

# **Grundlagen von Caching-Mechanismen beim Zusammenspiel von Mikroprozessor und Betriebssystem**

*Klaus Kusche  
Dezember 2015*

# Inhalt

- **Ziele & Voraussetzungen**
- **Grundidee & Beispiele von Caches**
- **Bedeutung effizienter CPU-Caches**
- **Caches in modernen CPU's**
- **Zusammenspiel Caches / Betriebssystem**

# Ziele

## Verständnis

- ... der Grundidee und der Arten von Caches,
- ... der Bedeutung von Caches,
- ... der Caches moderner Prozessoren,
- ... ihrer “Rückwirkung” auf das Betriebssystem, ihrer effizienten Nutzung

## Kenntnis

- ... der wichtigsten Fachbegriffe

# Voraussetzungen

## Grundkenntnisse von

- Hardware:

Arbeitsweise Prozessor und Speicher,  
Adressierung, MMU

Multicore- / Multiprozessor-Systeme

- Betriebssystem:

Virtuelle Speicherverwaltung  
(virtuelle / reale Adressen)

Prozesse & Scheduling

# Ein Cache ist ...

... ein

**schnellerer Zwischenspeicher**

für

vermutlich **mehrmals** benötigte Daten

die

im Original **langsam** zu beschaffen

sind.

# Idee des Caching

Wenn du

**dieselben Daten**

**zum zweiten, dritten, ... Mal**

brauchst, bekommst du sie

**“sofort” aus dem Cache**

und musst kein zweites, drittes, ... Mal  
langsam auf die Originaldaten zugreifen.

# Ein Cache soll ...

... *transparent*

= *ohne* Änderung der Funktionalität,

*“unsichtbar”*

für darauf aufbauende Software

und für den Anwender

***die Performance steigern!***

# Beispiele für Software-Caches

- In Anwendungen / *Libraries* / *Diensten*:
  - Webseiten-Cache im Browser
  - DNS-Cache (Name Resolution Cache)
- Im Betriebssystem:
  - Cache für **Filesystem-** und **Directory-Daten**
  - Netzwerk: **ARP-Cache** (Level 2 Adressen / IP-Adressen)

==> Cache-Verwaltung in **Software**

==> Cache-Daten im **normalen RAM**



# Beispiele für Hardware-Caches

- In Platten-Laufwerken & RAID-Controllern:

- Disk-Cache

==> Cache-Verwaltung durch *SW in eigenem  $\mu$ C*

==> Cache-Daten in *eigenen DRAM-Chips*

- Im Prozessor: ... kommt gleich!

==> Cache-Verwaltung in *HW-Logik*

==> Cache-Daten in *eigenem On-Chip-RAM*

(technologisch: schnelles SRAM, nicht DRAM)

# Praktischer Nutzen ... (1)

Entwicklung von Mikroprozessoren:

intel 8080 (2 MHz) ... intel i7 (4 GHz)

Takt-Speedup 2000, aber realer Speedup  $\gg$  1 Mio.

**$\Rightarrow$  HW-Architektur-Speedup**

(bei gleichem Takt)

**$\gg$  500**

Davon (grob geschätzt)

***Anteil Speedup durch Prozessor-Caches***

***~ 10***

# Praktischer Nutzen ... (2)

Instruktions-Bedarf eines modernen Prozessors:

4 Cores \*

~ 3,5 Instruktionen pro Core & Takt \*

~ 2,5 Bytes pro Instruktion \*

4 GHz =

**140 GB / s** (nur Instruktionen, noch ohne Daten!)

Realer RAM-Durchsatz: **max. 10 - 20 GB / s**

**=> Ohne Caches “verhungert” der Prozessor!**

# Praktischer Nutzen ... (3)

*... siehe Demo valgrind*

***Cache Hit Rates***

***> 99 % !***

# Caches im Prozessor (1)

- Für Daten und Instruktionen,  
d.h. für das “normale” RAM

	Größe	Zugriffszeit (Takte)	Anordnung (typisch)
L1 I	16 - 128 KB	2 - 5	per Core
L1 D	16 - 128 KB	2 - 5	per Core
L2	0,25 - 4 MB	8 - 25	per 1-4 Cores
L3	0 - 64 MB	25 - 90	shared per Chip
L4	0 - 256 MB	> 100	externes eDRAM

*Zum Vergleich: Zugriffszeit RAM: > 250 Takte !!!*

# Caches im Prozessor (2)

- **TLB (Translation Lookaside Buffer)**

In der MMU: Cache für Pagetable Entries

L1: 16 - 256 Entries + L2: 64 - 2048 Entries

reicht für 256 KB - 8 MB Working Set (“aktives” RAM)

- **BTB (Branch Target Buffer)**

Im Instruction Decoder / Sprungvorhersage:

Cache für bisherige Sprungziele

256 - 8192 Sprünge

(jede “falsche” Sprungvorhersage kostet ~ 15 Clocks!)

**=> Mehr als 50 % der Chipfläche sind Caches!**

# Merkmale eines Caches

- Schreib-Strategie:  
“*Write through*” / “*Write back*”
- Größe einer Cache-Zelle = “*Cache Line Size*”  
(32 - 128 Bytes)
- Was kommt wohin? (Wie findet man die Daten wieder?)  
“*Assoziativität*” (bzw. “*Directly mapped*”) + “*Tags*”
- Was wird rausgeworfen? (wenn der Cache voll ist)  
“*Replacement strategy*”

# Zusammenspiel ... (1)

*... zwischen Mikroprozessor und Betriebssystem???*

Auf den ersten Blick im Normalbetrieb minimal:

- Hardware-Caches sind

*transparent für das Betriebssystem*

- Software-Caches des Betriebssystems sind

*Prozessor-unabhängig*



# Zusammenspiel ... (2)

*Aber:*

- ***VIVT-Caches: “Showstopper” für moderne OS!***
- Performance-Kriterium:  
***“Cache-freundliches Scheduling”***
- Allgemein:  
***“Cache-freundliche Programmierung”***
- Kleinigkeiten, z.B.
  - **Disk Cache Flush** bei Shutdown oder Unmount
  - **TLB Flush** bei Adressraum-Wechsel
  - I/O- und DMA-Bereiche ev. **“uncacheable”** setzen

# VIVT-Caches: Idee, Vorteil

*VIVT ... virtually indexed, virtually tagged*

*VIPT ... virtually indexed, physically tagged*

*PIPT ... physically indexed, physically tagged*

Arbeitet der Cache mit **virtuellen / realen RAM-Adressen?**

*VIVT-Caches sind “**vor**” der MMU (CPU-seitig)*

*PIPT-Caches sind “**dahinter**” (RAM-seitig)*

- **Cache Hit:** *Zeit für MMU / Address Translation entfällt!*  
(2-4 Clocks bei TLB Hit, “ewig” bei TLB Miss)
- **Cache Miss:** *Zeit für Cache Lookup entfällt!*  
(weil Cache Lookup und Address Translation parallel laufen)

**==> VIVT-Caches sind schneller!**

# VIVT-Caches: Nachteile (1)

Nach jedem Prozess- (Adressraum-) Wechsel:

***Cache ist ungültig!***

(selbe virtuelle Adresse = selbe Cache-Zelle  
entspricht plötzlich anderen Daten im RAM)

- Betriebssystem muss Cache komplett invalidieren  
(meist per SW-Schleife pro Cacheline),  
dabei Writeback aller “dirty Cachelines”
- Nach jedem Prozesswechsel ist der Cache leer:  
Neuer Prozess läuft anfangs sehr langsam,  
Cache muss erst wieder “warmlaufen”

**==> Sehr langsam!!!**

# VIVT-Caches: Nachteile (2)

Weitere Probleme:

- Wenn das OS zwei verschiedene virtuelle Adressen auf dieselbe reale Adresse mappt:

***Zwei Cache-Zellen speichern selben RAM-Inhalt!***

***“Cache Aliasing”*** (Vermeidung durch OS oder HW)

- VIVT-Caches sind nicht Multicore- oder I/O-fähig:

Datenänderung “von außen” mit realen Adressen würde “Reverse Address Translation” erfordern!

(zur Invalidierung der betreffenden Daten im Cache)

# VIVT-Caches: Realität

- **Echte VIVT-Caches** nur bei *sehr kleinen oder sehr alten Prozessoren* (z.B. ARM-Prozessoren > 12 Jahre alt)
  - ==> **Für moderne OS (Linux) de facto ungeeignet**
  - ==> OS ohne virtuelle Speicherverwaltung nötig!  
(oder mit gemeinsamen / disjunkten Adressräumen pro Prozess)
- Bei “großen” Prozessoren:
  - **L1 ist VIPT** + Hardware-Tricks:  
Fast so schnell wie VIVT  
Prozesswechsel-sicher in Hardware
  - **L2, L3 ist PIPT**

# Caches und Scheduler (1)

- **Prozessor:**

Shared Cache ist aufwändiger und langsamer

=> ***L1 und L2*** sind meist ***lokal pro Core*** (ev. 2 Cores)

- **Scheduler:**

Ziel: Optimale Auslastung, schnellste Reaktion

=> Ein “Runnable” Prozess wird ***möglichst sofort***  
***auf irgendeinen freien Core*** gelegt

# Caches und Scheduler (2)

Problem, wenn der Prozess zuletzt auf einem anderem Core gelaufen ist:

***Code & Daten liegen im “falschen” Cache***

- Prozess läuft langsamer
- Cache-Ausnutzung sinkt  
(Daten des Prozesses belegen 2 Caches!)

==> Jeden Prozess *“wenn es leicht geht”* (Heuristik!) ***auf “seinem” Core halten***

Bei Servern: Dasselbe für Prozessor-Chips (L3-Cache)

# “Cache-freundlicher” Code (1)

Caches laden immer

***eine ganze Cacheline***  
(meist 64 Bytes)

Nach jedem Zugriff ist der Zugriff auf die

***64 Nachbar-Bytes “fast gratis”***  
(Daten sind schon im L1-Cache)

- 1024 “einzeln verstreute” Bytes lesen  
kostet 64 KB Cache und 1024 RAM-Zugriffe
- 1024 “sequentielle” Bytes lesen  
kostet 1 KB Cache und 16 RAM-Zugriffe



# “Cache-freundlicher” Code (2)

- ==> Häufig / gemeinsam benutzte Daten  
unmittelbar nacheinander speichern!
- ==> In Strukturen und Objekten:  
Alle kleinen und häufigen Member vorne,  
alle großen oder seltenen Member hinten!
- ==> Mehrdim. Arrays zeilenweise durchlaufen,  
nicht spaltenweise!
- ==> Daten auf Cacheline-Grenzen ausrichten!  
(z.B. 32 Bytes quer über 2 Cachelines: Unklug!)

*“The end”*

*Fragen?*