1. Uvod:

Uobičajena podela softvera

- Sistemski ↔ Aplikativni
- Šematski, sistemski softver je na nižem nivou

Podela sistema

- Sistem = ...?
- ... računarski sistem
- Podele po različitim kirterijumima :

– nisu apsolutne, uvek ima graničnih slučajeva
– približno: koji kvadrant koordinatnog sistema posmatramo

- Po složenosti/kapacitetu resursa : Integirsani sistemi (embedded); Računari opšte namene; Superračunari
- Po tipu korisničke interakcije : Sa posrednom interakcijom; Sa neposrednom interakcijom
- Po tipu veze s okruženjem : Izolovan; Umrežen
- Po broju korisnika : Jednokorisnički ; Višekorisnički
- Po broju istovremenih korisnika
- Po broju istovremenih korisničkih procesa: Jednoprocesni; Višeprocesni

Naše koordinate

- Računar opšte namene...
- ... sa neposrednom interakcijom
- ... umrežen
- ... višekorisnički
- ... višeprocesni

Razgraničavanje softvera

- Sistemski softver pruža usluge softveru na višem nivou
- Korisnik najčešće nema neposrednu interakciju sa sistemskim softverom
- Osnovna motivacija: modularizacija ; apstrakcija

Primeri sistemskog softvera

- Operativni sistemi
- Sistemske biblioteke
- Softverske platforme
- Udaljeni servisi

Operativni sistemi

- Na najnižem nivou: softverski skup po imenu kernel
- Neposredna interakcija s hardverom
- Apstrahovanje hardvera i računarskih resursa: Drajveri ; Procesi ; Fajlsistem...
- Upravljanje resursima
- Privilegije i prava pristupa

Sistemske biblioteke

- Apstrahovanje veze sa kernelom
- Standardizovani interfejs za softver višeg nivoa
- Teorijski, omogućava razvoj prenosivog softvera
- U praksi, izvodljivo ali s dosta napora

Softverske platforme

- Okruženja za izvršavanje specijalizovanog softvera
- ... numerička izračunavanja
- ... simulacije
- ... igre
- Okruženja za korisnički softver

Udaljeni servisi

- „X as a service“
- ... gde X mogu biti razne stvari
- Nama je zanimljivo „software“
- Takođe se može smatrati nekom vrstom sistemskog softvera
- Spada u proučavanje distribuiranih sistema

Sistemsko programiranje

- Ima specifičnosti u odnosu na aplikativni razvoj
- Zahteva manje ili veće poznavanje hardverskih detalja
- Posvećuje se veća pažnja optimalnom korišćenju resursa
- Ne mogu se očekivati pogodnosti kao što je automatsko upravljanje memorijom
- Mogućnosti za testiranje su često ograničene

Sistemski programski jezici

- Moraju biti prilagođeni navedenim zahtevima
- Interpretirani/dinamički prevođeni jezici sa automatskim upravljanjem memorijom uglavnom nisu pogodni (osim možda za SaaS)
- Prvi izbor: asembler
- Viši nivo: PL/I (IBM), C
- Kasnije: C++
- Još kasnije: Rust

Okruženje za sistemsko programiranje

- Razvojni alati: Prevodilac (kompajler) ; Linker ; Debager
- Alati za analizu objektnog koda
- Alati za analizu rada s memorijom
- Alati za profilisanje programa
- Sistem za praćenje verzija izvornog koda

Naše koordinate

- Hardverski detalji, uopšteno sa nekim specifičnim ilustracijama
- Uticaj hardvera na ponašanje programa
- Softverske tehnike u sistemskim programima i bibliotekama
- Razvojno okruženje
- Analiza ponašanja programa

2. Savremeni mikroprocesori

Mikroprocesori kao kategorija računarskih sistema

- Današnji računari su isključivo zasnovani na mikroprocesorima
- Pre 50 godina to nije bilo tako ~

● Procesor je sklop velikog broja elektronskih komponenti; dominantno tranzistora
● Mikroprocesor je moguć samo ako se te komponente mogu integrisati u jedinstveni sklop: integrisano kolo

Početak razvoja mikroprocesora
● Početkom 1970-ih, kada postaje moguće integrisati 10 ~ 3 komponenti na integrisano kolo
● Taj stepen integracije je tada nazvan **LSI** (=Large Scale Integration), kasnije povećano na **VLSI** (10 ~ 6, Very Large Scale…), kasnije je predloženo **ULSI** (Ultra…) ali se od te nomenklature odustalo
● Prvi mikroprocesori su po svim parametrima performansi zaostajali za tadašnjim diskretnim računarima „opšte namene“ (ako dopustimo kategoriju)

Tempo razvoja mikroprocesora
● U veoma dugom razdoblju razvoj se ponašao kao eksponencijalni proces
● Eksponencijalni procesi se odupiru intuicijskom razumevanju (primeri: hiperinflacija, epidemije)
● Ovakav razvoj je imao opipljive posledice na oblast sistemskog programiranja
● Razvoj je dobro opisan sa dva „zakona“ (pre empirijska zapažanja)

Denardovo skaliranje
● Robert Denard (Dennard): elektroinženjer, istraživačka karijera u IBM-u
● 1974.: snaga integrisanog kola po jedinici površine ostaje konstantna sa porastom gustine komponenti
● Uprošćeno: promena fizičkih dimenzija tranzistora menja njegove električne karakteristike tako da očuva gustinu snage, a dopušta rast radne frekvencije

Murov zakon
● Gordon Mur (Moore): inženjer, saosnivač Intela, osnivač Fairchild Semiconductors
● 1965.: broj komponenti integrisanog kola se udvostručuje svake godine; kasnije revidirano na 18 meseci ili dve godine (zbog ove promenljivosti i nije „zakon“ u fizičkom smislu)
● Murovo zapažanje je poznatije od Denardovog
● Rade zajedno—posledica je da performanse/W rastu brže od porasta broja komponenti

Kraj eksponencijalnog rasta
● Svaka eksponencijalna progresija ima granicu
● Denardovo skaliranje je prestalo da važi oko 2004., ne zato što su tranzistori prestali da se smanjuju, već zbog toga što napon i jačina struje tranzistora imaju granične vrednosti ako se želi očuvati pouzdanost rada
● Murov zakon je prestao da važi oko 2012.– 2014.; broj komponenti i dalje raste, ali daleko od toga da se udvostručuje na dve godine

Procesori

8086	29.000	3 µm
80386	275.000	1,5 µm
80486	1.200.000	1 µm
Pentium	3.100.000	0,8 µm
Pentium 4	42.000.000	0,18 µm
Core 2 Duo	291.000.000	65 nm
i7 Skylake	1.750.000.000	22 nm

Napomena o dimenzijama
● Dužina data kao oznaka proizvodnog procesa integrisanog kola danas nema direktne veze ni sa jednom fizičkom dimenzijom tranzistora
● Fizička veza se mogla naći otprilike do 32 nm: razstojanje između istovetnih elemenata u matrici memorijskih ćelija izvedenih u ovoj tehnologiji
● Danas ima više smisla posmatrati gustinu komponenti, recimo milioni tranzistora/mm2

Uticaj na sistemsko programiranje— početak

● 1975.: MOS Technologies 6502
● 8-bitni mikroprocesor
● 3500 tranzistora; 3 korisnička registra opšte namene; radni takt 1 Mhz
● Programiranje uglavnom u asembleru; mikroprocesor je nepogodan za više programske jezike zbog siromašnog skupa registara
● Osnovna memorija se koristi kao proširenje skupa registara

Uticaj na sistemsko programiranje— eksponencijalni rast

● Dok je tehnologija u periodu eksponencijalnog rasta, osnovne performanse prirodno rastu s protokom vremena (=treba samo sačekati)
● Osnovno merilo performansi kod mikroprocesora opšte namene je broj celobrojnih operacija u jedinici vremena
● Nema potrebe drastično menjati ili dopunjavati arhitekturu ako ne postoji spoljašnji pritisak koji to diktira

Primeri spoljašnjih uticaja na arhitekturu

● Primer 1: operacije u pokretnom zarezu (razlomljene vrednosti), koje se masovno koriste u numeričkim modelima i naučnim izračunavanjima
● Rešenje: matematički koprocesori (dodatni procesori sa instrukcijama za rad u pokretnom zarezu), kasnije sastavni deo procesora
● Primer 2: rad sa većim količinama radne memorije (RAM=Random Access Memory), izvedene u dinamičkoj tehnologiji

Dinamička memorija

● Tip memorije koji se dominantno koristi za „blisku“ radnu memoriju računarskog sistema
● Rast kapaciteta saobrazan porastu broja tranzistora u integrisanom kolu
● 1989: 1 MB
● 2014: 1 GB
● Porast brzine rada ni izdaleka tako dramatičan kao kod mikroprocesora, zbog čega su mikroprocesori morali interno da se prilagođavaju

Kategorije paralelizma

● Kad linearne pojedinačne performanse nisu dovoljne, može se razmatrati paralelizam
● Paralelizam podataka: jer postoje podskupovi podataka nad kojima se operacije mogu obavljati istovremeno
● Paralelizam zadataka: jer se celokupni zadaci često mogu podeliti na relativno nezavisne podzadatke
● Ove kategorije se mogu iskoristiti organizaciono ili tehnički—nas zanima tehničko iskorišćenje u okviru mikroprocesora

Iskorišćenje paralelizma na nivou mikroprocesora

● Paralelizam na instrukcijskom nivou: korišćenje paralelizma podataka za istovremeno izvršavanje skupova instrukcija
● Vektorske arhitekture, grafički koprocesori, multimedijalne instrukcije: korišćenje paralelizma podataka tako što se jedna instrukcija primenjuje na veću količinu podataka
● Paralelizam na nivou lanaca izvršavanja (thread-level): paralelizam podataka ili zadataka u tesno spregnutoj interakciji pojedinačnih lanaca izvršavanja

Flinova taksonomija

● Metod klasifikacije procesorskih sistema i/ili skupova instrukcija po stepenu paralelizma (Flynn, 1966)
● SISD (Single Instruction, Single Data), klasične celobrojne ili pokretno-zarezne instrukcijama
● SIMD (Single Instruction, Multiple Data), vektorske ili multimedijalne instrukcije
● MISD, ne postoji u praksi
● MIMD, kombinacija SIMD za više izvršnih jedinica, podrazumeva paralelizam zadataka

Mikroprocesorske arhitekture

● Danas, dve komercijalno dominantne
● Intel x86 – dominatno 64-bitni, serverski – neuspešan pokušaj serije mobilnih mikroprocesora

● ARM – 32-bitni i 64-bitni, mobilni uređaji, sa pokušajima serverskih i desktop/laptop varijanti
● Napuštene: MIPS, SPARC, Alpha, HPPA, Itanium
● Još postoji IBM Power

3. Savremeni mikroprocesori

Model rada (klasična fon Nojmanova arhitektura)
● Memorija je linearna i sekvencijalno adresirana (veličina najmanje adresibilne jedinice nije bitna, danas je to 8-bitni bajt/oktet)
● Instrukcije se izvršavaju strogo sekvencijalno
– izvršavanje naredne instrukcije počinje tek kad je prethodna završena
– naredna instrukcija vidi sve efekte prethodne
● Integrisana kola su sinhrona
– ovo nema neposredne veze sa načinom izvršavanja instrukcija, ali ima presudan uticaj na arhitekturu

Instrukcijski ciklus
● Koraci potrebni za izvršavanje jedne instrukcije, apstraktno gledano; realni procesori imaju mašinski ciklus
● (1) Prihvatanje instrukcije
– iz memorijske lokacije na koju pokazuje **programski brojač,** registar koji prati izvršavanje instrukcija
– po završetku ovog koraka programski brojač će pokazivati na sledeću instrukciju u nizu
● (2) Dekodiranje instrukcije
● (3) Čitanje efektivne adrese (može se posmatrati i kao potkorak drugog koraka)
– ako instrukcija koristi indirektno adresiranje, efektivna adresa se čita iz memorije
– ako je instrukcija memorjska i direktna, u ovom koraku se ništa ne dešava
– isto važi i za instrukcije koje ne operišu s memorijom
● (4) Izvršavanje
– ukoliko je u pitanju izvršena instrukcija grananja, postaviće novu vrednost u programski brojač

Petostepeni transport
● „5-stage pipeline“, jedan od načina da se instrukcijski ciklus mapira na hardver
● Istorijski, prvobitno upotrebljen (za mikroprocesore) u RISC arhitekturama 1980-ih

IF	Instruction fetch	Prihvatanje instrukcije
ID	Instruction decode	Dekodiranje instruckije
EX	Execute	Izvršavanje
MEM	Memory access	Pristup memoriji
WB	Register writeback	Zapis registara

Primarni zahtevi „pipelined“ ahitekture
● Instrukcije su uniformne dužine, najčešće 32 bita
● Koriste se proste instrukcije za memorijski pristup
– „Load/store architecture“, bez komplikovanih režima adresiranja
● Sve ostale instrukcije rade s registrima
● Široko korišćeno u RISC arhitekturama, kanonički primer: MIPS, moderna iteracija: RISC-V

Problemi u „pipelined“ arhitekturi
● Dve osnovne kategorije problema
● Prva, veće zauzeće memorije za instrukcije nego što je neophodno
– ovo znači i indirektan pritisak na interne procesorske memorijske strukture, što ćemo obraditi kasnije
● Druga, narušavanje sekvencijalne semantike izvršavanja:
– prilikom rada s podacima
– prilikom grananja

Zauzeće memorije za instrukcije

● Uobičajen izbor je bio veličina mašinske reči, 32 bita
● Ovo je dovoljno da se nedvosmisleno kodiraju troadresne instrukcije sa skupom od 32 registra
● Značajan podskup programa ne zahteva ovako veliki registarski prostor
● Uvode se kompaktne, 16-bitne instrukcije koje rade na smanjenom skupu registara i ograničenom skupu instrukcija
● Kanonički primer: ARM Thumb

Varijante kompaktne reprezentacije

● Prva varijanta (Thumb u prvobitnoj realizaciji) zahteva da se procesor prebaci u poseban režim rada da bi izvršavao ovakve instrukcije
● Ovo je problematično jer se često dešava da je u nizu kompaktnih instrukcija potrebno izvršiti nekoliko instrukcija standardne veličine, pa treba preskakati između režima rada
● Modernija izvedba (RISC-V, ARM Thumb2) omogućava da se standardne i kompaktne instrukcije slobodno mešaju
● Rezultat (RISC-V): gustina koda vrlo slična onoj koju ima Intel x86_64 arhitektura

Semantika pristupa podacima

● Uopšteno: problem nastaje ako paralelno izvršavanje delova instrukcija ili čitavih instrukcija izazove efekat nemoguć u sekvencijalnom kodu
● Uvek je moguće sprečiti semantičke probleme ako se u izvršavanje unese čekanje, čija je krajnja granica ekvivalent sekvencijalnog izvršavanja
● Ovo obara performanse, pa se, ako je moguće, traže rešenja koja će raditi bez čekanja

Klasifikacija opasnosti

● Nepoželjni efekti se zovu opasnosti (hazards) i označavaju se skraćenicama
● Oznake mogu da deluju zbunjujuće, jer ukazuju na poželjan efekat
● Svaka od opasnosti uključuje zapisivanje vrednosti u nekoj od kombinacija; samo čitanje nikad ne može narušiti semantiku
● Kad budemo opisivali opasnosti, navešćemo prvo poželjan, pa nepoželjan efekat, koji treba sprečiti
● RAW (Read After Write):
– želimo da čitanje posle pisanja vrati upravo zapisanu vrednost
– opasnost: naredna instrukcija ne vidi rezultat izvršavanje prethodne
● WAR (Write After Read):
– želimo da čitanje pre pisanja vrati originalnu vrednost – opasnost: prethodna instrukcija vidi rezultat izvršavanja naredne
– nemoguće u klasičnoj petostepenoj arhitekturi, može da se desi ako se instrukcije izvršavaju preko reda
● WAW (Write After Write):
– želimo da konačan rezultat ove sekvence bude druga zapisana vrednost
– opasnost: ukupan efekat je rezultat izvršavanja prethodne instrukcije, a ne naredne
– takođe nemoguće u standardnoj petostepenoj arhitekturi
● Ilustrovaćemo samo RAW opasnost jer je najčešća, relativno lako shvatljiva, i često ima tehničko rešenje

RAW problem (rešiv bez čekanja)	ADDU R2,R1,R3	IF	ID	EX	MEM	WB		
	SUBU R4,R2,R5		IF	ID	EX	MEM	WB	

● Instrukcija ADDU izračunava rezultat u stepenu EX, ali ga zapisuje u R2 tek u stepenu WB
● Instrukcija SUBU čita registar R2 u stepenu ID, tako da vidi raniju vrednost
● Rešenje: hardverska indikacija promene tako da rezultat postaje vidljiv već u stepenu ID

RAW problem (nerešiv bez čekanja)	LW R1,0(R2)	IF	ID	EX	MEM	WB		
	SUBU R4,R1,R5		IF	ID	EX	MEM	WB	

● Instrukcija LW (Load Word) ima vrednost na raspolaganju tek u stepenu MEM, kada se instrukcija SUBU već izvršava

● Mora se čekati bar jedan mašinski takt da bi učitana vrednost mogla da se pročita u stepenu ID od strane instrukcije SUBU

Grananje

● Dva osnovna problema s grananjem
● U principu se ne može pouzdano znati koja će se instrukcija semantički izvršiti posle instrukcije grananja
● Kod petostepene arhitekture, u trenutku dekodiranja instrukcije grananja naredna instrukcija (u memoriji) već je pročitana, tako da je trošak njenog čitanja i dekodiranja straćen ukoliko se grananje izvrši
● Za prvi problem, bez dodatnih komplikacija nema rešenja sem čekanja
● Za drugi, neki RISC procesori čine vidljivom (i izvršavaju) instrukciju iza grananja bez obzira da li je do grananja došlo: „branch delay slot“
● Ovo je zbunjujuće za programere i nezgodno za kompajlere, tako da su kasniji procesori ovo izbegavali

Dodatno unapređenje paralelizma

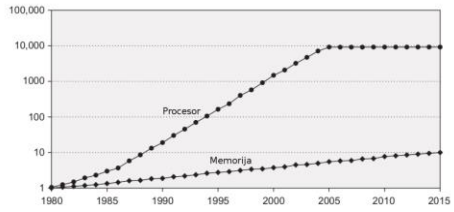
● Prekoredno izvršavanje (Out of order execution) – izvršava instrukcije u zavisnosti od dostupnih podataka
● Superskalarni procesori – izvršavaju više od jedne instrukcije istovremeno
– koriste postojanje više izvršnih jedinica u okviru procesora
– (ovo još uvek nije višeprocesorski sistem)
● Spekulativno izvršavanje (Speculative execution) – izvršavanje unapred da bi se amortizovali troškovi grananja
● Predviđanje grananja (Branch prediction), sklop se zove prediktor
– prosti statički (uslovno grananje se neće izvršiti)
– statički (→ se neće izvršiti, ← će se izvršiti)
– primena spekulativnog izvršavanja

4. Memorijska hijerarhija i radna memorija

Memorijska hijerarhija

● Što je izvor podataka bliži procesoru, ima veće performanse, višu cenu po jedinici kapaciteta i manji kapacitet
● Hijerarhija postoji od početka razvoja računara, a nivoi su bili srazmerno stabilni tokom decenija
● Standardni nivoi:
– Procesorski registri
– Keš memorija
– Radna memorija
– [PCM, Flash] masovna memorija
– Magnetni diskovi

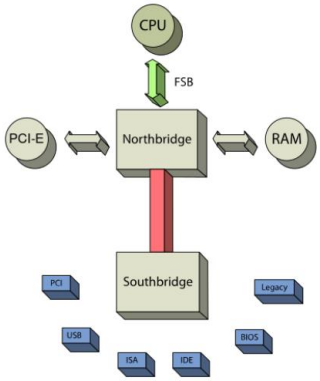
Osnovna disproporcija u performansama



Tipičan hardver računara opšte namene

● Jedan do dva fizička procesora (socket) – retko se ide na više fizičkih procesora zbog komplikovanja konstrukcije
● Prostor za 16 GB – 256 GB dinamičke memorije
– velike razlike između serverskih i desktop varijanti
– za veće servere su dostupni i memorijski kapaciteti reda TB
● Integrisani periferni uređaji
● Relativno slična osnovna arhitektura

Šema hardverske organizacije (do ~2011.)



Osnovni elementi organizacije

● Northbridge: direktno komunicira s procesorom
– FSB = Front Side Bus
– zadužen za upravljanje memorijom
– dodatno, za brze periferne uređaje
● Southbridge: ostali periferni uređaji
– sve sporije magistrale
– memorija gde brzina pristupa nije kritična (BIOS)

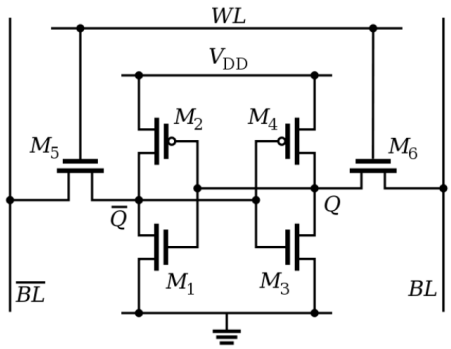
Posledice organizacije

● Komunikacija između fizičkih procesora koristi FSB
● Sva komunikacija s memorijom ide preko NB – memorija ima jednu pristupnu tačku (port)
– višestruki pristup kod specijalizovanih uređaja
● Komunikacija sa perifernim uređajima zakačenim na SB ide preko NB
● Brzi periferni uređaji direktno pristupaju memoriji – DMA (Direct Memory Access)
– Ovime se rasterećuje procesor...
● ... Ali stvara konkurenciju s procesorom za pristup memoriji preko NB
● Veza NB s memorijom i njeno iskorišćenje su kritični za performanse

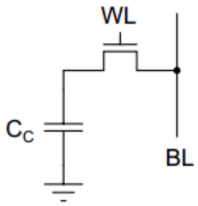
Kako povećati propusnu moć

● NB ne mora sam da upravlja memorijom, već za to mogu postojati posebna kola
● Više memorijskih kontrolera na NB – veća količina podržane memorije
– bolja propusna moć
● Ograničenje postaje interna propusna moć NB
● Integrisani memorijski kontroler na fizičkom procesoru – današnja varijanta
● U ovoj organizaciji ne postoji poseban NB, SB se zove „Platform Controller Hub“ (kod Intela)
● Više procesora znači i više kontrolera, sa prednostima kao kod prethodne varijante
● Problem: kod više fizičkih procesora, pristup memoriji više nije uniforman
– ekstremna varijanta: NUMA (Non-Uniform Memory Access)

Statički RAM (6 tranzistora)



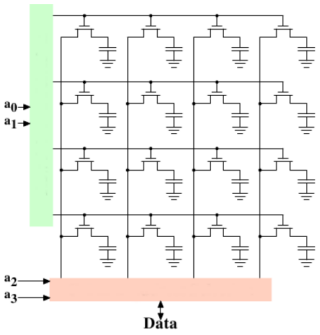
Dinamički RAM (1 tranzist



Opšte napomene

- Dinamički RAM je očigledno jednostavniji od statičkog
- Problemi: – čitanje sadržaja je posle izvesnog broja ciklusa destruktivno
  – kapacitet kondenzatora ne može biti veliki
  – mora se periodično osvežavati
- U toku osvežavanja pristup je nemoguć
- Direktno čitanje je nemoguće
- Moraju postojati pauze tokom čitanja/pisanja

Organizacija dinamičkog RAM-a

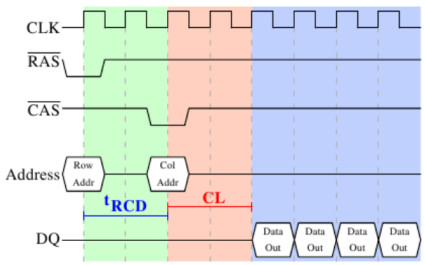


- Ćelije su organizovane u matricu, koja treba da ima jednak broj vrsta i kolona ako je moguće

– raskorak komplikuje hardver za (de)multipleksiranje na većoj strani

- Vrsta se bira signalom RAS (Row Address Selection) – linija iznad znači da je signal invertovan
- Kolona se bira signalom CAS (Column Address Selection)
- RAS se demultipleksira na osnovu dela adrese (tj. signal se usmerava na jednu od vrsta u zavisnosti od kombinacije **a0/a1**)
- Aktiviranje vrste će kao rezultat imati iščitavanje svih ćelija u okviru te vrste, čiji se signali sprovode do multipleksera kolona
- CAS aktivira multiplekser na osnovu ostatka adrese, kombinacije **a2/a3**, i na izlazu se pojavljuje očitavanje adresirane ćelije
- Sama adresa je često multipleksirana

Dijagram pristupa RAM-u



Posledice načina pristupa

● Pristup je mnogo sporiji u odnosu na kapacitet procesora – vremenski, 10–15:1
● Pristup proizvoljnoj lokaciji zahteva čekanje
● Prenos susednih lokacija je efikasan
● Moguće je raditi čitanje unapred da bi se smanjilo čekanje

Drugi korisnici memorije

● Grafički podsistem – integrisana grafika bez odvojene memorije
– ako se zahtevaju bolje performanse ove komponente su izdvojene
● Disk kontroleri
● Mrežni kontroleri – uporedivi sa (magnetnim) diskovima po sirovoj brzini

5. Keširanje

Rekapitulacija za memoriju/procesor

● Inverzna korelacija složenosti memorije i njenih performansi
● Praktičan sistem mora imati veliku količinu memorije da bi mogao da obrađuje realne skupove podataka
– dakle, radna memorija mora biti sporija u odnosu na procesor
● Skupovi podataka su često veći i od praktično upotrebljive količine radne memorije
– sekundarna memorija je nekoliko redova veličine sporija od glavne
– ovo se donekle menja postojanjem bržih sekundarnih memorija (flash)

Moguće arhitekture međumemorije

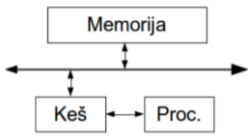
● Uvodi se mala količina brze međumemorije
● Jedna od mogućnosti je da se memorija tretira kao proširenu skup registara pod kontrolom korisničkih procesa
– procesi moraju znati hardverske detalje pojedinačnih procesora
– svaki proces mora imati disjunktni podskup u odnosu na druge
– trošak organizovanja pristupa bi poništio prednosti
● Druga mogućnost, koja se univerzalno koristi, jeste međumemorija (uglavnom) transparentna za korisnika

Keš memorija: principi

● Brza memorija koja sadrži privremene kopije podataka za koje se može pretpostaviti da će biti potrebni za izvršavanje koda
● Ovo je izvodljivo jer programi ispoljavaju lokalitet
● Prostorni lokalitet: mali kontinualni podskup podataka se koristi u kontinuitetu
– na primeru koda: petlja
– izvesne konfiguracije podataka su takođe prostorno bliske
● Vremenski lokalitet: verovatnoća da će neki podskup podataka biti opet potreban ne dugo posle obrade
– za kod: funkcijski poziv u petlji (kod funkcije je prostorno udaljen, ali potreban kod svake iteracije)
– za podatke: ukupan radni skup relativno ograničen, ali ne mora biti prostorno kontinualan

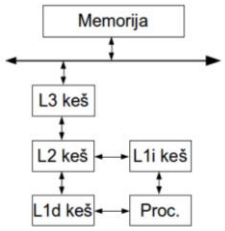
Konfiguracija keša

● Prvobitni sistemi:



● Sav pristup memoriji ide kroz keš
● Veza između keša i procesora je specijalno projektovana za brzinu
● Ovo funkcioniše, ali moguća su poboljšanja:
– razdvajanje memorije za instrukcije i podatke: modifikacija klasične fon Nojmanove arhitekture
– organizacija u nivoima, zbog neekonomičnosti povećanja jedinstvenog memorijskog prostora

Moderna konfiguracija keša



Osnovni parametri keša

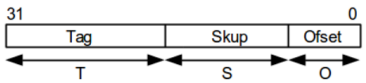
Vrsta pristupa	Br. ciklusa	Vreme ns
L1 pogodak	4	1,2
L2 pogodak	10	3,0
L3 pogodak	40	12,0
lokalni RAM	100	60
udaljeni RAM	160	100

● Pretpostavka: 100 elemenata, 100 iteracija
● Svaki pristup kroz lokalni RAM: 100·100·100 = 1.000.000 ciklusa
● Svaki pristup kroz L2 keš: 100·100·10 = 100.000 ciklusa
● Ušteda: 90%
● U realnim situacijama iskorišćenja iznad 90% su uobičajena

Osnovna organizacija keša

● Tipično, keš je hiljadama puta manji od količine RAM-a
● Potrebno je naći efikasan način za popunjavanje i adresiranje sadržaja
● Svaki zapis u kešu mora biti dostupan na osnovu adrese podatka koji se traži
– deo zapisa koji predstavlja adresu zove se tag
● Granularnost zapisa na nivou mašinske reči bila bi krajnje neefikasna
– zato se postavlja na 32 ili 64 bajta, što predstavlja jednu liniju
● Prilikom izmene memorije, cela linija prvo mora da se učita u keš
– u principu, procesor nema operacije koje rade nad celom linijom
● Izmena neke lokacije u liniji označava celu liniju kao zaprljanu
– zaprljane linije su one koje su izmenjene, a nisu zapisane nazad u memoriju
● Prilikom čitanja, najčešće se neka od postojećih linija mora izbaciti iz keša
– ako su svi kandidadi zaprljani, moraju se zapisati, što unosi kašnjenje
– izbor kandidata je generalno težak problem
● Za primer: 4 MB, 64 bajta, 65536 linija (22 bita)

Adresiranje linija



● Segment O je fiksan i zavisi od dužine linije: O = log2L
● Segmenti S i T zavise od načina organizacije adresiranja
– potpuno asocijativno
– direktno mapirano
– skupovno asocijativno

Potpuno asocijativni keš

● S = 0, T = 32 - O
● Svaka linija može sadržati kopiju bilo koje memorijske lokacije – obratite pažnju: i dalje postoji mogućnost promašaja
● Za svaku od linija mora postojati komparator koji će proveriti jednakost tagova

● Hardverski preterano zahtevno
● Komparatori ne mogu da rade iterativno jer se inače gubi smisao keširanja

Direktno mapirani keš
● Hardverski problem se rešava ograničavanjem opsega pretrage
● Na jednoj krajnosti, svaki tag se mapira na tačno jednu liniju
● S = log2N, gde je N broj linija – u našem slučaju N = 65536, S = 16
● Može dobro da radi ako su adrese ravnomerno raspoređene u prostoru određenom za mapiranje
– najčešće nisu
– praktična posledica: neke linije se stalno zamenjuju, neke ostaju prazne

Skupovno asocijativni keš
● Kombinacija prethodna dva
● S više ne bira pojedinačne linije, već skupove linija
– slično direktno mapiranom, ali je promenjena granularnost adresiranja
● Unutar svakog skupa, linije koje mu pripadaju porede se s tagom u paraleli
– slično potpuno asocijativnom, samo za mali broj elemenata
● Na našem primeru, ako je keš 8-struko asocijativan, imaće 8192 skupa – 8192·8 = 65536
– S = log28192 = 13

Primer—direktno mapiran keš
● Polazna memorijska adresa: 0x9cb2008
● Binarno, sa grupisanim ciframa: 1001 1100 1011 0010 0000 0000 1000
● Ista ta vrednost, sa razdvojenim tagom i selektorom linije unutar keša: 1001 1100 1011 0010 0000 0000 1000
tag: 0x27, selektor: 0x2c80, ofset: 0x8
● Adresa: 0xa0b2008, isti postupak: 1010 0000 1011 0010 0000 0000 1000
tag: 0x28, selektor: 0x2c80, ofset: 0x8
● Ako program naizmenično pristupa ovim adresama, linija će svaki put morati da se obnavlja iz RAM-a

Primer—8-struko asocijativni keš
● Polazna memorijska adresa: 0x9cb2008, isti postupak kao i ranije, ali ovog puta će selektor biti 13 bita umesto 16:
1001 1100 1011 0010 0000 0000 1000
tag: 0x139, selektor: 0xc80, ofset: 0x8
● Polazna adresa 0xa0b2008: 1010 0000 1011 0010 0000 0000 1000
tag: 0x141, selektor: 0xc80, ofset: 0x8
● Obe adrese se mapiraju na isti skup, ali imaju različite tagove, pa pošto skup ima kapacitet od osam linija, veće su šanse da za obe linije ima mesta u kešu

Keširanje instrukcija
● Jednostavnije od keširanja podataka
● Programski prevodioci generišu kod koji bolje vodi računa o iskorišćenju keša
– autori prevodilaca generalno bolje poznaju ponašanje procesora
● Tok izvršavanja programa je predvidljiviji od šema pristupa podacima
● Programi po pravilu imaju dobar prostorni i vremenski lokalitet
● Eventualni problem: samomodifikujući kod

6. Virtuelna memorija

Jednoprocesni sistem
● Ograda: sistem koji sinhrono reaguje na spoljašnje događaje nikad ne može da bude striktno jednoprocesni
● Proces vidi fizičku memoriju, kako je hardver mapira u adresni prostor procesora – ± memorijski mapirani uređaji
● Sve memorijske adrese su fizičke adrese

● Ovo stvara poteškoće: – ako je potrebno pokretati pozadinske procese
– ako memorijski raspored nije fiksan

Primer: MS-DOS

Problemi kod višeprocesnog rada

● Nestabilnost adresnog prostora
– apsolutna odredišta grananja ne mogu da se koriste u fiksnom obliku
– reference na statički alocirane podatke takođe
– rešava se dinamičkom relokacijom
● Problem stabilnosti sistema
– proces može da pristupi osetljivim delovima memorije (npr. mapiranja za uređaje, prekidni vektori)
– ovo može da izazove nestabilnost ili krah
● Problem bezbednosti/privatnosti
– svaki proces može da vidi podatke svih drugih
– podnošljivo kod jednokorisničkog sistema, nedopustivo kod višekorisničkog

Upravljanje memorijom

● Za rešavanje navedenih problema uvodi se koncept upravljanja memorijom
● Poseban podsistem na procesoru koji se zove jedinica za upravljanje memorijom (Memory Management Unit, MMU)
● Jedan od zadataka je da se memorija organizuje tako da svaki proces ima:
– sliku raspoložive memorije kao da je jedini koji se izvršava na sistemu
– memorijski prostor zaštićen od drugih procesa

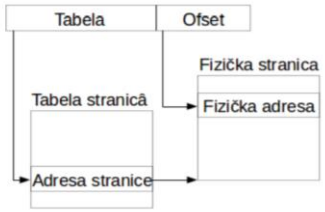
Virtuelna memorija

● Uvođenje MMU omogućava virtuelizaciju memorije
● Zaštita procesa je samo jedan (ali veliki) zadatak virtuelizacije
– drugi je omogućavanje rada sa adresnim prostorom većim od raspoložive fizičke memorije
– korišćenje sekundarne memorije (disk) kao podrške za memorijski sadržaj
– mehanizam sličan keširanju
● Memorijske reference koje program koristi više se ne odnose direktno na fizičku memoriju
– sve adrese su virtuelne adrese
– mora postojati način za prevođenje iz virtuelnih u fizičke adrese

Principi prevođenja

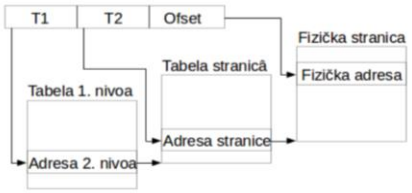
● Memorija se deli na stranice (pages, 4 KB – 4 MB)
● Mora postojati memorijska struktura koja pokazuje na pojedinačne stranice
● Ova struktura se zove tabela stranica (Page Directory)
● Mogući su različiti pristupi prevođenju, u zavisnosti od veličine i broja stranica koje treba pokriti

Prosto prevođenje



● Veoma nalik na keširanje – pokazivač na tabelu stranica = tag
– ofset u okviru stranice = ofset u okviru linije
● Realan primer: stranice od 4 MB – 22 bita za ofset, 10 bita za indeks stranice
– 1024 elementa u tabeli
● U opštem slučaju, velike stranice su neefikasne
– veliki broj operacija zahteva poravnanje na veličinu stranice, što povećava utrošak memorije
● Za manje stranice, jedan nivo je takođe neefikasan
– tabela stranica je velika
– svakom procesu je potrebna posebna tabela
– posledica: velika potrošnja memorije na tabele za upravljanje memorijom(!)

Više nivoa prevođenja



● Podelom adrese stranice na više delova omogućava se izbegavanja zauzimanja memorije za nealocirane delove u memorijskoj mapi
● Kod realne organizacije procesa čest je slučaj da su zauzeti disjunktni blokovi memorije na suprotnim krajevima adresnog prostora
● Ova organizacija omogućava i korišćenje većeg fizičkog adresnog prostora u odnosu na adresni kapacitet procesora
● Tabele su hijerarhijske
● Proces konstruisanja prevođenja je nužno iterativan, ne može se paralelizovati
● Ako je MMU aktivan, prevođenje je potrebno prilikom svakog memorijskog pristupa
– za dva nivoa, dva pristupa memoriji
– tabele mogu imati do četiri nivoa
– ako su tabele u L1d, bar četiri dodatna ciklusa po pristupu
● Potreban je način da se ovaj postupak ubrza

TLB keš

● Rešenje koje se primenjuje jeste da se kešira ukupan rezultat prevođenja: fizička adresa stranice
● Keš u kome se drže ovi podaci zove se TLB (=Translation Lookaside Buffer)
● Ovaj keš mora biti veoma brz da bi se izbeglo čekanje prilikom svakog memorijskog pristupa
● Obično realizovan kao potpuno asocijativni, sa malim brojem elemenata