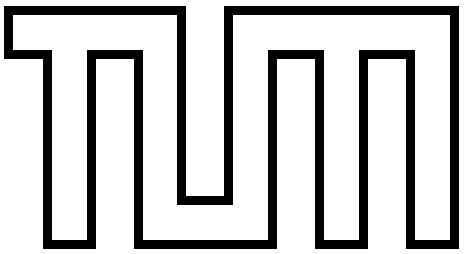

# Technische Universität München

# Fakultät für Informatik

Assemblerbasierte Optimierung für EPIC- und CISC-Architekturen bei iterativen numerischen Codes

Diplomarbeit

**Manfred Hauser**

**Aufgabensteller:** Prof. Dr. Arndt Bode

**Betreuer:** Dr. Josef Weidendorfer

**Abgabedatum:** 15. Juli 2004

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München den 15. Juli 2004

_____

Manfred Hauser

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 DiME

Das DiME[1] Projekt ist eine Zusammenarbeit des Lehrstuhls Informatik X „Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur“ an der Technischen Universität München mit dem Lehrstuhl für Informatik 10 „Systemsimulation„ an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Im Rahmen des Projekts werden die Möglichkeiten der Optimierung bei der Verwendung von Assoziativspeichern[2] in Hinblick auf iterative Verfahren wie dem Mehrgitterverfahren[3] untersucht.

Dieser Gleichungslöser muß bei umfangreichen Problemen sehr große Datenmengen verarbeiten. Somit stellt sich die Frage, welche Optimierungen bei gegebenem Algorithmus möglich sind. Gängige Optimierungen, die auf Hochsprachenebene eingesetzt werden können, wie beispielsweise das Umsortieren der verwendeten Daten, wurden bereits untersucht und werden auch eingesetzt. Hier scheint mit dem verwendeten Algorithmus keine weitere Optimierung mehr möglich. Dennoch sind weitere Optimierungen denkbar, wenn man sich auf Assemblerebene begibt. Hier stehen sämtliche Möglichkeiten der verwendeten Hardware offen, weshalb sich eine Untersuchung von Optimierungsmöglichkeiten in diese Richtung anbietet.

[1] Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations
[2] engl.: Caches
[3] Hierbei handelt es sich um einen effizienten Löser für Poisson-Gleichungen.

## 1.2 Problemstellung

Diese Arbeit entstand im Rahmen des DiME Projekts. Dabei galt es zu untersuchen, inwieweit eine Optimierung auf Assemblerebene durchführbar ist, was damit zu erreichen ist und ob ein Einsatz im täglichen Gebrauch sinnvoll und möglich ist. Hierzu war es notwendig, mögliche Optimierungen auf Hochsprachenebene zu untersuchen und deren Einsatzmöglichkeiten festzustellen. Nach erfolgter Optimierung auf Hochsprachenebene konnte dann eine weitere Optimierung mithilfe von Assembler innerhalb des Quellcodes, soweit dies vom Compiler unterstützt wird, oder aber innerhalb des, vom Compiler erzeugten Assemblercodes, stattfinden.

Gleichzeitig sollte diesbezüglich ein Vergleich zweier unterschiedlicher Architekturen erfolgen, der IA32- und der IA64-Architektur von Intel. Es sollte gezeigt werden, worin die Unterschiede der Optimierung auf Assemblerebene bestehen und ob sich diese für beide Architekturen eignet.

## 1.3 Kapitelübersicht

Kapitel 2 gibt einen Einblick in die moderne Prozessorarchitektur. Es werden allgemeingültige Eigenschaften beschrieben und anhand zweier unterschiedlicher Architekturen dargestellt.

In Kapitel 3 werden grundlegende Optimierungsstrategien aufgezeigt und Optimierungen innerhalb einzelner Entwicklungsebenen beschrieben.

Anhand einer Reihe von Messungen werden in Kapitel 4 die möglichen Bandbreiten der einzelnen Speicherebenen ermittelt. Auch ein kurzer Überblick über die verwendeten Werkzeuge ist hier zu finden.

In Kapitel 5 wird das Prinzip beschrieben, das dem Mehrgitterverfahren zugrundeliegt. Außerdem werden diesbezüglich Messungen auf beiden zu untersuchenden Architekturen durchgeführt.

Kapitel 6 enthält abschließende Bemerkungen über den Sinn des Einsatzes von Optimierungen auf Assemblerebene, die auf den Erfahrungswerten basieren, die während der Optimierung und Messung gewonnenen wurden.

Einen Ausblick über weitere Entwicklungen auf Soft-, wie auf Hardwareebene ist in Kapitel 7 zu sehen.

# Kapitel 2

# Moderne Prozessorarchitekturen

In diesem Kapitel soll ein Überblick über die verwendeten Architekturen gegeben werden. Weiter sollen grundlegende Details moderner Prozessoren vorgestellt werden, die auch in späteren Kapiteln zum Verständnis der Programme vorausgesetzt werden.

## 2.1 Caches und Datenlokalität

Da, gemessen an der Prozessorgeschwindigkeit, die Zugriffszeiten auf den Hauptspeicher sehr groß sind, besitzen die meisten heute zum Einsatz kommenden Prozessorarchitekturen[1] Speicherhierarchien. Abbildung 2.1 zeigt ein Beispiel einer solchen Hierarchie. Die Assoziativspeicher, die sich zwischen den Registern und dem Hauptspeicher befinden, nennt man *Caches*. Um diese leichter unterscheiden zu können, werden sie vom Prozessor in Richtung Arbeitsspeicher aufsteigend nummeriert. Mit zunehmender Nähe zum Prozessorkern nimmt die Geschwindigkeit dieser Caches immer mehr zu und so sind auch meist der *First Level Cache* und der *Second Level Cache* direkt mit im Prozessorkern[2] untergebracht. Ein *Third Level Cache*[3] befindet sich häufig außerhalb des Prozessors wie beim Itanium 2-Prozessor [Sch04], findet aber immer mehr Einzug direkt in den Prozessorkern, wie beispielsweise beim Xeon-Prozessor [Int04d]. Da die Assoziativspeicher mit zunehmender Nähe zum Prozessorkern immer kleiner werden, können sie nicht den gesamten Hauptspeicherinhalt fassen. Sie enthalten le-

[1] engl.: instruction set architecture, oder kurz ISA
[2] engl.: Die
[3] Als Kurzbezeichnungen findet man in der Literatur oft auch die Bezeichnungen L1-Cache für *First Level Cache*, L2-Cache für *Second Level Cache* und L3-Cache für TLC. Diese sollen hier auch in Zukunft entsprechend Verwendung finden

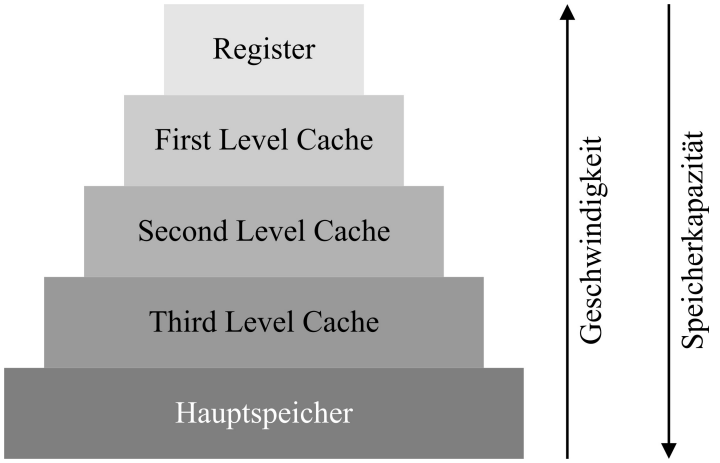


Abbildung 2.1: Speicherhierarchie moderner Prozessoren

diglich eine Kopie eines Teils der darunter liegenden Ebenen. Abhängig von der zum Einsatz kommenden Strategie enthält ein Cache nicht immer eine echte Teilmenge der Daten der nachfolgenden Ebene. Um das Potential, das diese schnellen Speicher bieten, effektiv zu nutzen, müssen gewisse Anstrengungen unternommen werden.

### 2.1.1 Räumliche Datenlokalität

Das Prinzip der räumlichen Datenlokalität geht davon aus, daß, falls eine gewisse Speicherstelle angesprochen wird, in Kürze deren benachbarte Speicherzellen ebenfalls verwendet werden. Dies wird vom Cache ausgenutzt, indem bei erfolgtem Speicherzugriff nicht nur diese eine Zelle[1] sondern weitere Zellen in deren Umkreis geladen werden. Sei $n$ eine Speicherzelle[2], auf die ein Zugriff stattfindet, so wird der Bereich von $\lfloor \frac{n}{CLS} \rfloor \cdot CLS$ bis $(\lfloor \frac{n}{CLS} \rfloor + 1) \cdot CLS - 1$ in den Cache geladen[3], und steht so unmittelbar zur Verarbeitung zur Verfügung.

Wird in einfachen Codestücken wie Listing 2.1 dieses Prinzip der räumlichen Datenlokalität meist ganz von selbst genutzt, so muß bei komplexeren Programmen wie Listing 2.2 explizit darauf geachtet werden. Allein die Änderung des Durchlaufes durch das Feld in der zweiten Schleife in Listing 2.3 verursacht einen beträchtlichen Geschwindigkeitsverlust, vgl. hierzu Abbildung 2.2[4] , da zwar insgesamt gesehen diesel-

[1] der Allgemeinheit wegen wird hier von Zellen gesprochen, da Speicherzugriffe nicht zwingend auf Bytes erfolgen
[2] die Speicherzählung beginne der Vollständigkeit halber bei 0
[3] $CLS$ steht hier stellvertretend für die Größe der *Cache Line* in Zellen
[4] Die Werte wurden mit dem Unix-Kommando *time* ermittelt

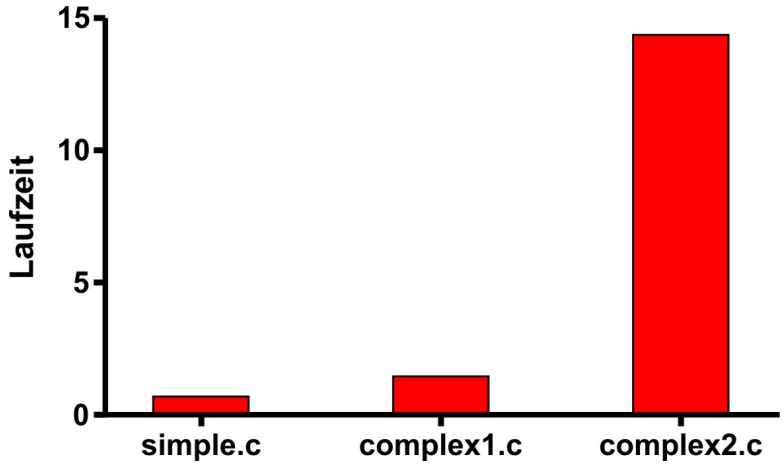


Abbildung 2.2: Vergleich zum Thema Datenlokalität

ben Speicherzellen angesprochen werden, jedoch in einer dem Prinzip der räumlichen Datenlokalität widersprechenden Art und Weise. Hierbei ist es hilfreich zu wissen, wie solche Felder im Speicher verwaltet werden, einen kleinen Einblick wie dies beispielsweise in der Programmiersprache *C* implementiert ist, gibt Abbildung 2.3. Es ist zu erkennen, daß in Listing 2.2 die Speicherzellen $f_0$, $f_1$, $f_2$, ... der Reihe nach angesprochen werden, während in Listing 2.3, welches eben dieses Prinzip der räumlichen Datenlokalität mißachtet, die Speicherzellen in einer anderen Reihenfolge und zwar $f_0$, $f_n$, $f_{2n}$, ..., $f_1$, $f_{n+1}$, ... angesprochen werden.

Listing 2.1: simple.c

```
1  int array[100000000];
2  int main () {
3    int i, k = 0;
4
5    // Initialisierung
6    for (i=0; i<100000000; i++)
7      array[i] = 1;
8
9    // Hauptteil
10   for (i=0; i<100000000; i++)
11     k += array[i];
12
13   printf("%d\n",k);
14 }
```

Listing 2.2: complex1.c

```
int array[10000][10000];
int main () {
  int i, j, k = 0;

  // Initialisierung
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      array[i][j] = 1;

  // Hauptteil
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      k += array[i][j];

  printf("%d\n",k);
}
```

Listing 2.3: complex2.c

```
int array[10000][10000];

int main () {
  int i, j, k = 0;

  // Initialisierung
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      array[i][j] = 1;

  // Hauptteil
  for (j=0; j<10000; j++)
    for (i=0; i<10000; i++)
      k += array[i][j];

  printf("%d\n",k);
}
```

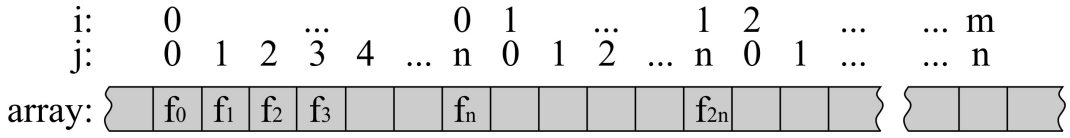


Abbildung 2.3: Speicherposition eines Feldes array[i][j]

### 2.1.2 Zeitliche Datenlokalität

Als nächstes wird das Prinzip der zeitlichen Datenlokalität betrachtet. Beim Zugriff auf eine Speicherzelle wird diese und ihre Nachbarzellen, wie beschrieben, in den L1-Cache geladen. Wichtig ist nun, diese solange wie möglich im L1-Cache oder zumindest im L2-Cache zu halten. Dadurch können erneute Zugriffe darauf schneller bearbeitet werden, als wenn diese erst erneut aus dem Hauptspeicher geladen werden müssten. Da die Caches in der Regel sehr klein sind, kann es natürlich passieren, daß ein bestimmter Speicherbereich bereits wieder aus dem Cache entfernt wurde, um einem anderen Platz zu machen. Hier ist einerseits der Softwareentwickler gefragt. Er muß darauf achten, daß die Zeit bis zur Wiederverwendung[1] einer Speicherzelle so kurz wie möglich ist. Da dies nicht unbegrenzt machbar ist, kann andererseits der Prozessorentwickler Hilfestellung leisten. Hierzu gibt es verschiedene Verfahren. Hier unterscheidet man verschiedene Cache-Strukturen, wie in Abbildung 2.4 dargestellt, die im Folgenden kurz erklärt werden:

- *Direktabbildung*[2]
  Hierbei wird der Speicher direkt auf den Cache abgebildet. Das bedeutet, daß es für eine bestimmte Speicherzelle genau eine mögliche Position im Cache gibt, an welche diese abgelegt werden kann. Die *Cache Line*, in die eine gewisse Speicherstelle $n$ geladen wird, errechnet sich wie folgt: $\lfloor \frac{n}{CLS} \rfloor mod \frac{CS}{CLS}$[3]. Es handelt sich im Grunde dabei um eine 1-Wege-Assoziativität, doch hierzu gleich mehr.

- *Mehrwege-Assoziativität*[4]
  Hier wird der Cache in Blöcke (*Sets*) unterteilt. Jedes solche Set enthält mehrere *Cache Lines*, in die eine Speicherzelle abgelegt werden kann, wobei die Anzahl der *Cache Line* je Set den Grad der Assoziativität angibt. Für eine geladene Speicherstelle $n$ gilt, daß sie in das Set mit der Nummer $\lfloor \frac{n}{CLS} \rfloor mod \frac{CS}{CLS \cdot ASSOC}$ geladen

[1] engl.: reuse distance
[2] engl.: direct mapped
[3] Die Zählung beginne wieder bei 0, $CLS$ stehe für die Größe der *Cache Line* und $CS$ für die Größe des Caches in Byte
[4] engl.: n-way associativity

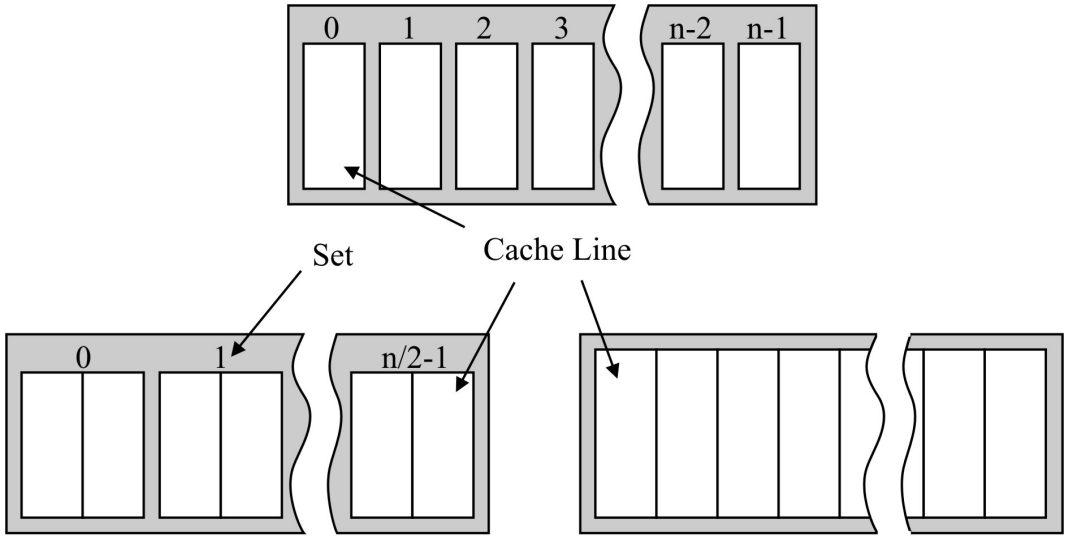


Abbildung 2.4: Direct-Mapped Cache (oben), 2-Wege-assoziativer Cache (links unten) und vollassoziativer Ache (rechts unten)

wird, wobei $ASSOC$ für die Assoziativität steht. Mittels $\frac{CS}{CLS \cdot ASSOC}$ errechnet sich die Anzahl der vorhandenen Sets.

- *Vollassoziativität*
  Beim vollassoziativen Cache gibt es keine Unterteilung des Caches, außer in die einzelnen *Cache Lines* selbst. Man kann den Cache also auch als ein einziges Set verstehen, welches sämtliche *Cache Lines* enthält, d.h. $ASSOC = \frac{CS}{CLS}$.

### 2.1.3 Ersetzungsstrategien

Beim mehrwege- und vollassoziativen Cache gibt es jedoch ein Problem. In welche *Cache Line* soll ein geladener Speicherblock abgelegt werden? Hierzu gibt es mehrere Lösungen, von denen einige im Folgenden aufgeführt sind:

- *Zufallsersetzung*[1]
  Am Einfachsten ist es, per Zufall eine der zutreffenden *Cache Lines* auszuwählen und den Speicherblock hier abzulegen.
- *Ersetzung der am längsten nicht mehr benutzten* Cache Line[2]
  Die am Längsten nicht mehr benutzte *Cache Line* eines Sets wird verdrängt und

[1] engl.: random
[2] engl.:least recently used, oder kurz *LRU*

der Speicherblock wird dort abgelegt. Diese Implementierung ist aufwendiger, da vermerkt werden muß, auf welche *Cache Lines* Zugriffe erfolgt sind.

- *Ersetzung der am wenigsten benutzten* Cache Line [1]
  Schließlich besteht die Möglichkeit den Inhalt jener *Cache Line* zu verdrängen, die am Wenigsten benutzt wurde. Auch dies erfordert eine aufwendige Implementierung, da die einzelnen Zugriffe vermerkt werden müssen.

Optimal wäre es im Grunde, der Cache wüsste im Voraus, welcher Speicherblock am Längsten nicht mehr benötigt wird, so daß dieser verdrängt werden kann[2]. Prinzipiell ist es möglich, ein Programm durchlaufen zu lassen und die nötigen Daten zu sammeln, was auch gemacht wird. Da sich vor allem auf Einprozessorsystemen mehrere Prozesse und das Betriebssystem einen Prozessor teilen, und sich das Programm je nach Benutzereingabe unterschiedlich verhalten kann, ist ein theoretisches Optimum in der Praxis nicht zu erreichen.

### 2.1.4 Fehlzugriffe

Da der Cache kleiner als der Hauptspeicher ist, kommt es vor, daß sich beim Zugriff auf eine Speicherstelle diese nicht im Cache befindet, und dadurch ein Fehlzugriff[3] auftritt. Man unterscheidet die Fehlzugriffe nach ihrere Ursache: [Wei01]:

- *compulsory* oder *cold misses*
  Beim ersten Referenzieren einer Speicherstelle tritt ein *compulsory* oder *cold miss* auf, da sich in diesem Falle die Daten noch nicht im Cache befinden können. Diese Art der Fehlzugriffe kann verringert werden, indem größere Blöcke aus dem Hauptspeicher geladen werden, was jedoch zu längeren Ladezeiten aufgrund der beschränkten Busbreite führen kann. Sollen mehr Daten geladen werden, als die Busbreite erlaubt, so müssen diese nacheinander übertragen werden, was einen höhere Übertragungszeit verursacht.

- *capacity misses*
  Da ein Assoziativspeicher, aufgrund seiner eingeschränkten Größe, unter Umständen nicht alle von einem Programm benötigten Daten aufnehmen kann, kommt es zum Auslagern von Blöcken, die später wieder angefordert werden. Diese Art von Fehlzugriffen nennt man *capacity misses*. Die Anzahl der *capacity*

[1] engl.: least frequently used, oder kurz *LFU*
[2] in der Literatur findet man hierfür oft den Namen *OPT*
[3] engl.: miss

*misses* kann durch Vergrößern des Cachespeichers vermindert werden, was aber zu einer langsameren Zugriffszeit führen kann, da festgestellt werden muß, ob und an welcher Stelle sich ein gewisses Datum im Cache befindet. Dies erfordert entweder einen höheren Aufwand an Hardware, damit diese Prüfung parallel und ohne Zeitverlust für alle Sets des Caches geschehen kann, oder aber sie muß für einzelne Teile des Cachespeichers nacheinander ausgeführt werden.

- *conflict misses*
  *Conflict misses* sind Fehlzugriffe, die nur in direkt abbildenden oder mehrwege-assoziativen Caches auftreten, da Speicherblöcke um dieselbe Position im Cache konkurrieren. In einem vollassoziativen Cache können diese Fehlzugriffe nicht auftreten, was bedeutet, daß deren Anzahl durch eine Erhöhung der Assoziativität verringert werden kann. Ermitteln kann man diese Fehlzugriffe durch direkten Vergleich des zu überprüfenden Assoziativspeichers mit einem äquivalenten aber vollassoziativen Cache. Tritt ein Fehlzugriff in beiden Caches auf, so handelt es sich um einen *capacity miss*. Tritt er nur im zu prüfenden Cache auf, so ist es ein positiv zu zählender *conflict miss*, während es sich um einen negativ zu zählenden *conflict miss* handelt, wenn der Fehlzugriff ausschließlich im vollassoziativen Cache auftritt.

Alle gemachten Beschreibungen beziehen sich im Übrigen auf Daten-Caches, obgleich vieles auch auf Instruktions-Caches zutreffen mag, da die Optimierung von Multigridlösern untersucht werden soll. Diese besitzen in der Regel nur kleine Code-Stücke, die lange durchlaufen werden, was eine Untersuchung des Quellcodes leicht bestätigen kann. Aus diesem Grund ist eine Optimierung in diesem Bereich eher zweitrangig.

## 2.2 Virtuelle Adressierung

Während im Grunde bisher immer von der typischen Speicher-Cache-Hierarchie die Rede war, existieren noch weitere Caches im Prozessor, die im Rahmen der Performance eine große Rolle spielen.

Typische Prozessorarchitekturen, wie der Pentium oder auch der Itanium 2, erlauben die Verwaltung des Hauptspeichers in Form von Seiten[1]. Diese Seiten besitzen auf den untersuchten Rechnerarchitekturen in der Regel eine Größe von 4 KByte bzw. 16 KByte, obgleich auch Seiten von beispielsweise 4 MByte möglich sind. Diese Seiten

[1] engl.: pages

dienen der Nutzung von virtuellem Speicher. Mit Hilfe von seitenbasierten Adressen kann ein Betriebssystem jedem Prozess einen eigenen virtuellen Adressraum zuweisen. Gleichzeitig wird es möglich, mehr Hauptspeicher zu nutzen, als tatsächlich vorhanden, da nicht benötigte Seiten vom Betriebssystem auf einen Datenträger, in der Regel eine Harddisk, ausgelagert werden können. Außerdem kann dadurch eine Seite an beinahe jeder beliebigen Stelle im Hauptspeicher zum Liegen kommen, mit der Einschränkung das diese im Speicher ausgerichtet sein muß, beispielsweise auf 4 KByte Grenzen. Beim Wiedereinlagern der Seite ist es also nicht nötig, ihren ursprünglichen Platz freizumachen. Sie kann irgendwo abgelegt werden und es ist lediglich ihr Eintrag in der Seitentabelle[1] entsprechend anzupassen.

Wie in [Mes00] beschrieben, liegt der Vorteil gegenüber der Auslagerung ganzer Segmente darin, daß sich die kleinen Seiten sehr schnell einlesen oder abspeichern lassen. Außerdem nutzen viele Programme das Prinzip der räumlichen Datenlokalität aus und greifen somit meist auf nahe beieinander liegenden Daten zu. Dadurch ist die Wahrscheinlichkeit sehr groß, daß sich die gewünschten Daten im Speicher befinden, obwohl die Seitengröße im Vergleich zum möglichen Adressraum auf 32 Bit -Prozessoren von 4 GByte um einen Faktor $\frac{2^{32}}{2^{12}} = 2^{20} \approx 10^6$ kleiner ist. Im Grunde kann man davon sprechen, daß der virtuelle Speicher den tatsächlich vorhandenen physikalischen Speicher seinerseits als Cache verwendet.

Bei der Verwendung von Seitentabellen treten jedoch zwei Probleme auf[Tan02]:

- Die enorme Größe der Seitentabelle
- Die Umrechnung von virtueller zu physikalischer Adresse muß schnell sein

Ersteres ergibt sich aus der Tatsache, daß moderne Rechnerarchitekturen einen Adressbus von wenigstens 32 Bit besitzen, womit eine Speichergröße von bis zu 4 GByte verwaltet werden kann, was bei einer Seitengröße von 4 KByte etwas über $10^6$ Seiten entspricht. Bei einer Adressbreite von 64 Bit ergibt sich bereits eine mögliche Seitenzahl von mehr als $4 \cdot 10^{15}$.

Beobachtungen von Programmen lassen darauf schließen, daß die meisten Programme ihre Daten innerhalb eines kleinen, sehr begrenzten Bereichs ablegen. Um nicht alle Seitentabellen im Speicher halten zu müssen, was angesichts der riesigen Menge an möglichen Seiten bei 64 Bit -Prozessoren ohnehin nicht mehr realisierbar ist, ging man den Weg über mehrstufige Seitentabellen, wie in Abbildung 2.5 dargestellt. Nicht

[1] engl.: page table

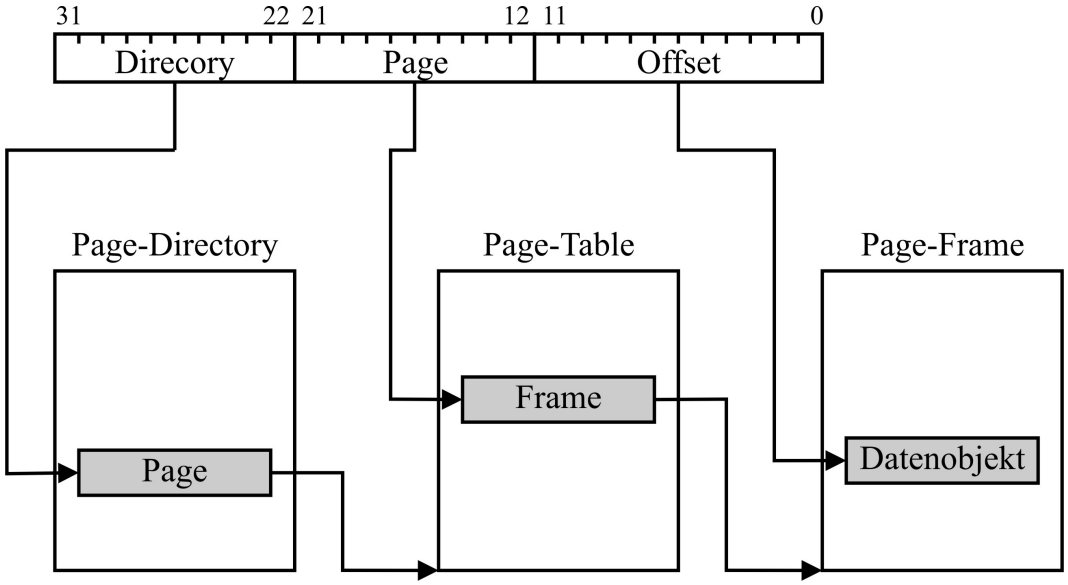


Abbildung 2.5: Typisches Paging gängiger Prozessoren mit 32 Bit Adressbus

benötigte Seitentabellen können somit ausgelagert werden und belegen nicht unnötig Hauptspeicher.

Beim Pentium, ebenso wie beim Xeon, besteht eine Speicherreferenz aus einem 32 Bit -Wort. Dieses ist nach der Umwandlung mittels Descriptor-Table von einer virtuellen in eine lineare Adresse in eine Referenz für die Directory-Page, eine Referenz auf die eigentliche Seite und schließlich den Offset des Datums unterteilt. In Abbildung 2.5 ist dies nochmals graphisch dargestellt.

Das zweite Problem ergibt sich, da die Umrechnung von der virtuellen zur physikalischen Adresse bei jedem Speicherzugriff stattfinden muß. Vor allem auf Prozessoren mit komplexen Anweisungen[1] besitzen viele Befehle, mit Ausnahme von beispielsweise reinen Register-Register-Operationen, einen oder mehrere Speicheroperanden. Somit lösen die meisten Befehle wenigstens einen Zugriff auf die Seitentabelle aus. Dauert der Zugriff auf die Seitentabelle zu lange, erweist sich dies sehr schnell als der Flaschenhals im System.

Das einfachste Design, diesem Problemen entgegen zu wirken, wäre eine einzige große Seitentabelle, die sich direkt in speziellen Hardwareregistern im Prozessor befindet. Wird ein neuer Prozess gestartet, oder dessen Bearbeitung wieder aufgenommen, wird dessen Seitentabelle in den Prozessor geladen. Dabei entsteht das Problem, einerseits

[1] engl.: complex instruction set computer, oder kurz CISC

die große Datenmenge im Prozessor unterbringen zu müssen und andererseits die enormen Datenmengen, die bei jedem Prozesswechsel anfallen, vom Prozessor in den Hauptspeicher und vom Hauptspeicher in den Prozessor zu bewältigen.

Das andere Extrem ist, die Seitentabelle gänzlich im Hauptspeicher zu belassen und nur ein einziges Hardwareregister vorzusehen, das den Beginn der Seitentabelle im Speicher beinhaltet. Das Problem dabei ist die lange Zugriffszeit auf die Daten im Speicher, die bei vielen Befehlen auftreten würde.

Zur Lösung dieses Problems wird meist ein Mittelweg beschritten, indem in vielen moderne Prozessorarchitekturen kleine Caches implementiert werden[1]. Der Pentium 4 wie auch der Xeon besitzen hier eine Reihe von Hardwareregistern. 64 dieser Register, die in vollassoziativer Weise belegt werden, dienen zur Aufnahme von Einträgen für Seiten die Daten beinhalten. Die Einträge für 4 KByte-Seiten und jene für 4 MByte-Seiten teilen sich diese Register. Weitere 128 Register, die in 4-Wege-assoziativer Weise beschrieben werden, stehen zur Verfügung um Einträge für Seiten, die Code enthalten, aufzunehmen. Befindet sich die Referenz auf eine Seite bei einem Zugriff darauf nicht im TLB, so wird ein TLB-Miss ausgelöst, der bei neueren Prozessorarchitekturen in der Regel vom Betriebssystem behandelt werden muß.

Ein sehr vereinfachtes Beispiel eines solchen TLB ist in Tabelle 2.1 zu sehen [Tan02]. Ein Prozess, welcher aus einer Schleife besteht, die sich in der virtuellen Seite 45 befindet, könnte beispielsweise den in Tabelle 2.1 angegebenen TLB erzeugen. Die vom Prozess verwendeten Daten könnten möglicherweise in der Form eines Feldes in der Seite 213 und 214 liegen. Seite 212 könnte dann die Array-Indizes enthalten, während der Stack des Prozesses in Seite 315 liegt.

## 2.3 Der Xeon-Prozessor

Auf dem bereits mehrfach zur Sprache gekommenen Xeon-Prozessor soll nun genauer eingegangen werden, schließlich werden darauf die Messungen vorgenommen, die in späteren Kapiteln folgen. Mittlerweile sind verschiedene Ausführungen von Xeon-Prozessoren erhältlich, dabei wird eine Einschränkung auf jenen Prozessortyp gemacht [Int03a], der bei den Messungen zum Einsatz kam.

[1] *Translation Lookaside Buffer*, oder kurz TLB

| Gültig | Verändert | Rechte | Virtuelle Seite | Seitenrahmen |
|---|---|---|---|---|
| 1 | 0 | RW | 212 | 52 |
| 1 | 0 | R X | 45 | 18 |
| 0 | 0 | R X | 59 | 95 |
| 1 | 0 | RW | 213 | 83 |
| 0 | 1 | RW | 140 | 19 |
| 1 | 1 | RW | 315 | 24 |
| | | | . | |
| | | | . | |
| | | | . | |
| 0 | 0 | R X | 49 | 36 |

Tabelle 2.1: Vereinfachte Darstellung eines *Translation Lookaside Buffer*

### 2.3.1 Der Cache

Der verwendete Xeon [Mai04] ist mit 2,4 GHz getaktet und besitzt, wie schon einige seiner Vorgänger, eine zweistufige Cachehierarchie. Sein L1-Cache besteht aus einem 8 KByte großen 4-Wege-assoziativen Datencache mit einer *Cache Line Size* von 64 Byte und einer sehr niedrigen Zugriffslatenzzeit von zwei Takten auf Integerwerte und neun Takten auf Fließkommawerte, sowie einen 8-Wege-assoziativen Cache[1] der bis zu 12K Mikroinstruktionen Platz bietet. Der Xeon verarbeitet x86-Anweisungen indem er diese in Mikroinstruktionen dekodiert und im *trace cache* ablegt. Intern werden diese Mikroinstruktionen wie in einem RISC-Prozessor verarbeitet. Der 8-Wege-assoziativer L2-Cache hat eine Größe von 256 KByte, oder wie im Falle des bei den Messungen verwendeten Xeon-Prozessors von 512 KByte, mit einer Latenzzeit von sieben Taktzyklen. Die *Cache Line Size* beträgt in beiden Fällen 64 Byte. Eine Besonderheit ist die *Sektorgröße* des L2-Cache von 128 Byte, was bedeutet, daß bei einem Zugriff auf den Speicher nicht nur eine *Cache Line* geladen wird sondern gleich zwei benachbarte. Bei neueren Varianten ist mittlerweile auch ein L3-Cache verfügbar, der bis zu 1 MByte [Int03b] umfaßt. Xeon-Prozessoren mit noch größerem direkt im Kern implementierten L3-Cache sollen folgen [WL04]. An den Hauptspeicher ist der Xeon mittels eines Systembusses, mit einer Breite von 64 Bit und einer Taktrate von 400 MHz angebunden. Hieraus errechnet sich eine theoretische Übertragungsrate von 3,2 GByte pro Sekunde.

[1] es handelt sich dabei um einen so genannten *trace cache*

### 2.3.2 Die Register

Der Xeon verfügt über acht Allzweckregister, sechs Segmentregister, ein Programmstatus- und Kontrollregister sowie über einen Instruktionszeiger. Er besitzt acht 64 Bit MMX Register, um mehrere Datenworte gleichzeitig bearbeiten zu können, sowie acht 128 Bit XMM Register für ähnliche Zwecke. Die Fließkommaeinheit verfügt über acht Register mit einer Breite von 80 Bit, deren Besonderheit es ist, daß sie in Form eines Stacks organisiert sind. Des weiteren sind noch diverse Register zum Verwalten von Tasks, für Segmentselektoren und Ähnliches mehr vorhanden. Interessant sind die Kontroll-Register, über die diverse Statusinformationen abgerufen werden können, die aber auch dazu dienen, verschiedene Prozessoreigenschaften zu verändern. Hierüber können beispielsweise Cache-Speicher deaktiviert werden. Verschiedene modellspezifische Register erlauben es, weitere Informationen über den Prozessor einzuholen.

### 2.3.3 Ereignisse und Vorausladememechanismen

Der Xeon-Prozessor kann dazu veranlaßt werden, gewisse Ereignisse, wie beispielsweise L2-Cache Misses, zu zählen. Über so genannte *Performance-Monitoring Counter*[1] können eine Reihe von Ergebnissen ausgelesen und ausgewertet werden [Int04d].

Der Xeon verfügt außerdem über die Möglichkeit, Daten vor dem eigentlichen Gebrauch bereits vorab in den Cache zu laden[2]. Hierbei kommen drei unterschiedliche Mechanismen zum Einsatz:

- Ein in die Hardware implementierter Mechanismus zum Laden von Instruktionen. Dieser lädt Instruktionen entlang eines Weges, der durch die Sprungvorhersage am wahrscheinlichsten erscheint.
- Ein weiterer in die Hardware implementierter Mechanismus zum automatischen Laden von Daten und Instruktionen in den L2-Cache. Dieser versucht, der aktuell zu lesenden Speicherposition immer 256 Byte voraus zu sein. Dieses Lesen eines Datenstroms[3] wird nur innerhalb der Grenze einer 4 KByte-Seite und jeweils nur für einen einzelnen Datenstrom unterstützt. Auf mehrere Seiten verteilt erkennt der Xeon aber bis zu acht solcher Datenströme, die gleichzeitig und unabhängig

[1] kurz PMCs
[2] engl.: *Prefetching*
[3] engl.: streaming

voneinander geladen werden. Die Daten werden nicht in den L1-Cache, sondern nur bis zum L2-Cache, bzw. L3-Cache geladen, da aufgrund der geringen Größe des L1-Cache dies zu enormen Performance-Einbußen führen kann.

- Ein dritter Mechanismus, der ausschließlich zum Vorausladen von Daten dient, und in zwei verschiedene Komponenten aufgeteilt ist: (1) einen hardwareseitig implementierten Mechanismus zum Laden der benachbarten *Cache Line* innerhalb eines 128 Bit breiten Sektors, für den Fall, daß ein Fehlzugriff auf den Cache eintritt. (2) einen steuerbaren Mechanismus, dem der Softwareentwickler durch spezielle Instruktionen Tips geben kann, welche Daten in die Assoziativspeicher zu laden sind. Dies geschieht durch diverse Instruktionen. *prefetchnta* gibt dem Xeon-Prozessor beispielsweise den Tip, ein bestimmtes Datum und seine benachbarten Speicherzellen nur in die erste *Cache Line* eines Sets im L2-Cache zu laden. *prefetcht2* hingegen versucht eine beliebige *Cache Line* im L2-Cache mit dem angegebenen Datum zu füllen.

## 2.4 Der Itanium 2

Parallel zur IA-32 des Pentiums entwickelte Intel in Zusammenarbeit mit Hewlett Packard eine IA-64 Architektur. Dabei handelt es sich um eine auf EPIC[1] basierende RISC Architektur. Die IA-64 definiert 128 Allzweckregister mit einer Breite von je 64 Bit, 128 Fließkomma-Register von 82 Bit Breite und 64 Predicat-Register bestehend aus je einem Bit. Diverse Spezialregister, wie 128 Applikationsregister für die Stack-Engine und den Kernel sind ebenso enthalten, wie acht Branch-Register und verschiedene ID- und Performance-Monitoring-Register.

### 2.4.1 Die Register

Die ersten 32 Allzweckregister werden statisch verwaltet. Das erste dieser Register, *r0* ist sogar konstant auf den Wert 0 verdrahtet. Die restlichen 96 Register sind so genannte dynamische Register. Diese können bei Bedarf in den Cache ausgelagert werden, so daß Unterprogramme diese erneut verwenden können. Dadurch lassen sich zeitaufwendige Push- und Pop-Operationen, wie in der IA-32, vermeiden. Außerdem bieten diese, ebenso wie auch die 96 dynamischen Fließkomma-Register, die Möglichkeit der

[1] Explicitly Parallel Instruction Computing

Rotation. Dadurch lassen sich beispielsweise Filterfunktionen bei der Bildbearbeitung beschleunigen. Berechnet man zum Beispiel die Farbe eines Bildpunktes anhand seiner Nachbarn, so ist es möglich, diese in die dynamischen Register zu laden und die Farbe des Punktes auszuwerten. Für den nächsten Bildpunkt müssen lediglich die dynamischen Register rotiert und nur noch einer der ursprünglichen Farbwerte nachgeladen werden.

Ähnlich verhält es sich mit den Fließkomma-Registern. Auch hier handelt es sich um 32 statische und 96 dynamische Register. So sind hier die ersten beiden Register, *f0* und *f1* konstant auf 0,0 bzw. 1,0 gesetzt, da diese Werte häufig benötigt werden. Im Gegensatz zum Xeon lassen sich jedoch die Fließkomma-Register direkt ansprechen und müssen nicht über einen Stack übergeben werden. Interessant ist auch die Tatsache [WL04], daß der Itanium 2 keinerlei Befehle zur Berechnung transzendenter Funktionen wie Logarithmus, Quadratwurzel oder Sinus zur Verfügung stellt. Nicht einmal die Division ist ihm bekannt. Somit müssen alle diese Funktionen vom Compiler nachgebildet werden. Durch die Eigenschaften des Prozessors, wie die parallele Ausführung von Programmteilen durch mehrere Fließkommaeinheiten und Softwarepipelining mit Registerrotation, ist die Fließkommaberechnung jedoch oft schneller als die von anderen Prozessoren, welche diese Befehle fest verdrahtet haben.

### 2.4.2 Parallelverarbeitung

Die Parallelverarbeitung wird beim Itanium 2 in Form von VLIW[1] unterstützt. Diese Bundles werden aus drei Einzelbefehlen zusammengesetzt, wobei der Itanium 2 bis zu zwei davon gleichzeitig verarbeiten kann.

Der Itanium 2 besitzt eine Reihe von funktionalen Einheiten unterschiedlichster Art, was eine Reihe an möglichen Kombinationen von zu verarbeitenden Operationen je Taktzyklus gestattet. Da nur maximal sechs Instruktionen je Taktzyklus zur Bearbeitung anstehen, wird nur ein kleiner Teil der im Itanium 2 implementierten funktionalen Einheiten je Takt verwendet.

Der Itanium 2 verfügt über sechs allgemeine ALU-Einheiten[2], zwei Integereinheiten, eine Einheit für Schiebeoperationen, vier Fließkommaeinheiten, vier Speichereinheiten, wobei diese unterteilt sind in zwei Einheiten für Lade- und zwei für Schreibzugriffe, sowie über eine ganze Reihe weiterer funktionaler Einheiten. Diese funktionalen

[1] Very Long Instrucion Words, beim Itanium 2 auch Bundles genannt
[2] Arithmetical Logical Unit

Einheiten können über elf mögliche Ports mit Befehlen beschickt werden. Vier davon werden für Speicheroperationen, zwei für Fließkommaoperationen, zwei für Integer- und drei für Sprungoperationen verwendet. Somit kann nicht jede erdenkliche Kombination der sechs möglichen Befehle je Taktzyklus verarbeitet werden. Sind alle Anschlüsse, die zu einer bestimmten Art von funktionalen Einheiten führen belegt, so wird die Abarbeitung der weiteren Befehle gestoppt, die anstehenden Befehle weitergereicht und eine Bündelrotation[1] durchgeführt. Konnten nicht beide Bundles abgearbeitet werden, wird nur ein neues Bundle nachgeladen und die Abarbeitung an der Stelle wieder angesetzt, an der sie im letzten Takt gestoppt wurde, andernfalls werden zwei neue Bundles nachgeladen.

Durch die Parallelverarbeitung der Befehle entsteht auch das Problem der Datenabhängigkeit, um das sich der Itanium 2 nicht kümmert. Der Compiler muß hierfür entsprechenden Code erzeugen. Dafür stellt der Itanium 2 Schablonen[2] zur Verfügung, die Trennbalken[3] enthalten, anhand derer der Itanium 2 die vermeintliche Datenabhängigkeit erkennen kann [Int04e]. Trifft der Itanium 2 im ersten Bundle auf solch einen Trennbalken, so wird hier die Befehlsabarbeitung angehalten und eine Bündelrotation durchgeführt, jedoch kein Bundle ersetzt. Befindet sich der Trennbalken im zweiten Bundle, wird eine Bündelrotation durchgeführt und das erste Bundle ersetzt, die Abarbeitung wird im nächsten Takt im zweiten Bundle fortgesetzt. Nur wenn kein Trennbalken gefunden wurde und die Abarbeitung aller sechs Befehle möglich war, wird eine doppelte Bündelrotation durchgeführt und beide Bundles ersetzt.

### 2.4.3 Vorausladen

Im Gegensatz zum Xeon unterstützt der Itanium 2 kein *hardware prefetching*. Hier ist der Compiler dafür zuständig, entsprechende Ladeoperationen anzustoßen. Während der Prozessor weiter rechnet findet parallel dazu das Laden von später benötigten Daten statt. Werden die Daten dann benötigt, sind sie bereits im Prozessor. Dabei stellt der Itanium 2 zwei unterschiedliche Arten zur Verfügung.

[1] engl.: bundle rotation
[2] engl.: Templates
[3] engl.: splits

#### 2.4.3.1 Register als Ziel

Der Itanium 2 bietet die Möglichkeit, Daten mit einem Register als Ziel vorauszuladen. Falls zwischen dem Anstoßen des Ladevorgangs und der Stelle, an der die Daten tatsächlich benötigt werden, der betreffende Speicherbereich verändert wird, so werden die im Voraus geladenen Daten verworfen und erneut eingelesen. Konkret findet also mittels eines Ladebefehls *ld.a* (Load Advanced), wie in Listing 2.4 dargestellt, eine Eintragung in die ALAT (Advanced Load Address Table) statt. Bevor der zu ladende Wert verwendet wird, erfolgt mittels *ld.c.clr* die Prüfung, ob dieser gültig ist. Ist der Wert rechtzeitig angekommen, kann er ohne Verzögerung verwendet werden, andernfalls wird er durch einen kostspieligen, aber unvermeidbaren Speicherzugriff nachgeladen.

Listing 2.4: Beispiel eines Vorab-Ladebefehls

```
1 ld8.a r21 = [r21];;
2 ...
3 mov r1=r40
4 mov r15=r8
5 adds r14=-104,r39;;
6 ld8.c.clr r21=[r21];;
7 adds r22=8,r21
8 adds r23=16,r21;;
```

#### 2.4.3.2 Cache Line als Ziel

Die zweite Möglichkeit, Daten im Voraus zu laden, bietet der Itanium 2 mit dem Befehl *lfetch* an. Dabei werden die Daten immer bis in die höchste verfügbare Cache-Hierarchie, also den L1-Cache geladen. Dennoch existieren verschiedene Möglichkeiten, das Laden der Daten und ihren Verbleib im Cache zu beeinflussen. Einerseits kann durch den optionalen Zusatz *fault* der Prozessor dazu veranlaßt werden, den Ladebefehl auch dann auszuführen, falls Fehler auftreten. Das bedeutet aber nicht nur, daß ausgelagerte Seiten eingelagert werden, sondern auch, daß Zugriffe auf nicht vorhandene Speicherbereiche einen Zugriffsfehler[1] auslösen. Andererseits kann durch Ver-

[1] engl.: segmentation fault

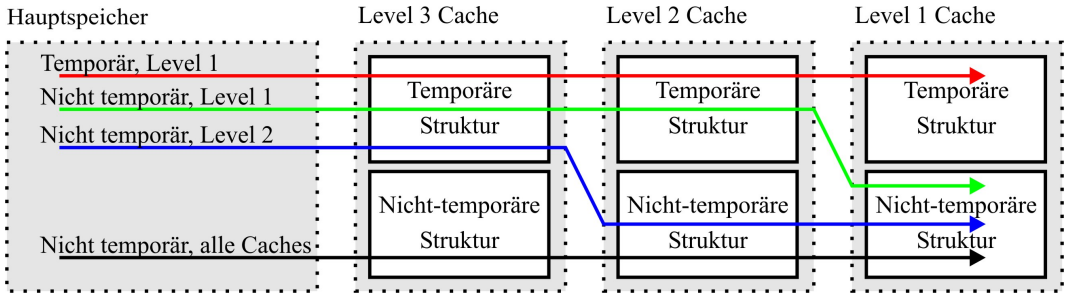


Abbildung 2.6: Weg der Daten durch die Speicherhierarchie bei der Verwendung des Befehls *lfetch*

wendung oder Weglassen[1] der Zusätze *nt1*[2], *nt2*[3] und *nta*[4], bestimmt werden, wo die Daten im Cache abgelegt werden sollen. Siehe hierzu auch Abbildung 2.6.

### 2.4.4 Spekulatives Laden

Der Itanium 2 bietet auch die Möglichkeit des spekulativen Ladens. Dabei wird versucht, ein Datum mittels des Befehls *ld.s* (Load Speculative) zu laden. Kann der Ladebefehl beispielsweise aufgrund einer fehlenden Speicherseite nicht durchgeführt werden, so wird das Register als ungültig markiert. Hierzu besitzen die 128 Allzweckregister ein zusätzliches Bit, mit der Bezeichnung *kein Ding*[5]. Im weiteren Programmablauf wird mit dem angegebenen Register gerechnet. Später erst wird mithilfe von *chk.s* (Check Speculation) überprüft, ob das Register, welches letztendlich mit dem Ergebnis der Berechnungen beschrieben wird, auch einen sinnvollen Wert enthält. Ist der Inhalt dieses Registers als *NaT* markiert oder enthält das Fließkommaregister den Wert *NatVal*, so wird über den *chk.s*-Befehl ein Sprung auf eine Korrektursequenz ausgeführt, die die Berechnung mit gültigen Werten erneut durchführt.

### 2.4.5 Der Cache

Zum Schluß des Kapitels betrachten wir noch die Cachehierarchie des Itanium 2. Sowohl beim L2-Cache als auch beim L3-Cache handelt es sich um gemeinsame Caches[6].

[1] Temporär, L1-Cache
[2] Nicht temporär, L1-Cache
[3] Nicht temporär, L2-Cache
[4] Nicht temporär, alle Caches
[5] engl.: Not a Thing, oder kurz NaT
[6] engl.: Unified Caches

Das bedeutet, daß diese sowohl Daten als auch Instruktionen speichern, während der L1-Cache in einen Daten- und Instruktionscache aufgeteilt ist. Der L1-Cache bezieht seine Daten aus dem L2-Cache, der L2-Cache und der L3-Cache können hingegen ihre Daten direkt über das Systeminterface aus dem Hauptspeicher laden. Während der L1-Cache eine 4-Wege-Assoziativität besitzt, ist es beim L2-Cache eine 8-Wege- und beim L3-Cache sogar eine 12-Wege-Assoziativität. Traditionelle Caches speichern die physikalische Adresse als Marke[1] zu jeder *Cache Line*, was bei jedem Zugriff auch einen Zugriff auf den TLB erfordert, um die virtuelle Adresse in ihr physikalisches Pendant umzuwandeln. Die Logik zur Treffererkennung[2] vergleicht anschließend die physikalische Adresse mit allen in den Cache-Tags gespeicherten. Da die Übersetzung und der Vergleich üblicherweise hintereinander ausgeführt werden, benötigt ein solcher Zugriff mehrere Taktzyklen. Beim Itanium 2 wurde dies für den L1-Cache wie auch für der L2-Cache anders gelöst. Bei ihnen handelt es sich um so genannte Prevalidated-Tag-Caches, wodurch beim L1-Cache eine Latenzzeit von nur einem einzigen Taktzyklus ermöglicht wird (der L2-Cache besitzt eine Latenzzeit von fünf oder mehr Takten, während ein Zugriff auf Daten im L3-Cache mindestens zwölf Takte beanspruchen). Dies funktioniert, da Prevalidated-Tag-Caches nicht die physikalische Adresse im Tag der zugehörigen *Cache Line* speichern, sondern einen Verweis auf den zugehörigen Eintrag im TLB. Erfolgt nun ein Speicherzugriff, wie in Abbildung 2.7 dargestellt, so wird dieser an den TLB weitergereicht. Dieser erzeugt daraus einen 32 Bit breiten Vektor, der genau ein 1-Bit[3] enthält, und zwar an der Stelle, an der die virtuelle Adresse mit dem TLB-Eintrag übereinstimmt. Die restlichen 31 Bits enthalten alle eine logische Null. Dieser Vektor wird nun parallel mit allen vom Mehrwege-Tag-Ram des betroffenen Sets gelieferten Tags bitweise verglichen. Ergibt sich durch diesen Vergleich ein Treffer, wird der Cache-Weg, der das entsprechende Datum enthält, ermittelt. Gleichzeitig hierzu werden die gespeicherten Datenwerte des Mehrwege-Caches parallel ausgelesen und die Daten des ermittelten Cache-Weges durch die nachfolgende Funktionseinheit weitergeleitet.

[1] engl.: tag
[2] engl.: *hit detection*
[3] dies ist ein so genannter one-hot vector

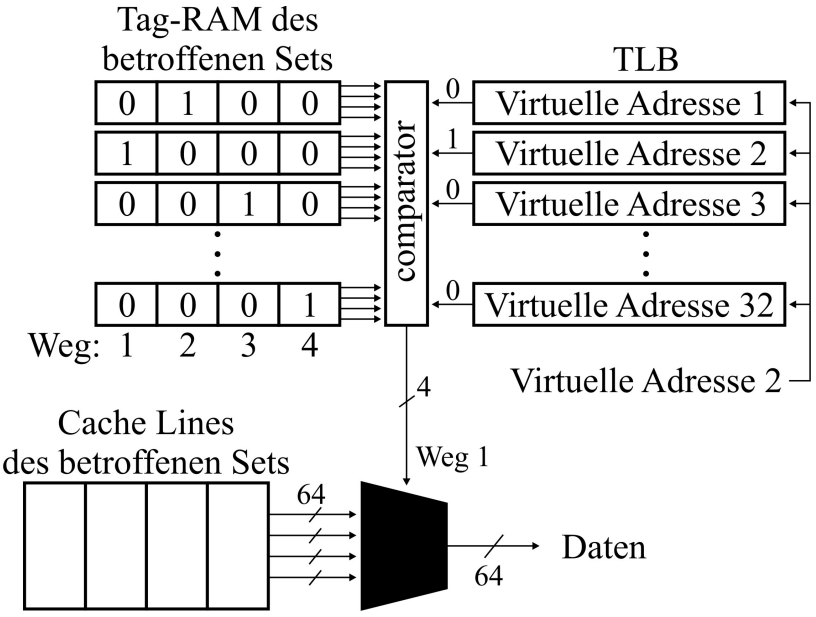


Abbildung 2.7: Ein Speicherzugriff beim Itanium 2

# Kapitel 3

# Optimierungen im Überblick

Ziel dieser Arbeit ist es, die Möglichkeiten assemblerbasierter Codeoptimierung auf EPIC- und CISC-Architekturen bei Multigrid-Verfahren zu untersuchen. Bevor dies im Detail näher betrachtet wird, Optimierungen direkt an Multigrid-Verfahren vorgenommen werden, ist es nötig sich mit den Grundlagen der Optimierung vertraut zu machen. Außerdem soll gezeigt werden, worum es bei Multigrid überhaupt geht und wie es funktioniert.

## 3.1 Grundlagen zur Optimierung

Prinzipiell gibt es viele verschiedene Möglichkeiten, Optimierungen vorzunehmen. Einige davon sollen genauer dargestellt werden.

### 3.1.1 Austausch des Algorithmus

Am Naheliegendsten ist wohl, einfach einen langsamen Algorithmus gegen einen schnelleren auszutauschen. In der Informatik gibt es viele Beispiele, die hier aufgeführt werden können.

Aus dem Bereich der Sortierverfahren dürfte jedem bekannt sein, daß die Laufzeit des Quicksort-Algorithmus unter bestimmten Voraussetzungen wesentlich niedriger ist, als die des Bubblesort-Algorithmus. Benötigt der Bubblesort-Algorithmus im Durchschnitt, wie auch im ungünstigsten Fall, etwa $\frac{N^2}{2}$ Vergleiche, so sind es beim Quicksort-Algorithmus im Mittel nur $2N \ln N$ [Sed95].

Beim Suchen in Zeichenfolgen gilt ähnliches. Dort bewährt sich beispielsweise der Algorithmus von Boyer-Moore gegenüber dem ganz grundlegenden Suchen durch byteweises Vergleichen sehr. Sind beim grundlegenden Suchen bis zu N·M Zeichenvergleiche notwendig, sind es beim Boyer-Moore-Algorithmus niemals mehr als N+M Vergleiche und es werden bei nicht kleinem Alphabet und kurzem Suchmuster in etwa N/M Schritte benötigt. N ist hierbei die Länge des Textes und M die Länge des zu suchenden Musters [Sed95].

### 3.1.2 Zusammenfassen von Schleifen

Hat man sich für einen Algorithmus entschieden, können auch innerhalb von diesem weitere Optimierungen vorgenommen werden. So beispielsweise durch das Zusammenfassen von Schleifen[1]. Hier werden unterschiedliche Schleifen zusammengefasst, die sich über denselben Laufbereich erstrecken, wie Listing 3.1 und Listing 3.2 zeigt. Spielt dies bei Schleifen, die unterschiedliche Daten verarbeiten, mit Ausnahme des vermeidbaren Overheads für den doppelten Schleifendurchlauf, keine sehr große Rolle, liegt die Sache bei unserem Beispiel etwas anders. Da hier beide Schleifen das Feld B bearbeiten, kann die Zeitersparnis bereits dadurch beträchtlich sein, da das mehrfache Laden der Daten in den Cache-Speicher entfällt. So benötigt Listing 3.1 eine Laufzeit von 2,10 Sekunden, Listing 3.2 hingegen nur 1,78 Sekunden und dies schon bei einem so einfachen Beispiel.

### 3.1.3 Vertauschen von Schleifen

Der Vollständigkeit halber soll nicht verschwiegen werden, daß auch das einfache austauschen von Schleifen[2] durchaus Laufzeitverbesserungen bewirken kann. Wie dies geschieht und welche Auswirkungen das haben kann, wurde bereits in Kapitel 2.1 gezeigt und soll hier nicht nochmals wiederholt werden.

### 3.1.4 Abrollen von Schleifen

Das Abrollen von Schleifen[3], welches auch direkt durch den Compiler unterstützt wird, wie später noch zu sehen ist, bringt Performance-Vorteile mit sich, da es dar-

[1] engl.: loop fusion
[2] engl.: loop interchange
[3] engl.:loop unrolling

Listing 3.1: withoutLoopFusion.c

```
#include <time.h>
int arrayA[10000][10000];
int arrayB[10000][10000];
int main () {
  clock_t c1, c2;
  int i, j, k = 0;

  // Initialisierung
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      arrayA[i][j] = 1;

  // Hauptteil
  c1 = clock();
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      arrayB[i][j] = 2 * arrayA[i][j];

  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      k += arrayA[i][j] + arrayB[i][j];
  c2 = clock();
  printf("%d\n",k);
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3f s\n",dt);
}
```

Listing 3.2: loopFusion.c

```
#include <time.h>
int arrayA[10000][10000];
int arrayB[10000][10000];
int main () {
  clock_t c1, c2;
  int i, j, k = 0;

  // Initialisierung
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      arrayA[i][j] = 1;

  // Hauptteil
  c1 = clock();
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++) {
      arrayB[i][j] = 2 * arrayA[i][j];
      k += arrayA[i][j] + arrayB[i][j];
    }
  c2 = clock();
  printf("%d\n",k);
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3f s\n",dt);
}
```

auf abzielt, den Schleifen-Overhead zu verringern. Hierzu soll Listing 2.2 nochmals aufgegriffen werden. Die innere Schleife wurde dabei auf eine Laufweite von 15120 geändert und die, in der Schleife liegende Addition, wurde je nach gewähltem Wert für das Abrollen mehrfach hintereinander ausgeführt, wie das Beispiel in Listing 3.3 zeigt. Der etwas seltsam anmutende Wert 15120 besitzt unter anderem 2,3,4,5,6,7,8,9 und 10 als ganzzahlige Teiler, weshalb dieser als geeignet erscheint, unterschiedliche Werte für loop unrolling zu testen und zu vergleichen. Es ist nicht die kleinstmögliche Zahl, die diese Eigenschaft besitzt, aber 2520 ist für diesen Test etwas zu klein. Dabei ist zu beobachten, daß nicht jeder Wert für das Abrollen auch geeignet ist. Der Übersichtlichkeit halber wurden die Messwerte in Abbildung 3.1 aufgeführt. Um möglichst Nebeneffekte, wie beispielsweise zufällig erhöhte Prozessorlast auszuschließen, wurden die Messungen bis zu zehn mal wiederholt und nur der beste Wert verwendet.

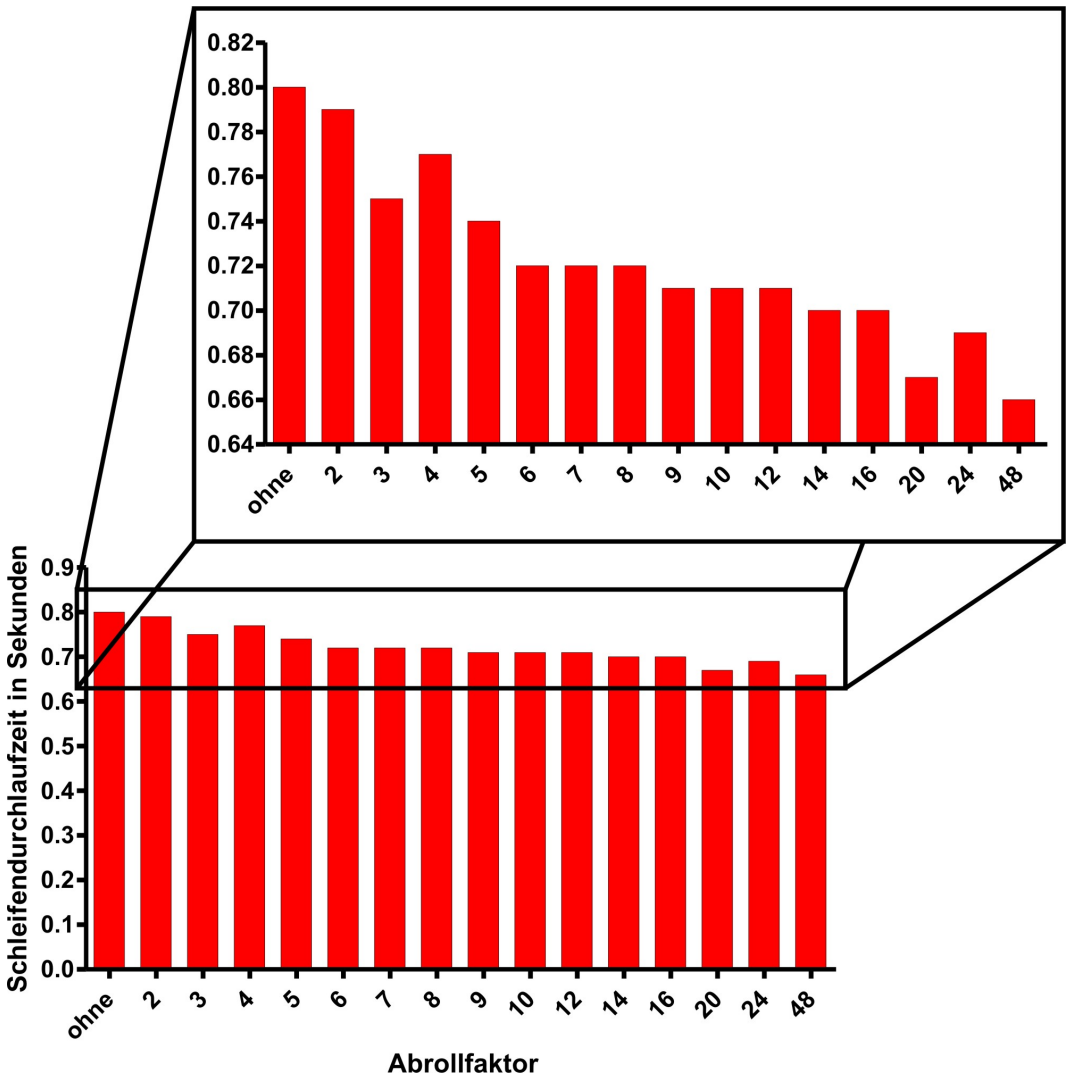


Abbildung 3.1: Laufzeit der Hauptschleife mit verschiedenen Abrollfaktoren

### 3.1.5 Blocken von Daten

Ein etwas komplexeres Verfahren, bei dem vor allem auch Überlegungen zum Aufbau von Caches eingeflossen sind, ist das Blocken von Daten[1]. Hierbei wird versucht, Daten die gemeinsam für eine Berechnung benötigt werden, daran zu hindern, sich gegenseitig aus dem Cache zu verdrängen und dadurch eine hohe Fehlzugriffsrate verursachen.

[1] engl.: blocking

Listing 3.3: loopUnrolling.c

```
1  #include <stdio.h>
2  #include <time.h>
3  int array[10000][15120];
4  int main () {
5    clock_t c1, c2;
6    int i, j, k = 0;
7
8    // Initialisierung
9    for (i=0; i<10000; i++)
10     for (j=0; j<15120; j++)
11       array[i][j] = 1;
12
13   // Hauptteil
14   c1 = clock();
15   for (i=0; i<10000; i++)
16     for (j=0; j<15120; j+=2) {
17       k += array[i][j];
18       k += array[i][j+1];
19     }
20   c2 = clock();
21   double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
22   printf("time : %.3f s\n",dt);
23   printf("sum: %d\n",k);
24 }
```

Als Beispiel dient das Programm aus Listing 3.4, welches nach Abbildung 3.2 zwei Felder A und B addiert und das Ergebnis in Feld A zurückschreibt. Die einzelnen Elemente von Feld A werden dabei mit einer Schrittweite von eins besucht, während die Elemente des Feldes B mit einer Schrittweite von $n$ (im Bild mit der Schrittweite sechs und im Programm mit der Schrittweite 8192) bearbeitet werden. Ein ähnliches Ergebnis würde man erhalten, würde man die beiden Schleifen vertauschen. In diesem Fall würden die Elemente aus Feld A mit einer Schrittweite von $n$ und die Elemente aus Feld B der Reihe nach angesprochen werden.

Durch Blocken soll nun versucht werden, den Cache effektiver zu nutzen. Eine Zweierpotenz als Laufweite wurde übrigens absichtlich gewählt, da diese am Besten zu den gegebenen Cache-Parametern des Xeon-Prozessors paßt, auf dem auch die Messungen durchgeführt wurden. Zum Einen kann damit das vom Prozessor automatisch angestoßene Vorausladen weitestgehend umgangen werden. Zum Anderen ist hierdurch

Listing 3.4: withoutBlocking.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main () {
  clock_t c1, c2;
  int i, j;
  int (*ptrA)[8192] = calloc(8192, sizeof *ptrA);
  int (*ptrB)[8192] = calloc(8192, sizeof *ptrB);

  // Initialisierung
  for (i=0; i<8192; i++)
    for (j=0; j<8192; j++)
      ptrA[i][j] = ptrB[i][j] = 1;

  // Hauptteil
  c1 = clock();
  for (i=0; i<8192; i++)
    for (j=0; j<8192; j++)
      ptrA[i][j] += ptrB[j][i];
  c2 = clock();
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3f s\n",dt);
}
```

eine sehr gute Annäherung an die *Cache Line Size* möglich, ohne zusätzliche Abfragen im Quellcode einzubauen.

In Listing 3.5 wird eine Blockgröße von 8 verwendet, welche durch Messungen ermittelt wurde. Die Ergebnisse der Messung sind in Abbildung 3.4 zu sehen, wobei der Einfachheit halber nur Blockungsfaktoren, die ganzzahlige Teiler von 8192 darstellen, verwendet wurden, da ansonsten zusätzliche Prüfungen auf das Erreichen des Array-Endes implementiert werden müssten, welche zusätzliche Rechenzeit beanspruchen. Deutlich ist in Abbildung 3.4 zu erkennen, daß ein sehr niedriger Blockungsfaktor, wie hier im Beispiel der Blockungsfaktor 1, einen großen Overhead durch die inneren Schleifen verursacht. Im rechten Teil der Skala ist zu erkennen, daß die äußeren Schleifen kaum Overhead produzieren, da sie wenig durchlaufen werden und sich die Laufzeit somit derjenigen von Listing 3.4, die ganz rechts aufgetragen ist, immer weiter annähert.

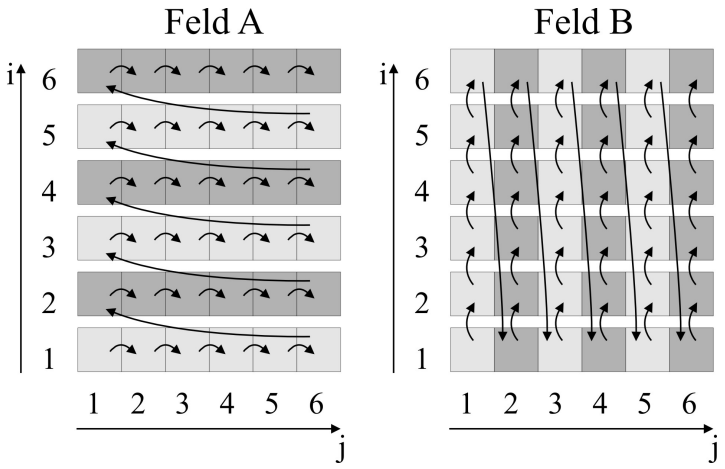


Abbildung 3.2: Addition eines Feldes A[i][j] mit einem Feld B[j][i] ohne Blocking

Anders steht es nun bei einem Blockungsfaktor von acht. Hier ist deutlich zu erkennen, daß es sich einerseits um den besten Faktor im Beispiel handelt und zum Anderen die Laufzeit bei einer weiteren Erhöhung des Blockungsfaktors um den Faktor 2 einen Sprung macht. Dieser Sprung ergibt sich deshalb, da der L2-Cache des Xeon-Prozessors eine Assoziativität von acht hat.

### 3.1.6 Einstreuen von Elementen

Die Feldgröße aus Kapitel 3.1.5 ist nicht ganz zufällig gewählt. Einerseits besitzt der L2-Cache des Xeon-Prozessors eine Assoziativität von acht. Andererseits beträgt der

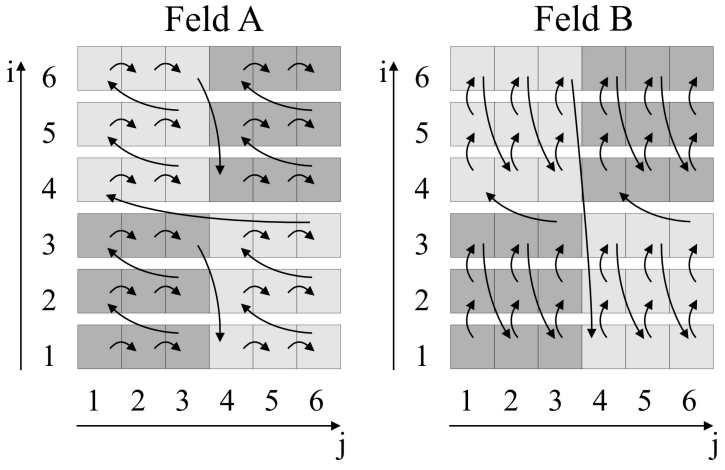


Abbildung 3.3: Addition eines Feldes A[i][j] mit einem Feld B[j][i] mit Blocking

Listing 3.5: blocking.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main () {
  clock_t c1, c2;
  int i, j, bi, bj;
  int (*ptrA)[8192] = calloc(8192, sizeof *ptrA);
  int (*ptrB)[8192] = calloc(8192, sizeof *ptrB);
  int blockSize = 8;

  // Initialisierung
  for (i=0; i<8192; i++)
    for (j=0; j<8192; j++)
      ptrA[i][j] = ptrB[i][j] = 1;

  // Hauptteil
  c1 = clock();
  for (bi=0; bi<8192; bi+=blockSize)
    for (bj=0; bj<8192; bj+=blockSize)
      for (i=bi; i<bi+blockSize; i++)
        for (j=bj; j<bj+blockSize; j++)
          ptrA[i][j] += ptrB[j][i];
  c2 = clock();
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3f s\n",dt);
}
```

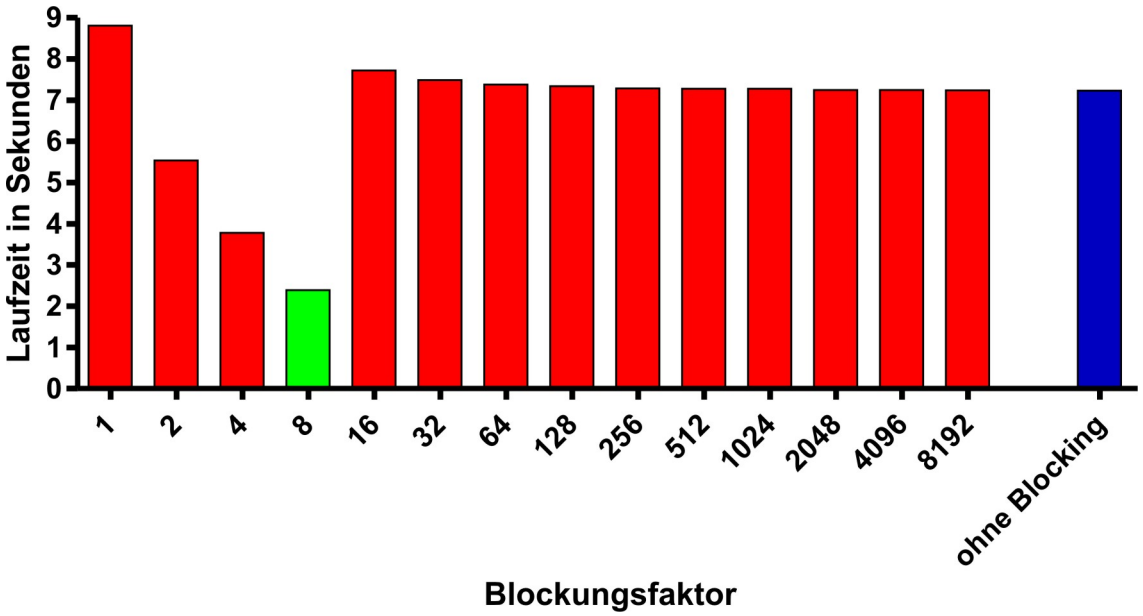


Abbildung 3.4: Messwerte von Listing 3.5 mit verschiedenen Blockgrößen

Abstand von Element B[0][0] zu Element B[1][0] genau 8192 Elementen, zu je 4 Byte[1]. Dies führt dazu, daß nur zwei Sets des L2-Cache während eines Durchlaufs der *bi*-Schleife verwendet werden, nämlich die Sets mit den Nummern 0 und 511. Diese Beiden können diese Menge an Daten nicht fassen und laufen über, was zu einer großen Anzahl an Fehlzugriffen führt.

Die Anzahl der Fehlzugriffe könnte wesentlich verringert werden, würde man den Abstand zwischen den Elementen B[0][0] und B[1][0] vergrößern. Genau dies wird gemacht, indem das Feld durch gezieltes Einstreuen von zusätzlichen Elementen[2] vergrößert wird. Dadurch kann eine einigermaßen homogene Verteilung der Blöcke über den L2-Cache erzwungen werden, was die Anzahl der Fehlzugriffe drastisch verringert. Abbildung 3.5 zeigt, wie solche Elemente in ein Feld eingestreut werden können. Dieses Einstreuen innerhalb eines Datenblocks nennt man auch *intra array padding*. Tritt das Problem der Verdrängung nicht innerhalb eines Feldes auf, sondern verdrängen sich mehrere gleichzeitig benutzte Felder gegenseitig, so kann durch die Verschiebung eines der Felder derselbe Effekt erzielt werden. Dies nennt man *inter array padding*. In diesem Fall werden zusätzliche Elemente zwischen den beiden Feldern A und B reserviert, wie beispielsweise in Abbildung 3.6 das Feld C. Im Folgenden soll nur *intra array padding* genauer betrachtet werden, da im Beispiel die Speicherbereiche dynamisch al-

[1] auf den eingesetzten Xeon-Prozessoren beträgt die Größe eines Integers 4 Byte
[2] engl.: padding

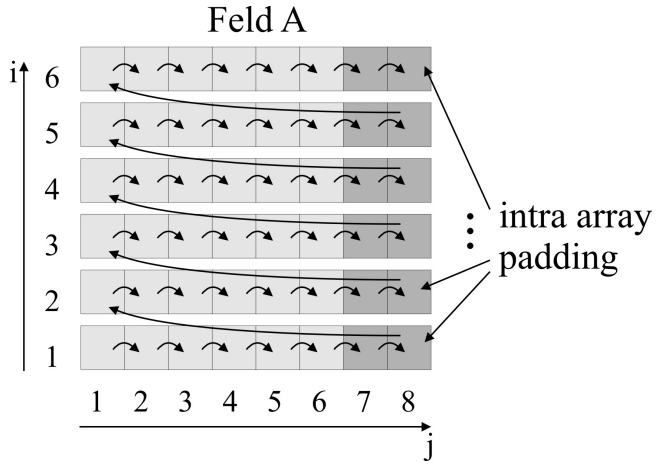


Abbildung 3.5: *intra array padding* am Feld A[i,j]

lozieren und deshalb kein Einfluß auf ihre Lage im Speicher genommen werden kann. Um dennoch die Möglichkeit des *inter array padding* zu nutzen, müßte ein einziger großer Speicherbereich angefordert werden, auf dem zu Beginn Feld A, anschließend Feld C und am Ende Feld B abgebildet wird.

Die möglichen Performance-Vorteile durch Padding sollen nun genauer untersucht werden. Dabei wird Listing 3.5 um Padding erweitert. Der neue Sourcode ist in Listing 3.6 zu sehen. Die gemessen Blockungsfaktoren und die Anzahl der eingestreuten Elemente[1] erstrecken sich über sämtliche Zweierpotenzen von $2^0$ bis $2^{13}$ zuzüglich weiterer Padding-Größen in der Umgebung von Eins sowie in der Umgebung von $2^{13}$. Aus diesen über 200 Messungen entstand unter Entfernung wenig interessanter Werte Abbildung 3.7. Um die Übersichtlichkeit zu verbessern, wurde die Zeitachse umgedreht und verläuft nun in negativer Richtungm, wodurch die optimalen Punkte besser zu

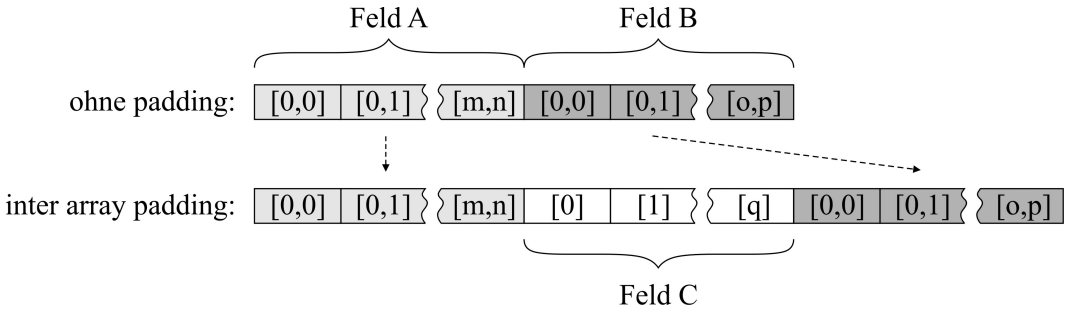


Abbildung 3.6: Beispiel für *inter array padding*

[1]im weiteren auch Padding-Größe genannt

| Blockungsfaktor | Padding-Elemente | Zeit | L1-Misses | L2-Misses |
|---|---|---|---|---|
| 256 | 32 Byte | 1,63s | 95,7 Mio. | 5,4 Mio. |
| 256 | 64 Byte | 1,62s | 92,6 Mio. | 5,4 Mio. |
| 512 | 32 Byte | 1,59s | 103,5 Mio. | 4,9 Mio. |
| 512 | 64 Byte | 1,58s | 103,2 Mio. | 4,9 Mio. |
| 1024 | 32 Byte | 1,55s | 103,8 Mio. | 4,6 Mio. |

Tabelle 3.1: Gemessene Bestwerte

erkennen sind. Außerdem wurden die benachbarten Funktionswerte verbunden, was jedoch nicht auf eine stetige Funktion hindeuten soll, sondern einzig und allein der Anschaulichkeit dient. Die 3 Punkte mit der besten Laufzeit wurden markiert. Diese wurden zum Vergleich mit weiteren interessanten Punkten in Tabelle 3.1 aufgeführt.

Es soll nun festgestellt werden, warum genau diese 3 Punkte so gute Resultate liefern. Hier betrachtet man am Besten in welchen Sets die einzelnen Elemente aus Feld B zum Liegen kommen, wobei der Einfachheit halber das Feld A nicht weiter betrachtet werden soll. Tabelle 3.2 zeigt die Belegungen ausgewählter Sets für unterschiedliche Anzahlen von eingestreuten Elementen.

Unter a) ist zu erkennen, daß die Folgeelemente aus Feld B[i,j] bezüglich der Dimension *i* im L1-Cache in benachbarte Sets abgelegt werden. Dadurch, daß der Xeon bei einem Speicherzugriff immer 128 Byte lädt, dürfte sich der zeitliche Unterschied zwischen einem Padding mit 16 Elementen und einem mit 32 Elementen erklären lassen, da hierdurch viele Verdrängungen im L1-Cache stattfinden.

Elemente aus Feld B:

| B: | [0,0] | [1,0] | [2,0] | [3,0] | .. | [256,0] | [257,0] | .. | [511,0] | .. | [1023,0] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a) Betroffene Sets beim Padding mit 16 Elementen: | | | | | | | | | | | |
| L1: | 0 | 1 | 2 | 3 | .. | 0 | 1 | .. | 31 | .. | 31 |
| L2: | 0 | 513 | 2 | 515 | .. | 256 | 769 | .. | 1023 | .. | 511 |
| b) Betroffene Sets beim Padding mit 32 Elementen: | | | | | | | | | | | |
| L1: | 0 | 2 | 4 | 6 | .. | 0 | 2 | .. | 30 | .. | 30 |
| L2: | 0 | 514 | 4 | 518 | .. | 512 | 2 | .. | 510 | .. | 510 |
| c) Betroffene Sets beim Padding mit 64 Elementen: | | | | | | | | | | | |
| L1: | 0 | 4 | 8 | 12 | .. | 0 | 4 | .. | 28 | .. | 28 |
| L2: | 0 | 516 | 8 | 524 | .. | 0 | 516 | .. | 508 | .. | 508 |

Tabelle 3.2: Padding mit 16, 32 und 64 Elementen

Sowohl unter b) als auch unter c) ist zu sehen, daß diese gegenseitige Verdrängung aufgrund der Sektorgröße nicht auftritt, was sich auch in einer besseren Laufzeit niederschlägt. Der geringe Laufzeitunterschied, der sich noch in den drei herausgepickten Punkten zeigt, lässt sich am leichtesten durch Messung der Fehlzugriffe in den einzelnen Caches zeigen. Tabelle 3.1 zeigt sowohl die Fehlzugriffe in L1-Cache als auch in L2-Cache.

Außerdem ist zu erkennen, daß beim Herantasten an die optimalen Werte für Blocking und Padding einerseits zwar die Fehlzugriffe auf den L1-Cache zunehmen, gleichzeitig verringern sich aber die zeitlich kostspieligeren Fehlzugriffe auf den L2-Cache.

Listing 3.6: padding.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main () {
  clock_t c1, c2;
  int i, j, bi, bj;

  int blockSize = 1024;
  int padding   = 32;

  int (*ptrA)[8192+padding] = calloc(8192, sizeof *ptrA);
  int (*ptrB)[8192+padding] = calloc(8192, sizeof *ptrB);

  // Initialisierung
  for (i=0; i<8192; i++)
    for (j=0; j<8192; j++)
      ptrA[i][j] = ptrB[i][j] = 1;

  // Hauptteil
  c1 = clock();
  for (bi=0; bi<8192; bi+=blockSize)
    for (bj=0; bj<8192; bj+=blockSize)
      for (i=bi; i<bi+blockSize; i++)
        for (j=bj; j<bj+blockSize; j++)
          ptrA[i][j] += ptrB[j][i];
  c2 = clock();
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3f s\n",dt);
}
```

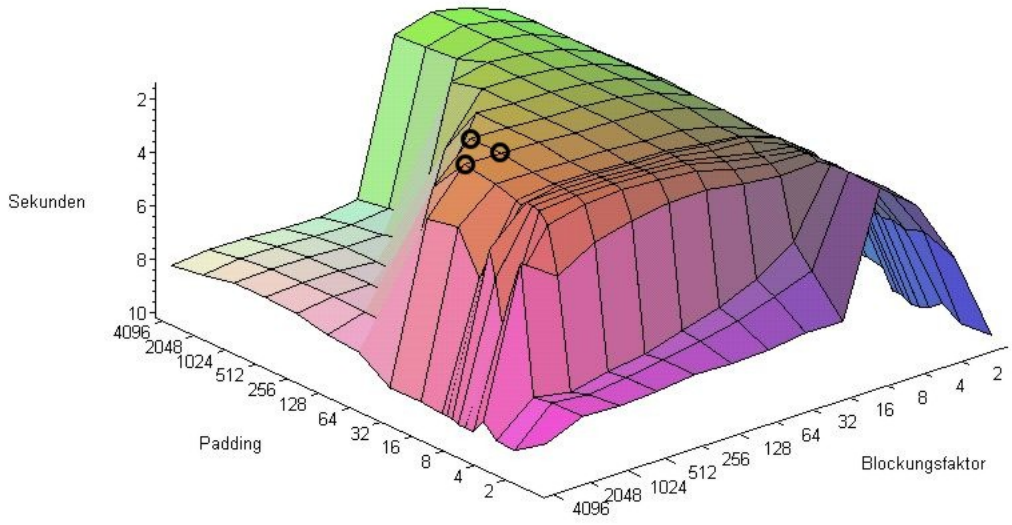


Abbildung 3.7: Messwerte von Listing 3.6 mit unterschiedlichen Padding-Werten

## 3.2 Codeoptimierung auf Hochsprachenebene

### 3.2.1 Compilerschalter

Anhand von zwei gängigen Compilern sollen hier die grundlegenden Möglichkeiten der Compilerschalter aufgezeigt werden. Im Folgenden kommen der Intel C/C++ Compiler in der Version 8.0 sowohl für 32 Bit als auch für Itanium 2 Prozessoren, der C/C++ Compiler aus der „GNU Compiler Collection“ in den Versionen 3.3.2 und 3.4 für 32 Bit, und der C/C++ Compiler aus der „GNU Compiler Collection“ in der Version 3.3.2 für Itanium 2 Prozessoren zum Einsatz.

Alle verwendeten Compiler stellen mehrere Optimierungsstufen zur Verfügung. Von vermeiden jeglicher Optimierung über gute Laufzeitoptimierung bis hin zu einer sehr aggressiven Optimierung, die jedoch nicht mehr zwingend die Laufzeit verbessert. Bei einigen Compilern ergeben sich durch die höchste Optimierungsstufe sogar Probleme [Vet00], die die Korrektheit des Programms beeinflussen. Diese Optionen sollten mit Vorsicht eingesetzt und die Warnungen des Compilers diesbezüglich nicht ignoriert werden.

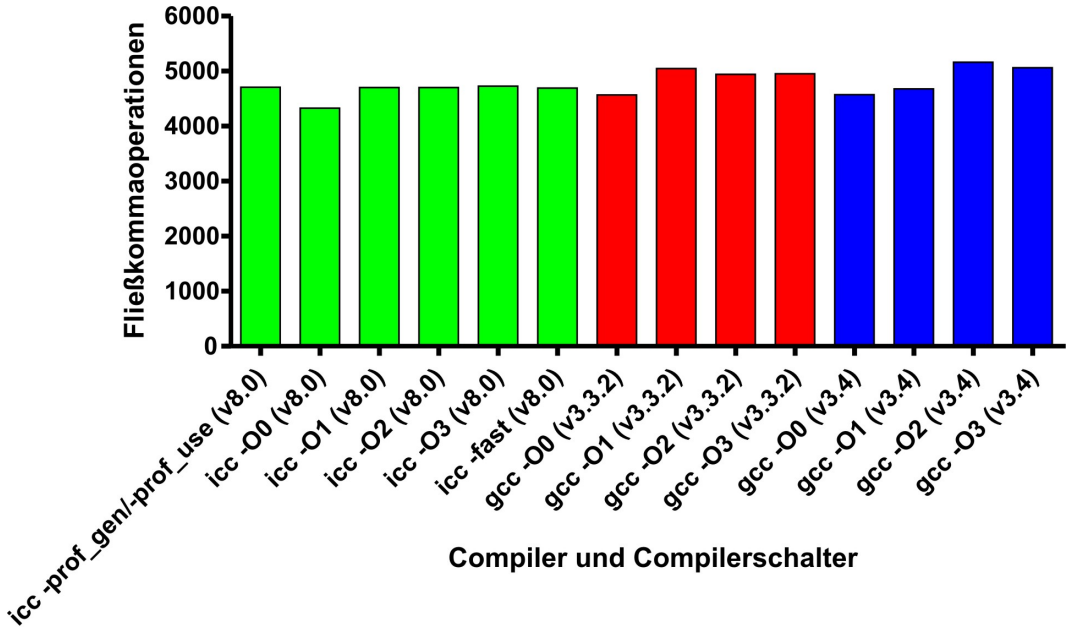


Abbildung 3.8: Gemessene Fließkommaoperationen auf dem Xeon-Rechner, je Sekunde und pro MHz, um einen Vergleich mit dem Itanium-Prozessor zu ermöglichen

Die Verwendung von Compilerschaltern wie *-fast* vereinfachen die Anwendung, da hier mehrere Optimierungsoptionen zusammen verwendet werden.

Sehr interessant ist die Möglichkeit der Feedback-Compilierung. Dabei wird das Quellprogramm vom Compiler insofern verändert, daß es bei der Ausführung Programmlaufeigenschaften sammelt und in eine Datei schreibt[1]. Bei jedem Ausführen des übersetzten Programms wird eine weitere dieser Dateien erzeugt. Durch erneutes Compilieren unter Einbeziehung der gesammelten Programmlaufeigenschaften ist es nun möglich, die gesammelten Werte in die Übersetzung einfließen zu lassen und dadurch eine bessere Laufzeit zu erreichen.

Das bereits mehrmals zum Einsatz gekommene Beispiel soll in etwas veränderter Form, wie in Listing 3.7 zu sehen, dazu dienen, einige Messwerte bei Anwendung der unterschiedlichen Compilerschalter zu erzeugen. Die Messwerte, die auf den verwendeten Xeon-Prozessoren erzeugt wurden, sind in Abbildung 3.8, die auf dem Itanium 2 gemessenen Werte in Abbildung 3.9 dargestellt.

Obwohl es sich beim Vergleich mit nur einem speziell ausgewählten Quellprogramm, das noch dazu keine SSE- und SSE2-Befehle enthält, unmöglich um eine umfassende und repräsentative Messung handeln kann, ist es doch interessant zu sehen, daß hier

[1] engl.: feedback file

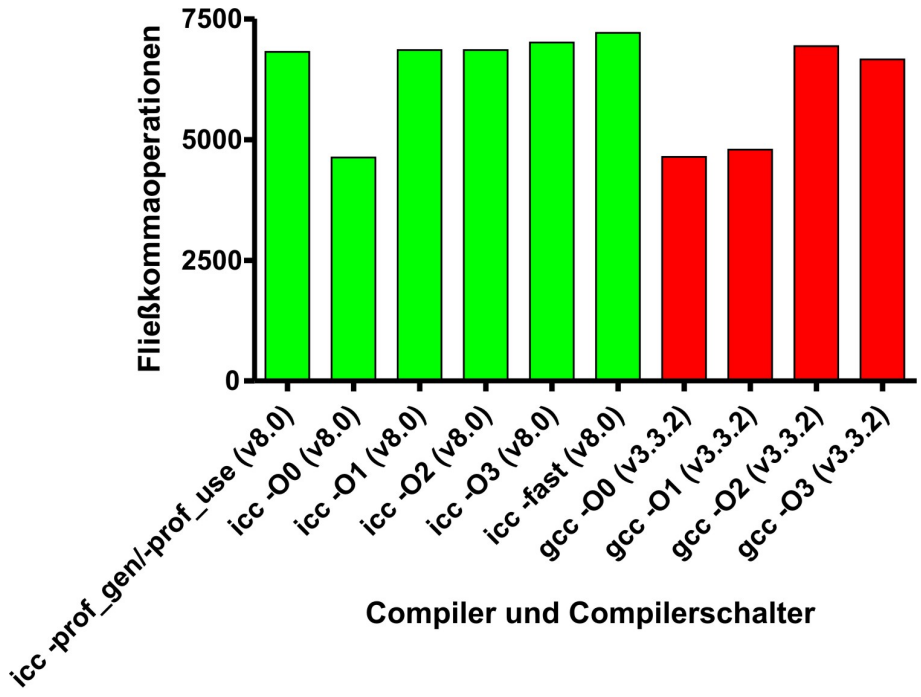


Abbildung 3.9: Gemessene Fließkommaoperationen auf dem Itanium-Rechner, je Sekunde und pro MHz, um einen Vergleich mit dem Xeon-Prozessor zu ermöglichen

zumindest auf dem Xeon der frei erhältliche C/C++ Compiler aus der „GNU Compiler Collection“ dem Intel C/C++ Compiler geringfügig überlegen ist.

### 3.2.2 Compileranweisungen

Compileranweisungen, in der Programmiersprache C so genannte *Pragmas* oder in der Programmiersprache Ada so genannte *significant comments*, werden vom Programmierer eingesetzt, um dem Compiler über gewisse Eigenarten des Codes zu informieren, oder diesen auch Anweisungen darüber zu geben, wie der jeweilige Quellcode zu behandeln ist.

Da sich die Pragmas von Compiler zu Compiler unterscheiden und auch nicht jeder dieselben Möglichkeiten bietet, sollen hier nur ein paar interessante Pragmas des Intel C/C++ Compiler dargestellt werden.

So ist es nicht notwendig, das bereits früher angesprochene Abrollen von Schleifen per Hand durchzuführen. Der Intel C/C++ Compiler [Int03c] bietet hierzu das Prag-

Listing 3.7: compilerSwitches.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 8192
int main () {
  clock_t c1, c2;
  int i = 0, j = 0, countF = 0;
  int mhzXeon = 2400, mhzItanium = 1300, mhz = mhzXeon;

  double (*ptrA)[N] = calloc(N+32, sizeof *ptrA);
  double (*ptrB)[N] = calloc(N+32, sizeof *ptrB);

  // Initialisierung
  for (i=0; i<8192; i++)
    for (j=0; j<8192; j++)
      ptrA[i][j] = ptrB[i][j] = 1.0;

  // Hauptteil
  c1 = clock();
  while (1) {
    while (1) {
      if (i%2==0)
        countF += ptrA[i][j] + 2*ptrB[i++][j];
      else
        countF += 2*ptrA[i][j] + ptrB[i++][j];
      if (i==N) break;
    }
    j++;
    i=0;
    if (j==N) break;
  }
  c2 = clock();
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3f s, flops/(s*mhz) : %.3f,"
    " value : %d\n", dt, (N*N*3)/dt/mhz, countF);
}
```

ma *#pragma unroll(n)* an. Als Beispiel soll hier ein kurzer Ausschnitt aus einem Code dienen, der in Listing 3.8 zu sehen ist.

Das Programm berechnet die Summe über die ganzen Zahlen von 1 bis 100, für die auch die Formel

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2} \tag{3.1}$$

existiert. Die beiden Inline-Assembler-Anweisungen, in diesem Fall ein so genannter Debug-Haltepunkt[1] *int $3* wurden gesetzt, um den Code, den der Compiler aus den Schleifen erzeugt, beim Debuggen leichter zu finden. Die entsprechende Zeilennummern des Quellcodes, die der Compiler im Assemblercode als Kommentar angibt, sind eine weitere Hilfe. Der Compilerschalters *-fsource-asm* veranlaßt den Compiler sogar, die Quellcodezeilen als Kommentare direkt über die zugehörigen Assemblerzeilen zu schreiben.

Listing 3.8: pragmaLoopUnrolling.c

```
1  #include <stdio.h>
2  int main () {
3    int i, j = 0;
4
5  asm ("int\t$3");
6
7  #ifdef PRAGMA
8  #pragma unroll(10)
9  #endif
10   for (i=1; i<=100; i++)
11     j += i;
12
13 asm ("int\t$3");
14
15   printf("sum: %d\n",j);
16 }
```

Wird das Programm ohne das Pragma *#pragma unroll(10)* übersetzt, so sieht man im Listing 3.9, daß der Compiler bei eingeschalteter Optimierung bereits die Schleife abrollt und zwar mit einem Faktor von fünf, während das gesetzte Pragma in Listing 3.10 den Compiler veranlaßt, die Schleife mit einem Faktor von zehn abzurollen.

[1] engl.: debug breakpoint

Zum Verständnis des Assembler-Codes sollen ein paar Zeilen näher betrachtet werden, um zu sehen, was darin passiert. Hierbei beschränken wir uns auf Listing 3.9, da Listing 3.10 äquivalent dazu abläuft. In Zeile 1 wird das Register *ESI*, welches zur Berechnung dient und am Ende das Ergebnis aufnehmen soll, auf Null gesetzt. In Zeile 9 wird die Laufvariable in Register *EAX* mit dem Startwert 1 initialisiert. In Zeile 13 wird der aktuelle Wert der Laufvariable auf das Ergebnisregister *ESI* addiert. In Zeile 14 wird das Register *EDX* mit der Summe der Werten aus Register *ESI*, Register *EAX* sowie der Konstanten 1 gefüllt. Zeile 15 bis 17 verlaufen äquivalent, nur erhöht sich die Konstante jeweils um den Wert 1. In Zeile 18 wird das Register *EAX*, welches die Laufvariable enthält, um den Abroll-Faktor der Schleife erhöht. In diesem Fall 5. In Zeile 19 wird die Abbruchbedingung geprüft und je nach Ergebnis der Prüfung in Zeile 20 entschieden, ob ein weiterer Schleifendurchlauf notwendig ist, oder ob das Ende der Schleife erreicht wurde.

Listing 3.9: withoutPragma.s

```
1   movl      $0, %esi                                  #3.10
2                       # LOE ebx esi edi
3 ..B1.9:               # Preds ..B1.1
4 # Begin ASM
5   int       $3
6 # End ASM
7                       # LOE ebx esi edi
8 ..B1.2:               # Preds ..B1.9
9   movl      $1, %eax                                  #8.8
10
11                      # LOE eax ebx esi edi
12 ..B1.3:              # Preds ..B1.3 ..B1.2
13  addl      %eax, %esi                                #9.5
14  lea       1(%esi,%eax), %edx                        #8.21
15  lea       2(%edx,%eax), %ecx                        #8.21
16  lea       3(%ecx,%eax), %esi                        #8.21
17  lea       4(%esi,%eax), %esi                        #8.21
18  addl      $5, %eax                                  #8.21
19  cmpl      $100, %eax                                #8.3
20  jle       ..B1.3        # Prob 99%                  #8.3
21                      # LOE eax ebx esi edi
22 ..B1.4:              # Preds ..B1.3
23 # Begin ASM
24  int       $3
25 # End ASM
```

Listing 3.10: pragmaUnroll.s

```
1   movl      $0, %esi                                  #3.10
2                       # LOE ebx esi edi
3 ..B1.10:              # Preds ..B1.1
4 # Begin ASM
5   int       $3
6 # End ASM
7                       # LOE ebx esi edi
8 ..B1.3:               # Preds ..B1.10
9   movl      $1, %eax                                  #8.8
10
11                      # LOE eax ebx esi edi
12 ..B1.4:              # Preds ..B1.4 ..B1.3
13  addl      %eax, %esi                                #9.5
14  lea       1(%esi,%eax), %edx                        #8.21
15  lea       2(%edx,%eax), %ecx                        #8.21
16  lea       3(%ecx,%eax), %esi                        #8.21
17  lea       4(%esi,%eax), %edx                        #8.21
18  lea       5(%edx,%eax), %edx                        #8.21
19  lea       6(%edx,%eax), %edx                        #8.21
20  lea       7(%edx,%eax), %edx                        #8.21
21  lea       8(%edx,%eax), %edx                        #8.21
22  lea       9(%edx,%eax), %esi                        #8.21
23  addl      $10, %eax                                 #8.21
24  cmpl      $100, %eax                                #8.3
25  jle       ..B1.4        # Prob 99%                  #8.3
26                      # LOE eax ebx esi edi
27 ..B1.5:              # Preds ..B1.4
28 # Begin ASM
29  int       $3
30 # End ASM
```

Ein weiteres Pragma ist *#pragma optimize*. Dieses dient dazu, direkt im Quellcode auf die Optimierung Einfluß zu nehmen. So läßt sich beispielsweise, wie in Listing 3.11 für die Fuktion *fred* zu sehen, die Optimierung für einzelne Funktionen an-, bzw. abschalten.

Listing 3.11: pragmaOptimize.c

```
1 #pragma optimize("",on)
2 fred() {
3 ...
4 }
```

Das Pragme *#pragma ivdep* teilt dem Compiler mit, ob die Schleife vektorisiert werden kann[1], falls dies für den Compiler nicht ersichtlich ist. Zum Beispiel kann der Compiler keinen Code erzeugen, der parallel ausgeführt wird, falls in der jeweiligen Schleife eine Datenabhängigkeit besteht, die nicht aufgelöst werden kann, wie in Listing 3.12 dargestellt. Da der Compiler den Wert der Variable j nicht kennt und diese einen negativen Wert wie beispielsweise -1 besitzen könnte, kann keine Parallelisierung durchgeführt werden, da ansonsten möglicherweise der Wert der Variable array[i-1] vor der Variable array[i] berechnet werden würde, was zu fehlerhaften Ergebnissen führen würde. Der Compiler geht also sicherheitshalber davon aus, daß eine mögliche Datenabhängigkeit auch tatsächlich gegeben ist. Da der Softwareentwickler jedoch in der Lage ist zu entscheiden, ob die Abhängigkeit auch tatsächlich vorhanden ist, kann er dies dem Compiler über das Pragma *#pragma ivdep* mitteilen.

Listing 3.12: pragmaIvdep.c

```
1 #pragma ivdep
2 for (i=0; i<z; i++) {
3   array[i] = array[i+j];
4 }
```

Das Pragma *#pragma prefetch* dient dazu, dem Compiler anzuweisen, Code für das Vorausladen gewisser Speicherinhalte zu erzeugen, so daß diese bei Bedarf zur Verfügung stehen. Vor die Schleife geschrieben, wie in Listing 3.13 zu sehen, fügt der Compiler Befehle zum Vorausladen von *array[i+delta]* in die Schleife ein, wobei der Compiler einen geeigneten Wert für *delta* einsetzt. Dieses Pragma steht aber erst bei der Optimierungsstufe *-O3* zur Verfügung, ansonsten wird es ignoriert.

[1] beim Vektorisieren kommen sogenannte SIMD Instruktionen zum Einsatz

Listing 3.13: pragmaPrefetch.c

```
1 #pragma prefetch arrayA
2 for(i=0; i<max; i++) {
3   arrayA[i] = 2 * arrayB[i];
4 }
```

Es gibt noch eine ganze Reihe weiterer Pragmas, die hier aber weder alle aufgeführt, noch umfassend besprochen werden können. Deshalb sei an dieser Stelle auf [Int03c] verwiesen.

### 3.2.3 Besondere Code-Erweiterungen

Viele der neueren Intel Prozessoren, darunter auch der Pentium 4, bieten Instruktionen an, um multimediale Anwendungen zu optimieren. Bei diesen Instruktionen handelt es sich um Erweiterungen von bereits in früheren Prozessorgenerationen implementierten Anweisungen. Diese ermöglichen es, mehrere Daten in einem einzigen Befehl parallel abzuarbeiten[1]. Beispielsweise handelt es sich bei *ADDPS Operand1, Operand2* um einen Additionsbefehl, der paarweise die vier im ersten Operanden stehenden Fließkommazahlen einfacher Genauigkeit mit den vier im zweiten Operanden stehenden Fließkommazahlen, ebenfalls einfacher Genauigkeit, addiert und das Ergebnis im ersten Operanden ablegt. Dies benötigt ohne SIMD vier Fließkommabefehle.

Der direkteste Weg, diese Instruktionen zu nutzen, ist der Einsatz von Assembler-Code im Quellcode[2]. Da einige Compiler dies nicht unterstützen, kann man bei diesen häufig auf besondere Befehle[3] ausweichen. Ein paar dieser Befehle, die beispielsweise der Intel C/C++ Compiler anbietet, sollen näher betrachtet werden.

Dabei sollen die zwei Programme Listing 3.14 und Listing 3.15, die zwei Felder miteinander addieren und die Summe über die einzelnen Ergebnisse bilden, verglichen werden. In eines davon wird das Intrinsic *_mm_add_pd* eingefügt, das der Compiler durch den Befehl *ADDPD* ersetzt. *ADDPD* addiert ähnlich wie *ADDPS* zwei Operanden elementweise, nur beinhaltet hier jeder Operand zwei Fließkommawerte doppelter Genauigkeit. Im Gegensatz dazu verwendet der Compiler in Listing 3.14 trotz Optimierung für Pentium 4 Prozessoren nur *ADDSD*-Befehle. Diese addieren zwar ebenfalls

[1] engl.: single instruction multiple data, oder kurz SIMD
[2] engl.: inline assembler instructions
[3] engl.: intrinsics

Fließkommazahlen doppelter Genauigkeit, jedoch nur eine und nicht zwei, wie es mit *ADDPD* möglich ist. Die Messung mit dem Tool *Perfsum*[1], das es auf einfache Weise ermöglicht, vom Prozessor zur Verfügung gestellte Ereignisse zu messen, sowie die eingebaute Zeitmessung, sind in Tabelle 3.3 aufgelistet.

Listing 3.14: noIntrinsic.c

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define M 10000
#define N 10000
int main () {
  clock_t c1, c2;
  int i, j;
  double count = 0.0;

  double (*ptrA)[M] = _mm_malloc(N*sizeof(*ptrA),16);
  double (*ptrB)[M] = _mm_malloc(N*sizeof(*ptrB),16);

  for (i=0;i<N;i++)
    for (j=0;j<M;j++) {
      ptrA[i][j]=1.0;
      ptrB[i][j]=2.0;
    }

  c1 = clock();
  for (i=0;i<N;i++) {
    for (j=0;j<M;j+=2) {
      ptrA[i][j]   += ptrB[i][j];
      ptrA[i][j+1] += ptrB[i][j+1];
      count += ptrA[i][j]+ptrA[i][j+1];
    }
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3fs , value : %.0f\n",dt,count);
}
```

Zum Listing 3.15 ist noch anzumerken, daß die SSE2-Datentypen *__m128d* auf eine 16-Byte Speichergrenze ausgerichtet werden müssen, deshalb wurde hier der Befehl

[1] siehe hierzu Kapitel 4.3.1

Listing 3.15: intrinsicAddpd.c

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <emmintrin.h>
#define M 10000
#define N 10000
int main () {
  clock_t c1, c2;
  int i, j;
  double count = 0.0;
  __m128d *s1, *s2;

  double (*ptrA)[M+2] = _mm_malloc(N*sizeof(*ptrA),16);
  double (*ptrB)[M+2] = _mm_malloc(N*sizeof(*ptrB),16);

  for (i=0;i<N;i++)
    for (j=0;j<M;j++) {
      ptrA[i][j]=1.0;
      ptrB[i][j]=2.0;
    }

  c1 = clock();
  for (i=0;i<N;i++) {
    s1 = (__m128d *)ptrA[i];
    s2 = (__m128d *)ptrB[i];
    for (j=0;j<M;j+=2) {
      *s1 = _mm_add_pd(*s1, *s2);
      s1++;
      s2++;
      count += ptrA[i][j]+ptrA[i][j+1];
    }
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3fs , value : %.0f\n",dt,count);
}
```

| Programm | L1-Misses | L2-Misses | Schleifenlaufzeit |
|---|---|---|---|
| Listing 3.14 | 151 Mio. | 12,5 Mio. | 1,28s |
| Listing 3.15 | 126 Mio. | 12,5 Mio. | 1,24s |

Tabelle 3.3: Gemessene Werte mit und ohne Intrinsic

*_mm_malloc* verwendet, dem als zweiten Parameter die Ausrichtung im Speicher übergeben wird.

Neben *ADDPD* existieren eine Reihe weiterer MMX-, SSE- und SSE2-Befehle, die mithilfe von Intrinsics eingesetzt werden können. Alle hier aufzuführen und zu beschreiben würde zu weit führen, deshalb sei hiermit auf [Int03c] oder auch [Roh01] verwiesen.

## 3.3 Codeoptimierung auf Assemblerebene

In diesem Kapitel wird die Möglichkeit der Optimierung auf Assemblerebene betrachtet. Obwohl bereits schon unter den Pragmas und Intrinsics möglich und dort auch aufgeführt, sollen einzelne Optimierungsmethoden hier nochmals aufgegriffen werden, da es sich entweder im Grunde um Assembleranweisungen handelt, oder aber einzelne Optimierungen auf Assemblerebene nochmals weiter verbessert werden können.

### 3.3.1 Compilerausgabe weiter optimieren

Zunächst betrachten wir nochmals Listing 3.15. Ein Teil der Ausgabe des Compilers für die innere Schleife ist in Listing 3.16 dargestellt. Leicht sieht man, daß hier mehrere Dinge nicht optimal sind. Zum Einen fällt auf, daß der Compiler in Zeile 15 die beiden Fließkommawerte doppelter Genauigkeit in Form eines 128 Bit -Wertes in das Register *XMM1* lädt, in Zeile 16 die Addition mithilfe des eingesetzten Intrinsics *ADDPD* addiert und anschließend in Zeile 17 das Ergebnis zurückschreibt. Da der Compiler nicht erkennt, daß im Folgenden der errechnete Wert erneut aufgegriffen wird um die Summe über alle Ergebnisse zu bilden, wird in Zeile 18 der zurückgeschriebene Wert erneut geladen und auf das Register *XMM0* addiert. XMM0 enthält also offensichtlich unsere Variable *count*. In Zeile 19 wird der zweite Fließkommawert erneut geladen und ebenfalls auf Register *XMM0* addiert. Anschließend werden die Zeiger erhöht und auf das Erreichen des Schleifenendes getestet.

Abbildung 3.10 zeigt nochmal graphisch, welche Speicherzugriffe während der Berechnung im Einzelnen erfolgen, wobei die Speicherzugriffe der Zeilen 15 bis 19 mit den römischen Nummern I bis V gekennzeichnet sind.

Dies soll im Folgenden verbessert werden. Zum Einen ist interessant, daß der *MOVSD*-Befehl in Zeile 2 eine 64 Bit-Speichervariable in die unteren 64 Bit des Registers *XMM0*

lädt. Dabei setzt er die oberen 64 Bit auf null, was gleichzeitig einem Fließkommawert doppelter Genauigkeit mit dem Wert 0.0 entspricht. Dies wird ausgenutzt, indem die beiden Additionen aus Zeile 18 und 19 durch eine einzige Addition mithilfe von *ADDPD*, wie in Listing 3.17 in Zeile 18 zu sehen, ersetzt wird. Dies führt aber zunächst zu einem falschen Ergebnis, da in Listing 3.16 in Zeile 33 nur der untere der beiden Fließkommawerte aus dem Register *XMM0* zurückgeschrieben wird. Dies kann korrigiert werden, indem man in Listing 3.17 die Zeilen 33 bis 35 einfügen.

Zum Anderen fällt auf, daß sich die beiden Register *EAX* und *ESI* in Listing 3.16 über denselben Bereich bewegen, wie in den Zeilen 8, 11, 20 und 22 zu sehen ist. Dies wird in Listing 3.17 geändert, indem Register *EAX* ausgeschlossen und alles auf die Verwendung von Register *ESI* abgeändet wird. Änderungen in Listing 3.17 gegenüber Listing 3.16 sind durch Großschreibung und eingefügte Kommentare kenntlich gemacht.

Die Ergebnisse der Messung mittels *Perfsum* sind in Tabelle 3.4 aufgeführt. Obwohl die Anzahl der L1-Misses um 27% zurückgegangen ist, hat sich die Laufzeit verschlechtert. An den SIMD-Ausnahmebehandlungen[1] kann es nicht liegen, denn ersetzt man in Listing 3.15 die beiden zu addierenden Werte 1.0 und 2.0 durch 1.1 und 2.1, sinkt die Anzahl der Ausnahmebehandlungen beinahe auf Null. Der Unterschied bei den Konflikten beim Zugriff auf Speicherstellen in einem Abstand von 64 KByte[2] ist vermutlich die Ursache, da hier auf nicht sehr effektive Nutzung des Caches geschlossen werden kann. Deshalb macht sich auch die fehlende Optimierung im, vom Compiler erzeugten Code, aufgrund der hohen Latenzzeiten, die wärend eines Fehlzugriffs auf den Cache auftreten, nicht bemerkbar. Listing 3.15 soll dahingehend modifiziert werden, daß die Schleifen nur über einen kleinen Speicherbereich laufen, wie Listing 3.18 zeigt. Nachdem auch die Optimierung auf Assemblerebene erneut durchgeführt wurde, zeigt sich ein ganz erstaunlicher Unterschied, der in Tabelle 3.5 aufgeführt ist.

Da der Compiler das eingefügte Intrinsic nicht ignorieren kann, wurde Listing 3.14 nochmals aufgegriffen, entsprechend der Speichernutzung und der inneren Schleife angepaßt und wie Listing 3.18, jedoch ohne Optimierung per Hand, übersetzt. Offenbar fällt es dem Compiler leichter, effizienten Code zu erzeugen, wenn kein Intrinsic eingefügt wird, an das es sich zu halten hat. Dennoch bringt die Verwendung des In-

[1] engl.: SSE Input Assist. Diese treten beispielsweise bei einem Unter- bzw. Überlauf wärend der Berechnung mit Fließkommawerten auf, aber auch ein nicht normalisierter Fließkommawert kann die Ursache dafür sein.

[2] engl.: 64k Aliasing Conflict. Diese Konflikte treten auf, wenn ein Speicherzugriff auf eine virtuelle Adresse erfolgt, welche eine *Cache Line* referenziert, die einen Abstand von modulo 64 KByte zu einer anderen *Cache Line* besitzt, die sich bereits im L1-Cache befindet. Der L1-Cache kann nur eine solche *Cache Line* aufnehmen und muß deshalb die andere verdrängen.

Listing 3.16: intrinsicAddpd.s

```
1  ..B1.20:                # Preds ..B1.31
2    movsd     56(%esp), %xmm0                               #23.8
3    movl      %esi, 52(%esp)                                #23.8
4    xorl      %edx, %edx                                    #23.8
5                           # LOE edx ebp xmm0
6  ..B1.21:                # Preds ..B1.23 ..B1.20
7    movl      52(%esp), %ecx                                #25.21
8    movl      %ebp, %eax                                    #24.21
9    lea       (%ecx,%edx), %edi                             #25.21
10   lea       80000(%ebp), %ecx                             #26.10
11   movl      %ebp, %esi                                    #26.10
12   .align    4,0x90
13                          # LOE eax edx ecx ebp esi edi xmm0
14 ..B1.22:                # Preds ..B1.22 ..B1.21
15   movapd    (%eax), %xmm1                                 #27.25
16   addpd     (%edi), %xmm1                                 #27.13
17   movapd    %xmm1, (%eax)                                 #27.7
18   addsd     (%esi), %xmm0                                 #30.7
19   addsd     8(%esi), %xmm0                                #30.27
20   addl      $16, %eax                                     #28.7
21   addl      $16, %edi                                     #29.7
22   addl      $16, %esi                                     #26.18
23   cmpl      %esi, %ecx                                    #26.5
24   ja        ..B1.22       # Prob 99%                      #26.5
25                          # LOE eax edx ecx ebp esi edi xmm0
26 ..B1.23:                # Preds ..B1.22
27   addl      $80016, %edx                                  #23.16
28   addl      $80016, %ebp                                  #23.16
29   cmpl      $800160000, %edx                              #23.3
30   jl        ..B1.21       # Prob 99%                      #23.3
31                          # LOE edx ebp xmm0
32 ..B1.24:                # Preds ..B1.23
33   movsd     %xmm0, 56(%esp)                               #
34   call      clock                                         #33.8
```

| Ereignis | Listing 3.15 | Listing 3.15 optimiert |
|---|---|---|
| **L1-Misses** | 126 Mio. | 92 Mio. |
| **L2-Misses** | 12,5 Mio. | 12,5 Mio. |
| **SSE Input Assist** | 0,4 Mio. | 1,5 Mio. |
| **64k Aliasing** | 38 Mio. | 42 Mio. |
| **Schleifenlaufzeit** | 1,24s | 1,28s |

Tabelle 3.4: Gemessene Werte vor und nach der Optimierung auf Assemblerebene

Listing 3.17: intrinsicAddpdOptimized.s

```
1  ..B1.20:                # Preds ..B1.31
2    movsd     56(%esp), %xmm0                                 #23.8
3    movl      %esi, 52(%esp)                                  #23.8
4    xorl      %edx, %edx                                      #23.8
5                           # LOE edx ebp xmm0
6  ..B1.21:                # Preds ..B1.23 ..B1.20
7    movl      52(%esp), %ecx                                  #25.21
8                                          # ZEILE GELOESCHT
9    lea       (%ecx,%edx), %edi                               #25.21
10   lea       80000(%ebp), %ecx                               #26.10
11   movl      %ebp, %esi                                      #26.10
12   .align    4,0x90
13                          # LOE eax edx ecx ebp esi edi xmm0
14 ..B1.22:                # Preds ..B1.22 ..B1.21
15   MOVAPD    (%ESI), %XMM1             # ZEILE VERAENDERT
16   addpd     (%edi), %xmm1                                   #27.13
17   MOVAPD    %XMM1, (%ESI)             # ZEILE VERAENDERT
18   ADDPD     %XMM1, %XMM0              # ZEILE VERAENDERT
19                                       # ZEILE GELOESCHT
20                                       # ZEILE GELOESCHT
21   addl      $16, %edi                                       #29.7
22   addl      $16, %esi                                       #26.18
23   cmpl      %esi, %ecx                                      #26.5
24   ja        ..B1.22       # Prob 99%                        #26.5
25                          # LOE eax edx ecx ebp esi edi xmm0
26 ..B1.23:                # Preds ..B1.22
27   addl      $80016, %edx                                    #23.16
28   addl      $80016, %ebp                                    #23.16
29   cmpl      $800160000, %edx                                #23.3
30   jl        ..B1.21       # Prob 99%                        #23.3
31                          # LOE edx ebp xmm0
32 ..B1.24:                # Preds ..B1.23
33   MOVAPD    %XMM0, %XMM1              # ZEILE HINZUGEFUEGT
34   PSRLDQ    $8, %XMM1                 # ZEILE HINZUGEFUEGT
35   ADDSD     %XMM1, %XMM0              # ZEILE HINZUGEFUEGT
36   movsd     %xmm0, 56(%esp)                                 #
37   call      clock                                           #33.8
```

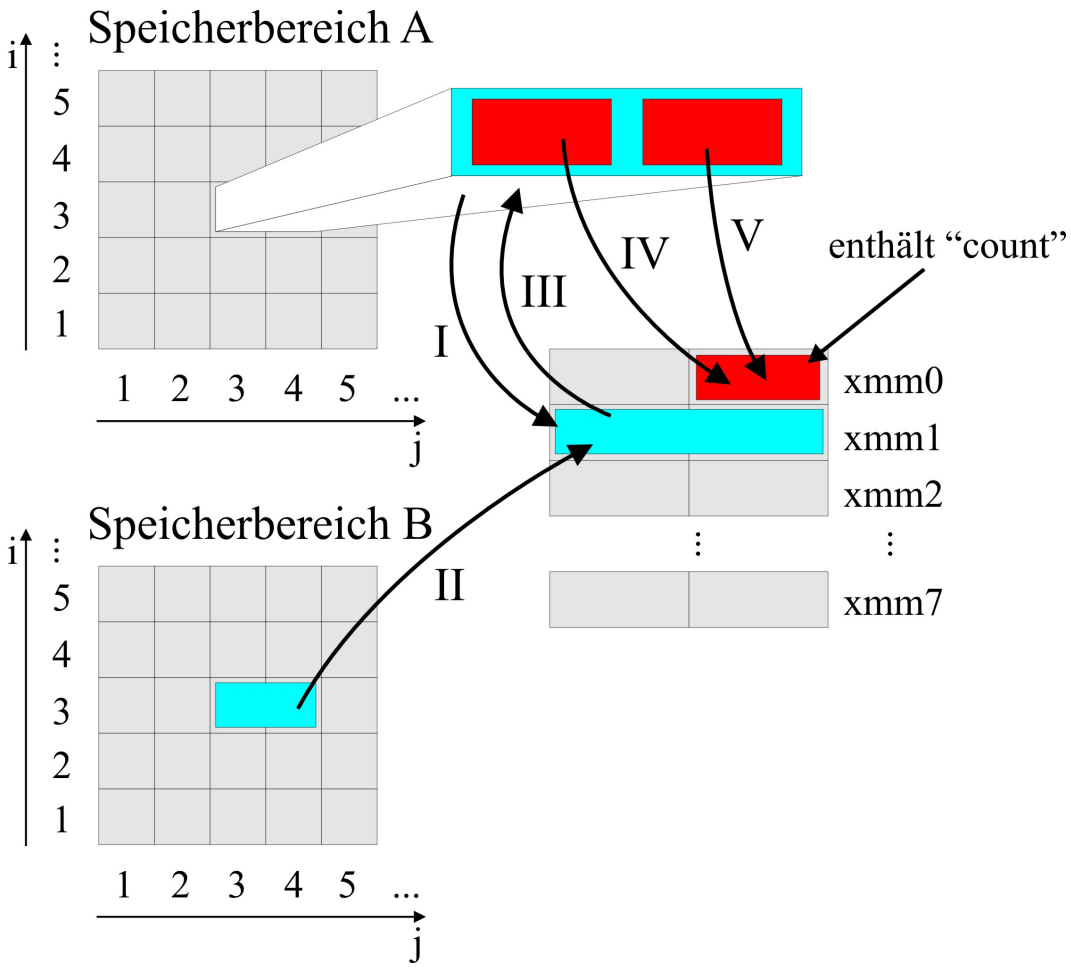


Abbildung 3.10: Überblick über die Berechnungen innerhalb der Schleife von Listing 3.15

trinsics und anschließend eine Optimierung per Hand immer noch einen deutlichen Zeitgewinn von fast 29%. Die Messungen wurden mithilfe der *Performance-Monitoring Counter* des Xeon-Prozessors durchgeführt, dabei kam unter anderem das Tool *Perfsum* zum Einsatz.

Eine erste Gegenüberstellung der Ergebnisse ist in Tabelle 3.5 zu sehen. Hier ist außer einem kleinen Unterschied in *SSE Input Asssist* und dem nach wie vor etwas großen Unterschied in *64k Aliasing* nichts zu erkennen, bis auf die Tatsache, daß die optimierte Version um Einiges schneller geworden ist. Deshalb sind in Tabelle 3.6 nochmals umfassendere Ausgaben der Messdaten aufgelistet.

Hier fallen im Wesentlichen drei Messwerte ins Auge:

| Ereignis | nicht optimiert | optimiert | Listing 3.14 |
|---|---|---|---|
| **L1-Misses** | 99 Mio. | 59 Mio. | 57 Mio. |
| **L2-Misses** | 4 Tsnd. | 4 Tsnd. | 4 Tsnd. |
| **SSE Input Assist** | 19 Tsnd. | 16 Tsnd. | 100 |
| **64k Aliasing** | 41 Tsnd. | 16 Mio. | 3,8 Mio. |
| **Schleifenlaufzeit** | 0,99s | 0,14s | 0,21s |

Tabelle 3.5: Listing 3.18 ohne und mit Optimierung auf Assemblerebene. Außerdem zum Vergleich Listing 3.14, das in Bezug auf die Speichernutzung an Listing 3.18 angepaßt wurde.

- 64k Aliasing:
  Diese könnten sicherlich durch den Einsatz von Padding stark reduziert werden. Das bisher erreichte Ergebnis soll uns aber hier genügen, da es zeigt, daß sich durch Optimierungen im Assemblercode durchaus etwas mehr Performance erreichen läßt.

- Scalar DP Retired:
  Der Unterschied in diesem Meßwert ist sehr markant. Daß dieser Meßwert beinahe gänzlich verschwindet, läßt sich dadurch erklären, daß die vom Compiler eingesetzten *ADDSD*-Befehle entfernt und durch *ADDPD*-Befehle ersetzt wurden.

- Packed DP Retired:
  Dieser Wert ist in der optimierten Version entsprechend angestiegen, da hierunter die verwendeten *ADDPD*-Befehle fallen. Sehr schön sieht man, daß durch den Einsatz dieses Befehls 50% der Fließkommaoperatoren eingespart werden konnten.

Dieses Beispiel zeigt, daß bei „Optimierungen“ am, vom Compiler erzeugten Assemblercode, vorsichtig vorzugehen ist, da sich das Laufzeitverhalten, entgegen aller Erwartungen verschlechtern kann. Ein zeitintensives Studium des Assemblerlistings und umfangreiche Messungen am veränderten Code sind deshalb unvermeidbar.

### 3.3.2 Vorausladen auf dem Xeon-Prozessor

Vergleicht man die Verarbeitungsgeschwindigkeit moderner Prozessoren mit der Zeit, die ein Speicherzugriff benötigt, so stellen sich enorme Unterschiede heraus. Damit der Prozessor nicht länger als notwendig auf die benötigten Daten aus dem Speicher

Listing 3.18: intrinsicAddpdMemoryReduced.c

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <emmintrin.h>
#define M 100
#define N 100
int main () {
  clock_t c1, c2;
  int i, j, k;
  double count = 0.0;
  __m128d *s1, *s2, ts1, ts2, td;

  double (*ptrA)[M] = _mm_malloc(N*sizeof(*ptrA),16);
  double (*ptrB)[M] = _mm_malloc(N*sizeof(*ptrB),16);

  for (i=0;i<N;i++)
    for (j=0;j<M;j++) {
      ptrA[i][j]=1.0;
      ptrB[i][j]=2.0;
    }

  c1 = clock();
  for (k=0;k<10000;k++)
    for (i=0;i<N;i++) {
      s1 = (__m128d *)ptrA[i];
      s2 = (__m128d *)ptrB[i];
      for (j=0;j<M;j+=2) {
        *s1 = _mm_add_pd(*s1, *s2);
        s1++;
        s2++;
        count += ptrA[i][j]+ptrA[i][j+1];
      }
    }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time : %.3fs , value : %.0f\n",dt,count);
}
```

| **Messung** | **Listing 3.18 optimiert** | **Listing 3.14** |
|---|---|---|
| Clockticks | 369,805,040 cycles | 510,729,908 cycles |
| Instr.Retired | 409,628,726 events | 556,620,727 events |
| Active Bus Cycles | 19,480 events | 21,836 events |
| Bus Accesses | 28,756 events | 30,340 events |
| 1st. L.Cache Misses | 59,298,120 events | 56,736,818 events |
| 2nd L.Cache Refs | 38,741,338 events | 37,903,205 events |
| 2nd L.Cache Misses | 3,896 events | 4,416 events |
| **64k Aliasing** | **15,489,737 events** | **3,762,164 events** |
| x87 SP/DP Retired | 42,128 events | 42,129 events |
| x87 Input Assist | 0 events | 0 events |
| SSE Input Assist | 9,718 events | 81 events |
| Scalar SP Retired | 0 events | 0 events |
| Packed SP Retired | 15 events | 15 events |
| **Scalar DP Retired** | **1 events** | **200,000,000 events** |
| **Packed DP Retired** | **100,000,004 events** | **4 events** |
| DTLB Misses | 2,543 events | 2,500 events |
| ITLB Misses | 721 events | 690 events |
| Branches Retired | 51,273,644 events | 51,273,644 events |
| Mispred. Branches | 1,031,103 events | 1,031,199 events |
| Split Loads | 10 events | 11 events |
| Split Stores | 17 events | 17 events |
| Blk.St.Forw. Retired | 13,626 events | 10,316 events |
| Bus Acc.(w/o Pref.) | 14,871 events | 14,759 events |

Tabelle 3.6: Vergleich von Listing 3.18 mit Listing 3.14 unter Verwendung von *Performance-Monitoring Counter* des Xeon-Prozessors

warten muß, bietet dieser die Möglichkeit, das Laden der Daten bereits im Voraus anzustoßen[1], so daß diese bei Bedarf bereits verfügbar sind oder zumindest nicht mehr lange auf sich warten lassen.

Der Xeon-Prozessor bietet hier mehrere Befehle an, die der Programmierer verwenden kann, um Ladebefehle anzustoßen. Hierzu noch ein paar Anmerkungen:

- Zum Einen existieren in älteren Pentium-Prozessoren mehrere Befehle zum Vorausladen von Daten, die sich darin unterscheiden, daß durch die jeweilige Auswahl des Befehls das Ziel der zu ladenden Daten angegeben werden konnte. War beim Pentium III der Befehl *PREFETCHT0* für das Vorausladen von 32 Byte Daten aus dem Speicher sowohl in den L1-Cache als auch in den L2-Cache und die beiden Befehle *PREFETCHT1* und *PREFETCHT2* nur für das Laden von 32 Byte Daten in den L2-Cache zuständig, so laden beim Pentium 4 alle drei Befehle [Int03b] nun 128 Byte, jedoch nur noch bis in den L2-Cache. Intel hat hier offenbar versucht, das Problem möglichst gering zu halten, daß beispielsweise noch benötigte Daten durch zu aggressives Vorausladen bereits zu früh wieder aus dem Cache verdrängt werden[2] [JDR].
- Neben diesen gibt es noch den Befehl *PREFETCHNTA*, welche ursprünglich beim Pentium III die Daten in den L1-Cache, aber nicht in den L2-Cache geladen hat. Dieser lädt nun beim Pentium 4 die Daten ebenfalls nur noch bis in den L2-Cache, jedoch mit dem Unterschied, daß in jedem Set ausschließlich die erste *Cache Line* benutzt wird. Hierdurch wird weitgehend vermieden, daß andere, noch vom Programm benötigte Daten verdrängt werden. Intel empfiehlt die Anwendung von *PREFETCHNTA* dann, wenn der Datenblock weniger als 32 KByte umfasst oder aber die Daten nur für einen einzigen Zugriff benötigt werden [Int03b].

Listing 3.19 soll zeigen, wie die Befehle zum Vorausladen von Daten eingesetzt werden können. Die Messergebnisse sind in Abbildung 3.11 zu finden. Die erste Viererkolonne wurde ohne weitere Compilerschalter übersetzt. Bei der zweiten Viererkolonne kam der Compilerschalter *-march=pentium4* zum Einsatz, damit der Compiler die Zielarchitektur kennt und selbständig effizienteren Code erzeugen kann. Durch Vergleichen der erzeugten Compilerausgaben ist festzustellen, daß ohne Hinweis der verwendeten Zielarchitektur beispielsweise ausschließlich Fließkommaoperationen wie *FADD* zum Einsatz kommen, während im anderen Fall SSE2-Befehle verwendet werden. Der Compiler erzeugt ohne diesen Schalter Programme mit so schlechter Laufzeit,

[1] engl.: prefetching
[2] engl.: cache pollution

daß sich das Vorausladen der Daten nicht bemerkbar macht. Mit Angabe der Zielarchitektur kann der Compiler das Programm entsprechend optimieren, so daß der per Intrinsic eingebettete Vorausladebefehl Wirkung zeigt. Aus der Dokumentation zum Xeon-Prozessor [Int03b] ist zu entnehmen, daß der Hardware-Prefetcher[1] nicht über die Begrenzung einer 4 KByte-Seite hinaus lesen kann. Außerdem kann ein Vorausladebefehl keine Ausnahmen[2], beispielsweise in Form eines TLB-Misses auslösen. Deshalb wird der Vorausladebefehl ignoriert, falls sich die zu ladenden Daten in einer Seite befinden, die im Moment nicht im Speicher liegt, oder auf die kein Verweis im *Translation Lookaside Buffer* existiert. Dies erklärt auch die benötigte Laufzeit von Listing 3.19 bei Angabe des Schalters *-DPREFETCH*. Hierdurch wird der zwischen der Zeile *#define PREFETCH* und dem zugehörigen *#endif* stehende Code verwendet. Um dies zu erzwingen, muß eine neue Seite vom Programm durch einen direkten Ladebefehl angestoßen werden. Berücksichtigt man dies und ersetzt *PREFETCHT2* in Listing 3.19 durch einen direkten Ladebefehl, kann die Laufzeit weiter gesenkt werden, da nun auch das Laden über Seitengrenzen hinweg ausgeführt wird. Dies geschieht durch Angabe des Schalters *-DMOVE* beim Compilieren. Nun wird jedoch das, vom Prozessor unterstützte Prefetching umgangen und das Vorausladen durch direkte Ladebefehle durchgeführt. Ändert man das Beispiel ab, so daß der direkte Ladebefehl nur zum Laden einer neuen Seite, ansonsten aber ein Prefetch-Befehl verwendet wird, was mit Verwendung des Schalters *-DMOVEPREF* geschieht, läßt sich eine weitere Laufzeitverbesserung feststellen. Zum Vergleich zeigt der jeweils erste Balken der beiden Kolonnen die Messung ohne Vorausladen der Daten an. Um diese Messung nicht durch den, beim Vorausladen von Daten notwendigen Overhead zu verfälschen, wurde beim Compilieren die Anweisung *-DNONE* verwendet.

Listing 3.19: prefetchXeon.c

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4  #include <emmintrin.h>
5
6  #define LINESIZE 8
7  #define BLOCKSIZE 2048
8  #define RUNS 1024
9  #define RUNSPERBLOCK 4
10 #define BLOCKS 64
11 #define AHEAD BLOCKSIZE*LINESIZE
12
```

[1] siehe hierzu Kapitel 2.3.3
[2] engl.: exception

```
13 int main() {
14   int M = LINESIZE*BLOCKSIZE*BLOCKS;
15   int i, iBlock, j, jBlock, jBlockEnd,
16       jStep, jBlockStep, l, lEnd;
17   int ahead = AHEAD;
18   double *ptrA, count = 0, dummy;
19   clock_t c1,c2;
20
21   ptrA = _mm_malloc(M*sizeof(*ptrA),4096);
22
23   printf("Elemente: %d\n",M);
24
25   for (j=0;j<M;j++)
26     ptrA[j] = 1.0;
27
28   jStep = BLOCKSIZE*LINESIZE;
29   jBlockStep = LINESIZE*RUNSPERBLOCK;
30
31   c1 = clock();
32   for (i=0;i<RUNS;i+=RUNSPERBLOCK) {
33     for (j=0;j<M;j+=jStep) {
34       for (iBlock=0;iBlock<RUNSPERBLOCK;iBlock++) {
35         jBlockEnd = j+jStep;
36         for (jBlock=j;jBlock<jBlockEnd;jBlock+=jBlockStep)
37         {
38 #ifdef PREFETCH
39           _mm_prefetch( (const char*)(&ptrA[ahead]),
40             _MM_HINT_T2);
41 #endif
42 #ifdef MOVE
43           dummy += ptrA[ahead];
44 #endif
45 #ifdef MOVEPREF
46           if (ahead % 512 == 0)
47             dummy += ptrA[ahead];
48           else
49             _mm_prefetch( (const char*)(&ptrA[ahead]),
50               _MM_HINT_T2);
51 #endif
52 #ifndef NONE
53           ahead += LINESIZE;
54           if (ahead==M) ahead = 0;
55 #endif
56           lEnd = jBlock+jBlockStep;
57           for (l=jBlock;l<lEnd;l+=LINESIZE) {
```

```
58              count += ptrA[l];
59              count += ptrA[l+1];
60              count += ptrA[l+2];
61              count += ptrA[l+3];
62
63              count += ptrA[l+4];
64              count += ptrA[l+5];
65              count += ptrA[l+6];
66              count += ptrA[l+7];
67            }
68          }
69        }
70      }
71    }
72    c2 = clock();
73    double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
74    printf("value = %.0f, time = %.3f s\n", count, dt);
75
76    if (dummy>0) return 0;
77    return 0;
78  }
```

### 3.3.3 Vorausladen auf dem Itanium 2 Prozessor

Im Gegensatz zum Xeon unterstützt der Itanium 2 kein hardwaremäßiges Prefetching und ist deshalb besonders auf die Hilfe von Compiler und Softwareentwickler angewiesen.

Mithilfe des Intel C/C++ Compiler wurden Messungen durchgeführt, um festzustellen, inwieweit das Vorausladen von Daten auf dem Itanium 2 durch das Eingreifen des Softwareentwicklers verbessert werden kann. Dabei kam Listing 3.20 zum Einsatz, welches in abgewandelter Form Listing 3.19 entspricht. Entsprechend des, im Vergleich zum Xeon-Prozessors nur halb so großen L1-Cache wurde die verwendete Blockgröße halbiert und stattdessen die Größe einer *Cache Line* auf 16 double Werte angepaßt, was ein Abrollen der Schleife auf 16 Elemente erforderlich machte.

Listing 3.20: prefetchItanium.c

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <stdlib.h>
4 #include <ia64intrin.h>
5 #define LINESIZE 16
```

```
6  #define RUNS 1024
7  #define RUNSPERBLOCK 4
8  int main(int argc, char* argv[]) {
9    int BLOCKSIZE=1024;
10   int BLOCKS=64;
11   int M = LINESIZE*BLOCKSIZE*BLOCKS;
12   int i, iBlock, j, jBlock, jBlockEnd,
13       jStep, jBlockStep, l, lEnd;
14   int ahead = BLOCKSIZE*LINESIZE;
15   double *ptrA, count = 0;
16   clock_t c1,c2;
17
18   ptrA = malloc(M*sizeof(*ptrA)+16384);
19   (long)ptrA = ((long)ptrA/16384+1)*16384;
20
21   printf("Elemente: %d\n",M);
22
23   for (j=0;j<M;j++)
24     ptrA[j] = 1.0;
25
26   jStep = BLOCKSIZE*LINESIZE;
27   jBlockStep = LINESIZE*RUNSPERBLOCK;
28
29   c1 = clock();
30   for (i=0;i<RUNS;i+=RUNSPERBLOCK) {
31     for (j=0;j<M;j+=jStep) {
32       for (iBlock=0;iBlock<RUNSPERBLOCK;iBlock++) {
33         jBlockEnd = j+jStep;
34         for (jBlock=j;jBlock<jBlockEnd;jBlock+=jBlockStep)
35         {
36 #ifdef PREFETCHNONE
37             __lfetch_fault(__lfhint_none, &ptrA[ahead]);
38 #endif
39 #ifdef PREFETCHNTE
40             __lfetch_fault(__lfhint_nt1, &ptrA[ahead]);
41 #endif
42 #ifdef PREFETCHNTZ
43             __lfetch_fault(__lfhint_nt2, &ptrA[ahead]);
44 #endif
45 #ifdef PREFETCHNTA
46             __lfetch_fault(__lfhint_nta, &ptrA[ahead]);
47 #endif
48 #ifndef NONE
49           ahead += LINESIZE;
50           if (ahead==M) ahead = 0;
```

```
51 #endif
52           lEnd = jBlock+jBlockStep;
53           for (l=jBlock;l<lEnd;l+=LINESIZE) {
54             count += ptrA[l];
55             count += ptrA[l+1];
56             count += ptrA[l+2];
57             count += ptrA[l+3];
58
59             count += ptrA[l+4];
60             count += ptrA[l+5];
61             count += ptrA[l+6];
62             count += ptrA[l+7];
63
64             count += ptrA[l+8];
65             count += ptrA[l+9];
66             count += ptrA[l+10];
67             count += ptrA[l+11];
68
69             count += ptrA[l+12];
70             count += ptrA[l+13];
71             count += ptrA[l+14];
72             count += ptrA[l+15];
73           }
74         }
75       }
76     }
77   }
78   c2 = clock();
79   double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
80   printf("value = %.0f, time = %.3f s\n", count, dt);
81 }
```

Der Intel C/C++ Compiler für Itanium 2 unterstützt die Option Inline-Assembler-Anweisungen zu verwenden nicht. Aus diesem Grund kamen Intrinsics zum Einsatz. Dabei stellte sich heraus, daß der Befehl *lfetch.fault* der geeignetste für dieses Beispiel ist, da andere teilweise eine Verschlechterung der Laufzeit von Listing 3.20 verursachen. Da der Intel C/C++ Compiler den Code bereits sehr gut optimiert, ist durch weiteres Vorausladen von Daten kaum mehr eine Verbesserung zu erreichen. Die mit verschiedenen Optionen ermittelten Messwerte sind in Tabelle 3.7 zu sehen. Der C/C++ Compiler aus der „GNU Compiler Collection“, der ebenfalls kurz getestet wurde, liefert leider auf dem Itanium 2 zwei- bis dreimal schlechtere Ergebnisse, weshalb darauf verzichtet wurde, damit weitere Messungen durchzuführen.

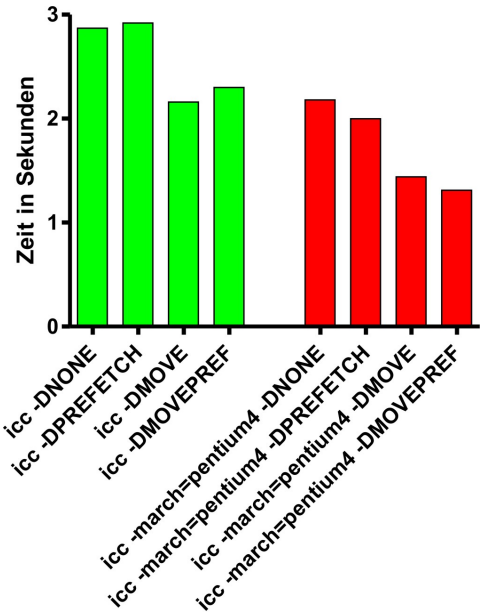


Abbildung 3.11: Prefetching auf dem Xeon. Der Text unter den Balken gibt die verwendeten Compileroptionen wieder.

Berechnungen mit im Quellcode vorhandenen Konstanten werden vom Compiler bereits bei der Übersetzung vorausberechnet. Diese fließen dann meist mit in die Optimierung ein, beispielsweise bei der Berechnung von Schleifendurchläufen. Entzieht man ihm das Wissen darüber, so kann er gewisse Optimierungen nicht mehr vornehmen. Genau dies geschieht, wenn man beispielsweise die beiden Zeilen aus Listing 3.21 in Listing 3.20 ab Zeile 11 einfügt. Obwohl diese Zeilen sinnlosen Code enthalten, der niemals ausgeführt wird, da die Bedingung niemals zutreffen kann, erkennt der Compiler nicht mehr, daß sich der Wert von *BLOCKSIZE* nie ändert. So seltsam dies klingen mag, verbessert sich dadurch das Laufzeitverhalten von Listing 3.20 ganz enorm, wie die dritte Spalte von Tabelle 3.7 zeigt.

| **Optionen** | **-O0** | **-O3** | **-O3 verändert** |
|---|---|---|---|
| **ohne Prefetching** | 34,929s | 3,490s | 1,785s |
| **lfetch.fault** | 32,479s | 3,473s | 1,792s |
| **lfetch.fault.nt1** | 32,480s | 3,455s | 1,787s |
| **lfetch.fault.nt2** | 32,510s | 3,454s | 1,788s |
| **lfetch.fault.nta** | 35,684s | 3,466s | 1,830s |

Tabelle 3.7: Messwerte für das Vorausladen von Daten auf dem Itanium 2 ohne und mit Optimierung, sowie ein leicht veränderter Quellcode

Listing 3.21: Zusätzliche Zeilen für Listing 3.20

```
1  if (argc<1 && argc>2)
2    BLOCKSIZE = atoi(argv[1]);
```

# Kapitel 4

# Bandbreiten und Messwerkzeuge

Dieses Kapitel soll zeigen, welche Speicherbandbreiten zur Verfügung stehen und wie weit diese ausgereizt werden können.

## 4.1 Messungen auf dem Xeon-Prozessor

Der eingesetzte Xeon-Prozessor ist mit 2,4 GHz getaktet. Sein L1-Cache ist 8 KByte groß, besitzt eine Assoziativität von vier und eine *Cache Line* kann 64 Byte an Daten aufnehmen. Der L2-Cache ist 512 KByte groß, besitzt eine Assoziativität von acht und jede *Cache Line* kann 64 Byte aufnehmen. Ein L3-Cache ist nicht vorhanden.

### 4.1.1 Fließkommaoperationen

Messungen anhand von Listing 4.1 ergeben eine sehr hohe Rate der pro Sekunde ausgeführten Fließkommabefehle[1], von über 3 Mrd. Dies allerdings nur deshalb, da der Compiler die Schleife optimiert. Ein Blick in den assemblierten Code wie Listing 4.2 zeigt, daß die Schleife immer um den Faktor fünf erhöht wird. Der Compiler hat aus einer Addition mit 1 eine Addition mit 5 gemacht und die Anzahl der Schleifendurchläufe auf ein Fünftel verringert. Damit ist die, von Listing 4.1 berechnete Rate der Fließkommabefehle um den Faktor 5 zu hoch. Zurückgerechnet ergibt dies einen Wert von etwa 600 Mio. tatsächlich ausgeführter Fließkommaoperationen pro Sekunde.

[1] engl.: floating-point operation, oder kurz flop

Listing 4.1: xeonFlops1.c

```
#include <stdio.h>
#include <time.h>
int main() {
  clock_t c1,c2;
  int i,j;
  double value = 0.0;

  c1 = clock();
  for (i=0;i<100;i++)
    for (j=0;j<64000000;j++)
      value+=1.0;
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("value: %.0f, time = %.3f, flops = %.3f\n",
    value, dt, value/dt);
}
```

Listing 4.2: xeonFlops1.s

```
..B1.4:                         # Preds ..B1.4 ..B1.3
  addsd     %xmm0, %xmm1                                  #14.7
  addl      $5, %edx                                      #13.25
  cmpl      $64000000, %edx                               #13.5
  jl        ..B1.4        # Prob 100%                     #13.5
..B1.5:                         # Preds ..B1.4
  addl      $1, %eax                                      #12.18
  cmpl      $100, %eax                                    #12.3
  jl        ..B1.3        # Prob 99%                      #12.3
```

Dennoch ist eine höhere Rate von Fließkommabefehlen je Sekunde möglich, wie Listing 4.3 zeigt. Hier kommt *ADDPS*, ein Befehl aus der Menge der SSE-Operationen zum Einsatz. Er addiert zwei XMM Register, die jeweils vier Fließkommawerte einfacher Genauigkeit enthalten. Auch die Kontrolle der Assemblercodes zeigt, daß die sechs Befehle innerhalb der Schleife, genauso wie im Quellcode zu sehen, durchlaufen werden. Es läßt sich eine Rate von 4.800 MFlops pro Sekunde messen, was immerhin zwei Fließkommaoperationen pro Taktzyklus ergibt. Diese Rate in der Praxis zu erreichen, ist allerdings ziemlich unwahrscheinlich. Im Beispiel werden nur Werte, die sich bereits im Prozessor befinden, aufsummiert, während in regulären Programmen weitere Operationen und Speicherzugriffe stattfinden, welche die Rate drücken.

### 4.1.2 Bandbreite des L1-Cache

Da die wenigsten Algorithmen während der Abarbeitung mit den im Prozessor zur Verfügung stehenden Registern auskommen, muß immer wieder ein Speicherzugriff erfolgen. Es soll untersucht werden, wie viele Daten binnen eines Taktzyklus vom L1-Cache in die Register gebracht werden können. Hierfür kommt Listing 4.4 zum Einsatz. Der verwendete SSE2-Befehl *MOVAPD* lädt dabei immer einen Datenblock der Größe 128 Bit in ein XMM Register. Um Datenabhängigkeiten zu vermeiden, werden zwei unterschiedliche Speicherblöcke in zwei separate XMM Register geladen. Dabei läßt sich eine Übertragungsrate von fast 39 GByte/s[1] messen.

Die Übertragungsrate bei schreibenden Zugriffen wird mithilfe von Listing 4.5 gemessen. Auch hier kommt wieder der SSE2-Befehl *MOVAPD* zum Einsatz, nur ist diesmal die Quelle ein XMM Register und das Ziel eine 128 Bit-Speichervariable. Beim Schreiben sinkt die gemessene Übertragungsrate auf fast 22 GByte/s ab. Die Messergebnisse sind nochmals in Tabelle 4.1 aufgeführt.

### 4.1.3 Bandbreite des L2-Cache

Die möglichen Übertragungsraten des L2-Cache werden in ähnlicher Weise gemessen, wie bereits die des L1-Cache. Nur muß hier sichergestellt werden, daß die Daten, bis auf wenige Ausnahmen während der Initialisierungsphase, auch wirklich aus dem L2-Cache und nicht aus dem L1-Cache, oder dem Hauptspeicher geladen werden. Dazu muß gezielt auf jene Adressen zugegriffen werden, die im L1-Cache alle im selben Set

[1] im folgenden sind mit GByte immer $10^9$ Byte gemeint.

Listing 4.3: xeonFlops2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <emmintrin.h>
#define RUNS 320000000
int main() {
  clock_t c1,c2;
  int i;
  float (*p) = _mm_malloc(10*sizeof(*p),16);
  p[0] = p[1] = p[2] = p[3] = 1.0;

  asm("MOV     %0, %%eax\n\t"
      "MOVAPS  (%%eax), %%xmm0\n\t"
      "MOVAPS  %%xmm0, %%xmm1\n\t"
      "MOVAPS  %%xmm0, %%xmm2\n"
        : : "m" (p) : "eax", "xmm0", "xmm1", "xmm2");
  c1 = clock();
  for (i=0;i<RUNS;i++) {
    asm("ADDPS   %%xmm2, %%xmm0\t\n");
    asm("ADDPS   %%xmm2, %%xmm1\t\n");
    asm("ADDPS   %%xmm2, %%xmm0\t\n");
    asm("ADDPS   %%xmm2, %%xmm1\t\n");
    asm("ADDPS   %%xmm2, %%xmm0\t\n");
    asm("ADDPS   %%xmm2, %%xmm1\t\n");
  }
  c2 = clock();
  asm("MOV     %0, %%eax\n\t"
      "MOVAPS  %%xmm0, (%%eax)\n" :: "m" (p) : "eax", "xmm0");

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, flops = %.3f\n", dt, RUNS*6/dt*4);
}
```

Listing 4.4: xeonL1Read.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <emmintrin.h>
#define L 8
#define M 32
#define N 8
#define RUNS 2000000000
main() {
  clock_t c1,c2;
  double (*p)[M][L] = _mm_malloc(N * sizeof(*p),16);
  int i, j, k, x;

  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      for (k=0;k<L;k++)
        p[i][j][k] = 1.0;

  c1 = clock();
  asm("MOV      %0, %%esi\n" : : "m" (p) : "esi");
  for (i=0; i<RUNS; i++) {
    asm("MOVAPD    (%%esi), %%xmm0\n\t"
        "MOVAPD  16(%%esi), %%xmm1\n\t" : : : "xmm0", "xmm1");
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*2*16/1000000000);
}
```

Listing 4.5: xeonL1Write.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <emmintrin.h>
#define L 8
#define M 32
#define N 8
#define RUNS 2000000000
main() {
  clock_t c1,c2;
  double (*p)[M][L] = _mm_malloc(N * sizeof(*p),16);
  int i, j, k, x;

  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      for (k=0;k<L;k++)
        p[i][j][k] = 1.0;

  c1 = clock();
  asm("MOV     %0, %%esi\n" : : "m" (p) : "esi");
  asm("MOVAPS  32(%%esi), %%xmm0\n\t"
      "MOVAPS  48(%%esi), %%xmm1\n\t" : : : "xmm0", "xmm1");
  for (i=0; i<RUNS; i++) {
    asm("MOVAPS  %%xmm0, (%%esi)\n\t"
        "MOVAPS  %%xmm1, 16(%%esi)\n\t":);
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*2*16/1000000000);
}
```

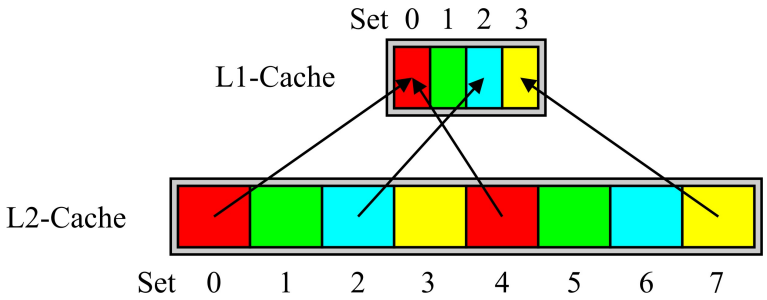


Abbildung 4.1: Zuordnung der einzelnen Sets innerhalb einer Cachehierarchie. Die Farben und Pfeile verdeutlichen dabei, wo Daten aus dem L2-Cache im L1-Cache abgelegt werden

abgelegt werden. Benutzt man dabei beispielsweise immer nur die erste Adresse in jeder *Cache Line*, führt dies spätestens nach fünf Zugriffen, und ab da für jeden weiteren Zugriff zu einem Miss im L1-Cache. Der L2-Cache kann die hierbei anfallende Datenmenge problemlos aufnehmen, so daß Fehlzugriffe im L2-Cache nur während der Initialisierungsphase auftreten. Abbildung 4.1 zeigt hierzu ein kleines Beispiel.

Aus den gegebenen Werten der L2-Cache werden die Mengen $\mathbb{S}_i$ ermittelt, welche jene Sets des L2-Cache enthalten, deren *Cache Lines* alle auf dasselbe Set $i$ im L1-Cache abgebildet werden.

$$\mathbb{S}_i = \{s | s = i + n \cdot 32, i = \text{Nr. des Sets im L1-Cache}, n = 0..31\} \tag{4.1}$$

Für das erste Set im L1-Cache bedeutet dies konkret, daß die Speicherstellen

$$\mathbb{M}_0 = \{m | m = s * 32, s \in \mathbb{S}_0\} \tag{4.2}$$

die gewünschte Bedingung erfüllen. Gleichzeitig muß darauf geachtet werden, daß der Prefetching-Mechanismus des Xeon-Prozessors nicht aktiv wird, deshalb werden nur jene Elemente aus $\mathbb{M}_0$ verwendet, die der Bedingung

$$\mathbb{N}_0 = \{n | n \in \mathbb{M}_0 \wedge (n = 0 \; mod \; 2^{12} \vee n = 128 \; mod \; 2^{12})\} \tag{4.3}$$

genügen. Somit können innerhalb einer 4 KByte-Seite immer zwei Zugriffe durchführt werden, was hier hilft, den Overhead für die Schleifen zu verringern. Hier sind die Werte $n = 128 \; mod \; 2^{12}$ noch willkürlich gewählt, ebenso gut wären beispielsweise die Werte $n = 64 \; mod \; 2^{12}$ geeignet. Bei den Messungen zum L2-Cache wird dies erneut aufgegriffen und die Sektorgröße des L2-Cache beim Xeon-Prozessor genauer untersucht.

Mit Listing 4.6 soll die Bandbreite des L2-Cache bezüglich Lesezugriffe gemessen werden. Um das gewünschte Ergebnis zu erhalten, muß auch hier der vom Compiler erzeugte Assemblercode in Listing 4.7 nachträglich bearbeiten. Um die zusätzlichen Speicherzugriffe für das Laden der Zeiger zu vermeiden, muß der Assemblercode wie in Listing 4.8 dargestellt, umgeschrieben werden.

Die Bandbreite des L2-Cache bezüglich Schreibzugriffe wurde mit Listing 4.9 gemessen. Ein Eingriff auf Assemblerebene bringt hier nichts, wie Messungen ergeben haben, da sich die Lesezugriffe aufgrund der größeren Latenzzeit beim Schreiben nicht bemerkbar machen. Die Ergebnisse der Messungen sind in Tabelle 4.1 aufgeführt.

### 4.1.4 Bandbreite des Hauptspeichers

Die Bandbreitenmessung des Hauptspeichers erfolgt in ähnlicher Weise wie beim L2-Cache. Allerdings müssen hierzu Überlegungen angestellt werden, welche Daten zu laden sind, so daß auch tatsächlich die Speicherbandbreite gemessen wird. In den Bandbreitenmessungen sollen folgende Speicherstellen geladen werden:

$$\mathbb{P}_0 = \{p | p = 0 \; mod \; 2^{16}\} \tag{4.4}$$

$$\mathbb{P}_1 = \{p | p = 2^6 \; mod \; 2^{16}\} \tag{4.5}$$

$$\mathbb{P}_2 = \{p | p = 2^7 \; mod \; 2^{16}\} \tag{4.6}$$

Die Speicherstellen aus $\mathbb{P}_i$, mit $i \in \{0, 1, 2\}$ werden dabei in das entsprechende Set $i$ des L2-Cache abgelegt. Im Grunde sollte sich die Bandbreitenmessung äquivalent zu der des L1-Cache durchführen lassen. Dies ist jedoch nicht der Fall. Nur ein Set zu überladen[1] führt nicht zum gewünschten Erfolgt. Dieses Problem kann umgangen werden, indem mehrere Sets benutzt werden, auf die im Wechsel zugegriffen wird. Im Listing 4.10 wurde dabei jedes vierundsechzigste Set überladen. Messungen mit *Perfsum* zeigen auch, daß die Rate der Misses des L2-Cache sich mit der Anzahl geladener *Cache Lines* deckt. Es werden doppelt so viele Lesezugriffe auf den Bus ausgeführt, wie Misses erzeugt wurden, da der L2-Cache des Xeon-Prozessors bei einem Lesezugriff aufgrund der Sektorgröße von 128 Byte immer zwei *Cache Lines* füllt. Dies ergibt eine gemessene Bandbreite auf den Hauptspeicher, von 2,12 GByte/s.

[1] mit überladen ist dabei das Auffüllen des Sets mit mehr *Cache Lines* als dieses fassen kann

Listing 4.6: xeonL2Read.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define L 8
#define M 32
#define N 20
#define RUNS 100000000
#define SET0 0
#define SET1 2
main() {
  clock_t c1,c2;
  double (*p)[M][L] = _mm_malloc(N * sizeof(*p),16);
  int i, j, k;

  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      for (k=0;k<L;k++)
        p[i][j][k] = 1.0;

  c1 = clock();
  for (i=0; i<RUNS; i++)
    for (j=0; j<N; j+=2) {
      asm("MOV␣␣␣␣␣␣␣%0,␣%%esi\n\t"
          "MOVAPD␣␣␣␣(%%esi),␣%%xmm0\n\t"
          "MOV␣␣␣␣␣␣␣%1,␣%%esi\n\t"
          "MOVAPD␣␣␣␣(%%esi),␣%%xmm1\n\t"
            ::"m" (&p[j][SET0][0]),
              "m" (&p[j][SET1][0]) : "esi", "xmm0", "xmm1");
    }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time␣=␣%.3f,␣GByte␣per␣second␣=␣%.3f\n",
    dt, RUNS/dt*(N/2)*2/1000000000);
}
```

Listing 4.7: xeonL2Read.s

```
1  ..B2.13:                        # Preds ..B2.13 ..B2.12
2    movl      %eax, 32(%esp)                                #38.0
3    movl      %edi, 36(%esp)                                #38.0
4    addl      $4096, %eax                                   #37.20
5    addl      $4096, %edi                                   #37.20
6  # Begin ASM
7    MOV       32(%esp), %esi
8    MOVAPD    (%esi), %xmm0
9    MOV       36(%esp), %esi
10   MOVAPD    (%esi), %xmm1
11 # End ASM                                                 #38.0
12   cmpl      %edi, %ecx                                    #37.5
13   jg        ..B2.13       # Prob 90%                      #37.5
```

Listing 4.8: xeonL2ReadOptimized.s

```
1  ..B2.13:                        # Preds ..B2.13 ..B2.12
2    movl      %eax, 32(%esp)                                #38.0
3    movl      %edi, 36(%esp)                                #38.0
4  # Begin ASM
5    MOVAPD    (%eax), %xmm0
6    MOVAPD    (%edi), %xmm1
7  # End ASM                                                 #38.0
8    addl      $4096, %eax    #37.20
9    addl      $4096, %edi                                   #37.20
10   cmpl      %edi, %ecx                                    #37.5
11   jg        ..B2.13       # Prob 90%                      #37.5
```

Listing 4.9: xeonL2Write.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define L 8
#define M 32
#define N 20
#define RUNS 100000000
#define SET0 0
#define SET1 2
main() {
  clock_t c1,c2;
  double (*p)[M][L] = _mm_malloc(N * sizeof(*p),16);
  int i, j, k;

  p[0][0][0] = 1.0;
  p[0][0][1] = 1.0;

  asm("MOV        %0, %%esi\n\t"
      "MOVAPD     (%%esi), %%xmm0\n\t"
      "MOVAPD     (%%esi), %%xmm1\n\t"
        ::"m" (&p[0][0][0]) : "esi", "xmm0", "xmm1");
  c1 = clock();
  for (i=0; i<RUNS; i++)
    for (j=0; j<N; j+=2) {
      asm("MOV        %0, %%esi\n\t"
          "MOVAPD     %%xmm0, (%%esi)\n\t"
          "MOV        %1, %%esi\n\t"
          "MOVAPD     %%xmm1, (%%esi)\n\t"
            ::"m" (&p[j][SET0][0]),
              "m" (&p[j][SET1][0]) : "esi", "xmm0", "xmm1");
    }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*(N/2)*2/1000000000);
}
```

Listing 4.10: xeonMemRead.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <emmintrin.h>
#define L 8
#define M 1024
#define N 16
#define RUNS 100000
main() {
  clock_t c1,c2;
  double (*p)[M][L] = _mm_malloc(N * sizeof(*p),16);
  int i, j, k;

  c1 = clock();
  for (i=0; i<RUNS; i++)
    for (j=0; j<N; j++) {
      for (k=0; k<1024 ;k+=256) {
        asm("MOVL      %0,      %%esi\n\t"
            "MOVL      %%esi,   %%edi\n\t"
            "ADDL      $4096,   %%edi\n\t"
            "MOVAPD    (%%esi), %%xmm0\n\t"
            "MOVAPD    (%%edi), %%xmm1\n\t"
            "ADDL      $8192,   %%esi\n\t"
            "ADDL      $8192,   %%edi\n\t"
            "MOVAPD    (%%esi), %%xmm2\n\t"
            "MOVAPD    (%%edi), %%xmm3\n\t"
             :: "m" (&p[j][k][0])
             : "esi", "edi", "xmm0", "xmm1", "xmm2"
        );
      }
    }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*N*4*(1024/256)*128/1000000000);
}
```

| Speicherebene | Bandbreite lesend | Bandbreite schreibend |
|---|---|---|
| **L1-Cache** | 38,56 GByte/s | 21,99 GByte/s |
| **L2-Cache** | 34,60 GByte/s | 9,04 GByte/s |
| **Hauptspeicher** | 2,12 GByte/s | 1,49 GByte/s |

Tabelle 4.1: Gemessene Bandbreiten der einzelnen Speicherebenen auf dem Xeon-Prozessor mit 2,40 GHz

Die Messungen für die Bandbreite beim Schreiben auf den Hauptspeicher erfolgten in ähnlicher Weise. In Listing 4.11 kamen deshalb dieselben Sets zum Einsatz. Wieder wurde auf jedes vierundsechzigste zugegriffen, diesmal jedoch schreibend. Dabei ergab die Messung eine Datenrate von 1,49 GByte/s.

Im Gegensatz zu anderen Messungen kamen hier Compilerschalter zur Optimierung zum Einsatz, da hier eine Verbesserung festgestellt werden konnte.

## 4.2 Messungen auf dem Itanium 2 Prozessor

Der verwendete Itanium 2-Prozessor ist mit 1,3 GHz getaktet. Die Cache-Größen, Assoziativitäten und die Größen der einzelnen *Cache Line* sind nochmals in Abbildung 4.2 zu sehen. Dort sind auch die maximalen Übertragungsraten zwischen den einzelnen Speicherebenen aufgeführt. Die folgenden Kapitel sollen zeigen, welche Übertragungsrate mit Programmen, die exzessiv auf Speicher zugreifen, erreicht werden kann.

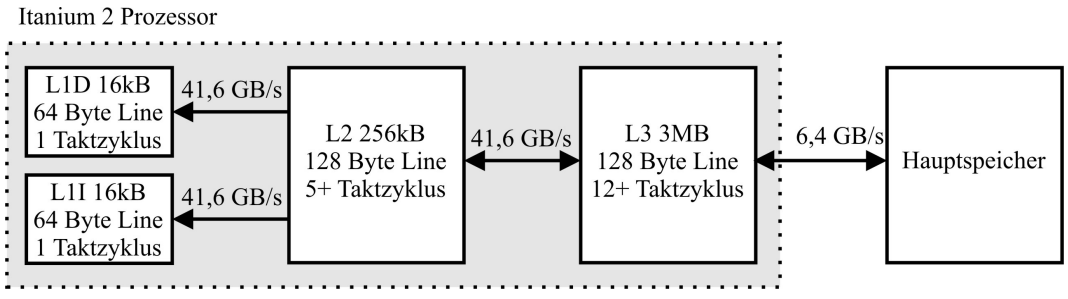


Abbildung 4.2: Übersicht über die Cache-Hierarchie des Itanium 2

Listing 4.11: xeonMemWrite.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <emmintrin.h>
#define L 8
#define M 1024
#define N 16
#define RUNS 100000
main() {
  clock_t c1,c2;
  double (*p)[M][L] = _mm_malloc(N * sizeof(*p),16);
  int i, j, k;

  c1 = clock();
  for (i=0; i<RUNS; i++)
    for (j=0; j<N; j++) {
      for (k=0; k<1024 ;k+=256) {
        asm("MOVL      %0,      %%esi\n\t"
            "MOVL      %%esi,   %%edi\n\t"
            "ADDL      $4096,   %%edi\n\t"
            "MOVAPD    %%xmm0, (%%esi)\n\t"
            "MOVAPD    %%xmm1, (%%edi)\n\t"
            "ADDL      $8192,   %%esi\n\t"
            "ADDL      $8192,   %%edi\n\t"
            "MOVAPD    %%xmm2, (%%esi)\n\t"
            "MOVAPD    %%xmm3, (%%edi)\n\t"
             :: "m" (&p[j][k][0])
             : "esi", "edi", "xmm0", "xmm1", "xmm2"
        );
      }
    }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*N*4*(1024/256)*128/1000000000);
}
```

### 4.2.1 Fließkommaoperationen

Die Geschwindigkeit bezüglich Fließkommaoperationen wurde mithilfe von Listing 4.12 gemessen. Indem die Schleife immer weiter abgerollt wird und so der Overhead verringert wird, nähert sich der Wert schließlich, bei einem Abrollfaktor von acht, 2.475 MFlops pro Sekunde an. Ein weiteres Abrollen bewirkt keine Verbesserung des Messwerts mehr.

Listing 4.12: itaniumFlops.c

```
1  #include <stdio.h>
2  #include <time.h>
3  int main() {
4   clock_t c1,c2;
5    int i;
6    double value[8];
7
8    for (i=0; i<8;i++)
9      value[i] = 0.0;
10
11   c1 = clock();
12   for (i=0;i<640000000;i++) {
13     value[0]+=1.0;
14     value[1]+=1.0;
15     value[2]+=1.0;
16     value[3]+=1.0;
17     value[4]+=1.0;
18     value[5]+=1.0;
19     value[6]+=1.0;
20     value[7]+=1.0;
21   }
22   c2 = clock();
23
24   double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
25   printf("value: %.0f, time = %.3f, flops = %.3f\n",
26     value[0], dt, value[0]/dt*8);
27 }
```

In Listing 4.13 sind kleine Ausschnitte des Assemblerlistings von Listing 4.12 zu sehen, um die Korrektheit der Messung zu überprüfen. In Zeile 5 wird das Register R16 mit dem Wert 0x026259fff = 639999999 geladen, da es sich dabei um eine fußgesteuerte Schleife handelt. In Zeile 10 wird das Schleifenregister[1], mit dem Wert aus Register R16

[1]Register LC, wobei LC für loop count steht

gefüllt. Im Anschluss erfolgen die einzelnen Berechnungen, mithilfe des Befehls *fma.d fa=fc,fd,fb*[1]. Dieser multipliziert die übergebenen Register *fc* und *fd* miteinander und addiert anschließend den Inhalt des übergebenen Registers *fb* hinzu. Das Ergebnis wird in das durch *fa* spezifizierte Register abgelegt. In Zeile 18 bedeutet dies konkret, daß der Inhalt des Registers *f13* mit dem Inhalt des Registers *f1*[2] multipliziert und anschließend der Inhalt des Registers f1 hinzu addiert. Das Ergebnis wird wieder in Register *f13* abgelegt. Dieser Befehl bewirkt also nichts anderes als eine Addition des Wertes 1.0 zum Register *f13*. Durch den Befehl *br.cloop.sptk label* wird zum angegebenen Label *label* gesprungen. Dabei wird der Wert des Schleifenregisters LC dekrementiert. Die Angabe von *sptk* teilt dem Prozessor mit, daß keine Vorhersage für diesen Sprung zu erzeugen ist, sondern der Sprung immer durchgeführt werden soll. Daraus ergibt sich, daß die errechneten Fließkommaoperationen auch tatsächlich durchgeführt werden und somit der gemessene MFlop-Wert korrekt ist.

### 4.2.2 Bandbreite des L1-Cache

Die Bandbreite des L1-Cache wird in ähnlicher Weise gemessen, wie schon beim Xeon-Prozessor geschehen. Dies erweist sich allerdings als nicht so einfach wie beim Xeon-Prozessor, da der Itanium 2 viel mehr Register besitzt und der Compiler deshalb versucht, diese mit den Daten zu füllen und anschließend nur noch mit den Registern zu rechnen. Da der Intel C/C++ Compiler für den Itanium 2 keinen Inline-Assemblercode unterstützt, wird hierfür der C/C++ Compiler aus der „GNU Compiler Collection“ verwendet.

Listing 4.14 zeigt den Quellcode, mit dessen Hilfe die Messungen durchgeführt wurden. Die Ergebnisse der Bandbreitenmessungen, mit unterschiedlicher Anzahl von Ladebefehlen innerhalb der Schleife, sind in Tabelle 4.2 zu sehen. Messungen über die Zugriffe auf den L1-Cache, die mit dem Programm *Pfmon* erfolgt sind, wurden ebenfalls in die Tabelle aufgenommen. Man sieht, wie sich die vom Programm erzeugten Zugriffe auf den L1-Cache, einschließlich Overhead für die Schleifen und aller sonstigen Initialisierungen, den Messergebnissen sehr gut annähern. Die Berechnung der Bandbreite anhand der Zugriffe erfolgte dabei mit der Formel

$$\frac{\text{READS} \cdot 8}{\text{TIME}} \cdot \frac{1}{10^9} \frac{\text{GByte}}{\text{s}} \tag{4.7}$$

[1] Floating-point Multiply Add

[2] es soll daran erinnert werden, daß im Itanium 2 das Register f0 fest auf den Wert 0.0 und das Register f1 fest auf den Wert 1.0 verdrahtet ist

Listing 4.13: itaniumFlops.s

```
[...]
//   14 :     for (i=0;i<640000000;i++) {
 {    .mlx
        ldfd     f11=[r38]                                  //1: 14   29
        movl     r16=0x026259fff                            //1: 14   33
 }
 {    .mmi
        ldfd     f10=[r37] ;;                               //1: 14   30
        ldfd     f9=[r42]                                   //2: 14   27
        mov      ar.lc=r16                                  //2: 14   34
 }
[...]
//   15 :       value[0]+=1.0;
//   16 :       value[1]+=1.0;
.b1_12:
 {    .mfi
        nop.m    0
        fma.d    f13=f13,f1,f1                              //0: 16  109
        nop.i    0
 }
 {    .mfi
        nop.m    0
        fma.d    f12=f12,f1,f1                              //0: 15  108
        nop.i    0 ;;
 }
[...]
 {    .mfb
        nop.m    0
        fma.d    f6=f6,f1,f1                                //3: 21  114
// Branch taken probability 0.99
        br.cloop.sptk    .b1_12 ;;                          //3: 14  116
 }
[...]
```

| **Lesezugriffe** | **10** | **20** | **50** | **100** | **200** | **800** | **2000** |
|---|---|---|---|---|---|---|---|
| **Gemessene GByte/s** | 7,45 | 10,91 | 15,25 | 17,58 | 19,03 | 20,20 | 20,54 |
| **Berechnete GByte/s** | 8,78 | 12,01 | 15,86 | 17,93 | 19,22 | 20,33 | 20,57 |

Tabelle 4.2: Gemessene Bandbreite des L1-Cache beim Itanium 2 und, mittels der von *Pfmon* ermittelten Anzahl von Zugriff auf den L1-Cache, berechnete Bandbreite

READS steht dabei für die mithilfe von *Pfmon* gemessene Anzahl von Lesezugriffe auf den L1-Cache und TIME für die dafür benötigte Zeit. Diese Berechnung ist nicht exakt, da davon ausgegangen wurde, daß bei jedem Zugriff auf den Speicher 8 Byte an Daten geladen werden, was nicht der Fall ist. Aufgrund der guten Übereinstimmung mit den Messwerten zeigt sich aber, daß diese Näherung hinreichend genau ist. Angemerkt sei noch, daß dem Abrollen der Schleife gewisse Grenzen gesetzt sind. Mit dem Abrollfaktor von 2000 nähert sich dieser Wert der Grenze dessen, was der L1-Instruktions-Cache aufnehmen kann, läuft dieser über, so verringert sich die Laufzeit wieder.

Da es sich beim L1-Cache des Itanium 2 um einen Cache handelt, der geschriebene Daten immer an die nächste Ebene weiter reicht[1], kann die Geschwindigkeit des L1-Cache im Schreibzugriff nicht gemessen werden, da in dem Fall immer die Zeit gemessen wird, die Schreibzugriffe vom Register bis in den L2-Cache benötigen. Dennoch kann die Datenmenge errechnet werden, die je Sekunde vom Prozessor in den L1-Cache übertragen wird. Dazu wird Listing 4.16 verwendet, welches die Bandbreite beim Schreiben auf den L2-Cache mißt. Da der Prozessor bei den einzelnen Schreibzugriffen immer 8 Byte in den L1-Cache schreibt und dieser 64 Byte an den L2-Cache weiter reicht, errechnet sich die Datenrate vom Register in den L1-Cache zu $1/8$ der Datenrate vom Register in den L2-Cache. Dies entspricht einer Bandbreite von 2,16 GByte/s.

### 4.2.3 Bandbreite des L2-Cache

Mithilfe von Listing 4.15 wurde die Bandbreite des L2-Cache gemessen. Um die beste gemessene Bandbreite von 31,60 GByte/s zu erreichen, müssen acht Datenströme verwendet werden, die ineinander verflechtet gelesen werden. Bei nur vier Datenströmen verringert sich der Durchsatz bereits auf 27,21 GByte/s.

Die Datenrate des L2-Cache beim Schreiben wurde mithilfe von Listing 4.16 ermittelt. Da der L1-Cache geschriebene Daten direkt an den L2-Cache weiter reicht, sind

[1] engl.: write-through

Listing 4.14: itaniumL1Read.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define L 8
#define M 64
#define N 8
#define RUNS 100000000
main() {
  clock_t c1,c2;
  long (*p)[M][L] = malloc(N * sizeof(*p));
  int i;

  c1 = clock();
  asm("ld8  r20 = %0\n\t"
      ";;"
      : : "m" (&p[0][0][0]) : "r20");

  for (i=0; i<RUNS; i++) {
    asm("ld8 r21 = [r20] \n\t"
        "ld8 r22 = [r20] \n\t"
        ";; \n\t"
        "ld8 r21 = [r20] \n\t"
        "ld8 r22 = [r20] \n\t"
        ";; \n\t"
        ::: "r21", "r22"
    );
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*4*8/1000000000);
}
```

Listing 4.15: itaniumL2Read.c

```
[...]
#define L 8
#define M 64
#define N 8
#define RUNS 100000000
main() {
  clock_t c1,c2;
  long (*p)[M][L] = malloc(N * sizeof(*p));
  int i, j;

  for (i=0;i<N-1;i++)
    for (j=0;j<8;j++)
      p[i][j][0] = (long)&p[i+1][j][0];
  for (j=0;j<8;j++)
    p[N-1][j][0] = (long)&p[0][j][0];

  long *p1 = &p[0][0][0];
[...]
  long *p8 = &p[0][7][0];

  c1 = clock();
  asm("ld8  r20 = %0\n\t" : : "m" (p1): "r20");
[...]
  asm("ld8  r27 = %0\n\t" : : "m" (p8): "r27");

  for (i=0; i<RUNS; i++) {
    asm("ld8 r20 = [r20] \n\t"
        "ld8 r21 = [r21] \n\t"
        ";; \n\t"
[...]
        "ld8 r26 = [r26] \n\t"
        "ld8 r27 = [r27] \n\t"
        ";; \n\t"
        :::"r20","r21","r22","r23","r24","r25","r26","r27"
    );
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*16*64/1000000000);
}
```

hier keine besonderen Maßnahmen zu ergreifen. Bei jedem Schreibzugriff erfolgt ein Schreiben in den L2-Cache, was auch Messungen mittels *Pfmon* bestätigen. Der erreichte Durchsatz liegt bei 17,16 GByte/s.

### 4.2.4 Bandbreite des L3-Cache

Die Messung der Bandbreite des L3-Cache erweist sich als nicht so einfach durchführbar. Die bisherige Messmethode immer ein Set zu überladen und so Fehlzugriffe zu erzwingen, führt hier nicht zum Ziel, was einerseits die Zeitmessungen vermuten lassen und andererseits Messungen mithilfe von *Perfsum* belegen. Der, mit der Formel

$$\text{ABSTAND} = \frac{\text{CS}}{\text{ASSOC}} = \text{CLS} \cdot \text{SETS} \tag{4.8}$$

für den L3-Cache berechnete Abstand[1] zwischen zwei, im selben Set abgelegte Speicheradressen, scheint nicht korrekt zu sein. Messungen mit potentiell möglichen Werten ergaben dabei eigenartigerweise einen notwendigen Abstand von mindestens 64 KByte, damit der gewünschte Effekt eintritt.

Aus diesem Grund wurde hier auf eine andere Strategie ausgewichen. Eine Bandbreitenmessung ist in jedem Fall möglich, wenn anstelle einzelner Sets der gesamte Cache mehrfach gefüllt wird, da hierbei zwangsweise Fehlzugriffe auftreten müssen. Wie in Listing 4.17 zu sehen, kam dabei ein Speicherbereich von 2 MByte zum Einsatz. Durch das Lesen eines 8 Byte großen long-Wertes alle 128 Byte wird erreicht, daß es sich bei allen Zugriffen auf den L2-Cache um Fehlzugriffe handelt. Außerdem wurde festgestellt, daß durch das Lesen von zwei Datenströmen die beste Bandbreite von immerhin 27,54 GByte/s erreicht werden kann. Da beim Überladen des gesamten Caches im Gegensatz von nur einem Set das Auftreten von Konflikten mit anderen laufenden Prozessen sehr viel wahrscheinlicher ist, muß diese Messung mehrmals wiederholt werden, da sich hierdurch durchaus auch sehr wesentlich schlechtere Werte ergeben können.

Die Bandbreite beim Schreiben in den L3-Cache wurde mithilfe von Listing 4.18 ermittelt. Dabei konnte eine Datenrate von 15,50 GByte/s erreicht werden.

[1] ABSTAND stehe für den Abstand in Byte, CS für die Größe des Caches in Byte, ASSOC für die Assoziativität, CLS für die Größe einer *Cache Line* in Byte und SETS für die Anzahl der Sets

Listing 4.16: itaniumL2Write.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define L 8
#define M 64
#define N 8
#define RUNS 20000000
main() {
  clock_t c1,c2;
  long (*p)[M][L] = malloc(N * sizeof(*p));
  int i;

  long *p1 = &p[0][0][0], *p2 = &p[0][1][0],
       *p3 = &p[0][2][0], *p4 = &p[0][3][0];

  c1 = clock();
  asm("ld8  r21 = %0\n\t"
      "ld8  r22 = %1\n\t"
      "add  r20 = 5,r0\n\t"
      ";;"
      "ld8  r23 = %2\n\t"
      "ld8  r24 = %3\n\t"
      ";;"
      : : "m" (p1), "m" (p2), "m" (p3), "m" (p4)
        : "r21", "r22", "r23", "r24");

  for (i=0; i<RUNS; i++) {
    asm("st8 [r21] = r20\n\t"
        "st8 [r22] = r20\n\t"
        ";; \n\t"
[...]
        "st8 [r23] = r20\n\t"
        "st8 [r24] = r20\n\t"
        ";; \n\t"
    );
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*40*64/1000000000);
}
```

Listing 4.17: itaniumL3Read.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define RUNS 10000000
#define MEMSIZE 2*1024*1024
#define ABSTAND 128/8
main() {
  clock_t c1,c2;
  int i, j;
  long n = MEMSIZE / (ABSTAND * 8), *p1;
  long (*p) = malloc(MEMSIZE + 128);

  for (i=0;i<n-4;i+=4)
    for (j=0;j<4;j++)
      p[(i+j)*ABSTAND] = (long)&p[(i+j+4)*ABSTAND];

  for (j=0;j<4;j++)
    p[(n-4-j)*ABSTAND] = (long)&p[j*ABSTAND];

  long q1 = (long)&p[0*ABSTAND];
[...]
  long q4 = (long)&p[3*ABSTAND];

  c1 = clock();
  asm("ld8  r20 = %0\n\t" : : "m" (q1): "r20");
[...]
  asm("ld8  r23 = %0\n\t" : : "m" (q4): "r23");
  for (i=0; i<RUNS; i++) {
    asm("ld8 r20 = [r20] \n\t"
        "ld8 r21 = [r21] \n\t"
        ";;"
[...]
        "ld8 r22 = [r22] \n\t"
        "ld8 r23 = [r23] \n\t"
        ";;"
        ::: "r20", "r21", "r22", "r23"
    );
  }
  c2 = clock();
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*16*128/1000000000);
}
```

Listing 4.18: itaniumL3Write.c

```
[...]
#define RUNS 20000000
#define MEMSIZE 12*1024*1024
#define ABSTAND 128
main() {
  clock_t c1,c2;
  int i, j;
  long n = MEMSIZE / (ABSTAND * 8);
  long (*p) = malloc(MEMSIZE + 128);  // 12 MB

  long q1 = (long)&p[0];
[...]
  long q4 = (long)&p[(MEMSIZE/8)/4*3];

  c1 = clock();
  for (j=0; j<RUNS/4000; j++) {
    asm("ld8  r21 = %0\n\t"
[...]
        "mov  r25 = %4\n\t"
        :: "m"(q1),"m"(q2),"m"(q3),"m"(q4),"i"(ABSTAND)
        : "r20","r21","r22","r23","r24","r25");

    for (i=0; i<4000; i++) {
      asm("st8 [r21] = r20\n\t"
          "st8 [r22] = r20\n\t"
          "add r21 = r21, r25\n\t"
          ";; \n\t"
          "st8 [r23] = r20\n\t"
          "st8 [r24] = r20\n\t"
          "add r22 = r22, r25\n\t"
          ";; \n\t"
          "add r23 = r23, r25\n\t"
          "add r24 = r24, r25\n\t"
          ";; \n\t"
          ::: "r20", "r21", "r22", "r23"
      );
    }
  }
  c2 = clock();
  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*4*128/1000000000);
}
```

| Speicherebene | Bandbreite lesend | Bandbreite schreibend |
|---|---|---|
| **L1-Cache** | 20,54 GByte/s | (berechnet) 2,16 GByte/s |
| **L2-Cache** | 31,60 GByte/s | 17,16 GByte/s |
| **L3-Cache** | 27,54 GByte/s | 15,50 GByte/s |
| **Hauptspeicher** | 4,35 GByte/s | 2,18 GByte/s |

Tabelle 4.3: Gemessene Bandbreiten der einzelnen Speicherebenen auf dem Itanium 2 Prozessor mit 1,30 GHz

### 4.2.5 Bandbreite des Hauptspeichers

Wie bereits beim L3-Cache muß auch hier der gesamte Cache überfüllt werden. Hierzu wird erneut Listing 4.19 verwendet, der Wert für *MEMSIZE* aber auf $6 \cdot 1024 \cdot 1024$ vergrößert. Hierbei konnte noch eine Datenrate im Lesezugriff von 4,35 GByte/s gemessen werden.

Beim Schreibzugriff ergab sich ein weiteres Problem bei den Messungen, welches jedoch mathematisch korrigiert werden konnte. Die Messungen mittels Listing 4.20 liefern eine Bandbreite von 2,18 GByte/s. Jedoch fiel auf, daß bei einem sehr geringen Wert für die Durchläufe die mittels *Pfmon* gemessene Anzahl der Schreibzugriffe auf den Bus nicht mit der vom Programm ermittelten Datenrate übereinstimmt. Dies liegt daran, daß der L3-Cache beim Beenden des Programms die noch verbliebenen Daten nicht zurück in den Hauptspeicher schreibt. Somit ist die ermittelte Datenrate um den $\frac{1}{n-1}$ zu hoch. Bei höherer Anzahl der Durchläufe läßt sich dieser aber aufgrund der ohnehin gegebenen Messungenauigkeiten vernachlässigen.

Alle gemessenen Bandbreiten des Itanium 2 sind nochmals in Tabelle 4.3 zusammengefaßt dargestellt.

## 4.3 Messwerkzeuge

Bei diesen und den folgenden Messungen kamen hauptsächlich zwei Tools zum Einsatz, die hier kurz beschrieben werden sollen.

Listing 4.19: itaniumMemRead.c

```
[...]
#define RUNS 1000000
#define MEMSIZE 6*1024*1024
#define ABSTAND 256/8
main() {
  clock_t c1,c2;
  int i, j;
  long n = MEMSIZE / (ABSTAND * 8), *q[8];
  long (*p) = malloc(MEMSIZE + 128);

  for (i=0;i<n-8;i+=8)
    for (j=0;j<8;j++)
      p[(i+j)*ABSTAND] = (long)&p[(i+j+8)*ABSTAND];

  for (j=0;j<8;j++)
    p[(n-8+j)*ABSTAND] = (long)&p[j*ABSTAND];

  for (i=0;i<8;i++)
    (long)q[i] = (long)&p[i*ABSTAND];

  c1 = clock();
  asm("ld8  r20 = %0\n\t" : : "m" (q[0]): "r20");
[...]
  asm("ld8  r27 = %0\n\t" : : "m" (q[7]): "r27");

  for (i=0; i<RUNS; i++) {
    asm("ld8 r20 = [r20] \n\t"
        "ld8 r21 = [r21] \n\t"
        ";;"
[...]
        "ld8 r26 = [r26] \n\t"
        "ld8 r27 = [r27] \n\t"
        ";;"
        ::: "r20","r21","r22","r23","r24","r25","r26","r27"
    );
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, RUNS/dt*16*128/1000000000);
}
```

Listing 4.20: itaniumMemWrite.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define RUNS 1000
#define MEMSIZE 12*1024*1024
#define ABSTAND 128
main() {
  clock_t c1,c2;
  int i, j;
  long (*p) = malloc(MEMSIZE);

  c1 = clock();
  for (j=0;j<RUNS;j++) {
    asm("ld8  r22 = %0\n\t"
        "mov  r20 = 2\n\t"
        "mov  r21 = %1\n\t"
        ";;\n\t"
        :: "m"(p),"i"(ABSTAND) : "r20","r21","r22");
    for (i=0; i<MEMSIZE/ABSTAND/4; i++) {
      asm("st8 [r22] = r20\n\t"
          "add r22   = r21, r22\n\t"
          ";; \n\t"
          "st8 [r22] = r20\n\t"
          "add r22   = r21, r22\n\t"
          ";; \n\t"
          "st8 [r22] = r20\n\t"
          "add r22   = r21, r22\n\t"
          ";; \n\t"
          "st8 [r22] = r20\n\t"
          "add r22   = r21, r22\n\t"
          ";; \n\t"
          :::"r20","r21","r22","r23","r24","r25","r26","r27"
      );
    }
  }
  c2 = clock();

  double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
  printf("time = %.3f, GByte per second = %.3f\n",
    dt, MEMSIZE/ABSTAND/4*4/dt*128*RUNS/1000000000);
}
```

### 4.3.1 PerfCtr

Das Tool PerfCtr [1] wird unter der *GNU General Public Licenses* für Linux entwickelt und steht frei zur Verfügung. Es ermöglicht, die bereits angesprochenen *Performance-Monitoring Counter* innerhalb von Benutzerprozessen unter Linux auszulesen. Dazu erhält jeder Prozess seine eigenen virtuellen PMCs, was es ermöglicht, die Ereignisse unabhängig von anderen Prozessen zu ermitteln. Der Aufruf von PerfCtr erfolgt von der Kommandozeile aus, mit Übergabe eines Programms und der zu messenden Ereignisse. Dadurch mißt PerfCtr die Anzahl aller angegebenen Ereignisse, die während des Programmablaufs aufgetreten sind.

Die Angabe der zu messenden Ereignisse ist sehr kompliziert. Mit dem Perl-Skript *perfsum* steht ein kleines Frontend zur einfacheren Messung einiger Ereignisse zur Verfügung. Ein Beispiel über die zu ermittelnden Ereignisse gibt Tabelle 3.6.

### 4.3.2 Pfmon

Auf dem Itanium 2 kam das Tool *pfmon* der Hewlett-Packard Development Company zum Einsatz. Es bietet, wie auch PerfCtr die Möglichkeit an, während eines kompletten Programmablaufs eingetretene Ereignisse zu zählen. Dabei werden beide IA64-Prozessoren von Intel unterstützt, sowohl den Itanium als auch den Itanium 2. Tabelle 4.4 gibt einen kleinen Überblick über Ereignisse, die mittels *pfmon* während der Ausführung des Unix-Befehls *ls* gemessen wurden.

Im Gegensatz zu *PerfCtr* reicht es bei *Pfmon* aus, beim Aufruf aus der Kommandozeile den Namen des Ereignisses anzugeben. *Pfmon* bietet hier über diverse Optionen umfangreiche Listen der möglichen Ereignisse sowie eine Erklärung dazu abzurufen.

[1] Linux Performance-Monitoring Counters Driver

| **Anzahl** | **Ereignisse** |
|---:|---|
| 140044 | L1D_READS_SET1 |
| 20290 | L1D_READ_MISSES_ALL |
| 1439426 | CPU_CYCLES |
| 21311 | L2_DATA_REFERENCES_L2_DATA_READS |
| 276 | L3_READS_DATA_READ_MISS |
| 1255 | L3_MISSES |
| 3665 | L3_READS_DATA_READ_ALL |
| 372 | BUS_MEM_READ_ALL_ANY |
| 2167 | BUS_DATA_CYCLE |
| 553 | BUS_MEMORY_ALL_ANY |

Tabelle 4.4: Beispielmessung mittels *pfmon* bei der Ausführung des Unix-Befehls *ls*

# Kapitel 5

# Mehrgitterverfahren

In diesem Kapitel sollen die Möglichkeiten der Optimierung von Mehrgitterverfahren auf Assemblerebene untersucht werden.

## 5.1 Einführung zum Mehrgitterverfahren

Viele Vorgänge in der Physik oder Technik lassen sich mithilfe von mathematischen Modellen, welche auf elliptischen partiellen Differentialgleichungen[1] basieren, erklären. Zur Lösung dieser kommen nummerische Verfahren zum Einsatz, von denen das Mehrgitterverfahren mitunter das effizienteste darstellt.

Partielle Differentialgleichungen lassen sich mithilfe numerischen Methoden aber nur näherungsweise berechnen. Um die linearen Gleichungssysteme mit iterativen Verfahren zu lösen, muß die DGL diskretisiert werden. Je feiner dabei die Diskretisierung vorgenommen wird, desto genauer ist die zu erwartende Lösung, um so mehr steigt aber auch der erforderliche Rechenaufwand.

[1] kurz DGLn

### 5.1.1 Problemstellung

Die generelle Form der verwendeten DGLn zweiter Ordnung auf einem Gebiet $G$ ist:

$$\begin{gathered} Au_{xx} + 2Bu_{xy} + Cu_{yy} + Dux + Euy + Fu \;=\; H \\ A^2 + B^2 + C^2 \neq 0 \\ AC - B^2 > 0, \forall (x,y) \in G \end{gathered} \tag{5.1}$$

Die gegebenen Koeffizienten $A$, $B$, $C$, $D$, $E$, $F$ und $H$ können dabei stückweise stetige Funktionen von x und y sein.

Klassische Beispiele von elliptischen Differentialgleichungen sind [SHR97]:

$$\begin{aligned} u_{xx} + u_{yy} &= 0 \qquad\quad Laplace - Gleichung \\ u_{xx} + u_{yy} &= f(x,y) \quad Poisson - Gleichung \end{aligned} \tag{5.2}$$

Die Poisson-Gleichung, welche beispielsweise die stationäre Temperaturverteilung in einem homogenen Medium oder den Spannungszustand bei bestimmten Torsionsproblemen beschreibt, soll im Weiteren betrachtet werden, da das folgende zu optimierende Beispiel eine DGL dieser Art löst. Dabei wird die Bedingung vorausgesetzt, daß die Funktion $u$ für den gesamten Rand des Gebietes $G$ gegeben ist. Diese Bedingung nennt man auch *Dirichlet-Bedingung*. Neben dieser existieren weiter, wie die *Neumann-* oder *Cauchy-Bedingung*, auf die im Folgenden jedoch nicht näher eingegangen wird.

### 5.1.2 Diskretisierung

Eine Poisson-Gleichung soll in einem Gebiet $G$ diskretisiert werden. Hierzu wird jede Dimension in $n$ Teilintervalle aufgeteilt. Diese Teilintervalle werden durch die Gitterpunkte $(x_i, y_j) = (ik, jk)$, mit $k = \frac{1}{n}$ und $i, j = 0, 1, ..., n$ beschrieben. Dies ist in Abbildung 5.1 nochmals graphisch für ein Gebiet $G$ dargestellt. An jedem inneren Punkt des Gitters wird der Differentialquotient anhand des Differenzenquotienten aus den Nachbarpunkten angenähert, wie Abbildung 5.2 für den Differentialquotienten $u_x$ im Punkt $(i, j)$ zeigt. Da es sich um DGLn zweiter Ordnung handelt, werden aus den an-

genäherten Differentialquotienten wiederum Differentialquotienten errechnet. Dabei ergeben sich folgende Gleichungen:

$$\begin{aligned} u_x(x_i, y_i) &\approx \frac{u_{i+1,j} - u_{i-1,j}}{2h} \\ u_y(x_i, y_i) &\approx \frac{u_{i,j+1} - u_{i,j-1}}{2h} \\ u_{xx}(x_i, y_i) &\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \\ u_{yy}(x_i, y_i) &\approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} \end{aligned} \tag{5.3}$$

Um im Weiteren die Schreibweise zu vereinfachen, werden die Indizes der Nachbarpunkte eines Punktes $P$ durch die Himmelsrichtungen mit N, S, W und O ersetzt:

$$\begin{aligned} u_P &:= u_{i,j} \\ u_N &:= u_{i,j+1} \\ u_S &:= u_{i,j-1} \\ u_W &:= u_{i-1,j} \\ u_O &:= u_{i+1,j} \end{aligned} \tag{5.4}$$

Setzt man nun die Gleichungen aus (5.3) in die Poisson-Gleichung aus den Gleichungen (5.2) ein, erhält man eine DGL, welche die Poisson-Gleichung im Gitterpunkt $P$ annähert:

$$\frac{u_O - 2u_P + u_w}{h^2} + \frac{u_N - 2u_P + u_S}{h^2} = f_P, \quad f_P = f(x_i, y_j) \tag{5.5}$$

Zusammengefaßt und mit $-h^2$ multipliziert ergibt dies:

$$4u_P - u_N - u_S - u_W - u_O + h^2 f_P = 0 \tag{5.6}$$

was häufig in der Operatorform [SHR97] dargestellt wird:

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} \odot u + h^2 f_P = 0 \tag{5.7}$$

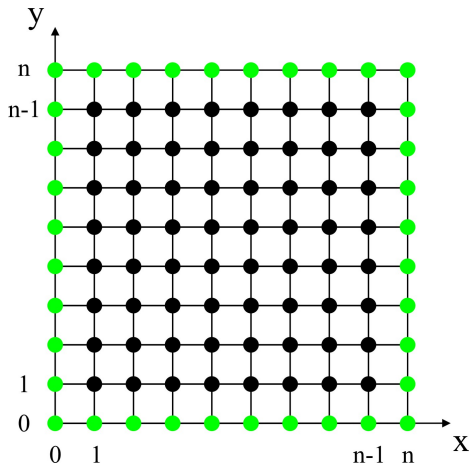


Abbildung 5.1: Das Gebiet $G$ einer DLG, aufgeteilt in diskrete Punkte und grün gekennzeichnete Randpunkte

Auf alle inneren Punkte angewandt erhält man folgendes lineares Gleichungssystem [Wei01]:

$$\begin{pmatrix} A & -I & & & \\ -I & A & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & A & -I \\ & & & -I & A \end{pmatrix} \begin{pmatrix} \vec{u}_1 \\ \vec{u}_2 \\ \vdots \\ \vec{u}_{n-2} \\ \vec{u}_{n-1} \end{pmatrix} = \begin{pmatrix} \vec{f}_1 \\ \vec{f}_2 \\ \vdots \\ \vec{f}_{n-2} \\ \vec{f}_{n-1} \end{pmatrix} \tag{5.8}$$

$$I = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & 1 \\ & & & 1 & \end{pmatrix} \quad A = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix}$$

$$\vec{u}_i = (u_{i_1}, u_{i_2}, ..., u_{i_{n-2}}, u_{i_{n-1}}),\ 1 \leq i \leq n-1$$

$$\vec{f}_i = (f_{i_1}, f_{i_2}, ..., f_{i_{n-2}}, f_{i_{n-1}}),\ 1 \leq i \leq n-1$$

Dieses lineare Gleichungssystem kann beispielsweise mit dem Gauß-Seidel-Verfahren gelößt werden. Dieses zerstört jedoch sehr schnell die Dünnbesetztheit der Matrix. Dies führt zu höherem Speicherverbrauch, was nicht akzeptabel ist. Außerdem ist die Komplexität der Gauß-Elimination mit $O(n)^6$ sehr hoch. Ein besseres Laufzeitverhalten erreicht man mit Bandmatrix-Verfahren ($O(n^4)$, auf der Fourier-Transformation basierende direkte Verfahren ($O(n^2 \log n)$) oder iterativen Methoden [Wei01].

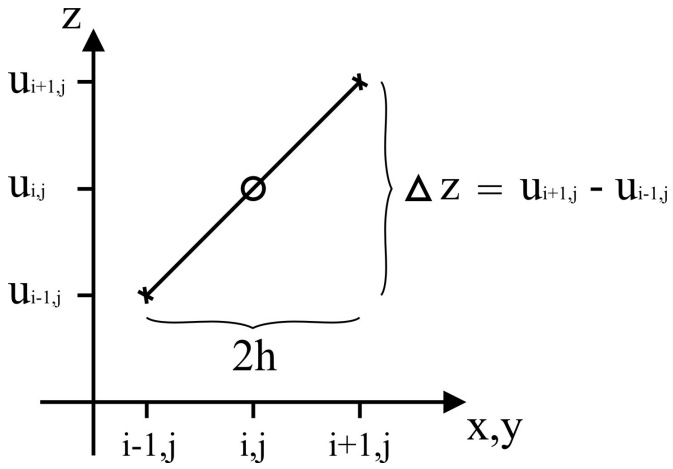


Abbildung 5.2: Näherung des Differentialquotienten $u_x(x_i, y_j)$ im Punkt $(i, j)$ durch Berechnung des Differenzenquotienten mithilfe der Werte an den Punkten $(i-1, j)$ und $(i+1, j)$

### 5.1.3 Idee des Mehrgitterverfahrens

Ein Gitter mit diskreten Punkten kann eine stetige Funktion nur endlich genau approximieren. Es besteht zwar die Möglichkeit, die Gitterpunkte enger zu wählen, verringert zwar diesen Fehler, allerdings leidet auch die Reduzierung niederfrequenter Fehler darunter, da diese mit feinerem Gitter langsamer wird. Gleichzeitig steigen aber die durchzuführenden Rechenschritte an. Somit entsteht durch feinere Gitter mehr Arbeit und darunter leidet wieder die Effizienz. Der wohl entscheidende Grund, Mehrgitterverfahren einzusetzen, ist, daß ein niederfrequenter Fehler auf einen groben Gitter zu einem hochfrequenten Fehler wird [Wei01]. Es existieren mehrere Arten, wie Vergröberung von Gittern in Mehrgitterverfahren eingesetzt werden kann. Wir beschränken uns hier auf das Korrekturverfahren.

Das Korrekturverfahren beginnt mit einer guten Näherung $v^h$ der exakten Lösung $u^h$ des Gleichungssystems $A^h u^h = f^h$. Anschließend wird wie folgt verfahren [Mat04]:

$$\begin{aligned}
&\text{glätte } A^h v^h = f^h \\
&r^h = f^h - A^h v^h \\
&f^{2h} = r^{2h} = R_h^{2h} r^h \\
&\text{löse } A^{2h} e^{2h} = f^{2h} \\
&e^h = P_{2h}^h e^{2h} \\
&v_{neu}^h = v^h + e^h
\end{aligned} \tag{5.9}$$

Im ersten Schritt wird die Näherung geglättet. Anschließend wird das Residuum ($r^h = A^h e^h = A^h(u^h - v^h) = A^h u^h - A^h v^h = f^h - A^h v^h$, wobei $e^h = u^h - v^h$). Der nächste Schritt, auch *Restriktion* genannt, überträgt das Residuum auf das nächst höhere Gitter. Als nächster wird durch Lösen der Gleichung löse $A^{2h}e^{2h} = f^{2h}$ der Fehler auf dem gröberen Gitter berechnet. Dieser wird im Anschluß auf das feinere Gitter übertragen, was man *Prolongation* nennt. Der errechnete Fehler $e^h$ kann nun zur Korrektur auf $v^h$ addiert werden. Werden auf das gröbere Gitter weiter Restriktionen rekursiv angewandt und folgen vom gröbsten Gitter an nur noch Prolongationen, so erhält man ein Schema namens V-Zyklus. Es sei noch erwähnt, daß weitere Schemata, wie beispielsweise das des $\mu$-Zyklus, existieren.

### 5.1.4 Glätten und Parallelisierung

Der rechenintensivste Teil von Mehrgitterverfahren ist das Glätten auf dem feinsten Gitter. Das Gauß-Seidel-Verfahren ist direkt geeignet, jeden Gitterpunkt parallel zu berechnen. Dazu müssen die neu berechneten Werte in einem weiteren Feld zwischengespeichert werden. Würde man die Werte direkt überschrieben, so würden bei der Berechnung einzelner Punkte sowohl alte als auch neue Werte einfließen. Diese Problem kann man aber umgehen, indem man die Punkte in zwei Gruppen unterteilt, was in Abbildung 5.3 dargestellt ist. Dieses Verfahren nennt man auch Rot-Schwarz-Gauß-Seidel-Verfahren. Dabei können neu berechnete Werte wieder in das Gitter zurückgeschrieben werden und dennoch findet die Berechnung der einzelnen Gitterpunkte nur mit Werten derselben Iteration statt, da rote Punkte bei der Berechnung nur von schwarzen und schwarze Punkte nur von roten Punkten abhängen, wie ebenfalls in Abbildung 5.3 zu sehen.

## 5.2 Optimierung von Mehrgitterverfahren

Der Einfachheit halber soll hier nur der Glätter eines Mehrgitterverfahrens auf einem 2-dimensionalen Gitter untersucht werden.

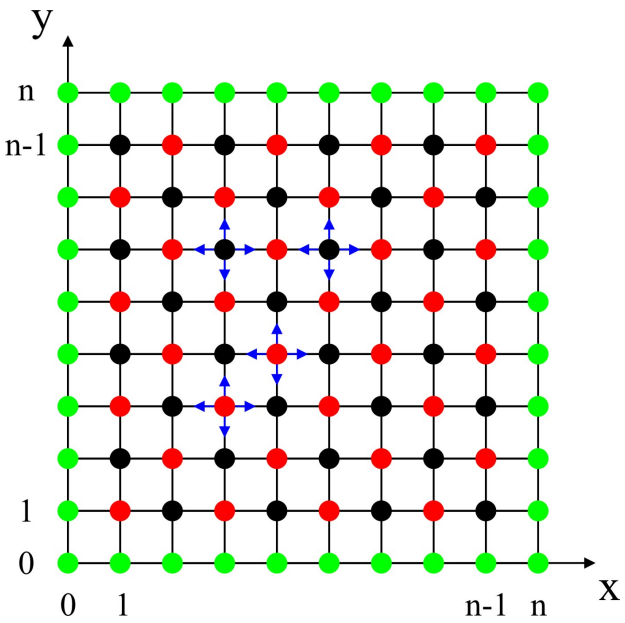


Abbildung 5.3: Gitter mit Randpunkten und in zwei Partitionen (Rot und Schwarz. Die blauen Pfeile stellen die Datenabhängigkeit einiger ausgewählter Punkte bei der Berechnung des Gleichungssystems dar.) unterteilten inneren Punkten

### 5.2.1 Vorüberlegungen

Zu den Datenstrukturen, die für das Glätten beim Mehrgitterverfahren verwendet werden, sollen vor der Implementation noch gewisse Überlegungen angestellt werden.

Es kommen zwei Felder unterschiedlichen Typs zum Einsatz:

- Zum Einen das Feld, das die Gitterwerte aufnehmen soll. Dieses ist entsprechend dem Gitter 2-dimensional und besteht aus Fließkommawerten doppelter Genauigkeit. Bei diesem könnte es notwendig werden, *intra array padding* einzusetzen, wie in Abbildung 3.5 dargestellt. Einfache Überlegungen in Hinsicht darauf, daß sich zwei gemeinsam gelesene Zeilen nicht gegenseitig verdrängen, führen hier schnell zu brauchbaren Werten. Bei einer Feldgröße von $4097^2$ beispielsweise verdrängen sich nahe beieinander liegende Zeilen mit geradzahliger Zeilennummer gegenseitig, gleiches gilt für Zeilen ungerader Zeilennummer. Eine weitere Optimierung könnte darin bestehen, die nach dem Rot-Schwarz-Gauss-Seidel-Verfahren markierten Punkte entsprechend ihrer Färbung sortiert im Speicher abzulegen. Abbildung 5.4 zeigt, daß dies eine bessere Speichernut-

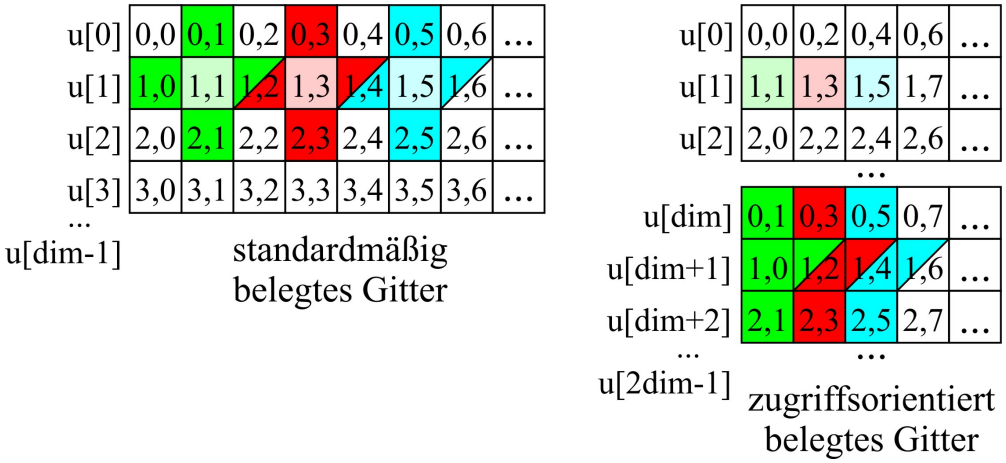


Abbildung 5.4: Zwei mögliche Gitterbelegungen. Links ein standardmäßig belegtes Gitter, rechts ein zugriffsorientiert belegtes Gitter. Die blaß markierten Felder (1,1), (1,3) und (1,5) entsprechen einem Teil der zu berechnenden Gitterpunkte, die kräftig markierten Felder den zur Berechnung benötigten Nachbarpunkten. Schräg markierte Felder werden entsprechend ihrer Färbung mehrmals verarbeitet. Die Zahlen *i,j* in den Feldern entsprechen den Positionen *(i,j)* in einem 2-dimensionalen Gitter.

zung zur Folge hat. Ob sich dieser Vorteil auch noch nach dem Einsatz von Blocking-Mechanismen bemerkbar macht, ist jedoch fraglich.

- Zum Anderen das Feld für die Koeffizienten. Dieses ist 3-dimensional, da für jede Gitterkoordinate ein Feld für sechs Koeffizienten, welche Fließkommawerten doppelter Genauigkeit entsprechen, reserviert werden muß. Da in unserem Fall das Gitter nur 2-dimensional ist, sollte hierdurch mehr Datenverkehr verursacht werden, als durch die Gitterpunkte selbst, da für jeden Gitterpunkt sechs Koeffizienten benötigt werden, die nur für einen Gitterpunkt verwendet werden können. Im Gegensatz dazu besitzt jeder innere Gitterpunkt vier Nachbarn, die bis zu vier mal in eine Berechnung einfließen können. Hier ist es ratsam, die zu einem Gitterpunkt gehörenden Koordinaten gemeinsam in ein Feld in der untersten Dimension[1] zu speichern. Bezüglich Padding sind zwar Vorüberlegungen anhand der gegebenen Spezifikationen der Caches nötig, exakte Werte hierzu lassen sich jedoch nur durch Laufzeitmessungen ermitteln.

Überlegungen bezüglich der Optimierung durch Blocking können nur durch die Betrachtung der gegebenen Cache-Spezifikationen der jeweiligen Prozessoren angestellt werden, da zu viele Faktoren eine Rolle spielen. Allgemein läßt sich sagen, daß dar-

[1]Dies entspricht in der Programmiersprache C für ein Feld f[i][j][k] der Dimension k.

auf zu achten ist, daß die Gitterpunkte, wie auch die Koeffizienten, die während des Durchlaufs durch einen Block benötigt werden, idealerweise im L1-Cache oder zumindest im L2-Cache Platz finden sollten. Aber bereits bei dieser einfachen Überlegung bleibt die Ersetzungsstrategie unberücksichtigt, da bei optimaler Ersetzung mehr Koeffizienten und dadurch wiederum mehr Gitterpunkte im Cache abgelegt werden könnten. Dies läßt sich nur durch konkrete Messungen am Beispiel ermitteln.

### 5.2.2 Optimierung auf CISC-Prozessoren

Ein kleines Perl-Skript, wie Listing 5.1 als Beispiel zeigt, wurde zu Beginn verwendet, um optimale Parameter für Blocken und das Einfüllen von Elementen zu finden. Dieses stupide Testen von möglichen Werten sollte nicht ohne Vorüberlegung ablaufen, da andernfalls ein zu untersuchendes Programm mit einer Laufzeit von wenigen Sekunden schnell stundenlange Testläufe zur Folge haben kann. Die Untersuchungen aus Kapitel 3.1.6 helfen dabei, brauchbare Grenzen, innerhalb derer die Werte zu suchen sind, abzuschätzen.

Listing 5.1: genTests.pl

```
1  #!/usr/bin/perl
2  open(SCRIPT, ">␣allTests");
3  for ($i=0; $i<=64; $i++) {
4    open(OUTPUT, ">␣padding/output".$i);
5    print OUTPUT "#define␣PADDING␣".$i."\n";
6    close(OUTPUT);
7    print SCRIPT "cp␣padding/output".$i."␣./mg_padd.h"."\n";
8    print SCRIPT "echo␣".$i."␣\n";
9    print SCRIPT "icc␣multiGrid.c␣-o␣multiGrid␣".
10     "-march=pentium4␣-fast␣-DXEON"."\n";
11   print SCRIPT "perfsum␣multiGrid␣>>␣messung"."\n";
12 }
13 close(SCRIPT);
```

Mithilfe von Listing 5.2 wurden Tests für ein Gitter mit konstanten Koeffizienten durchgeführt, wenn die Übersetzung mittels des Compilerschalters *-DKONST* durchgeführt wurde. Ohne diesen Compilerschalter wird das Programm für ein Gitter mit variablen Koeffizieten übersetzt. In den zugehörigen Header-Dateien Listing 5.4 und Listing 5.3 sind die entsprechenden Einträge zu sehen.

Listing 5.2: multigridXeon.c

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4  #include <emmintrin.h>
5
6  #define DIM 4106
7  #define LINESIZE 64
8  #define RUN 2
9  #define KPADf 2
10 #define XPADf 0
11 #define XBLOCK 32
12 #define YBLOCK 2
13 #define XPADu 4112-DIM
14
15 #include "mg_pref.h"
16 #include "mg_func.h"
17
18 int main() {
19   int x, xblock, xend, y, yblock, yend, run;
20   clock_t c1, c2;
21   long ops = 0;
22   double (*u)[DIM+XPADu] = malloc(DIM*sizeof(*u)+LINESIZE);
23   double (*f)[DIM+XPADf][6+KPADf] = malloc(DIM*sizeof(*f)+
24     +LINESIZE);
25   (int)u = ((int)u/LINESIZE+1)*LINESIZE;
26   (int)f = ((int)f/LINESIZE+1)*LINESIZE;
27   double dummy;
28   int aheadX=AHEAD, aheadY=1;
29
30   while (aheadX>DIM-XBLOCK) {
31     aheadX-=DIM;
32     aheadY+=YBLOCK;
33   }
34
35   init(u, f);
36
37   c1 = clock();
38   for (yblock=1;yblock<DIM-1;yblock+=YBLOCK) {
39     yend = yblock+YBLOCK;
40     if (yend>DIM-1) {
41       yend = DIM-1;
42     }
43     for (xblock=1;xblock<DIM-1;xblock+=XBLOCK) {
44       xend = xblock+XBLOCK;
```

```
45        if (xend>DIM-1) {
46    xend = DIM-1;
47        }
48        for (run=0;run<RUN;run++) {
49 #ifdef PREFETCH
50        if (&u[aheadY][aheadX]%4096<16) {
51          dummy += u[aheadY][aheadX];
52          dummy += u[aheadY][aheadX+8];
53        }
54        else {
55          _mm_prefetch( (const char*)(&u[aheadY][aheadX]),
56            _MM_HINT_T2);
57          _mm_prefetch( (const char*)(&u[aheadY][aheadX+8]),
58            _MM_HINT_T2);
59        }
60        aheadX+=XBLOCK/2;
61        if (aheadX>DIM+15) {
62          aheadX=0;
63          aheadY+=YBLOCK;
64        }
65        if (aheadY>DIM-3)
66          aheadY = 0;
67 #endif
68
69 // Rote Punkte
70    for (y=yblock;y<yend;y++) {
71      if (y%2) {
72        for (x=xblock+(xblock+1)%2;x<xend;x+=2) {
73          RELAX;
74          ops += 10;
75        }
76      }
77            else {
78              for (x=xblock+(xblock)%2;x<xend;x+=2) {
79                RELAX;
80                ops += 10;
81              }
82      }
83          }
84
85 // Schwarze Punkte
86          for (y=yblock;y<yend;y++) {
87            if (y%2) {
88              for (x=xblock+(xblock)%2;x<xend;x+=2) {
89                RELAX;
```

```
90                ops += 10;
91              }
92            }
93            else {
94              for (x=xblock+(xblock+1)%2;x<xend;x+=2) {
95                RELAX;
96                ops += 10;
97              }
98            }
99          }
100       }
101     }
102   }
103   c2 = clock();
104
105   double dt = (double)(c2-c1)/CLOCKS_PER_SEC;
106   printf("Operationen: %ld, Zeit = %.3f s\n", ops, dt);
107
108   return 0;
109 }
```

Da ein anfängliches Prefetchen eines vollständigen Blocks zu wesentlich schlechterer Laufzeit führte, wurde versucht, wie Abbildung 5.5 darstellt, nur Teile eines Blockes im Voraus zu laden, so daß beim anschließenden Bearbeiten des Blockes zumindest ein Teil bereits im Cache vorhanden ist.

Beides führte nicht zu Verbesserung der Laufzeit. Das Laufzeitverhalten der einzelnen Programme ist in Tabelle 5.1 dargestellt. Eine Messung mit *Perfsum* ergab eine Nutzung der Speicherbandbreite von etwa 75%, hier ist also noch etwas Spielraum. Eine Inspektion des Assemblercodes ergibt, daß der Intel C/C++ Compiler bei der Berechnung der Gitterelemente mittels *RELAX*[1] ausschließliche Befehle wie *MOVSD*, *ADDSD* und *MULSD* verwendet, die nur zwei Fließkommawerte verarbeiten. Besser ist eine Verwendung von Befehlen wie *MOVAPD* und *ADDPD*. Dazu würde sich die Änderung der Speicherbelegung wie in Abbildung 5.4 anbieten, da hier immer zwei Punkte mittels einem *MOVAPD* geladen werden könnten. Für eine weitere Optimierung sollte man also hier ansetzen, den Assemblercode optimieren und anschließend versuchen, eine bessere Nutzung der Speicherbandbreite zu erreichen.

[1] siehe Listing 5.4

Listing 5.3: mg_pref.h

```
1 #define AHEAD DIM+512
```

### 5.2.3 Optimierung auf EPIC-Prozessoren

Ebenso wie beim Xeon-Prozessor ließen sich auch auf dem Itanium 2, durch das Vorausladen von Daten, keine Laufzeitverbesserungen erreichen. Da mithilfe desselben Programms gemessen wurde, wie auf dem Xeon-Prozessor, zeigt Listing 5.5 nur die Änderungen, die für den Itanium 2 vorgenommen wurden. Bei den Header-Dateien Listing 5.4 und Listing 5.3 handelt es sich um dieselben, die bereits beim Xeon-Prozessor zum Einsatz kamen.

Die Ergebnisse der Laufzeitmessungen sind in Tabelle 5.2 zu sehen. Im Gegensatz zum Xeon-Prozessor fällt hier auf, daß sich die Laufzeit bei der Verwendung von konstanten Koeffizienten gegenüber der Verwendung variabler Koeffizienten kaum ändert. Wird das erzeugte Programm disassembliert[1], zeigt sich, daß der Compiler auf höchster Optimierungsstufe selbst Befehle zum Vorausladen von Daten einfügt. Bei Verwendung der Vorausladebefehle im Quellprogramm werden daher dieselben Daten mehrfach vorausgeladen, was zu einer schlechteren Laufzeit führt. Wird die Optimierung mithilfe von *-O0* unterbunden, ist zwar das Programm mit den implementierten Vorausladebefehlen schneller, als jenes ohne, jedoch macht sich die fehlende Optimierung in schlechterer Laufzeit bemerkbar. Dies zeigt, daß in diesem Fall der Compiler sehr guten Code erzeugt.

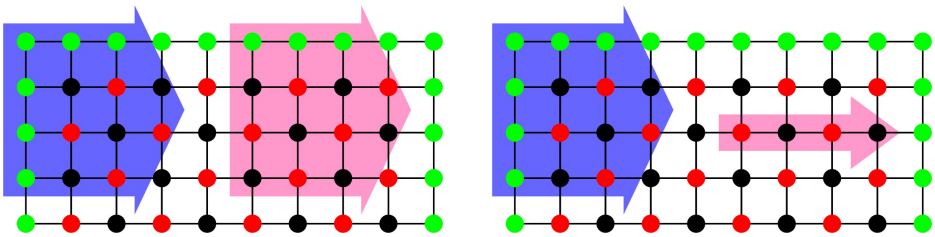

Abbildung 5.5: Vorgehensweise beim Prefetchen. Der blaue Pfeil deutet den abzuarbeitenden Block an, der rosa Pfeil die aktuelle Prefetch-Position. Links wird ein vollständiger Block im Voraus geladen, rechts nur der Teil eines Blockes.

[1] mithilfe von *objdump -d*

Listing 5.4: mg_func.h

```
#define RAND 0.0
#define GITTER 1.0
#define CF 0
#define CE 1
#define NO 2
#define SO 3
#define WE 4
#define EA 5
#ifndef KONST
#define RELAX u[y][x] = f[y][x][CF]+ \
  +f[y][x][CE]*u[y  ][x  ]+f[y][x][NO]*u[y-1][x  ]+ \
  +f[y][x][SO]*u[y+1][x  ]+f[y][x][WE]*u[y  ][x-1]+ \
  +f[y][x][EA]*u[y  ][x+1]
#endif
#ifdef KONST
#define RELAX u[y][x] = 1.0/2+ \
 +4.0     *u[y  ][x]+(-1.0/4)*u[y-1][x  ]+ \
 +(-1.0/4)*u[y+1][x  ]+(-1.0/4)*u[y  ][x-1]+ \
 +(-1.0/4)*u[y  ][x+1]
#endif
void init(double u[][DIM+XPADu], double f[][DIM+XPADf][6+KPADf]){
  int x,y;
  int mev;
  for (y=0;y<DIM;y++)
    for (x=0;x<DIM;x++)
      u[y][x] = GITTER;
  for (y=0;y<DIM;y++) {
    u[y][0] = u[y][DIM-1] = RAND;
  }
  for (x=0;x<DIM;x++) {
    u[0][x] = u[DIM-1][x] = RAND;
  }
  mev = -1.0/4;
  for (y=0;y<DIM;y++)
    for (x=0;x<DIM;x++) {
      f[y][x][CF] = 1.0/2;
      f[y][x][CE] = 4.0;
      f[y][x][NO] = f[y][x][SO] = mev;
      f[y][x][WE] = f[y][x][EA] = mev;
    }
}
```

Listing 5.5: multigridItanium.c

```
[...]
#include <stdlib.h>
#include <ia64intrin.h>

#define DIM 4106
#define LINESIZE 128
#define RUN 2
#define KPADf 2
#define XPADf 0
#define XBLOCK 64
#define YBLOCK 2
#define XPADu 4160-DIM

#include "mg_pref.h"
[...]
  long ops = 0;
  double (*u)[DIM+XPADu] = malloc(DIM*sizeof(*u)+LINESIZE);
  double (*f)[DIM+XPADf][6+KPADf] = malloc(DIM*sizeof(*f)+LINESIZE);
  (long)u = ((long)u/LINESIZE+1)*LINESIZE;
  (long)f = ((long)f/LINESIZE+1)*LINESIZE;
  double dummy;
[...]
      for (run=0;run<RUN;run++) {
#ifdef PREFETCH
        __lfetch_fault(__lfhint_none, &u[aheadY][aheadX]);
        __lfetch_fault(__lfhint_none, &u[aheadY][aheadX+8]);
        __lfetch_fault(__lfhint_none, &u[aheadY][aheadX+16]);
        __lfetch_fault(__lfhint_none, &u[aheadY][aheadX+24]);
        __lfetch_fault(__lfhint_none, &u[aheadY+1][aheadX]);
        __lfetch_fault(__lfhint_none, &u[aheadY+1][aheadX+8]);
        __lfetch_fault(__lfhint_none, &u[aheadY+1][aheadX+16]);
        __lfetch_fault(__lfhint_none, &u[aheadY+1][aheadX+24]);
        aheadX+=XBLOCK/2;
        if (aheadX>DIM+15) {
          aheadX=0;
          aheadY+=YBLOCK;
        }
        if (aheadY>DIM-3)
          aheadY = 0;
#endif

// Rote Punkte
[...]
```

| **Option** | **Laufzeit** |
|---|---|
| -DKONST -DNONE -march=pentium4 -fast | 0,250s |
| -DKONST -DPREFETCH -march=pentium4 -fast | 0,260s |
| -DNONE -march=pentium4 -fast | 0,740s |
| -DPREFETCH -march=pentium4 -fast | 0,760s |

Tabelle 5.1: Laufzeiten mit und ohne Prefetching, sowie mit konstanten und variablen Koeffizienten des Glätters eines Mehrgitterverfahrens auf dem Xeon-Rechner

| **Option** | **Laufzeit** |
|---|---|
| -DKONST -DNONE -O3 | 0,833s |
| -DKONST -DPREFETCH -O3 | 0,894s |
| -DNONE -O3 | 0,919s |
| -DPREFETCH -O3 | 0,980s |

Tabelle 5.2: Laufzeiten mit und ohne Prefetching, sowie mit konstanten und variablen Koeffizienten des Glätters eines Mehrgitterverfahrens auf dem Itanium-Rechner

# Kapitel 6

# Zusammenfassung und Ausblick

Nach anfänglicher Beschreibung heute eingesetzter Prozessorarchitekturen und einem detaillierteren Eingehen auf den Xeon- und Itanium-Prozessor, wurde ein Überblick über die Optimierungsmöglichkeiten auf Hochsprachenebene gegeben. Hier zeigte sich, daß ein Verständnis für viele Optimierungen, auch für ungeübte Programmierer angesichts der von modernen Compilern angebotenen Möglichkeiten wie *Pragmas*, auch ohne Assemblerwissen, schnell zu erreichen ist. Der vom Compiler erzeugte Code weist, je nach verwendeter Optimierungsstufe, gute Laufzeiteigenschaften auf.

Dennoch zeigen die vorangegangenen Messungen, daß durch die Optimierung auf Assemblerebene, trotz der mittlerweile guten Qualität verfügbarer Compiler, immer noch Laufzeitverbesserungen möglich sind. Die Verwendung von *Intrinsics* macht es, wie beschrieben, sehr einfach, auf Hochsprachenebene entsprechende Befehle einzubinden.

## 6.1 CISC-Architektur

Zum Xeon-Prozessor, der stellvertretend für die CISC-Architektur zum Einsatz kam, ist abschließend zu sagen, daß sich der Einsatz von Optimierung auf Assemblerebene durchaus lohnen kann.

Hierzu bietet sich die Möglichkeit an, Assemblercode direkt in den Quellcode einzubauen, wenn der Compiler dies unterstützt. Der assemblierte Code muß anschließend auf die überflüssige Verwendung von Registern durch den Compiler untersucht und entsprechend angepaßt werden, da ansonsten Laufzeiteinbußen die Folge sind.

Andererseits können auch mithilfe von Intrinsics Optimierungen vorgenommen werden. Ein nachträgliches Überarbeiten des assemblierten Codes an der entsprechenden Stelle ist dann allerdings oft zu empfehlen, da sich gezeigt hat, daß der verwendete Compiler den Datenzusammenhang nicht mehr erkennen kann und unnötige Speicherzugriffe durchführt.

Eine prinzipielle Optimierung des, vom Compiler assemblierten Codes ist, nicht anzuraten, da die wenigen vorhandenen Register bereits verwendet werden und auch ein Zusammenhang zwischen einzelnen Codestücken nur mit viel Zeitaufwand herstellbar ist.

Beim verwendeten Prozessor zeigte sich auch, daß die vorhandenen *XMM* Register eine hervorragende Möglichkeit der Parallelverarbeitung darstellen. Dazu sind jedoch, wie hier beim Mehrgitterverfahren, die verwendeten Datenstrukturen für eine Nutzung entsprechend anzupassen. Der Intel C/C++ Compiler bietet die Möglichkeit, die *XMM* Register und die zugehörigen SSE-/SSE2-Befehle, sowohl über Intrinsics als auch direkt über eingebetteten Assemblercode zu verwenden. Bei höheren Optimierungsstufen verwendet auch der Intel C/C++ Compiler selbst diese Befehle und Register, leider kommen in der Version 8.0 aber meist nur SSE2-Befehle zum Einsatz, die nur einzelne Fließkommazahlen verarbeiten. Hier ist oft durch direkten Eingriff in den Quellcode eine Laufzeitverbesserung zu erreichen.

## 6.2 EPIC-Architektur

Für den Itanium 2, der Vertreter für die EPIC-Architekturen, kommt eine direkte Optimierung auf Assemblerebene kaum in Frage.

Die Verwendung von Intrinsics ist jedoch zu empfehlen, da diese durchaus eine Verringerung der Laufzeit zur Folge haben können. Ein gezieltes Nachbearbeiten des assemblierten Codes zeigt sich jedoch als sehr schwierig, da der Compiler einzelne Quellcodezeilen oft im gesamten Assemblercode verstreut.

Inline-Assembler kam nur kurz mithilfe des C/C++ Compiler aus der „GNU Compiler Collection“ zum Einsatz, da der Intel C/C++ Compiler dies auf dem Itanium 2 nicht unterstützt. Der erzeugte Code weist aber große Mängel bei der Optimierung auf, weshalb der Einsatz des C/C++ Compiler aus der „GNU Compiler Collection“ auf dem Itanium 2 (noch) nicht anzuraten ist.

Seine Stärke zeigt der Itanium 2 sowohl in der riesigen Anzahl an Registern, die es erlauben, häufig benötigte Daten ständig vorrätig zu halten und dadurch Speicherzugriffe zu vermeiden, als auch durch die bedingte Befehlsausführung. Beides erschwert aber gleichzeitig die Programmierung und Optimierung von Assemblercode. Dies macht es umso wichtiger, daß der Compiler bereits gut optimierten Code erzeugt, was der Intel C/C++ Compiler durchaus leistet.

## 6.3 Fazit

Die Aufgabenstellung war durchaus sinnvoll, angesichts des gewaltigen Unterschieds heutiger Prozessorgeschwindigkeiten gegenüber der Geschwindigkeit des Hauptspeichers, zu untersuchen, ob hierbei die Bandbreite unter optimierter Nutzung der Assoziativspeicher besser genutzt werden kann.

Überraschend war angesichts der vom Prozessorhersteller angegebenen Bandbreiten zwischen Registern, Assoziativspeichern und Arbeitsspeicher, die geringere zu erreichende Geschwindigkeit bei den Bandbreitenmessungen. Ein Performancegewinn durch das Vorausladen von Daten war zu erwarten, dennoch überraschte der Fehlschlag beim Einsatz in das Mehrgitterverfahren. Möglicherweise wäre hier ein intensiveres und zeitaufwendiges Studium des Assemblercodes notwendig, um die vom Compiler erzeugten Berechnungen zu optimieren und anschließend zu prüfen, ob die Bandbreite überhaupt ein besseres Zugriffsverhalten auf die Daten erlaubt.

Da eine Optimierung auf Assemblerebene generell sehr viel zeitaufwendiger ist, als die Nutzung der vom Compiler zur Verfügung gestellten Optionen, bleibt von Fall zu Fall abzuwägen, ob der Einsatz eines Algorithmus diese Art der Optimierung erfordert. Angesichts dessen, daß der Erfolg eines Optimierungsversuches auf Assemblerebene nicht garantiert werden kann und oft viel Vorarbeit auf Hochsprachenebene erfordert, was beispielsweise die Verteilung der Daten im Speicher betrifft, ist der Aufwand der Einarbeitung in Assembler für einen ungeübten Anwendungsprogrammierer als zu hoch einzuschätzen.

Während der Arbeit hat sich gezeigt, daß oft sich wiederholende Tätigkeiten auftreten, wie beispielsweise die Untersuchung der Belegung von Caches um eine optimale Nutzung abzuschätzen. Hierbei wären verschiedene Tools denkbar, die die Cachenutzung während des Programmablaufs anzeigen und möglicherweise sogar zusammenhängende Datenstrukturen unterschiedlich kennzeichnen. Eine selbständige

Erkennung von Kollisionen unterschiedlicher Datenstrukturen im Cache, die eine hohe Rate an *conflict misses* zur Folge haben, wäre oft sehr hilfreich gewesen. So mußte dies mit Zettel und Papier geschehen. Dadurch ließe sich auch die Korrektheit von Code beispielsweise bezüglich *Prefetching* schneller abschätzen und würde nicht stundenlanges Debuggen nötig machen.

## 6.4 Ausblick

Verschiedene Konzepte, die möglicherweise zukünftig bei der Optimierung von Mehrgitterverfahren eine wichtige Rolle spielen, werden im Folgenden noch kurz angesprochen und das zugrundeliegende Prinzip erklärt.

### 6.4.1 Datenverschiebung im Cache

Oft ist der Einsatz von *padding* unerläßlich, um *conflict misses* zu vermeiden. Beispielsweise kommt es bei der Matrixmultiplikation vor, daß sich die, zur Berechnung benötigten Daten, gegenseitig verdrängen.

Eine interessante Alternative zum Einfüllen zusätzlicher Elemente bietet das Verschieben dieser Elemente im Cache[1] [Bey04]. Hierzu wird der Cache in drei Blöcke aufgeteilt. Zwei gleichgroße Blöcke $A_1$ und $A_2$, dienen zur Aufnahme der Feldelemente und ein kleinerer Block $B$ enthält skalare Variablen, die nicht in ein Register passen. Nun kommen zwei Threads zum Einsatz, von denen einer Daten aus dem Speicher lädt und in einen der beiden Blöcke, beispielsweise $A_1$ schreibt, während der andere die Daten aus dem zweiten Block $A_2$ verarbeitet. Ist die Verarbeitung von Block $A_2$ abgeschlossen, so werden die Blöcke vertauscht.

Um dieses Verfahren einzusetzen, müssen drei Bedingungen erfüllt sein:

- Es muß die Möglichkeit bestehen, Daten am Cache vorbei zu laden. Andernfalls würden die zu Ladenden Daten, den Inhalt des anderen Blockes überschreiben.
- Der Prozessor muß mehrere Instruktionen parallel verarbeiten können.

[1] engl.: cache remapping

- Der Prozessor darf bei einem Fehlzugriff auf den Cache nicht warten, bis die Daten geladen werden, solange davon unabhängige Befehle zur Bearbeitung anstehen[1].

Alle EPIC-Architekturen bieten diese Anforderungen [Bey04], so auch der Itanium 2.

### 6.4.2 Ausführen außerhalb der Reihenfolge

Beim Itanium 2 legt der Compiler die Reihenfolge der Abarbeitung von Befehlen fest, was die Komplexität des Prozessors verringert und die Skalierbarkeit erhöht. Der Prozessor hat selbst keine Möglichkeit, die Reihenfolge der Befehle umzustellen[2], sondern muß diese in der angegebenen Reihenfolge[3] abarbeiten. Eine kleine Ausnahme stellt hier die Möglichkeit des spekulativen Ladens dar, was jedoch explizit eingesetzt werden muß und somit nicht für jede Instruktion in Frage kommt.

Die Ausführung von Befehlen in der gegebenen Reihenfolge hat große Auswirkungen auf die Anzahl der zu bearbeitenden Instruktionen pro Zyklus[4]. Muß eine Instruktion auf ein Register warten, welches gerade von einer in Bearbeitung befindlichen Anweisung gefüllt wird, so müssen auch alle nachfolgenden Instruktionen warten. In [CCC02] wird nun beschrieben, wie durch Verwendung der funktionalen Einheiten des Itanium-Prozessors als abwechselnd zu beschickende funktionale Einheiten[5] eine Ausführung außerhalb der Reihenfolge mit geringem Aufwand zu implementieren wäre. Hintergedanke ist dabei die Idee, die funktionalen Einheiten des Itanium-Prozessors mit Befehlen in der gegebenen Reihe zu beschicken, die diese allerdings unabhängig voneinander ausführen können, sobald die benötigten Daten zur Verfügung stehen. Wird eine PFU frei und es steht ein entsprechender Befehl an, so kann diese neu belegt werden, ohne auf andere funktionale Einheiten zu warten.

[1] Dies kann beim Itanium 2 durch spekulatives Laden erreicht werden, muß jedoch explizit verwendet werden und ist somit auf spezielle Einsatzfälle beschränkt.
[2] engl.: out-of-order execution
[3] engl.: in-order execution
[4] engl.: instructions per cycle, oder kurz IPC
[5] engl.: Pending Functional Units, oder kurz PFU

### 6.4.3 Simultane Nebenläufigkeit

Die simultane Nebenläufigkeit[1] von einzelnen Threads eines Prozesses mithilfe von virtuellen Prozessoren[2] nennt Intel *Hypterthreading*. Hierbei handelt es sich allerdings nicht um echt Integration von mehreren Prozessoren auf einem Die, sondern lediglich um mehrfach vorhandene Kopien gewisser Ressourcen eines Prozessors. Zu unterscheiden sind dabei folgende Ressourcen[Int02b]:

- Nachgebildete Ressourcen[3] sind tatsächlich mehrfach vorhandene Kopien von Ressourcen. Auf denjenigen Xeon- und Pentium 4 Prozessoren, die diese Technologie bereits implementieren, gehören hierzu ein vollständiger Registersatz einschließlich Kellerzeiger[4] und Befehlszeiger[5].
- Unterteilte Ressourcen[6], sind Ressourcen, die unterteilt wurden. Hierzu gehören beispielsweise Warteschlangen, die zwischen den einzelnen Phasen der Befehlsabarbeitung benötigt werden. Interessant ist, daß unterteilte Ressourcen wieder kombiniert werden können, wenn der Prozessor nur einen Thread ausführt.
- Geteilte Ressourcen[7]. Alle anderen Ressourcen des Prozessors werden gemeinsam benutzt. Hierunter fallen beispielsweise die arithmetisch-logische Einheit[8] oder die Einheit zur Berechnung von Fließkommawerten[9]

[1] engl.: Hyperthreading
[2] engl.: siblings
[3] engl.: replicated resources
[4] engl.: stack pointer
[5] engl.: instruction pointer
[6] engl.: partitioned resources
[7] engl.: shared resources
[8] engl.: arithmetical logical unit, oder kurz ALU
[9] engl.: floating point unit, oder kurz FPU

# Anhang A

# Kuriositäten

Einige Kuriositäten, die während der Bearbeitung aufgetreten sind, sollen hier noch kurz aufgezeigt werden:

- Bei der Verwendung des Intel C/C++ Compiler auf den Xeon-Rechnern kam es mitunter zu defekten Code, wenn die Optimierungsstufe *-O3* ohne Angabe der Architektur mittels *-march=pentium4* verwendet wurde. Dabei kam es vor, daß nur die äußere von verschachtelten Schleifen durchlaufen wurde.

- Beim Einsatz des Pragmas *#pragma unroll(4)* kam es beim Intel C/C++ Compiler auf dem Xeon vor, daß dieser eine Schleife, die durch Angabe einer Konstanten acht mal durchlaufen werden sollte, zwar ausrollte, durch dieses Konstrukt allerdings nur einmal durchlief und die restlichen vier Schleifendurchläufe in einer separaten Schleife wieder einzeln durchführte.

- Bei der Messung der Bandbreite auf dem Itanium 2, durch überfüllen eines einzelnen Sets des L2-Cache, gelang es nicht, dieses so oft zu füllen, daß sich die Fehlzugriffe diesbezüglich häuften, was eine Messung mithilfe von *Pfmon* zeigte. Deshalb wurde hier auf die Methode ausgewichen, den gesamten Cache zu überfüllen.

- Der auf dem Itanium 2 vom Intel C/C++ Compiler erzeugte Code für das Programmbeispiel zum Vorausladen von Daten enthielt mehrere Konstanten über die zu durchlaufenden Schleifen. Wie Tabelle 3.7 zeigt, wurde der Code durch einfügen unsinniger Befehle aus Listing 3.21 schneller, obwohl diese niemals durchlaufen werden konnten, da die Bedingung niemals wahr wird. Der Compi-

ler konnte dadurch allerdings nicht mehr feststellen, ob die angegebenen „Konstanten“ möglicherweise während des Programmablaufs verändert werden.

# Anhang B

# Verwendete Rechnerausstattung

## B.1 CISC-Architektur

Als Vertreter der CISC-Architektur kam ein Xeon-Prozessor der Firma Intel zum Einsatz. Tabelle B.1 gibt einen Überblick der Hardwareausstattung [Mai04].

Als Betriebssystem läuft „Red Hat Linux release 7.3 (Valhalla)“.

| | |
|---|---|
| Prozessor | Intel Xeon DP 2,4 GHz |
| Anzahl der Prozessoren | 2 |
| Größe des L1-Cache | 8 KByte |
| Größe einer *Cache Line* | 64 Byte |
| Größe des L2-Cache | 512 KByte |
| Größe einer *Cache Line* | 64 Byte |
| Besonderheit | 128 Byte Sektor size |
| Breite des Systembuses | 64 Bit |
| Taktrate | 400 MHz |
| Hauptspeicher | 4 GByte |
| Breite des Speicherbuses | 128 Bit |
| Taktrate | 200 MHz |
| Chipsatz | Intel ET7500 |

Tabelle B.1: Verwendeter Xeon-Rechner

## B.2 EPIC-Architektur

Ein Itanium 2 der Firma Intel kam als Vertreter der EPIC-Architektur zum Einsatz. Die technischen Daten sind in Tabelle B.2 aufgelistet [Mai04].

Als Betriebssystem läuft „Red Hat Linux Advanced Server release 2.1AS (Derry)“.

| | |
|---|---|
| Prozessor | Intel Itanium 2 1,3 GHz |
| Anzahl der Prozessoren | 4 |
| Größe des L1-Cache | 16 KByte |
| Größe einer *Cache Line* | 64 Byte |
| Größe des L2-Cache | 256 KByte |
| Größe einer *Cache Line* | 128 Byte |
| Größe des L3-Cache | 3 MByte |
| Größe einer *Cache Line* | 128 Byte |
| Breite des Systembuses | 128 Bit |
| Taktrate | 400 MHz |
| Hauptspeicher | 8 GByte |
| Breite des Speicherbuses | 256 Bit |
| Taktrate | 200 MHz |
| Chipsatz | Intel 8870 |

Tabelle B.2: Verwendeter Itanium-Rechner

# Anhang C

# Übersetzung der Programme

In diesem Kapitel sind die einzelnen Listings aufgeführt. Dazu ist, soweit möglich, jeweils angegeben, mit welchem Compiler und welchen Compiler-Switches die Übersetzung erfolgt ist und auf welchem Rechner der übersetzte Code zum Einsatz kam. Falls es sich um nicht übersetzbare Beispiele handelt, sind hier weitere Informationen dazu aufgeführt.

- Listing 2.1
  Compiler: gcc simple.c -o simple
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 2.2
  Compiler: gcc complex1.c -o complex1
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 2.3
  Compiler: gcc complex2.c -o complex2
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 2.4
  Nicht übersetzbares Beispiel eines Vorabladebefehls
- Listing 3.1
  Compiler: gcc withoutLoopFusion.c -o withoutLoopFusion
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.2
  Compiler: gcc loopFusion.c -o loopFusion
  Rechner: Infinicluster Technische Universität München, Xeon - Node6

- Listing 3.3
  Compiler: gcc loopUnrolling.c -o loopUnrolling
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.4
  Compiler: gcc withoutBlocking.c -o withoutBlocking
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.5
  Compiler: gcc blocking.c -o blocking
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.6
  Compiler: gcc padding.c -o padding
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.7
  Compiler: siehe Abbildung 3.8 sowie Abbildung 3.9
  Rechner: Infinicluster Technische Universität München, Xeon - Node6 sowie Infinicluster Technische Universität München, Itanium - Node4
- Listing 3.8
  Compiler: icc pragmaLoopUnrolling.c -S withoutPragma.s
  Compiler: icc pragmaLoopUnrolling.c -S pragmaUnroll.s -DPRAGMA
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.9
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 3.8
- Listing 3.10
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 3.8
- Listing 3.11
  Nicht übersetzbares Beispiel für Pragma *#pragma optimize*
- Listing 3.12
  Beispiel für Pragma *#pragma ivdep*
- Listing 3.13
  Beispiel für Pragma *#pragma prefetch*

- Listing 3.14
  Compiler: icc noIntrinsic.c -o noIntrinsic -march=pentium4 -tpp7
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.15
  Compiler: icc intrinsicAddpd.c -o intrinsicAddpd -march=pentium4 -tpp7
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.16
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 3.15
- Listing 3.17
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 3.15
- Listing 3.18
  Dieses Listing wurde zuerst assembliert (1), dann per Hand optimiert und anschließend compiliert (2).
  Compiler (1): icc listingIntrinsicAddpdMemoryReduced.c -o listing.s -S -march=pentium4 -tpp7
  Compiler (2): icc listing.s -o listingIntrinsicAddpdMemoryReduced
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.19
  Compiler: icc prefetchXeon.c -o prefetchXeon -march=pentium4
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 3.20
  Compiler: icc prefetchItanium.c -o prefetchItanium -O3
  Rechner: Infinicluster Technische Universität München, Itanium - Node4
- Listing 3.21
  Hierbei handelt es sich um einen Zusatz für Listing 3.20. Näheres siehe dort.
- Listing 4.1
  Compiler: icc xeonFlops1.c -o xeonFlops1 -march=pentium4 -fast
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.2
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 4.1

- Listing 4.3
  Compiler: icc xeonFlops2.c -o xeonFlops2
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.4
  Compiler: icc xeonL1Read.c -o xeonL1Read
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.5
  Compiler: icc xeonL1Write.c -o xeonL1Write
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.6
  Dieses Listing wurde zuerst assembliert (1), dann per Hand optimiert und anschließend compiliert (2).
  Compiler (1): icc xeonL2Read.c -o xeonL2Read -S
  Compiler (2): icc xeonL2Read.s -o xeonL2Read
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.7
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 4.6
- Listing 4.8
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 4.6
- Listing 4.9
  Compiler: icc xeonL2Write.c -o xeonL2Write
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.10
  Compiler: icc xeonMemRead.c -o xeonMemRead
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.11
  Compiler: icc xeonMemWrite.c -o xeonMemWrite
  Rechner: Infinicluster Technische Universität München, Xeon - Node6
- Listing 4.12
  Compiler: icc itaniumFlops.c -o itaniumFlops -fast
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 4.13
  Hierbei handelt es sich um eine Compilerausgabe. Näheres zur Generierung siehe Beschreibung zu Listing 4.12

- Listing 4.14
  Compiler: gcc itaniumL1Read.c -o itaniumL1Read
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 4.15
  Compiler: gcc itaniumL2Read.c -o itaniumL2Read
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 4.16
  Compiler: gcc itaniumL2Write.c -o itaniumL2Write
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 4.17
  Compiler: gcc itaniumL3Read.c -o itaniumL3Read
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 4.18
  Compiler: gcc itaniumL3Write.c -o itaniumL3Write
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 4.19
  Compiler: gcc itaniumMemRead.c -o itaniumMemRead
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 4.20
  Compiler: gcc itaniumMemWrite.c -o itaniumMemWrite
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

- Listing 5.1
  Hierbei handelt es sich um ein Perl-Skript
  Rechner: Infinicluster Technische Universität München, Xeon - Node6 sowie Infinicluster Technische Universität München, Itanium - Node4

- Listing 5.2
  Hier kamen verschiedene Compileroptionen zum Einsatz, näheres siehe Tabelle 5.1
  Rechner: Infinicluster Technische Universität München, Xeon - Node6

- Listing 5.3
  Hierbei handelt es sich um eine Header-Datei für Listing 5.2 und Listing 5.5
- Listing 5.4
  Hierbei handelt es sich um eine Header-Datei für Listing 5.2 und Listing 5.5
- Listing 5.5
  Hier kamen verschiedene Compileroptionen zum Einsatz, näheres siehe Tabelle 5.2
  Rechner: Infinicluster Technische Universität München, Itanium - Node4

# Anhang D

# Literaturverzeichnis


[Bey04] K. Beyls, Software Methods to Improve Data Locality and Cache Behavior, Gent Juni 2004

[CCC02] L. Carter, W. Chuang, B. Calder, An EPIC Processor with Pending Functional Units, 4th International Symposium on High Performance Computing (ISHPC), Springer-Verlag, May 2002

[Int02a] Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture, Revision 2.1, Document Number 245317-004, Intel Corporation, Denver October 2002

[Int02b] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, M. Upton, Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History, Intel Technology Journal, `http://developer.intel.com/technology/itj/2002/volume06issue01/`, Februar 2002

[Int03a] IA-32 Intel® Xeon™ Processor with 512-KB L2 Cache at 1.80 GHz to 3 GHz, Datasheet, Order Number 298642-006, Intel Corporation, Denver March 2003

[Int03b] IA-32 Intel® Architecture Optimization, Reference Manual, Order Number 248966-009, Intel Corporation, Denver March 2003

[Int03c] Intel® C++ Compiler for Linux* Sytems User's Guide; Document number 253254-014, Intel Corporation, Denver 2003

[Int04a] IA-32 Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture, Order Number 253665, Intel Corporation, Denver 2004

[Int04b] IA-32 Intel® Architecture Software Developer's Manual, Volume 2a: Instruction Set Reference A-M, Order Number 253666, Intel Corporation, Denver 2004

[Int04c] IA-32 Intel® Architecture Software Developer's Manual, Volume 2a: Instruction Set Reference N-Z, Order Number 253667, Intel Corporation, Denver 2004

[Int04d] IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Proramming Guide, Order Number 253668, Intel Corporation, Denver 2004

[Int04e] Intel® Itanium® 2 Processor Reference Manual, For Software Development and Optimization, Order Number 251110-003, Intel Corporation, May 2004

[JDR] P. Jain, S. Devadas, L. Rudolph, Controlling Cache Pollution in Prefetching With Software-assisted Cache Replacement, Laboratory for Computer Science Massachusetts Institute of Technology, Cambridge, MA 02139

[Mat04] H. G. Matties, Praktikum Wissenschaftliches Rechnen, Mehrgitterverfahren, Technische Universität Braunschweig, Braunschweig April 2004

[Mai04] M. Mairandres, Hardware of the Infiniband Cluster, `http://wwwbode.cs.tum.edu/Par/arch/infiniband/ClusterHW/cluster.html`, München 26. April 2004

[Mes00] H.-P. Messmer, PC Hardwarebuch, Aufbau Funktionsweise Programmierung, Ein Handbuch nicht nur für Profis, 6. Auflage, Addison-Wesley, München 2000

[MNS03] C. McNairy, D. Soltis, Itanium 2 Processor Microarchitecture, 0272-1732/03, IEEE Computer Society, March-April 2003

[Roh01] J. Rohde, Assembler GE-PACKT, 1. Auflage, mitp-Verlag, Bonn 2001

[Sch04] P. Schnabel, Das Elektronik-Kompendium, `http://www.elektronik-kompendium.de`, Ludwigsburg 2004

[SHR97] H. R. Schwarz, Numerische Mathematik, 4., überarbeitete und erweiterte Auflage, B.G. Teubner, Stuttgart 1997

[Sed95] R. Sedgewick , Algorithmen, 2. korrigierter Nachdruck, Addison-Wesley, Bonn, München 1995

[Tan02] A. S. Tanenbaum, Moderne Betriebssysteme, 2., überareitete Auflage, Pearson Studium, München 2002

[Vet00] S. Vetter, Aufruf der Fortran Compiler und Bibliotheken, `http://www.urz.uni-heidelberg.de/UnixCluster/Hinweise/Hilfe/Anwendung/Compiler/fortlibs.html`, Universitätsrechenzentrum Heidelberg, März 2000

[Wei01] Ch. Weiß, Data Locality optimizations for Multigrid Methods on Structured Grids, Thesis, `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2001/weiss.pdf`, Technische Universität München, 2001-09-26

[WL04] Ch. Wiegand, A. Laucher, tecCHANNEL-Compact ITechnologie-Ratgeber 2004, IDG Interactive GmbH, München 2004