Calculatoare Numerice (2)

- Cursul 1 -

Sistemul de memorie (1)

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

HEY, PAL, I'VE FINALLY UNDERSTOOD HOW MY CPU WORKS AND WHY IT CRASHES SOMETIMES WHEN I TRY TO RUN SOME NASTY CODE!
OH, YOU THINK SO?
OH, YEAH, I'VE JUST PAST MY "CN 1st" COURSE
HMM... AND YOU THINK THAT'S ALL?
WHAT DO YOU MEAN ....
YOU KNOW SO LITTLE... OH, WAIT... THERE'S MORE!
MY ULTIMATE IS READY... ... THERE IT COMES... 'CN 2'!!
...
TO BE CONTINUED...

Proiectarea sistemului de memorie

Probleme de proiectare – Vrem un circuit de memorie:

- Poate să țină pasul cu viteza de execuție a CPU
- Are destulă capacitate pentru program și date
- Ieftină, fiabilă și eficientă energetic

Tehnologia și organizarea memoriei principale a unui calculator

- SRAM (cache), DRAM (main), și flash (nonvolatile)
- Întrețesere & pipelining pentru a combate “memory wall”

Primele tehnologii de fabricație pentru memoria Read-Only

Cartele perforate , din anii 1700, războaie Jaquard, Babbage, IBM până în anii ‘80

Bandă perforată – programul rulat de Harvard Mk 1

IBM Balanced Capacitor ROS

Diode Matrix, EDSAC-2 µcode store

IBM Card Capacitor ROS

Primele tehnologii de fabricație pentru memoria Read-Write

Babbage, anii 1800: Cifre stocate pe roți mecanice

Tub Williams, Manchester Mark 1, 1947

Mercury Delay Line, Univac 1, 1951

Memorie regenerativă cu condensatoare pentru calculatorul Atanasoff-Berry și memorie pe tambur magnetic rotativ pentru IBM 650

Memorie pe miez de ferită - MIT Whirlwind

50mm

2mm

http://www.corememoryshield.com/report.html

Memoria pe miez de ferită

- A fost prima tehnologie de fabricație fiabilă pentru memoriile principale
  - Inventată de Forrester sfârșitul anilor 40/începutul anilor 50 la MIT pentru proiectul Whirlwind
- Biți stocați prin polarizarea magnetică a unor miezuri foarte mici de ferită țesute într-o matrice bidimensională de fire conductoare
- Pulsurile concomitente de curent pe conductorii X și Y pot scrie starea bitului de memorie și să citească starea originală (destructive read)

- Stocare robustă, non-volatilă
- Folosită la primele nave spațiale (de la Apollo la navetele spațiale)
- Inelele de ferită țesute de mână (25 de miliarde/an)
- Timp de acces ~ 1μs

Memorii semiconductoare

- Memoriile semiconductoare au început să fie competitive în anii 70
  - Intel a apărut pentru a exploata piața memoriilor semiconductoare
  - Primele memorii semiconductoare au fost RAM-urile Statice (SRAM). Structura internă a unei celule SRAM este similară cu aceea a unui latch (inversoare în anti-paralel).
- Primul RAM Dinamic (DRAM) comercial a fost chipul Intel 1103
  - 1Kbit de memorie pe un singur chip
  - Sarcina unui condensator este folosită pentru a memora un bit

Memoria semiconductoare a înlocuit rapid memoria pe miez de ferită în anii 70

One-Transistor Dynamic RAM [Dennard, IBM]

https://en.wikipedia.org/wiki/Dynamic_random-access_memory

DRAM Scaling

Structura modernă a unui DRAM

[Samsung, sub-70nm DRAM, 2004]

Structura Internă

Arhitectura DRAM

- Biții sunt stocați în matrici bidimensionale pe chip
- Chipurile moderne au în jur de 4-8 bancuri logice
- Fiecare banc logic este implementat fizic ca o matrice de biți

Încapsularea DRAM (Laptop-uri/Desktop-uri/Servere)

- DIMM (Dual Inline Memory Module) conține mai multe chipuri cu semnalele de ceas/control/adresă conectate în paralel (câteodată este nevoie de buffering pentru a duce semnalele la toate chip-urile)
- Pinii de date lucrează împreună pentru a returna un cuvânt întreg (de. ex., bus de date de 64 de biți cu patru chipuri de 16 biți)

Încapsularea DRAM, Dispozitive mobile

[Capsulă Apple A4 pe PCB]

[Capsulă Apple A11 în secțiune]

Two stacked DRAM die

Processor plus logic die

[Capsulă Apple A4 în secțiune, iFixit 2010]

[iPhone X PCB, 2017]

Memoria SRAM – celula de memorie

- Memorie cu stocare volatilă
- Structura internă de bistabil (flip-flop)
- Nu necesită cicli de refresh
- Viteză mai mare citire/scriere comparativ cu DRAM
- Costuri mai mari de fabricație

[Celulă (1 bit) de memorie SRAM]

https://en.wikipedia.org/wiki/Static_random-access_memory

Structura memoriei SRAM

Structura internă a unui chip SRAM $2^h \times g$ biți și simbolul lui echivalent în schema electrică.

Multiple-Chip SRAM

Opt chipuri SRAM 128K × 8 biți legate să formeze o memorie 256K × 32 biți

SRAM cu bus de date bidirecțional

Atunci când intrarea și ieșirea de date a unui chip SRAM sunt partajate sau conectate la un bus bidirecțional, ieșirea trebuie dezactivată în timpul operațiilor de scriere.

Memoria DRAM și ciclii de refresh

DRAM vs. SRAM – complexitatea unei celule de memorie

(a) DRAM cell (b) Typical SRAM cell

Celula de memorie DRAM conține un singur tranzistor și e mult mai simplu de fabricat decât analogul ei SRAM => memorii DRAM de capacitate mai mare și mai dense.

Ciclii și rata de refresh pentru memoria DRAM

Variația căderii de tensiune pe condensatorul unei celule DRAM după scrierea unui 1 logic si a mai multor operații de refresh.

Pierderea lățimii de bandă cu ciclii de refresh

O memorie DRAM de 256 Mb e organizată ca 32M × 8 intern și 16K × 16K intern. Rândurile trebuie reîmprospătate cel puțin la fiecare 50ms pentru a nu pierde datele; refresh-ul pentru o coloană durează 100ns. Cât % din lățimea totală de bandă este pierdută cu ciclii de refresh?

(a) SRAM block diagram

(b) SRAM read mechanism

Soluție

Refresh-ul pt toate 16K rânduri durează 16 × 1024 × 100 ns = 1.64 ms. Pierderea a 1.64 ms la fiecare 50 ms duce la 1.64/50 = 3.3% pierdere din lățimea totală de bandă.

Încapsularea DRAM

24-pin dual in-line package (DIP)

Legend:

A*i*	Address bit *i*
CAS	Column address strobe
D*j*	Data bit *j*
NC	No connection
OE	Output enable
RAS	Row address strobe
WE	Write enable

Capsulă tipică DRAM ce conține o memorie 16M × 4 [circa 1990]

556-pin FBGA package

Samsung 12GB DRAM [2019]

https://www.samsung.com/semiconductor/dram/lpddr4x/

Atingerea zidului memoriei

Densitatea și capacitatea memoriei au crescut odată cu puterea și complexitatea CPU, dar viteza memoriei nu a ținut pasul.

Trecerea prăpastiei de viteză CPU-Memorie

Idee: Citim mai multe date din memorie la fiecare acces

(a) Buffer and multiplexer
at the memory side

(a) Buffer and multiplexer
at the processor side

Două căi de a folosi o memorie cu lățime mare de bandă pentru a reduce diferența de viteză dintre procesor și memorie.

Memorie și b.a.

Latența memoriei poate fi dată și de alți factori, în afară de timpul de acces fizic.

- Translatarea adresei virtuale
- Compararea etichetelor pentru a determina rata hit/miss pentru cache

Memorie cache în bandă de asamblare

Întrețeserea memoriei

Memoria întrețesută e mai flexibilă decât memoria cu lățime de bandă mare, pentru că poate susține accese multiple independente în același timp.

Memoria ne-volatilă

Organizarea Read-Only Memory. Conținutul memoriei este afișat în dreapta.

Memoria Flash

180 nm Flash technology

Organizarea memoriei EEPROM sau Flash. Fiecare celulă conține un tranzistor MOS cu poartă flotantă.

https://en.wikipedia.org/wiki/Flash_memory

Nevoia unei ierarhii de memorie

Discrepanța în latență dintre CPU și memoria principală

- Operațile unui procesor sunt de ordinul nanosecundelor
- Accesele la memorie necesită timpi de ordinul zecilor sau sutelor de ns

Limitările lățimii de bandă pentru memorii reduc rata de execuție a instrucțiunilor

- Fiecare instrucțiune executată necesită cel puțin un acces la memorie
- Rezultă că performanța procesorului este redusă la câteva sute de MIPS
- O memorie rapidă poate reduce timpii de acces la date
- Cele mai rapide memorii sunt costisitoare și nu au capacitate mare.
- Două (sau trei) niveluri de cache sunt folosite, din această cauză

Ierarhia tipică a circuitelor de memorie

Capacity	Access latency	Cost per GB
100s B	ns	$Millions
10s KB	a few ns	$100s Ks
MBs	10s ns	$10s Ks
100s MB	100s ns	$1000s
10s GB	10s ms	$10s
TBs	min+	$1s

Speed gap

Numele și caracteristicile tipice pentru memorii în organizarea ierarhică

Tendințele prețurilor memoriilor

https://hblok.net/blog/posts/2017/12/17/historical-cost-of-computer-memory-and-storage-4/

Memorii de mare capacitate

În zilele noastre, memoria principală este imensă, totuși inadecvată pentru toate necesitățile

- Discurile magnetice furnizează capacități extinse pentru stocare și back-up
- Discurile optice și memoriile solid-state sunt alte opțiuni de stocare a datelor

Disk Memory 101

Elementele unui hard-disc și termenii principali.

Unități de disc

Timpul de acces pentru un disc

Cele trei componente ale timpului de acces la un disc. Discurile cu o viteză de rotație mai mare au timpi de acces mai buni, atât în medie cât și în cel mai rău caz.

Discuri magnetice

Caracteristicile a trei tipuri diferite de discuri magnetice (cca. 2003)

Manufacturer and Model Name	Seagate Barracuda 180	Hitachi DK23DA	IBM Microdrive
Application domain	Server	Laptop	Pocket device
Capacity	180 GB	40 GB	1 GB
Platters / Surfaces	12 / 24	2 / 4	1 / 2
Cylinders	24 247	33 067	7 167
Sectors per track, avg	604	591	140
Buffer size	16 MB	2 MB	1/8 MB
Seek time, min,avg,max	1, 8, 17 ms	3, 13, 25 ms	1, 12, 19 ms
Diameter	3.5″	2.5″	1.0″
Rotation speed, rpm	7 200	4 200	3 600
Typical power	14.1 W	2.3 W	0.8 W

Organizarea datelor pe disc

Înregistrarea magnetică a datelor pe piste și capul de citire/scriere.

	0	16	32	48	1	17	33	49	2	Track *i*
	30	46	62	15	31	47	0	16	32	Track *i* + 1
	60	13	29	45	61	14	30	46	62	Track *i* + 2
	27	43	59	12	28	44	60	13	29	Track *i* + 3

Numerotarea logică a sectoarelor pe mai multe piste adiacente.

Performanța discurilor

Timpul de căutare = $a + b(c - 1) + \beta(c - 1)^{1/2}$

Latență medie dată de rotire = (30 / rpm) s = (30 000 / rpm) ms

Arrival order of access requests:

A, B, C, D, E, F

Possible out-of-order reading:

C, F, D, E, B, A

Reducerea timpului de căutare și a latenței de rotire prin accesarea datelor în altă ordine.

Disk Caching

Aceeași idee ca și la caching-ul pentru procesoare: micșorarea latenței dintre memoria principală și disc

Discurile au memorii tampon în funcție de capacitate (de ordinul 10-100 MB)

Latența datorată rotației este eliminată; pot să încep de la orice sector

Am nevoie de energie pentru back-up pentru a nu pierde schimbările din memoria tampon

(ne trebuie oricum o rezervă de energie pentru retragerea capului de citire la căderea sursei de energie electrică)

Opțiuni de plasare a memoriei cache pentru discuri

În controllerul de disc:

Suntem afectați de latența magistralei de date și a controllerului în sine, chiar și pentru un cache hit

Mai aproape de CPU:

Reduce latența și permite o utilizare mai bună a spațiului

Soluții intermediare sau mixte

Disk Arrays & RAID

Necesitatea de memorii secundare (disc) de capacitate și productivitate mărită

Processor speed	RAM size	Disk I/O rate	Number of disks	Disk capacity	Number of disks
1 GIPS	1 GB	100 MB/s	1	100 GB	1
1 TIPS	1 TB	100 GB/s	1000	100 TB	100
1 PIPS	1 PB	100 TB/s	1 Million	100 PB	100 000
1 EIPS	1 EB	100 PB/s	1 Billion	100 EB	100 Million

1 RAM byte pentru fiecare IPS

1 I/O bit/sec pentru fiecare IPS

100 disk bytes pt. fiecare RAM byte

Legea lui Amdahl pentru un sistem bine echilibrat

Redundant Array of Independent Disks (RAID)

RAID0: Multiple disks for higher data rate; no redundancy

RAID1: Mirrored disks

RAID2: Error-correcting code

RAID3: Bit- or byte-level striping with parity/checksum disk

RAID4: Parity/checksum applied to sectors,not bits or bytes

RAID5: Parity/checksum distributed across several disks

RAID6: Parity and 2nd check distributed across several disks

Nivelurile 0-6 RAID, cu o vedere simplificată a organizării datelor.

Exemple de produse RAID

[HighPoint RocketStor 6618 Thunderbolt 3 DAS: 8-Bays, Up to 96 TB, 2.7 GB/s]

HADOOP.Big Data Rx8500/8600 250TB-Enterprise Cloud Storage Solution

Alte tipuri de medii de stocare

Discuri optice

Vedere simplificată a înregistrărilor și mecanismul de acces la date pentru un CD-ROM sau DVD-ROM.

Biblioteci automate de benzi pentru arhivare

https://en.wikipedia.org/wiki/Tape_library

Calculatoare Numerice (2)

- Cursul 2 -

Memoria Cache

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the Day

http://dilbert.com/strips/comic/1999-06-21/

Organizarea memoriei cache

Viteza procesoarelor crește mai rapid decât cea a memoriilor

- Diferența de viteză dintre procesor și memorie crește

Cuprins
Necesitatea unei memorii cache
Cum funcționează un cache?
Cache mapat direct
Cache mapat set-asociativ
Memoria cache și memoria principală
Îmbunătățirea performanței memoriei cache

CPU-Memory Bottleneck

Performanța calculatoarelor este de obicei limitată de lățimea de bandă a memoriei și de latență

- Latență (timpul necesar pentru un singur acces)
  - Memory access time >> Processor cycle time
- Lățime de bandă (numărul de accese pe unitatea de timp)

  Daca un procent m% instrucțiuni dintr-un program accesează memoria:

  -> 100% + m referiri la memorie/instrucțiune

  -> CPI = 1 necesită 100% + m referințe la memorie/ciclu (p.p. o arhitectură RISC)

Diferența dintre procesor și DRAM (latență)

Procesor la 3GHz superscalar accesează în 100ns memoria DRAM sau poate să execute 1,200 instrucțiuni în timpul unui singur acces la memorie!

Dimensiunea fizică afectează latența

- Semnalele parcurg un drum mai lung
- Se distribuie în mai multe locații

Dimensiunile relative ale celulelor de memorie

On-Chip SRAM in logic chip

1 Memory cell in 0.5µm processes

a) Gate Array SRAM
b) Embedded SRAM
c) Standard SRAM (6T cell with local interconnect)
d) ASIC DRAM
e) Standard DRAM (stacked cell)

DRAM on memory chip

[Foss, "Implementing Application-Specific Memory", ISSCC 1996]

Memory	Process	Cell size (μm^2)	Cell efficiency	Bits in $100mm^2 (10^3)$	Gate size (μm^2)	Gate utilization	Gates in $100mm^2 (10^3)$
Gate array SRAM	3-metal ASIC	370	80%	216	185	70%	378
Embedded SRAM	3-metal ASIC	67	70%	1045	185	70%	378
Standard SRAM	2-metal 6T local int.	43	65%	1512	245	40%	163
Embedded ASIC-DRAM	3-metal ASIC	23	60%	2609	185	70%	378
Standard DRAM	2-metal stacked cell	3.2	50%	15625	411	40%	97

Table 1: Memory and logic density for a variety of 0.5µm implementations.

Necesitatea memoriei cache

125 MHz

CPI = 1

500 MHz

CPI ≅ 4

Pipelined

500 MHz

CPI ≅ 1.1

Toate cele 3 implementări presupun timpi de acces de 2-ns pentru date și instrucțiuni; memoriile RAM tipice sunt de 10-50x mai lente

Organizarea ierarhică a memoriilor

holds frequently used data

- *capacitate*: Registre << SRAM << DRAM
- *latență*: Registre << SRAM << DRAM
- *lățime de bandă:* on-chip >> off-chip

Pentru un acces de date:

if data $\in$ fast memory $\Rightarrow$ low latency access *(SRAM)*

if data $\notin$ fast memory $\Rightarrow$ high latency access *(DRAM)*

Managementul ierarhiei de memorii

- Stocare rapidă/mică, de ex. registre
  - Adresa este specificată de obicei în corpul instrucțiunii
  - Implementată direct ca o tabelă de registre
    - *Dar hardware-ul poate să facă operații transparente software-ului, de ex. managementul stivei, redenumirea registrelor etc.*
- Stocare de mari dimensiuni/lentă, de ex. memoria principală
  - Adresa calculată de obicei din valorile stocate în registre
  - Implementată de obicei ca o ierarhie de memorii (cache-mem. princ.) în care hardware-ul decide ce date sunt aduse și ținute în memoria rapidă
    - *Software-ul poate să dea anumite indicii, de ex. nu face prefetch sau nu stoca în cache anumite date*

Analogia dulap de fișiere – sertar de birou

Odată ce paginile cel mai des utilizate sunt în sertarul de birou, nu sunt necesare foarte multe drumuri la dulapul de fișiere.

Foile de pe birou (registre) sau dintr-un sertar (cache) sunt mult mai ușor de accesat decât cele dintr-un dulap de fișiere (memoria principală).

Cache, rata Hit/Miss și timpul efectiv de acces

Memoria cache e transparentă pentru utilizator;
Transferul se face automat

Data is in the cache fraction *h* of the time (say, hit rate of 98%)

Go to main 1 – *h* of the time (say, cache miss rate of 2%)

Un nivel cache cu rata de hit *h*

$$C_{eff} = hC_{fast} + (1-h)(C_{slow} + C_{fast}) = C_{fast} + (1-h)C_{slow}$$

Niveluri multiple de cache

(a) Level 2 between level 1 and main

(b) Level 2 connected to “backside” bus

Memoriile cache funcționează ca un buffer între procesorul ultra-rapid și memoria principală care este mult mai lentă.

Performanțele unui sistem cu cache pe două niveluri

Un sistem cu cache L1 și L2 are un CPI de 1.2 fără cache miss. Pentru fiecare instrucțiune sunt, în medie p=1.1 accese la memorie.

Care este CPI efectiv dacă luăm în calcul și rata cache miss?

Care este rata efectivă de hit și penalizarea pentru miss dacă cache-ul L1 și L2 sunt modelate ca un singur cache?

Level	Local hit rate	Miss penalty
L1	95 %	8 cycles
L2	80 %	60 cycles

Soluție

$C_{eff} = C_{fast} + p(1 - h_1)[C_{medium} + (1 - h_2)C_{slow}]$

Deoarece C_{fast} e inclus în CPI de 1.2, trebuie să calculăm pentru restul

CPI = 1.2 + 1.1(1 – 0.95)[8 + (1 – 0.8)60] = 1.2 + 1.1 × 0.05 × 20 = 2.3

Global: hit rate 99% (95% + 80% of 5%), miss penalty 60 cicli

Parametrii memoriei cache

Cache size (în octeți sau cuvinte). Un cache mai mare poate să țină mai multe date, dar este mai mare și mai lent.

Block or *cache-line size* (unitatea de transfer de date dintre cache și mem. Principală). O linie mai mare pentru cache inseamna mai multe date aduse în cache la fiecare miss.Poate să îmbunătățească rata de hit dar poate să aducă și date mai puțin utile în cache.

Placement policy. Determinarea locului unde o linie cache este plasată. Politicile mai flexibile implică un cost hardware mai mare și pot sau nu să aducă performanțe mărite (datorate complexității mărite a localizării).

Replacement policy. Determinarea blocului din cache ales pentru suprascriere (în care plasăm o linie nouă din cache). Abordări tipice: alegerea unui bloc aleator sau Least Recently-Used (LRU).

Write policy. Determină dacă actualizările în cache sunt transmise imediat memoriei principale (*write-through*) sau blocurile modificate sunt copiate înapoi în memoria principală atunci când trebuie copiate (*write-back* sau *copy-back*).

Graficul referințelor la memorie

Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Șabloane tipice de referință la memorie

Două proprietăți previzibile ale referințelor la memorie

- **Localitate temporală**: Dacă o locație de memorie este accesată, este foarte probabil ca aceeași locație de memorie să fie accesată și în viitorul apropiat.

- **Localitate spațială**: Dacă o locație de memorie este accesată, este foarte probabil ca programul să acceseze și locațiile din imediata vecinătate în viitorul apropiat.

Modele și corelații pentru accesul la memorie

Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Localitatea temporală și spațială

Temporal:
Accesele la aceeași adresă sunt de obicei grupate în timp

Spațial:
La accesul unei locații, tind să fie accesate și adresele din imediata vecinătate

From Peter Denning's *CACM* paper, July 2005 (Vol. 48, No. 7, pp. 19-24)

Exemplu: Pagini memorie accesate la deschiderea și închiderea Firefox

https://cartesianproduct.wordpress.com/2011/11/05/some-thoughts-on-the-linux-page-cache/

Memoriile cache exploatează ambele tipuri de predicții

- Exploatează localitatea temporală prin memorarea conținutului adreselor de memorie accesate recent.
- Exploatează localitatea spațială prin aducerea de blocuri de date din proximitatea locațiilor recent accesate.

Cum funcționează un cache?

Localitate temporală
Localitate spațială

Presupunem că nu există un conflict în maparea adreselor, cache-ul va ține o mică buclă din programul principal, ceea ce va duce la o execuție foarte rapidă.

În interiorul unei memorii cache

Beneficiile caching-ului legate de legea lui Amdahl

Formulați problema cu sertarele din slide-ul anterior folosind legea lui Amdahl. Presupuneți că aveți un hit rate h.

Soluție

Fără sertarul din birou, orice document e accesat în 30s. De exemplu, ca să accesăm 1000 documente durează 30 000s. Sertarul face ca un procent h din cazuri să fie tratate de 6 ori mai repede, timpul de acces rămânând acelasi pentru restul de 1 - h procente. Prin urmare, avem un speedup de $1/(1 - h + h/6) = 6 / (6 - 5h)$. Dacă îmbunătățim timpul de acces la sertar (viteza memoriei cache) putem să creștem speedup-ul, dar, cât timp rata de miss rămâne $1 - h$, nici un speed-up nu poate depăși $1 / (1 - h)$. Pentru un $h = 0.9$, de exemplu, avem un speed-up de 4, iar limita superioară este 10, pentru un timp de acces extrem de scurt.

Notă: Unii oameni ar pune toate dosarele pe birou, în ideea că așa se poate atinge un speed-up mai mare. Nu vă recomand așa ceva!

Compulsory, Capacity, & Conflict Misses

Compulsory misses: Dacă avem o politică de fetching *on-demand,* primul acces la orice resursă va fi întotdeauna un miss. O parte din aceste miss-uri ”obligatorii” pot fi evitate prin prefetching.

Capacity misses: Trebuie să ne debarasăm de o parte din date pentru a face loc altora. Acest lucru duce la miss-uri, datorate capacității limitate a memoriei cache.

Conflict misses: Ocazional, există spațiu ocupat de date care sunt inutile, dar strategia de alocare/mapare ne forțează să invalidăm intrări utile pentru a aduce date noi. Acest lucru poate duce de asemenea la miss-uri.

Dacă avem un cache de capacitate fixă, primele două tipuri de miss sunt mai mult sau mai puțin fixate în jurul unor valori date. Al treilea tip, însă, este influețat de strategia de mapare, care este sub controlul utilizatorilor.

Discutăm în continuare despre două tipuri de mapare: directă și set-asociativă.

http://dilbert.com/strips/comic/1993-12-13/

Algoritmul de funcționare (Read)

Parcurge adresele date de procesor și caută etichetele care corespund. Atunci, ori:

Am găsit eticheta în cache a.k.a. HIT

Întoarce procesorului Copia datelor din cache

Datele nu sunt în cache a.k.a. MISS

Citește blocul de date din mem. princ.

Așteaptă …

Întoarce datele procesorului și actualizează cache-ul

Q: Ce linie din cache înlocuim?

Politica de plasare a liniilor

Numărul blocului

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Memorie

Numărul setului

Cache

Complet Associativ

oriunde

(2-way) Set Associativ

oriunde în setul 0

(12 mod 4)

Mapat direct

numai în blocul 4

(12 mod 8)

blocul 12 poate fi plasat

Cache mapat direct

Memorie cache mapată direct ce conține 32 de cuvinte cu opt linii a câte 4 cuvinte. Fiecare linie are o etichetă asociată și un bit de validitate.

Accesul la o memorie cache mapată direct

Descrieți adresarea unei memorii cache cu adrese pe 32 de biți, cu datele accesibile la nivel de octet. Linia de cache are o lățime W = 16 B. Dimensiunea cache L = 4096 linii (64 KB).

Soluție

Poziția unui octet în linie este codificată pe $\log_2 16 = 4$ b. Adresa de index a unei linii de cache este $\log_2 4096 = 12$ b.

Rămân 32 – 12 – 4 = 16 b pentru etichetă.

Componentele unei adrese de 32 de biți pentru un cache mapat direct cu adresare la nivel de octet.

Comportamentul unui cache mapat direct

Trace pentru adrese:
1, 7, 6, 5, 32, 33, 1, 2, . . .

1: miss, fetch la liniile 3, 2, 1, 0
7: miss, fetch la liniile 7, 6, 5, 4
6: hit
5: hit
32: miss, fetch la 35, 34, 33, 32
(înlocuiește 3, 2, 1, 0)
33: hit
1: miss, fetch la 3, 2, 1, 0
(înlocuiește 35, 34, 33, 32)
2: hit
... ș.a.m.d.

Cache set-asociativ

Cache set-asociativ ce conține 32 de cuvinte de date aranjate în două seturi ce conțin linii de 4 cuvinte.

Accesul la un cache set-asociativ

Descrieți schema de adresare pentru o memorie adresabilă la nivel de octet cu adrese pe 32 de biți. Lățimea liniei de cache 2^W = 16 B.

Mărimea setului 2^S = 2 linii.

Mărimea cache 2^L = 4096 linii (64 KB).

Soluție

Adresa unui octet din linie $\log_2 16 = 4$ b.

Adresa de index într-un set cache $(\log_2 4096/2) = 11$ b.

Rămân 32 – 11 – 4 = 17 b pentru etichetă.

Componentele unui cache set-asociativ cu adresare pe 32 de biți și 2 seturi.

Maparea adreselor

Un cache set-asociativ de 64 KB eight-way (cu opt seturi) este adresabil la nivel de octet și conține linii de 32 de octeți. Adresele de memorie au 32 de biți.

a. Care este dimensiunea etichetelor acestui cache?

b. Ce adrese din memoria principală sunt mapate în setul numărul 5?

Soluție

a. O adresă (32 b) = 5 b byte offset + 9 b set index + 18 b tag

b. Adresele care au set-index (lung de 9 biți) egal cu 5 au o formă generală $2^{14}a + 2^5 \times 5 + b$, de ex. 160-191, 16 554-16 575, . . .

32-bit address	Tag	Set index	Offset
	18 bits	**9 bits**	**5 bits**

Tag width = $32 - 5 - 9 = 18$

Set size = 4×32 B = 128 B
Number of sets = $2^{16}/2^7 = 2^9$

Line width = 32 B = 2^5 B

Cache set-asociativ cu 2 căi

Cache complet asociativ

Politica de înlocuire

Într-un cache asociativ, care bloc dintr-un set trebuie invalidat atunci când setul se umple?

- Aleatoriu
- Least-Recently Used (LRU)
  - Starea pentru cache LRU trebuie actualizată la fiecare acces
  - Implementare funcțională și fezabilă posibiă doar pentru un număr mic de seturi (2-way)
  - pseudo-LRU – arbore binar folosit pentru cache 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - folosit în cache-urile complet-asociative
- Not-Most-Recently Used (NMRU)
  - FIFO, cu excepția blocului/blocurilor cel mai recent folosite

Este un efect de ordin secundar. De ce?

Înlocuirea se petrece NUMAI la un cache miss

Mărimea blocurilor și localitatea spațială

Un bloc este unitatea de transfer dintre cache și memoria principală

4 word block, b=2

Split CPU address

2^b = block size *a.k.a* line size (in bytes)

Un bloc de dimensiuni mari are avantaje distincte d.p.d.v. hardware

- mai puțin overhead pentru etichete
- exploatează capabilitatea memoriei DRAM de a transfera date în rafală
- exploatează transferurile în rafală prin magistralele de date de lățime mare (32-64 biți)

Care sunt dezavantajele creșterii dimensiunii blocurilor?

Mai puține blocuri => mai multe conflicte. Poate să irosească lățime de bandă.

Cache și memoria principală

Cache separat: memorii cache diferite pentru date și instrucțiuni (L1)
Cache unificat: conține instrucțiuni și date (L1, L2, L3)

Arhitectură Harvard: memorii de date și instrucțiuni separate
Arhitectură von Neumann: o singură memorie pentru date și instrucțiuni

Probleme la scriere:

Write-through încetinește memoria cache pentru a permite memoriei principale să scrie datele

Write-back sau copy-back este mai puțin problematic, dar tot dăunează performanțelor din cauza acceselor multiple la memorie.
Soluție: Dotează memoria cache cu buffering la scriere a.î. nu trebuie să aștepte memoria principală.

Transferuri de date rapide între cache și memoria principală

Un chip DRAM de 256 Mb este organizat ca o memorie 32M × 8: patru astfel de chipuri constituie o memorie de 128MB.

Îmbunătățirea performanțelor memoriei cache

Pentru o dimensiune cache dată, există următoarele probleme de design:
Lățimea liniei (2^W). O valoare prea mică a *W* cauzează mai multe accese la memoria principală; o valoare prea mare crește penalizarea pentru un miss și poate să încarce memoria chache cu date de utilitate scăzută, care pot fi înlocuite înainte de a fi utilizate.

Mărimea setului sau asociativitatea (2^S). Mapare directă ($S = 0$) este simplă și rapidă; o mai mare asociativitate duce la o complexitate mărită și la timpi de acces mai mari dar tinde să reducă miss-urile conflictuale.

Line replacement policy. De obicei este algoritmul LRU (least recently used); nu este o problemă pentru cache-ul mapat direct. Funcționează bine și un algoritm de selecție aleatoare a liniilor pe care să le invalidăm.

Write policy. Memoriile cache moderne sunt foarte rapide, a.î. *Write-through* nu este aproape niciodată o politică bună. De obicei alegem *write-back* sau *copy-back*, folosind buffering la scriere pentru a minimiza impactul adus de latența memoriei principale.

Efectele asociativității asupra performanței

Îmbunătățirea performanțelor memoriei cache în funcție de asociativitate.

http://dilbert.com/strips/comic/2008-02-22/

Embedded Systems Laboratory

Catedra de Calculatoare

Facultatea de Automatica si Calculatoare

Calculatoare Numerice (2)

- Cursul 3 -

Memoria Cache (2)

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the Day

http://dilbert.com/strips/comic/1995-11-17/

Din episodul anterior

- RAM Dinamic (DRAM) este principalul tip de memorie principală folosită în ziua de azi
  - Stochează valorile binare ca tensiuni pe condensatoare mici, care au nevoie de refresh (de aici partea de dinamic)
  - Acces lent, în mai mulți pași: precharge, read row, read column
- RAM Static (SRAM) este mai rapid dar mai scump
  - Folosit pentru a construi memoria cache
- Cache-ul conține un set de valori în memoria rapidă (SRAM) aproape de procesor
  - Trebuie să avem un algoritm de căutare a valorilor din cache și o politică de înlocuire pentru a face loc pentru locațiile nou accesate
- Memoriile cache exploatează două forme de predictibilitate în adresarea memoriei
  - Localitate temporală – aceeași locație accesată de mai multe ori
  - Localitate spațială – mai multe accese la locațiile învecinate

Dimensiunea liniilor și localitatea spațială

O linie este unitatea de transfer dintre cache și memoria principală

2^b = line size *a.k.a* line size (in bytes)

O linie de dimensiuni mari are avantaje din punct de vedere hardware

- overhead mai mic la tag-uri
- exploatează tranferurile în rafală din DRAM
- exploatează transferurile în rafală pe magistralele de date

Care sunt dezavantajele unei dimensiuni mari pentru liniile de cache?

Mai puține linii => mai multe conflicte. Poate să irosească lățime de bandă

Cache mapat direct

2-Way Set-Associative Cache

Fully Associative Cache

Politica de înlocuire

Într-un cache asociativ, care linie dintr-un set ar trebui invalidată atunci când setul se umple?

- Random
- Least-Recently Used (LRU)
  - Starea unui cache LRU trebuie actualizată la fiecare acces
  - Implementare fezabilă doar pentru seturi mici (2-way)
  - Arbore binar Pseudo-LRU folosit pentru 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - Folosit în cache-urile complet-asociative
- Not-Most-Recently Used (NMRU)
  - FIFO, cu excepția liniei utilizate cel mai recent

E un efect de ordin secundar. De ce?

O linie este înlocuită doar la un cache miss

Interacțiunea Cache-CPU
(5-stage pipeline)

Îmbunătățirea performanțelor memoriei cache

Average memory access time (AMAT) =

Hit time + Miss rate x Miss penalty

Pentru a îmbunătăți performanța:

- reducerea hit time
- reducerea miss rate
- reducerea miss penalty

Care este cel mai bun design cache pentru o b.a. Cu 5 etape?

Cel mai mare cache care nu mărește hit-time mai mult de un ciclu (approx 8-32KB în tehnologia modernă)

[probleme de design mai mari atunci când avem de-a face cu b.a. mai mari sau procesoare superscalare]

Cauzele pentru cache miss: The 3 C's

Compulsory: prima referință la o linie (a.k.a. cold start misses)

- *Miss-uri care se petrec și pentru un cache de dimensiuni infinite*

Capacity: cache prea mic pentru a ține toate datele necesare unui program

- *Miss-uri care se petrec și dacă avem o politică perfectă de înlocuire a liniilor din cache*

Conflict: miss-uri care se petrec datorită coliziunilor datorate strategiei de amplasare a liniilor

- *Miss-uri care nu ar apare dacă am avea o asociativitate completă*

Efectele parametrilor cache asupra performanței

- Cache de dimensiuni mari
  + Reduce miss-urile de capacitate și conflict
  - Crește hit time

- Asociativitate mărită
  + Reduce conflict misses
  - Poate să marească hit time

- Linii de dimensiuni mai mari
  + Reduce miss-urile obligatorii și de capacitate (la reload)
  - Crește miss-urile de conflict și penalizările pentru un miss

Exemplul 1 performanța cache

- Un program are 2000 de operații load și store
- 1250 de date sunt aduse de aceste operații în cache
- Restul sunt luate din alte locații de memorie din afara cache

- **Care este rata de hit și miss pentru cache?**

Hit Rate = 1250/2000 **= 0.625**

Miss Rate = 750/2000 = **0.375** = 1 – Hit Rate

Exemplul 2 performanța cache

Presupunem că procesorul are două niveluri ierarhice de memorie: cache și mem. principală

- t_{cache} = 1 ciclu, t_{MM} = 100 cicli

Care este valoarea AMAT pentru exemplul 1?

AMAT = *t*cache + *MR*cache(*tMM*)
= [1 + 0.375(100)] cicli
= **38.5 cicli**

Cache mapat direct

Cache mapat direct – performanță

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = ?

Cache mapat direct – performanță

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 3/15

= 20%

Temporal Locality

Compulsory Misses

Cache mapat direct – conflict miss

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = ?

Cache mapat direct – conflict miss

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 10/10

= 100%

Conflict Misses

N-Way Set Associative Cache

N-Way Set Associative Cache

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = ?

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
0			0			Set 1
0			0			Set 0

N-Way Set Associative Cache

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 2/10
= 20%

Associativity reduces conflict misses

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

Tendințe miss-rate pentru N-way cache

- Un cache mare reduce rata de miss de capacitate
- Asociativitatea mai mare reduce rata de miss de conflict

Din Patterson & Hennessy, *Computer Architecture: A Quantitative Approach*, 2011

Tendințe miss-rate pentru N-way cache

- Blocurile mari reduc rata compulsory miss
- Dar, măresc rata conflict miss

Algoritmi pentru write-back

- Cache hit:
  - ***write through***: orice modificare în cache este actualizată automat și în RAM
    - De obicei mai mult trafic, dar duce la o implementare mai simplă a b.a. și a memoriei cache
  - ***write back***: scrie numai în cache, memoria este actualizată doar când linia este invalidată
    - Fiecare linie are un flag de ”dirty” care marchează liniile ce trebuiesc actualizate în RAM – micșorează traficul
    - Trebuie să permită 0, 1 sau 2 accese la memorie pentru fiecare load/store
- Cache miss:
  - ***no write allocate***: scrie doar înapoi în RAM
  - ***write allocate*** (aka fetch on write): face fetch în cache
- Combinații întâlnite:
  - write through, fără write allocate
  - write back împreună cu write allocate

Reducerea timpului la Write Hit

Problemă: Scrierile durează doi cicli, unul pentru verificarea tag-urilor și altul pentru scrierea datelor la un hit.

Soluții:

- Proiectăm un circuit RAM care să permită o citire și o scriere într-un singur ciclu de ceas, reface vechea valoare dacă avem un tag miss
- Fully-associative (CAM Tag) caches: Linia este activată doar dacă avem un hit
- Pipelined writes: Ține datele ce trebuiesc scrise într-un singur buffer, în afara memoriei cache. Scrie datele în timpul ciclului de tag check

Write Buffering pentru reducerea penalizărilor la Read

Evicted dirty lines for write back cache
OR
All writes in write through cache

Procesorul nu este blocat la o scriere și un read miss poate să acceseze mai repede decât un write memoria RAM

Problema: Buffer-ul pentru write poate să conțină o valoare actualizată a locației cerute de un read miss

Soluție simplă: la un read miss, așteaptă să se golească write buffer.

Soluție mai rapidă: Verifică adresele din write buffer și compară-le cu cele ale liniilor pentru care avem read miss. Dacă nu coincid, lasă read miss să acceseze memoria RAM; dacă nu, întoarce valoarea conținută în write buffer.

Reducerea întârzierilor de parcurgere prin indexarea pe sub-blocuri

- **Problemă**: Etichetele pentru o linie sunt prea mari -> un overhead prea mare
  - Soluție simplă: Linii mai mari; va duce automat la creșterea penalizărilor ce apar la un miss.
- **Soluție**: Sub-block placement (aka sector cache)
  - Se adaugă un bit de validitate pentru unități mai mici de o linie, denumite sub-blocuri
  - La miss – citește doar un sub-bloc
  - *Dacă avem tag match, este cuvântul respectiv în cache?*

100	***1***		***1***		***1***		***1***	
300	***1***		***1***		***0***		***0***	
204	***0***		***1***		***0***		***1***	

http://dilbert.com/strips/comic/2005-04-29/

Memoria Cache pe mai multe niveluri

Problemă: O memorie nu poate să fie și mare și rapidă

Soluție: Creștem dimensiunea cache cu fiecare nivel

Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

Misses per instruction = misses in cache / number of instructions

Prezența unui L2 cache influențează design-ul L1 cache

- Folosim un cache L1 dacă avem și un L2
  - Dimensiunea lui L1 va fi probabil mai mică -> va avea un miss rate mai mare, dar reducem drastic timpul de hit
  - Existența unui cache L2 reduce penalizarea la miss pentru L1
  - Reduce consumul mediu de energie pe acces
- Putem folosi L1 write-through (simplu de implementat) cu L2 on-chip
  - Cache L2 write-back L2 absoarbe traficul pentru write, nu accesează resurse din afara chip-ului
  - Simplifică problemele de coerență
  - Simplifică procesul de error recovery pentru L1 (poate să folosească doar biți de paritate și să reîncarce din L2 atunci când este detectată o eroare de paritate la citirea din L1)

Politica de incluziune

- Inclusive multilevel cache:
  - Cache-ul interior conține liniile care sunt prezente și în cache-ul exterior
  - Menținerea coerenței se face doar prin accese la cache-ul exterior (snooping)
- *Exclusive* multilevel caches:
  - Cache-ul interior conține liniile care nu sunt prezente în cel exterior
  - Interschimbăm liniile din cache interior-exterior la un miss
  - Folosit la AMD Athlon cu 64KB primary și 256KB secondary cache

Care variantă este mai bună?

Intel Pentium III die

Itanium-2 On-Chip Caches
(Intel/HP, 2002)

Level 1: 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

Level 2: 256KB, 4-way s.a, 128B line, quad-port (4 load or 4 store), five cycle latency

Level 3: 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

Power 7 On-Chip Caches [IBM 2009]

32KB L1 I$/core

32KB L1 D$/core

3-cycle latency

256KB Unified L2$/core

8-cycle latency

32MB Unified Shared L3$

Embedded DRAM (eDRAM)

25-cycle latency to local slice

Core i7-6950X [Intel 2016]

Apple A14 [2020]

z196 Mainframe Caches [IBM 2010]

- 96 cores (4 cores/chip, 24 chips/system)
  - Out-of-order, 3-way superscalar @ 5.2GHz
- L1: 64KB I-$/core + 128KB D-$/core
- L2: 1.5MB private/core (144MB total)
- L3: 24MB shared/chip (eDRAM) (576MB total)
- L4: 768MB shared/system (eDRAM)

Prefetching

- Speculează care vor fi următoarele accese la instrucțiuni și date și încarcă acele înregistrări în cache(uri)
  - Accesele la instrucțiuni sunt mai ușor de prezis decât accesele la date
- Variante de prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- Care tipuri de miss-uri sunt afectate de prefetching?

Probleme generate de prefetching

- Utilitate – ar trebui să mărească rata de hit
- Punctualitate – nu trebuie să ajungă prea târziu (dar nici prea devreme)
- Poluarea cache-ului și a lățimii de bandă

Hardware Instruction Prefetching

Prefetch al instrucțiunilor la Alpha AXP 21064

- Fetch la două linii pentru un miss; linia cerută (i) și linia imediat următoare (i+1)
- Linia cerută plasată în cache, următoarea linie într-un buffer special - instruction stream buffer
- Dacă miss în cache dar hit în stream buffer, mută linia din stream buffer în cache și prefetch la următoarea linie (i+2)

Hardware Data Prefetching

- Prefetch-on-miss:
  - Prefetch la b + 1 dacă am miss b
- One-Block Lookahead (OBL)
  - Initializează prefetch pentru blocul b + 1 când accesăm blocul b
  - De ce e diferit față de dublarea dimensiunii blocurilor?
  - Poate fi extins la N-block lookahead
- Strided prefetch
  - Dacă există o secvență de accese de tipul b, b+N, b+2N, atunci prefetch la linia b+3N etc.

- Exemplu: IBM Power 5 [2003] implementează opt căi independente de strided prefetch per procesor și face prefetch cu 12 linii înainte pentru un acces dat

Software Prefetching

```
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
 }
```

Software Prefetching Issues

- Cea mai mare problemă este timing-ul, nu predictibilitatea
  - Dacă faci prefetch foarte aproape de momentul în care ai nevoie de date, s-ar putea să fie prea târziu
  - Prefetch prea devreme – poluezi cache-ul
  - Estimăm cât de mult timp e nevoie pentru a aduce datele în L1 pentru a face un prefetch exact
  - *De ce e greu de făcut asta?*

```
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
  }
```

Trebuie să luăm în considerare costul instrucțiunilor de prefetch

Optimizări de compilator

- Restructurarea codului afectează secvența de acces la date
  - Grupăm accesele de date împreună pentru a îmbunătăți localitatea spațială
  - Re-aranjăm accesele la date pentru a îmbunătăți localitatea temporală
- Prevenim intrările nedorite de date în cache
  - Folositor pentru variabilele care vor fi accesate o singură dată înainte de a fi înlocuite
  - Necesită un mecanism prin care software-ul să spună hardware-ului să nu facă data caching (“no-allocate” instruction hints sau page table bits)
- Invalidează datele care nu vor fi folosite niciodată
  - Un stream de date exploatează localitatea spațială, dar nu și cea temporală
  - Înlocuiește în locațiile ”moarte” din cache

Loop Interchange

```
for(j=0; j < N; j++) {
    for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

```
for(i=0; i < M; i++) {
    for(j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

Ce fel de localitate exploatează schimbarea de sus?

Loop Fusion

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

for(i=0; i < N; i++)
      d[i] = a[i] * c[i];
```

```
for(i=0; i < N; i++)
{
      a[i] = b[i] * c[i];
      d[i] = a[i] * c[i];
}
```

Dar schimbarea de sus?

Matrix Multiply, Naïve Code

```
for(i=0; i < N; i++)
     for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = r;
     }
```

Not touched *Old access* *New access*

Matrix Multiply with Cache Tiling

```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
```

Ce tip de localitate este îmbunătățită?

Exemplul 1

Program A

```
struct DATA
{
    int a;
    int b;
    int c;
    int d;
};
DATA * pMyData;

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

Program B

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

DATA * pMyData;

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

~2X mai rapid!

Exemplul 2

Program C

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

DATA * pMyData;

for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

Program D

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

DATA * pMyData;

for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

60% mai rapid!

Exemplul 3

Program E	Program F

`char * p;`
 	
`p = new char[SIZE];`
 	
`for (long x=0; x<sRowSize; x++)`
`for (long y=0; y<nbRows; y++)`
`{`
`  p[x+y*sRowSize]++;`
`}`
	`char * p;`
	
	`p = new char[SIZE];`
	
	`for (long y=0; y<nbRows; y++)`
	`for (long x=0; x<sRowSize; x++)`
	`{`
	`  p[x+y*sRowSize]++;`
	`}`

de 4x mai rapid!

Exemplul 3

Program E	Program F
	x mai oid!

```
char * p;

p = new char[S

for (long x=0;
for (long y=0;
{
    p[x+y*sRow
}
```

```
+)
```

http://dilbert.com/strips/comic/2010-02-22/

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

-Cursul 4 –

Memoria virtuală

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the Day

http://dilbert.com/strips/comic/2008-02-12/

Din episodul anterior

- 3 C's of cache misses
  - Compulsory, Capacity, Conflict
- Write policies
  - Write back, write-through, write-allocate, no write allocate
- Ierarhiile cache multi-nivel reduc penalizările la miss
  - 3 niveluri în sistemele moderne (unele au chiar 4!)
  - Prezența L2 modifică organizarea L1
- Prefetching: aduce date în cache înainte de o cerere a CPU-ului
  - Prefetching poate să irosească lățime de bandă și să cauzeze poluarea cache-ului
  - Software vs hardware prefetching
- Optimizări software pentru cache
  - Loop interchange, loop fusion, cache tiling

Mașina de calcul simplă

- Într-o mașină de calcul foarte simplă, orice adresă este o adresă fizică

Adrese absolute

EDSAC, anii 50'

- Nu aveam concurență, un singur fir de execuție cu acces nerestricționat la întreaga mașină (RAM + I/O devices)
- Adresele dintr-un program depindeau de locația de memorie unde programul era încărcat
- *Dar* era mult mai convenabil pentru programatori să scrie subrutine independente de locație

Cum poate fi obținută independența de locații?

Linker și/sau loader care modifică adresele subrutinelor atunci când creează imaginea în memorie a programului respectiv

Dynamic Address Translation

- Motivație
  - La primele mașini de calcul I/O era lent și fiecare transfer pe I/O implica și procesorul (programmed I/O)
  - Productivitate mărită dacă CPU și I/O pentru 2 sau mai multe programe erau suprapuse. Cum?

  -> multiprogramare cu DMA , I/O devices, întreruperi
- Programe independente de locație
  - Ușurință la programare și la manangementul memoriei

    -> avem nevoie de un registru de ***bază***
- Protecție
  - Programele independente nu ar trebui să se afecteze reciproc

    -> avem nevoie de un registru ***limită***
- Programe multiple care rulează pe aceeași mașină necesită software supervizor pentru a mijloci schimbarea de context dintre programele respective

Translatarea adresei de bază și a limitei

Registrele pentru bază și limită sunt vizibile/accesibile doar atunci când procesorul funcționează în modul *supervizor*

Zone separate pentru program și date

Care este avantajul acestei delimitări?

Fragmentarea memoriei

Pe măsură ce utilizatorii ”vin și pleacă”, memoria este ”fragmentată”.
Prin urmare, la un moment dat programele trebuie să fie mutate pentru a compacta memoria.

http://dilbert.com/strips/comic/2008-02-13/

Writer

```c
#include <stdio.h>

int
main (void)
{

  int *address = (int *) 0xCAFEBABE;

  printf ("Memory address is: 0x%x\n", address);

  *address = 0xDEADBEEF;

  return 0;
}
```

Reader

```c
#include <stdio.h>

int
main (void)
{

  int *address = (int *) 0xCAFEBABE;

  printf ("Memory address is: 0x%x\n", address);

  printf ("Content of that address is: 0x%x\n", *address);

  return 0;
}
```

Memoria Virtuală

- Creează iluzia unei memorii mai mari
- Memoria principală (DRAM) funcționează drept cache pentru hard-disk

Ierarhia memoriilor

Technologie	Preț / GB	Timp acces (ns)	Lățime de bandă (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

- **Memorie fizică:** DRAM (Memoria principală)
- **Memoria virtuală:** Mapată peste RAM și hard drive
  - Lentă, mare, ieftină

Hard Disk

Durează milisecunde pentru a căuta locația corectă de pe disc (*seek time*).

Necesitatea unei memorii virtuale

Segmente dintr-un program în memoria principală și pe disc

Ierarhia memoriilor: tabloul complet

Fluxul de date în ierarhia de memorii

Memoria virtuală

- **Adrese virtuale**

– Programele folosesc adresele virtuale
– Întregul spațiu de adrese virtuale stocat pe hard disk
– Subset al datelor din DRAM
– CPU translatează adresele virtuale în ***adrese fizice*** (adrese DRAM)
– Datele care nu sunt în DRAM sunt luate de pe hard-drive

- **Protecția memoriei**

– Fiecare program are propria mapare de la adrese virtuale la adrese fizice
– Două programe pot folosi aceleași adrese virtuale pentru a accesa date complet diferite
– Programele nu trebuie să fie ”conștiente” de faptul că există alte programe care rulează concurent pe aceeași mașină
– Un program (sau virus) nu poate corupe memoria folosită de alt program

http://dilbert.com/strips/comic/2008-02-14/

Analogia cache/memorie virtuală

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

Memoria fizică joacă rolul de cache pentru memoria virtuală

Definiții

- **Dimensiunea paginii:** cantitatea de memorie transferată de la hard-disk către DRAM într-o singură tranzacție
- **Translatarea adreselor:** determinarea adresei fizice din cea virtuală
- **Tabela de pagini:** lookup table folosit pentru translatarea adreselor virtuale în adrese fizice

Adrese fizice și virtuale

© 2007 Elsevier, Inc. All rights reserved

Majoritatea acceselor fac hit în memoria fizică

Dar, programele au capacitatea memoriei virtuale

Sisteme cu memorie paginată

- Adresa generată de procesor poate fi împărțită în:

Page Number	Offset

- Tabela de pagini conține adresa fizică pentru începutul fiecărei pagini

Tabela de pagini permite stocarea paginilor unui program în zone de memorie care nu sunt contigue.

Spațiu de adrese privat pentru fiecare utilizator

- Fiecare utilizator are o tabelă de pagini
- Tabela conține câte o intrare pentru fiecare pagină a utilizatorului

Unde ar trebui să rezide tabelele de pagini?

- Spațiul necesar pentru tabele de pagini (PT) este proporțional cu spațiul de adresă, numărul de utilizatori, etc.

  $\Rightarrow$ *Prea mare pentru a fi ținut în registrele generale*

- Idee: Ține PT în memoria principală
  - Are nevoie de o referință la memorie pentru a citi adresa de bază și alta pentru a accesa un cuvânt de date (de fiecare dată)

    $\Rightarrow$ *dublează numărul de accese la memorie!*

Tabelele de pagini în memoria principală

O problemă la începutul anilor șaizeci

- Existau deja multe aplicații pentru care toate datele nu puteau să intre în memoria principală (de ex. Programe de contabilitate cu mii de înregistrări)
  - *Sistemul cu memorie paginată reducea fragmentarea dar tot necesita ca întreg programul să rezide în memoria principală*

Manual Overlays

- Presupunem că o instrucțiune poate accesa orice adresă din memoria principală

- *Metoda 1:* programatorul ține evidența tuturor adreselor din memoria princ. Și inițiază un transfer I/O doar când este necesar
  - *Dificil, predispus la erori!*
- *Metoda 2:* inițializare automată a unui tranfer I/O prin translatarea software a adreselor
  - *Brooker's interpretive coding, 1960*
  - *Ineficient!*

Ferranti Mercury
1956

Nu e doar o tehnică de "magie neagră" antică, este folosită și la procesoarele IBM Cell din Playstation 3, de exemplu.

Paginare la cerere pentru Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

Tom Kilburn

Memoria primară servește drept "cache" Pentru memoria secundară

Utilizatorul "vede" 32 x 6 x 512 cuvinte pentru stocare

Algoritmul de paginare pentru Atlas

La un page fault:

- Se inițializează transferul unei pagini libere noi
- Este actualizat Page Address Register (PAR)
- Dacă nu a mai rămas nici o pagină liberă, selectează o pagină pentru a fi înlocuită (bazat pe utilizare)
- Pagina înlocuită este scrisă în memorie
  - Pentru a minimiza latența de acces la memorie, se va selecta prima pagină goală întâlnită în memorie
- Tabela de pagini este actualizată pentru a indica la noua locație a paginii din memorie

Tabelă liniară de pagini

- Page Table Entry (PTE) conține:
  - Un bit care indică dacă pagina există
  - PPN (physical page number) pentru o pagină rezidentă în memorie
  - DPN (disk page number) pentru o pagină de pe disc
  - Biți de status pentru protecție și utilizare
- SO setează Page Table Base Register de fiecare dată când procesul activ pentru utilizator se schimbă

Dimensiunea unei tabele de pagini liniare

- Cu adrese pe 32 de biți, pagini de 4KB & PTE de 4 octeți:
  - 220 PTE, 4 MB tabelă pagini per utilizator
  - 4GB de swap necesari pentru a face back-up la tot spațiul de adrese virtuale
- Pagini mai mari?
  - Fragmentare internă (Nu e folosită toată memoria dintr-o pagină)
  - Penalizare pentru dimensiune (trebuie mai mult timp pentru a citi de pe disc)
- Cum scalează asta la un spațiu de adrese pe 64 de biți???
  - Chiar și o pagină de 1MB ar avea nevoie de 244 PTE-uri de 8 biți (=35TB!)

Care este ideea salvatoare?

Tabele de pagini ierarhice

Tabele de pagini pe două niveluri în memoria fizică

http://dilbert.com/strips/comic/2008-02-15/

Translatarea adreselor în memoria virtuală

Parametrii translatării virtual-fizic

Translatarea adreselor

© 2007 Elsevier, Inc. All rights reserved

Tabele de pagini si translatarea adreselor

Rolul tabelei de pagini în procesul translației adreselor virtuale în adrese fizice

Exemplu memorie virtuală

- **Sistem:**
  - Dimensiunea mem. virtuale: 2 GB = 2^{31} octeți
  - Dimensiunea memoriei fizice: 128 MB = 2^{27} octeți
  - Dimensiunea paginii: 4 KB = 2^{12} octeți

Exemplu memorie virtuală

- **Sistem:**
  - Memoria virtuală: 2 GB = 2^{31} octeți
  - Memoria fizică: 128 MB = 2^{27} octeți
  - Dimensiune pagină: 4 KB = 2^{12} octeți
- **Organizare:**
  - Adrese virtuale: **31** biți
  - Adrese fizice: **27** biți
  - Offset pagină: **12** biți

  - # pagini virtuale = $2^{31}/2^{12}$ = $\mathbf{2^{19}}$ (VPN = 19 biți)
  - # pagini fizice = $2^{27}/2^{12}$ = $\mathbf{2^{15}}$ (PPN = 15 biți)

Exemplu memorie virtuală

- **19-biți – numere de pagină virtuală**
- **15-biți – numere de pagină fizică**

© 2007 Elsevier, Inc. All rights reserved

Exemplu memorie virtuală

Care e adresa fizică a adresei virtuale **0x247C?**

© 2007 Elsevier, Inc. All rights reserved

Exemplu memorie virtuală

Care e adresa fizică a adresei virtuale **0x247C?**

–VPN = **0x2**

–VPN 0x2 se mapează la PPN **0x7FFF**

–12-bit page offset: **0x47C**

–Adresa fizică = **0x7FFF47C**

© 2007 Elsevier, Inc. All rights reserved

Cum se face translatarea?

- **Tabela de pagini**
  - Intrare pentru fiecare pagină virtuală
  - Câmpuri pentru fiecare intrare:
    - **Bit validitate:** 1 dacă pagina este în memoria fizică
    - **Numărul paginii fizice:** unde este localizată pagina

Exemplu paginare

VPN este indexul din tabela de pagini

Exemplul 1

Care este adresa fizică corespunzătoare adresei virtuale **0x5F20**?

V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Page Table

Exemplul 1

Care este adresa fizică corespunzătoare adresei virtuale **0x5F20**?

–VPN = **5**

–Intrarea 5 în page table VPN 5 => pagina fizică **1**

–Adresa fizică: **0x1F20**

Exemplul 2

Care este adresa fizică a adresei virtuale **0x73E0**?

Exemplul 2

Care este adresa fizică a adresei virtuale **0x73E0**?

–VPN = **7**

–Intrarea 7 este invalidă

–Pagina virtuală trebuie adusă în memoria fizică de pe disc

Translatarea și protecția adreselor

- Fiecare instrucțiune și acces la date are nevoie de translatarea adreselor și verificări la protecție

O mașină virtuală bună trebuie să fie rapidă (~ un ciclu) și eficientă la ocuparea spațiului

Protecția memoriei principale

- Procese multiple (programe) rulează concomitent pe o mașină de calcul
- Fiecare proces are tabela de pagini proprie
- Fiecare proces poate folosi întregul spațiu de adrese virtuale
- Un proces poate accesa doar paginile fizice mapate în propria tabelă de pagini

Protecția și partajarea memoriei virtuale

Memoria virtuală facilitează partajarea datelor și protecția memoriei

Penalizarea de latență pentru memoria virtuală

Probleme cu tabela de pagini

- **Tabela de pagini este prea mare**
  - de obicei este localizată în memoria fizică
- Load/store necesită 2 accese la memoria principală:
  - Unul pentru translație (page table read)
  - Unul pentru acces date (după translație)
- Taie performanța memoriei la jumătate
  - ***Nu și dacă facem lucrurile deștept…***

Translation Lookaside Buffer (TLB)

Translatarea adreselor este foarte costisitoare!

La o tabelă cu două niveluri, fiecare referință se traduce prin mai multe referințe la memorie

Soluție: *Translatarea TLB în cache*

TLB hit ⇒ *Single-Cycle Translation*

TLB miss ⇒ *Page-Table Walk to refill*

Translation Lookaside Buffer (TLB)

- Este un cache mic care conține cele mai recente translații
- Reduce numărul de accese la memorie pentru *majoritatea* operațiilor load/store (de la 2 la 1)

TLB

- Accese la tabela de pagini: localitate temporală mărită
  - Pagină de dimensiuni mari, deci load/store consecutive probabil vor accesa aceeași pagină
  - TLB
  - Mic: accesat în < 1 ciclu
  - De obicei 16 - 512 intrări
  - Complet asociativ
  - > 99 % hit rate
  - Reduce nr. accese la memorie de la 2 la 1 pentru operații load/store

Design TLB

- De obicei 32-128 intrări, de obicei complet asociativ
  - Fiecare intrare descrie o pagină mare, deci mai puțină localitate spațială între diferitele pagini ➔ mai probabil ca două intrări să fie în conflict
  - Câteodată TLB mai mare (256-512 intrări) care sunt 4-8 way set-asociative
  - Sistemele mai mari au TLB multi-nivel (L1 și L2)
- Random or FIFO replacement policy
- TLB Reach: dimensiunea celui mai mare spațiu virtual de adresă care poate fi mapat simultan de TLB

Exemplu: 64 intrări TLB, pagini de 4KB, o pagină per intrare

TLB Reach = *64 intrări * 4 KB = 256 KB (dacă e contiguu)*?

Translation Lookaside Buffer

Program page in virtual memory

```
     lw     $t0,0($s1)
     addi   $t1,$zero,0
L:   add    $t1,$t1,1
     beq    $t1,$s2,D
     add    $t2,$t1,$t1
     add    $t2,$t2,$t2
     add    $t2,$t2,$s1
     lw     $t3,0($t2)
     slt    $t4,$t0,$t3
     beq    $t4,$zero,L
     addi   $t0,$t3,0
     j      L
D:   ...
```

Toate instrucțiunile de pe această pagină au aceeași adresă virtuală de pagină – sunt supuse aceleiași translații

Translatare virtual-fizic a adreselor folosind un TLB și cum este folosită adresa fizică obținută pentru a accesa memoria cache

Tratarea unui TLB Miss

- Software (MIPS, Alpha)
  - Un TLB miss produce o excepție și sistemul de operare reîncarcă TLB. Trebuie să existe un mod de adresare privilegiat ”netranslatat” pentru refacerea TLB.
- Hardware (SPARC v8, x86, PowerPC, RISC-V)
  - Avem o unitate memory management unit (MMU) care reface TLB.
  - Dacă o pagină lipsă (date sau PT) este găsită în timpul reîncărcării TLB, MMU renunță și generează o excepție de Page Fault pentru instrucțiunea originală.

Exemplu TLB cu două intrări

Translatarea adreselor via TLB

Un algoritm de translatare convertește o adresă virtuală de 32 de biți într-o adresă fizică de 32 de biți. Memoria este adresabilă la nivel de octet și are pagini de 4KB. Este folosit un TLB cu 16 intrări mapat direct. Care sunt componentele adreselor virtuale și fizice și lățimea diferitelor câmpuri ale TLB

Soluție

TLB word width =
16-bit tag +
20-bit phys page # +
1 valid bit +
Other flags
≥ 37 bits

http://dilbert.com/strips/comic/1990-11-08/

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

– Cursul 5 –

Memoria virtuală

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the Day

http://xkcd.com/378/

Memoria Virtuală

- Programele fac referințe la adrese din memoria virtuală
  - `movl (%ecx),%eax`
  - Organizată conceptual ca un vector foarte mare de octeți
  - Fiecare octet are propria adresă
- De fapt, implementat ca o ierarhie de diferite tipuri de memorii
- Sistemul furnizează spații de adresă pentru fiecare proces
- Alocare: La compilare și run-time
  - Unde trebuie stocate diferitele părți ale programului
  - Toată alocare se face într-un singur spațiu virtual de adresă
- *Dar de ce avem memorie virtuală?*
- *De ce nu avem direct memorie fizică?*

Problema 1:Unde încape totul?

Memorie fizică:
Câțiva GigaBytes

Adrese pe 64 de biți:
16 ExaByte
(1EB = 1.000.000.000GB)

Și sunt multe procese...

Problema 2: Memory Management

Memoria fizică (RAM)

Process 1
Process 2
Process 3
…
Process n

X

stack
heap
`.text`
`.data`
…

Problema 3: Protecție

Problema 4: Partajarea memoriei

Soluție: Mapare

- Fiecare proces primește un spațiu privat de memorie
- Rezolvă toate problemele anterioare

Spații de adresă

- Spațiu liniar de adresă: Set ordonat și contiguu de adrese întregi ne-negative:

{0, 1, 2, 3 … }

- Spațiu virtual de adresă: Set de $N = 2^n$ adrese virtuale

{0, 1, 2, 3, …, N-1}

- Spațiu fizic de adresă: Set de $M = 2^m$ adrese fizice

{0, 1, 2, 3, …, M-1}

- Distincție clară între date (bytes) și atributele acestora (adrese)
- Fiecare obiect poate avea acum adrese multiple
- Fiecare octet din memoria principală:
  o adresă fizică, una (sau mai multe) adrese virtuale

Sistem ce folosește adresarea fizică

- [Încă] folosite în sisteme ”simple” cum sunt microcontrollerele (din mașini, lifturi, cuptoare cu microunde, telefoane mobile, rame foto digitale...)

Sistem cu adresare virtuală

- Folosite în toate calculatoarele moderne (desktop, laptop, server)
- Una din marile ”invenții” ale computer science
- *MMU verifică cache-ul*

De ce avem Virtual Memory (VM)?

- Utilizare eficientă a memoriei principale(RAM)
  - Folosește RAM-ul ca un cache pentru părți dintr-un spațiu virtual de adrese
    - Unele părți care nu sunt în cache sunt stocate pe disc
    - Restul de părți nealocate nu sunt stocate nicăieri
  - Ține în memorie doar zonele active ale spațiului virtual de adresă
    - Transferă datele înainte și înapoi după necesități
- Simplifică managementul memoriei pentru programatori
  - Fiecare proces primește același spațiu privat și liniar de adrese
- Izolează spațiile de adrese
  - Un proces nu poate modifica zona de memorie a altui proces
    - Pentru că operează în spații de memorie diferite
  - Utilizatorii nu pot accesa informații privilegiate
    - Spații diferite de adresă au diferite permisiuni

VM ca utilitar pentru caching

- Memorie virtuală: vector de $N = 2^n$ octeți
  - gândiți-vă că vectorul (partea alocată din el) este stocat pe disc
- Memorie fizică principală (DRAM) = cache pentru memoria virtuală alocată
- Blocurile sunt denumite pagini; dimensiune = 2^p

Ierarhia de memorii: Core 2 Duo

Organizarea cache DRAM

- Organizarea DRAM ca un cache este provocată de penalizările enorme pentru miss
  - DRAM este de aprox ***10x*** mai lentă decât SRAM
  - Hard-disk este de ***10,000x*** mai lent decât DRAM
    - Pentru primul octet, mai rapid pentru următorii

- Consecințe
  - Dimensiuni mari ale paginilor: de obicei 4-8 KB, câteodată 4 MB
  - Complet-asociativ
    - Orice VP poate fi plasată în orice PP
    - Necesită o funcție ”mare” de mapare – diferit față de cache CPU
  - Algoritmi de înlocuire foarte sofisticați și costisitori
    - Prea complicat de implementat în hardware
  - Write-back în loc de write-through

Translatarea adreselor: Tabela de pagini

- Tabela de pagini este un vector de adrese ale paginilor virtuale (page table entries sau PTE) care stabilește corespondența cu paginile fizice. În acest exemplu avem 8 pagini virtuale.

Translatarea adreselor cu Tabela de Pagini

Page Hit

- *Page hit:* referință la un cuvânt din VM care există în memoria fizică

Page Miss

- *Page miss:* referință la un cuvânt din VM care nu există în memoria fizică

Page Fault Handling

- Page miss cauzează un page fault (o excepție)
- Handler-ul de Page Fault selectează o victimă pentru a fi evacuată (aici VP 4)

Page Fault Handling

- Page miss cauzează un page fault (o excepție)
- Handler-ul de Page Fault selectează o victimă pentru a fi evacuată (aici VP 4)

Page Fault Handling

- Handler-ul de Page Fault selectează o victimă pentru a fi evacuată (aici VP 4)
- Instrucțiunea care a cauzat excepție este restartată: page hit!

De ce funcționează? Localitatea datelor

- Memoria virtuală funcționează din cauza localității
- În orice moment de timp, programele tind să acceseze un set de pagini virtuale active, numit și *setul de lucru* (*working set*)
  - Programele care au localitate temporală bună, vor avea și seturi de lucru mai mici
- Dacă (working set size < main memory size)
  - Performanță bună pentru un proces după compulsory miss
- Dacă (SUM(working set sizes) > main memory size)
  - *Thrashing:* Degradarea performanței când facem swap la pagini (le copiem) în continuu din memoria virtuală în fizică și invers

VM ca suport pentru Memory Management

- Idee de bază: fiecare proces are propriul spațiu de adrese virtuale
  - Poate să ”vadă” memoria ca un spațiu liniar
  - Funcția de mapare împrăștie adresele prin memoria fizică
    - O mapare bine aleasă simplifică alocarea și managementul memoriei

VM ca suport pentru Memory Management

- Alocarea memoriei
  - Fiecare pagină virtuală poate fi mapată în orice pagină fizică
  - O pagină virtuală poate fi stocată în pagini fizice diferite la momente diferite de timp
- Partajarea de cod și de date între procese
  - Maparea paginilor virtuale la aceeași pagină fizică (aici: PP 6)

Simplifică Linking și Loading

- Linking
  - Fiecare program are un spațiu virtual de adrese similar
  - Codul, stiva și bibliotecile partajate încep întotdeauna de la aceleași adrese
- Loading
  - `execve()` alocă pagini virtuale pentru secțiunile `.text` și `.data`
    = creează PTE-uri marcate invalid
  - Secțiunile `.text` și `.data` sunt copiate, pagină cu pagină, la cererea sistemului de memorie virtuală

VM ca utilitar pentru protecția memoriei

- Extinde PTE cu biți pentru permisiuni
- Page fault handler verifică acești biți înainte de remapare
  - Dacă sunt corupți, trimite SIGSEGV (segmentation fault)

Translatarea adreselor: Page Hit

1) Procesorul trimite adresa virtuală la MMU

2-3) MMU face fetch la PTE din tabela de pagini în memorie

4) MMU trimite adresa fizică la cache/memorie

5) Cache/memoria trimite cuvântul de date la procesor

Translatarea adreselor: Page Fault

1) Procesorul trimite adresa virtuală la MMU

2-3) MMU face fetch la PTE din tabela de pagini în memorie

4)Valid bit este zero, MMU declanșează page fault exception

5)Handler-ul identifică victima (și, dacă e ”murdară”, o paginează pe disc)

6)Handler-ul aduce o nouă pagină și actualizează PTE în memorie

7) Handler-ul face return la procesul original, restartând instrucțiunea care a generat excepția

Facilitarea tranzacțiilor prin TLB

- Page table entries (PTEs) sunt stocate în cache L1 ca orice alt cuvânt din memorie
  - PTE-urile pot fi invalidate din cache ca orice alte referințe la date
  - PTE hit în L1 tot necesită o întârziere de 1 (sau 2) cicli
- Soluție: *Translation Lookaside Buffer* (TLB)
  - Cache hardware de mici dimensiuni în MMU
  - Mapează numerele de pagini virtuale la numere corespunzătoare de pagini fizice
  - Conține tabele de pagini întregi pentru un număr mic de pagini

TLB Hit

TLB hit elimină necesitatea unui acces la memorie

TLB Miss

TLB miss produce un acces suplimentar la memorie (pentru a aduce un PTE)

Din fericire, TLB miss sunt rare

Exemplu simplu

- Adresare
  - Adrese virtuale pe 14-biți
  - Adrese fizice pe 12-biți
  - Page size = 64 bytes

Exemplu de tabelă de pagini

Doar primele 16 intrări (din 256)

VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	*PPN*	*Valid*
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Sistem simplu de memorie cu TLB

- 16 intrări
- 4-way associative

Set	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Cache pentru sistemul simplu de memorie

- 16 linii, 4 octeți per bloc
- Adresare fizică
- Mapat direct

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Exemplu de translatare de adrese

Adresă virtuală: **0x03D4**

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO					

VPN **0x0F** TLBI **3** TLBT **0x03** TLB Hit? **Y** Page Fault? **N** PPN: **0x0D**

Adresă fizică

CT						CI					CO
11	10	9	8	7	6	5	4	3	2	1	0
0	0										
PPN						PPO					

CO **0** CI **0x5** CT **0x0D** Hit? **Y** Byte: **0x36**

Alt exemplu

Adresă virtuală: **0x0B8F**

TLBT ← → TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	0	0	1	1	1	1

VPN ← → VPO

VPN **0x2E** TLBI 2 TLBT **0x0B** TLB Hit? N Page Fault? Y PPN: **TBD**

Adresa fizică

CT ← → CI ← → CO

11	10	9	8	7	6	5	4	3	2	1	0

PPN ← → PPO

CO ___ CI___ CT ____ Hit? __ Byte: _____

Alocarea paginilor virtuale

- Exemplu: alocarea VP 5
  - Kernel-ul alocă VP 5 pe disc și pointează PTE 5 spre ea

Translatarea adreselor:
putting it all together

Politici de înlocuire a paginilor

Least-recently used (LRU)

Implementată prin menținerea unei stive

Pagini →		A	B	A	F	B	E	A
LRU stack								
MRU	D	A	B	A	F	B	E	A
	B	D	A	B	A	F	B	E
	E	B	D	D	B	A	F	B
LRU	C	E	E	E	D	D	A	F

Memoria principală și memoria de masă

Definim setul de lucru al unui proces (working set), $W(t, x)$: Setul de pagini accesat de ultimele x instrucțiuni la timpul t

Localitatea datelor asigură faptul că setul de lucru se modifică lent

Variația în dimensiune a setului de lucru pentru un program

Impactul politicii de înlocuire asupra performanței

Rata de page fault în funcție de numărul de pagini locate și de politica de înlocuire a paginilor

Memoria virtuală - concluzii

- Memoria virtuală mărește **capacitatea**
- Avem un subset de pagini virtuale în memoria fizică
- **Tabela de pagini** mapează paginile virtuale pe pagini fizice – translatare de adrese
- **TLB** mărește viteza cu care se fac translațiile adreselor
- Tabele de pagini diferite pentru programe diferite asigură **protecția memoriei**

Ierarhia memoriilor - concluzii

Memoria cache: dă iluzia unei viteze mari de execuție

Memoria principală: cost rezonabil, dar lentă și mică

Memoria virtuală: dă iluzia unei dimensiuni foarte mari a memoriei

Localitatea datelor face ca toate iluziile să funcționeze

Fluxul de date într-o ierarhie de memorii

Sistemul de memorii - rezumat

- L1/L2 Cache
  - Există numai pentru a accelera execuția
  - Comportamentul este invizibil programatorului de aplicații și (unei mari părți a) sistemului de operare
  - Implementate în întregime în hardware
- Memoria virtuală
  - Este fundamentul multor funcții ale SO
    - Process creation, task switching, protection
  - Software
    - Alocă/partajează memoria fizică între procese
    - Construiește tabele care urmăresc tipul memoriei, sursele, partajarea
    - Exception handling. Completează tabelele de mapare
  - Hardware
    - Translatează adresele virtuale via tabelele de mapare, impune permisiuni
    - Accelerează maparea prin intermediul TLB

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

– Cursul 6 –

Dispozitive periferice

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the Day

http://dilbert.com/strips/comic/2007-01-25/

I/O Devices

- Ce este un dispozitiv?
- Registre
  - Exemplu: NS16550 UART
- Întreruperi
- Direct Memory Access (DMA)
- PCI (Peripheral Component Interconnect)
- Rezumat

Ce este un device?

Concret, pentru un programator de SO:

- Un hardware care este vizibil din software
- Ocupă un spațiu de adresă pe un bus
- La adrese sunt mapate registre
  - Spațiul I/O mapat ca o memorie
- Generează întreruperi
- Poate să inițializeze transferuri Direct Memory Access (DMA)

Registre

- CPU poate să încarce dintr-un registru al unui device I/O:
  - Obține informații legate de status
  - Citește date de intrare

- CPU poate să încarce într-un registru al unui device I/O:
  - Setează starea dispozitivului și configurația
  - Scrie datele de ieșire
  - Resetează starea dispozitivului

I/O Mapat în memorie

- Procesorul accesează dispozitivele I/O în aceeași manieră ca memoria (tastaturi, monitoare, imprimante)
- Fiecare dispozitiv I/O are atribuite una sau mai multe adrese
- Atunci când adresa este pusă de procesor pe magistrală, datele sunt citite/scrise din I/O în loc din memorie
- Un segment al spațiului de adresă este dedicat dispozitivelor I/O

Hardware pentru maparea I/O

- **Decodificator de adrese:**
  - Inspectează adresele pentru a determina care dispozitiv/memorie comunică cu procesorul
- **Registre I/O:**
  - Memorează valorile scrise către dispozitivele I/O
- **Multiplexor pentru datele citite:**
  - Selectează care este sursa de date (memorie sau dispozitive I/O) care trebuie să ajungă la procesor

Interfața cu memoria

Hardware pentru maparea I/O

Cod pentru maparea I/O

- Presupunem că Dispozitivul I/O 1 are adresa 0xFFFFFFF4
  - **Scrie valoarea 42 la I/O Device 1**
  - **Citește date din I/O Device 1 și plasează-le în `$t3`**

Cod pentru maparea I/O

- **Scrie valoarea 42 în I/O Device 1 (0xFFFFFFF4)**

```
addi $t0, $0, 42
sw $t0, 0xFFF4($0)
```

Cod pentru maparea I/O

- **Citește valoarea de la I/O Device 1 și pune-o în $t3**

```
lw $t3, 0xFFF4($0)
```

Sisteme Input/Output (I/O)

- Sisteme I/O Embedded
  - Cuptoare cu microunde, mașini de spălat, routere, nave spațiale, etc.
- PC I/O Systems

Sisteme I/O embedded

- Exemplu microcontroller: AVR

– microcontroller

– Procesor pe 8 biți, arhitectură RISC

– Periferice low-level:

  - Porturi de I/O
  - Interfețe seriale (RS232, I2C, SPI)
  - Timere
  - Convertoare analog-digitale

I/O Digital

```
// C Code
#include <avr/io.h>


int main(void) {
  unsigned char switches;
  DDRD = 0xFF;              //PORTD output
  DDRC = 0x00;              //PORTC input


  while (1) {
  switches = PINC & 0x0F; //read & mask switches
  PORTD = switches;        //display on LEDs
  }
  }
```

I/O Serial

- Exemple protocoale seriale
  - **SPI:** Serial Peripheral Interface
  - **UART:** Universal Asynchronous Receiver/Transmitter
  - De asemenea: I^2C, USB, Ethernet, etc.

SPI: Serial Peripheral Interface

- Master-ul inițializează comunicația cu slave-ul prin generearea de impulsuri de ceas pe linia SCK
- Master-ul trimite SDO (Serial Data Out) către slavee, msb first
- Slave-ul trimite datele (SDI) la master, msb first

(a)

(b)

UART: Universal Asynchronous Rx/Tx

- Configurare:
  - Bit de start (0), 7-8 biți date, bit paritate (opțional), 1+ biți stop (1)
  - data rate: 300, 1200, 2400, 9600, …115200 baud
- Linia rămâne pe idle în starea HIGH (1)
- Configurație tipică:
  - 8 biți date, fără paritate, 1 bit stop, 9600 baud

Timere

```
void initTimer1(void)
{
        TCCR1A = 0x00;    //prescale the timer to be clock source/1024
        TCCR1B = _BV(WGM12)|_BV(CS12)|_BV(CS10);
        OCR1A = 10000;    // match 1Hz
        //set 8-bit Timer/Counter1 Output Compare Interrupt Enable
        TIMSK |= _BV(OCIE1A);
}

ISR(TIMER1_COMPA_vect)
{
        PORTB ^=0x01; //blink an LED once a second
}

int main(void)
{
        DDRB = 0xFF;
        initTimer1();
        sei();

        while(1)
        {}
        return 0;
}
```

I/O Analogic

- Necesar pentru a interfața procesorul cu mediul în care funcționează
- **Analog input:** Conversie Analog-to-Digital (A/D)
  - De cele mai multe ori este inclus în microcontroller
  - *N*-biți: convertește o tensiune analogică din gama V_{ref-}-V_{ref+} într-un întreg de la 0-2^{N-1}
- **Analog output:**
  - Conversie Digital-to-Analog (D/A)
    - De obicei este nevoie de un circuit exterior (e.g., AD558 or LTC1257)
    - *N*-biți: convertește semnalul digital de la 0-2^{N-1} înapoi la V_{ref-}-V_{ref+}
  - Pulse-width modulation

Pulse-Width Modulation (PWM)

- Valoarea medie este proporțională cu factorul de umplere

Duty cycle = $\frac{\text{Pulse width}}{\text{Period}}$

•Adăugăm un filtru trece-sus pentru a transforma trenul de impulsuri într-o tensiune analogică de valoare = valuarea medie a semnalului PWM

Alte periferice întâlnite pe un uController

- Exemple
  - LCD cu caractere
  - MonitorVGA
  - Transceiver Bluetooth, WiFi
  - Motoare

Sisteme I/O pentru Personal Computers (PC)

- USB: Universal Serial Bus
  - USB 1.0 apărut în 1996
  - Cabluri și conectori standard/software pentru periferice
- PCI/PCIe: Peripheral Component Interconnect/PCI Express
  - Dezvoltat de Intel, apare în 1994
  - Magistrală paralelă pe 32-biți
  - Folosit pentru plăci de expansiune (de ex. Placă de sunet, video, ethernet etc.)
- DDR: double-data rate memory

Sisteme I/O pentru Personal Computers (PC)

- TCP/IP: Transmission Control Protocol and Internet Protocol
  - Conexiune fizică: cablu Ethernet sau Wi-Fi
- SATA: interfață pentru hard-drive
- Input/Output (senzori, actuatoare, microcontrollere etc.)
  - Data Acquisition Systems (DAQs)
  - USB Links

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

– Cursul 7 –

Dispozitive periferice (2)

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Sisteme I/O pentru Personal Computers (PC)

- USB: Universal Serial Bus
  - USB 1.0 apărut în 1996
  - Cabluri și conectori standard/software pentru periferice
- PCI/PCIe: Peripheral Component Interconnect/PCI Express
  - Dezvoltat de Intel, apare în 1994
  - Magistrală paralelă pe 32-biți
  - Folosit pentru plăci de expansiune (de ex. Placă de sunet, video, ethernet etc.)
- DDR: double-data rate memory

Sisteme I/O pentru Personal Computers (PC)

- TCP/IP: Transmission Control Protocol and Internet Protocol
  - Conexiune fizică: cablu Ethernet sau Wi-Fi
- SATA: interfață pentru hard-drive
- Input/Output (senzori, actuatoare, microcontrollere etc.)
  - Data Acquisition Systems (DAQs)
  - USB Links

Sisteme I/O

Processor
interrupts
Cache
Memory - I/O Bus
Main Memory
I/O Controller
I/O Controller
I/O Controller
Disk
Disk
Graphics
Network

Ce este o magistrală?

O magistrală este:

- O legătură partajată de comunicație
- Un singur set de linii de legătură folosit pentru a conecta diferite subsisteme

- O magistrală este o componentă vitală în construcția unor sisteme complexe de calcul
  - Metodă sistematică de abstractizare

Avantajele magistralelor

- Versatilitate:
  - Dispozitivele noi pot fi adăugate cu ușurință
  - Perifericele pot fi mutate între diverse sisteme de calcul care respectă același standard
- Cost redus:
  - Un singur set de fire de legătură este partajat de toate perifericele

Dezavantajele magistralelor

- Creează o gâtuire în comunicație (bottleneck)
  - Lățimea de bandă a magistralei de memorie limitează I/O throughput
- Viteza maximă de transfer de date este limitată de:
  - Lungimea magistralei
  - Numărul de dispozitive de pe magistrală
  - Necesitatea de-a conecta o gamă largă de dispozitive, fiecare cu:
    - Latențe diferite
    - Viteze de transfer diferite

Organizarea generală a unei magistrale

- Linii de control:
  - Signal requests and acknowledgments
  - Indică ce fel de informație circulă pe liniile de date la acel moment
- Linii de date transferă informația de la sursă la destinație:
  - Date și adrese
  - Comenzi complexe

Master versus Slave

- O tranzacție pe magistrală se face în doi pași:
  - Emiterea comenzii (și a adresei) – cerere
  - Transferul de date – acțiune
- Master-ul este cel care începe orice tranzacție:
  - Emite comanda (și adresa slave-ului)
- Slave-ul este cel care răspunde comenzii:
  - Trimite datele master-ului, dacă master-ul cere datele
  - Recepționează date de la master, dacă master-ul dorește să trimită datele

TECHNOLOGY

'Master' and 'slave' computer labels unacceptable, officials say

Wednesday, November 26, 2003 Posted: 2024 GMT (4:24 AM HKT)

LOS ANGELES, California (Reuters) -- Los Angeles officials have asked that manufacturers, suppliers and contractors stop using the terms "master" and "slave" on computer equipment, saying such terms are unacceptable and offensive.

The request -- which has some suppliers furious and others busy re-labeling components -- came after an unidentified worker spotted a videotape machine carrying devices labeled "master" and "slave" and filed a discrimination complaint with the county's Office of Affirmative Action Compliance.

In the computer industry, "master" and "slave" are used to refer to primary and secondary hard disk drives. The terms are also used in other industries.

"Based on the cultural diversity and sensitivity of Los Angeles County, this is not

advertisement

Story Tools
SAVE THIS E-MAIL THIS
PRINT THIS MOST POPULAR

RELATED
• Business 2.0: A new language for the server room

Tipuri de magistrale

- Magistrală Procesor-Memorie (design specific)
  - Scurtă și de viteză mare
  - Trebuie doar să conecteze sistemul de memorie
    - Maximizează lățimea de bandă memory-to-processor
  - Conectată direct la procesor
  - Optimizată pentru transferuri de blocuri cache
- Magistrală I/O (industry standard)
  - De obicei, de lungime mai mare și mai lentă
  - Trebuie să acomodeze o gamă largă de dispozitive I/O
  - Se conectează la magistrala procesor-memorie sau backplane bus
- Backplane Bus (standard / proprietar)
  - Backplane: o structură de interconectare din interiorul șasiului
  - Permite procesoarelor, memoriei și dispozitivelor I/O să coexiste
  - Avantaje de cost: o magistrală pentru toate componentele
- Magistrale Seriale (tendință generală de interconectare serială)

Sistem de calcul cu o magistrală: Backplane Bus

- O singură magistrală (backplane bus) este folosită pentru:
  - Comunicație processor-to-memory
  - Comunicația dintre dispozitivele I/O și memorie
- Avantaje: Simplu și cost redus
- Dezavantaje: lent / magistrala poate deveni un bottleneck
- Exemplu: IBM PC - AT

Sistem cu două magistrale

- Magistralele I/O se conectează la magistrala principală prin adaptoare:
  - Processor-memory bus: în special pentru traficul procesor-memorie
  - I/O buses: furnizează conectori de expansiune pentru diferitele dispozitive I/O
- Apple Macintosh-II
  - NuBus: Procesor, memorie și câteva dispozitive I/O
  - SCCI Bus: restul de dispozitive I/O

Sistem cu trei magistrale (+ backside cache)

- Un număr mic de magistrale backplane conectate la magistrala procesor-memorie
  - Magistrala procesor-memorie este optimizată pentru traficul procesor-memorie
  - Magistralele I/O sunt conectate la magistrala backplane
- Avantaj: încărcarea magistralei de procesor este cu mult redusă

Trecerea de la I/O paralel la I/O serial

- Frecvența ceasului pe un bus paralel e limitată de lungimea magistralei (~100MHz)
- Consum de energie mare pentru a conecta un număr mare de periferice
- Central bus arbiter crește latența fiecărei tranzacții
- Conectori scumpi, multe linii de interconectare ce cresc costurile (orice linie în plus adaugă un cost suplimentar)
- Exemple: VMEbus, Sbus, ISA bus, PCI, SCSI, IDE

Linii seriale punct-la-punct dedicate

- Legăturile punct-la-punct funcționează la viteze multi-gigabit folosindcodificări avansate pentru ceas/semnale de date (necesită multă electronică la ambele capete ale comunicației)
- Low power – avem o singură legătură de date cu un singur periferic
- SATA, USB, Firewire, etc.
- Transferuri simultane multiple
- Conectori și cabluri ieftine (mai puțin cupru pentru cabluri/ mai mult siliciu pentru logica de comunicație)
- Fiecare dispozitiv are legătura lui de date, cu parametri proprii de viteză și comunicație
- Exemple: Ethernet, Infiniband, PCI Express,

Diagrama interconexiunilor unei plăci de bază

- Northbridge:
  - Handles memory
  - Graphics
- Southbridge: I/O
  - PCI bus
  - Disk controllers
  - USB controllers
  - Audio
  - Serial I/O
  - Interrupt controller
  - Timers

Exemplu: National Semiconductor ns16550 UART

- *Universal Asynchronous Receiver/Transmitter*
  - Dispozitiv serial RS-232
- Port serial IBM PC standard
  - Chip-ul este integrat acum în Southbridge-ul oricărui PC
  - Încă e folosit la servere PC
  - Nu mai e folosit pe laptopuri si foarte rar întâlnit pe desktop-uri în zilele noastre ☹
- Primul device driver care este scris de obicei pentru un SO…

Registre

- Detaliile de implementare pentru registre date în “datasheets”
- Informațiile de acolo sunt câteodată mai puțin adevărate ☺

Din datasheet-ul NS16550 UART

8.4 LINE STATUS REGISTER

This register provides status information to the CPU concerning the data transfer. Table II shows the contents of the Line Status Register. Details on each bit follow.

Bit 0: This bit is the receiver Data Ready (DR) indicator. Bit 0 is set to a logic 1 whenever a complete incoming character has been received and transferred into the Receiver Buffer Register or the FIFO. Bit 0 is reset to a logic 0 by reading all of the data in the Receiver Buffer Register or the FIFO.

Bit 1: This bit is the Overrun Error (OE) indicator. Bit 1 indicates that data in the Receiver Buffer Register was not read by the CPU before the next character was transferred into the Receiver Buffer Register, thereby destroying the previous character. The OE indicator is set to a logic 1 upon detection of an overrun condition and reset whenever the CPU reads the contents of the Line Status Register. If the FIFO mode data continues to fill the FIFO beyond the trigger level, an overrun error will occur only after the FIFO is full and the next character has been completely received in the shift register. OE is indicated to the CPU as soon as it happens. The character in the shift register is overwritten, but it is not transferred to the FIFO.

Bit 2: This bit is the Parity Error (PE) indicator. Bit 2 indicates that the received data character does not have the correct even or odd parity, as selected by the even parity

Adresarea registrelor

1. Mapate în memorie:
   - Registrele apar ca locații de memorie
   - Accesate folosind load/store (`movb/movw/movl/movq`)
2. "I/O instructions":
   - Spațiu diferit (16 biți) de adrese pentru dispozitive I/O mai vechi
   - Specific (zilele noastre) arhitecturii
   - Instrucțiuni speciale: `inb`, `outb`, etc.
3. Indirecție:
   - Scrie un registru "index" cu un offset, apoi un registru "data" cu valoarea efectivă a registrului
   - Folosit pentru a salva spațiul de adresă (de obicei spațiul I/O)

Registre ns16550 (fiecare are 8 biți)

Addr.	Name	Description	Notes
0	RBR	Receive Buffer Register (read only)	DLAB=0
0	THR	Transmit Holding Register (write only)	DLAB=0
1	IER	Interrupt Enable Register	DLAB=0
2	IIR	Interrupt Identification Register (read only)	
2	FCR	FIFO Control Register (write only)	
3	LCR	Line Control Register	
4	MCR	MODEM Control Register	
5	LSR	Line Status Register	
6	MSR	MODEM Status Register	
7	SCR	Scratch Register	
0	DLL	Divisor Latch (LSB)	DLAB=1
1	DLM	Divisor Latch (MSB)	DLAB=1

DLAB = bit 7 of the LCR register

ns16550 LSR:
Line Status Register

Un driver UART foarte simplu

```
#define UART_BASE 0x3f8
#define UART_THR  (UART_BASE + 0)
#define UART_RBR  (UART_BASE + 0)
#define UART_LSR  (UART_BASE + 5)

void serial_putc(char c)
{
// Wait until FIFO can hold more chars
while( (inb(UART_LSR) & 0x20)== 0);
// Write character to FIFO
outb(UART_THR, c);
}

char serial_getc()
{
// Wait until there is a char to read
while( (inb(UART_LSR) & 0x01) == 0);
// Read from the receive FIFO return
inb(UART_RBR);
}
```

Register addresses from data sheet 0x3f8: location on a PC

Send a character (wait until we can first)

Read a character (spin waiting until one is there to read)

Un driver UART foarte simplu

- Defapt, mult prea simplu!
  - Dar așa arată întotdeauna prima versiune pentru orice…
- Fără cod de inițializare, fără error handling.
- Folosește ***Programmed I/O*** (PIO)
  - CPU citește sau scrie explicit toate valorile din registre
  - Toate datele trec automat prin registrele CPU
- Folosește ***polling***
  - CPU face busy-waiting înainte de fiecare send/receive
    - Buclă strânsă!
  - Nu poate să facă nimic altceva între timp
    - Deși, putem să ne gândim la un sistem de threading și sincronizare…
  - Fără CPU polling, nu putem face nici un fel de operații I/O

Registrele I/O nu sunt memorie

Registrele unui device nu se comportă ca un RAM!

- Conținutul registrelor se schimbă fără scrieri de la CPU
  - Flag-uri (biți) de status
  - Date recepționate
- Scrierile în registre sunt folosite pentru a declanșa acțiuni
  - Trimiterea de date
  - Resetarea unor mașini de stări din periferic

Problema cu cache-urile

- Citirile din I/O nu pot veni din cache
  - Valoarea reg I/O se schimbă ⇒ cache-ul devine **inconsistent**
- Write-back caches (și write buffers) cauzează probleme
  - Nu știi când linia respectivă va fi scrisă efectiv în periferic
- Citirile și scrierile nu pot fi combinate în linii de cache
  - Registrele fac scrierea/citirea la nivel de octet/cuvânt
  - Scrierea unei linii întregi poate să suprascrie și alte registre în afara celor utile
  - Chiar și citirile în masă pot declanșa schimbări în starea unui device

⇒ Accesul la registrele de I/O trebuie să nu implice memoria cache

- Handling în MMU: PTE-urile au un flag “no cache”
- Spațiul I/O oricum nu permite caching

Alte probleme

1. Cum evităm să facem polling?
   - Cum știe CPU-ul când device-ul este gata sau a terminat ultima comandă?
2. Cum lăsăm CPU-ul în pace?
   - Putem face transfer de date fără a trece prin CPU sau cache?
   - CPU-ul poate să facă altceva?
1. De unde vin aceste locații de memorie pentru registre?
   - Cum poate SO să găsească dispozitivele mapate în spațiul fizic de adresă?
   - Cum sunt alocate adresele fizice?

Evităm cu totul polling-ul

- Un CPU nu poate să facă poll pentru fiecare dispozitiv ca să vadă dacă este gata
  - Pierdere de timp
  - Durează prea mult ca să reacționeze

- Soluția: dispozitivele pot să *întrerupă* procesorul
  - Funcționează ca o excepție: salvează starea curentă și sare la o adresă prestabilită din memoria alocată kernelului
  - Informația legată de sursa întreruperii este codificată în *vector*

Întreruperi

- CPU are linii **Interrupt-request** declanșate de dispozitivul I/O
  - Pot fi declanșate pe nivel logic sau pe front
- **Interrupt handler** recepționează cererea de întrerupere
- **Mascabile,** pentru a evita sau întârzia anumit întreruperi
- Vectorul de întrerupere face corespondența între întrerupere și handler-ul corect
  - Bazate pe priorități
  - Unele **nonmascabile**
- Mecanismul de întreruperi e folosit și pentru întreruperi

Vectori excepție la x86

0	Divide error
1	Debug exception
2	**Non-maskable interrupt (NMI)**
3	Breakpoint
4	Overflow
5	Bounds check
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	Coprocessor segment overrun (386 or earlier)
A	Invalid task state segment
B	Segment not present
C	Stack fault
D	General protection fault
E	Page Fault
F	Reserved
10	Math fault
11	Alignment check
12	Machine check
13	SIMD floating point exception
14-1F	Reserved to Intel
20-FF	**Available for external interrupts**

Programmable Interrupt Controller (PIC)

Direct Memory Access

- Evită *I/O programat* pentru majoritatea datelor
  - De ex. rețea rapidă sau interfețe de disc
- Necesită un *controller DMA*
  - De obicei implementat în procesor, în zilele noastre
- Ocolește CPU și transferă datele direct de la I/O la memorie
  - Nu blochează CPU-ul
  - Salvează lățimea de bandă
  - Doar o singură întrerupere pe transfer

Transfer DMA old-style

1. Device driver requests data transfer of S bytes from disk controller; specifies buffer address A and size S to DMA controller

2. Disk controller starts DMA transfer of S bytes

3. Disk controller initiates bus request for each data byte from disk

4. DMA controller transfers byte to address A, increments A, decrements S.

5. When S == 0, DMA controller interrupts CPU to indicate transfer complete

Avantajul de bază al DMA

- **Decuplează** transferul de date de procesarea de date
  - CPU nu trebuie să copieze datele de la/la dispozitiv
  - Nu poluează cache-ul CPU
  - Pot fi procesate când decide CPU (sau SO)
  - Performanță mărită: CPU și device I/O merg în paralel
- Dezavantaje posibile:
  - Overhead mare pentru transferuri foarte mici de date
  - De obicei nu e o problemă (până și UART-urile fac DMA!)

DMA și memoria Cache

- DMA înseamnă ca memoria devine **inconsistentă** cu cache-ul CPU
- Opțiuni:

1. CPU poate să marcheze bufferele DMA ca non-cacheable

⇒ large hit – probabil vrea să proceseze datele oricum

2. Cache poate să facă “snoop” la tranzacțiile DMA (dar nu scalează foarte bine la sistemele multiprocesor)
3. SO poate să golească/invalideze explicit regiuni din cache

⇒ cache management este o parte importantă a driverelor de dispozitive!

DMA și Memoria Virtuală

- Adresele DMA sunt **fizice**
  - Apar pe magistrala externă
- Codul utilizatorilor și al SO lucrează cu adrese **virtuale** (în majoritatea timpului)
- SO (și device drivers) trebuie să translateze manual virtual ↔ fizic atunci când programează controllerele DMA
  - Acest lucru necesită mai mult de o tablelă hardware de pagini!
  - DMA al unei singure regiuni de adrese virtuale s-ar putea să **nu aibă un corespondent contiguu** în spațiul de adrese fizice
  - **Scatter-gather** DMA controllers: DMA de la/la o listă de regiuni
- Sisteme foarte recente: implementează IOMMU
  - Funcționează ca MMU, dar pentru DMA scrie din dispozitive
  - Tot trebuie programat de SO pentru a fi în concordanță cu starea MMU

Unde sunt toate aceste registre?

- De unde știe SO câte dispozitive I/O sunt conectate?
- Unde sunt mapate registrele dispozitivelor în spațiul fizic?
  - Și căror vectori de întrerupere le corespund?
- Soluție: include-le în designul *magistralei de I/O*
  - Exemplu: PCI

PCI este…

Peripheral **C**omponent **I**nterconnect

- Standard electric pentru conectarea dispozitivelor
  - Ca și PCMCIA, PCI-X, PCI-Express, etc.
- Un standard pentru conectorii fizici
- Un set de ”protocoale de bus” pentru comunicația dintre dispozitive
- O interfață vizibilă software-ului pentru operații I/O hardware

PCIe a urmat PCI, dar extinde aceeași interfață software

PCI încearcă să rezolve probleme multiple:

- Device discovery
  - Detecția tuturor dispozitivelor din sistem
- Alocarea de adrese
  - La ce adrese apar diferitele registre ale dispozitivelor conectate?
- Interrupt routing
  - Ce semnale de întrerupere de la fiecare dispozitiv se mapează și în ce vector de excepție?
- DMA inteligent
  - Dispozitive ce au “Bus mastering” nu mai au nevoie de controller DMA

Conexiuni fizice: PCI este un arbore

- PCI bridges form a hierarchy
- Also:
  - PCI-ISA
  - PCI-USB
  - PCI-SCSI
  - Etc.

Spațiul de adresă PCI este plat

- Fiecare device PCI cere un set de adrese
  - Spațiu fizic de adrese (32-biți sau 64-biți)
  - Spațiu adresă I/O (de obicei 16-biți)
- Rezultat:
  - Fiecare dispozitiv apare ca un segment contiguu de adrese
    - În spațiul de memorie
    - În spațiul I/O (doar pe x86)

Dispozitivele PCI sunt self-describing

- Fiecare dispozitiv are un header pentru configurație
  - Accesat prin bridge-ul părinte, inițial
- Câteva câmpuri:

Bits	Description
16	Manufacturer ID (idenfies Intel, 3Com, NVidia, etc.
16	Model ID (specific to manufacturer)
24	Class code (what kind of device is this?)
8	Version identifier

- Plusuri:
  - Alocă automat spațiul de adresă
  - Intreruperi
  - Informații de natură electrică
  - Etc.

Localizarea tuturor dispozitivelor

- Găsește bridge-ul PCI “root”
  - PCI bridge este în vârful arborelui
  - PC-urile mari pot avea mai mult de unul
- Citește configurațiile pentru a găsi toate dispozitivele atașate
  - Adaugă la lista de dispozitive și funcții
  - Înregistează cerințele pentru spațiul de adresă
  - If a bridge, recurse!
- Rezultatul:
  - Lista completă a tuturor dispozitivelor din sistem cu toate cerințele de spațiu de adresă aferente

Alocarea adreselor

- Găsește adresele pentru fiecare dispozitiv și bridge
  Cerințele includ:
  - Fiecare dispozitiv are dată dimensiunea spațiului de adresă necesar
  - Toate dispozitivele de “sub” un bridge au adrese care sunt incluse în spațiul de adresă al bridge-ului
  - Fiecare bridge are un segment de memorie care include toate segmentele de memorie ale tuturor “copiilor”.
  - Fiecare segment este limitat de adrese putere a lui 2
- Apoi programează:
  - Fiecare bridge PCI cu informații legate de translatarea adreselor
  - Fiecare dispozitiv cu registre “base-address/range” (BAR)

Întreruperi PCI

- Patru linii de întrerupere
  - INTA, INTB, INTC, INTD…
  - Bridge-urile permit cablarea arbitrară a liniilor de IRQ a dispozitivelor la cele patru linii
  - Translatate de root bridge în intreruperi de sistem
- PCI Express introduce MSI
  - Message-signalled interrupts
  - Întreruperea codificată ca un PCI write într-un spațiu anume de adrese
  - Translatate de root bridge în intreruperi de sistem
  - Întreruperile pot fi rutate individual către un anumit core/APIC

DMA peste PCI

- PCI permite **Bus Mastering**
  - Device-ul poate emite tranzacții de scriere/citire oriunde în memorie
  - Chiar (în anumite cazuri) spre alte dispozitive PCI
- Controller-ul DMA extern nu mai e relevant
  - Controller integrat în dispozitivul însuși
  - Principiul se aplică: device-ul face DMA pentru date de la/la memorie
  - Mult mai flexibil / dispozitive inteligente

Dispozitive inteligente

- Bus mastering, plus foarte mult spațiu de adresă acum
- Dispozitivele pot acum să acceseze autonom orice:
  - Locație din memoria principală
  - Alte dispozitive
- Permite protocoale complexe pentru interacțiunea CPU ↔ Device
  - Încearcă să țină și CPU și device-ul ocupat în perioadele de activitate mare
  - In RAM buffering
  - “Descriptor rings” – mecanism de schimbare de cereri și răspunsuri

Exemplu: Intel e1000 PCI- Express Ethernet card

- Multiple send queues
- Multiple receive queues
- Hashing of packet headers to queues
- Directing interrupts to different cores
- Packet checksumming in hardware
- etc.

Rezumat

- Dispozitivele și CPU comunică via:
  - Registre I/O mapate în memorie
  - Întreruperi și vectori de întrerupere
  - Direct Memory Access (DMA)

- Magistralele I/O (precum PCI):
  - Permit dispozitivelor să partajeze/aloce adrese fizice
  - Alocă întreruperi
  - Permit bus mastering direct memory access

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

– Cursul 8 –

Magistrale si arhitecturi de calcul

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day

Din episodul anterior

1. **Processor-memory (local) bus:**
   - De obicei scurt și de viteză mare pentru a maximiza lățimea de bandă
2. **I/O Bus**
   - Lung, trebuie să suporte viteze de date diferite de la dispozitive diferite
   - Trebuie să facă față unei game largi de latențe, lățimi de bandă și specificații
3. **Backplane Bus**
   - Este numit așa pentru că era la început pe spatele șasiului
   - Permite interconectarea memoriei, procesorului și a dispozitivelor I/O
   - Necesită logică suplimentară pentru a se lega la alte magistrale

- Magistralele locale sunt de obicei design-specific în timp ce cele de I/O și backplane sunt portabile și de obicei respectă un standard industrial
- Un sistem poate să folosească un backplane sau o combinație de toate trei

Configurații de bus

Exemplu de calcul performanță

Un bus sincron are o perioadă de ceas de 50ns și fiecare transmisie durează un ciclu de ceas. Alt bus asincron are nevoie de 40ns pentru fiecare handshake. Fiecare bus are 32 de biți de date. Care este lățimea de bandă a fiecărui bus pentru citiri de un cuvânt dintr-o memorie cu 200ns per read.

Răspuns:

Pașii pentru magistrala sincronă:

1. Trimite adresa la memorie: 50 ns
2. Citește memoria: 200 ns
3. Trimite datele la dispozitiv: 50 ns

} 300 ns

Lățimea maximă de bandă este 4 octeți la 300 ns $\Rightarrow \frac{4 \text{ bytes}}{300 \text{ ns}} = \frac{4 \text{ MB}}{0.3 \text{ sec}} = 13.3 \text{ MB/sec}$

Exemplu de calcul performanță

Secvența de pași pentru magistrala asincronă:

1. Memory read address atunci când apare ReadReq : 40 ns
2,3,4. Data ready & handshake: max(3x40 ns, 200 ns)= 200 ns
5,6,7. Read & Ack. : 3 × 40 ns = 120 ns

} 360 ns

Lățimea maximă de bandă este 4 octeți la 360 ns $\Rightarrow \frac{4 \text{ bytes}}{360 \text{ ns}} = \frac{4 \text{ MB}}{0.36 \text{ sec}} = 11.1 \text{ MB/sec}$

Creșterea lățimii de bandă

- Lățimea de bandă este de obicei determinată de protocol și de caracteristicile de temporizare a semnalelor
- Alți factori:
  - Lățimea magistralei de date:
    - Transferă cuvinte multiple, necesită mai puțini cicli de bus
    - Creșterea numărului de linii de date (scump!)
  - Multiplexarea liniilor de adrese și date:
    - Folosirea liniilor separate de date și adrese accelerează tranzacțiile
      - Simplifică logica de control pe bus
      - Mărește numărul de linii de date
  - Transfer pe blocuri:
    - Transferă cuvinte multiple de la adrese consecutive în rafală
    - Mărește timpul de răspuns deoarece tranzacțiile vor fi mai lungi
    - Mărește complexitatea logicii de control a magistralei

Bus Master

- Single master
  - Bus master (ex. procesorul) controlează toate accesele pe bus
  - Bus slave (ex. device sau memorie) doar răspunde la cereri
- Multiple master
  - Mai multe dispozitive pot iniția o tranzacție
  - Bus control logic și protocolul adoptat rezolvă conflictele

Arbitrare

- Arbitrarea pe bus coordonează utilizarea magistralei folosind mecanisme de request, grant, release
- Arbitrarea încearcă de obicei să echilibreze doi factori atunci când alege dispozitivul care preia magistrala:
  - Dispozitivele cu prioritate mai mare trebuie servite primele
  - Menține corectitudinea; nici un dispozitiv nu va fi reprimat total
- Timpul de arbitrare este un overhead
  - Vrem să îl minimizăm

Tehnici de arbitrare

- *Distributed arbitration by self-selection:* (ex. NuBus folosit la Apple Macintosh)
  - Folosește linii multiple de request pentru dispozitive
  - Dispozitivele care fac request determină cui i se va da acces
  - Dispozitivele asociază un cod cu cererea, indicând prioritatea
  - Dispozitivele cu prioritate mică cedează busul dacă observă o cerere de prioritate mai mare
- *Distributed arbitration by collision detection:* (ex. Ethernet)
  - Dispozitivele cer independent acces la bus și preiau accesul
  - Cereri simultate produc o coliziune
  - Un algoritm de selecție între entitățile care au generat coliziunea
    - Ethernet: back off and try again later

Tehnici de arbitrare (Cont.)

- Daisy chain arbitration: (e.g. VME bus)
  - Linie specială de Grant Access, partajată de toate dispozitivele în ordinea priorității
  - Dispozitivele de prioritate mai mare arbitrează cererile dispozitivelor cu prioritate mai mică
  - Simplu de implementat
  - Duce la starvation pentru dispoztitivele de prioritate mică
  - Limitează viteza transmisiilor (semnalele de grant ajung cel mai târziu la ultimul device din lanț)
  - Permite propagarea defectelor
    - (o defecțiune a unui dispozitiv duce la defectarea întregului lanț)

Rolul sistemului de operare

- Sistemul de operare are rol de interfață între I/O hardware și programe
- Caracteristici importante ale sistemelor cu I/O:
  - Sistemul I/O este partajat între mai multe programe
  - Sistemele I/O folosesc întreruperi pentru a comunica starea lor procesorului
    - Întreruperile trebuie să fie tratate de SO pentru că transferă execuția în modul supervizor
  - Controlul low-level al unui dispozitiv I/O este complex:
    - Evenimente concurente
    - Cerințele pentru controlul corect al dispozitivului pot să fie foarte complexe

Responsablitățile SO

- Protecție pentru resursele I/O partajate
  - Garantează că un proces nu poate să acceseze I/O-ul altor procese
- Abstractizare pentru accesarea dispozitivelor
  - Rutine care supervizează operarea low-level a dispozitivului (drivere).
  - Interrupt handling pentru I/O
  - Acces echitabil la resursele I/O
    - Toate programele trebuie să aibă acces egal la resursele I/O
  - Planifică accesele pentru a mări productivitatea sistemului

Comunicația cu dispozitivele I/O

- SO trebuie să știe când:
  - Dispozitivul I/O a terminat o operație
  - Dispozitivul I/O a întâlnit o eroare
- Poate fi realizat în două moduri:
  - Polling:
    - Dispozitivul I/O pune informația de stare într-un registru
    - SO verifică periodic registrul de stare
  - I/O Interrupt:
    - Eveniment extern, asincron execuției principale, dar care NU interferează cu rularea codului principal
    - Ori de câte ori un dispozitiv I/O device cere atenție de la procesor, îl întrerupe
    - Unele procesoare tratează întreruperile ca pe niște excepții speciale

Slide: in part David Patterson

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

– Cursul 9 –

Interfațarea cu dispozitivele I/O

Dan Tudose

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day

http://xkcd.com/932/

Dispozitive Input/Output

Cuprins
1. Dispozitive și controllere Input/Output
2. Keyboard and Mouse
3. Unități de display
4. Dispozitive de imprimare
5. Alte dispozitive Input/Output
6. Conectarea în rețea a dispozitivelor Input/Output

intel inside
HEWLETT PACKARD

Dispozitive și controllere de intrare/ieșire

Input type	Prime examples	Other examples	Data rate (b/s)	Main uses
Symbol	Keyboard, keypad	Music note, OCR	10s	Ubiquitous
Position	Mouse, touchpad	Stick, wheel, glove	100s	Ubiquitous
Identity	Barcode reader	Badge, fingerprint	100s	Sales, security
Sensory	Touch, motion, light	Scent, brain signal	100s	Control, security
Audio	Microphone	Phone, radio, tape	1000s	Ubiquitous
Image	Scanner, camera	Graphic tablet	1000s-10^6s	Photos, publishing
Video	Camcorder, DVD	VCR, TV cable	1000s-10^9s	Entertainment
Output type	**Prime examples**	**Other examples**	**Data rate (b/s)**	**Main uses**
Symbol	LCD line segments	LED, status light	10s	Ubiquitous
Position	Stepper motor	Robotic motion	100s	Ubiquitous
Warning	Buzzer, bell, siren	Flashing light	A few	Safety, security
Sensory	Braille text	Scent, brain stimulus	100s	Personal assistance
Audio	Speaker, audiotape	Voice synthesizer	1000s	Ubiquitous
Image	Monitor, printer	Plotter, microfilm	1000s	Ubiquitous
Video	Monitor, TV screen	Film/video recorder	1000s-10^9s	Entertainment
Two-way I/O	**Prime examples**	**Other examples**	**Data rate (b/s)**	**Main uses**
Mass storage	Hard/floppy disk	CD, tape, archive	10^6s	Ubiquitous
Network	Modem, fax, LAN	Cable, DSL, ATM	1000s-10^9s	Ubiquitous

Diagrama unui sistem cu Input/Output

Sistem de input/output printr-un singur bus de date comun.

Organizarea I/O pentru performanță mărită

Input/output prin bus-uri de date intermediare sau dedicate

Tastatură și mouse

Butoanele tastaturii și codificarea lor

(a) Mechanical switch with a plunger

(b) Membrane switch

(c) Logical arrangement of keys

Două tipuri de butoane și schema logică a conectării lor într-o tastatură hexazecimală

Tastatură cu proiecție sau capacitivă

2. A capacitive-type touchscreen (self or absolute capacitive type)—a laminate of multiple substrates (either PET, or glass as shown) or, in some cases a single substrate—contains a matrix of clear conductive electrodes (made by patterning a clear conductor like ITO). This substrate(s) is then laminated to a top cover lens (either glass or PET). When a user's finger touches the cover lens, it increases the measured capacitance of the electrodes closest to the finger. This change in capacitance is used to compute the location of the finger within the capacitive touchscreen.

Dispozitive pentru indicat

Cum funcționează un mouse

(a) Mechanical mouse

(b) Optical mouse

Mouse mecanic (cu bilă) și optic.

Unități de afișare

(a) Image formation on a CRT

(b) Data defining the image

Display CRT și imaginea stocată în frame buffer.

Cum funcționează un display CRT

(a) The RGB color stripes

(b) Use of shadow mask

Schema e culori RGB pentru un display CRT modern.

Codificarea culorilor în formatul RGB

Nuanța și saturația afectează aspectul unei culori (saturație maximă sus, saturație minimă la bază)

Display-uri LCD

(a) Passive display (b) Active display

Display-uri LCD active și pasive

Pixelii într-un ecran LCD modern

Unități de display flexibile

Afișaj subțire flexibil folosind e-ink

Display flexibil Sony organic light-emitting diode (OLED)

E-Ink

1. upper layer
2. transparent electrode layer
3. transparent micro-capsules
4. positive charged white pigments
5. negative charged black pigments
6. transparent oil
7. electrode pixel layer
8. bottom supporting layer
9. light
10. white
11. black

Alte tehnologii pentru display-uri

Dispozitive de tipărire

Mecanismul de scanare al imaginii într-un copiator modern.

Formarea caracterelor într-o imprimantă matriceală

Formarea literei ”D” folosind matrici de puncte de dimesiuni diferite

Simularea diferitelor niveluri de instensitate prin difuzia punctelor

Formarea a cinci neveluri de gri pe un dispozitiv care suportă doar alb și negru (e.g., imprimantă ink-jet sau laser)

Folosirea acelorași modele în RGB pentru producerea a $5 \times 5 \times 5 = 125$ culori diferite

Mecanism de imprimare pentru o imprimantă dot-matrix

Unitate de tipărire

(a) Ink jet printing

(b) Laser printing

Imprimante Ink-jet și Laser.

Cum funcționează imprimantele color

Schema RGB pentru un monitor este aditivă: diferite cantități de culoare primară sunt adăugate pentru a forma culoarea dorită.

Absența lui VERDE

Schema de culori CMY pentru imprimate este substractivă: diferite cantități de culoare primară sunt substrase din alb pentru a forma culoarea dorită

Pentru a produce o nuanță de negru mai bună este folosită schema CMYK (K = black)

Procesul de imprimare CMYK

Iluzia unei culori pure folosind puncte CMYK

Roți de culoare

Paleta de culoare pentru artiști, folosită pentru amestecarea vopselurilor

Paleta de culori substractivă pentru imprimare (CMYK)

Paletă de culori aditivă, folosită în display-uri

Culorile primare apar la centru și la distanțe egale de-a lungul perimetrului paletei. Culorile secundare sunt la mijloc între culorile primare. Culorile terțiare sunt între ulorile primare și secundare.

Alte dispozitive de I/O

Senzori și actuatoare

Colectează informații despre mediul înconjurător

- Senzori de lumină (fotocelule)
- Senzori de temperatură (contact și noncontact)
- Senzori de presiune

(a) Initial state

(a) After rotation

Principiul de operare pentru un motor pas cu pas

Convertirea mișcării circulare în mișcare liniară

Locomotivă

Șurub

Legarea în rețea a mai multor dispozitive de calcul

Pentru perifericele cu capabilități de rețea, I/O este făcut cu ajutorul transferului de fișiere.

Input/Output în sistemele de control și embedded

Structura unui sistem de control în buclă închisă.

Programarea Input/Output

Ca orice altceva, I/O-ul este controlat prin instrucțiuni în cod mașină

- Adresarea I/O (memory-mapped) și performanța acestuia
- Scheduled vs demand-based I/O: polling vs interrupts

Cuprins
1. Performanța I/O și benchmarking
2. Adresarea Input/Output
3. I/O planificat: Polling
4. Demand-Based I/O: Interrupts
5. Transfer de date I/O și DMA
6. Îmbunătățirea performanțelor I/O

Performanță și benchmarking pentru I/O

Zidul I/O

O aplicațe de control industrial petrecea 90% din timp pentu operații cu CPU când a fost dezvoltată în anii 80. De atunci, componenta CPU fost actualizată la fiecare 5 ani dar componenta I/O a rămas neschimbată. Presupunând că performanța CPU s-a îmbunătățit de 10x cu fiecare upgrade, determinați fracțiunea de timp petrecută în operații I/O pe întreaga durată de viață a sistemului.

Soluție

Aplicăm legea lui Amdahl pentru 90% din task cu speedup de 10, 100, 1000 și 10000 peste o perioadă de 20 de ani. Pe parcursul acestor upgrade-uri, timpul de funcționare a fost redus de la originalul 1 la 0.1 + 0.9/10 = 0.19, 0.109, 0.1009 și 0.10009, deci fracția de timp petrecut în operații I/O este 52.6, 91.7, 99.1 respectiv 99.9%. Ultimele două upgrade-uri de CPU nu au ajutat foarte mult.

Tipuri de Input/Output Benchmarking

Benchmarking pentru supercomputing I/O

Citirea unei cantități mari de date de intrare
Scrierea multor instanțe pentru checkpointing
Salvarea unui set foarte mic de rezultate
Productivitatea I/O, în MB/s, e importantă

Benchmarking pentru procesarea tranzacțiilor I/O

Bază de date imensă dar fiecare tranzacție este de mici dimensiuni
Câteva (2-10) accesuri la disc per tranzacție
Rata I/O (accese la disc pe secundă) este importantă

Benchmarking pentru file system I/O

Crearea unui fișier, managementul directoarelor, indexare...
Benchmark-urile sunt de obicei specifice domeniului

Adresarea perifericelor I/O

Registre de control și date pentru tastatură și unitatea de display la procesorul MiniMIPS

Hardware pentru adresarea I/O

Logica de adresare pentru un controller I/O

Citirea datelor dintr-un periferic I/O

Exemplu de cod pentru procesorul MiniMIPS care așteaptă până când tastatura are de transmis un simbol apoi citește simbolul respectiv în registrul `$v0`.

Soluție

Programul examinează în continuu registrul de control al tastaturii și iese din ciclul de ”busy waiting” atunci când bitul R a fost activat.

```
        lui  $t0,0xffff      # put 0xffff0000 in $t0
idle:   lw   $t1,0($t0)      # get keyboard's control word
        andi $t1,$t1,0x0001  # isolate the LSB (R bit)
        beq  $t1,$zero,idle  # if not ready (R = 0), wait
        lw   $v0,4($t0)      # retrieve data from keyboard
```

Acest tip de citire este potrivit numai dacă procesorul așteaptă un input critic de la utilizator și nu poate continua în absența acestuia.

Scrierea de date într-un periferic I/O

Exemplu de secvență de cod pentru procesorul MiniMIPS pentru scrierea în unitatea de afișare. Programul așteaptă până când unitatea de display este gata să accepte un nou simbol apoi transferă conținutul registrului `$a0`

Soluție

Programul examinează în continuu registrul de control al unității de display și iese din ciclul de busy waiting atunci când bitul R este activat de periferic.

```
      lui  $t0,0xffff      # put 0xffff0000 in $t0
idle: lw   $t1,8($t0)      # get display's control word
      andi $t1,$t1,0x0001  # isolate the LSB (R bit)
      beq  $t1,$zero,idle  # if not ready (R = 0), wait
      sw   $a0,12($t0)     # supply data to display unit
```

Acest tip de scriere este potrivit doar dacă ne permitem să avem un procesor dedicat scrierii în unitatea de display.

I/O planificat: Polling

Ce procent din timpul de execuție al unui procesor de 1GHz este petrecut în polling dacă fiecare acces durează 800 cicli de ceas?

Tastatura trebuie să fie interogată cel puțin de 10 ori pe secundă

Unitatea floppy trimite pachete de 4 octeți la o rată de 50 KB/s

Hard discul trimite pachete de 4 octeți la o rată de 3 MB/s

Soluție

Pentru tastatură, împărțim numărul de cicli necesari pentru 10 interogări cu numărul de cicli disponibili într-o secundă:
$(10 \times 800)/10^9 \cong 0.001\%$

Unitatea floppy trebuie interogată de 50K/4 = 12.5K ori pe secundă
$(12.5K \times 800)/10^9 \cong 1\%$

Unitatea hard disk trebuie interogată 3M/4 = 750K ori pe secundă
$(750K \times 800)/10^9 \cong 60\%$

I/O asincron: Întreruperi

Luăm hard disk-ul din exemplul anterior (transferul de pachete de 4B la 3 MB/s atunci când e activ). Presupunem că discul este activ 5% din timp.

Overhead-ul generat de întreruperea CPU-ului și efectuarea transferului este de 1200 cicli de ceas. Ce procent din timpul total de execuție al unui CPU de 1GHz este petrecut în operații cu hard disk-ul?

Soluție

Atunci când e activ, hard disk-ul produce 750K întreruperi pe secundă

$0.05 \times (750K \times 1200)/10^9 \cong 4.5\%$ (în comparație, aveam 60% pt polling)

Chiar dacă întreruperea CPU-ului pentru polling generează o penalizare mai mare, din cauză că discul este in general nefolosit, operațiile bazate pe întreruperi duc la performanță mărită.

Transferul de date I/O și DMA

Un controller DMA partajează aceeași magistrală de memorie cu procesorul.

Operații DMA

(a) DMA transfer in one continuous burst

(b) DMA transfer in several shorter bursts

Funcționarea DMA și semnalele asociate de control de pe magistrală.

Îmbunătățirea performanțelor I/O

Lățimea de bandă I/O efectivă pentru o unitate de disc

Avem un hard disk cu sectoare de 512B și o latență medie de acces de date de 10ms la rata maximă de transfer de 10MB/s. Afișați grafic variația lățimii de bandă efective pe măsură ce unitatea de transfer de date (bloc de date) variază în dimensiune de la 1 sector (0.5 KB) la 1024 sectoare (500 KB).

Soluție

Calcularea productivității efective

Lățimea de bandă efectivă pentru I/O cu unitatea de disc

Timp total de acces pentru x octeți = 10 ms + xfer time = $(0.01 + 10^{-7}x)$ s

Timp de acces efectiv pentru un octet = $(0.01 + 10^{-7}x)/x$ s/B

Rata efectivă de trasfer = $x/(0.01 + 10^{-7}x)$ B/s

Pentru x = 100 KB: Rata efectivă de transfer = $10^5/(0.01 + 10^{-2}) = 5\times10^6$ B/s

Latență medie = 10 ms

Throughput maxim = 10 MB/s

Input/Output distribuit

Exemplu de configurație pentru I/O distribuit Infiniband

Magistrale, legături și interfațare

Magistralele de date partajate sunt des întâlnire în calculatoarele moderne:

- Mai puține fire și conexiuni, flexibilitate și extendabilitate
- Trebuie să implementeze mecanisme de arbitrare și sincronizare

Cuprins
1. Legături Intra- și Inter-sistem
2. Legături între sisteme
3. Protocoale de comunicație pe magistrală
4. Arbitrare și performanță
5. Interfațare
6. Standarde pentru interfațare

Legături inter și intra-sistem

(a) Cross section of layers

(b) 3D view of wires on multiple metal layers

Conectivitatea în interiorul microprocesoarelor și a circuitelor imprimate (PCB) este realizată prin straturi multiple de conductori.

Straturi multiple metalizate pe un microchip sau un PCB

Secțiune transversală prin straturile conductoare

Elementele active și conexiunile dintre ele

-Chipurile moderne au în jur de 8-9 straturi de conductor metalizat
-Straturile superioare conțin de obicei liniile mai lungi sau conexiunile pentru alimentare

Legături între sisteme

(a) RS-232 (b) Ethernet (c) ATM

Exemple de scheme de conexiune pentru diferite protocoale

Interfața serială RS-232. Conectorul de 9 pini și semnalele standard.

Medii de comunicație a datelor

Twisted pair

Coaxial cable

Optical fiber

Cele mai folosite medii de comunicație de date pentru transmisia prin cablu.

Comparație între legăturile dintre sisteme

Performanțele celor trei protocoale de conectare prezentate

Proprietăți	RS-232	Ethernet	ATM
Lungime maximă segment (m)	~10m	~100m	~1000m
Lungime maximă rețea(m)	~10m	~100m	Nelimitat
Bit rate (Mb/s)	Up to 0.02	10/100/1000	155-2500
Unitate de transmisie (B)	1	~100	53
Latență maximă (ms)	< 1	10-100	100
Domeniul tipic de aplicație	Input/Output	LAN	Backbone
Complexitatea și costul unității	Mic	Mic	Mare

Magistralele de date

Conexiunile punct la punct dintre n unități necesită $n(n-1)$ canale, sau $n(n-1)/2$ legături bidirecționale; adică, $O(n^2)$ legături

Conectivitatea pe bus necesită doar un port de intrare și unul de ieșire pentru fiecare unitate, adică $O(n)$ conexiuni în total.

Tipuri de magistrale și semnale de date

Cele trei tipuri de linii care se găsesc într-o magistrală.

Un sistem ce calcul tipic poate să conțină în jur de 10 magistrale diferite de date:

1. Magistrale Legacy: PC bus, ISA, RS-232, port paralel
2. Magistrale Standard: PCI, SCSI, USB, Ethernet
3. Magistrale Proprietare: acces de mare viteză sau dedicat

Protocoale de comunicație pe magistrală

Magistrală sincronă cu dispozitive de latență fixă.

Handshaking pe o magistrală asincronă pentru o operație de date (e.g., o citire din memorie).

Exemplu de comunicație pe bus

Operație I/O read pe un bus PCI.

Arbitrarea accesului la magistrala de date

Structura generalizată a unui arbitru centralizat de magistrală.

Implementări simple de arbitri

Round robin

Fixed-priority

Starvation avoidance

Dacă avem priorități fixe, unitățile cu prioritate mică pot să nu aibă niciodată acces la bus (pot să ”moară de foame”)

Prioritatea trebuie combinată cu un mecanism de asigurare a serviciilor

Rotating priority

Idee: Ordonăm circular unitățile și permitem ca prioritatea cea mai mare să se ”rotească” între unități (combinăm un numărător circular cu un circuit de acordare a priorității)

Daisy Chaining

Daisy chaining permite unui arbitru de bus central să acceseze un număr mare de dispozitive care folosesc o resursă partajată.

Elementele de bază ale interfațării

Giruetă care dă o tensiune de ieșire de 0-5V în funcție de direcția vântului.

Standarde de interfațare

Sumar a patru dintre cele mai folosite standarde de interfațare

Attributes ↓ Name →	PCI	SCSI	FireWire	USB
Type of bus	Backplane	Parallel I/O	Serial I/O	Serial I/O
Standard designation	PCI	ANSI X3.131	IEEE 1394	USB 2.0
Typical application domain	System	Fast I/O	Fast I/O	Low-cost I/O
Bus width (data bits)	32-64	8-32	2	1
Peak bandwidth (MB/s)	133-512	5-40	12.5-50	0.2-15
Maximum number of devices	1024*	7-31#	63	127$
Maximum span (m)	< 1	3-25	4.5-72$	5-30$
Arbitration method	Centralized	Self-select	Distributed	Daisy chain
Transceiver complexity or cost	High	Medium	Medium	Low

Notes: * 32 per bus segment; # One less than bus width; $ With hubs (repeaters)

Conectori standard

Pin 1: +5V DC
Pin 4: Ground

Pin 2: Data –
Pin 3: Data +

Conectori USB și structura de conectivitate.

Pin 1: 8-40V DC, 1.5 A
Pin 2: Ground
Pin 3: Twisted pair B –
Pin 4: Twisted pair B +
Pin 5: Twisted pair A –
Pin 6: Twisted pair A +
Shell: Outer shield

Conector IEEE 1394 (FireWire). Același conector este folosit la ambele capete.

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

- **Cursul 9 –**

ILP & Superscalar

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day

Din episoadele anterioare

- Sistemele moderne de memorie paginată pun la dispoziție:
  - Translație, Protecție, Memorie Virtuală.
- Informațiile legate de translație și protecție stocate în tabele de pagini, ținute în memoria principală
- Informațiile legate de translație și protecție caching în “translation-lookaside buffer” (TLB) pentru a permite translatarea + verificarea protecției într-un singur ciclu de ceas, în majoritatea cazurilor
- Memoria virtuală interacționează cu design-ul memoriei cache

Complex Pipelining: Motivație

Banda de asamblare devine complexă atunci când avem nevoie de performanță mărită pentru:

- Unități floating-point cu latență mare sau care sunt doar parțial în b.a.
- Sisteme de memorie cu timp variabil de acces
- Unități aritmetice și de memorie multiple

Floating-Point Unit (FPU)

- Mult mai mult hardware decât o unitate pe întregi
  - Single-cycle FPU nu este o idee fezabilă
- E comun să avem mai multe FPU-uri
- E comun să avem diferite tipuri de FPU-uri: Fadd, Fmul, Fdiv, …
- Un FPU poate să fie în b.a., parțial în b.a. sau fără b.a.
- Pentru a folosi concurent mai multe FPU-uri, trebuie să avem nevoie de o tabelă de registre dedicată (FPR) care să aibă mai multe porturi de citire și de scriere

Caracteristici pentru unitățile funcționale

fully pipelined

partially pipelined

Unitățile funcționale au registre interne dedicate b.a.

⇒ operanzii sunt memorați atunci când o instrucțiune intră în unitatea funcțională

⇒ instrucțiunile următoare pot rescrie tabela de registre în timpul operațiilor cu latență mare

Floating-Point ISA

- Interacțiunea dintre căile de date pentru întregi și floating-point sunt determinate de ISA
- RISC-V ISA
  - tabele de registre separate pentru instrucțiuni FP și cu întregi
    - Singura interacțiune este printr-un set de instrucțiuni tip move/convert (unele ISA nu permit nici măcar asta)
  - load/store separate pentru FPR și GPR, dar ambele folosesc GPR pentru calculul adreselor

Sisteme de memorie reale

Abordări comune pentru îmbunătățirea performanței memoriei:

- Cache-uri - single cycle, mai puțin în cazul unui miss

  → stall

- Memorie pe bancuri – accese multiple la memorie

  → conflicte pe bancuri

- Operații în două faze (separă cererea la memorie de răspuns), majoritatea fără pregătiri inițiale

  → răspunsuri out-of-order

Latența acceselor la memoria principală este de obicei mult mai mare de un ciclu, și de cele mai multe ori, total imprevizibilă

Găsirea unei soluții este o problemă centrală pentru arhitectura calculatoarelor

Probleme în controlul b.a. complexe

- Conflicte structurale în etapa de execuție dacă un FPU sau o unitate de memorie nu este în b.a. și durează mai mult de un ciclu
- Conflicte structurale la write-back datorate latențelor variabile ale diferitelor unități funcționale
- Hazarde la un write out-of-order datorate latențelor diferitelor unități funcționale
- Cum să rezolvăm excepțiile?

Complex In-Order Pipeline

- Întârzie writeback a.î. toate operațiile au aceeași latență la etapa W
  - Porturile de Write nu sunt niciodată supra-aglomerate (one inst. in & one inst. out la fiecare ciclu)
  - Întârzie b.a. la op. cu latență mare ex.: divide, cache miss
  - Tratează excepțiile in-order la commit point

Cum putem să prevenim ca latența mare la writeback să afecteze operațiile simple pe întregi (single-cycle)? *Bypassing*

In-Order Superscalar Pipeline

- Fetch la două instrucțiuni per ciclu; lansează ambele simultan dacă una este integer/memory și cealaltă este floating point
- Mod eficient de a mări productivitatea, de ex. Alpha 21064 (1992) & Seria MIPS R5000 (1996)
- Aceeași idee poate fi extinsă prin duplicarea unităților funcționale (ex. 4-issue UltraSPARC & Alpha 21164) dar porturile suplimentare de GPR și costurile pt. Bypassing cresc rapid

Tipuri de hazarde de date

Luați, de exemplu, următorul tip de operație

$r_k \rightarrow r_i \text{ op } r_j$

Dependențe de date

r_3 -> r_1 op r_2 Read-after-Write

r_5 -> r_3 op r_4 (RAW) hazard

Anti-dependență

r_3 -> r_1 op r_2 Write-after-Read

r_1 -> r_4 op r_5 (WAR) hazard

Dependență de ieșiri

r_3 -> r_1 op r_2 Write-after-Write

r_3 -> r_6 op r_7 (WAW) hazard

Dependențe de registre vs. memorie

Hazardele de date datorate operanzilor din registre pot fi depistate în etapa de decode, dar hazardele datorate operanzilor din memorie pot fi determinate numai după calculul adresei efective

Store: **`M[r1 + disp1] <- r2`**

Load: **`r3 <- M[r4 + disp2]`**

Unde apare aici hazardul?

Oare **`(r1 + disp1) = (r4 + disp2)`** ?

Hazarde de date, un exemplu

I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMUL.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

RAW Hazards
WAR Hazards
WAW Hazards

Instruction Scheduling

I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMULT.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

Ordonări valide:

in-order	I_1	I_2	I_3	I_4	I_5	I_6
out-of-order	I_2	I_1	I_3	I_4	I_5	I_6
out-of-order	I_1	I_2	I_3	I_5	I_4	I_6

Out-of-order Completion

In-order Issue

					Latency
I_1	FDIV.D	f6,	f6,	f4	*4*
I_2	FLD	f2,	45(x3)		*1*
I_3	FMULT.D	f0,	f2,	f4	*3*
I_4	FDIV.D	f8,	f6,	f2	*4*
I_5	FSUB.D	f10,	f0,	f6	*1*
I_6	FADD.D	f6,	f8,	f2	*1*

in-order comp 1 2 1 2 3 4 3 5 4 6 5 6

out-of-order comp 1 2 2 3 1 4 3 5 5 4 6 6

B.A. Complexe

Putem să rezolvăm hazardele la write fără să egalăm toate b.a. și fără să facem bypass?

Când este sigur să lansăm o instrucțiune?

Presupunem că o structură de date ține evidența tuturor instrucțiunilor din toate unitățile funcționale

Trebuie făcute următoarele verificări înainte de a lansa în execuție o instrucțiune:

- Este disponibilă unitate funcțională cerută?
- Sunt disponibile datele de intrare? → RAW?
- Este sigur să scrii la destinație? →WAR? -> WAW?
- Există vreun conflict structural în etapa de WB?

Structură de date pentru execuții corecte

Ține evidența tuturor unităților funcționale

Name	*Busy*	*Op*	*Dest*	*Src1*	*Src2*
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

Instrucțiunea i, la lansare în execuție, consultă tabela

FU available?	check the busy column
RAW?	search the dest column for i's sources
WAR?	search the source columns for i's destination
WAW?	search the dest column for i's destination

Se adaugă o intrare în tabelă dacă nu se detectează nici un hazard
Intrarea este scoasă din tabelă după Write-Back

Simplificarea structurii de date
Presupunem execuție In-order

Presupunem că instrucțiunea nu este trimisă în execuție dacă există un hazard RAW sau unit. funcț. este ocupată, și că operanzii sunt memorați intern de către unit. funcț. la primire:

Instrucțiunea trimisă poate să cauzeze:

WAR hazard ?

NU: Operanzii sunt citiți la lansare în execuție

WAW hazard ?

DA: Out-of-order completion

Simplificăm structura de date ...

- Nu avem hazard WAR

  -> nu e nevoie să memorăm src1 și src2

- Nu se lansează în execuție o operație în cazul unui hazard WAW

  -> numele unui registru poate apare cel mult odată în coloana *dest*

- WP[reg#] : vector de biți pentru memorarea regitrelor în care se vor face operații de scriere (Write Pending)

  –Biți setați la lansare și șterși în etapa de WB

  -> Fiecare etapă din b.a. a fiecărei unit. funcț. trebuie să conțină câmpul dest și un flag care să indice dacă acesta este valid “perechea (we, ws)”

Tabelă de scoruri pentru execuție In-order

Busy[FU#] : vector de biți ce indică disponibilitatea FU.

(FU = Int, Add, Mult, Div)

Acești biți sunt hardcodați în FU.

WP[reg#] : vector de biți care să înregistreze dacă scrierile în reg. sunt în așteptare (Write Pending).

Acești biți sunt setați la lansarea în execuție a op. și șterși la Write Back

Unitatea verifică instrucțiunea (opcode dest src1 src2) în tabelă înainte de a o lansa în execuție

FU available?	Busy[FU#]
RAW?	WP[src1] sau WP[src2]
WAR?	*nu poate apare*
WAW?	WP[dest]

Dinamica tabelei de scoruri

	Functional Unit Status										Registers Reserved for Writes	
	Int(1)	Add(1)	Mult(3)			Div(4)				WB		
t0	I_1					f6					f6	
t1	I_2 f2						f6				f6, f2	
t2								f6		f2	f6, f2	I_2
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	I_1
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	I_3
t7	I_5	f10						f8			f8, f10	
t8									f8	f10	f8, f10	I_5
t9										f8	f8	I_4
t10	I_6	f6									f6	
t11										f6	f6	I_6

I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMULT.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

Limitările execuției In-Order: un exemplu

					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	f4,	f2,	f8	*4*
6	FADD.D	f10,	f6,	f4	*1*

In-order: 1 (2,1) . . . . . 2 3 4 4 3 5 . . . 5 6 6

Restricțiile execuției In-order previn ca instrucțiunea 4 să fie lansată

Execuția Out-of-Order

- Buffer-ul din etapa Issue ține mai multe instrucțiuni care așteaptă lansarea în execuție.
- Decode adaugă instrucțiunea următoare la buffer dacă este loc și dacă instrucțiunea nu cauzează un hazard WAR sau WAW.
  - Notă: WAR este posibil deoarece execuția este out-of-order (WAR nu era posibil pentru in-order cu memoriarea operanzilor în FU)
- Orice instrucțiune din buffer pentru care hazardele RAW sunt rezolvate, poate fi lansată. La write back (WB) se pot introduce în buffer noi instrucțiuni.

Limitări: In-Order și Out-of-Order

					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	f4,	f2,	f8	*4*
6	FADD.D	f10,	f6,	f4	*1*

In-order: 1 (2,1) . . . . . 2 3 4 4 3 5 . . . 5 6 6

Out-of-order: 1 (2,1) 4 4 . . . . 2 3 . . 3 5 . . . 5 6 6

Execuția Out-of-order nu ne aduce o îmbunătățire semnificativă!

Câte instrucțiuni pot fi ținute în b.a.?

Care dintre proprietățile ISA limitează numărul de instrucțiuni din b.a?

Numărul de registre

Execuția Out-of-order de sine stătătoare nu aduce nici o îmbunătățire seminificativă!

Rezolvarea lipsei de registre

B.a. Floating Point nu pot fi umplute de cele mai multe ori cu un număr mic de registre.

IBM 360 avea doar 4 registre floating-point

Poate o microarhitectură să folosească mai multe registre decât cele specificate în ISA fără să piardă compatibilitatea cu ISA ?

Robert Tomasulo de la IBM a sugerat o soluție ingenioasă în 1967 folosind tehnica *register renaming*

Limitări: In-Order și Out-of-Order

					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		long
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	**f4'**,	f2,	f8	*4*
6	FADD.D	f10,	f6,	**f4'**	*1*

In-order: 1 (2,1) . . . . . 2 3 4 4 3 5 . . . 5 6 6

Out-of-order: 1 (2,1) 4 4 5 . . . 2 (3,5) 3 6 6

Orice antidependență poate fi eliminată prin redenumire.

(renaming != spațiu de stocare suplimentar)

Poate fi făcut în hardware? *DA!*

Register Renaming

- Etapa Decode face register renaming și adaugă instrucțiunile etapei Issue în instruction ReOrder Buffer (ROB)
  - redenumirea face imposibile hazardele WAR sau WAW

- Orice instrucțiune din ROB al carei hazarde RAW au fost satisfăcute poate fi lansată.
  - Out-of-order or dataflow execution

Structuri folosite la renaming

- Instruction template (i.e., tag t) este alocat de etapa Decode care asociază acesta cu registrul din tabela de registre
- După execuția instrucțiunii, tag-ul este dealocat

Management-ul Reorder Buffer

ROB este alocat circular

- Bitul "exec" este setat când instrucțiunea începe execuția
- Când o instrucțiune a terminat execuția, bitul "use" este șters
- ptr_2 este incrementat doar dacă bitul "use" este marcat ca liber

Un slot de instrucțiune este gata de execuție când:

- Conține o instrucțiune validă (bitul "use" e setat)
- Nu a început încă execuția (bitul "exec" este liber)
- Ambii operanzi sunt disponibili (p1 și p2 sunt setați)

Renaming & Out-of-order Issue

Un exemplu

Renaming table

	p	data
f1		
f2		t1
f3		
f4		t5
f5		
f6		t3
f7		
f8		t4

v1

Reorder buffer

	Ins#	use	exec	op	p1	src1	p2	src2
t_1	1	0	0	LD				
t_2	2	0	0	LD				
t_3	3	1	0	MUL	0	t2	1	v1
t_4	4	0	0	SUB	1	v1	1	v1
t_5	5	1	0	DIV	1	v1	0	t4
.								
.								

data / t_i

1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *Când sunt tag-urile înlocuite de date?*
  Oricând o unit. funcț. produce date
- *Când poate fi reutilizat un nume?*

La terminarea execuției unei instrucțiuni

IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967

Magistrala comună asigură faptul că datele sunt disponibile imediat pentru toate instrucțiunile care au nevoie de ele.

Compară tag-urile, dacă egale atunci copiază valoarea & și setează Presence Bit "p".

Eficiență?

Register Renaming și Out-of-order execution au fost implementate prima dată în 1969 la IBM 360/91 dar nu au apărut în modelele următoare până în anii 90.

De ce ?

Motive

1. Eficiente doar pentru o clasă foarte mică de programe
2. Latența memoriei era o problemă mult mai mare
3. Excepțiile nu sunt precise!

Încă o problemă trebuia rezolvată mai întâi

Transferul controlului

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

- **Cursul 10 –**

Multiprocesoare

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day

http://xkcd.com/538/

Recapitulare: Memoria Virtuală

- Suport pentru memorie virtuală implementat standard în toate procesoarele moderne
  - Creează iluzia unui spațiu amplu și partajat de memorie protejată
  - Programele pot fi scrise independent de configurația de memorie a mașinii pe care acestea rulează
- Tabelele de pagini ierarhice exploatează faptul că spațiul de adrese virtual nu este compact pentru a reduce dimensiunea informațiilor legate de mapare
- TLB cache translation/protection – informații care fac VM practică
  - Nu ar fi acceptabil să avem referințe la memorie multiple pentru o instrucțiune
- Interacțiunea dintre TLB lookup și cache tag lookup
  - Vrem să evităm inconsistențele la adresarea virtuală

Uniprocessor Performance (SPECint)

- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Déjà vu?

"… today's processors … are nearing an impasse as technologies approach the speed of light.."

David Mitchell, *The Transputer: The Time Is Now* (1989)

- Transputer-ul nu a apărut pe piață la momentul potrivit (Performanțele uniprocesor↑)
  ⇒ Procrastinarea recompensată: 2X seq. perf. / 1.5 ani
- "We are dedicating all of our future product development to multicore designs. … This is a sea change in computing"

  Paul Otellini, President, Intel (2005)
- Toate companiile de microprocesoare trec la MP (2X CPUs / 2 yrs)
  ⇒ Procrastinarea penalizată: 2X sequential perf. / 5 ani

Manufacturer/Year	AMD/'07	Intel/'07	IBM/'07	Sun/'07
Processors/chip	4	2	2	8
Threads/Processor	1	1	2	8
Threads/chip	4	2	4	64

Alți Factori ⇒ Multiprocesoare

- Creșterea aplicațiilor data-intensive
  - Baze de date, servere fișiere, …
- Creșterea interesului în servere, performanța serverelor
- Creșterea performanțelor desktop-urilor nu mai e așa de importantă
  - Mai puțin partea de grafică
- Înțelegere mai bună a cum pot fi folosite eficient multiprocesoarele
- Avantajul reducerii costurilor de design prin replicare
  - Comparativ cu reproiectarea de la (aproape) zero

Taxonomia Flynn

M.J. Flynn, "Very High-Speed Computers", *Proc. of the IEEE*, V 54, 1900-1909, Dec. 1966.

Embedded Systems Laboratory

- Flynn - clasificare după flux de date și de control (1966)

Single Instruction, Single Data (SISD) (Uniprocessor)	Single Instruction, Multiple Data SIMD (single PC: Vector, CM-2)
Multiple Instruction, Single Data (MISD) (????)	Multiple Instruction, Multiple Data MIMD (Clusters, SMP servers)

- SIMD $\Rightarrow$ Data-Level Parallelism
- MIMD $\Rightarrow$ Thread-Level Parallelism
- MIMD populare datorită
  - Flexibilității: N programe sau 1 program multithreaded
  - Cost-effective: același MPU într-un desktop și în mașina MIMD

SISD

- Sisteme Uniprocesor

MISD

- Nu există exemple comerciale
  - Aplică aceleași operații pe un set de date și găsește numerele prime

SIMD

- Vector/Array Computers

MIMD

- Message Passing
- Shared memory/distributed memory
    - Uniform Memory Access (UMA)
    - Non-Uniform Memory Access (NUMA)

Back to Basics

- “Un calculator paralel este o colecție de elemente de procesare care cooperează și comunică pentru a rezolva probleme mari de calcul într-un timp scurt.”
- Arhitectura Paralelă = Arhitectura Calculatoarelor + Arhitecturi de Comunicație

Două modele pentru Arhitectura de memorie și comunicații

1. Comunicațiile apar prin pasarea explicită de mesaje între procesoare: message-passing multiprocessors (aka multicomputers)
   - Sistemele moderne tip *cluster* conțin unități de calcul independente ce comunică prin mesaje
2. Comunicațiile se petrec prin intermediul unui spațiu de adresă partajat (via operații load și store): shared-memory multiprocessors care pot fi:
   - UMA (Uniform Memory Access time) cu adrese partajate și memorie centralizată
   - NUMA (Non-Uniform Memory Access time multiprocessor) cu adrese partajate și memorie distribuită

Memorie centralizată vs. distribuită

Memorie centralizată **Memorie distribuită**

Memorie Centralizată Partajată

- Procesoarele partajează o singură memorie centralizată (UMA) printr-o magistrală de interconexiuni
- Fezabil pentru un număr mic de procesoare
- Arhitecturile cu memorie centralizată sunt cea mai comună formă de design MIMD

Memorie Distribuită

Folosește memorie distribuită fizic (NUMA) ce permite un număr mare de procesoare

Avantaje:

- Permite scalarea lățimii de bandă a memoriei
- Reduce latența memoriei

Dezavantaj:

- Complexitate mărită în comunicația de date

Procesoare cu memorie centralizată

Embedded Systems Laboratory

- Numite și symmetric multiprocessors (SMPs) deoarece memoria partajată unică are o relație simetrică cu toate procesoarele
- Cache mare $\Rightarrow$ o singură memorie poate să satisfacă cererile unui număr mic de procesoare
- Poate să scaleze la câteva zeci de procesoare prin folosirea unui switch și a mai multor bancuri de memorie
- Deși, d.p.d.v. tehnic se poate scala și mai mult, devine mai puțin atrăgător pe măsură ce crește numărul de procesoare ce partajează aceeași memorie centralizată

Multiprocesoare cu memorie distribuită

- **Pro:** Metodă Cost-effective de a scala lățimea de bandă
  - Doar dacă majoritatea acceselor sunt la memoria locală
- **Pro:** Reduce latența acceselor locale la memorie
- **Contra:** Comunicația de date dintre procesoare e mai complexă
- **Contra:** Software-ul trebuie să fie conștient de localitatea datelor pentru a profita de lățimea de bandă mărită

Problemele procesării paralele

Embedded Systems Laboratory

- O mare problemă e că % dintr-un program este inerent secvențial
  - Ce înseamnă inerent secvențial?
- Presupunem 80X speedup de la 100 procesoare. Care este procentul original de program care este paralelizabil?

a. 10%

b. 5%

c. 1%

d. <1%

Răspunsul, conform legii lui Amdahl

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times \left(\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}\right) = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

Comunicare și sincronizare

Embedded Systems Laboratory

- Procesele paralele trebuie să coopereze pentru a termina mai repede un singur task
- Necesită comunicație distribuită și sincronizare
  - Comunicație pentru valorile datelor, sau ”ce”
  - Sincronizare pentru control, sau ”când”
  - Comunicația și sincronizarea sunt deobicei inter-dependente
    - Ex. “ce” depinde de “când”
- Message-passing agregă datele și semnalele de control
  - Sosirea mesajelor codifică “ce” și “când”
- În mașinile cu memorie partajată, comunicația este menținută prin cache-uri coerente și sincronizarea prin operații atomice cu memoria
  - Datorită apariției multiprocesoarelor single-chip, este foarte posibil ca sistemele cache-coherent cu memorie partajată să fie forma dominantă de multiprocesoare
  - Cursul de astăzi se axează pe problema sincronizării

Multiprocesoare simetrice

simetrie

- Toate memoriile sunt la distanțe egale de toate procesoarele
- Orice procesor poate să facă orice operație de I/O (setează un transfer DMA)

Sincronizare

Embedded Systems Laboratory

Nevoia de sincronizare apare de fiecare dată când avem procese concurente într-un sistem
(chiar și într-unul uniprocesor)

Forks & Joins: În programarea paralelă, un proces poate să aștepte până când s-au produs mai multe evenimente

Producer-Consumer: Un proces consumator trebuie să aștepte până ce un proces producător a generat datele

Folosirea exclusivă a unei resurse: Sistemul de operare trebuie să asigure că doar un singur proces utilizează o resursă la un moment dat

Exemplu poducător-consumator

Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
$R_{tail}=R_{tail}+1$
Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if $R_{head}==R_{tail}$ goto spin
Load R, (R_{head})
$R_{head}=R_{head}+1$
Store (head), R_{head}
process(R)

Programul este scris presupunând că instrucțiunile sunt executate în ordine.

Probleme?

Catedra de Calculatoare
Facultatea de Automatica si Calculatoare

Exemplu producător-consumator

Embedded Systems Laboratory

Producer posting Item x:

Load R_{tail}, (tail)

1 Store (R_{tail}), x

$R_{tail}=R_{tail}+1$

2 Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)

spin: Load R_{tail}, (tail) 3

if $R_{head}==R_{tail}$ goto spin

Load R, (R_{head}) 4

$R_{head}=R_{head}+1$

Store (head), R_{head}

process(R)

Poate fi actualizat pointer-ul tail înainte ca elementul x să fie stocat?

Programatorul presupune că dacă 3 se execută după 2, atunci 4 se execută după 1.

Secvențele cu probleme sunt:

2, 3, 4, 1

4, 1, 2, 3

Consistența secvențială

Model de memorie

" A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =

întrețesere arbitrară cu păstrarea ordinei referințelor la memorie pentru programele secvențiale

Consistența secvențială

Task-uri secvențiale concurente: T1, T2
Variabile partajate: X, Y (inițial X = 0, Y = 10)

T1:
Store (X), 1 *(X = 1)*
Store (Y), 11 *(Y = 11)*

T2:
Load R_1, (Y)
Store (Y'), R_1 *(Y'= Y)*
Load R_2, (X)
Store (X'), R_2 *(X'= X)*

Care sunt răspunsurile corecte pentru X' și Y' ?

(X',Y') ε {(1,11), (0,10), (1,10), ~~(0,11)~~} ?

Dacă y este 11 atunci x nu poate fi 0

Consistența secvențială

Consistența secvențială impune mai multe contrângeri de ordonare de memorie ca și cele impuse de dependențele de memorie ale programelor uni-procesor (⟶)

Care sunt cele din exemplele noastre ?

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

⟶ Cerințe adiționale SC

T2:

Load R_1, (Y)

Store (Y'), R_1 *(Y'= Y)*

Load R_2, (X)

Store (X'), R_2 *(X'= X)*

Poate un sistem cu cache și out-of-order execution să pună la dispoziție o imagine consistentă secvențial a memoriei?

Mai multe despre asta mai târziu

Exemplu consumatori multipli

Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
$R_{tail}=R_{tail}+1$
Store (tail), R_{tail}

Consumer:

spin: Load R_{head}, (head)
Load R_{tail}, (tail)
if $R_{head}==R_{tail}$ goto spin
Load R, (R_{head})
$R_{head}=R_{head}+1$
Store (head), R_{head}
process(R)

Secțiune critică:
Trebuie executată atomic de un singur consumator ⇒ locks

Ce nu e în regulă cu acest cod?

Locks or Semaphores

E. W. Dijkstra, 1965

Embedded Systems Laboratory

Un semafor (mutex) este un întreg ne-negativ ce implementează următoarele operații:

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P() și V() trebuie executate atomic, adică fără

- *intreruperi* sau
- *accese intercalate la s* de către alte procesoare

```
Process i
    P(s)
        <critical section>
    V(s)
```

Valoarea inițială a lui s determină numărul maxim de procese din regiunea critică

Implemetarea semafoarelor

Semafoarele (mutual exclusion) pot fi implementate folosind instrucțiuni obișnuite Load and Store în modelul unei memorii cu Consistență Secvențială. Cu toate acestea, protocoalele de excluziune mutuală sunt greu de proiectat...

Soluție mai simplă:

instrucțiuni atomice read-modify-write

Exemple: *m este o locație de memorie, R un registru*

Test&Set (m), R:
$R \leftarrow M[m]$;
if R==0 *then*
$M[m] \leftarrow 1$;

Fetch&Add (m), R_V, R:
$R \leftarrow M[m]$;
$M[m] \leftarrow R + R_V$;

Swap (m), R:
$R_t \leftarrow M[m]$;
$M[m] \leftarrow R$;
$R \leftarrow R_t$;

Exemplu consumatori multipli *Test&Set Instruction*

```
P:      Test&Set (mutex),R_temp
        if (R_temp!=0) goto P
        Load R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head=R_head+1
        Store (head), R_head
V:      Store (mutex),0
        process(R)
```

Critical Section

Alte instrucțiuni atomice read-modify-write (Swap, Fetch&Add, etc.) pot de asemenea să implementeze P și V

Ce se întâmplă dacă procesul se oprește în regiunea critică?

Sincronizare nonblocantă

Embedded Systems Laboratory

```
Compare&Swap(m), Rt, Rs:
  if (Rt==M[m])
      then  M[m]=Rs;
            Rs=Rt ;
            status ← success;
      else  status ← fail;
```

status *este un argument implicit*

```
try:   Load Rhead, (head)
spin:  Load Rtail, (tail)
       if Rhead==Rtail goto spin
       Load R, (Rhead)
       Rnewhead = Rhead+1
       Compare&Swap(head), Rhead, Rnewhead
       if (status==fail) goto try
       process(R)
```

Load-reserve & Store-conditional

Embedded Systems Laboratory

Registre speciale pentru a ține flag-ul și adresa și rezultatul lui store-conditional

```
Load-reserve R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs'
          reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-reserve R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head = R_head + 1
        Store-conditional (head), R_head
        if (status==fail) goto try
        process(R)
```

Performanța mutex-urilor

Instrucțiuni blocante atomice read-modify-write

e.g., Test&Set, Fetch&Add, Swap

vs

Instrucțiuni atomice non-blocante read-modify-write

e.g., Compare&Swap,

Load-reserve/Store-conditional

vs

Protocoale bazate pe operații Load Store obișnuite

Performanța depinde de mai mulți factori:

degree of contention,

cache-uri,

out-of-order execution și Loads & Stores

mai târziu in curs ...

Probleme în implementarea Consistenței Secvențiale

Implentarea CS este complicată de două probleme

- Capabilități de execuție *Out-of-order*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Cache-uri*

  Cache-urile pot preveni ca efectul unui store să fie văzut de alte procesoare

Bariere la memorie

Instrucțiuni care secvențializează accesul la memorie

Embedded Systems Laboratory

Procesoarele cu arhitecturi de memorie slab-cuplate (ex. Permit reordonarea op. Loads & Stores la adrese diferite) trebuie să implementeze bariere la memorie (instrucțiuni) pentru a forța serializarea acceselor la memorie

Exemple de procesoare cu memorii slab-cuplate:

- Sparc V8 (TSO,PSO): Membar
- Sparc V9 (RMO):
  - Membar #LoadLoad, Membar #LoadStore
  - Membar #StoreLoad, Membar #StoreStore
- PowerPC (WO): Sync, EIEIO

Barierele la memorie (Memory fences) sunt operații costisitoare dar pot fi folosite numai atunci când sunt necesare

Folosirea barierelor la memorie

Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
$Membar_{SS}$
$R_{tail}=R_{tail}+1$
Store (tail), R_{tail}

ensures that tail ptr is not updated before x has been stored

Consumer:

Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if $R_{head}==R_{tail}$ goto spin
$Membar_{LL}$
Load R, (R_{head})
$R_{head}=R_{head}+1$
Store (head), R_{head}
process(R)

ensures that R is not loaded before x has been stored

Data-Race Free Programs

a.k.a. Properly Synchronized Programs

Embedded Systems Laboratory

Process 1

```
...
Acquire(mutex);
  < critical section>
Release(mutex);
```

Process 2

```
...
Acquire(mutex);
  < critical section>
Release(mutex);
```

Variabilele de sincronizare (e.g. mutex) sunt separate de variabilele de date

Accesele la variabilele partajate de date sunt protejate în regiunile critice

$\Rightarrow$ *nu avem conflicte de date în afară de lock-uri*

În general, nu poate fi dovedit că un program este lipsit de conflicte de date.

Bariere pentru programele concurente

Process 1

```
...
Acquire(mutex);
membar;
    < critical section>
membar;
Release(mutex);
```

Process 2

```
...
Acquire(mutex);
membar;
    < critical section>
membar;
Release(mutex);
```

- Modelul de memorie permite reordonarea instrucțiunilor de către compilator, cât timp această reordonare nu este făcută peste o barieră
- Procesorul nu trebuie să facă prefetch sau să speculeze peste o barieră

Excluziune mutuală folosind Load/Store

Un protocol bazat pe două variabile partajate c1 și c2.
Inițial c1 și c2 sunt 0 *(not busy)*

Process 1

```
     ...
    c1=1;
L:  if c2=1 then go to L
      < critical section>
    c1=0;
```

Process 2

```
     ...
    c2=1;
L:  if c1=1 then go to L
      < critical section>
    c2=0;
```

Care este problema?

Deadlock!

Excludere mutuală: *a doua încercare*

Embedded Systems Laboratory

Pentru a evita *deadlock*, lasă un proces să renunțe la lock (i.e. Process 1 setează c1 la 0) cât timp așteaptă.

Process 1

```
    ...
L:  c1=1;
    if c2=1 then
        { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
    ...
L:  c2=1;
    if c1=1 then
        { c2=0; go to L}
      < critical section>
    c2=0
```

- Deadlock nu mai este posibil dar este o posibilitate redusă de *livelock*.

- Un proces nenorocos poate să nu intre niciodată în secțiunea critică $\Rightarrow$ *starvation*

Un protocol pentru excludere mutuală

T. Dekker, 1966

Embedded Systems Laboratory

Protocol bazat pe trei variabile partajate c1, c2 și turn.
Inițial, c1 și c2 sunt 0 *(not busy)*

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

- turn = *i* asigură că doar procesul i poate să aștepte
- variabilele c1 și c2 asigură *excluderea mutuală*

Soluția pentr n procese a fost dată de Dijkstra și este puțin mai complicată!

Analiza algoritmului lui Dekker

Embedded Systems Laboratory

Scenario 1

```
    ...                 Process 1
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                        then go to L
      < critical section>
    c1=0;
```

```
    ...                 Process 2
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                        then go to L
      < critical section>
    c2=0;
```

Scenario 2

```
    ...                 Process 1
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                        then go to L
      < critical section>
    c1=0;
```

```
    ...                 Process 2
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                        then go to L
      < critical section>
    c2=0;
```

N-process Mutual Exclusion

Lamport's Bakery Algorithm

Embedded Systems Laboratory

Process i

Initially num[j] = 0, for all j

Entry Code

```
choosing[i] = 1;
num[i] = max(num[0], …, num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++)  {
    while( choosing[j] );
    while( num[j] &&
            ( ( num[j] < num[i] ) ||
              ( num[j] == num[i] &&  j < i ) ) );
}
```

Exit Code

```
num[i] = 0;
```

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Embedded Systems Laboratory

Catedra de Calculatoare

Facultatea de Automatica si Calculatoare

Calculatoare Numerice (2)

– Cursul 12 –

Multiprocesoare 2

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day

http://dilbert.com/strips/comic/1992-07-18/

Memoria partajată centralizată

Structură multiprocesor cu memorie partajată

Rețele de interconectare Procesor-Memorie

(a) Butterfly network (b) Beneš network

Rețelele fluture și Beneš: exemple de rețele de interconectare procesor-memorie

Rețele de interconectare Procesor-Memorie

Interconectarea a opt procesoare la 256 bancuri de memorie la Cray Y-MP (1988), un supercomputer cu procesoare vectoriale multiple

Consistența secvențială

Model de memorie

" A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =

întrețesere arbitrară cu păstrarea ordinei referințelor la memorie pentru programele secvențiale

Consistența secvențială

Task-uri secvențiale concurente: T1, T2
Variabile partajate: X, Y (inițial X = 0, Y = 10)

T1:
Store (X), 1 *(X = 1)*
Store (Y), 11 *(Y = 11)*

T2:
Load R_1, (Y)
Store (Y'), R_1 *(Y'= Y)*
Load R_2, (X)
Store (X'), R_2 *(X'= X)*

Care sunt răspunsurile corecte pentru X' și Y' ?

(X',Y') ε {(1,11), (0,10), (1,10), ~~(0,11)~~} ?

Dacă y este 11 atunci x nu poate fi 0

Consistența secvențială

Consistența secvențială impune mai multe contrângeri de ordonare de memorie ca și cele impuse de dependențele de memorie ale programelor uni-procesor (⟶)

Care sunt cele din exemplele noastre ?

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)

Store (Y'), R_1 *(Y'= Y)*

Load R_2, (X)

Store (X'), R_2 *(X'= X)*

⟶ Cerințe adiționale SC

Poate un sistem cu cache și out-of-order execution să pună la dispoziție o imagine consistentă secvențial a memoriei?

Excluziunea mutuala si instructiuni blocante

Instrucțiuni blocante atomice read-modify-write

e.g., Test&Set, Fetch&Add, Swap

vs

Instrucțiuni atomice non-blocante read-modify-write

e.g., Compare&Swap,

Load-reserve/Store-conditional

vs

Protocoale bazate pe operații Load Store obișnuite

Performanța depinde de mai mulți factori:

degree of contention,

cache-uri,

out-of-order execution și Loads & Stores

Probleme în implementarea Consistenței Secvențiale

Implentarea CS este complicată de două probleme

- Capabilități de execuție *Out-of-order*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Cache-uri*

  Cache-urile pot preveni ca efectul unui store să fie văzut de alte procesoare

Consistența memoriei la SMP-uri

Presupunem că CPU-1 actualizează A la 200.

write-back: *memoria si cache*-2 au valori vechi

write-through: cache-2 are valoarea veche

Contează aceste valori neactualizate?

Cum este văzută memoria partajată de software?

Write-back Caches & SC

Embedded Systems Laboratory

	prog T1	cache-1	memory	cache-2	prog T2

	ST X, 1
	ST Y,11
		X= 1
		Y=11
			X = 0
			Y =10
			X'=
			Y'=
				Y =
				Y'=
				X =
				X'=
					LD Y, R1
					ST Y', R1
					LD X, R2
					ST X',R2

• T1 is executed					
• cache-1 writes back Y		X= 1 Y=11	X = 0 Y =11 X'= Y'=	Y = Y'= X = X'=	
• T2 executed		X= 1 Y=11	X = 0 Y =11 X'= Y'=	Y = 11 Y'= 11 X = 0 X'= 0	
• cache-1 writes back X		X= 1 Y=11	X = 1 Y =11 X'= Y'=	Y = 11 Y'= 11 X = 0 X'= 0	
• cache-2 writes back X' & Y'		X= 1 Y=11	X = 1 Y =11 X'= 0 Y'=11	Y =11 Y'=11 X = 0 X'= 0	

inconsistent

Write-through Caches & SC

Embedded Systems Laboratory

prog T1	cache-1	memory	cache-2	prog T2

ST X, 1
ST Y,11
	X= 0
	Y=10
		X = 0
		Y =10
		X'=
		Y'=
			Y =
			Y'=
			X = 0
			X'=
				LD Y, R1
				ST Y', R1
				LD X, R2
				ST X',R2

- T1 executed

cache-1	memory	cache-2
X= 1 Y=11	X = 1 Y =11 X'= Y'=	Y = Y'= X = 0 X'=

- T2 executed

cache-1	memory	cache-2
X= 1 Y=11	X = 1 Y =11 X'= 0 Y'=11	Y = 11 Y'= 11 X = 0 X'= 0

Nici cache-urile write-through nu mențin consistența secvențială

Menținerea consistenței secvențiale (CS)

Embedded Systems Laboratory

CS este suficientă pentru programe tip producer-consumer și cu excluziune mutuală (e.g., Dekker)

Copiile multiple ale unei locații în diferite Cache-uri pot cauza degradarea CS.

Este nevoie de suport hardware pentru

- doar un singur procesor la un moment dat are permisiune de write la o locație de memorie
- nici un procesor nu poate să încarce o copie a vechii locații după o scriere

$\Rightarrow$ Protocoale de coerență a cache-ului

Protocoale de menținere a coerenței cache pentru CS

write request:

adresa este *invalidată* (*actualizată)* în toate cache-urile înainte (după) o operație de write

read request:

dacă o este gasită o copie ”murdară” într-un cache, se efectuează un write-back înainte de citirea memoriei

Ne vom concentra pe protocoale de Invalidare și nu pe protocoale Update

Warmup: Parallel I/O

Fie Cache-ul fie DMA-ul pot să fie Bus Master și să facă transferuri la un mom. dat

(DMA vine de la Direct Memory Access)

Probleme cu Parallel I/O

Memorie → Disc: Memoria fizică poate fi neactualizată dacă copia din Cache este ”murdară”

Disc → Memorie: Cache-ul poate să conțină date vechi și să nu vadă write-urile la memorie

Cache-uri multiple și coerența cache-urilor

Memoriile cache dedicate fiecărui procesor reduc traficul la memoria procipală (prin rețeaua de interconectare) dar introduc o serie de probleme de consistență.

Starea copiilor de date

Diferite tipuri de blocuri de date din cache pentru un procesor paralel cu memorie principală centralizată și cache-uri locale pentru fiecare procesor

Snoopy Cache *Goodman 1983*

- Idee: Să avem un cache care spionează (snoop upon) transferurile DMA, și atunci “do the right thing”
- Etichetele snoopy cache sunt dual-port

Acțiunile Snoopy Cache pentru DMA

Embedded Systems Laboratory

Observed Bus Cycle	Cache State	Cache Action
DMA Read Memory → Disk	Address not cached	No action
	Cached, unmodified	No action
	Cached, modified	Cache intervenes
DMA Write Disk → Memory	Address not cached	No action
	Cached, unmodified	Cache purges its copy
	Cached, modified	???

Shared Memory Multiprocessor

Folosește mecanismul snoopy pentru a păstra consistența memoriei pentru toate procesoarele

Protocol snoopy de coerență cache

Automat FSM pentru un protocol de coerență cache ce folosește cache write-back

Diagrama de tranziții pentru Cache

Protocolul MSI

Embedded Systems Laboratory

Fiecare linie din cache are tag

M: Modified
S: Shared
I: Invalid

Starea cache pentru procesorul P_1

Exemplu cu două procesoare
(Citesc și scriu aceeași linie din cache)

Embedded Systems Laboratory

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes

Observație

- Dacă o linie este în starea M atunci nici un alt cache nu poate să aibă o copie a linei!
  - Memoria rămâne coerentă, nu pot exista copii diferite

MESI: Protocol MSI îmbunătățit
performanțe mărite pentru date locale

Fiecare linie are un tag

M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid

Starea cache pt. procesorul P1

Implementarea Algoritmului Snoopy Cache

Structura principală a unui algoritm snoopy de coerență cache

Snoop optimizat cu cache-uri Level-2

- Procesoarele au de obice cache pe două niveluri
  - mic L1, mare L2 (de obicei ambele on chip acum)
- *Proprietatea de incluziune:* intrările din L1 trebuie să fie în L2

  invalidare în L2 $\Rightarrow$ invalidare în L1
- Snooping în L2 nu afectează lățimea de bandă CPU-L1

Ce probleme pot să apară?

Intervenție

Când avem un read-miss pentru A în cache-2, se emite pe bus un read request pentru A

- Cache-1 trebuie să facă vizibilă și să își schimbe starea în "shared"
- Memoria poate să răspundă și ea la cerere!

Știe memoria că are date vechi?

Cache-1 trebuie să intervină prin controllerul de memorie pentru a da datele corecte pentru cache-2

False Sharing

state	blk addr	data0	data1	...	dataN

Un bloc cache conține mai mult de un cuvânt de date

Coerența cache-ului este făcută la nivel e bloc și nu la nivel de cuvânt

Presupunem că P_1 scrie $word_i$ și P_2 scrie $word_k$ și ambele cuvinte au aceeași adresă de bloc.

Ce se poate întâmpla?

Sincronizarea și cache-urile:
Probleme de performanță

Protocoalele de coerență a cache-ului vor face mutex-ul să facă *ping-pong* între cache-urile lui P1 și P2.
Acest fenomen poate fi redus prin citirea locației inițiale a mutex-ului (non-atomic) și execuția unui schimb doar dacă acesta are valoarea zero.

Performanța și traficul pe magistrale

În general, o instrucțiune *read-modify-write* necesită două operații pe magistrală, dacă nu avem alte operații la memorie de către alte procesoare

Într-un scenariu multiprocesor, accesul tuturor celorlalte procesoare la magistrală trebuie să fie blocat pe durata execuției operației atomice de read-modify-write

$\Rightarrow$ costisitor pentru magistrale simple

ISA-urile moderne folosesc

load-reserve

store-conditional

Load-reserve & Store-conditional

Embedded Systems Laboratory

Registre speciale pentru stocarea flag-urilor de reserve, adresa și rezultatul store-conditional

```
Load-reserve R, (a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional (a), R:
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

Dacă un alt procesor vede o tranzacție de store la adresa din registrul de rezervare, bitul de rezervare este setat la 0

- Mai multe procesoare pot rezerva 'a' simultan
- Aceste instrucțiuni sunt similare cu load și store obișnuite, din pct de vedere al traficului pe bus

Performanță:
Load-reserve & Store-conditional

Embedded Systems Laboratory

Numărul total de tranzacții pe bus nu este neapărat redus, dar spargerea unei instrucțiuni atomice în load-reserve & store-conditional:

- crește utilizarea magistralei, mai ales pentru magistralele care efectuează tranzacții în mai multe etape

- *reduce efectul ping-pong pentru cache* deoarece procesoarele care încearcă să obțină un semafor nu trebuie să facă un store de fiecare dată

Lățimea de bandă limitează performanțele

Embedded Systems Laboratory

Avem un sistem multiprocesor cu memorie partajată construit în jurul unui singur bus cu lățimea de bandă de *x* GB/s. Cuvintele de instrucțiuni și date au fiecare 4B lățime, fiecare instrucțiune necesită accesul în medie la 1.4 cuvinte din memorie (inclusiv la instrucțiunea însăși). Rata combinată de hit a cache-urilor este de 98%. Calculați limita sperioară a performanței sistemului multiprocesor în GIPS. Liniile de adresă sunt separate și nu afectează lățimea de bandă de date.

Soluție

Execuția unei instrucțiuni implică un transfer de $1.4 \times 0.02 \times 4 = 0.112$ B. Astfel, limita absolută a perfomanței este de $x/0.112 = 8.93x$ GIPS. Dacă presupunem o lățime a busului de 32 de biți, că nici un ciclu pe bus nu se irosește și o frecvență de ceas pe bus de *y* GHz, limita superioară a perfomanței devine $286y$ GIPS. Magistralele operează la frecvențe de 0.1 - 1 GHz. Prin urmare, o performanță apropiată de 1 TIPS (chiar și ¼ TIPS) este peste capabilitățile acestui tip de arhitectură.

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252

Calculatoare Numerice (2)

- **Cursul 12 –**

Multiprocesoare 3

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: Single instruction operates on single data element
- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Why Parallel Computers?

Embedded Systems Laboratory

- Parallelism: Doing multiple things at a time
- Things: instructions, operations, tasks

- Main (or Original) Goal
  - Improve performance (Execution time or task throughput)
    - Execution time of a program governed by Amdahl's Law

- Other Goals
  - Reduce power consumption
    - (4N units at freq F/4) consume less power than (N units at freq F)
    - Why?
  - Improve cost efficiency and scalability, reduce complexity
    - Harder to design a single unit that performs as well as N simpler units
  - Improve dependability: Redundant execution in space

Types of Parallelism and How to Exploit Them

- Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- Task Level Parallelism
  - Different "tasks/threads" can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

Task-Level Parallelism: Creating Tasks

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively
- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task

Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

Main Design Issues in Tightly-Coupled MP

Embedded Systems Laboratory

- Shared memory synchronization
  - How to handle locks, atomic operations
- Cache coherence
  - How to ensure correct operation in the presence of private caches
- Memory consistency: Ordering of memory operations
  - What should the programmer expect the hardware to provide?
- Shared resource management
- Communication: Interconnects

Main Programming Issues in Tightly-Coupled MP

Embedded Systems Laboratory

- Load imbalance
  - How to partition a single task into multiple tasks
- Synchronization
  - How to synchronize (efficiently) between tasks
  - How to communicate between tasks
  - Locks, barriers, pipeline stages, condition variables, semaphores, atomic operations, …
- Ensuring correct operation while optimizing for performance

Aside: Hardware-based Multithreading

Embedded Systems Laboratory

- Coarse grained
  - Quantum based
  - Event based (switch-on-event multithreading), e.g., switch on L3 miss
- Fine grained
  - Cycle by cycle
  - Thornton, "CDC 6600: Design of a Computer," 1970.
  - Burton Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
- Simultaneous
  - Can dispatch instructions from multiple threads at the same time
  - Good for improving execution unit utilization

Parallel Speedup Example

- $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions

- How fast is this with 3 processors?

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Single processor : 11 operations (DRAW the data flow graph)

$\tau_1 = 11$ cycles

edded
tems
oratory
Catedra
Facultatea d

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Three processors : τ_3 (exec. time with 3 proc.)

$\tau_3 = \underline{5 \text{ cycles}}$

Speedup with 3 Processors

$\tau_3 = \underline{5 \text{ cycles}}$

Speedup with 3 processors $= \dfrac{11}{5} = 2.2$

$\left(\dfrac{\tau_1}{\tau_3}\right)$

<u>Is this a fair comparison?</u>

Revisiting the Single-Processor Algorithm

Embedded Systems Laboratory

Revisit τ_1

Better single-processor algorithm:

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0$$

(Horner's method)

Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.

$\tau_1 = 8$ cycles

Speedup with 3 procs. $= \dfrac{\tau_1^{best}}{\tau_3^{best}} = \dfrac{8}{5} = \underline{\underline{1.6}}$

(not 2.2)

Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- Unfair comparisons

  Compare best parallel algorithm to wimpy serial algorithm → unfair

- Cache/memory effects

  More processors →
  more cache or memory →
  fewer misses in cache/mem

Utilization, Redundancy, Efficiency

Embedded Systems Laboratory

- Traditional metrics
  - Assume all P processors are tied up for parallel computation
- Utilization: How much processing capability is used
  - U = (# Operations in parallel version) / (processors x Time)
- Redundancy: how much extra work is done with parallel processing
  - R = (# of operations in parallel version) / (# operations in best single processor algorithm version)
- Efficiency
  - E = (Time with 1 processor) / (processors x Time with P processors)
  - E = U/R

Utilization of a Multiprocessor

Embedded Systems Laboratory

Multiprocessor metrics

Utilization: How much processing capability we use

$$U = \frac{\text{10 operations (in parallel version)}}{\text{3 processors} \times \text{5 time units}} = \frac{10}{15}$$

$$U = \frac{\text{Ops with p proc.}}{p \times \tau_p}$$

Redundancy: How much extra work due to multiprocessing

$$R = \frac{\text{Ops with } p \text{ proc.}^{best}}{\text{Ops with } 1 \text{ proc.}^{best}} = \frac{10}{8}$$

R is always ≥ 1

Efficiency: How much resource we use compared to how much resource we can get away with

$$E = \frac{1 \cdot T_1^{best}}{p \cdot T_p^{best}}$$

(tying up 1 proc for T_p time units)

(tying up p proc. for T_p time units)

$$= \frac{8}{15} \qquad \left(E = \frac{U}{R} \right)$$

Caveats of Parallelism (II)

- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
- Maximum speedup limited by serial portion: Serial bottleneck
- Parallel portion is usually not perfectly parallel
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among N processors)

Sequential Bottleneck

Why the Sequential Bottleneck?

- Parallel machines have the sequential bottleneck
- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

- There are other causes as well:
  - Single thread prepares data and spawns parallel tasks (usually sequential)

Another Example of Sequential Bottleneck

Embedded Systems Laboratory

LEGEND

A,E: Amdahl's serial part
B: Parallel Portion
C1,C2: Critical Sections
D: Outside critical section

```
InitPriorityQueue(PQ);
SpawnThreads();                                   A

ForEach Thread:

    while (problem not solved)
        Lock (X)
            SubProblem = PQ.remove();             C1
        Unlock(X);
        Solve(SubProblem);
        If(problem solved) break;                 D1        B
        NewSubProblems = Partition(SubProblem);
        Lock(X)
            PQ.insert(NewSubProblems)             C2
        Unlock(X)
                                                  D2
        . . .

PrintSolution();                                  E
```

Bottlenecks in Parallel Portion

Embedded Systems Laboratory

- Synchronization: Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - Communication: Tasks may need values from each other
  - Causes thread serialization when shared data is contended

- Load Imbalance: Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects
  - Reduces speedup in parallel portion

- Resource Contention: Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive
  - Additional latency not present when each task runs alone

Bottlenecks in Parallel Portion: Another View

Embedded Systems Laboratory

- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications
- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, …
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- Even within a thread, some "code segments" may be on the critical path of execution; some are not

Critical Sections

- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

Each thread:

```
loop {
  Compute                 N
  lock(A)
     Update shared data
  unlock(A)               C
}
```

Barriers

- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving to the barrier is on the critical path

```
Each thread:
   loop1 {
      Compute
   }
   barrier
   loop2 {
      Compute
   }
```

Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path

```
loop {
  Compute1   A
  Compute2   B
  Compute3   C
}
```

Difficulty in Parallel Programming

Embedded Systems Laboratory

- Little difficulty if parallelism is natural
  - "Embarrassingly parallel" applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Onur Mutlu (CMU)

- MIT material derived from course 6.823
- UCB material derived from course CS252
- CMU material derived from course ECE447