

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) [2], v ktorom sa predpokladá možnosť pristupovať k ľubovoľnému úseku pamäte v konštantnom čase. To znamená, že vo výslednej asymptotickej analýze počítame len počet vykonaných operácií.

V skutočnosti však moderné počítače využívajú niekoľko úrovňovú pamäťovú hierarchiu [11]. 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 μ s po 10 ms podľa typu¹, kapacita rádovo 1 TiB). Približné hodnoty pre všetky úrovne sú v tabuľke 1.1. Ako vidieť, rozdiel v prístupovej dobe k rôznym dátam môže byť až 10 000 000-násobný. Naskytuje sa teda otázka, či je užitočné a presné hovoriť o konštantnom čase.

Dôsledkom tejto hierarchie je závislosť výslednej rýchlosti algoritmu od jeho prístupov k pamäti. Operácie, ktoré využívajú dáta uložené na disku potrvajú dlhšie ako tie, ktoré využívajú iba dáta v registroch. Pri prístupe k dátam na disku sa v skutočnosti tieto dáta postupne presunú z disku do hlavnej pamäte, do *cache* na procesore a nakoniec do registrov. Toto zabezpečí, že ich opakované použitie, pokiaľ nebudú dovtedy z *cache* odstránené, bude rýchlejšie. Pre všetky susedné dvojice pamäťových úrovní teda slúži tá menšia a rýchlejšia ako vyrovnávacia pamäť pre tú väčšiu a pomalšiu.

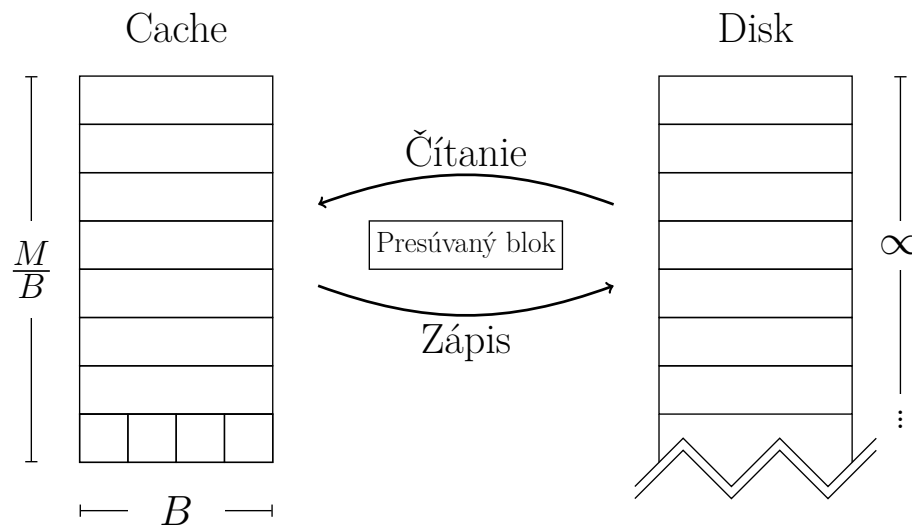
¹Klasické pevné disky (HDD) alebo disky bez pohyblivých častí (SSD)

Tabuľka 1.1: Približné parametre rôznych úrovní pamäte. Hodnoty troch úrovní *cache* na procesore (L1, L2 a L3) uvádzame pre mikroarchitektúru Intel Haswell. [13, 14]

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

¹ Hodnota pre jedno jadro procesora.

² Počet cyklov procesora, uvedené časové aproximácie pri 3 GHz.



Obrázok 1.1: External-memory model

1.1 External-memory model

Jedným zo spôsobov, 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 1.1), 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 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. Toto nemusí byť explicitnou súčasťou algoritmu a môže byť riešené na inej úrovni.

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 pre všetky tieto dvojice, môžeme tieto algoritmy optimalizovať pre všetky susedné dvojice a ich parametre. Stále však zostáva problémom viazanosť algoritmu na tieto parametre a pri ich zmene prestáva byť optimálny.

1.2 Cache-oblivious model

Druhým spôsobom, ktorý zohľadňuje nekonštantný prístupový čas k údajom v pamäti, je takzvaný *cache-oblivious model* (na *cache* nedbajúci), v ktorom uvažujeme rovnakú dvoj-úrovňovú pamäť zloženú z disku a *cache* [12, 20]. 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.

Výhodou oproti *cache-aware* algoritmom 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.

Ďalšou výhodou je, že takýto *cache-oblivious* algoritmus bude (rovnako) efektívny medzi každou dvojicou susedných úrovní. Vzhľadom na to, že hodnoty parametrov nepozná, bude 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 stratégiá 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 možné efektívne pracovať len s jedným blokom v *cache*, napriek tomu, že by sa ich do *cache* zmestilo viac. Vzhľadom na to, že analýza množstva *cache-oblivious* algoritmov predpokladá istý minimálny počet blokov, ktorý sa zmestí do *cache*, bol by tento predpoklad nenaplnený. To by mohlo spôsobiť, že by algoritmus vykonal viac pamäťových presunov ako táto analýza predpovedala.

Ďalším problémom je takzvaná *asociatívnosť* cache – počet pozícií v *cache*, na ktoré môžeme daný blok uložiť, ktorý je z praktických dôvodov často obmedzený. Inak by bolo potrebné ukladať spolu s každým blokom jeho plnú adresu na disku, 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*. V moderných systémoch (tabuľka 1.1) sa asociativita pohybuje okolo 8, čo znamená, že daný blok je v *cache* možné umiestniť len na 8 z $\frac{M}{B}$ pozícií.

Ideálna *cache*

Tieto problémy *cache-oblivious* model obchádza predpokladom ideálnej *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 nerealizovateľný, keďže by stratégia výmeny stránok musela predpovedať budúce kroky algoritmu. Nasledovné lemy však ukazujú, že aj 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. [12, 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. [12, Lemma 16]*

1.3 Prehľad výsledkov v cache-oblivious modeli

Pre širší rozhľad v tejto problematike uvádzame v tabuľke 1.2 na strane 6 zoznam niekoľkých bežných problémov. Ku každému uvádzame ako príklad najlepšie známe *cache-aware* a *cache-oblivious* algoritmy a dátové štruktúry, ktoré ten problém riešia, spolu s asymptotickou analýzou počtu pamäťových presunov.

V tejto práci sa budeme venovať prevažne problému vyhľadávacích stromov, konkrétne B-stromom. V nasledujúcej kapitole uvedieme ich fungovanie a analýzu počtu pamäťových presunov v *external-memory* modeli. V tretej kapitole popíšeme vizualizácie, ktoré sme pre tieto štruktúry vytvorili.

Tabuľka 1.2: Prehľad výsledkov *cache-aware* a *cache-oblivious* algoritmov a dátových štruktúr pre rôzne problémy. Odhady udávajú počet pamäťových operácií, pričom M a B sú parametre *cache*. Skratka *amort.* označuje amortizovanú zložitosť.

Vyhľadávacie stromy	Vyhľadávanie	Vkladanie	Prechod ¹
<i>cache-aware</i> B-strom	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\frac{K}{B})$ [5, 24]
<i>cache-oblivious</i> B-strom	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N + \frac{\log^2 N}{B})$ <i>amort.</i>	$\mathcal{O}(\frac{K}{B})$ [6, 7, 8], časť ??
<i>cache-oblivious</i> B-strom	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N)$ <i>amort.</i>	[6, 7, 8], časť ??
Usporiadaná postupnosť	Vkladanie	Prechod ¹	
<i>cache-aware</i> spájaný zoznam	$\mathcal{O}(1)$ <i>amort.</i>	$\mathcal{O}(\frac{K}{B})$	[10, 19]
<i>cache-oblivious</i> usporiadané pole	$\mathcal{O}(\frac{\log^2 N}{B})$	$\mathcal{O}(\frac{K}{B})$	[6, 9]
Triedenie			
<i>cache-aware</i> mergesort	$\mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$		[10, 24]
<i>cache-oblivious</i> funnelsort	$\mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$		[4, 10]
Prioritné fronty	Vkladanie	Výber minima	
<i>cache-aware</i> buffer tree	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ <i>amort.</i>	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ <i>amort.</i>	[3]
<i>cache-oblivious</i> prioritná fronta	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ <i>amort.</i>	$\mathcal{O}(\frac{1}{B} \log_{M/B} \frac{N}{B})$ <i>amort.</i>	[4, 10]

¹ Prechod K po sebe idúcich prvkov

Literatúra

- [1] Alok Aggarwal, Jeffrey Vitter, et al.: *The input/output complexity of sorting and related problems*, in: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.
- [2] Alfred V Aho, John E Hopcroft: *Design & Analysis of Computer Algorithms*, Pearson Education India, 1974.
- [3] Lars Arge: *The buffer tree: A new technique for optimal I/O-algorithms*, Springer, 1995.
- [4] Lars Arge, G Brodal, Rolf Fagerberg: *Cache-oblivious data structures*, in: *Handbook of Data Structures and Applications* 27 (2005).
- [5] Rudolf Bayer: *Binary B-trees for Virtual Memory*, in: *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '71, San Diego, California: ACM, 1971, pp. 219–235.
- [6] Michael A. Bender, Erik D. Demaine, Martin Farach-Colton: *Cache-Oblivious B-Trees*, in: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, Redondo Beach, California, 2000, pp. 399–409.
- [7] Michael A. Bender, Erik D. Demaine, Martin Farach-Colton: *Cache-Oblivious B-Trees*, in: *SIAM Journal on Computing* 35.2 (2005), pp. 341–358.
- [8] Michael A Bender et al.: *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, pp. 29–38.
- [9] Michael A Bender et al.: *Two simplified algorithms for maintaining order in a list*, in: *Algorithms—ESA 2002*, Springer, 2002, pp. 152–164.
- [10] Erik D. Demaine: *Cache-Oblivious Algorithms and Data Structures*, in: *Lecture Notes from the EEF Summer School on Massive Data Sets*, BRICS, University of Aarhus, Denmark, 2002.
- [11] Ulrich Drepper: *What every programmer should know about memory*, in: *Red Hat, Inc* 11 (2007).
- [12] Matteo Frigo et al.: *Cache-oblivious algorithms*, in: *Foundations of Computer Science, 1999. 40th Annual Symposium on*, IEEE, 1999, pp. 285–297.

- [13] Intel Corporation: *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 248966-029, 2014.
- [14] Intel Corporation: *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 325462-050US, 2014.
- [15] Zardosht Kasheff: *Cache-oblivious dynamic search trees*, PhD thesis, Massachusetts Institute of Technology, 2004.
- [16] Katarína Kotrlová: *Vizualizácia háld a intervalových stromov*, bakalárska práca, Univerzita Komenského v Bratislave, 2012.
- [17] Jakub Kováč: *Vyhľadávacie stromy a ich vizualizácia*, bakalárska práca, Univerzita Komenského v Bratislave, 2007.
- [18] Pavol Lukča: *Perzistentné dátové štruktúry a ich vizualizácia*, bakalárska práca, Univerzita Komenského v Bratislave, 2013.
- [19] Rasmus Pagh: *Basic external memory data structures*, in: *Algorithms for Memory Hierarchies*, Springer, 2003, pp. 14–35.
- [20] Harald Prokop: *Cache-oblivious algorithms*, PhD thesis, Massachusetts Institute of Technology, 1999.
- [21] Ladislav Pápay: *Príloha k bakalárskej práci - spustiteľná verzia*, 2014, DOI: 10.5281/zenodo.10082, URL: <http://dx.doi.org/10.5281/zenodo.10082>.
- [22] Ladislav Pápay et al.: *Príloha k bakalárskej práci - zdrojový kód*, 2014, DOI: 10.5281/zenodo.10080, URL: <http://dx.doi.org/10.5281/zenodo.10080>.
- [23] Viktor Tomkovič: *Vizualizácia stromových dátových štruktúr*, bakalárska práca, Univerzita Komenského v Bratislave, 2012.
- [24] Jeffrey Scott Vitter: *Algorithms and data structures for external memory*, in: *Foundations and Trends in Theoretical Computer Science* 2.4 (2008), pp. 305–474.

Kolofón

Sadzba tejto práce bola vykonaná systémom \LaTeX s použitím šablóny `memoir`. Literatúra bola spravovaná systémom `BibTeX`. Ilustrácie sú vyrobené pomocou balíčkov `TikZ/PGF`.



Text tejto práce je zverejnený pod Creative Commons licenciou verzie BY-NC-ND 3.0, ktorej plné znenie sa nachádza na stránke <https://creativecommons.org/licenses/by-nc-nd/3.0/>.

Zdrojový text tejto práce spolu so všetkými ilustráciami je dostupný na stránke <https://github.com/lacop/thesis>.