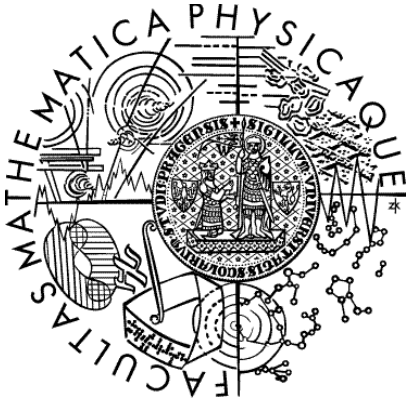
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Karol Hrdina

Vliv cache na efektivitu třídění

Katedra softwarového inženýrství

Vedoucí diplomové práce: *RNDr. Alena Koubková, Csc.*

Studijní program: *Informatika, Softwarové systémy*

Na tomto mieste by som rád poďakoval RNDr. Aleně Koubkové, Csc. za jej cenný čas a rady.

Prohlašuji, že jsem svou diplomovou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 8.8.2008 *Karol Hrdina*

Obsah

Název práce: Vliv cache na efektivitu třídění
Autor: Karol Hrdina
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Alena Koubková, Csc.
e-mail vedoucího: koubkova@ksi.ms.mff.cuni.cz

Abstrakt:

Klasické algoritmy pre triedenie vo vnútornej pamäti boli navrhnuté za predpokladu, že táto pamäť je homogénna. V moderných počítačoch je ale štruktúra pamäte hierarchická s rozdielnou rýchlosťou jednotlivých vrstiev. Doba výpočtu algoritmu teda závisí nielen na počte vykonaných operácií (napr. porovnanie prvkov), ale aj na počte presunov dát mezi jednotlivými vrstvami. Interné algoritmy tak získávajú niektoré rysy algoritmov externých.

V tejto práci si kladieme za úlohu stručne zhrnúť existujúce prístupy k problematike a opísať známe vylepšenia niektorých algoritmov pre prácu v nehomogénnej pamäti. Hlavný dôraz je kladený na implementáciu vybraných algoritmov a ich experimentálne overenie.

Kľúčové slová: cache, multimergesort, multiquicksort, triedenie.

Title: Effects of caches on performance of sorting
Author: Karol Hrdina
Department: Department of Software Engineering
Supervisor: RNDr. Alena Koubková, Csc.
Supervisor's e-mail address: koubkova@ksi.ms.mff.cuni.cz
Abstract:

Classical algorithms for sorting in internal memory were designed with an assumption, that the memory is homogenous. But modern computers have hierarchically structured memory with various speeds of it's layers. Execution time of algortihm is dependant not only on operation count, but also on count of transfers between memory layers. Therefore internal algorithms are having some characteristics of external algorithms.

In this paper we set our goal to summarize some existing approaches to this problem and summarize known optimalizations of internal sorting algorithms. Our main goal however is to impelent chosen algorithms and measure their performance experimentally.

Keywords: multiquicksort, funnelsort, sorting, cache

Kapitola 1

Úvod

Témou tejto diplomovej práce je skúmanie vplyvu cache pamätí na vnútorné triedenie. Našou hlavnou úlohou je v prvom rade prakticky overiť niektoré teoretické výsledky v tejto oblasti a ďalej zhrnúť existujúce prístupy k rišeniu problematiky. Na presnosť implementácie a merania triediacich algoritmov je kladený veľký dôraz.

V kapitole 2 sú rozobrané aspekty moderných pamäťových systémov. V kapitole 3 uvádzame prehľad výpočtových modelov. V Kapitole 4 sú uvedené dosiahnuté teoretické výsledky vo forme algortimov. Kapitola 5 zhŕňa výsledky našich testov.

Kapitola 2

Pamäťové systémy a latencia

Cache-oblivious algoritmy sú analyzované v Ideal-cache modeli a tento model je naozaj abstrakciou reálnych pamäťových hierarchií. Pochopenie moderných pamäťových systémov je preto nevyhnutné k pochopeniu cache-oblivious prístupu.

V kapitole 2.1 sú popísané aspekty moderných pamäťových systémov ktoré sú dôležité s ohľadom na cache-oblivious algoritmy. V sekcii 2.2 je popísaný problém pamäťovej latencie.

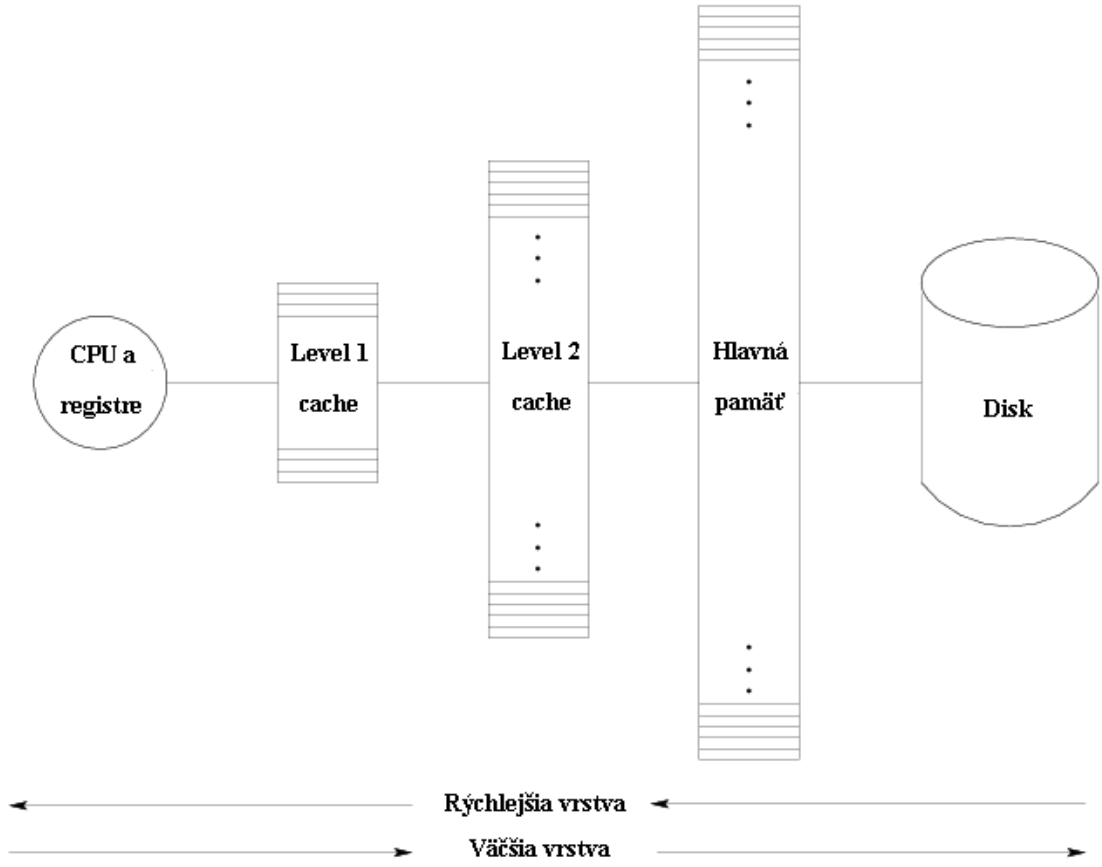
2.1 Dôležité asopekty pamäťovej hierarchie

V tejto sekcii sú popísané aspekty moderných pamäťových systémov, ktoré sú dôležité v kontexte cache-oblivious algoritmov. Dôraz sa preto nekladie na popis detailov určitého systému od nejakého výrobcu, ale skôr na popis myšlienok niektorých obecných a rozšírených konceptov/postupov.

Pamäťový systém obsahuje dáta s ktorými pracuje program (vstupné, výstupné dáta) a samotný kód programu. V kontexte cache-oblivious algoritmov nás nezaujíma ako pamäťový systém ukladá programový kód. Z toho dôvodu bol popis konceptov venujúci sa tejto téme vynechaný.

Moderné pamäťové systémy sú zložené z niekoľkých vrstiev pamätí zoradených do hierarchickej štruktúry. Typická pamäťová hierarchia je zobrazená na obr. 2.1 . Malá ale veľmi rýchla pamäťová vrstva sa nachádza najbližšie k procesoru, jedna alebo viacero väčších, ale pomalších pamäťových vrstiev sa nachádza ďalej od procesoru. Ak sa budeme dívať na registre ako integrovanú časť procesoru, typická pamäťová hierarchia pozostáva z jednej alebo viacerých vrstiev pamäte medzi procesorom, hlavnou pamäťou a diskom – ako najväčšou pamäťovou vrstvou. Pojem cache sa používa pri popise vrstiev medzi procesorom a hlavnou pamäťou, takže hierarchia zobrazená na obr. 2.1 má dve vrstvy cache a disk na celkový počet štyroch pamäťových vrstiev. Pojem cache sa tiež používa pri popise úlohy menšej z dvoch po sebe nasledujúcich pamäťových vrstiev, pr. hlavná pamäť slúži ako

cache disku a pod. Ak nebude uvedené inak, v tejto práci pod pojmom cache rozumieme prvý z uvádzaných vysvetlení slova cache.



Obr. 2.1: Pamäťová hierarchia pozostáva z viacerých pamäťových vrstiev. Čím ďalej sa nachádza vrstva od procesoru, tým je jej veľkosť väčšia a jej rýchlosť pomalšia.

Je rozdiel medzi dočasnou časťou pamäťovej hierarchie, kde dáta nepretrvajú po vypnutí prúdu (pr. vrstvy bližšie k procesoru), a stálou časťou pamäťovej hierarchie, kde dáta pretrvávajú nezmenené aj po vypnutí počítaču (pr. disk). Dočasná časť pamäťovej hierarchie reprezentuje primárnu pamäť, a disk reprezentuje sekundárnu pamäť. Niekedy sa primárna a sekundárna pamäť nazýva vnútorná a vonkajšia pamäť.

V ideálnom prípade pamäťvá hierarchia dodržuje vlastnosť nazývanú „*inclusion property*“. (Algoritmus dodržuje „inclusion property“, ak pre daný vstup, pamäťový stav algoritmu v každom časovom bode pre veľkosť $m+1$ zahrňuje stav m). Pre pamäťovú hierarchiu to znamená že pamäťová vrstva bližšie k procesoru obsahuje správnu podmnožinu dát ktorejkoľvek ďalšej vrstvy. Ak si označíme vrstvy v rastúcom poradí, vrstva i sa správa ako buffer pre vrstvu $i+1$. Vrstva najďalej od procesoru obsahuje všetky dáta. V tabuľke 2.2. sa nachádza prehľad vlastností cache L1 a L2 procesorov rodiny Intel.

Počítač	Popis
Intel Pentium	8 alebo 16 kB 2 alebo 4 cestná L1 cache, 32 bytová cache line L2 cache je externá, závisí na matičnej doske
Intel Pentium 2	16 kB 4-cestná L1 cache, 32 bytové cache line 256 alebo 512 kB 4-cestná L2 cache , 32 bytová cache line
Intel Pentium 3	16 kB 4-cestná L1 cache, 32 bytová cache line 256 alebo 512 4- alebo 8-cestná L2 cache, 32 bytová cache line
Intel Pentium 4	8 kB 4-cestná L1 cache, 64 bytová cache line 128, 256, alebo 512 kB 2-,4-, alebo 8-cestná L2 cache, 64 bytová cache line

Tab. 2.2: Prehľad vlastností cache L1 a L2 procesorov rodiny Intel. Zdroj: http://sandpile.org

2.1.1 Princíp lokality referencií

Pamäťová hierarchia je založená na dvoch pozorovaniach ohľadne správania sa počítačových programov. Tieto pozorovania všeobecne nazývajú princípy lokality referencií:

temporal locality – je pozorovanie, že prístup na rovnaké pamäťové miesto je väčšinou vykonaný veľa krát v krátkom časovom úseku. Príkladom je počítadlo vo for cykle, ktoré je čítané a na ktoré je zapisované aspoň raz počas každej iterácie cyklu. Inak povedané toto pozorovanie hovorí, že ak bolo na dané pamäťové miesto pristúpené, pavdepodobnosť že bude na toto isté pamäťové miesto pristúpené znovu v krátkom čase je vysoká.

spatial locality – je pozorovanie, že prístup na pamäťové miesto v blízkosti pamäťového miesta, na ktoré bolo nedávno pristúpené, je veľmi pravdepodobný. Príkladom tohoto pozorovania je for cyklus iterujúci cez pole prvkov.

Tieto dva princípy sú dôležité v súvislosti s cache-oblivious algoritmami, nakoľko programy ktoré sú navrhnuté v súlade s týmito princípmi sa správajú efektívne čo do práce s pamäťou. To ako pamäťová hierarchia zaručuje princíp lokality bude zrejmé z popisu kedy a ako sa dáta presúvajú medzi pamäťovými vrstvami.

2.1.2 Presúvanie a adresovanie dát v pamäťovej hierachií

Procesor môže pristupovať iba na dáta umiestnené v najližšej pamäťovej vrstve, takže temporal locality je zachovaná udržovaním často používaných dát vo vrstve čo najbližšej procesoru.

V prípade že si procesor vypýta dáta ktoré sú uložené v najbližšej pamäťvej vrstve, procesor má okamžitý prístup k týmto dátam a hovoríme že nastal *cache hit*. Ak pamäťová vrstva najbližšie k procesoru neobsahuje dáta požadované procesorom hovoríme že nastal *cache miss* (cache výpadok). V takom prípade sa dáta musia presunúť zo vzdialenejšej pamäťovej vrstvy skôr ako k nim bude môcť procesor pristúpiť. Čisto teoreticky teda môže procesor spôsobiť cache miss v každej vrstve okrem poslednej. Je zrejmé že procesor nemôže spôsobiť cache miss v poslednej pamäťovej vrstve, pretože posledná pamäťová vrstva obsahuje všetky dáta.

V každej pamäťovej vrstve sa obsah delí na po sebe idúce nepretržité kusy. V cache vrstvách sa tieto kusy nazývajú *bloky* alebo *cache bloky* a môžu byť rôzne čo do veľkosti medzi jednotlivými pamäťovými cache vrstvami. Väčšinou jeden blok obsahuje niekoľko slov. (Pozn.: Na 32 bitovom počítači veľkosť pamäťového miesta je 32 bitov – a teda môže uložiť akékoľvek číslo reprezentovateľné 32 bitmi. V tomto zmysle je slovo rovnaké ako pamäťová bunka.)

V prípade cache výpadku, spatial locality sa zachová presunutím dát medzi vrstvami po blokoch. Z toho vyplýva, že veľkosť bloku danej pamäťovej vrstvy určuje granularitu pamäťového adresovania v tej danej vrstve. Adresovanie pamäte vrámci pamäťovej hierarchie je preto iné ako adresovanie pamäti procesorom.

Čím ďalej sa pamäťová vrstva nachádza od procesoru, tým je jej veľkosť väčšia a tým sú väčšie aj pamäťové „kúsky“ v ktorých sa pamäť adresuje a presúva. V hlavnej pamäťovej vrstve sú tieto kusy veľké aj niekoľko kilobytov a používa sa pre ne pojem *stránka* (*page*) namiesto bloku. Následkom toho sa neschopnosť nájsť dáta v hlavnej pamäti nazýva *page fault* (*výpadok stránky*) namiesto cache miss. Stránka obsahujúca požadované dáta sa potom presunie z disku do hlavnej pamäte.

Kapacita pamäťovej vrstvy je určená počtom blokov alebo stránok, ktoré môže poňať. Cache sú rozdelené na *cache lines* (*cache riadky*) ktoré môžu obsahovať každá jeden blok, takže kapacita cache vrstvy sa jednoducho spočíta vynásobením veľkosti jedného bloku s počtom cache riadkov. Hlavná pamäť je rozdelená na *frames* (*rámce*), ktoré môžu poňať každá jednu stránku. Veľkosť hlavnej pamäte je definovaná počtom stránok ktoré môže poňať.

Ktorý blok sa má vymeniť v prípade cache výpadku určuje *associativity* (*asociativita*) a *replacement policy* (*stratégia výmeny*) cache pamäťovej vrstvy v ktorej cache výpadok nastal.

Asociativita obmedzuje počet cache riadkov v ktorých môže byť daný blok uložený. Cache riadky, ktoré môžu potenciálne obsahovať daný blok pamäte nazývame *cadidate line* (*kandidátny riadok*) pre daný blok. Inými slovami, blok sa mapuje na určitý počet kandidátnych riadkov. Cache sú rozdelené do troch kategórií podľa asociativity:

fully-associative (*plne asociatívna*) cache – nekladie žiadne obmedzenia na to, kam sa môže blok umiestniť. Počet kandidátnych riadkov pre ktorýkoľvek blok je rovnaký ako celkový počet cache riadkov v cache pamäti, t.j. ktorýkoľvek blok môže byť umiestnený kamkoľvek.

direct-mapped (*priamo mapovaná*) cache – je najviac obmedzujúci typ. Každý blok má iba jeden kadidátny riadok. Index kandidátneho riadku v cache je vypočítaný ako adresa bloku modulo počet cache riadkov.

set-associative (*množinovo-asociatívna*) cache – je hybrid plne asociatívnej cache a priamo mapovanej cache. X-cestná set-associative cache je rozdelená na množiny, z ktorých každá obsahuje x cache riadkov. Blok pamäte sa mapuje presne na jednu z týchto množín, ale vrámci množiny môže byť blok uložený kamkoľvek, t.j. do ktoréhokoľvek riadku. Ak je pamäť rozdelená na bloky očíslované rastúc podľa adresy od 0 do m potom index množiny v cache na ktorý sa mapuje blok a ($0 \leq a \leq m$) je vypočítaný ako $a \bmod x$. Číslo x definuje stupeň asociativity cache pamäte. Typické set-asociatívne cache majú stupeň 2, 4, 8 príp 16

V prípade cache miss v množinovo-asociatívnej alebo plne-asociatívnej cache potrebujeme nejaký typ stratégie na výber cache riadku spomedzi kandidátnych riadkov do ktorého uložíme nový blok. Ak existujú prázdne riadky medzi kandidátmi, výber je ľahký. Ak všetky kandidátne riadky obsahujú dáta, potom voľba, ktorý cache riadkok sa má vymeniť, sa robí podľa stratégie výmeny danej cache pamäte. Aby sme zachovali princíp temporal locality, optimálna stratégia výmeny by vymenila cache riadok ktorý je referencovaný najviac v budúcnosti. Zisťovanie, ktorý cache riadok by to mal byť, vyžaduje značnú znalosť postupnosti inštrukcií CPU v budúcnosti a preto sa v praxi využívajú jednoduchšie stratégie výmeny.

Pre 2-cestnú množinovo-asociatívnu cache je implementovaná LRU (Least-Recently-Used) stratégia tak, že sa drží pre každý cache riadok jeden bit: Keď sa pristúpi na cache riadok nastaví sa bit tomuto riadku a jeho susediacim riadkom sa vynuluje. Tento bit sa nazýva „*recency bit*“. Teoreticky by mohla byť LRU stratégia implementovaná aj pre vyššie stupne asociativity, použitím vaicerých „recency“ bitov. V praxi sa však používa len akási aproximácia LRU ako napr. „not-recently-used“ stratégia. Totiž pre cache obmedzenej asociativity, stratégia výmeny nemá skoro žiadny dopad na počet cache výpadkov. Dokonca bolo zistené, že pre 2-, 4- a 8-cestné set-asociatívne cache pamäte, náhodný výber riadku pre výmenu funguje skoro tak dobre ako LRU pre cache obsahujúce 16, 64 a 256 kB dáta a 64 bytové bloky [6]. Čím väčšia cache pamäť, tým menší rozdiel a pre 256 kB cache pamäte, uvádzané dve stratégie výmeny boli rovnako dobré. Pre menšie veľkosti cache pamätí vyšší stupeň asociativity vyústil do menšieho počtu cache výpadkov ale pre 256 kB cache tomu už tak nebolo.

Rozhodnúť, ktorú stránku nahradiť v hlavnej pamäti ak dôjde k výpadku stránky je oveľa dôležitejšie. Vyhnúť sa postihu za čo i len pár výpadkov stránok v hlavnej pamäti je oveľa dôležitejšie a vyplatí sa použiť chytrú stratégiu výmeny v hlavnej pamäti. Vo vrstve hlavnej pamäti je stratégia výmeny implementovaná softwarovo a teda závisí od operačného systému.

2.1.3 Kategorizácia výpadkov cache

Výpadky cache sa delia do troch hlavných kategórií podľa toho ako nastanú:

complusory výpadky – nastávajú ak je blok adresovaný prvý krát, tzn. nenachádzal sa predtým v cache pamäti. Tieto výpadky sú preto neodstrániteľné a dejú sa v každej vrstve pamäťovej hierarchie.

capacity výpadky – nastávajú keď cache nemá žiadny voľný cache riadok do ktorého by mohol byť umiestnený blok, ktorý už bol predtým v cache. Veľkosť cache pamäte a kvalita stratégie výmeny určujú počet capacity výpadkov programu. Tento typ výpadku nespôsobuje nutne výpadok v každej úrovni pamäťovej hierarchie, pr. ak bol blok vyhodený z L1 cache stále sa ešte môže nachádzať v L2 cache.

conflict výpadky – nastávajú ak adresovaný blok môže byť umiestnený iba do obsadeného cache riadku. Tento typ výpadkov môže nastať iba v cache pamätiach ktoré nie sú plne asociatívne aj napriek tomu, ak sa v cache nachádzajú prázdne riadky. Je to spôsobené faktom, že v množinovo-asociatívnych cache pamätiach sa na jeden cache riadok mapuje viacero blokov. V plne asociatívnych cache pamätiach tento typ výpadkov nenastáva, nakoľko ktorýkoľvek blok môže byť umiestnený do ktoréhokoľvek cache riadku. Ak plne asociatívna cache pamäť nemá žiadny prázdny riadok do ktorého by sa umiestnil blok, nastáva capacity výpadok a nie conflict výpadok.

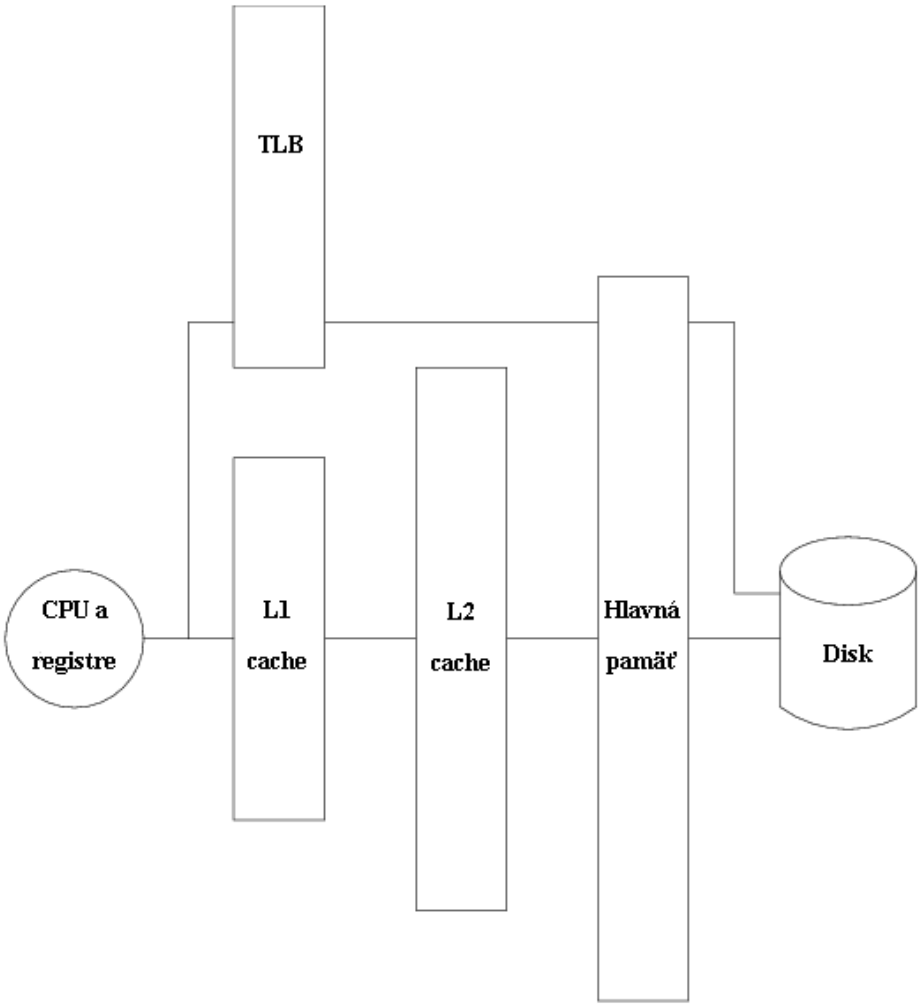
2.1.4 Virtuálna pamäť

Na vituálny pamäťový systém sa môžeme pozerať ako na cachovací systém bežiaci paralelne k hierarchickému pamäťovému systému (obr.2.3). Tento paralelný systém sa stará o presuny dát medzi primárnou a sekundárnou pamäťou. Jeho úlohou je aby si programy bežiace simultánne mysleli že majú k dispozícií celú hlavnú pamäť. Program si neuvedomuje že sa o hlavnú pamäť delí s ostatnými programami. Ako dôsledok, systém virtuálnej pamäte odstraňuje nutnosť aby programátor manažoval hlavnú pamäť manuálne. To znamená že systém virtuálnej pamäte musí zabezpečiť aby sa zdielanie hlavnej pamäte dialo bezpečne (napr. aby jeden program neprepísal dáta druhému programu).

Znalosť presných implementačných detailov nie je pre nás dôležitá, ale stojí za to spomenúť aspoň pár konceptov:

virtuálna a fyzická adresa. Je rozdiel medzi virtuálnymi adresami odkazujúcimi sa na imaginárny priestor a medzi fyzickými adresami odkazujúcimi sa na pamäť reprezentovanú pamäťovými čipmi RAM. Keďže každý proces by mal byť schopný vykonať sa tak akokeby mal k dispozícií celú hlavnú pamäť, každý proces ma svoj vlasntý virtuálny adresový priestor. Veľkosť pointeru je 32 bitov na 32 bitovej architektúre, takže počet adresovateľných pamäťových miest je obmedzený na 2^{32} , teda virtuálny adresový priestor je 4 GB. Často

sa stáva že veľa procesov beží simultánne a suma ich virtuálneho adresového priestoru je pravdepodobne väčšia ako dostupná fyzická pamäť a systém virtuálnej pamäte sa musí stále starať o to ktoré virtuálne stránky sa momentálne nachádzajú vo fytickej pamäti a ktoré na disku. Veľkosť dostupnej fyzickej pamäte neobmedzuje priamo koľko pamäte môže alokovať jediný proces. Na počítači s malým množstvom fyzickej pamäte, systém virtuálnej pamäte jednoducho musí udržovať väčšiu časť naalokovanej pamäte na disku. Ako dôsledok, menšie množstvo fyzickej pamäte znamená väčšie množstvo výpadkov stránok hlavnej pamäte.



Obr. 2.3: Virtuálna pamäť vytvára pamäťovú hierarchiu paralelne k hierarchií na obr.2.1

preklad adresy a tabuľka stránok: Keď proces adresuje nejaké dáta, musí sa virtuálna adresa preložiť na fyzickú adresu. Keďže hlavná pamäť je zdielaná medzi viacerými

programami, spravidla obsahuje iba podmnožinu celého virtuálneho priestoru adresovateľného daným procesom. Takže dáta na ktoré ukazuje virtuálna adresa môžu byť kľudne umiestnené aj na disku. Každý proces preto má tabuľku stránok obsahujúcu mapovanie z virtuálnych adries na fyzické adresy. Tabuľka stránok obsahuje informácie o tom, ktoré virtuálne stránky sa momentálne nachádzajú v hlavnej pamäti a ktoré sa nachádzajú na disku. Samotné tabuľky stránok sú umiestnené v hlavnej pamäti, ale tak ako ktorákoľvek časť hlavnej pamäte, aj tabuľka stránok môže dočasne byť presunutá na disk, ak sa tak rozhodne operačný systém.

Translation Lookaside Buffer (TLB): Tak ako tabuľka stránok, TLB obsahuje mapovanie z virtuálnej na fyzické adresy. Slúži ako cache pre tabuľky stránok tým že obsahuje najposlednejšie preklady adries. TLB je spravidla implementovaný hardwarovo aby sa zabezpečilo že často odkazované virtuálne stránky (vďaka temporal locality) sa preložia čo možno najrýchlejšie.

Keď procesor adresuje nejaké dáta, virtuálna adresa dát sa najskôr hľadá zároveň v TLB aj v L1 cache. L1 cache kontroluje či obsahuje blok s požadovanou virtuálnou adresou, TLB prekladá virtuálnu adresu na fyzickú adresu ktorú dodá L1 cache. Ak nastane hit v L1 cache na virtuálnu adresu, potom L1 použije fyzickú adresu ktorú dostalo od TLB aby skontrolovalo či daný blok je ten správny, alebo či nádhodou nepatrí inému programu a s ktorým náhodou zdieľajú rovnakú virtuálnu adresu.

2.2 Latencia pamäte

Pojem latencia pamäte sa používa na popísanie režijných nákladov spojených s pamäťovým systémom. Ale čo je v skutočnosti latencia pamäte, ako ovplyvňuje návrh pamäťových systémov?

Latencia je časový úsek, ktorý strávi jedna komponenta čakaním na inú komponentu. Inými slovami latencia znamená premrhaný čas. Teda pamäťová latencia znamená čas procesoru premrhaný čakaním na pamäťový systém. Dopad latencie pamäte je často vyjadrovaný v počte cyklov procesoru, ktoré procesor stráca čakaním na pamäťový systém. To je praktické z dvoch dôvodov:

Taktovacia frekvencia procesoru a pamäťového systému sa málokedy zhoduje. Taktovacia frekvencia pamäťového systému je menšia ako taktovacia frekvencia procesoru. Jeden typ procesoru sa často vyrába s rôznymi taktovacími frekvenciami procesoru ale iba s jednou taktovaciou frekvenciou pamäte. Takže aj keď meranie latencie v cykloch frekvencie pamäte je nezávislé od procesoru, nie je veľmi prakticky použiteľné. Latencia pamäte je zaujímavá vo vzťahu k výkonu procesoru.

Vzhľadom k stupňu paralelizmu na úrovni inštrukcií, moderné procesory sú schopné vykonať niekoľko inštrukcií behom jedného cyklu. Ak poznáme tento stupeň paralelizmu a latenciu pamäte v cykloch procesoru, môžeme bez problémov interpretovať dopad latencie jednoducho spočítaním počtu inštrukcií ktoré sa mohli vykonať kým procesor čakal na pamäťový systém.

Latencia pamäte však nie je nemenná metrika. Je špecifická pre danú úroveň pamäťovej hierarchie a zvyšuje sa čím ďalej sme od procesoru. Teda každá vrstva pamäťovej hierarchie pridáva k latencii celého pamäťového systému. Za predpokladu, že dáta sa nenačítavajú paralelne, pre každú vrstvu je latencia suma času stráveného hľadaním požadovaného bloku a času stráveného presúvaním tohoto bloku do rýchlejšej vrstvy. Preto požiadavka na dáta, ktorá spôsobí výpadky cache naprieč celou pamäťovou hierarchiou spôsobí latenciu odpovedajúcu sume latencií všetkých vrstiev.

Čas ktorý procesor strávi čakaním na pamäťový systém nie je niekedy celkom stratený. Ak inštrukcia i spôsobí výpadok cache a vykonávanie nasledujúcej inštrukcie $i+1$ nezávisí od výsledku i, procesor môže vykonať inštrukciu $i+1$ kým bude čakať na pamäťový systém.

Latencia spôsobnená konkrétnou vrstvou pamäťovej hierarchie je ovplyvnená asociativitou danej vrstvy pamäťovej hierarchie. Čo do vrstiev pamäťovej hierarchie najbližšie k procesoru, asociativita týchto vrstiev je implementovaná hardwarovo a vyšší stupeň asociativity znamená zložitejšie obvody a preto vyššiu latenciu. Úbytok výpadkov cache spôsobený vyššou asociativitou proste nevyrovná nárast latencie pamäťového systému.

2.4 Zhrnutie

V tejto kapitole sme nahliadli do kľúčových konceptov moderných pamäťových systémov a videli sme ako pamäťová hierarchia dodržuje princíp „locality of reference“. Popísali sme ako sú organizované rôzne vrstvy pamäťovej hierarchie a videli sme ako rôzne parametre ako napr. veľkosť bloku, asociativita a stratégia výmeny ovplyvňujú pohyb dát medzi jenotlivými vrstvami pamäťovej hierarchie a ako nastávajú rôzne typy výpadkov cache. Rozobrali sme pohľad na systém virtuálnej pamäte ako na hierarchiu paralelnú k pamäťovému systému. Vieme čo je pamäťová latencia a ako môže ovplyvniť čas behu programov

Kapitola 3

Prehľad Pamäťových modelov

3.1 I/O Model (External Memory Model)

Myšlienka analýzy algoritmov počítaním počtu prístupov do pamäti, ktoré algoritmus vykoná, nie je vôbec nová. V oblasti algoritmov a datových štruktúr vonkajšej pamäte je analýza zložitosti zahŕňajúca počítanie prístupov do pamäti známa už veľa rokov.

Algoritmy a datové štruktúry vo vonkajšej pamäti pracujú s množinami dát, ktoré veľkosťou ďaleko presahujú hlavnú pamäť. Hlavná myšlienka je že explicitným riadením pohybu dát medzi vnútornou a vonkajšou pamäťou môžeme minimalizovať dopad čakacej doby (latencie) spojený s prístupom na disk. Ako také, algoritmy a datové štruktúry vo vonkajšej pamäti obchádzajú systém virtuálnej pamäti.

Dôsledkom explicitného riadenia dát je, že cachovacie parametre vnútornej pamäte môžu byť optimalizované pre špecifický algoritmus, napr. stratégia výmeny (replacement policy) môže byť ušitá na mieru, lebo dokonca zmenená počas vykonávania programu. Pre uľahčenie analýzy algoritmov vo vonkajšej pamäti sa v priebehu minulosti vyvinulo niekoľko rôznych modelov. Najznámejší z nich je external-memory model (tiež známy ako I/O Model) ktorý navrhli Aggrawal a Vitter [1].

Prístup autorov k problému bol skúmať hlavné obmedzenia z hľadiska počtu I/O operácií pre vonkajšie triedenie. I/O model ktorý navrhli pre výskum pozostáva z jediného procesoru a vonkajšia pamäť je modelovaná ako všeobecný (random-access) magnetický disk. Parametre modelu sú:

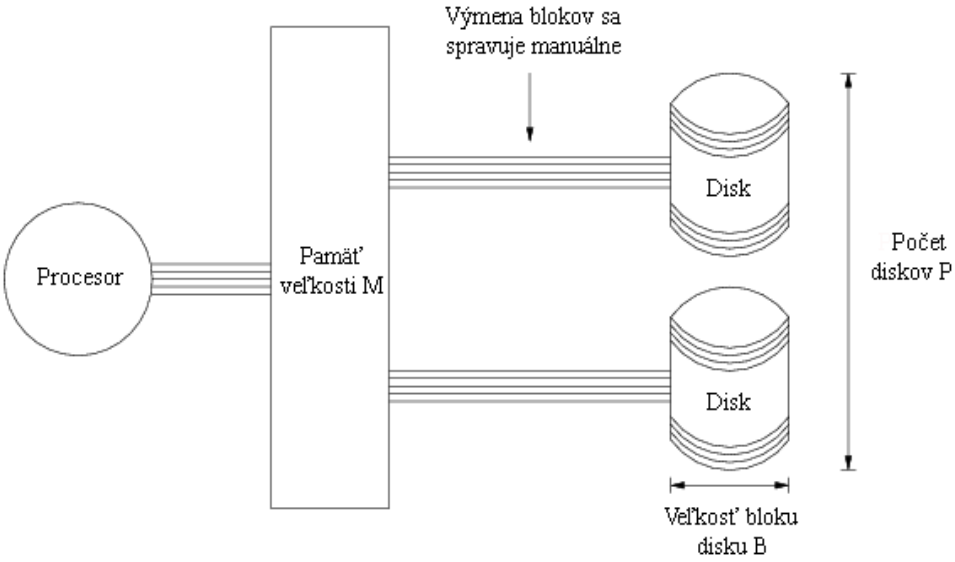
- N označuje počet triedených záznamov,
- M označuje aký počet záznamov sa zmestí do vnútornej pamäte,
- B označuje aký počet záznamov môže byť prenesených v jednom bloku,
- P označuje aký počet blokov môže byť prenesených súčasne,

kde $1 \le B \le M < N$ a $1 \le P \le \lfloor M/B \rfloor$.

Parametre N , M a B sa označujú tiež ako veľkosť súboru, veľkosť pamäte a veľkosť bloku. Každý prenos bloku má prístup k ľubovolnej súvislej skupine B záznamov na disku. Paralelizmus je v probléme inherentný dvoma spôsobmi: Každý blok môže naraz preniesť B záznamov (čo modeluje fakt že disk vie preniesť I/O operáciami blok dát zhruba rovnako rýchlo ako vie preniesť jeden bit). Druhý faktor je že súčasne môže prebiehať P prenosov blokov čo čiastočne modeluje špeciálne vlastnosti ktoré môže mať disk (napr. viacnásobné I/O kanály a čítacie/zapisovacie hlavy, schopnosť pristupovať k záznamom, ktoré nie sú uložené súvisle za sebou, v jednej I/O operácií). Na obr. 3.1 je zobrazený I/O model.

Vyššie spomenutá nerovnosť $1 \le B \le M < N$ hovorí, že blok obsahuje aspoň jede prvok a že

veľkosť problému (teda vstupných dát) musí presahovať veľkosť vnútornej pamäte. Nerovnosť $1 \leq P \leq \lfloor M / B \rfloor$ hovorí že počet blokov ktoré môžu byť prenesená naraz nesmie presiahnuť počet blokov v pamäti.

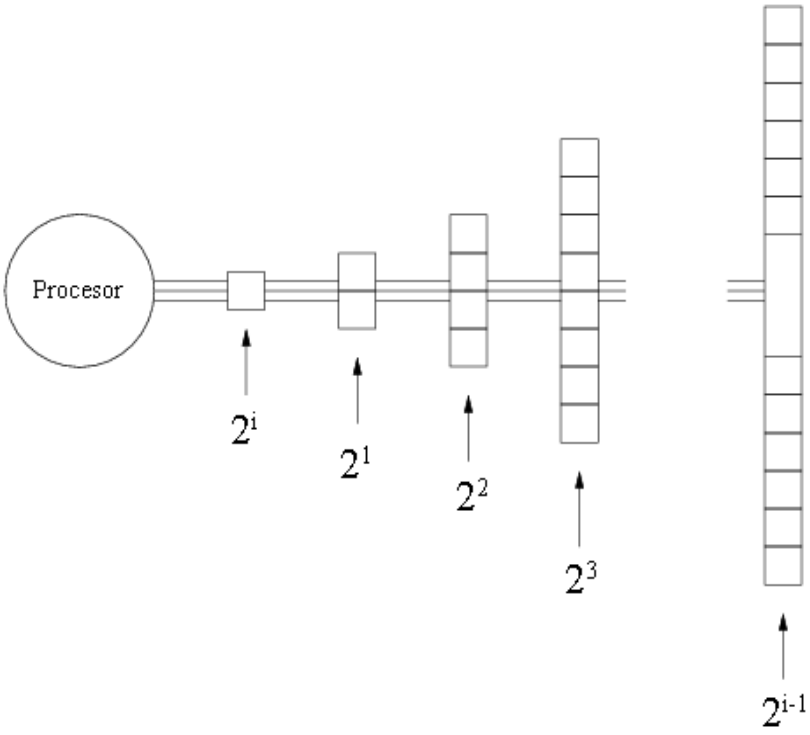


Obr 3.1: I/O model je dvojvrstvový pamäťový model pozostávajúci z vnútornej a vonkajšej pamäte. Stratégia výmeny blokov sa deje manuálne, t.j. algoritmus sám rozhoduje stratégiu výmeny a preto je znalosť B a M dôležitá.

Nevýhodou I/O modelu je že aj keď znalosť presných hodnôt B a M nie je potrebná pri navrhovaní algoritmu v modeli, tieto hodnoty musia byť známe pri implementácií algoritmu. Táto nevýhoda je zrejmá nakoľko celá myšlienka algoritmov v externej pamäti je vyladiť algoritmus na vlastnosti daného systému manuálnym presúvaním dát. Presné hodnoty B a M sa môžu líšiť systém od systému a tým obmedzujú efektívne použitie implementácie algoritmu iba na niektoré pamäťové systémy. Portovanie takýchto algoritmov na iné platformy je preto netriviálna úloha.

3.2 Hierarchický pamäťový model

V kontraste k dvojúrovňovému modelu externej pamäte hierarchický pamäťový uvažuje niekoľkoúrovňovú pamäťovú hierarchiu.
Hierarchický pamäťový model pripomína RAM. Poskytuje rovnaké operácie a tiež predpokladá potenciálne neobmedzený počet pamäťových registrov $R_1, \ldots, R_n$ (každý o veľkosti jedného integeru). Tak ako pri RAM modeli, zaujíma nás hlavne čas behu algoritmu, ale na rozdiel od RAM modelu, prístup na miesto R_i trvá $f(i)$ (oproti konštantnému času RAM modelu).
Predpokladá sa že funkcia $f(i)$ je monotónna neklesajúca a výber f určuje veľkosť a počet pamäťových vrstiev. Spravidla sa volia funkcie pr. $f(i)=x^{\alpha}$, kde $\alpha>0$ alebo $f(i)=\lceil \log_2(i) \rceil$. Druhý z príkladov naznačuje pamäťovú hierarchiu o niekoľkých vrstvách so zväčšujúcou sa veľkosťou jednotlivých vrstiev. Uvedené funkcie f sú iba príklady a je možné aplikovať aj iné funkcie f. Príklad hierarchického pamäťového modelu pre $f(i)=\lceil \log_2(i) \rceil$ je zobrazený na Obr. 3.2.



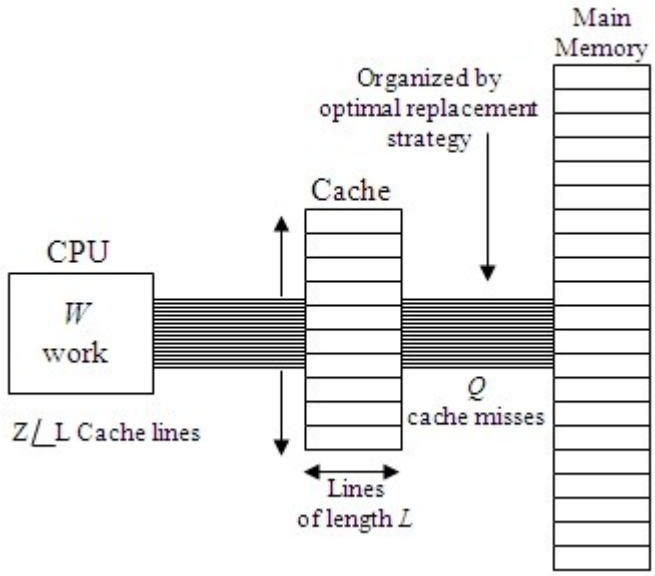
Obr. 3.2: Pre funkciu $f(i)=\lceil \log_2(i) \rceil$ hierarchický pamäťový model pozostáva z niekoľkých, polynomiálne sa zväčšujúcich, pamäťových vrstiev.

Je zrejmé že hierarchický pamäťový model simuluje správanie pamäťovej hierarchie pozostávajúcej z čím ďalej tým väčších a pomalších pamäťových vrstiev. Tento model má však svoje nevýhody. Predovšetkým zlyháva v simulácií rôznych stupňov asociativity medzi pamäťovými vrstvami. Ďalej predpokladá že dáta sú medzi vrstvami presúvané programátorom. V praxi však programátor

nemá takúto kontrolu nad dátami hlavne v rýchlejších vrstvách. A na záver, vybrať funkciu f takú, aby odrážala aspoň čiastočne realitu, je veľmi tažká úloha. Vyžaduje presné znalosti vlastností daného pamäťového systému. Týmpádom je tažké aby čas behu programu predpovedaný modelom odpovedal skutočnému času behu programu v praxi. Fukcia f by totiž asi musela byť vysoko závislá na špecifických vlastnostiach daného pamäťového systému pre ktorý bol algoritmus navrhnutý.

3.3 Ideal-cache model

Frigo, Leiserson, Prokop, Ramachandran [4] navrhli (Z, L) ideal-cache model k štúdiu cache zložitosti algoritmov. Tento model je zobrazený na obr.12 a pozostáva z počítača s dvojúrovňovou pamäťovou hierarchiou, ktorá sa skladá z ideálnej (data) cache o Z slovách a z ľubovoľne veľkej hlavnej pamäti. Pretože skutočná veľkosť slov v počítači je väčšinou malá a pevne daná (pr. 4 byty, 8 bytov a pod.), môžme predpokladať že veľkosť slova je konštanta. Cache je rozdelená do cache riadov (cache lines), každý pozostávajúci z L následných (po sebe nasledujúcich) slov, ktoré sa medzi cache a hlavnou pamäťou vždy prenášajú naraz. Návrhári cache pamätí väčšinou používajú L > 1, za účelom vylepšenia priestorovej lokality (pozn. - typ lokality referencií) k umoreniu režijnych nákladov spojených s presúvaním cache riadkov. Obvykle sa predpokladá, že cache pamäť je „vysoká“, teda že platí Z = omega(L^2), čo je väčšinou v praxi pravda.



Obr.3.3: Ideal-cache model

Procesor môže odkazovať len slová ktoré sa nachádzajú v cache. Ak odkazované slovo patrí riadku ktorý už je v cache pamäti, nastáva cache hit a slovo sa odovzdá procesoru. V opačnom prípade nastáva cache miss a riadok sa prenesie do cache. Ideálna cache pamäť je plne asociatívna: riadky môžu byť uložené kdekoľvek v cache pamäti. V prípade ak je cache pamäť plná, musí sa nejaký cache riadok vyhodiť.

Narozdiel od iných hierarchických pamäťových modelov v ktorých sú algoritmy analyzované z jedného hľadiska, ideal-cache model používa dve miery. Algoritmus so vstupnými dátami veľkosti n je skúmaný z hľadiska zložitosti (work complexity) W(n): čas behu programu v modeli RAM a cache zložitosti (cache complexity) Q(n; z, L): počet cache miss-ov spôsobených algoritmom ako funkcia veľkosti Z a dĺžky riadku L ideálnej cache pamäte. Ak sú hodnoty premenných Z a L jasné z kontextu zapisuje sa cache zložitosť jednoducho ako Q(n).

Hovoríme, že algoritmus je „cache aware“ ak obsahuje parametre (nastavované pri kompilácií alebo spúšťaní programu) ktorých nastavovaním môžme optimalizovať cache zložitosť pre konkrétnu veľkosť cache pamäte a dĺžku riadku. V opačnom prípade hovoríme, že algoritmus je „cache oblivious“

Aby sme získali predstavu o pojme „cache awareness“ vezmime si úlohu násobenia dvoch n × n matíc A a B výsledkom čoho dostaneme n × n maticu C. Predpokladáme, že všetky tri matice sú uložené po riadkoch. Za účelom zjednodušenia analýzy ďalej predpokladáme, že n je dostatočne veľké, tzn. n > L. Obvyklý spôsob ako násobiť matice na počítači s cache pamäťou je použiť blokový algortimus. Myšlienka spočíva v tom pozerať sa na každú maticu M ako na zostavu (n/s) × (n/s) podmatíc Mij (bloky), každá veľkosti s × s, kde s je ladiaci parameter.

```
BLOCK-MULT(A, B, C, n)
1 for i ←1 to n/s
2       do for j ← 1 to n/s
3             do for k ← 1 to n/s
4                   do ORD-MULT( A_ik, B_kj, C_ij, s)
```

Podprogram ORD-MULT spočíta C ← C + AB na maticiach veľkosti $s \times s$ použitím obyčajného O(s^3) algoritmu. Algoritmus popisovaný v príklade predpokladá pre jednoduchosť že s je celočíselným deliteľom n . V skutočnosti s a n nemusia byť v žiadnom (špeciálnom) vzťahu čo sa odrazí iba v zložitejšom zápise kódu.

V závislosti na veľkosti cache pamäte na počítači na ktorom bude BLOCK-MULT spustený, môžme zmenou parametra s optimalizovať algoritmus tak aby bežal rýchlejšie. Teda program BLOCK-MULT sa nazýva „cache aware“. Aby sme minimalizovali cache zložitosť zvolíme hodnotu s najväčšiu možnú tak aby sa tri $s \times s$ podmatice súčasne zmestili do cache pamäte. Matica veľkosti $s \times s$ je uložená v $\Theta(s + s^2 / L)$ cache riadkoch. Z predpokladu, že cache pamäť je „vysoká“ vidíme, že $s = \Theta(\sqrt{(Z)})$. Teda každé zavolanie ORD-MULT spôsobí najviac $Z/L = \Theta(s^2/L)$ cache missov potrebných k presunutiu matíc do cache pamäte. Z toho vyplýva že cache zložitosť celého algoritmu je $\Theta(1+n^2/L+(n/\sqrt{(Z)})^3(Z/L)) = \Theta(1+n^2/L+n^3/L\sqrt{(Z)})$, nakoľko algoritmus musí prečítať n^2 prvkov umietnených v $\lceil (n^2/L) \rceil$ cache riadkoch.

Také isté medze môžme dosiahnuť použitím jednoduchého cache-oblivious algoritmu, ktorý nevyžaduje žiadne ladiace parametre ako parameter s prípade BLOCK-MULT.

Kapitola 4

Algoritmy

4.1 Multimergesort, multiquicksort, memory-optimized heapsort

Na prácu Aggrawalla a Vittera voľne naviazali vo svojej práci LaMarca a Ladner[2]. Niektoré algoritmické myšlienky v externom pamäťovom modeli sa môžu dobre uplatniť aj v hierarchickom pamäťovom modeli. Jedná sa hlavne o multimerging a multipartitioning. Aj keď je veľa vecí medzi I/O modelom a ahierarchickým pamäťovým modelom podobných, sú tu rozdiely. Algoritmus vo vonkajšej pamäti má kontrolu nad tým ktoré dáta sú v hlavnej pamäti a ktoré na disku. Naopak pri hardwarových cache , keď nastane výpadok, algoritmus nemá na výber kam da nahrá nový blok alebo ktorý blok sa vyhodí. Ďalším rozdielom je, že v I/O modeli, jediné čo sa počíta sú I/O operácie s diskom, výpočet je „zadarmo“. Je to logické z dôvodu že diskové I/O operácie sú rádovo drahšie ako výpočtový krok. A nakoniec je tu rozdiel vo veľkosti.

Cache najbližšie k hlavnej pamäti je relatíves veľká v pomere k hlavnej pamäti. Naopak v I/O modeli, kapacita disku je väčšinou veľmi veľká v pomere k veľkosti hlavnej pamäte. Vo svojej práci LaMarca a Ladner predpokladajú že existuje veľká pamäť rozdelená na bloky a menšia cache pamäť rozdelená na C blokov. Na vstupe je n kľúčov a B je počet kľúčov ktoré sa zmestia do jedného cache bloku. Ďalej predpokladajú že kľúče sú uložene v súvislom poli veľkosti n/B pamäťových blokov.

4.1.1 Base Mergesort

Za základný algoritmus bol vybraný iteratívny mergesort, pretože je ľahko implementovateľný a prístupný tradičným optimalizačným technikám. Algoritmus používa dve polia, vstupné a pomocné. Štandardný iteračný mergesort robí horní mez($\log_2 n$) priechodov nad vstupným poľom, kde i-ty priechod zlieva zotriedené podpostupnosti dĺžky 2^{i-1} do zotriedených podpostupností dĺžky 2^i v pomocnom poli. Dôležitá optimalizácia je pozmeniť proces zlievania z jedného poľa do druhého aby sme sa vyhli kopírovaniu. Ak je počet priechodov nepárny, musíme nakoniec prekopírovať obsah pomocného poľa do vstupného poľa. Ďalše optimalizácie (na zníženie počtu vykonávaných inštrukcií):

- ukladanie podpostupností (podpolí), ktoré sa majú zlučovať, v opačnom poradí
- triedenie podpostupností dĺžky 4 rýchlou triediacou metódou na mieste.
- loop unrolling (optimalizačná technika – nepíše sa či na ktorej úrovni high/low ani sa ďalej

nerozdiskutovávajú riziká)

Počet priechodov vstupným poľom je teda horná medza($\log_2(n/4)$). Ak je toto číslo nepárne je nakoniec treba prekopírovať zotriedené pole do vstupného poľa. Náš základný mergersort vykonáva veľmi malý počet inštrukcií, najnižší zo všetkých našich comparison-based triediacich algoritmov.

4.1.2 Pamäťové optimalizácie pre Mergesort

Základný mergesort použije každé dáto iba jeden krát za priechod. Ak vstupné pole presahuje kapacitu cache pamäte, kľúče sa vyhodia predtým (skôr) ako sa použijú znovu. Ak množina kľúčov je menšia alebo rovná $BC/2$, celé triedenie sa môže vykonať v cache pamäti a utrpíme iba compulsory misses. Ak je množina väčšia ako $BC/2$ opätovná využiteľnosť (temporal reuse) prudko klesá a ak je množina väčšia ako cache pamäť, opätovné využitie je nulové. Aby sme vylepšili popísanú neefektivitu aplikujeme na základnom mergesorte dve pamäťové optimalizácie. Aplikáciou prvej vzniká tiled mergesort multimergesort

Tiled mergesort uplatňuje myšlienku nazývanú tiling, ktorá sa tiež používa pri optimalizácií kompilátorov. Tiled mergesort má dve fáze Aby sa zlepšila temporal locality, v prvej fáze triedia sa podpostupnosti dĺžky $BC/2$ použitím obyčajného mergesortu. V druhej fáze sa vrátime k základnému mergesortu a dokončíme triedenie celého poľa. Aby sme sa vyhli záverečnému kopírovaniu ak je $\log_2(n/4)$ nepárne čislo, triedime in-line podpostupnosti dĺžky 2 namiesto 4. Aplikácia techniky tiling dramaticky redukuje počet spôsobených výpadkov.

Tiling zlepšuje cache performance prvej fáze. Druhá fáza však stále trpí tým istým problémom ako základný mergeosrt. Každý priechod zdrojovým poľom v druhej fáze musí zlyhať vo všetkých blokoch a nedosiahneme žiadnej znovu použiteľnosti ak je množina väčšia ako cache. . Aplikujeme viaccestné zlučovanie podobné tomu použitému pri externom triedení. Výsledkom je algorimus nazývaný multimergesort. V multimergesorte vymeníme záverečných $\lceil \log_2(n/(BC)) \rceil$ zlučovacích priechodov tiled mergeosrtu jedným priechodom, ktorý naraz zlúčí všetky časti. Tento záverečný priechod využíva pamäťovo optimalizovanú heapu, v ktorej si ukladá hlavičky zlučovaných zoznamov/postupností.

4.1.3 Base Quicksort

Za základný quicksort bola zvolená implementáciu quicksortu poďla Sedgewick-a[3]. Sedgewick doporučuje tri základné optimalizácie. Použitie stacku namiesto rekurzie. Použitie mediánu 3 na výber pivota. Základný algoritmus nepoužíva quicksort na triedenie malých postupností. Nechá všetky podpostupnosti pod určitou hranicou (čo do veľkosti podpostupnosti) nezotriedené. Nakoniec sa počas záverečného priechodu zotriedi pole pomocou optimalizovaného insertion sortu.

4.1.4 Pamäťové optimalizácie pre Quicksort

V praxi quicksort obecne vykazuje výbornú cache performance. Pretože algoritmus prechádza vstupné pole sekvenčne , všetky kľúče v bloku sa vždy použijú. Ak je podmnožina triedeného poľa dosť malá na to aby sa zmestila do cache, quicksort spôsobí nanajvýš 1 výpadok per block skôr ako sa celá podmnožina utriedi. Aj napriek tomu faktu, aplikáciou optimalizácií na základný quicksort sa môže docieliť lepších verzií. Autori aplikáciou optimalizácií vyvinuli dva algoritmy[2]:

- memory-tuned quicksort
- multiquicksort

Memory-tuned quicksort jednoducho odstráni Sedgewickov elegantný insertion sort na konci. Namiesto toho triedi malé podpolia hneď ako sa na ne narazí použitím obyčajného insertion sortu.

Multiquicksort vzniká aplikáciou multipartitioningu, ktorým sa rozdelí vstupné pole na začiatku na množstvo menších podproblémov (ktoré majú vačšiu pravdepodobnosť zmestiť sa do cache).

Maltipartitioning sa používa v paralelných triediacich algoritmoch. Autori zvolili počet pivotov tak aby počet podpolí väčších ako cache bol malý s veľkou pravdepodobnosťou. Je známe že ak rozdelíme k bodov náhodne na úsečke o veľkosti 1, pravdepodobnosť že výsledný podúsek bude mať dĺžku x je $(1-x)^k$. Multiquicksort delí vstupné pole na $3n/BC$ kúskov, na čo je potrebných 3{n}/{B}{C} – 1 pivotov. Po rozdelení je teda pravdepodobnosť že podpole bude väčšie ako BC je $(1-BC/n)^{(3n/BC-1)}$.

4.1.5 Heapsort

Za základný heapsort autori zvolili Williams-Floydovu varaintu heapsortu, tzv. bottom-up heapsort.

Pamäťové optimalizácie pre heapsort zahrňujú nahradenie binárnej haldy B-haldou, kde B je počet kľúčov ktoré sa zmestia do jendého cache bloku. Ďalšou optimalizáciou je zarovnať haldu v pamäti tak že všetci synovia (ktorých počet je B) sú v jednom cache bloku.

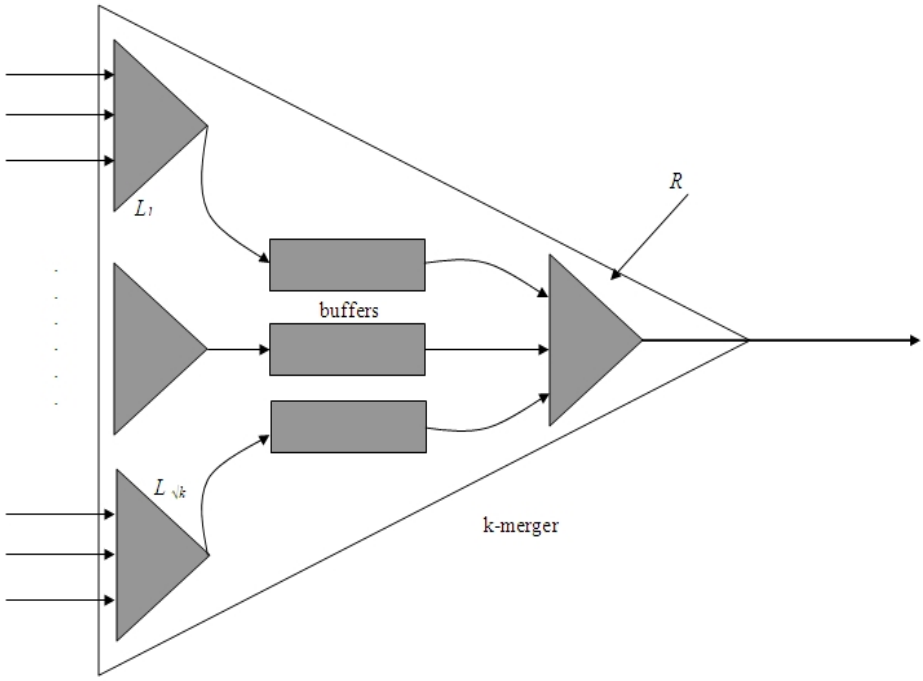
4.2 Funnelsort

Cache-oblivious algoritmy ako napr. two-way mergesort nie sú optimálne (asymptoticky) vzhľadom k počtu cache výpadkov. Funnelsort je triediaci, cache-oblivious algoritmus. Má optimálnu $O(nlogn)$ zložitosť a optimálnu cache zložitosť $O(1+(n/L)(1+\log_z(n)))$

Funnelsort je podobný mergesortu. Na zotriedenie súvislého poľa n prvkov funnelsort vykonáva nasledujúce dva kroky:

1. Rozdeľ vstupné pole na $n^{(1/3)}$ súvislých polí veľkosti $n^{(2/3)}$ a zotrieď tieto polia rekurzívne.
2. Zlej $n^{(1/3)}$ zotriedených postupností použitím $n^{(1/3)}$ -mergeru, ktorý je popísaný nižšie.

Funnelsort sa líši od mergesortu v spôsobe akým funguje (akým sa vykonáva) zlievanie. Zlievanie sa vykonáva zariadením nazývaným k -merger, ktoré na vstupe dostane k zotriedených postupností a zleje ich. K -merger pracuje tak, že rekurzívne zlieva zotriedené postupnosti ktoré sa postupne stávajú dlhšími. Narozdiel od mergesortu k -merger pozastaví prácu na zlievaní podproblému keď sa výstupná postupnosť stane „dostatočne dlhou“ a začne pracovať na zlievaní iného podproblému. Na obrázku obr.4.2 je zobrazenie k-mergeru:



Obr.4.2: *k-merger je vystavaný rekurzívne z* $\sqrt{(k)}$ *„ľavých“* $\sqrt{(k)}-$*mergerov* $L_1,\ldots,L_{\sqrt{(k)}}$ *, zo skupiny bufferov a jedného „pravého“* $\sqrt{(k)}-$*mergeru*

Počas práce k -merger udržuje nasledujúci invariant.

Invariant. Každé zavolanie k -mergeru vyprodukuje nasledujúcich k^3 prvkov zotriedeného poľa, získaného zlievaním k vstupných postupností.

K-merger je vybudovaný rekurzívne z $\sqrt{k}$ -mergerov nasledujúcim spôsobom: k vstupov je rozdelených do $\sqrt{k}$ množín po $\sqrt{k}$ prvkoch ktoré tvoria vstup pre $\sqrt{k}$ $\sqrt{k}$ -mergerov $L_1, L_2, \ldots, L_{\sqrt{k}}$ v ľavej časti obrázku. Výstupy týchto mergerov sú napojené na vstupy $\sqrt{k}$ bufferov. Každý buffer je FIFO veľkosti $2k^{3/2}$. Nakoniec výstupy bufferov sú napojené na $\sqrt{k}$ vstupov $\sqrt{k}$ -mergeru R v pravej časti obrázku. Výstup tohoto posledného $\sqrt{k}$ -mergeru je výstupom celého k-mergeru. Buffery sú predimenzované, každý môže poňať $2k^{3/2}$ prvkov čo je dvojnásobok $k^{3/2}$ prvkov vyprodukovaných jedným $\sqrt{k}$ -mergerom. Toto predimedzovanie je potrebné pre správny chod algoritmu ako bude ukázané ďalej. Konečným prípadom rekurzie je k-merger s k rovným 2 ktorý vyprodukuje k^3 = 8 prvkov.

K-merger pracuje rekurzívne. Aby vyprodukoval k^3 prvkov k-merger zavolá R $k^{3/2}$ krát. Pred každým zavolaním k-merger naplní všetky buffre ktoré sú menej ako spolovice plné (tzn. všetky buffre ktoré obsahujú menej ako $k^{3/2}$ prvkov). Aby sme mohli naplniť buffer i algoritmus zavolá odpovedajúci „ľavý“ merger L_i jeden krát. Keďže L_i vyprodukuje $k^{3/2}$ prvkov, buffer obsahuje aspoň $k^{3/2}$ prvkov po tom ako L_i skončí.

Dá sa indukciou dokázať, že zložitosť funnelsortu je $O(n\log_2(n))$ a že funnelsort na n prvkoch vyžaduje najviac $Q(n)$ cache výpadkov, kde $Q(n)=\Theta(1+(n/L)(1+\log_Z n))$.

Kapitola 5

Testy

Táto kapitola pojednáva o metodike a prostredí testov vybraných triediacich algoritmov. Cieľom testov nebolo navzájom porovnať všetky možné známe triediace algoritmy, ale v praxi naimplementovať a reálne overiť pár vybraných algoritmov.

5.1 Prostredie

Testy boli spúšťané v nasledujúcom prostredí:

Hardware:

- Procesor: CPU Intel Celeron D 336, 4-cestná 256kB L2 cache pamäť s veľkosťou bloku 128 bytov
- Matičná doska: Asus P5LD2
- Pamäť: DDR SDRAM 512 MB

OS: Gentoo Linux, verzia jadra 2.6.2.2

Kompilátor: gcc verzia 4.12

$ cat /proc/cpuintfo

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 4
model name      : Intel(R) Celeron(R) CPU 2.80GHz
stepping        : 9
cpu MHz         : 2810.137
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe lm constant_tsc pni
monitor ds_cpl tm2 cid cx16 xtpr lahf_lm
bogomips        : 5624.95
clflush size    : 64
```

Prečo sme pre testy vybrali práve prostedie unixového systému bude vysvetlené ďalej.

5.2 Meranie času

Meranie času behu program je na počítačových systémoch obecný problém. Neexistuje portabilná metóda naprieč všetkými platformami, ktorá by zaručila presné zmeranie času behu daného programu. Dôvod, prečo je tomu tak spočíva jednak v rôznosti operačných systémov ale hlavne v tom, že jadro operačného systému pri súčasnom behu viacerých programov musí nutne procesy preplánovávať a teda zmeraný čas behu programu spravidla zahrňuje aj časti behov iných procesov. Ideálne by bolo vytvoriť nepreemptívny proces, ktorého kritické časti merania nemôžu byť prerušené prerušením. V takom prípade by namerané hodnoty boli presné a použiteľné. Presne z toho dôvodu bol pre testy zvolený unixový operačný systém (linux). Vyššie popísaný spôsob merania totiž môžeme ľahko dosahnuť následovne:

- nabootovaním cez `init=/bin/sh` , čím nám systém pobeží bez nevyhnutných služieb. Výsledkom je naozaj minimálne zaťažený systém.
- zamaskovaním prerušení , čím dosiahneme nepreemptívnosť procesu. Teda ak by aj jadro chcelo náš meraný program prerušiť , stane sa tak až po jeho dokončení.

Technicky sa to dá docieliť pomocou funkcie `iopl(3)` ktorým sa prepneme do príslušného runlevelu (čo potrebujeme k zakázaniu prerušení) a následným zakázaním prerušení volaním `asm("CLI")`. Po skončení samotného výpočtu môžme bezpečne povoiľ prerušenia pomocou `asm("STI")`. Stačí nám chrániť výpočtové časti programu.

Samotné meranie času sa dá robiť viacerými spôsobmi. Pôvodne sme zamýšľali merať čas pomocou počítania cyklov procesoru (teda cez cycle counter), tak ako to navrhuje [5]. Táto metóda vyžaduje funkciu v nasledujúcom zmysle:

```
void access_counter(unsigned *hi, unsigned *lo)
{
 asm("rdtsc; movl %%edx,%0; movl %%eax, %1"
     : "=r" (*hi), "=r" (*lo)
     : /**/
     : "%edx", "%eax");
}
```

Teda pred meraným a po meranom úsekom zaznačíme stav počítadla a odčítame. Táto metóda je veľmi presná ale nie je nutne portabilná a stále trpí nepresnosťami v prípade veľkého zataženia systému. Preto som sa rozhodol pre meranie funkciou clock_t gettime() ktorá je skoro rovnako presná (presnosť v nanosekundách) a je portabilná. Nakoľko sme vyššie popísanou metódou zaručili, že proces pobeží bez prerušenia s maximom dostupných systémových prostriedkov, je táto metóda merania času dostačujúca.

5.3 Realizácia testov

Pre realizáciu testov sme vybrali algoritmy popísané v [2] (Kapitola 3.1). Naimplementované boli všetky algoritmy okrem poslednej optimalizácie hepasortu.

5.3.1 Testovacie dáta

Testy boli realizované na dvoch typoch vstupných dát a to na integeroch a reťazcoch. Reťazce boli veľkosti 128 bytov.

Parametre a CB algoritmov z [2] boli pre prostredie behu testov zvolené následovne. Pre typ integer boli parametre $B=32$ a $C=2048$. Pre typ string (reťazec o veľkosti 128 bytov) boli parametre $B=1$ a $C=2048$.

Kedže testované algoritmy sú algoritmami vo vnútornej pamäti, je vhodné aby veľkosť vstupných dát bola taká veľka aby mohol algoritmus bežať čisto vo vnútornej pamäti bez nutnosti swapovania, čo je zrejme logické. Otázkou je ako zistiť pre danú veľkosť vnútornej pamäte a veľkosť vstupných dát či daný proces pobeží bez nutnosti swapovania.

Dá sa to urobiť empiricky, nasledujúcim spôsobom. Systému sa zakáže swapovanie príkazom `swapoff -a` a postupne sa pre, po malých krokoch, zväčšujúce sa vstupné dáta otestujú naraz všetky metódy. V prípade že program spadne (skončí) je to výsledok toho že v systému už nebola voľná pamäť – a tým dostaneme hornú hranicu veľkosti vstupných dát.

Za generátor permutácií som zvolil funkciu lrand48() ktorá generuje pseudo-náhodné čísla uniformne distribuované v rozmedzí (0, 2^{31}). Ako seed som použíl čas.

5.3.2 Implementačné poznámky

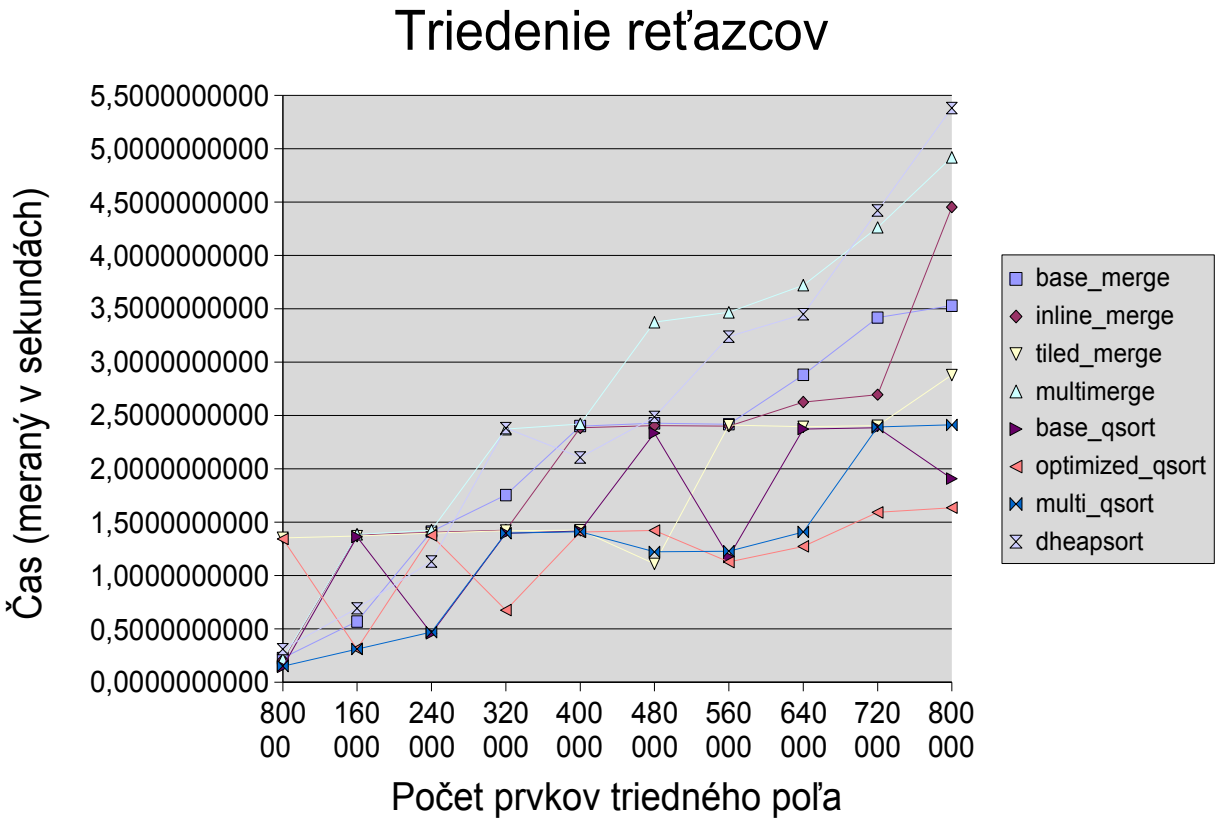
Zvolený jazyk testov je jazyk c.
Operácie porovnania, kópie a výmeny boli implementované týmto spôsobom:

```
compare()
{
        no_comparisons++;
        /*  Porovnanie  */
}

swap()
{
        no_swaps++;
        /*  Výmena prvkou pomocou volania funkcie copy()  */
}

copy()
{
        no_copies++;
        /*  kopírovanie  */
}
```

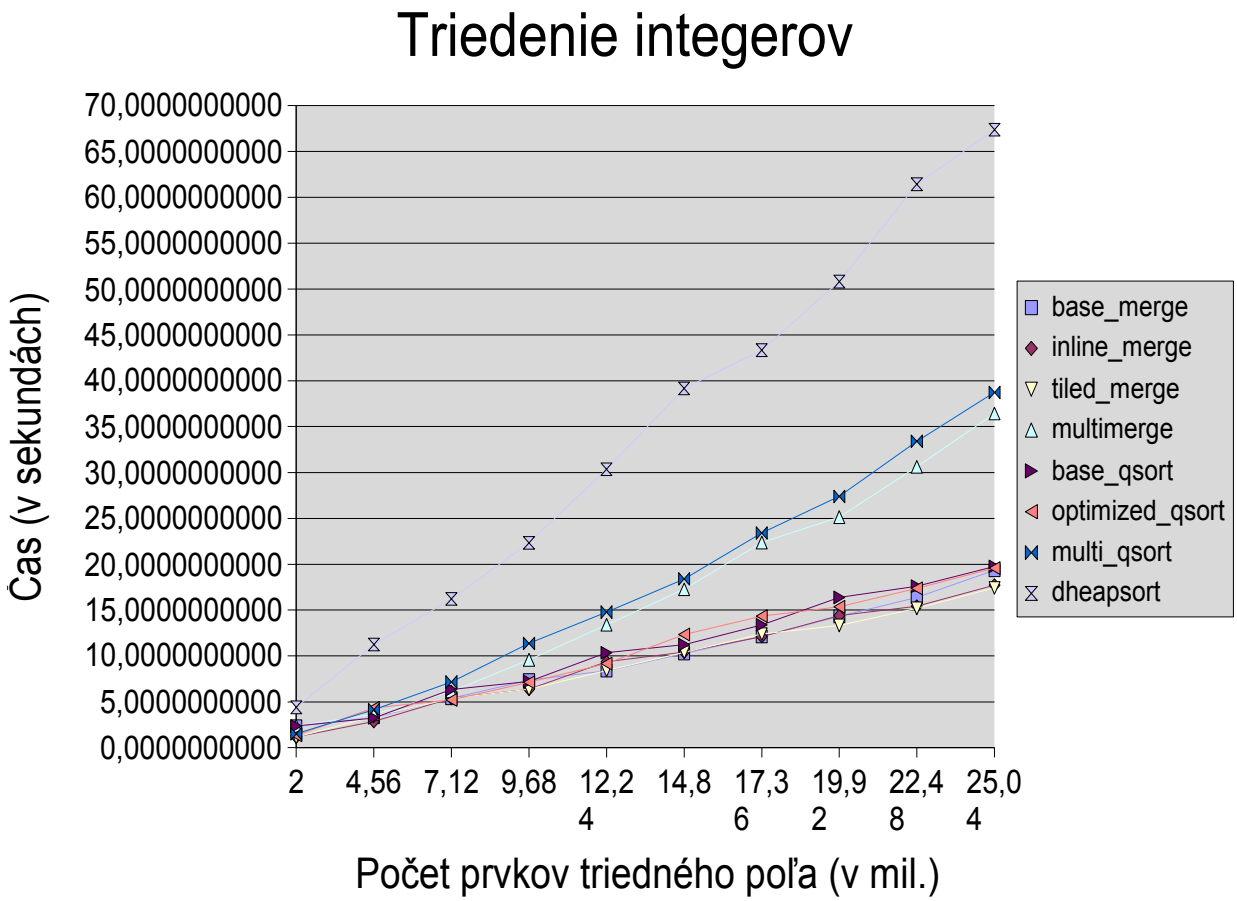
Z tohoto dôvodu je v tabuľkách výsledkov uvádzaný počet kópií s poznámkou že zahŕňa výmeny.

5.4 Výsledky testov

Cieľom tejto práce bolo hlavne prakticky implementáciou overiť proklamované teoretické výsledky niektorých triediacich algoritmov. Táto úloha sa nám podarila avšak výsledky testov uvedené v tejto kapitole naznačujú že vylepšenia algoritmov nemusia nutne viesť k celkovému zrýchleniu času za ktorý sa vykonajú.
Tabuľky nameraných dát sa nachádzaju v prílohách. Tabuľka behu s celými číslami sa nachádza v Prílohe B a tabuľka behu s reťazcami sa nachádza v Prílohe A. Časy uvádzané v tabuľkách v prílohách sú uvedené v sekundách.
Prvý z grafov je test na triedenie reťazcov, kde je vidieť že najlepšie sú na tom quicksorty. Lepšie priemerné časy vykazuje quicksort ako jeho optimalizované varianty. Takisto základný mergesort si vedie lepšie ako multimergesort, ale už nie ako tiled mergesort. Toto je v kontraste s teoretickými výsledkami uvádzanými v [2].



Druhý z grafov uvádza beh triediacich algorimov na celých číslach. Najhoršie si vedie d-árny heapsort. Rozdiel medzi mergesortom a multimergesortom je minimálny, tak ako u všetkých variánt quicksortov. Pomerne dobre si vediet tiled mergesort.



Tieto uvádzané grafy a hodnoty sú samozrejme len časťou možných veľkostí vstupných dát a je možné že pre ďaleko väčšie dáta by optimalizované varianty boli nesporne rýchlejšie. Avšak treba brať do úvahy aj výpočtové prostredie a to v takomto zmysle: Pokiaľ by testovanie (a dôkaz) opodstatnenia optimalízácií vyžadovalo vstupné dáta ďaleko preshujúce možné veľkosti vnútorných pamätí, tak takéto optimalizácie strácajú sčasti zmysel, nakoľko ich zameraním je vylepšiť rýchlosť programu _vnútorného_ triedenia. V prípade dát presahujúcich vnútornú pamäť musíme brať do úvahy swapping a to už je doména externých algoritmov, kde algoritmy vnútorného triedenia nemôžu konkurovať.

Príloha A

Základný mergesort bez optimalizácií

počet prvkov	80000	160000	240000	320000	400000
čas	0,2280804430	0,5702965320	1,4132789364	1,7553557800	2,4012990498
# porovnaní	1229185	2618814	4002244	5558066	7065935
# kópií (zahŕňa výmeny)	1440000	3200000	4800000	6400000	8000000
# výmen	0	0	0	0	0

počet prvkov	480000	560000	640000	720000	800000
čas	2,4276430347	2,4182560128	2,8815633590	3,4185351248	3,5298262280
# porovnaní	8485666	10375177	11755045	13228551	14931937
# kópií (zahŕňa výmeny)	9600000	12320000	14080000	15840000	17600000
# výmen	0	0	0	0	0

Mergesort s in-line triedením

počet prvkov	80000	160000	240000	320000	400000
čas	0,2092465950	1,3819008780	1,4051454410	1,4261531561	2,3856640070
# porovnaní	1235935	2632121	4022258	5584638	7099063
# kópií (zahŕňa výmeny)	1280000	2880000	4320000	5760000	7200000
# výmen	0	0	0	0	0

počet prvkov	480000	560000	640000	720000	800000
čas	2,4056181651	2,4016479158	2,6260307780	2,6960423660	4,4538405814
# porovnaní	8525838	10421746	11808462	13288332	14998574
# kópií (zahŕňa výmeny)	8640000	11200000	12800000	14400000	16000000
# výmen	0	0	0	0	0

Tiled Mergesort

počet prvkov	80000	160000	240000	320000	400000
čas	1,3540739328	1,3704035877	1,3989636747	1,4210777834	1,4252678308
# porovnaní	1235935	2618814	4002244	5584638	7099063
# kópií (zahŕňa výmeny)	1280000	3039968	4559968	5760000	7200000
# výmen	0	0	0	0	0

počet prvkov	480000	560000	640000	720000	800000
čas	1,1106093570	2,4099297216	2,3953573939	2,4051107500	2,8801449800
# porovnaní	8525838	10375177	11755045	13228551	14931937
# kópií (zahŕňa výmeny)	8640000	11759968	13439968	15119968	16799968
# výmen	0	0	0	0	0

Multimergesort

počet prvkov	80000	160000	240000	320000	400000
čas	0,2181762660	1,3873488682	1,4264252795	2,3726845587	2,4219643627
# porovnaní	1879521	4078081	6379870	8795411	11254000
# kópií (zahŕňa výmeny)	559968	1119968	1679968	2239968	2799968
# výmen	0	0	0	0	0

počet prvkov	480000	560000	640000	720000	800000
čas	3,3751291250	3,4681032200	3,7217246850	4,2628070430	4,9196012430
# porovnaní	13719006	16248695	18866602	21487437	24112463
# kópií (zahŕňa výmeny)	3359968	3919968	4479968	5039968	5599968
# výmen	0	0	0	0	0

Základný quicksort

počet prvkov	80000	160000	240000	320000	400000
čas	0,1457516330	1,3623659073	0,4588565900	1,3969148523	1,4171240956
# porovnaní	1454431	3158655	4835219	6716060	8394190
# kópií (zahŕňa výmeny)	1381864	2865485	4404243	5955496	7546469
# výmen	407288	848495	1308081	1771832	2248823

počet prvkov	480000	560000	640000	720000	800000
čas	2,3367255195	1,1754509030	2,3740669431	2,3873341468	1,9092032840
# porovnaní	10207094	12031401	13841924	15702460	17889474
# kópií (zahŕňa výmeny)	9160791	10781806	12423596	14076414	15682285
# výmen	2733597	3220602	3714532	4212138	4694095

Pamäťovo optimalizovaný quicksort

počet prvkov	80000	160000	240000	320000	400000
čas	1,3435484209	0,3138708490	1,3766856319	0,6770760680	1,4091337382
# porovnaní	1418041	3086132	4726652	6571308	8212803
# kópií (zahŕňa výmeny)	1360758	2823309	4341087	5871134	7440723
# výmen	407288	848495	1308081	1771832	2248823

počet prvkov	480000	560000	640000	720000	800000
čas	1,4230735187	1,1281993390	1,2744862390	1,5924959580	1,6369577090
# porovnaní	9989607	11777724	13552059	15376439	17526723
# kópií (zahŕňa výmeny)	9033947	10634078	12254838	13886470	15471145
# výmen	2733597	3220602	3714532	4212138	4694095

Multiquicksort

počet prvkov	80000	160000	240000	320000	400000
čas	0,1519517910	0,3117350620	0,4708995720	1,3984943861	1,4119865577
# porovnaní	1421982	3022776	4709047	6388327	8099743
# kópií (zahŕňa výmeny)	1224350	2468626	3734610	5016701	6319147
# výmen	301334	609309	924520	1245100	1572465

počet prvkov	480000	560000	640000	720000	800000
čas	1,2223978300	1,2296429870	1,4090277500	2,3932223089	2,4130072462
# porovnaní	9952872	11777981	13552885	15365262	17279097
# kópií (zahŕňa výmeny)	7649520	8998733	10393414	11814330	13181915
# výmen	1909138	2252092	2610202	2977057	3326135

D-árny heapsort

počet prvkov	80000	160000	240000	320000	400000
čas	0,3114432530	0,6935568010	1,1323130810	2,3834584419	2,1080616700
# porovnaní	2395407	5110328	7942562	10859772	13838655
# kópií (zahŕňa výmeny)	1437051	3033508	4689500	6386411	8114903
# výmen	0	0	0	0	0

počet prvkov	480000	560000	640000	720000	800000
čas	2,4862091860	3,2401922200	3,4491664310	4,4207814612	5,3825651244
# porovnaní	16843820	19892451	23001111	26132438	29277971
# kópií (zahŕňa výmeny)	9857120	11620815	13414380	15219945	17030874
# výmen	0	0	0	0	0

Príloha B

Základný mergesort bez optimalizácií

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	2,3498102020	3,3019741500	5,3938346252	7,3786110399	8,3896436400
# porovnaní	39432038	97901543	153830196	216998799	274196164
# kópií (zahŕňa výmeny)	44000000	109440000	170880000	251680000	318240000
# výmen	0	0	0	0	0

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	10,2716384440	12,1221698890	14,3160018000	16,3863433620	19,3469993025
# porovnaní	335997449	409539940	465478758	525893412	586053260
# kópií (zahŕňa výmeny)	384800000	451360000	517920000	584480000	651040000
# výmen	0	0	0	0	0

Mergesort s in-line triedením

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	1,1821253520	2,8522683710	5,3797039533	6,3682019250	9,3628315373
# porovnaní	39598878	98280624	154423337	217805504	275215314
# kópií (zahŕňa výmeny)	40000000	100320000	156640000	232320000	293760000
# výmen	0	0	0	0	0

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	10,4266696149	12,1199296970	14,3972780207	15,4272259778	17,7134895990
# porovnaní	337230088	410987193	467140504	527766963	588140075
# kópií (zahŕňa výmeny)	355200000	416640000	478080000	539520000	600960000
# výmen	0	0	0	0	0

Tiled mergesort

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	1,1369543220	3,4072658624	5,3737968036	6,3858438420	8,4264600403
# porovnaní	39598878	98280624	154423337	216998799	274196164
# kópií (zahŕňa výmeny)	39997696	100320000	156637696	241994752	305996544
# výmen	0	0	0	0	0

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	10,3970023035	12,3926123890	13,3263435230	15,1888203830	17,4173726611
# porovnaní	335997449	410987193	467140504	527766963	588140075
# kópií (zahŕňa výmeny)	369994752	416637696	478080000	539517696	600960000
# výmen	0	0	0	0	0

Multimergesort

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	2,3733156050	3,6470778680	6,1323522210	9,5796717710	13,3986006872
# porovnaní	49419652	123563036	202228579	283509534	367332302
# kópií (zahŕňa výmeny)	29996544	68394752	106796544	145194752	183596544
# výmen	0	0	0	0	0

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	17,2712586900	22,3688184396	25,1365433840	30,6214902720	36,4273158229
# porovnaní	451132934	536119516	625026896	713947949	802856746
# kópií (zahŕňa výmeny)	221994752	260396544	298794752	337196544	375594752
# výmen	0	0	0	0	0

Základný quicksort

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	2,3644652934	3,2542049970	6,3538583621	7,2379885650	10,3543379130
# porovnaní	46856068	115228676	190697846	258205867	334475727
# kópií (zahŕňa výmeny)	41209930	97587837	155147012	214328227	273625668
# výmen	12403310	29489279	46969004	64989409	83048556

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	11,2448854430	13,3975233730	16,3796562812	17,6051267520	19,7653290420
# porovnaní	408014481	482971115	551006835	636896170	705846438
# kópií (zahŕňa výmeny)	333736472	394474960	456921684	516860966	580101625
# výmen	101378824	119918320	139027228	157300322	176673875

Pamäťovo optimalizovaný quicksort

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	1,3371636190	4,3514761565	5,2180465220	7,1348842710	9,1690232330
# porovnaní	45949851	113162787	187473756	253819997	328931752
# kópií (zahŕňa výmeny)	40682486	96385847	153270254	211774645	270396464
# výmen	12403310	29489279	46969004	64989409	83048556

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	12,3383704007	14,3521062675	15,3617217250	17,3714760900	19,5750433270
# porovnaní	401311297	475112004	541984363	626720045	694516331
# kópií (zahŕňa výmeny)	329832516	389896238	451666512	510934076	573499347
# výmen	101378824	119918320	139027228	157300322	176673875

Multiquicksort

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	1,5137272660	4,1484930900	7,1651316400	11,3729937025	14,7796370600
# porovnaní	47171993	119121860	200614012	289903455	386974627
# kópií (zahŕňa výmeny)	36639017	102089250	189503815	297889580	426427869
# výmen	9543411	27943072	53664177	297889580	125804695

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	18,3937179110	23,4067140848	27,3904156150	33,4056357006	38,7279224500
# porovnaní	493708903	606780836	726098901	851535592	990393689
# kópií (zahŕňa výmeny)	576859585	747695639	940055736	1150680259	1389879296
# výmen	172531517	226059785	286762734	353553825	429869754

D-árny heapsort

počet prvkov	2000000	4560000	7120000	9680000	12240000
čas	4,4040643789	11,2402791250	16,2142872570	22,3330589410	30,3556442720
# porovnaní	78392104	189597883	305391684	423642013	544308145
# kópií (zahŕňa výmeny)	45176799	108434760	173986964	240764708	308749898
# výmen	0	0	0	0	0

počet prvkov	14800000	17360000	19920000	22480000	25040000
čas	39,1668669660	43,3377366440	50,8054795610	61,4207541619	67,3537261170
# porovnaní	665910200	788521682	913402198	1039039029	1165240878
# kópií (zahŕňa výmeny)	377208723	446165792	516263704	586741614	657493545
# výmen	0	0	0	0	0

Použitá Literatura

[1] Aggarwal A., Vitter J. S. (1988): The input/output complexity of sorting and related problems . Communications of the ACM 31(9) , 1116-1127

[2] Ladner E.R., LaMarca A. (1998): The Influence of Caches on the Performance of Sorting. Journal of Algorithms 31, 66-104

[3] R.Sedgewick, Implementing quicksort programs, Comm. ACM 21(10), 847-857

[4] Frigo, Leiserson, Prokp, Ramachandran, Cache-Oblivious Algorithms.

[5] R.E. Bryant, D.R.O Hallaron: Computer Systems, A Proggramer's Perspective. (2001, manuscript) Prentice Hall, 2003

[6] D.A.Patterson, J.L. Hennessy: Computer Organization and Design, 2nd edition, Morgan Kaufmanm Publishers Inc. (1996)