Autoren: Emirhan Afsin, Kartik Krishnan, Nirbheek Setia

Recherche bezüglich Gruppenprojekt (Grundlagenpraktikum Rechnerarchitektur) - Caches

Motivation für die Nutzung von Caches[1]:

Der Arbeitsspeicher in Rechnerarchitekturen weist eine große Speicherkapazität auf, hat aber auf Grund dieser enormen Größe verglichen zu Cachestrukturen eine merkbar höhere Latenz. Zum einen ist der Suchaufwand in einem Hauptspeicher groß und es ist auch physisch weiter von dem Prozessor entfernt als der Cache.
Caches sind im Vergleich zum Hauptspeicher kleiner aber ihre Latenz ist vergleichsweise ebenfalls niedriger.

Die Cache Memory ist auf einem schnelleren „static RAM“ aufgebaut, während der Systemspeicher auf dem langsameren „dynamic RAM“ aufgebaut ist. Bei dem „dynamic RAM“ ist das Problem, dass es konstant aufgefrischt werden muss, da es bei den Kondensatoren, aus dem es besteht, zu Ladungsverlusten kommen kann. Der „static RAM“ hingegen hat diese Probleme nicht.

Relevanz von Caches im Detail: Moderne Prozessoren funktionieren aktuell mit einer Frequenz von mehr als 4GHz während die DDR4 Module mit weniger als 1800 MHz arbeiten. Der Systemspeicher ist also zu langsam, um direkt mit dem Prozessor zu arbeiten, ohne ihn dabei zu drosseln. Der Cache ist in diesem Konstrukt von Prozessor, Cache und Hauptspeicher der „Mann in der Mitte“.

Jedoch kann auf Basis der geringeren Speichergröße des Cache nur ein Teil der Speicherelemente zwischengespeichert werden. Die restlichen Daten müssen im Hauptspeicher gelagert und bei Gebrauch in den Cache verschoben werden. Für effiziente Speicherverwaltung existieren folgende Prinzipien…

Prinzipien der Kommunikation von Cache und Hauptspeicher[2]:

Damit die Cachestruktur und der Hauptspeicher gemeinsam zu einer effizienteren Rechnerarchitektur beitragen können werden folgende Prinzipien in Betracht gezogen:

Lokalitätsprinzip:
Ziel bei diesem Prinzip ist es, eine möglichst hohe Hit-Rate und kleine Miss-Rate zu erreichen. Zum Erreichen dieser Werte ist eine Vorhersage über die zukünftigen Zugriffe nötig. Eine schlechte Hit-Rate entspricht ca. 50%, eine mittlere 70% und eine akzeptable 90%.
Das Lokalitätsprinzip teilt sich in zwei Unterprinzipien:

Zeitliche Lokalität: Hierbei erhöht ein Zugriff auf eine Zelle die Wahrscheinlichkeit, dass ein weiteres Mal auf die Zelle zugegriffen wird.
Räumliche Lokalität: Ein Zugriff auf eine Zelle erhöht die Wahrscheinlichkeit, dass auf naheliegende andere Zellen ebenfalls zugegriffen wird (bspw. Arrays, Listen).

[1] https://hardwaretimes.com/difference-between-l1-l2-and-l3-cache-how-does-cpu-cache-work/

[2] ERA-Vorlesungen (Caches)

Aufbau eines Cache[3]:

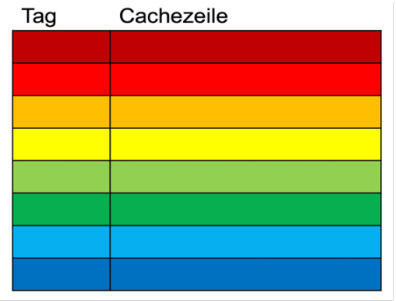
Ein Cache besteht aus einer Menge an Speicherzellen einer festen Länge. Dies wird die Cachezeilenlänge genannt. Jede Speicherzelle ist mit einem Tag identifiziert. Der Zugriff auf Daten im Cache findet durch folgende Parameter statt:

Offset: Dies sind die unteren Bits der Adresse als Index.
Index: Bestimmt die Cachezeile, in der das Datum gespeichert werden sollte.
Tag: Entspricht den Daten, die in dem Cache abgelegt werden.

Variation von Caches:

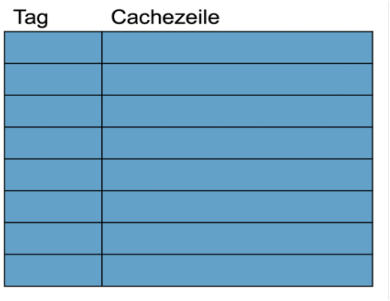
1. Direct Mapped Caches:

Funktionsweise: Bei einem direct mapped Cache wird ein Datum auf eine spezifische Zeile im Cache gemappte. Dieses Datum bzw. indexweise ähnliche Daten werden auf dieselbe Zeile gemappte. Dementsprechend sind bei dieser Art von Caching mit einer höheren Rate von „Kollisionskonflikten“ zu rechnen. Nichtsdestotrotz sind hier die Zugriffe schneller als bei anderen Caches, da beispielsweise bei Read-Aufrufen nur eine Zeile des Cache überprüft werden muss. Also ist pro Aufruf nur ein einziger Vergleich notwendig.



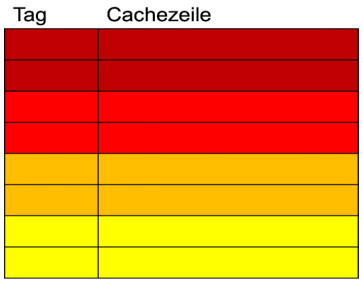
2. Vollassoziative Caches:

Funktionsweise: Bei dieser Art von Caches haben die Daten keine feste Position. Folglich haben sie auch keine Indexbits, um ihre Zeile zu berechnen. Zu Beginn werden die Daten bei einem leeren Cache Zeile für Zeile in den Cache geladen. Dem Fakt, dass jedes Datum in jeder Zeile des Cache ohne jegliches System stehen kann, zu Folge sind in dieser Form des Caching eine höhere Anzahl an Vergleichen und aus diesem Grund auch mit einer höheren Latenz zu rechnen.



3. Mengenassoziative Caches:

Funktionsweise: Ein Datum kann auf eine Teilmenge des Cache gemappt werden. Diese Teilmenge wird durch die Indexbits bestimmt. Bei Zugriffen werden die Tags innerhalb der Teilmenge betrachtet, um die passende Teilmenge zu finden. Ein Datum wird nur in dem Fall verhängt, falls die Menge voll ist. Dieser Cache folgt oftmals dem Prinzip des „Last-Recently-Used“ kurz LRU. Das am längsten ungenutztes Datum wird verdrängt.



[3] **ERA-Vorlesungen (Caches) + ERA Repetitorium**

Berechnung der Bits zur Speicherung der Einträge[4]:

Vollassoziativ:
Anzahl der Cachezeilen: Cachesize / Zeilenlänge
Index = 0
Offset = $\log_2(Cachezeilenlänge)$
Tag = Adresse – Offset

Mengenassoziativ:
Anzahl der Cachezeilen: Cachesize / Zeilenlänge
Index = $\log_2(Anzahl\ der\ Cachesets)$
Offset = $\log_2(Cachezeilenlänge)$
Tag = Adresse - Index – Offset

Direct Mapped:
Anzahl der Cachezeilen: Cachesize / Zeilenlänge
Index = $\log_2(Anzahl\ der\ Cachezeilen)$
Offset = $\log_2(Cachezeilenlänge)$
Tag = Adresse - Index – Offset

Virtuelle und physische Adressen:

Die Speicherformen von Caches können sich nach Design ändern. Beispielsweise ist die Intel Sany Bridge folgendermaßen aufgebaut:
1. L1: virtueller Speicher mit physischen Tags
2. L2: physischer Speicher
3. L3: physischer Speicher

Speichergrößen in Caches:

Da Rechnerarchitekturen auf Basis von Binärzahlen funktionieren und arbeiten, ist es ungünstig Dezimalformen von Daten für das Caching zu verwenden.
Im Folgenden ist der Vergleich und dadurch Unterschiede zwischen Binär- und Dezimalpräfixen sichtbar:

Dezimalpräfixe			Binärpräfixe	
Name (Symbol)	Bedeutung	Unterschied (gerundet)	IEC-Name (IEC-Symbol)	Bedeutung
Kilobyte (kB)	10^3 Byte = 1.000 Byte	2,40 %	Kibibyte (KiB)	2^{10} Byte = 1.024 Byte
Megabyte (MB)	10^6 Byte = 1.000.000 Byte	4,86 %	Mebibyte (MiB)	2^{20} Byte = 1.048.576 Byte
Gigabyte (GB)	10^9 Byte = 1.000.000.000 Byte	7,37 %	Gibibyte (GiB)	2^{30} Byte = 1.073.741.824 Byte
Terabyte (TB)	10^{12} Byte = 1.000.000.000.000 Byte	9,95 %	Tebibyte (TiB)	2^{40} Byte = 1.099.511.627.776 Byte
Petabyte (PB)	10^{15} Byte = 1.000.000.000.000.000 Byte	12,6 %	Pebibyte (PiB)	2^{50} Byte = 1.125.899.906.842.624 Byte
Exabyte (EB)	10^{18} Byte = 1.000.000.000.000.000.000 Byte	15,3 %	Exbibyte (EiB)	2^{60} Byte = 1.152.921.504.606.846.976 Byte
Zettabyte (ZB)	10^{21} Byte = 1.000.000.000.000.000.000.000 Byte	18,1 %	Zebibyte (ZiB)	2^{70} Byte = 1.180.591.620.717.411.303.424 Byte
Yottabyte (YB)	10^{24} Byte = 1.000.000.000.000.000.000.000.000 Byte	20,9 %	Yobibyte (YiB)	2^{80} Byte = 1.208.925.819.614.629.174.706.176 Byte

[4] ERA-Vorlesungen (Caches) + ERA-Repetitorium

Cachegrößen in modernen Prozessoren[5]:

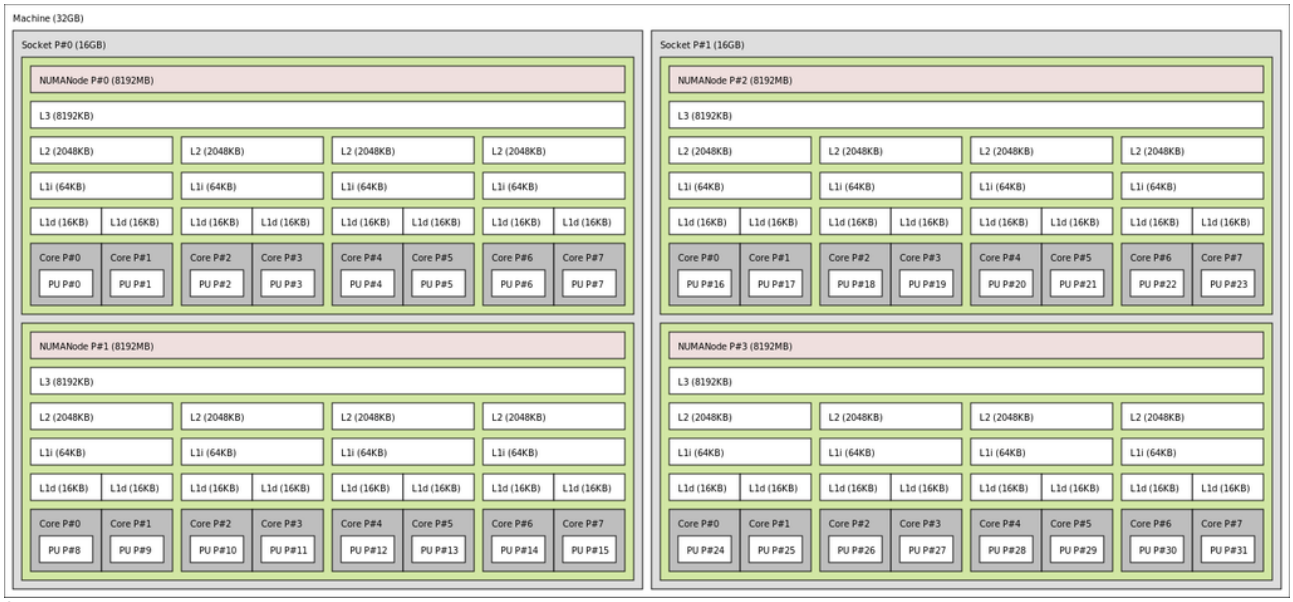
In modernen, aktuellen Prozessoren werden die Caches in Cachestufen unterteilt. Dabei finden L1, L2 und L3 Caches die häufigste Verwendung. Diese sind nach zunehmender Größe und der daraus resultierenden, fallenden Geschwindigkeit (erhöhter Latenz) aufgeteilt.

Das L3 Cache bildet dabei den größten und langsamsten Abschnitt. Der Systemspeicher interagiert mit dem L3 Cache, da dieser die niedrigste und langsamste Stufe des Cache ist. Dadurch werden die L1 und L2 Caches nicht durch den Systemspeicher gedrosselt.
Der L3 Cache variiert zwischen 10MB bis 64MB an Speicherplatz. Serverchips bieten hierbei bis zu 256MB L3 Caches an.

Der L2 Cache ist die Mitte zwischen dem schnellen, kleinen L1 Cache und dem großen, langsamen L3 Cache. Die Größe dieses Abschnittes reicht von 4 bis 8 MB. Für Flagship Prozessoren werden zum Teil 512kb L2 Caches pro Kern verwendet.

Den Abschluss macht der L1 Cache, welcher in 2 Bereiche aufgeteilt wird: Data Cache und Instruction Cache. Der L1 Cache enthält Instruktionen, Branching-Informationen, vorprogrammierte Daten und weiteres. Der Data Cache operiert oft als Output-Cache und der Instruction Cache als Input-Cache.

Bei modernen Prozessoren besitzt jedes Core einen eigenen L1 und L2 Cache, während der L3 Cache universell mit allen Cores geteilt wird.



6

[5] https://hardwaretimes.com/difference-between-l1-l2-and-l3-cache-how-does-cpu-cache-work/
[6] https://en.wikipedia.org/wiki/CPU_cache#/media/File:Hwloc.png

Latenzen bei Hauptspeicher und Caches in modernen Prozessoren[7]:

Die Latenzen pro Cacheabschnitt und Hauptspeicher variieren stark, abhängig von der große der Caches:

L1 Cache:
Die Latenz ist in diesem Abschnitt sehr niedrig (1-3 Zyklen). Dazu trägt auch der Fakt bei, dass die Caches direkt in die CPU-Kerne integriert sind und dadurch physisch sehr nah sind.

L2 Cache:
Der L2 Cache besitzt eine höhere Latenz als der L1 Cache (3 - 10 Zyklen). Diese sind natürlicherweise größer als L1 Caches und sind zusammen mit L1 Caches direkt im CPU-Kern enthalten.

L3 Cache:
Dieser Cache besitzt eine höhere Latenz als der L2 Cache resultierend aus dem enormen Größenunterschied. Dieser Cache wird von allen CPU-Kernen geteilt.

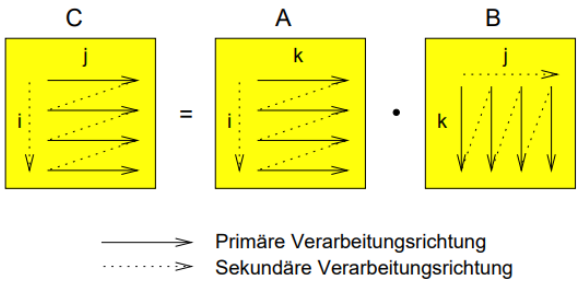
Systemspeicher / Hauptspeicher:
Im Vergleich zu Caches hat der Hauptspeicher eine sehr hohe Verzögerung. Hierauf wird über den Speicherkontroller zugegriffen und dieser Zugriff dauert länger als ein Cachezugriff.

[7] https://geekysoumya.com/what-is-cache-memory-of-processor-what-are-l1-l2-l3-cache/
https://ar5iv.labs.arxiv.org/html/2103.14808

Untersuchung des Speicherzugriffsverhaltens eines speicherintensiven Algorithmus:

Im Folgenden wird die Matrix-Matrix-Multiplikation näher betrachtet. Dabei soll aber die einfache, ungeblockte Matrix-Matrix-Multiplikation als Beispiel dienen, um daran die Einflüsse von Speicherhierarchien auf die Ausführungszeit zu beobachten. Die erwartete Entwicklung der Ausführungszeit, die man für eine Matrix-Matrix-Multiplikation bei wachsender Matrixgröße benötigt, ist $O(n^3)$. Jedoch wächst die tatsächliche Laufzeit um einiges schneller.

Dies liegt an der steigenden Anzahl an Speicherzugriffen bei wachsender Matrixgröße. Um das genauer zu verstehen, schaue man sich untenstehende Grafik an. Seien A_Z, A_S, B_z, B_S die Anzahl der Spalten beziehungsweise Spalten von A und B. Wenn man die gewählte Schleifenreihenfolge beibehält, dann wird C Spalte für Spalte berechnet. Das heißt, dass die erste Zeile von A in den Cache geladen wird und dann mit jeder Spalte von B das Skalarprodukt berechnet wird, bevor die nächste Zeile von A in den Cache geladen werden muss. Damit sind die Speicherzugriffe auf Elemente von A effizienter, da jede Zeile für genau B_Z Skalarprodukte hintereinander gebraucht wird, aber nur einmal in den Cache geladen werden muss. Speicherzugriffe auf B hingegen sind deutlich langsamer, da für jedes Skalarprodukt eine neue Spalte gebraucht wird, weshalb bei jedem Skalarprodukt neue Daten in den Cache geladen werden müssen. Daher dominieren die Zugriffe auf Matrix B für größer werdende Matrizen die Gesamtausführungszeit des Programms.




```
for (i = 0; i < matrixSize; i++)
  for (j = 0; j < matrixSize; j++)
    for (k = 0; k < matrixSize; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Trotzdem ist die Ausführungszeit mit unserer Implementierung, die 2 Caches als Zwischenspeicher und den Hauptspeicher simuliert, deutlich geringer als ohne Zwischenspeicher. Dies kann man gut an unserer Implementierung sehen:

Die Ausführungszeit ohne Zwischenspeicher: 3840 Zyklen

Die Ausführungszeit per Zwischenspeicher mit folgenden Eigenschaften (Größe L1Cache: 16 Zeilen, Größe L2Cache: 32 Zeilen): 3342 Zyklen, 12 hits, 14 misses

Abrundendes Fazit zur Recherche über Caches im Rahmen des Grundlagenpraktikums Rechnerarchitektur:

Caches ist unverzichtbar, um die Geschwindigkeiten der modernen Prozessoren des heutigen Zeitalters zu verwirklichen. Durch die Verwendung von schnellem „static RAM“ werden bei Caches niedrigere Latenzen und optimieren die Speicherverwaltung ermöglicht. Die Prinzipien zur Speicherung von Daten, die unterschiedlichen Arten von Caches (direkt assoziativ, etc.) ermöglichen effiziente Datenzugriffe, die die Zeit, die ein Rechner zur Abarbeitung einer Instruktion braucht, erheblich verringert. Insgesamt verbessern Caches die Effizienz und Leistung moderner Rechner signifikant bemerkbar.