

# Pamäťový model

Memory is the thing you  
forget with.

Alexander Chase

Pri časovej analýze algoritmov sa zvyčajne používa takzvaný *RAM model* (skratka z anglického *Random-Access Machine*, stroj s náhodným prístupom k pamäti), v ktorom sa predpokladá možnosť prístupovať k ľubovoľnému úseku pamäte v konštantnom čase. To znamená, že vo výslednej asymptotickej analýze počítame len počet vykonaných operácií.

ref

V skutočnosti však moderné počítače využívajú niekoľko úrovňovú pamäťovú hierarchiu [7]. Tá sa typicky skladá z registrov a troch úrovní *cache* (vyrovnávacej pamäte) priamo na procesore, následne z hlavnej operačnej pamäte a disku. V tomto poradí sú tieto úrovne zoradené od najrýchlejšej a najmensej (odozva rádovo 1 ns, kapacita 64 KiB) až po najpomalšiu ale najväčšiu (odozva od 100 μs po 10 ms podľa typu<sup>1</sup>, kapacita rádovo 1 TiB). To znamená, že rozdiel v prístupovej dobe je 10 000 000-násobný a len ťažko môžeme hovoriť o konštantnom čase. Približné hodnoty pre všetky úrovne sú v tabuľke ??.

ram blok - dobre?

registre? tazko zratat velkost

Dôsledkom tejto hierarchie je závislosť výslednej rýchlosti algoritmu od jeho prístupov k pamäti. Operácie, ktoré využívajú dáta uložené na disku potrvávajú dlhšie ako tie, ktoré využívajú iba dáta v registroch. Dôvodom je, že pri prístupe k dátam na disku sa v skutočnosti sa tieto dáta postupne presunú z disku do hlavnej pamäte, do *cache* na procesore a napokon do registrov. Pre všetky susedné dvojice pamäťových úrovní teda slúži tá menšia a rýchlejšia ako vyrovnávacia pamäť pre tú väčšiu a pomalšiu. Následne

<sup>1</sup>Klasické pevné disky (HDD) alebo disky bez pohyblivých častí (SSD)

Tabuľka 1.1: Približné parametre rôznych úrovní pamäte. Hodnoty troch úrovní *cache* na procesore (L1, L2 a L3) uvádzame pre mikroarchitektúru Intel Haswell. [9, 10]

Úroveň	Veľkosť	Odozva	Asociativita	Veľkosť bloku
L1	64 KiB <sup>1</sup>	4clk <sup>2</sup>	$\approx 1$ ns	8
L2	256 KiB <sup>1</sup>	11clk <sup>2</sup>	$\approx 4$ ns	8
L3	2–20 MiB	36clk <sup>2</sup>	$\approx 12$ ns	
RAM	$\approx 8$ GiB	$\approx 100$ ns		16 B
Disk	$\approx 1$ TiB	$\approx 0.1$ – $10$ ms		4 KiB–2 MiB

<sup>1</sup> Hodnota pre jedno jadro procesora.

<sup>2</sup> Počet cyklov procesora, uvedené časové aproximácie pri 3 GHz.

sa môže požadovaná operácia vykonať rovnako rýchlo ako v druhom prípade (keď už boli dáta v registroch), avšak nejaký čas bol algoritmus nečinný a čakal na presun dát.

## 1.1 External-memory model

Spôsobom, ako zohľadniť tieto skutočnosti pri analýze algoritmov je takzvaný *external-memory model* (model externej pamäte), nazývaný tiež *I/O model* alebo *cache-aware model* [1]. Ten popisuje pamäť skladajúcu sa z dvoch častí (obrázok ??), ktoré voláme *cache* a *disk*.

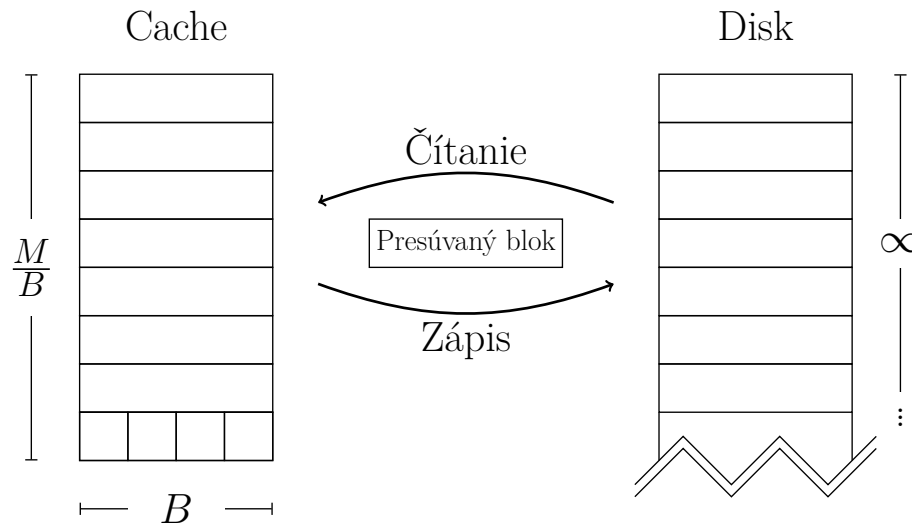
Všetky výpočty prebiehajú nad dátami v *cache*, ktorá má obmedzenú veľkosť. Ostatné dáta sú uložené na disku neobmedzenej veľkosti, no nemôžeme s nimi priamo manipulovať a je ich potrebné najskôr preniesť do *cache*. V samotnej analýze algoritmov potom počítame počet týchto prenosov z disku do *cache* a naopak.

Prístup k dátam, ktoré sa práve nachádzajú v *cache* voláme *cache hit* (zásah *cache*). Naopak prístup, ktorý vyžaduje dáta najskôr presunúť z *disk*-u do *cache* voláme *cache miss* (minutie *cache*). Tieto presuny sú realizované v *blokoch* pamäte veľkosti  $B$ . Disk aj *cache* sa skladajú z takýchto blokov a za jednu operáciu považujeme presun jedného bloku medzi nimi. *Cache* má obmedzenú veľkosť  $M$  a skladá sa teda z  $\frac{M}{B}$  blokov.

### 1.1.1 Cache-aware algoritmy

Pokiaľ poznáme parametre  $B$  a  $M$ , môžeme skonštruovať algoritmus, ktorý bude túto dvojicu pamätí využívať efektívne. Takýto algoritmus voláme *cache-aware* (uvedomujúci si *cache*). Súčasťou tohto algoritmu by bolo explicitné spravovanie presunov pamäte - je potrebné riešiť čítanie blokov z disku a ich umiestňovanie do *cache*, nahrádzanie blokov v *cache* pri zaplnení a spätný zápis blokov na disk.

Tento model popisuje len dve úrovne pamäte a teda funguje efektívne len pre danú susednú dvojicu, pre ktorú ho na základe znalosti parametrov optimalizujeme. V moderných systémoch ale máme takýchto dvojíc niekoľko. Keby sme poznali parametre



Obrázok 1.1: External-memory model

pre všetky tieto dvojice, môžeme zovšeobecniť tento model pre viac úrovní a explicitne riešiť presun blokov medzi nimi. Takto sa však samotná správa pamäte potenciálne stáva komplikovanejším problémom ako pôvodný algoritmus.

## 1.2 Cache-oblivious model

Riešením týchto problémov je *cache-oblivious model* (na *cache* nedbajúci), v ktorom uvažujeme rovnakú dvoj-úrovňovú pamäť zloženú z disku a *cache* [8, 15]. Na rozdiel od *cache-aware* modelu však algoritmus nepozná parametre  $B$  a  $M$ . Pokiaľ sa nám napriek tomu podarí navrhnúť algoritmus, ktorý vykonáva (asymptoticky) rovnaký počet pamäťových presunov ako *cache-aware* algoritmus, bude bežať efektívne pre ľubovoľné takéto parametre.

Takéto algoritmy majú na rozdiel od *cache-aware* algoritmov v *external-memory* modely mnohé výhody. Samotná implementácia algoritmu nemôže explicitne riešiť presun blokov pokiaľ nepozná veľkosť bloku ani koľko blokov môže do *cache* uložiť. Táto úloha zostane ponechaná na nižšiu vrstvu (operačný systém respektíve hardvér v prípade *cache* na procesore) - algoritmus bude pristupovať k pamäti priamo bez ohľadu na to, či sa nachádza v *cache* alebo nie a v prípade potreby prebehnú nutné prenosy na nižšej úrovni (z pohľadu algoritmu) automaticky.

Ďalšou výhodou je automatická optimalizácia pre dané parametre. V prípade *cache-aware* algoritmov môže byť problémom získať presné hodnoty týchto parametrov a potrebné pri ich zmene upraviť algoritmus. Vývoj algoritmu, ktorý bude fungovať na rozličných architektúrach, môže byť značne problematický.

V neposlednom rade bude takýto *cache-oblivious* algoritmus efektívny medzi každou dvojicou susedných úrovní. Vzhľadom na to, že hodnoty parametrov nepozná, musí pre ľubovoľnú takú dvojicu pracovať rovnako efektívne ako pre každú inú.

### 1.2.1 Správa pamäte

V momente, keď sa *cache-oblivious* algoritmus pokúsi o vykonanie operácie, ktorá potrebuje dáta mimo *cache*, je potrebné ich najskôr z disku skopírovať. V prípade, že je v *cache* voľný blok, je možné presunúť dáta bez nutnosti nahradenia. V opačnom prípade je však potrebné uvoľniť miesto tým, že sa vyberie blok z *cache* (ak bol tento blok upravený, najskôr sa jeho obsah zapíše späť na disk), ktorý bude následne prepísaný požadovaným blokom. Tento proces sa nazýva výmena stránok (*page replacement*), a algoritmus rozhodujúci, ktorý blok z *cache* odstrániť, voláme stratégiou výmeny stránok (*page-replacement strategy*). Dve základné stratégie výmeny stránok sú *LRU* a *FIFO*.

Stratégia *LRU* (least recently used – najdlhšie nepoužitý) vyberá vždy blok, ktorý bol najdlhšie nepoužitý. Implementácia vyžaduje udržiavať si ku každému bloku počítadlo, ktoré sa pri prístupe nastaví na nulu a pri prístupe k iným blokom zvýši o jedna. Pri potrebe uvoľniť miesto v *cache* vyberieme blok s najväčšou hodnotou počítadla – ten, ku ktorému najdlhšie nebol prístup.

Stratégia *FIFO* (first in, first out – prvé dnu, prvé von) je ešte jednoduchšia – bloky udržiavame zoradené podľa poradia, v akom sme ich vložili do *cache*. Keď vyberáme blok na odstránenie, vezmeme ten, ktorý bol pridaný najskôr.

Ak by sa táto stratégia správala tak, že vždy odstráni blok, ktorý bude potrebný v ďalšom kroku algoritmu, tak by bolo zakaždým presúvať bloky medzi diskom a *cache*. To by znamenalo, že počet blokov, s ktorými v *cache* môžeme prakticky pracovať je  $\frac{M}{B} = 1$ . Ďalším problémom je *asociatívnosť* *cache* – z praktických dôvodov je často možné daný blok z disku uložiť len na niekoľko pozícií v *cache*. Inak by bolo potrebné ukladať spolu s každým blokom jeho plnú adresu na disk, čo by redukovalo celkový počet blokov, ktoré sa do *cache* zmestia. Znížením asociativity je možné ukladať iba časť adresy, pričom zvyšok je implicitne určený pozíciou v *cache*. V prípade nízkej asociativity však môžu opäť nastať situácie, kedy je algoritmus schopný využiť iba malý počet blokov v *cache*.

Tieto problémy *cache-oblivious* model obchádza predpokladom ideálnej *cache* – *cache*, ktorá je plne asociatívna (každý blok disku je možné uložiť v každom bloku *cache*) a používa optimálnu stratégiu výmeny stránok, ktorá vždy odstráni blok, ktorý bude potrebný najneskôr. Prvý predpoklad je síce v reálnych systémoch nepraktický, no z teoretického hľadiska je v poriadku. Druhý predpoklad je však nerealizovateľný, keďže by stratégia výmeny stránok musela predpovedať budúce kroky algoritmu. Nasledovné lemy však ukazujú, že bez týchto predpokladov na reálnom systéme s nízkou asociativitou a jednoduchou stratégiou výmeny stránok sa algoritmus zhorší len o konštantný faktor.

**Lema 1.2.1.** *Algoritmus, ktorý v ideálnej cache veľkosti  $M$  s blokmi veľkosti  $B$  vykoná  $T$  pamäťových operácií, vykoná najviac  $2T$  pamäťových operácií v cache veľkosti  $2M$  s blokmi veľkosti  $B$  pri použití stratégie LRU alebo FIFO. [8, Lemma 12]*

**Lema 1.2.2.** *Plne asociatívna cache veľkosti  $M$  sa dá simulovať s použitím  $\mathcal{O}(M)$  pamäte tak, že prístup ku každému bloku v cache zaberie v priemernom prípade  $\mathcal{O}(1)$  času. [8, Lemma 16]*

# Literatúra

- [1] AGGARWAL, Alok ; VITTER, Jeffrey u. a.: The input/output complexity of sorting and related problems. In: *Communications of the ACM* 31 (1988), Nr. 9, S. 1116–1127
- [2] BAYER, Rudolf: Binary B-trees for Virtual Memory. In: *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. New York, NY, USA : ACM, 1971 (SIGFIDET '71), 219–235
- [3] BENDER, Michael A. ; DEMAINE, Erik D. ; FARACH-COLTON, Martin: Cache-Oblivious B-Trees. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*. Redondo Beach, California, November 12–14 2000, S. 399–409
- [4] BENDER, Michael A. ; DEMAINE, Erik D. ; FARACH-COLTON, Martin: Cache-Oblivious B-Trees. In: *SIAM Journal on Computing* 35 (2005), Nr. 2, S. 341–358
- [5] BENDER, Michael A. ; DUAN, Ziyang ; IACONO, John ; WU, Jing: A locality-preserving cache-oblivious dynamic dictionary. In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms* Society for Industrial and Applied Mathematics, 2002, S. 29–38
- [6] DEMAINE, Erik D.: Cache-Oblivious Algorithms and Data Structures. In: *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, June 27–July 1 2002
- [7] DREPPER, Ulrich: What every programmer should know about memory. In: *Red Hat, Inc* 11 (2007)
- [8] FRIGO, Matteo ; LEISERSON, Charles E. ; PROKOP, Harald ; RAMACHANDRAN, Sridhar: Cache-oblivious algorithms. In: *Foundations of Computer Science, 1999. 40th Annual Symposium on* IEEE, 1999, S. 285–297
- [9] INTEL CORPORATION: *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2014 ( 248966-029)

- [10] INTEL CORPORATION: *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2014 ( 325462-050US)
- [11] KASHEFF, Zardosht: *Cache-oblivious dynamic search trees*, Massachusetts Institute of Technology, Diss., 2004
- [12] KOTRLOVÁ, Katarína: *Vizualizácia háld a intervalových stromov*, Univerzita Komenského v Bratislave, bakalárska práca, 2012
- [13] KOVÁČ, Jakub: *Vyhľadávacie stromy a ich vizualizácia*, Univerzita Komenského v Bratislave, bakalárska práca, 2007
- [14] LUKČA, Pavol: *Perzistentné dátové štruktúry a ich vizualizácia*, Univerzita Komenského v Bratislave, bakalárska práca, 2013
- [15] PROKOP, Harald: *Cache-oblivious algorithms*, Massachusetts Institute of Technology, Diss., 1999
- [16] TOMKOVIČ, Viktor: *Vizualizácia stromových dátových štruktúr*, Univerzita Komenského v Bratislave, bakalárska práca, 2012