Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen
Seminar „Effiziente Programmierung in C“
Betreuer: Nathanael Hübbe

CPU-Caches

Christian Duße

Studiengang: B.Sc. Software-System-Entwicklung
Matrikelnummer.: 634 39 14

Hamburg, den 31.03.2014

Inhaltsverzeichnis

1 Vorwort

Die vorliegende Arbeit beschäftigt sich mit der Fragestellung, inwiefern die Nutzung von (Daten-) CPU-Caches optimiert werden kann, um ein performanteres Programm zu erhalten. Mit Performanz ist hier die Laufzeit des Programms gemeint.

Die Arbeit beginnt mit einer Einleitung, die die Funktionsweise von CPU-Caches beschreibt, Begriffe klärt und die historische Entwicklung beschreibt. Es folgt der Hauptteil, in dem semantisch-identische, aber syntaktisch-unterschiedliche Programmbeispiele gegenübergestellt werden, wobei jeweils das erste Beispiel nicht Cache-optimiert ist, das zweite jedoch schon. Es wird beschrieben, inwieweit sich die Cache-Nutzung unterscheidet und wie groß der Performance-Unterschied ist. Es folgt das Kapitel Caching durch Entwickler, in dem das automatische Caching mit dem expliziten Caching durch den Entwickler verglichen wird. Im letzten Kapitel werden Werkzeuge aufgezeigt, die den Entwickler bei der Cache-Optimierung helfen können.

Diese Arbeit gibt keine Empfehlung ab, welche Optimierungsmaßnahmen in einem Software-Projekt einzusetzen sind. Es werden jedoch Entwicklungsphasen und die dazugehörigen Aufwände abgeschätzt, in diesen Phasen die Optimierung aus den genannten Beispielen umzusetzen.

Die Programmbeispiele wurden auf dem Cluster des Arbeitsbereichs mit der GNU Compiler Collection[1] kompiliert und ausgeführt. Es wurde außerdem die Werkzeugsammlung Valgrind[2] - insbesondere mit Cachegrind[3] – auf dem Cluster genutzt.

1 http://gcc.gnu.org/, 21.03.2014
2 http://valgrind.org/, 21.03.2014
3 http://valgrind.org/docs/manual/cg-manual.html, 21.03.2014

2 Einleitung

2.1 Caches allgemein

Ein Cache ist ein „schneller Pufferspeicher, der (…) Zugriffe auf ein langsames Hintergrundmedium (…) zu vermeiden hilft“[4]. Dies geschieht, indem das Objekt beim ersten Zugriff in den schnellen Pufferspeicher gelegt wird. Bei einem weiteren Zugriff auf das Objekt wird nun zunächst der Cache durchsucht. Wird das gesuchte Objekt dort gefunden, hat ein „Cache Hit“ stattgefunden und das Objekt kann zurückgeliefert werden, ohne vom langsamen Hintergrundmedium erneut geladen oder berechnet zu werden. Wird das gesuchte Objekt nicht im Cache gefunden – zum Beispiel, da es sich um den ersten Zugriff auf das Objekt handelt – handelt es sich um einen „Cache Miss“ und das Objekt muss vom Hintergrundmedium geladen werden.
Aufgrund der Strategie, dass zunächst der Cache durchsucht wird, ergibt sich für Cache Misses eine leicht erhöhte Zugriffszeit. Diese kann aber durch gängige Suchoptimierungen wie Adressierung oder Hashing stark verringert werden.
Ein weiteres Problem bei der Nutzung von Caches ist die Synchronisation vom originalen Objekt des Hintergrundmediums mit der Kopie im Cache. Falls bereits eine Kopie von einem Objekt im Cache liegt und sich das Original verändert, soll die veraltete Kopie durch eine Neue ersetzt werden. Hashing kann hier die Überprüfung auf Änderung des originalen Objekts optimieren. Ein anderes Synchronisationskonzept, welches bei CPU-Caches eingesetzt wird, wird gegen Ende folgenden Kapitels „Funktionsweise von CPU-Caches“ behandelt behandelt.
Caching lohnt sich also bei stark unterschiedlichen Latenz-Zeiten verschiedener Medien bei mehrmaligem Zugriff auf gleiche – möglichst unveränderliche – Objekte.
Caches werden beispielsweise von Internet-Browsern eingesetzt, um Grafiken, die Bestandteil von Webseiten sind und mehrmals aufgerufen werden, zwischenzuspeichern. Die Strategie, wann eine Grafik oder andere Inhalte im Browser-Cache landen, kann je nach Browser unterschiedlich sein.

4 http://de.wikipedia.org/wiki/Cache, 21.03.2014

2.2 Funktionsweise von CPU-Caches

Ein CPU-Cache ist ein Speicher zwischen dem Hauptspeicher (RAM) und dem Prozessor (CPU). Der Hauptspeicher ist hier also das langsame Hintergrundmedium.

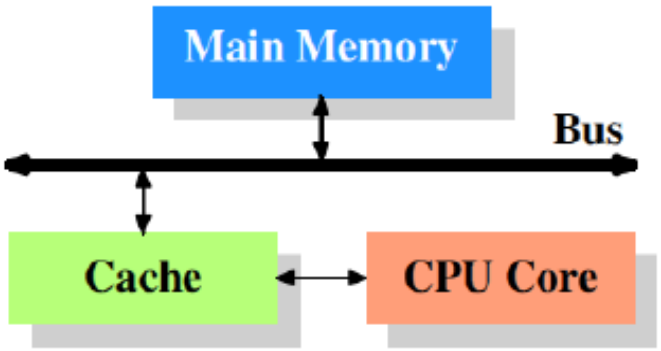

Abbildung 1: Minimum Cache Configuration, Drepper (2007), S. 14.

Der CPU-Cache verringert die Zahl an Anfragen der CPU an den RAM dadurch, dass er Zeilen des Hauptspeichers – die sogenannten „Cachelines“ - hinterlegt. Eine solche Cacheline hat heutzutage eine Größe von 64 Byte und wird zusammen mit der Angabe, welchen Adressraum sie abbildet[5] im Cache gespeichert.

Bei einer Lese-Anfrage durch den Prozessor wird zunächst im CPU-Cache eine Cacheline gesucht, die die gesuchte Adresse enthält und bei einem Cache-Hit der Wert hinter dieser Adresse aus dem Cache zurück gegeben. Bei einem Cache-Miss wird der Speicherbereich über der Zieladresse zusammen mit dem umliegenden Speicher als Cacheline im CPU-Cache gespeichert und an den Prozessor zurückgegeben[6].

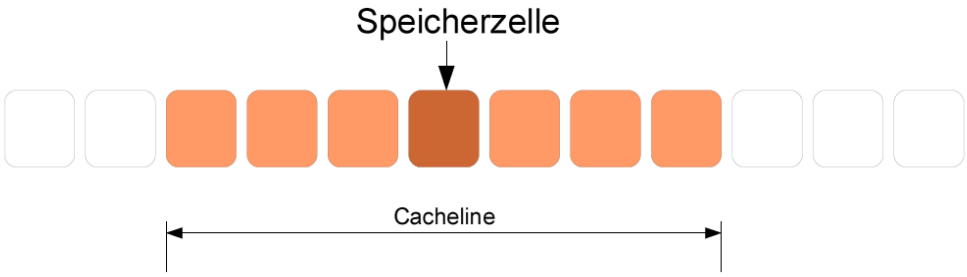

Abbildung 2: Eine Cacheline über einer Speicherzelle

5 Dies geschieht in der Regel dadurch, dass die Adresse der ersten abgedeckten Speicherzelle gespeichert wird. Die anderen Adressen erhält man über Differenzbildung oder Verschiebung.

6 Bei einer Schreib-Operation auf einer Adresse wird zunächst in den Cache geschrieben, welcher dann den Wert in den Hauptspeicher zurückschreibt. Dies kann parallel zu anderen Prozessor-Instruktionen erfolgen.

Im Gegensatz zum klassischem Cache wird also nicht nur das Zielobjekt gecached, sondern auch die „anliegenden“ Speicherzellen. Zwar kommt es daher vor, dass Daten aus dem RAM zwar teuer geladen werden und trotzdem nicht verwendet werden. Jedoch gehen Prozessorhersteller davon aus, dass Daten, die im Speicher physikalisch „nahe beieinander“ liegen, mit geringen zeitlichen Abstand ebenfalls genutzt werden, da sie semantisch zusammen gehören („Datenlokalität“). Da bei heutiger Hauptspeichertechnologie ein mehrmaliger serieller Zugriff sehr viel langsamer ist als ein einmaliger paralleler Zugriff, ist Zeitverlust bei ungenutztem Restinhalt einer Cacheline nicht so schwerwiegend. Die Nutzung von Datenlokalität ist dagegen ein viel größerer Vorteil.
CPU-Caches wurden zum ersten Mal in den neunziger Jahren eingebaut, als die Geschwindigkeit von Prozessoren stiegen, sich aber aufgrund der langsamer steigenden Geschwindigkeit von Hauptspeichern nicht bemerkbar machten. Schnellere Speicher in der Größenordnung von RAMs sind auch heute nicht wirtschaftlich und können nur mit kleinerer Kapazität – zum Beispiel als CPU-Cache – eingesetzt werden. Die weitere Performancesteigerung von CPUs führte dazu, dass verschiedene Level von weiteren Caches eingebaut wurden, damit ein Cache-Miss am CPU-nächsten Cache durch ein Cache-Hit eines anderen Caches „aufgefangen“ werden kann. Der Level-1-Cache ist der CPU-nächste Cache mit der geringsten Speicherkapazität und der geringsten Latenz und ist heutzutage bis zu 256kB groß. Der Level-3-Cache ist in heutigen PCs der sogenannte „Last-Level-Cache“ mit der größten Kapazität und höchsten Latenzzeit aller Caches und ist bis zu mehrere MB groß.

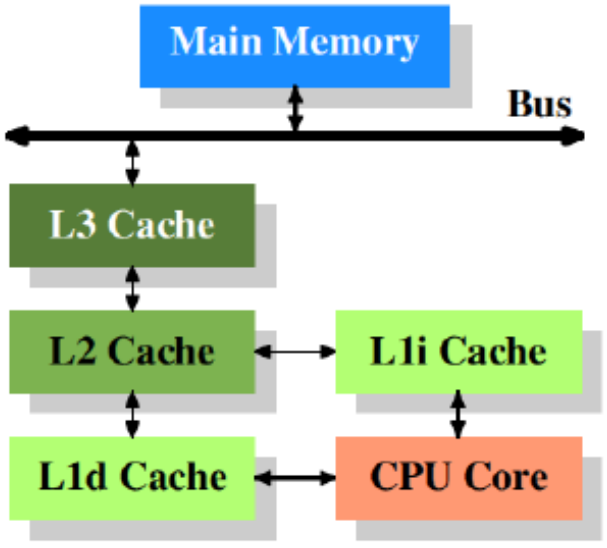

Abbildung 3: Processor with Level 3 Cache, Drepper (2007), S. 15.

Bewährt hat sich auch die „Harvard architecture“, also die physikalische Trennung von Daten und Programmcode. Für die CPU hat dies den Vorteil, dass gleichzeitig Daten aus dem „Data-Cache“ und Programmcode aus dem „Instruction Cache“ geladen werden können. Die Trennung erlaubt auch die Anwendung verschiedener Cachingstrategien. Dies ist auch sinnvoll, denn Daten verhalten sich häufig anders als Programmcode. Semantisch zusammengehörige Daten liegen meist physikalisch nahe beieinander, während Programmcode gerichtet ist und bei der Ausführung auch unweigerlich große „Sprünge“ vollzieht[7].

Heutige Prozessoren besitzen mehrere Prozessorkerne, die gleichzeitig auf den selben Daten arbeiten können. Daraus ergeben sich Vor- und Nachteile für die Verwendung von kerneigenen Caches und gemeinsame Caches: Auf gemeinsame Caches kann nicht parallel zugegriffen werden und in kerneigenen Caches können die gleichen Daten in den Caches verschiedener Kerne vorkommen. Wird nun von einem dieser Kerne ein mehrfach auftretender Wert verändert, muss eine Synchronisation spätestens dann erfolgen, wenn ein anderer Kern auf der gleichen Adresse arbeitet. Dazu wird die geänderte Cacheline als „dirty“ markiert und die anderen Kerne informiert[8]. Heutzutage haben sich kerneigene Level-1-Caches zusammen mit gemeinsamen Last-Level-Caches durchgesetzt.

Memory System Level	Relative Latency
L1 Cache	1x
Higher Cache Levels	10x
Main Memory	100x

Abbildung 4: Heutige Zugriffszeiten verschiedener Speicher, Rogue Wave Software (2011), S. 4.

Für die Verständlichkeit der folgenden Beispiele gehen wir von einem Ein-Kern-Prozessor mit nur einem einzigen Cache-Level aus.

7 Sprünge im Programmfluss entstehen zum Beispiel bei Funktionsaufrufen, bedingten Anweisungen oder GOTOs.

8 Eine solche Synchronisation ist ineffizient und sollte wenn möglich vermieden werden.

3 Optimierungsmaßnahmen

Wie bereits festgestellt wurde, wird das Caching von der CPU selbst vorgenommen. Dies ist auch sinnvoll, damit nicht sämtliche Programme bei Einführung anderer Cache-Architekturen umgeschrieben werden müssen. Neben der Möglichkeit, das Caching als Entwickler durch Prozessor-Instruktionen selbst durchzuführen[9], kann der Entwickler vor Allem den Inhalt des CPU-Caches optimieren um damit die Anzahl an Hauptspeicheraufrufen zu verringern.

3.1 Redundanz

Redundante Daten sind nicht nur in speicherkritischen Anwendungen zu vermeiden, sondern können auch die Ausführungszeit erhöhen, selbst wenn nicht auf ihnen gearbeitet wird. Dies zeigt das folgende Beispiel:

```
struct data
{
    int a;
    int b;
    int c;
    int d;
};

void foo(data myData[AMOUNT])
{
    for (long i = 0; i < AMOUNT; i++)
    {
        myData[i].a = myData[i].b;
    }
}
```

Abbildung 5: Die Funktion "foo" iteriert über ein Array der Struktur "data". Die Größe des Arrays ist als Konstante definiert. Die Komponenten "c" und "d" der Struktur werden nicht verwendet. Nach Rogue Wave Software (2011), S. 5.

Für eine Array-Größe von 100.000.000 ergibt sich eine Laufzeit von 1,75 s bei Verwendung von O0 und 1,5 s mit O2.

Durch Entfernung der ungenutzten Komponenten erhält man für die gleiche Anzahl an Elementen eine Laufzeit von 1,1 s (O0), beziehungsweise 0,8 s (O2). Diesen Sachverhalt kann man anhand der folgenden Grafiken erklären:

Abbildung 6: Eine 64-Byte-Cacheline, die vier Strukturen des Typs "data" enthält. Jedes Kästchen entspricht einem 32-bit-Integer. Rogue Wave Software (2011), S. 6.

9 Siehe dazu das Kapitel „Caching durch Entwickler“

Abbildung 7: Eine 64-Byte-Cacheline, die acht Strukturen des Typs "data" enthält, deren ungenutzte Komponenten entfernt wurden. Rogue Wave Software (2011), S. 6.

Bei Entfernung der ungenutzten Komponenten „c“ und „d“ der Struktur „data“ enthält eine Cacheline doppelt so viele Objekte der Struktur. Somit wird das Programm mit weniger, teuren Arbeitsspeicheraufrufen ausgeführt, was sich in einer stark verringerten Laufzeit bemerkbar macht.

Solche Redundanzen können bereits in der Planung oder beim erstmaligen Schreiben des Codes auffallen und leicht behoben werden. Eine Optimierung kann jedoch auch vorgenommen wurde, falls eine Struktur identifiziert wurde, deren Inhalte nicht redundant sind, aber selten oder gar nicht zusammen genutzt werden. So kann es sein, dass „c“ und „d“ nur in einem ganz anderen Kontext benutzt werden, aber trotzdem semantisch aus objektorientierter Sicht zusammengehören[10]. In einem solchen Fall muss abgewogen werden, ob eine solche Struktur zugunsten verbesserter Performance aufgeteilt wird. Dies ist jedoch eine softwaretechnische Entscheidung und wird hier nicht näher betrachtet.

10 In dem anderen Kontext kann es ja genau anders herum sein. Nämlich, dass ausschließlich die Komponenten „c“ und „d“ verwendet werden, während „a“ und „b“ ungenutzt bleiben.

3.2 Alignment

```
struct data
{
    char a;
    int b;
    char c;
};

void foo(data myData[AMOUNT])
{
    for (long i = 0; i < AMOUNT; i++)
    {
        myData[i].a++;
    }
}
```

Abbildung 8: Die Funktion "foo" iteriert über ein Array der Struktur "data". Die Größe des Arrays ist als Konstante definiert. Die Komponenten der Struktur sind nicht effizient ausgerichtet. Nach Rogue Wave Software (2011), S. 7.

Wie im vorangegangenen Beispiel beschrieben, kann eine ineffiziente Speichernutzung zu einem langsameren Programm führen. Dies ist vor allem bei Strukturen der Fall, da ihre Komponenten hintereinander im Speicher abgelegt werden. Daher führt auch eine erhöhte Speicherbelegung durch schlechtes Alignment zu Performanzeinbußen, wie Abbildung 8 zeigt.
Für ein Array der Größe 100.000.000 ergab sich eine Laufzeit von 1,35 s für O0 und 0,62 s für O2. Vertauscht man die Reihenfolge der Komponenten „b“ und „c“ erhält man eine Laufzeit von 1 s (O0), beziehungsweise 0,45 s (O2). Dies liegt ebenfalls an einer unnötigen Füllung von Cachelines bei Verwendung des Codebeispiels aus Abbildung 8.
Statische Code-Analyse-Werkzeuge wie Clang[11] können solche Schwachstellen auch in größeren Projekten identifizieren

3.3 Datenlokalität

Wenn zusammen genutzte Daten redundanzfrei und effizient ausgerichtet im Speicher liegen und bearbeitet werden, bedeutet dies noch nicht, das der Cache effizient genutzt wird. Dass auch die Reihenfolge, in der auf diese Daten zugegriffen wird, wichtig ist, zeigt das folgende Beispiel:

11 http://clang.llvm.org/, 21.03.2014

```
void foo(char p[ROW_SIZE*AMOUNT_ROWS])
{
    for (long x=0; x<ROW_SIZE; x++)
    {
        for (long y=0; y<AMOUNT_ROWS; y++)
        {
            p[x + y * ROW_SIZE]++;
        }
    }
}
```

Abbildung 10: Die Funktion greift auf ein Array von Chars mit konstanter Größe elementweise zu. Nach Rogue Wave Software (2011), S. 8.

Man kann sich das übergebene eindimensionale Array auch als zweidimensionales Array vorstellen. Dann würde man spaltenweise auf die Elemente der Matrix zugreifen. Betrachtet man es als eindimensionales Array, dann wird für ROW_SIZE=4 und AMOUNT_ROWS=6 in folgender Reihenfolge auf die Elemente des Arrays zugegriffen: 0, 4, 8, 12, 16, 20, 1, 5, 9, 13, 17, 21, 2, 6, 10, usw.

Dies ist jedoch ineffizient, da die Cachelines einen kontinuierlichen Bereich des Arrays abdecken, wie Abbildung 11 zeigt. Um über eine Spalte zu iterieren, werden mehrere Cachelines benötigt. Die gleichen Cachelines könnten aber auch bei dem Durchlauf über der nächsten Spalte usw. genutzt werden. Bei einem Arrays mit einer großen Zeilenanzahl können nicht alle bei einem Spaltendurchlauf ermittelten Cachelines im Cache behalten werden und müssen beim Durchlauf der zweiten Spalte teuer neu aus dem Hauptspeicher geladen werden.

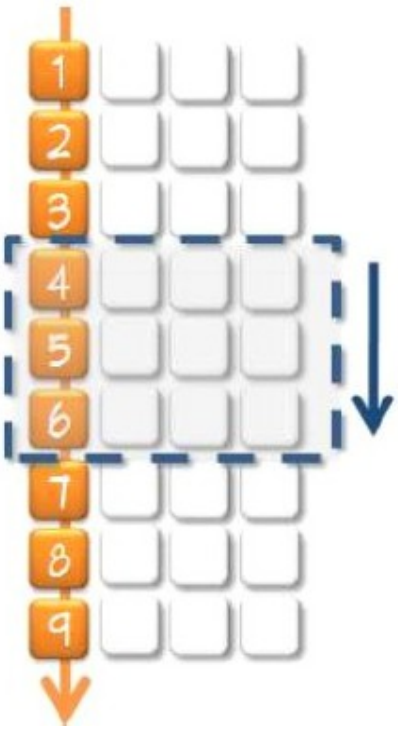

Abbildung 11: Eine Cacheline über einem zweidimensionalem Array, auf das Spaltenweise zugegriffen wird. Die gestrichelte Linie gibt den Bereich des Arrays an, der von einer Cacheline abgedeckt wird. Rogue Wave Software (2011), S. 9.

Für eine Array-Größe von 100.000.000 – dies entspricht 10.000 Zeilen und 10.000 Spalten – ergab sich eine Laufzeit von 3,2 s (O0), beziehungsweise 3,1 s (O2). Greift

man hingegen zeilenweise auf das Array zu[12], ergaben sich Zeiten von 0,4 s (O0) und 0,12 s (O2).

Bei einem zeilenweisen Zugriff wird jede Cacheline nacheinander komplett abgearbeitet und danach nicht mehr benötigt (siehe Abbildung 12). Der Prozessor braucht also weniger Hauptspeicheraufrufe um das gleiche Ergebnis zu erzielen.

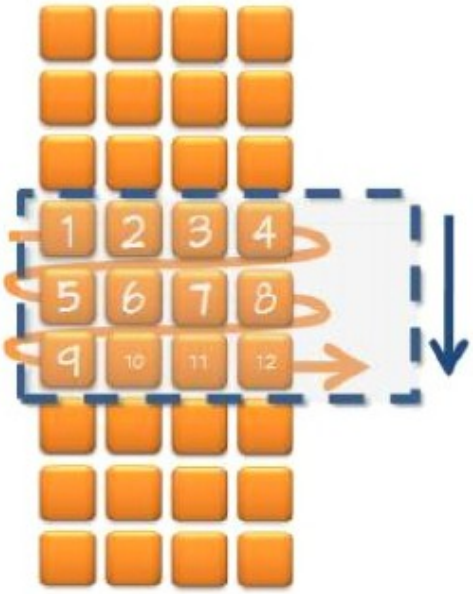

Abbildung 12: Eine Cacheline über einem zweidimensionalem Array, auf das Zeilenweise zugegriffen wird. Die gestrichelte Linie gibt den Bereich des Arrays an, der von einer Cacheline abgedeckt wird. Rogue Wave Software (2011), S. 9.

Wenn auf einem Array gearbeitet wird, sollte dieses also generell zeilenweise geschehen. Falls dies nicht ohne weiteres möglich ist, können trotzdem einige Cache-Optimierungen vorgenommen werden:

Wenn auf ein Array explizit spaltenweise zugegriffen werden muss[13], bietet es sich möglicherweise an, erst eine transponierte Kopie des Arrays anzulegen, auf der dann zeilenweise gearbeitet wird. Oder man zerlegt das (zweidimensionale) Array in kleinere Arrays in Abhängigkeit der Cachelinegröße[14] und/oder der Cache-Größe, damit nicht bereits – dem Hauptspeicher entnommene und in Zukunft zu nutzende – Cachelines aus dem Cache aufgrund von Platzmangel entfernt werden.

Solche Änderungen sind mit einem erheblichen Mehraufwand und größeren Codeänderungen verbunden, weswegen sie nur bei Bedarf durchgeführt werden sollten. Der Merksatz, dass auf Arrays zeilenweise zugegriffen werden soll, sollte allerdings bereits beim erstmaligen Verfassen von Code berücksichtigt werden.

12 Dies erreicht man beispielsweise dadurch, dass man die beiden Schleifen vertauscht.

13 Als Beispiel sei hier die Matritzenmultiplikation genannt, bei der – salopp gesagt – eine Zeile mit einer Spalte multipliziert wird.

14 Die Größe einer Cacheline kann dem Compiler unter Unix beispielsweise über den Befehl „`getconf LEVEL1_DCACHE_LINESIZE`“ als Makrodefinition übergeben und so in das Programm eingesetzt werden.

4 Caching durch Entwickler

Caching wird automatisch von der CPU übernommen. Viele Prozessoren bieten jedoch mittlerweile Schnittstellen für Entwickler an – sogenannte „Instruction Sets“ - mit denen der CPU-Cache explizit manipuliert werden kann. So kann „Data Prefetch“ vorgenommen werden; also das Lagern von Daten im Cache vor ihrem Zugriff. GCC bietet dazu beispielsweise das Interface „`void __builtin_prefetch (const void *addr, ...)`“[15] an, welche die konkreten, hardwareabhängigen Instruktionen (wie SSE oder MMX[16]) kapselt.

Heutige Prozessoren können mittlerweile sehr gut Zugriffsmuster für den Datenzugriff vorhersehen, weswegen sich Data Prefetch sich in vielen Fällen nicht lohnt oder sogar das Programm verlangsamen kann[17]. Nur bei komplexen Zugriffsmustern, wie zum Beispiel bei Sprüngen oder Manipulation von Zählern in Abhängigkeit von erst zur Laufzeit bekannten Datensätzen lohnt sich dieses Vorgehen.[18]

15 http://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Other-Builtins.html, 31.03.2014
16 http://gcc.gnu.org/projects/prefetch.html, 31.03.2014
17 Corbet (2011)
18 Hier sollten eigentlich zwei Beispiele folgen. Ein Beispiel sollte zeigen, dass es bei einem einfachen Zugriffsmuster keine oder nur eine sehr geringe Zeitersparnis gibt. Das andere Beispiel sollte einen verständlichen Fall aufzeigen, bei dem sich Data Prefetch aufgrund eines komplexen Zugriffsmuster lohnt. Leider hatte ich keine Zeit, um meinen Betreuer um Rat zu fragen. Hier gibt es ein Beispiel in C++ http://stackoverflow.com/questions/7327994/prefetching-examples (31.03.2014)

5 Unterstützung für Entwickler

Um Entwickler beim Identifizieren von Speicherlecks und ineffizienter Cache-Nutzung zu unterstützen, gibt es eine Reihe von Werkzeuge, die qualitative Aussagen über das Programm zu Laufzeit geben können. Dazu gehört unter Anderem die Werkzeugsammlung Valgrind.
Valgrind ist eine Art virtuelle Maschine, die Code in die Sprache Vex-IR übersetzt. Dieser übersetzte Code kann dann durch Werkzeuge modifiziert und dann gezielt ausgeführt werden. Die Anwendung des Werkzeugs „Memcheck“ listet beispielsweise die Menge des zur Laufzeit alloziierten Speichers und den nicht-wieder-freigegebenen Speicher auf.

5.1 Cachegrind

Cachegrind ist ein weiteres Werkzeug für Valgrind. Es misst unter Anderem die Anzahl an CPU-Instruktionen, sowie die Anzahl von Zugriffen auf den Instruction Cache und Data-Cache mitsamt der Anzahl an Cache Misses (unterteilt in Level-1-Cache und Last-Level-Cache).
Das zu untersuchende Programm muss dabei nicht einmal als Quelldatei vorliegen, sondern kann bereits kompiliert sein. Bei Angabe der Quelldatei kann jedoch auch eine zeilenweise Auswertung des Programms vorgenommen werden: Es wird also aufgelistet, in welcher Zeile es wie viele Aufrufe mit wie vielen Cache Misses gab.
Da es sich um eine virtuelle Maschine handelt, können in Cachegrind außerdem die Größen der Caches eingestellt werden und damit verschiedene Prozessoren simuliert werden.
Anhand des Beispiels aus Kapitel 3.3 Datenlokalität wird nun exemplarisch die Ausgabe von Cachegrind gedeutet. Wir betrachten den Code aus Abbildung 10.

```
I    refs:      1,782,818,167
D    refs:        734,099,211
D1   misses:      104,860,683
D1   miss rate:           14.2%
```

Abbildung 13: Auszug aus der Cachegrind-Ausgabe für den Code aus Abbildung 10. Das Array hat 100.000.000 Elemente und als Compliler-Optimierung wurde O0 verwendet.

Zur Ausführung des Programms wurden also 1.782.818.167 Prozessor-Instruktionen benötigt und es gab 734.099.211 Datenaufrufe. Dabei gab es bei

104.860.683 Datenaufrufe Cache Misses (14,2 %), bei denen auf den Hauptspeicher zugegriffen werden musste.
Wenden wir auf das gleiche Programm die Compiler-Optimierung O2 an, erhält man folgende Ausgabe:

```
I   refs:      419,659,593
D   refs:      104,912,813
D1  misses:    104,860,690
D1  miss rate:        99.9%
```

Abbildung 14: Auszug aus der Cachegrind-Ausgabe für den Code aus Abbildung 10. Das Array hat 100.000.000 Elemente und als Compliler-Optimierung wurde O2 verwendet.

Durch die Vektorisierung bei der Optimierungsstufe 2 gibt es weniger Instruktionen und weniger Datenaufrufe. Die absolute Zahl an Cache Misses ist jedoch nahezu gleich geblieben. Daher erhält man eine sehr viel höhere Cache-Miss-Rate, obwohl das Programm durch die Optimierung schneller geworden ist. Daher sind solche Zahlen mit Vorsicht zu genießen und sollten nur mit Ergebnissen bei gleicher Optimierungsstufe verglichen werden.
Wenn wir jedoch die Cache-optimierte Variante des Programms verwenden (also die beiden Schleifen vertauschen), erhalten wir für O0 folgende Ausgabe:

```
I   refs:    1,782,818,198
D   refs:      734,099,222
D1  misses:      1,641,484
D1  miss rate:         0.2%
```

Abbildung 15: Auszug aus der Cachegrind-Ausgabe für den Cache-optimierten Code von Abbildung 10. Das Array hat 100.000.000 Elemente und als Compliler-Optimierung wurde O0 verwendet.

Wir stellen fest, dass die Anzahl an Instruktionen und Datenzugriffen im Vergleich zur nicht-Cache-optimierten Version nahezu identisch ist. Jedoch hat sich die Anzahl der Level-1-Cache-Misses auf 0,2 % verringert.
Der Vollständigkeit halber betrachten wir auch die Ausgabe für O2 der Cache-optimierten Variante:

```
I   refs:          419,638,004
D   refs:          104,912,342
D1  misses:          1,641,485
D1  miss rate:             1.5%
```

Abbildung 16: Auszug aus der Cachegrind-Ausgabe für den Cache-optimierten Code von Abbildung 10. Das Array hat 100.000.000 Elemente und als Compliler-Optimierung wurde O2 verwendet.

Hier hat sich im Vergleich mit O0 nur die absolute Zahl an Instruktionen und Datenzugriffen verringert, nicht jedoch die Anzahl an Cache Misses.
Diese qualitativen Ergebnisse zeigen noch einmal, dass Compiler nicht unbedingt die Cache-Nutzung optimieren.

6 Fazit

Beim Vergleich von Zugriffszeiten von RAM und CPU-Cache wird deutlich, dass es sich lohnt, ein Programm bezüglich des Caches zu optimieren. Dies gelingt, indem die Datenmenge, mit der gearbeitet wird, möglichst klein gehalten wird. Außerdem sollten die Daten nahe beieinander liegen und auch so auf sie zugegriffen werden. Zwar kann dies aufgrund von bereitgestelltem virtuellem Speicher nicht immer sichergestellt werden. Doch zusammen alloziierter Speicher liegt in der Regel auch physikalisch nebeneinander.
Die CPU verwaltet den Cache selbstständig und bei heutigen Architekturen auch sehr effizient und vorausschauend. Falls in einem Programm auch einmal komplexe Zugriffsmuster vorherrschen, kann es sinnvoll sein, das Caching selbst vorzunehmen. Dies kann beliebig ausführlich geschehen.
Die ersten Merksätze sollten Entwickler verinnerlichen, damit sie bereits beim erstmaligen Verfassen ihren Code optimieren. Falls dies nicht der Fall ist, können sie Werkzeuge wie Cachegrind benutzen, um ineffiziente Code-Stellen zu identifizieren und zu optimieren. Diese Tools liefern qualitatives Feedback über die Anzahl und die Position teurer Speicheraufrufe im Gegensatz zu subjektiven Zeitmessungen, die je Rechnerauslastung unterschiedlich sein können.

7 Literaturverzeichnis

Ulrich Drepper, „What Every Programmer Should Know About Memory“, Red Hat, Inc., 2007, http://people.freebsd.org/~lstewart/articles/cpumemory.pdf (31.03.2014)

Rogue Wave Software, „CPU Cache Optimization: Does It Matter? Should I Worry? Why?“, 2011, http://www.roguewave.com/DesktopModules/Bring2-mind/DMX/Download.aspx?entryid=1134 (31.03.2014)

Jonathan Corbet, „The Problem with prefetch“, 2011, https://lwn.net/Articles/444336/ (31.03.2014)

http://de.wikipedia.org/wiki/Cache (31.03.2014)

http://en.wikipedia.org/wiki/CPU_cache (31.03.2014)

http://valgrind.org/docs/manual/cg-manual.html (31.03.2014)