-Cache Kapazität, also ab 32 MB, festzustellen war, bereits ab der Speichergröße von 8 MB auftritt.
Diese Beobachtung lässt sich mit dem Blick auf die in Kapitel 2 erläuterte Speicherverwaltung erklären. Wie dort beschrieben und in diesem Kapitel zu Beginn aufgeführt, wird der L3-Cache innerhalb eines NUMA-Knotens von allen Prozessorkernen gemeinsam genutzt. Dies hat bei der Ausführung mehrerer Threads zur Folge, dass eine größere Anzahl von unterschiedlichen Speicherdaten im L3-Cache zwischengespeichert werden und dadurch bereits zwischengespeicherte Daten früher ersetzt werden. Dies wiederum wirkt sich auf eine größere Anzahl an Hauptspeicherzugriffen aus, sodass beim Prozessor mehr Wartezyklen entstehen und demzufolge auch mehr Energie verbraucht wird.

4.4.2 Einfluss auf den Energieverbrauch des Hauptspeichers

Um zu überprüfen, inwiefern sich die gleichzeitige Ausführung von vier Threads auf das Verhalten des Hauptspeichers auswirkt, werden hier ebenfalls die in Tabelle 4.5 dargestellten Messwerte bezüglich der Anzahl an Cache- und Hauptspeicherzugriffen mit dem gemessenen Energieverbrauch verglichen. In Abbildung 4.20 werden die zugehörigen Energiewerte veranschaulicht.

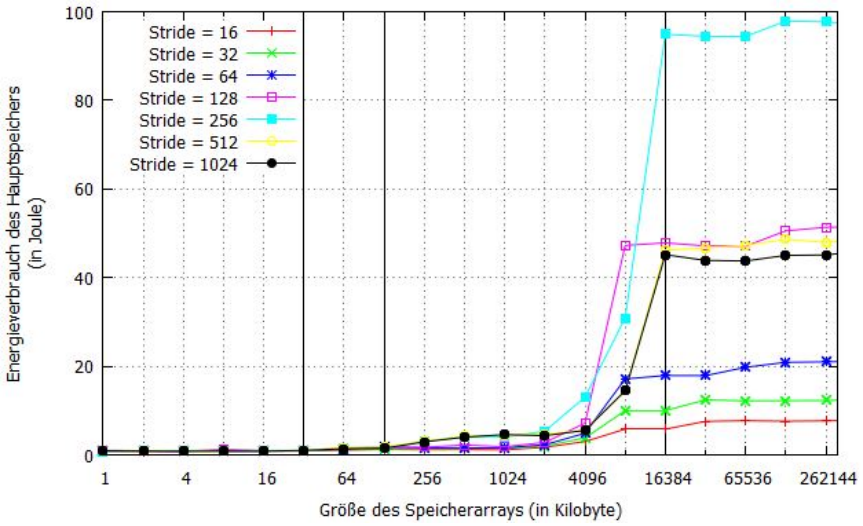


Abbildung 4.20: Energieverbrauch des Hauptspeichers bei Multithreading (Thread:0, Speicher:0, sequentieller Zugriff)

Wie sich bereits aus der Auswertung der Messungen mit einem einzigen ausgeführten Thread ergeben hat, wurde auch bei der gleichzeitigen Ausführung von mehreren Threads ein deutlich erhöhter Energieverbrauch im Fall von Stride = 256 Byte gemessen. Diesen Effekt können wir auch an dieser Stelle nicht mit den gemessenen Hauptspeicherzugriffen erklären. Anhand der gemessenen Ausführungszeit, die in Abbildung 4.21 dargestellt ist,

wäre für Stride = 256 Byte im Bezug zu den anderen Stride-Werten ein niedrigerer Energieverbrauch zu erwarten.
Für den Vergleich des Energieverbrauchs im Bezug auf verschiedene Speichergrößen sind in Tabelle 4.6 sowohl die gemessenen Energiewerte für vier Threads, als auch die eines einzigen Threads am Beispiel von Stride = 1024 Byte dargestellt.

Speicher	Energieverbrauch		prozentuale Erhöhung
	1 T	4 T	
1 kB	0,78 J	0,93 J	19,23%
64 kB	1,33 J	1,38 J	3,76%
1.024 kB	3,02 J	4,57 J	51,23%
131.072 kB	38,78 J	45,03 J	16,12%

Tabelle 4.6: Energie-Verhalten des Hauptspeichers bei Multithreading (Stride = 1024 Byte, Thread:0, Speicher:0, sequentieller Zugriff)

Für alle Speicherbereiche ist, durch die Ausführung mehrerer Threads, eine Erhöhung des Energieverbrauchs festzustellen.
Zu erwarten wäre, dass sich der Energieverbrauch des Hauptspeichers um einen größeren Faktor erhöht. Denn nicht nur die Anzahl der Threads, sondern auch die Anzahl der zu verarbeitende Datenmenge wurde, durch das Anlegen der vier Speicherarrays, erhöht.
Aus der Erhöhung der Anzahl auszuführender Threads von 1 auf n Threads resultiert demnach nicht gleichzeitig ein n-facher Anstieg des Energieverbrauchs.

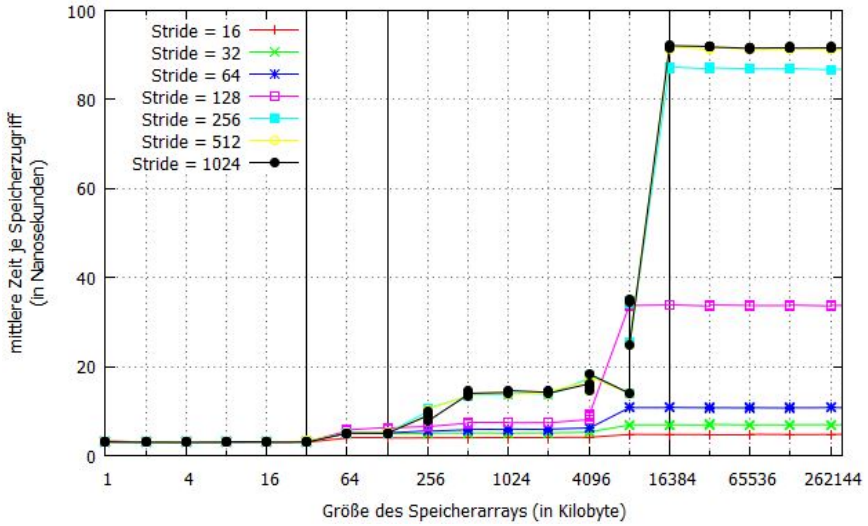


Abbildung 4.21: Zugriffszeit bei Multithreading (Thread:0, Speicher:0, sequentieller Zugriff)

Kapitel 5

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde der Energieverbrauch von Speicherzugriffen basierend auf einem modernen Multiprozessorsystem untersucht. Dafür wurde ein Programm eingesetzt, dessen Ausführung, Zugriffe auf den Speicher veranlasste.
Das Programm ermöglichte es, Zugriffe nach bestimmten Zugriffsmustern auszuführen, sodass der entstandene Energieverbrauch der Speicherzugriffe stets einem kontrollierbaren Zugriffsmuster zuzuordnen war. Für die Analyse des Energieverbrauchs wurden zahlreiche Messungen durchgeführt und im Bezug auf die Eigenschaften von Speicherzugriffen unterschiedlich konfiguriert.
Bei der Auswertung der Messergebnisse haben wir zwischen dem Energieverbrauch des Prozessors und dem Energieverbrauch des Hauptspeichers differenziert.
Die Messungen des Energieverbrauchs des Prozessors haben ergeben, dass der Verbrauch hauptsächlich von der Ausführungszeit eines Programms abhängig ist. Dementsprechend sind Zugriffe bei denen die benötigten Speicherdaten aus dem Cache bedient werden können, im Vergleich zu Zugriffen auf den Hauptspeicher, besonders energieeffizient. Dies liegt an der deutlich schnellere Zugriffszeit von Cache-Speichern.
Für das Entwickeln von Programmen ist von Bedeutung, dass es im Bezug auf den Energieverbrauch sinnvoller ist, Datensätze in der jeweiligen Reihenfolge anzuordnen, in der sie für die Ausführung benötigt werden. Ein anschließend sequentieller Zugriff auf diese Daten würde dazu führen, dass Speicherdaten häufiger aus dem Cache bedient werden und die Ausführung des Programms sich somit energieeffizienter gestaltet.
Im Bezug auf den Energieverbrauch des Hauptspeichers ist zum einen ebenfalls festgestellt worden, dass dieser davon abhängig ist, ob der Prozessor die Daten aus dem Hauptspeicher oder aus dem Cache verarbeitet. Zum anderen wird der Energieverbrauch durch weitere Faktoren beeinflusst, die mit den in dieser Arbeit eingesetzten Messmethoden jedoch nicht ergründet werden konnte.
Die Analyse des Energieverbrauchs hat außerdem ergeben, dass die Ausführung von Programmen, bei mehreren zur Verfügung stehenden NUMA-Knoten, auf einem dieser Knoten

energieeffizienter ist. Demnach ist es empfehlenswert, moderne Multiprozessorsysteme auf diese Besonderheit zu überprüfen. Des Weiteren kann bei modernen Computersystemen durch die Aufteilung eines Programms in mehrere Bestandteile Energie gespart werden, indem diese gleichzeitig verarbeitet werden.
Ein weiterer denkbarer Ansatz zur Reduzierung des Energieverbrauchs, welcher durch die Zugriffe auf Speicherdaten verursacht wird, ist die Modifizierung moderner Multiprozessorsysteme durch weitere Zwischenspeicher. Da jedoch die SRAM-Speicherzellen, die für Zwischenspeicher eingesetzt werden, sowohl in der Herstellung als auch in der Realisierung mit Mehrkosten verbunden sind, ist eine mögliche Einsparung von dem Einsatzgebiet solcher Prozessoren abhängig.

Abbildungsverzeichnis

2.1 Modell einer Von-Neumann-Architektur [3] . . . 3
2.2 Arbeitsweise der CPU [4] . . . 4
2.3 Adressierung von Speicherzellen . . . 5
2.4 Speicherhierarchie [7] . . . 6
2.5 Aufbau einer dynamischen Speicherzelle [3] . . . 8
2.6 Aufbau einer statischen Speicherzelle [6] . . . 8
2.7 Das Zwischenspeichern und Wiederverwenden eines Datums im Cache [8] . . . 9
2.8 Direkt-abgebildeter Cache [11] . . . 11
2.9 2-fach assoziativer Cache [11] . . . 12
2.10 UMA-Multiprozessormodell [13] . . . 13
2.11 Beispiel eines NUMA-Systems mit zwei Knoten[13] . . . 14

4.1 Energieverbrauch der CPU (Thread:1, Speicher:1, sequentieller Zugriff) . . . 24
4.2 Zugriffszeit (Thread:1, Speicher:1, sequentieller Zugriff) . . . 24
4.3 Zugriffszeiten und Energieverbrauch der CPU (Thread:1, Speicher:1, sequentieller Zugriff) . . . 26
4.4 Leistungsaufnahme der CPU (Thread:1, Speicher:1, sequentieller Zugriff) . . . 26
4.5 Energieverbrauch der CPU (Thread:1, Speicher:1, zufälliger Zugriff) . . . 28
4.6 Leistungsaufnahme der CPU (Thread:1, Speicher:1, zufälliger Zugriff) . . . 29
4.7 Energieverbrauch der CPU (Thread:1, Speicher:0, sequentieller Zugriff) . . . 29
4.8 Zugriffszeiten (Thread:1, Speicher:0, sequentieller Zugriff) . . . 30
4.9 Leistung der CPU (Thread:1, Speicher:0, sequentieller Zugriff) . . . 31
4.10 Zugriffszeiten und Energieverbrauch der CPU (Thread:1, Speicher:0, zufälliger Zugriff) . . . 31
4.11 Energieverbrauch der CPU mit zufälligem Zugriffsmuster . . . 32
4.12 Energieverbrauch des Hauptspeichers (Thread:1, Speicher:1, sequentieller Zugriff) . . . 33
4.13 Energieverbrauch des Hauptspeichers (Thread:1, Speicher:0, sequentieller Zugriff) . . . 34
4.14 Energieverbrauch des Hauptspeichers (Thread:1, Speicher:0, zufälliger Zugriff) 36

4.15 Energieverbrauch des Hauptspeichers (Thread:0, Speicher:1, zufälliger Zugriff) 37
4.16 Energieverbrauch des Hauptspeichers (Thread:0, Speicher:1, sequentieller Zugriff) 38
4.17 Leistungsaufnahme des Hauptspeichers (zufälliger Zugriff) 39
4.18 Leistungsaufnahme des Hauptspeichers für gleiche Lokalität des Threads und des Speichers (sequentieller Zugriff) 39
4.19 Energieverbrauch der CPU bei Multithreading (Thread:0, Speicher:0, sequentieller Zugriff) 41
4.20 Energieverbrauch des Hauptspeichers bei Multithreading (Thread:0, Speicher:0, sequentieller Zugriff) 42
4.21 Zugriffszeit bei Multithreading (Thread:0, Speicher:0, sequentieller Zugriff) . 43

Literaturverzeichnis

[1] Tanenbaum, Andrew: *Moderne Betriebssysteme.* Pearson Studium, MünchenBoston u.a, 2009.

[2] Blesl, Markus: *Energieeffizienz in der Industrie.* Springer Vieweg, Berlin, 2013.

[3] *Computer Architektur.* `http://www.elektronik-kompendium.de/sites/com/1309261.htm`. Zugegriffen am 22.04.2015.

[4] *Aufbau der CPU.* `http://chemnitzer.linux-tage.de/2001/vortraege/slides/munke/cpu.jpg`. Zugegriffen am 22.04.2015.

[5] Oberschelp, Walter: *Rechneraufbau und Rechnerstrukturen.* Oldenbourg, München u.a, 2006.

[6] Drepper, Ulrich: *What every programmer should know about memory.* 2007.

[7] *Speicherhierarchie.* `http://christianherta.de/lehre/informatik/speicherhierarchie.php`. Zugegriffen am 02.04.2015.

[8] Pfister, Gregory F: *In search of clusters.* Prentice-Hall, Inc., 1998.

[9] González, Antonio, Carlos Aliagas und Mateo Valero: *A data cache with multiple caching strategies tuned to different types of locality.* In: *25th Anniversary International Conference on Supercomputing Anniversary Volume*, Seiten 217–226. ACM, 2014.

[10] Meyer, Ulrich: *Algorithms for memory hierarchies advanced lectures.* Springer, Berlin New York, 2003.

[11] Ersfeld, Christoph: *Organisation von Caches.* 2002.

[12] Jacob, Bruce: *Memory systems cache, DRAM, disk.* Morgan Kaufmann Publishers, Burlington, MA, 2008.

[13] *Optimizing Applications for NUMA.* `https://software.intel.com/en-us/articles/optimizing-applications-for-numa`, 2011. Zugegriffen am 27.04.2015.

[14] Lameter, Christoph: *Local and remote memory: Memory in a Linux/NUMA system.* 2006.

[15] Tanenbaum, Andrew: *Computerarchitektur : Strukturen, Konzepte, Grundlagen.* Pearson Studium, München u.a, 2006.

[16] *Intel® Performance Counter Monitor V2.7.* `www.intel.com/software/pcm`, 2014.

[17] *Intel Xeon E5-2690 specifications.* `http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2690.html`; Zugegriffen am: 29.04.2015.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 26. Mai 2015

Nikolaus Madalinski