

Performanz

Philipp Klaus Krause

2019-07-09

Table of Contents

- 1** Cache
 - Speicherhierarchie
 - Aufbau
 - Hardwarebeispiele
 - Arrays
 - Strukturen

- 2** Information
 - Datentypen
 - const
 - static
 - inline
 - restrict
 - register

- 1** Cache
 - Speicherhierarchie
 - Aufbau
 - Hardwarebeispiele
 - Arrays
 - Strukturen
- 2** Information
 - Datentypen
 - const
 - static
 - inline
 - restrict
 - register

- Schneller Speicher ist teuer
- Kompromiss zwischen kleinem, schnellen Speicher und großem, billigen Speicher zu finden
- Lösung: Speicherhierarchie: Kombination aus kleinen, schnellen Speicher mit großen, langsamen Speicher
- Z.B. von Prozessorregistern bis zu Festplatten, mit Zwischenstufen
- (Prozessor)Cache und Scratchpad Memory ist Speicher, der schneller ist, als der Hauptspeicher, in dem ein Teil der Daten aus dem Hauptspeicher zwischengespeichert wird.
- Beim Cache entscheidet weitgehend die Hardware, was darin gespeichert ist, beim Scratchpad Memory entscheidet die Software (Programmierer oder Compiler).

Cache Line

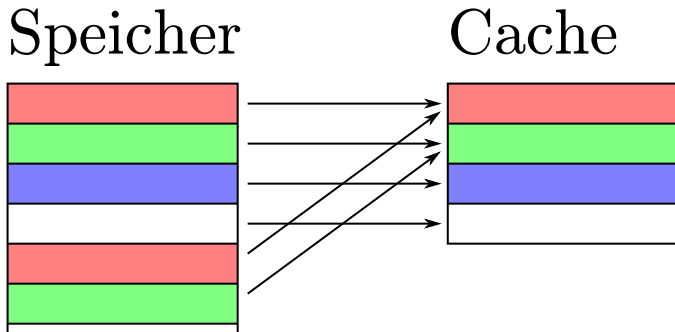
- Der Cache enthält mehrere Einträge, jeweils bestehend aus einem Tag und einer Cache Line.
- Im Tag stehen Informationen darüber, was in der Cache Line gespeichert ist, insbesondere die Adresse.
- Die Cache Line enthält einen Speicherbereich aus dem Hauptspeicher (übliche Größen sind 32B bis 128B).

Zugriff

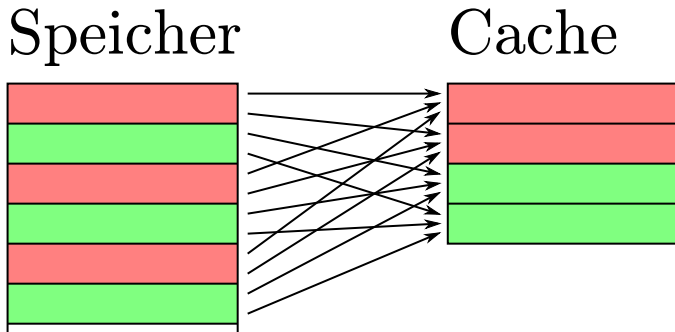
- Beim Lesen aus dem Hauptspeicher wird erst (anhand des Tags) geprüft, ob die Daten im Cache liegen.
- Falls ja: Cache Hit. Schneller Zugriff.
- Falls nein: Cache Miss. Zugriff auf nächstlangsammere Ebene (weiterer Cache oder Hauptspeicher). Langsamer. Gesamte Cache Line wird geladen.

- Voll assoziativer Cache: Jeder Teil des Speichers könnte in jeder Cache Line stehen. Nachteil: Bei jedem Zugriff muß mit allen Tags verglichen werden (Cache wird langsamer).
- Direkt abbildender Cache: Für jeden Teil des Speichers gibt es nur eine mögliche Cache Line. Nachteil: Oft können zwei Objekte nicht gleichzeitig im Cache stehen.
- Mittelweg: n -fach assoziativer Cache. Zu jedem Teil (in Größe einer Cache Line) des Speichers gibt es n Cache Lines.

Direkt abbildend (1-fach assoziativ)



2-fach assoziativ



STM32F72 (ST Microelectronics)

- L1I: 4KB, Cache Lines zu 32B, 2-fach assoziativ, Zugriff in 1 Zyklus.
- L1D: 4KB, Cache Lines zu 32B, 4-fach assoziativ, Zugriff in 1 Zyklus.

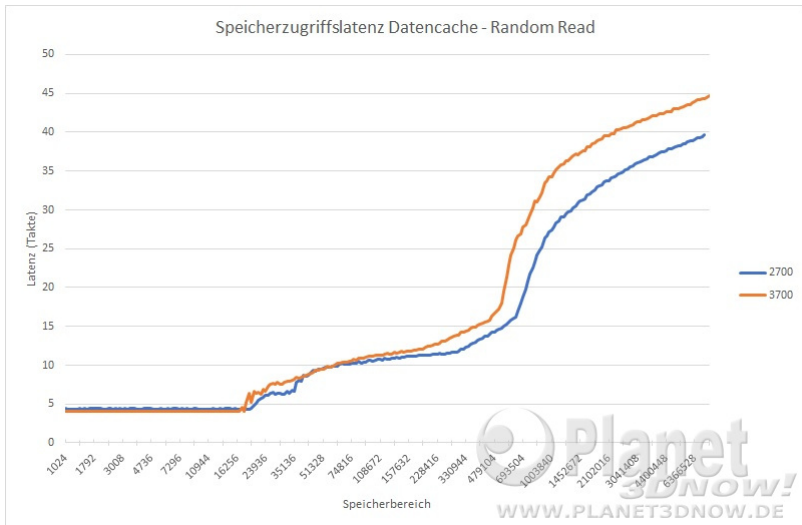
Ryzen 2700X (AMD Zen+)

- L1I: 64KB je Kern, Cache Lines zu 64B, 4-fach assoziativ, Zugriff in 4 Zyklen.
- L1D: 32KB je Kern, Cache Lines zu 64B, 8-fach assoziativ, Zugriff in 4 Zyklen.
- L2: 512KB je Kern, Cache Lines zu 64B, 8-fach assoziativ, Zugriff in 12 Zyklen.
- L3: 16MB, Cache Lines zu 64B, 16-fach assoziativ, Zugriff in 35 Zyklen.

Ryzen 3700X (AMD Zen 2)

- L1I: 32KB je Kern, Cache Lines zu 64B, 8-fach assoziativ, Zugriff in 4 Zyklen.
- L1D: 32KB je Kern, Cache Lines zu 64B, 8-fach assoziativ, Zugriff in 4 Zyklen.
- L2: 512KB je Kern, Cache Lines zu 64B, 8-fach assoziativ, Zugriff in 12 Zyklen.
- L3: 32MB, Cache Lines zu 64B, 16-fach assoziativ, Zugriff in 40 Zyklen.

Ryzen 2700X (AMD Zen+) vs. Ryzen 3700X (AMD Zen 2)



Ryzen 3700X (AMD Zen 2)

en werkzeuge Hilfe

Bericht

AIDA64 Cache & Memory Benchmark

i Memory Write
Write performance measurements offer insight into CPU architecture, but do not reflect typical application workloads.

	Read i	Write i	Copy i	Latency
Memory	42655 MB/s	23466 MB/s	43023 MB/s	83.5 ns
L1 Cache	1731.8 GB/s	870.36 GB/s	1731.7 GB/s	1.1 ns
L2 Cache	872.35 GB/s	843.78 GB/s	872.28 GB/s	3.4 ns
L3 Cache	510.67 GB/s	457.26 GB/s	505.53 GB/s	12.5 ns


CPU Type: OctaCore AMD Ryzen (Matisse, Socket AM4)
CPU Stepping: MTS-B0
CPU Clock: 3500.0 MHz
CPU FSB: 100.0 MHz (original: 100 MHz)
CPU Multiplier: 35x North Bridge Clock

Memory Bus: 1466.7 MHz DRAM:FSB Ratio: 44:3
Memory Type: Dual Channel DDR4-2933 SDRAM (14-14-14-35 CR1)
Chipset: AMD Bixby, AMD K17.7 FCH, AMD K17.7 IMC
Motherboard: Unknown Motherboard
BIOS Version: 0066 (AGESA: Combo-AM4 1.0.0.2)

AIDA64 v6.00.5122 Beta / BenchDLL 4.4.800-x64 (c) 1995-2019 FinalWire Ltd.

Save Start Benchmark Close

Multimedia



Sicherheit

Planet 3DNOW!
WWW.PLANET3DNOW.DE

Cachefreundliche Anordnung von Daten

- Caches sind klein, platzsparende Datenstrukturen sind nützlich.
- Daten sollten so angeordnet werden, dass sie nahe beieinander liegen, wenn oft gemeinsam auf sie zugegriffen wird.
- Somit wird weniger Cacheassoziativität gebraucht.
- Beispiel: `int x[N]` und `int y[N]`. Effizient, wenn meist auf mehrere Elemente von `x` zugegriffen wird, oder auf mehrere von `y`, aber nicht durcheinander.
- Beispiel: `struct s {int x; int y;} s[N]`. Effizient, wenn meist auf `x[i]` zusammen mit `y[i]` zugegriffen wird.

Wieviel Assoziativität?

- Grob: 1 Assoziativität für Programmcode, 1 für den Stack, und bis zu 1 pro Objekt auf dem Heap oder als globale Variable.
- Beispiel: Für `strlen` ist direkt abbildender Datencache ausreichend, wenn der Programmcode in einem anderen Cache liegt und alle lokalen Variablen in Registern stehen (es wird nur ein Array durchlaufen).
- Beispiel: Für `strcmp` will man entsprechend mindestens 2-fach assoziativen Cache (es werden gleichzeitig 2 Arrays durchlaufen).
- Beispiel: Multiplikation von mittelgroßen Matrizen auf dem Heap, lokale Variablen auf dem Stack, Programmcode im gleichen Cache will man mindestens 5-fach assoziativen Cache.

Wieviel Assoziativität? - Game of Life

- Game of Life mit 2 `intplane_t`
- Jede `intplane_t` selbst auf dem Heap, verwendetes Array auf dem Heap.
- Durchlaufen z.B. alte `intplane_t` zeilenweise, schreiben neue `intplane_t`.
- Zugriff auf 3×3 -Bereich der alten `intplane_t`.
- Bei großem `intplane_t`, je nach Größe: Jede Zeile wie eigenes Array.
- Wollen 6-fache Assoziativität (je 2 für die `intplane_t`, 1 für die Zeile der neuen `intplane_t`, die wir schreiben, 3 für die 3 Zeilen der alten `intplane_t`, die wir lesen).
- Beispiel: `strs_t` hat cachefreundlicheres Interface als `intplane_t`.

Beispiel: Suchbaum

- Da Zugriffe auf den Cache viel schneller sind, als auf den Hauptspeicher, kann eine einseitige Fixierung auf die O -Notation irreführend sein.
- In einem Baum mit einzeln allozierten Knoten ist die Anzahl der Knoten entscheidend, da beim jeweils ersten Zugriff meist ein Cache Miss auftritt.
- Selbst unsortierte Arrays sind bis zu einer gewissen Größe (im Bereich von tausenden Einträgen) effizienter als binäre Suchbäume mit je einem Element pro Knoten.
- Somit ist ein Baum, bei dem bei jedem Knoten mehr als 1 Element gespeichert wird, effizienter - man sollte mindestens eine Cacheline mit den Daten eines Knotens füllen.

OpenGL: Vertex Arrays

- Per `glEnableClientState(GL_COLOR_ARRAY)`; und mit `glColorPointer` lässt sich effizient die Farbe einzelner Ecken angeben.
- Cachefreundlicher, wenn die Farben im gleichen Array wie die Koordinaten der Ecken liegen (mittels `stride` machbar).

```
void glVertexPointer(GLint size, GLenum type,  
    GLsizei stride, const void *pointer);
```

```
void glColorPointer(GLint size, GLenum type,  
    GLsizei stride, const void *pointer);
```

Buntes Quadrat aus zwei Dreiecken

```
const GLfloat square_corners[] = {
    0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
    0.8f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 0.8f, 0.0f, 0.0f, 1.0f, 1.0f,
    0.8f, 0.8f, 1.0f, 1.0f, 1.0f, 1.0f,
};
...
glVertexPointer(2, GL_FLOAT,
               6 * sizeof(GLfloat), square_corners + 0);
glColorPointer(4, GL_FLOAT,
              6 * sizeof(GLfloat), square_corners + 2);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

Alignment

- Auf vielen Systemen sollen Objekte abhängig vom Typ nicht an beliebigen Stellen im Speicher liegen, sondern bei Adressen, die ein Vielfaches des einer Zahl $a > 0$ sind (a heißt *Alignment* des Typs).
- Zugriffe an anderen Stellen sind langsam (z.B. x86, amd64) oder führen gar zu Fehlern (z.B. ARMv4).
- Der Compiler kümmert sich darum.
- `char`, `signed char`, `unsigned char` haben immer Alignment 1, können also überall im Speicher stehen.
- `malloc` liefert immer Speicher, der für jegliche Objekte geeignet ist. Effizienter kann `aligned_alloc` sein.

```
#include <stdalign.h>
int *a0 = malloc(13 * sizeof(int));
int *a1 = aligned_alloc(alignedof(int), 13 * sizeof(int));
```

Alignment

Wegen der Einschränkungen an das Alignment muss der Compiler zusätzliche Bytes (*Padding Bytes*) in Strukturen einfügen:

```
struct inefficient;
{ // int habe Alignment i.
  char a; // Nach a: i - 1 Padding Bytes.
  int i;
  char b; // Nach b: i - 1 Padding Bytes.
  int l;
};

struct efficient;
{ // int habe Alignment i.
  char a;
  char b; // Nach b: max{i - 2, 0} Padding Bytes.
  int i;
  int l;
};
```

bit-fields

- Datentypen haben feste Größen (müssen sie in gewissen Grenzen auch, um adressierbar zu sein).
- Ausweg: bit-fields (Member von struct, die nicht adressierbar sind), mit frei wählbarer Größe.
- Typen: `bool`, `signed int` (das `signed` kann man hier nicht einfach weglassen), `unsigned int`.
- Bereich entsprechend dem Typ und der angegebenen Anzahl an Bits.
- Zugriff aufwendig (damit langsamer als normale struct, die voll im Cache liegt), aber speichersparend (cachefreundlich).

```
struct s; // Passt oft in ein Byte
{
    unsigned int i : 6; // {0, ..., 63}
    bool b : 1; // {true, false}
    signed int j : 1; // {-1, 0}
};
```

- 1 Cache
 - Speicherhierarchie
 - Aufbau
 - Hardwarebeispiele
 - Arrays
 - Strukturen
- 2 Information
 - Datentypen
 - const
 - static
 - inline
 - restrict
 - register

- float ist üblicherweise schneller und platzsparender als double, welches wiederum schneller und platzsparender als long double ist.
- Vorzeichenlose Typen (`unsigned`) sind oft schneller als Vorzeichenbehaftete.
- Für Ganzzahlen: Typen aus `stdint.h`: Nächste Folie.

typedefs für Ganzzahlen

- `uint_fastN_t`, `int_fastN_t`: Schnell
- `uint_leastN_t`, `int_leastN_t`: Platzsparend
- $N = 8, 16, 32, 64$.
- Gilt im Allgemeinen für einzelne Variablen (wie z.B. Schleifenzähler). Insbesondere bei Arrays kann es passieren, dass die platzsparenden Varianten zu schnellerem Code führen (wegen Cache).

- `const` gibt an, dass auf ein Objekt nur lesend zugegriffen wird.
- Dies ermöglicht dem Compiler, z.B. diese in nur-lese-Speicher abzulegen, der unter Umständen schneller oder in größerer Menge verfügbar ist.
- Man erhält eine Fehlermeldung, wenn versucht wird, schreibend zuzugreifen.

const

```
const GLfloat square_corners[] = {  
    0.0f, 0.0f,  
    0.8f, 0.0f,  
    0.0f, 0.8f,  
    0.8f, 0.8f,  
};
```

- Bei globalen (file scope) Objekten gibt `static` an, dass diese nur in der aktuellen Übersetzungseinheit (.c-Datei) sichtbar sind.
- Der Compiler kann sich also darauf verlassen, dass andere Übersetzungseinheiten die Variable nur ändern, wenn sie irgendwie einen Zeiger darauf bekommen.
- Sinnvoll bei Variablen, die von mehreren Funktionen, die sich aber alle in der selben .c-Datei befinden, verwendet werden.
- Sinnvoll bei Hilfsfunktionen, die nur von anderen Funktionen aus der gleichen .c-Datei genutzt werden.

Im Header:

```
// Jede .c-Datei erhält ihre eigene Kopie
// (kein Linkerfehler, trotz mehrerer f und i)
static void f(void)
{
    ...
}
static int i; // Je ein Objekt pro .c-Datei,
// Schreibzugriff ändert nur das jeweils eigene!
```

static

Wenn in der .c-Datei nirgendwo ein `&i` steht, kann der Compiler davon ausgehen, dass `p[0]` ein anderes Objekt ist als `i`.

```
static int i
int j;

static void f(int *p)
{
    i = 1;
    p[0] = 23;
    // Compiler kann sich darauf verlassen, dass i == 1.
    j += i;
}
```

- Funktionen sind zwar sehr nützlich zur Strukturierung des Codes, können aber zu Performanzeinbußen führen, da die Optimierung über Funktionsgrenzen hinweg aufwendig ist.
- Funktionen können als `static inline` in einem Header definiert werden. Solche Funktionen sind der Optimierung zugänglicher.
- Dies sollte für kleine, häufig aufgerufene Funktionen erfolgen.
- Nachteile für Modularisierung: Unvollständige Typen unter Umständen nicht möglich, Änderungen am Header wahrscheinlicher, dann mehr neu zu kompilieren.

Beispiel:

- `intplane_t` von Übungsblatt 7 / 8.
- `intplane_set` und `intplane_get` in `intplane.h` als `static inline` definiert.
- Andere Funktionen bleiben in `intplane.c`.

- Wenn auf ein Objekt über einen restrict-Zeiger zugegriffen wird, müssen alle Zugriffe über Zeiger, die von diesem abgeleitet sind, erfolgen.
- Dies ist insbesondere bei als Funktionsparameter übergebenen Zeigern nützlich.
- Beispiel aus der Standardbibliothek: `memcpy` vs. `memmove`.

```
// Kopiert. Bereiche dürfen nicht überlappen.  
void *memcpy(void *restrict s1, const void *restrict s2,  
             size_t n);  
// Kopiert. Bereiche dürfen überlappen. Langsamer.  
void *memmove(void *s1, void *s2, size_t n);
```

restrict

Beispiel:

```
static int i, j;
static void f(int *restrict p, int *restrict *q)
{
    (*q)++;
    *p = i; // Ändert *q nicht!
    j = *q;
}
```

```
void h(int k)
{
    f(&k, &k); // Fehler
}
```

- `register` gibt bei lokalen (d.h. function scope oder block scope) Variablen an, dass deren Adresse nicht verwendet wird.

```
void f(void)
{
    register int i;
    ...
    &i; // Fehler
    ...
}
```