CPU-Caches

Christian Duße

Seminar „Effiziente Programmierung in C“

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

30.01.2014

Gliederung

1. Einführung
    1. Definition
    2. Motivation
    3. Funktionsweise
2. Optimierungsmöglichkeiten für Entwickler
    1. Anwendungsbeispiele
    2. Beispiel für Vorgehensweise
3. Unterstützung für Entwickler
    1. Cachegrind

Literatur

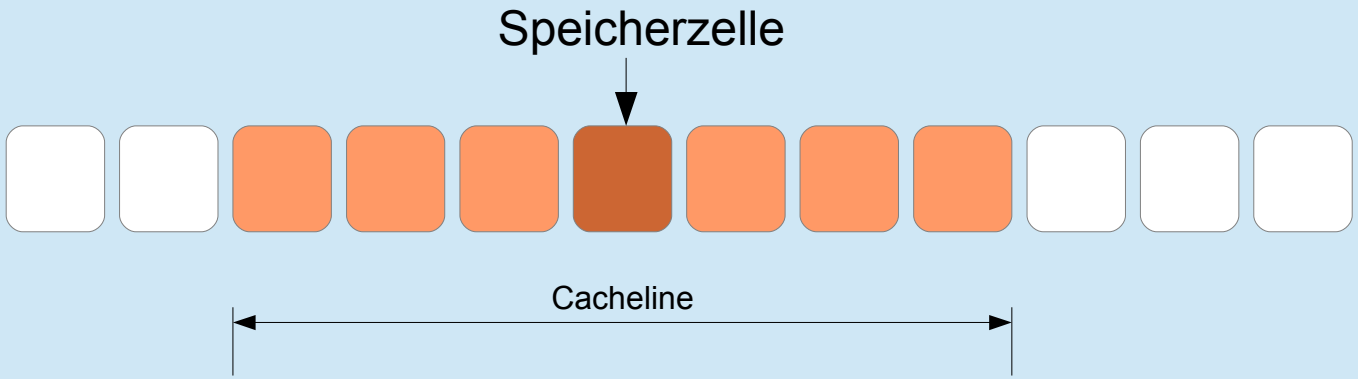
- Ulrich Drepper, „What Every Programmer Should Know About Memory“, 2007, http://people.freebsd.org/~lstewart/articles/cpumemory.pdf (22.01.2014)
- Rogue Wave Software, „CPU Cache Optimization: Does It Matter? Should I Worry? Why?“, 2011, http://www.roguewave.com/DesktopModules/Bring2mind/DMX/Download.aspx?entryid=1134&command=core_download&PortalId=0&TabId=607 (22.01.2014)

Definition Cache

- Schneller Pufferspeicher
- Verwahrt „teure Objekte“ eines Hintergrundmediums
- Bei Anfrage auf das Medium: zunächst Suche im Cache
- „Cache Hit“ und „Cache Miss“
- Beispiele:
  - Grafiken aus dem WWW in temp-Ordner
  - Dateiinhalte in Arbeitsspeicher

CPU-Cache

- Speicher zwischen RAM und CPU-Kern
- Wird von CPU verwaltet
- Hält Zeilen aus RAM bereit („Cacheline“)
- Schreibt Änderungen in RAM zurück



Rückblick (1/2)

- Bis 80-er-Jahre: Frequenz von RAM und CPU etwa gleich schnell
- Anfang 90-er: CPU-Taktrate steigt überproportional zum RAM
- Einführung von kleinem SRAM

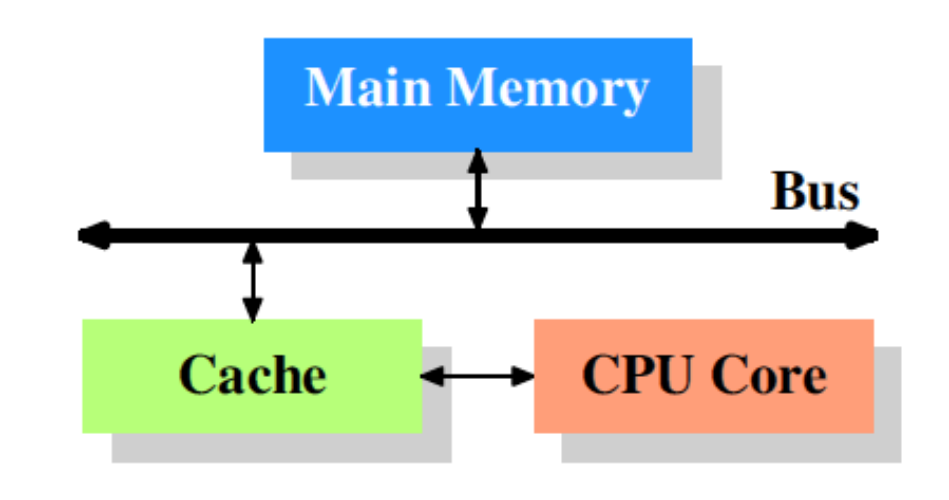


Abb. 2: Minimum Cache Configuration, Drepper (2007), S. 14.

Rückblick (2/2)

- Einführung weiterer Caches
  - Für L1-Cache: Trennung in „Instruction Cache“ und „Data Cache“ (L1i und L1d) → „Harvard architecture“

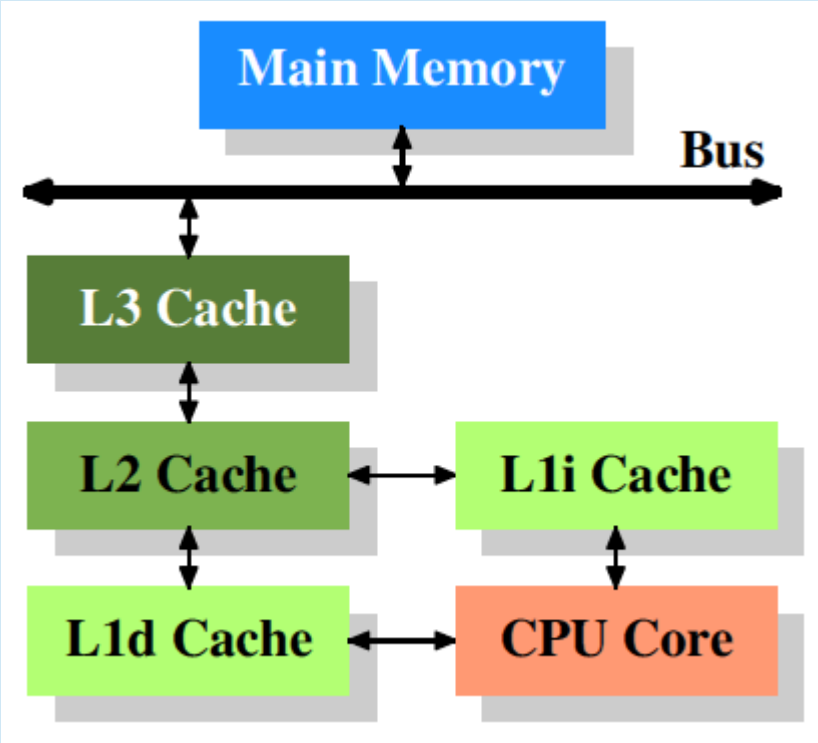


Abb. 3: Processor with Level 3 Cache, Drepper (2007), S. 15.

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

Abb. 4: Zugriffszeiten, Rogue Wave Software (2011), S. 4.

Ausblick: aktuelle Entwicklungen

- Top-20-Prozessoren laut chip.de[1]
  - L1-Cache: 256kB
  - L2-Cache: 4 x 256 kB (Intel) bis 4 x 2048 kB (AMD)
  - L3-Cache: 6 MB (Intel) bis 15 MB (Intel)
- Kerne, die sich Cache teilen vs. kerneigene Caches
  - Synchronisation

[1] http://www.chip.de/bestenlisten/Bestenliste-Desktop-Prozessoren--index/detail/id/693/ (22.01.2014)

Redundanz (1/2)

```
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;
    }
}
```

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

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

Abb. 5: Codebeispiele nach Rogue Wave Software (2011), S. 5.

Für AMOUNT = 100.000.000 → 1,75 sec (O0) bzw. 1,5 sec (O2)

Für AMOUNT = 100.000.000 → 1,1 sec (O0) bzw. 0,8 sec (O2)

Redundanz (2/2)

Abb. 6: Cachelines, Rogue Wave Software (2011), S. 6.

Alignment (1/2)

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

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

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

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

Abb. 7: Codebeispiele nach Rogue Wave Software (2011), S. 7.

Für AMOUNT = 100.000.000 → 1,35 sec (O0) bzw. 0,62 sec (O2)

Für AMOUNT = 100.000.000 → 1,00 sec (O0) bzw. 0,45 sec (O2)

Alignment (2/2)

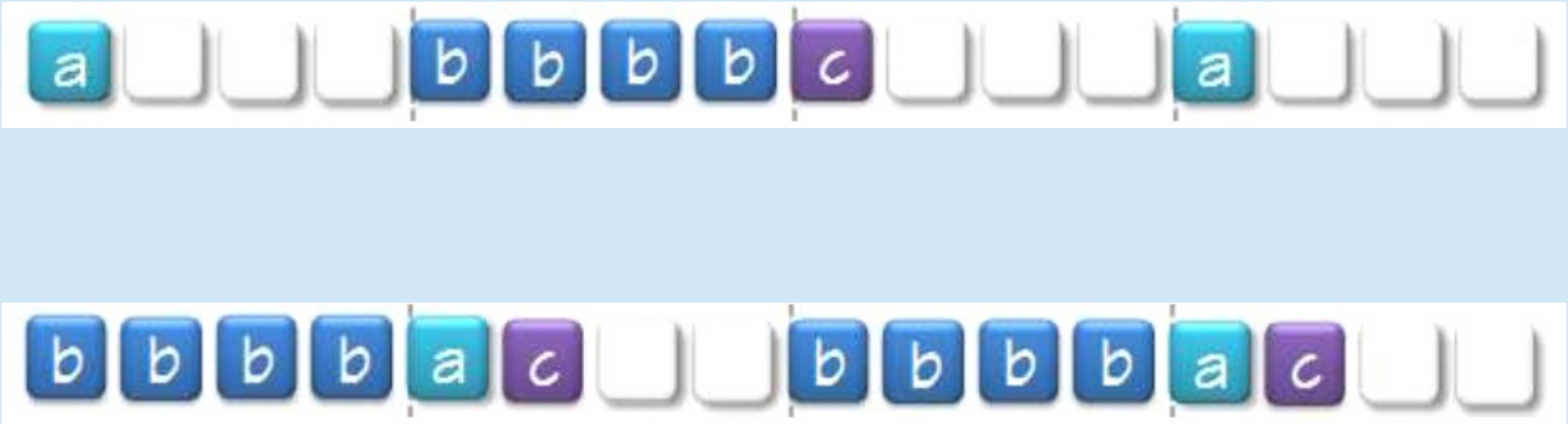


Abb. 8: Cachelines, Rogue Wave Software (2011), S. 7f.

Daten-Lokalität (1/2)

```
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]++;
        }
    }
}
```

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

Abb. 9: Codebeispiele nach Rogue Wave Software (2011), S. 8.

Für ROW_SIZE = AMOUNT_ROWS = 10.000 → 3,2 sec (O0) bzw. 3,1 sec (O2)

Für ROW_SIZE = AMOUNT_ROWS = 10.000 → 0,4 sec (O0) bzw. 0,12 sec (O2)

Daten-Lokalität (2/2)

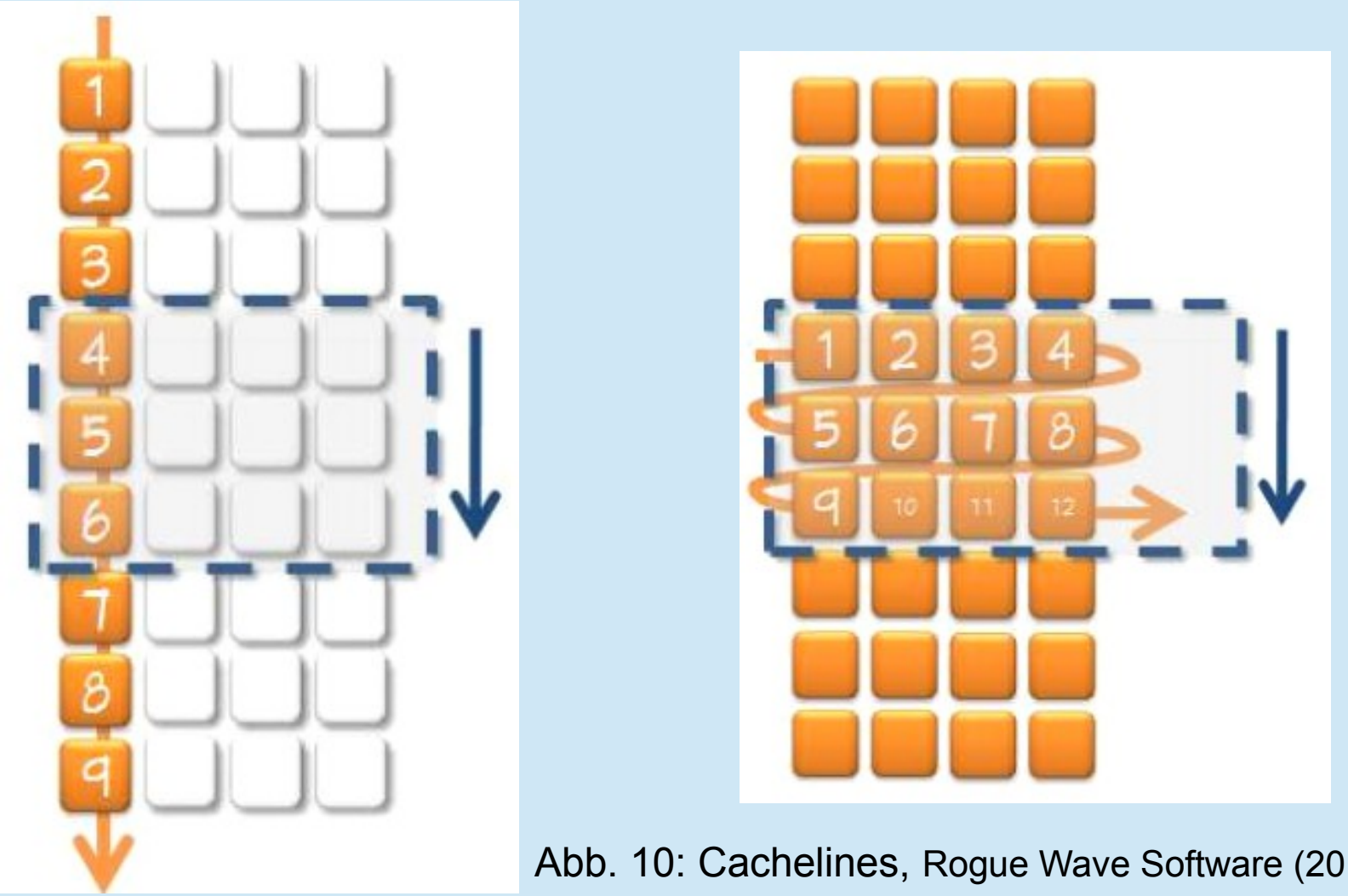


Abb. 10: Cachelines, Rogue Wave Software (2011), S. 9.

Cache-Optimierung für Algorithmen (1/3)

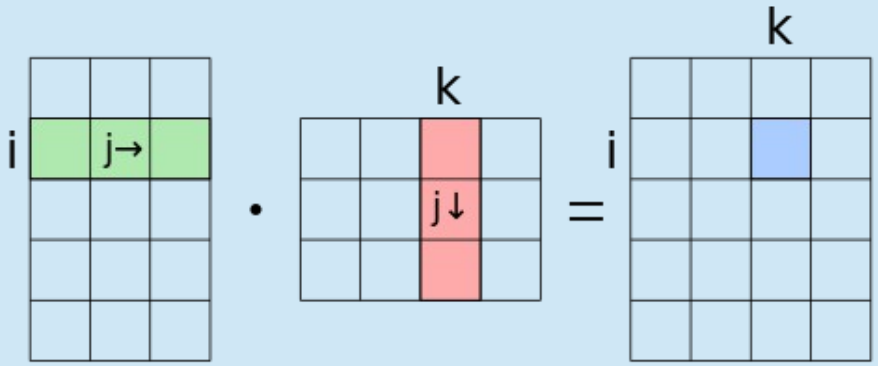


Abb. 11: Matrizenmultiplikation, http://commons.wikimedia.org/wiki/File:Matrix_multiplication_qtl2.svg (22.01.2014)

```
void foo(double p1[SIZE][SIZE],
         double p2[SIZE][SIZE],
         double p3[SIZE][SIZE])
{
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            for (int k = 0; k < SIZE; k++)
            {
                p3[i][j] += p1[i][k] * p2[k][j];
            }
        }
    }
}
```

Abb. 12: Codebeispiel nach Drepper (2007), S. 46.

Für SIZE = 1.000 → 15 sec (O0) bzw. 15 sec (O2)

Cache-Optimierung für Algorithmen (2/3)

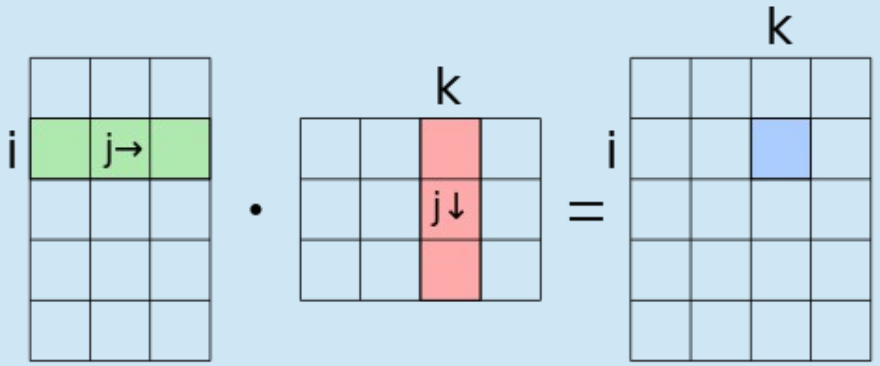


Abb. 11: Matrizenmultiplikation, http://commons.wikimedia.org/wiki/File:Matrix_multiplication_qtl2.svg (22.01.2014)

```
void foo(double p1[SIZE][SIZE],
         double p2[SIZE][SIZE],
         double p3[SIZE][SIZE])
{
   double temp[SIZE][SIZE]; // malloc
   for (int i = 0; i < SIZE; i++)
   {
       for (int j = 0; j < SIZE; j++)
       {
           temp[i][j] = p2[j][i];
       }
   }
   for (int i = 0; i < SIZE; i++)
   {
       for (int j = 0; j < SIZE; j++)
       {
           for (int k = 0; k < SIZE; k++)
           {
               p3[i][j] += p1[i][k] * temp[j][k];
           }
       }
   }
}
```

Abb. 13: Codebeispiel nach Drepper (2007), S. 46.

Für SIZE = 1.000 → 8 sec (O0) bzw. 1,3 sec (O2)

Cache-Optimierung für Algorithmen (3/3)

- Iterationen in Abhängigkeit der Cacheline-Größe
  - `getconf LEVEL1_DCACHE_LINESIZE`
- Data Prefetch
  - Daten in Cache lagern bevor Zugriff
  - z.B. MMX-Prozessoren

Unterstützung für Entwickler - Valgrind

- Virtuelle Maschine
- Übersetzt Code in Vex IR
- Modifizierung durch Werkzeuge
  - Memcheck
- Gezielte Ausführung

Unterstützung für Entwickler - Cachegrind

- Valgrind-Werkzeug
- Misst CPU-Takte
- Misst Cache-Hits und Cache-Misses pro Level
- Simulation verschiedener CPU's durch Einstellung der Cache-Größe

Messdaten von Cachegrind

- Instruction Cache
  - Read
- Data Cache
  - Read/Write
- Generell
  - Unterteilung in L1-Cache und Last-Level-Cache (LL)
  - Miss-Rate
  - Zeilenweise Auswertung
    - Quelldateien
    - Binaries

Auswertung mit Cachegrind – Beispiel Datenlokalität

```
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]++;
        }
    }
}
```

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

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

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

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

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

Zusammenfassungsfolie

- Caching wird automatisch vorgenommen
  - Kann teilweise vom Entwickler übernommen werden
- Entwickler kann Caching optimieren
  - Effiziente Nutzung von Speicher
  - Datenlokalität nutzen und einhalten
  - Der Compiler optimiert den Cache nicht!
- Tools, die qualitatives Feedback über Ressourcennutzung geben: Valgrind und (hier:) Cachegrind