Grundlagen der Rechnerarchitektur (GRA)

6. Speicherhierarchie

Prof. Dr. Marco Platzner

Inhaltsverzeichnis

6.1 Einführung

- Ein Speichersystem muss Speicher zur Verfügung stellen
  - soviel Speicher wie möglich
  - der Speicher soll so schnell wie möglich sein
  - der Speicher soll so günstig wie möglich sein

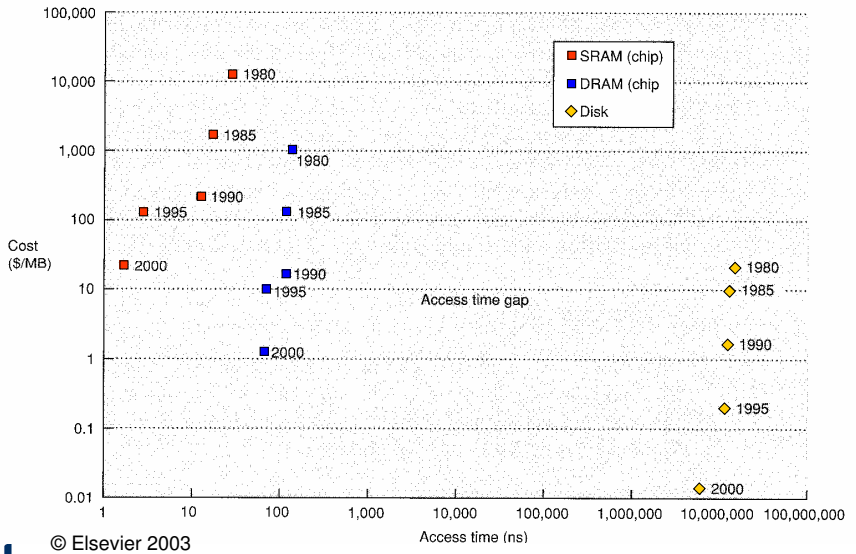
- Typische Werte im Jahr 2004

Speichertechnologie	Zugriffszeit	Kosten pro GByte
SRAM	0.5 – 5 ns	$4'000 – $10'000
DRAM	50 – 70 ns	$100 – $200
Festplatte	5 – 20 ms	$0.5 – $2

- Moderne Speichersysteme versuchen, dem Benutzer eine praktisch unbegrenzte Menge von sehr schnellem Speicher kostengünstig zur Verfügung zu stellen.

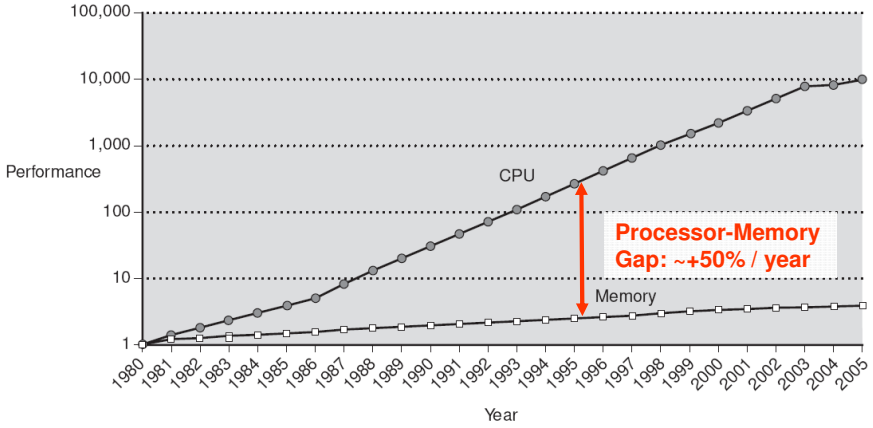
Technologische Trends

Entwicklung der Kosten und der Zugriffszeit von SRAM, DRAM und Festplatten



Technologische Trends

Entwicklung der Performance von CPUs und DRAM relativ zur Performance von DRAM 1980

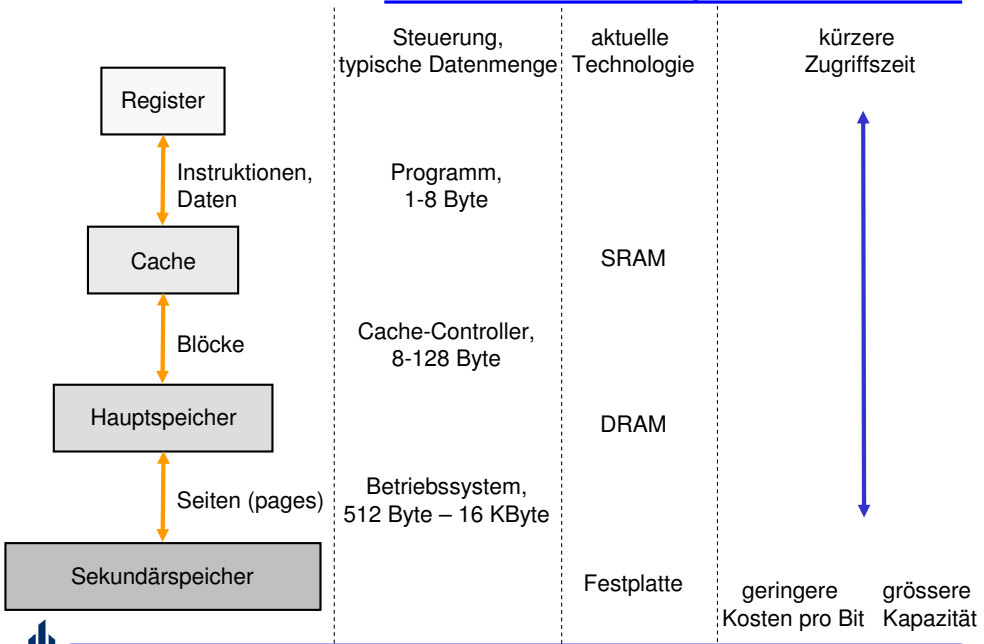


DRAM: +7% / year
CPU: +35% / year 1980-86, +55% / year 1986-2003

Das Lokalitäts-Prinzip

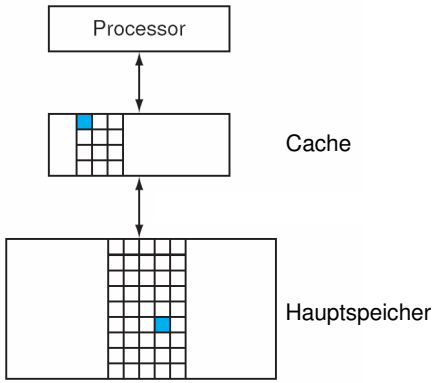
- Beobachtung: Programme greifen in einem kleinen Zeitintervall auf einen relativ kleinen Teil des Adressraums zu. Das gilt sowohl für Instruktionen als auch für Daten.
- Zeitliche Lokalität
  - Wenn ein Zugriff auf eine Adresse erfolgt, wird auf diese Adresse bald wieder zugegriffen.
- Räumliche Lokalität
  - Wenn ein Zugriff auf eine Adresse erfolgt, werden bald Zugriffe auf in der Nähe liegende Adressen erfolgen.
- Aufgrund der Lokalität kann man Speichersysteme hierarchisch aufbauen
  - wenig schnellen und teuren Speicher
  - viel langsamen und billigen Speicher

Übersicht - Speicherhierarchie



Speicherhierarchie

- Funktionsprinzipien
  - Daten werden in bestimmten Einheiten zwischen den Ebenen der Speicherhierarchie übertragen.
  - Daten werden nur zwischen benachbarten Ebenen übertragen.
  - Ist eine Dateneinheit auf einer Ebene vorhanden, muss sie auch auf den tieferen Ebenen vorhanden sein.



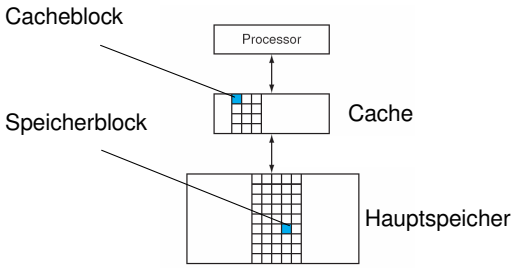
Terminologie

- Wird auf eine Ebene zugegriffen und die gewünschten Daten befinden sich dort, ist der Zugriff erfolgreich und es kommt zu einem hit (Treffer).
  - Die hit-rate ist der Anteil der Speicherzugriffe, die erfolgreich sind. Die hit-rate ist ein Leistungsmass für die jeweilige Ebene der Speicherhierarchie.
  - Die hit-time ist die Zeit für einen erfolgreichen Zugriff und beinhaltet die Zeit, um festzustellen, ob ein hit vorliegt, plus die Zeit für den eigentlichen Datenzugriff.
- Befinden sich die gewünschten Daten nicht dort, kommt es zu einem miss (Fehlzugriff) und die Daten müssen erst aus der unteren Ebene geholt werden.
  - Die miss-rate ist der Anteil der Speicherzugriffe, die nicht erfolgreich sind.

    → miss-rate = 1 – hit-rate
  - Die miss-penalty (Miss-Strafe) ist die Zeit für das Bereitstellen der Daten aus der unteren Ebene.
- mittlere Zugriffszeit = hit-time + miss-rate * miss-penalty

6.2 Caches

- Speicherhierarchie: Cache ↔ Hauptspeicher



- Wichtige Fragen
  - Auf welchen Cacheblock wird ein Speicherblock abgebildet?
  - Wie stellt man fest, ob ein Datum im Cache ist, und falls ja, in welchem Cacheblock?
  - Welcher Cacheblock wird bei einem Cache-Miss ersetzt?
  - Was geschieht beim Schreiben eines Datums?

Cache mit Direkter Abbildung

- Beim Cache mit direkter Abbildung (direct mapped cache) kann jeder Speicherblock nur an einer Stelle im Cache stehen.
  - einfachste Abbildung:

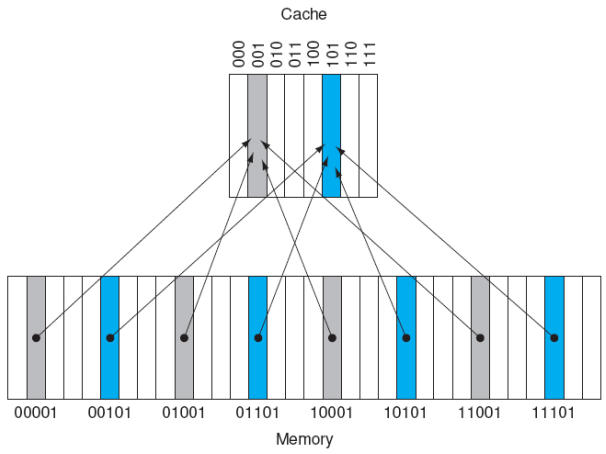
    Cacheblockadresse = Speicherblockadresse *mod* (Anzahl Cacheblöcke)
  - Die Cacheblockadresse wird auch als Index bezeichnet.
  - Typischerweise wählt man für die Anzahl der Cacheblöcke eine 2er-Potenz. Dadurch lässt sich die *mod* Operation besonders einfach realisieren.
    → für *mod* (2^n) muss man nur die n niederwertigsten Bits der Adresse nehmen

Cache mit Direkter Abbildung

Beispiel

8 Cacheblöcke; ein Block besteht aus einem Wort

→ Cacheblockadresse = Speicherblockadresse *mod* 8
= Speicherblockadresse[2:0]



Cache-Struktur

- Cache Tag
  - Der Index (Cacheblockadresse) kann nicht eindeutig einer Adresse zugeordnet werden.
  - Deshalb verwendet man die restlichen Bits der Adresse als Tag. Im Cache wird zu jedem Datum auch der Tag der gespeicherten Adresse abgelegt.
  - Durch Vergleich der Tags der gesuchten Adresse und des Cacheeintrages kann man feststellen, ob sich das gesuchte Datum im Cache befindet.
- Valid Bit
  - Am Anfang (und nach jedem Leeren des Caches) enthält der Cache keine gültigen Daten.
  - Jeder Cacheeintrag bekommt ein valid bit (Gültigkeitsbit), das gesetzt wird, wenn der Cacheeintrag gültig ist.

Beispiel

Cachezugriffe

5 Bit Adressen, Cache mit direkter Abbildung, 8 Cacheblöcke, ein Block besteht aus einem Wort → 32 Speicherblöcke

Zeit ↓

	Speicheradresse	Hit / Miss	Cacheblock
(1)	$22_{10} = 10110$	miss	10110 *mod* 8 = 110
(2)	$26_{10} = 11010$	miss	11010 *mod* 8 = 010
(3)	$22_{10} = 10110$	hit	10110 *mod* 8 = 110
(4)	$26_{10} = 11010$	hit	11010 *mod* 8 = 010
(5)	$16_{10} = 10000$	miss	10000 *mod* 8 = 000
(6)	$3_{10} = 00011$	miss	00011 *mod* 8 = 011
(7)	$16_{10} = 10000$	hit	10000 *mod* 8 = 000
(8)	$18_{10} = 10010$	miss	10010 *mod* 8 = 010

Beispiel

Cachezugriffe

vor (1)

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

nach (1)

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (2)

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (5)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (6)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

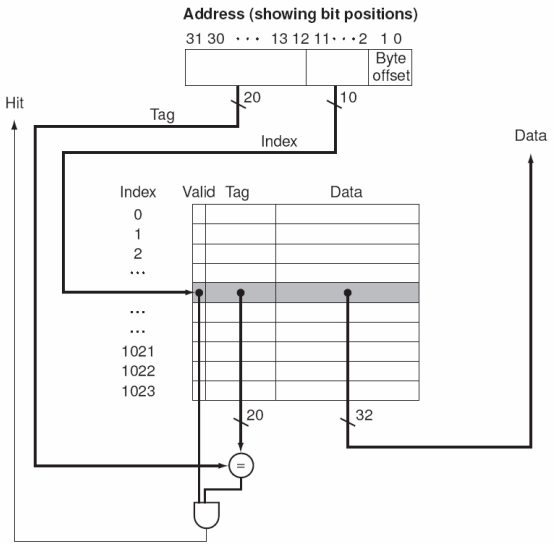
nach (8)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Beispiel

Cache-Struktur

4 KByte Cache mit direkter Abbildung
32-bit Byteadressen,
1024 Cacheblöcke,
ein Block besteht aus einem Wort



Wie viele Speicherbits braucht man für die Implementierung eines solchen 4 KByte (=32 KBit) Cache?

→ $2^{10}*(32+20+1) = 53$ KBit !

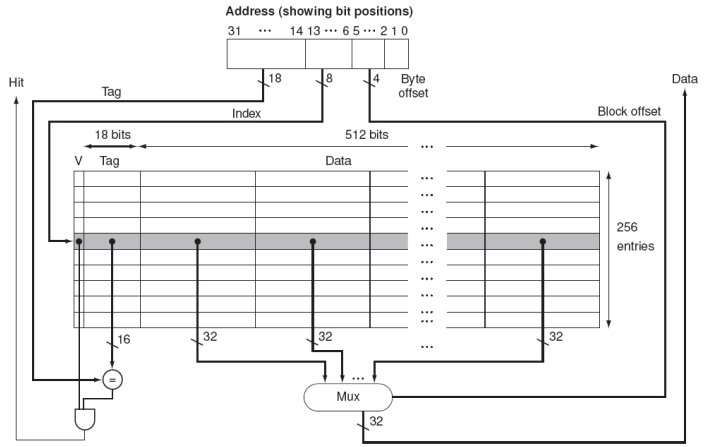
data tag valid bit

Cache mit Direkter Abbildung

- Um auch die örtliche Lokalität zu nutzen, wählt man grössere Blöcke.
  - Jeder Block hat einen Tag. Die Worte im Block haben aufsteigende Adressen.
  - Die l-Bit Adresse wird aufgeteilt in einen Tag, einen n-Bit Index und einen m-Bit Blockoffset. Der Blockoffset wählt unter den Worten eines Blockes aus.

Beispiel

l=32
n=8
m=4



Cache mit Direkter Abbildung

- Zusammenhänge

Gegeben: l Bit Adresse (Byteadressen)
Cache mit 2^n Blöcken
Blöcke mit 2^m Worten

Adressaufteilung:

Byteoffset	**Address**[1:0]
Blockoffset	**Address**[m+1:2]
Index	**Address**[m+n+1: m+2]
Tag	**Address**[l-1: m+n+2]

Cache-"Nutzdaten": $2^n * 2^{m+5}$ [bit]

Gesamte Cachegrösse: $2^n * (2^{m+5} + l - m - n - 1)$ [bit]

Cacheindex: $\left\lfloor \frac{\mathbf{Address}_{10}}{2^{m+2}} \right\rfloor \bmod 2^n = \mathbf{Address}[m+n+1:m+2]$

Cache mit Direkter Abbildung

- Mehrere Worte pro Block (multiword cache block)
  - Vorteile:
    - Durch die Nutzung örtlicher Lokalität sinkt die miss-rate.
    - Effizientere Speicherung, dh. weniger Overhead durch Tags und Valid Bits.
  - Nachteile:
    - Mit wachsender Blockgrösse gibt es immer weniger Cacheblöcke (bei konstanter Cachegrösse). Dadurch werden Blöcke immer öfter ersetzt, noch bevor die Lokalität gut ausgenutzt werden kann. Als Konsequenz steigt die miss-rate wieder.
    - Die miss-penalty wird grösser, da bei einem Cache-Miss mehr Daten aus dem Speicher geladen werden müssen.

- Weitere Massnahmen zur Reduktion der miss-rate
  - grösserer Cache
  - assoziativer Cache

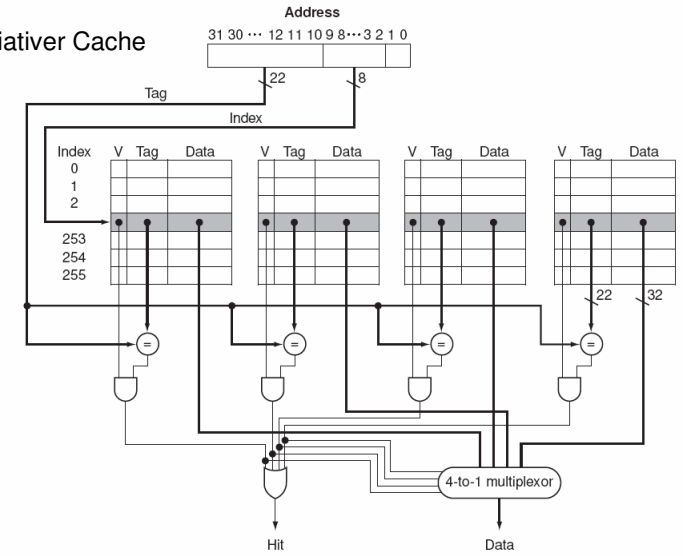
Assoziativer Cache

- Ein assoziativer Cache hat für jeden Cacheindex k Einträge (Blöcke).
  - Die Menge von Blöcken, die mit einem Indexwert referenziert wird, nennt man auch Satz (set).
  - Der Index der Adresse selektiert den Satz, der Tag der Adresse wird parallel mit allen k Tags der Cacheblöcke in diesem Satz verglichen. (Man kann sich auch vorstellen, dass k Caches parallel arbeiten.)
  - bei 2^n Cacheblöcken:
    - $k = 1$: Cache mit direkter Abbildung
    - $k > 1$: k-fach satzassoziativer Cache (k-way set associative cache), teilassoziativer Cache
    - $k = 2^n$: voll assoziativer Cache (fully associative cache)
  - Vorteil
    - miss rate wird reduziert
  - Nachteile
    - hit time wird etwas vergrössert
    - bei grossem k wird der Hardwareaufwand beträchtlich

Beispiel

Assoziativer Cache

l=32, n=10, m=0
4-fach satzassoziativer Cache



Beispiel

Assoziativer Cache

n=3

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

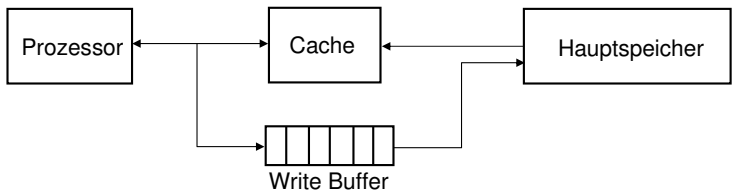
Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Ersetzungsstrategien

- Welchen Block "wirft" man aus dem Cache, wenn ein neuer Block geladen werden muss und keine freien Cacheblöcke vorhanden sind?
- Cache mit direkter Abbildung
  - keine Auswahl, da jeder Speicherblock auf genau einen Cacheblock abgebildet wird
- Assoziativer Cache
  - zufällige Auswahl eines Blocks
    - einfach in Hardware zu implementieren
  - LRU (least recently used): der am längsten nicht benutzte Block wird ersetzt
    - die Zugriffszeiten auf die Blöcke müssen gespeichert werden
    - aufwendig in Hardware, wird typischerweise bis k=4 gemacht

Schreibtechniken

- Durchgängiges Schreiben (write-through)
  - Beim Schreiben wird das Wort in den Cache und gleichzeitig in den Hauptspeicher geschrieben. Dadurch sind Hauptspeicher und Cache immer konsistent.
  - Bei einem write-miss wird meist zuerst der Block in den Cache geladen, und dann das Wort in den Cache und gleichzeitig in den Hauptspeicher geschrieben.
  - Um den Prozessor beim Schreiben nicht durch den langsamen Hauptspeicher zu bremsen, wird ein Pufferspeicher (write buffer) verwendet.
    - Ist die Rate, mit der der Prozessor Schreibinstruktionen ausführt, kleiner als die mittlere DRAM-Schreibzykluszeit, funktioniert write-through gut.
    - Wenn nicht, füllt sich der Pufferspeicher und der Prozessor muss anhalten und warten, bis im Pufferspeicher wieder Platz ist.



Schreibtechniken

- Zurückkopieren (write-back)
  - Beim Schreiben wird das Wort nur in den Cache geschrieben. Dadurch werden Hauptspeicher und Cache inkonsistent.
  - Erst wenn der Cacheblock ersetzt werden muss, wird er in den Hauptspeicher zurückgeschrieben.
    - Jeder Cacheblock bekommt ein dirty bit. Das dirty bit wird bei jedem Schreibzugriff auf den Block gesetzt und zeigt damit an, ob der Block beim Ersetzen zurückkopiert werden muss.
  - Vorteil
    - höhere Performance als write-through, wenn die Schreibrate hoch ist
  - Nachteil
    - aufwendigere Steuerung als write-through

Leistungsberechnungen

- Zugriff auf den Cache
  - Bei einem cache hit kann das Wort in einem Taktzyklus gelesen/geschrieben werden (Annahme: hit time ≤ T).
  - Bei einem cache miss initiiert der cache controller das Holen der Daten aus dem Hauptspeicher. Ein cache miss führt zu einem Anhalten des Prozessors (memory stall).

- Die Ausführungszeit eines Programms kann unterteilt werden in

CPU execution cyles:	Taktzyklen, in denen der Prozessor die Instruktionen abarbeitet
memory stall cyles:	Taktzyklen, in denen der Prozessor auf den Speicher warten muss

$$T_{exe} = (\text{CPU execution cycles} + \text{memory stall cycles}) * T \overset{!}{=} I_c * CPI * T$$

→ Der Performanceverlust durch memory stalls schlägt sich in einem vergrösserten CPI-Wert nieder.

Leistungsberechnungen

- Bestimmung der memory stall cyles durch getrennte Betrachtung von read (load) und write (store) Instruktionen

  memory stall cycles = read stall cycles + write stall cycles

  read stall cyles = reads/program * read miss-rate * read miss-penalty

  write stall cyles = writes/program * write miss-rate * write miss-penalty + write buffer stall cycles

  (Annahme: Cache mit durchgängigem Schreiben)

- Vereinfachte Betrachtung
  - write buffer stalls werden als vernachlässigbar angenommen, read miss-penalty = write miss-penalty
  - Dadurch lassen sich reads und writes zu memory accesses mit einer gemeinsamen miss-rate und einer miss-penalty zusammenfassen:

    memory stall cycles = memory_accesses/program * miss-rate * miss-penalty

Beispiel

Leistungsberechnungen

Annahmen:
- 36% aller Instruktionen eines Programms sind Speicherzugriffe (load, store)
- Ohne memory stalls hat der Prozessor einen CPI-Wert von 2 Taktzyklen.
- Der Prozessor hat getrennte Instruktions- und Datencaches. Beide Caches haben eine miss-penalty von 100 Taktzyklen.
- Die miss-rate für Instruktionen ist 2%; die miss-rate für Daten ist 4%.

Um wieviel schneller wäre der Prozessor mit einem perfekten Cache? (ohne Cache-Misses)

- memory stall cycles durch Instruktionszugriffe = I_C * 0,02 * 100 = 2*I_C
- memory stall cycles durch Datenzugriffe = I_C * 0,36 * 0,04 * 100 = 1,44*I_C

  → gesamte memory stall cycles = 3,44*I_C

- $CPI_{no\ stalls}$ = 2
- $CPI_{with\ stalls}$ = 2+3,44 = 5,44

  → Performancegewinn durch perfekten Cache = 5,44/2 = 2,72

Leistungsberechnungen

Beispiel

Annahmen:

- gleiches Beispiel wie auf vorhergehender Seite, aber
- die Taktfrequenz des Prozessors wird verdoppelt, während die Zugriffszeit auf den Speicher unverändert bleibt

Um wieviel steigt die Prozessorperformance?

- neue miss-penalty = 200 Taktzyklen
- memory stall cycles durch Instruktionszugriffe = I_C * 0,02 * 200 = 4*I_C
- memory stall cyles durch Datenzugriffe = I_C * 0,36 * 0,04 * 200 = 2,88*I_C

  → gesamte memory stall cycles = 6,88*I_C

- $CPI_{clock\ period\ T}$ = 5,44
- $CPI_{clock\ period\ T/2}$ = 8,88

→ Performancegewinn durch doppelte Taktfrequenz = (5,44*T)/(8,88*T/2) = 1,23

Speicherorganisation

- Ziel der Speicherorganisation ist es, die miss-penalty zu minimieren
  - Prozessor/Cache und Speicher sind über den Speicherbus verbunden. Die Taktrate des Speicherbusses ist typischerweise um eine Grössenordnung kleiner als die des Prozessors.
  - Die Organisation des Speichers und die Blockgrösse müssen aufeinander abgestimmt sein.

- Der Zugriff auf DRAM erfolgt in mehreren Schritten
  (sehr grobes Modell, hier für Lesezugriffe)

	Annahme
1. Adresse senden	1 Speicherbuszyklus
2. Speicherzugriff	15 Speicherbuszyklen
3. Wort transferieren	1 Speicherbuszyklus

  - Annahme: Bei der Übertragung eines Blocks (aufeinanderfolgende Adressen) wird nur die Startadresse und die Wortanzahl gesendet.

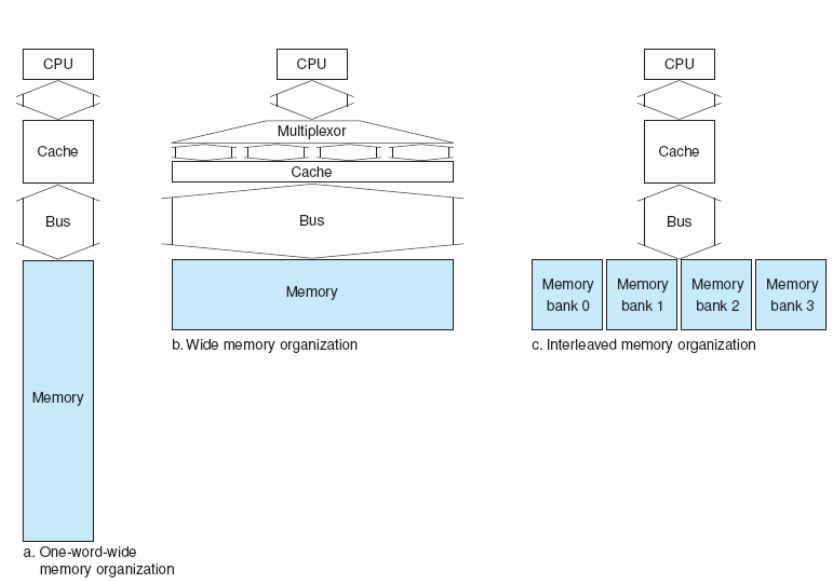
Speicherorganisation

- einfache Organisation (nächste Seite, Variante a.)
  - der Speicherbus und der Speicher haben die Breite eines Wortes
  - alle Worte eines Blocks werden sequentiell übertragen, dadurch grosse miss-penalty
  - einfache Realisierung
  - Bsp.: miss-penalty bei Blockgrösse 4 Worte = 1+4*15+4*1=65 Zyklen

- breite Organisation (nächste Seite, Variante b.)
  - der Speicherbus und der Speicher werden so breit wie der Block gemacht
  - nur eine Übertragung pro Block, dadurch reduzierte miss-penalty
  - teure Realisierung; der Multiplexer zwischen Prozessor und Cache erhöht die Zugriffszeit auf den Cache
  - Bsp.: miss-penalty bei Blockgrösse 4 Worte = 1+15+1=17 Zyklen

- verschränkte (interleaved) Organisation (nächste Seite, Variante c.)
  - Die Speicherbausteine sind in Bänken organisiert. Die Bänke und der Speicherbus sind ein Wort breit. Die Anzahl der Bänke entspricht der Anzahl der Worte pro Block.
  - die Speicherzugriffszeit muss nur einmal abgewartet werden
  - Bsp.: miss-penalty bei Blockgrösse 4 Worte = 1+15+4*1=20 Zyklen

Speicherorganisation



Speicherorganisation

Beispiel

- Speicherbus ist 4 mal langsamer getaktet als der Prozessor:
  Adresse senden: 4 (Prozessor)Zyklen
  Speicherzugriff: 60 Zyklen
  Wort transferieren: 4 Zyklen
- 1,2 Speicherzugriffe pro Instruktion, CPI ohne cache misses = 2
- miss-rate bei Blockgrösse 1 = 3%, miss-rate bei Blockgrösse 2 = 2%, miss-rate bei Blockgrösse 4 = 1%
- Performance verschiedener Organisationen des Speichersystems
  - 32 Bit Bus, 32 Bit Speicher, 1 Wort/Block, einfach
    → CPI = 2 + (1,2 * 0,03 * 68) = 4,448
  - 32 Bit Bus, 32 Bit Speicher, 2 Worte/Block, einfach
    → CPI = 2 + (1,2 * 0,02 * (4+2*60+2*4)) = 5,168
  - 32 Bit Bus, 32 Bit Speicher, 2 Worte/Block, 2-fach interleaved
    → CPI = 2 + (1,2 * 0,02 * (4+60+2*4)) = 2,728
  - 64 Bit Bus, 64 Bit Speicher, 2 Worte/Block, breit
    → CPI = 2 + (1,2 * 0,02 * (4+60+4)) = 2,632
  - 32 Bit Bus, 32 Bit Speicher, 4 Worte/Block, 4-fach interleaved
    → CPI = 2 + (1,2 * 0,01 * (4+60+4*4)) = 2,864

Weitere Massnahmen zur Cache-Optimierung

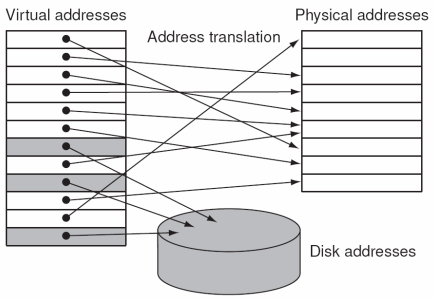
- Reduktion der miss-rate
  - Cacheorganisation
    - grössere Blöcke
    - grössere Caches
    - höhere Assoziativität
  - Compileroptimierungen
    - Instruktionen: Umordnen der Instruktionen, um instruction misses zu reduzieren
    - Daten: array merging, loop interchange, loop fusion, blocking, ...
- Reduktion der miss-penalty
  - Multi-level Cache
    - kleiner 1st level cache optimiert auf hit-time (geringe Assoziativität, kleine Blöcke)
    - grosser 2nd level cache optimiert auf miss-rate (höhere Assoziativität, grosse Blöcke)
  - Victim Cache
    - zusätzlicher "Cache", der kürzlich ersetzte Cacheblöcke ("victims") speichert

Weitere Massnahmen zur Cache-Optimierung

- Reduktion der miss-penalty und der miss-rate
  - Blöcke werden auf Verdacht in Register, Caches oder Prefetch-Puffer geladen
  - macht nur Sinn, wenn das Speichersystem freie Bandbreite zur Verfügung hat
  - Hardware Prefetching
    - Prefetching von Instruktionen,
      Bsp.: Alpha 21064 lädt 2 Blöcke bei einem instruction miss, der zweite Block kommt in einen separaten Puffer; beim nächsten instruction miss wird zuerst in diesem Puffer nachgesehen
    - Prefetching von Daten:
      Bsp.: stream buffer, die dauernd mit den Blöcken der nächsten Adressen nachgeladen werden
  - Software Prefetching
    - der Compiler fügt spezielle prefetch-Instruktionen ein

6.3 Virtueller Speicher

- Speicherhierarchie: Hauptspeicher ↔ Sekundärspeicher (Disk)
  - Der Prozessor arbeitet mit virtuellen Adressen, die durch eine Kombination aus Hardware (memory management unit, MMU) und Software (Betriebssystem) in physikalische Adressen abgebildet werden. Die physikalischen Adressen werden benutzt, um auf den Hauptspeicher zuzugreifen.
- Terminologie
  - Ein Block virtuellen und physikalischen Speichers wird Seite (page) genannt.
  - Ein miss beim Zugriff auf den Hauptspeicher heisst Seitenfehler (page fault).

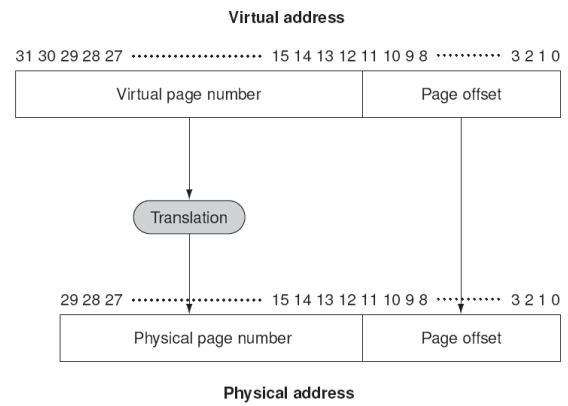


Virtueller Speicher

- Motivation für virtuellen Speicher
  - Einem Programm kann mehr Speicher zur Verfügung gestellt werden, als physikalisch vorhanden ist (heute nicht mehr so relevant).
  - Programme können schneller starten, da nicht erst das ganze Programm in den Speicher geladen werden muss.
  - Viele Betriebssysteme erlauben das "gleichzeitige" Ausführen mehrerer Prozesse:
    - deren gesamter Speicherbedarf kann wesentlich grösser sein als der physikalisch vorhandene Speicher
    - jeder Prozess kann seinen eigenen Adressraum haben und getrennt übersetzt werden
    - die Prozesse können voreinander geschützt werden (protection)
  - Das hier behandelte paging (Blöcke konstanter Grösse) ist eine wichtige Technik, um virtuellen Speicher zu implementieren. Daneben gibt es weitere Techniken, zB. segmentation (Blöcke variabler Länge), oder paged segmentation.

Adressabbildung

- Eine virtuelle Adresse wird aufgeteilt in
  - eine virtuelle Seitennummer (virtual page number), die in eine physikalische Seitennummer abgebildet wird
  - einen Seitenoffset (page offset), der die Adresse innerhalb einer page darstellt
    - die Anzahl der Bits im page offset bestimmt die Grösse einer page



Page Faults

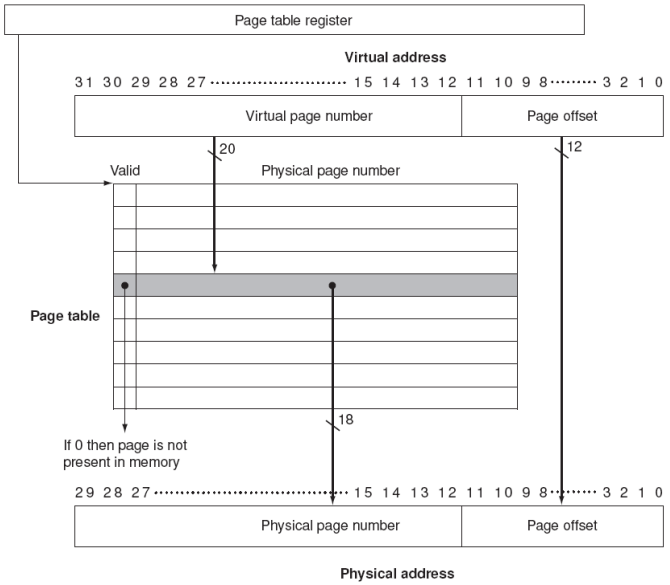
- Ein page fault bedingt eine penalty von Millionen von Taktzyklen!
  - Pages werden relativ gross gemacht, um die grosse Zugriffszeit auf den Sekundärspeicher zu amortisieren (typisch: 4 KByte -16 KByte).
  - Die Anzahl der page faults muss minimiert werden. Deshalb werden die pages im Hauptspeicher voll assoziativ platziert.
  - Page faults und die Seitenersetzung werden in Software (Betriebssystem) behandelt. Für die Seitenersetzung werden zT. komplexe Algorithmen eingesetzt. Die Laufzeit dieser Algorithmen spielt im Vergleich zu den Kosten eines page faults eine geringe Rolle.
  - Durchgängiges Schreiben funktioniert wegen der grossen Zugriffszeit auf den Sekundärspeicher nicht. Deshalb werden Zurückkopier-Techniken verwendet (write back, copy back).

Finden von Pages

- Das Finden von pages wird durch die page table realisiert.
  - Aufgrund der grossen Anzahl von pages ist ein Suchen (volle Assoziativität) nicht praktikabel. Stattdessen wird eine page table verwendet, die für jede virtuelle Seitennummer die dazugehörige physikalische Seitennummer speichert.
  - Ein valid bit zeigt zusätzlich an, ob sich die gesuchte page im Speicher befindet.
  - Jeder Prozess hat seine eigene page table.
  - Bsp.: 32 Bit virtuelle Adresse, 4 KByte page size, 4 Byte pro Eintrag
    → Grösse der page table = 2^{22} = 4 MB

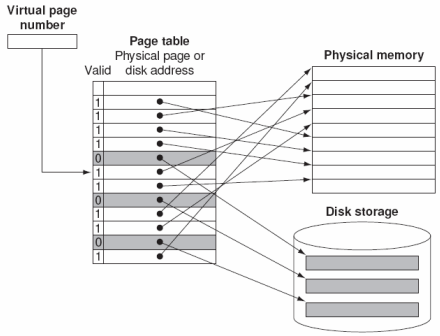
    Es existieren verschiedene Techniken, um den Grössenbedarf der page table zu reduzieren.
  - Die page table ist selbst im Speicher abgelegt. Der Beginn der page table wird in einem speziellen Register, dem page table register, gespeichert. Dieses Register ist Teil des Prozesszustandes (Kontext), der bei einem Prozesswechsel gesichert werden muss.

Adressabbildung mit Page Table



Page Faults

- Erkennung und Behandlung
  - Das entsprechende valid bit in der page table ist auf 0 gesetzt.
  - Ein page fault löst eine exception aus. Das Betriebssystem übernimmt die Kontrolle und behandelt den page fault. Die gesuchte Seite wird aus dem Sekundärspeicher nachgeladen.
  - Das Betriebssystem speichert notwendigerweise die Adressen der pages im Sekundärspeicher (Festplatte). Eine Möglichkeit ist, diese Diskadressen auch in die page table mit aufzunehmen.



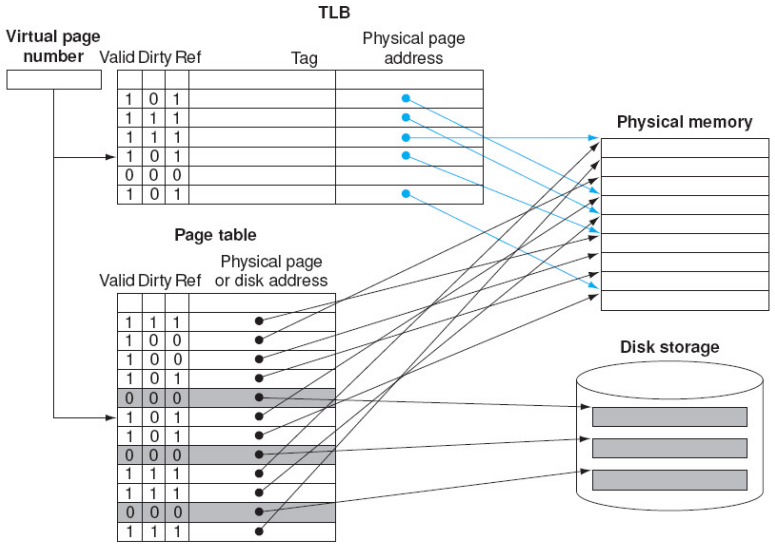
Page Faults

- Ersetzen von pages
  - Wenn bei einem page fault keine freie page im Speicher mehr vorhanden ist, muss eine Ersetzung stattfinden, dh. eine page wird vom Hauptspeicher in den Sekundärspeicher ausgelagert.
  - Meist wird die LRU (least recently used) Ersetzungsstrategie verwendet.
    - LRU ist sehr aufwendig zu implementieren, da bei jedem Speicherzugriff eine Datenstruktur (Liste mit Zugriffszeiten) modifiziert werden müsste.
    - Oft wird LRU deshalb approximiert. Eine Variante ist, die Einträge in die page table mit einem reference bit zu erweitern. Wenn auf die page zugegriffen wird, wird das reference bit gesetzt. Von Zeit zu Zeit löscht das Betriebssystem alle reference bits.
  - Die Einträge in die page table werden mit einem dirty bit erweitert, das beim Schreiben in die page gesetzt wird und dadurch angibt, ob die page beim Ersetzen in den Sekundärspeicher zurückkopiert werden muss.
  - Typischerweise legt das Betriebssystem bei der Erzeugung eines Prozesses Platz auf der Disk an für das spätere, eventuelle Auslagern (swap space).

TLB

- Beschleunigung der Adressumsetzung
  - Da die page table im Hauptspeicher steht, würde man pro Speicherzugriff eigentlich zweimal auf den Speicher zugreifen: einmal auf die page table, um die physikalische Adresse festzustellen und dann ein zweites mal, um die Daten zu lesen bzw. zu schreiben.
  - Die Zugriffe auf die page table zeigen zeitliche Lokalität. Wenn eine virtuelle Seitenadresse abgebildet wird, wird sie bald wieder abgebildet (da die Zugriffe auf die Adressen zeitliche und räumliche Lokalität zeigen).
- Moderne Prozessoren besitzen einen translation-lookaside buffer (TLB)
  - Der TLB ist ein Cache für die Adressumsetzungen der page table.
  - Bei einem TLB miss muss festgestellt werden, ob sich die gesuchte page im Hauptspeicher befindet. Falls ja, wird der entsprechende page table Eintrag in den TLB geladen. Falls nein, handelt es sich um einen page fault und die Kontrolle wird dem Betriebssystem übergeben.

TLB



Schutzmechanismen

- Motivation für Schutzmechanismen
  - Prozesse sollen daran gehindert werden können, Speicherbereiche anderer Prozesse (oder gar des Betriebssystems) zu lesen oder zu verändern.
- Implementierung von Schutzmechanismen
  - In der page table werden zusätzlich Lese- und Schreibrechte abgelegt.
  - Nur das Betriebssystem darf die page table Einträge ändern.
- Hardwareunterstützung für Schutzmechanismen
  - Der Prozessor hat zwei Betriebsmodi, einen für Benutzerprozesse (user process) und einen für Betriebssystemprozesse (kernel process, supervisor process).
  - Manche Operationen können nur im kernel mode ausgeführt werden, zB. Schreiben des page table pointers und des TLB.
  - Für das Umschalten zwischen user und kernel mode gibt es spezielle Befehle (`syscall` und `eret` Instruktionen in MIPS).

Vergleich Cache vs. Virtueller Speicher

- Cache ↔ Hauptspeicher
  - Die Cacheblöcke gehören (ia.) zum gleichen Programm.
  - Cache misses werden durch die Hardware bearbeitet.
  - Die Grösse des Cache ist unabhängig vom Adressraum des Prozessors.
  - Der Cache wird exklusiv für die Speicherhierarchie verwendet.
- Hauptspeicher ↔ Sekundärspeicher
  - Die pages gehören verschiedenen Prozessen, die "gleichzeitig" ausgeführt werden. Jeder Prozess besitzt eine Menge von pages.
  - Page faults werden durch die Software bearbeitet.
  - Die Grösse des virtuellen Speichers wird durch den Adressraum des Prozessors bestimmt.
  - Der sekundäre Speicher wird nicht nur für die Speicherhierarchie, sondern auch zur Datenhaltung (Filesystem) verwendet.

Zusammenfassung – 4 Wesentliche Fragen

1. Wo wird ein Block platziert?
   - direkt abgebildet
   - teilassoziativ
   - vollassoziativ
2. Wie wird ein Block gefunden?
   - direkt abgebildet: Index
   - teilassoziativ: Index → Suche unter den Blöcken (paralleler Vergleich)
   - vollassoziativ: Suche unter allen Blöcken bzw. Tabelle mit Abbildungen
3. Welcher Block wird bei einem Miss ersetzt?
   - zufällig
   - LRU (least recently used)
4. Was geschieht beim Schreiben?
   - durchgängiges Schreiben
   - zurückkopieren

- Typische Werte für Computersysteme im Jahr 2004

Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	4000–250,000	16,000–250,000	16–512
Total size in kilobytes	16–64	500–8000	250,000–1,000,000,000	0.25–16
Block size in bytes	32–64	32–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

L1 = 1st level cache
L2 = 2nd level cache

- Vergleich Intel Pentium P4 und AMD Opteron
  - P4 ist eine 32 Bit Architektur
  - Opteron kann IA-32 Programme ausführen, hat aber zusätzlich einen 64-Bit Modus (PC und weitere Register sind 64 Bit breit)
  - Virtueller Speicher und TLB:

Characteristic	Intel Pentium P4	AMD Opteron
Virtual address	32 bits	48 bits
Physical address	36 bits	40 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data Both are four-way set associative Both use pseudo-LRU replacement Both have 128 entries TLB misses handled in hardware	2 TLBs for instructions and 2 TLBs for data Both L1 TLBs fully associative, LRU replacement Both L2 TLBs are four-way set associativity, round-robin LRU Both L1 TLBs have 40 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

- Vergleich Intel Pentium P4 und AMD Opteron
  - beide Prozessoren haben die sekundären Caches (L2) am Chip
  - Cache Organisation:

Characteristic	Intel Pentium P4	AMD Opteron
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	8 KB for data, 96 KB trace cache for RISC instructions (12K RISC operations)	64 KB each for instructions/data
L1 cache associativity	4-way set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-through	Write-back
L2 cache organization	Unified (instruction and data)	Unified (instruction and data)
L2 cache size	512 KB	1024 KB (1 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	128 bytes	64 bytes
L2 write policy	Write-back	Write-back