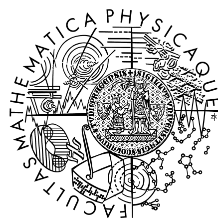
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jan Ondruš

LZPXj - vylepšení LZP algoritmu

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Jan Lánský
Studijní program: Informatika, obor Obecná informatika

2007

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

V Praze dne 22.5.2007 Jan Ondruš

Obsah

1 Úvod **5**
1.1 Komprese dat 5
1.2 Algoritmus LZP 6
1.3 Implementace LZP algoritmu - verze od Charlese Blooma . . 7
1.4 Předchozí práce - program LZPX 8

2 Vylepšení LZPXj **10**
2.1 LZPXj - úvod 10
2.2 Vylepšení výkonu pro špatně komprimovatelná data 11
2.3 Komprese pro 16 / 24 / 32 bitová data 13
2.4 Filtr pro instrukční kód x86 15

3 Kódování v LZPXj **18**
3.1 Kódování - 2 modely 18
3.2 Model 1 - PPM 18
3.3 Model 2 19
3.4 Výběr modelu 20

4 Shrnutí a závěr **21**
4.1 Celkový postup komprese 21
4.2 Srovnání komprese 22
4.3 Závěr 26

Literatura **28**

Název práce: LZPXj - vylepšení LZP algoritmu
Autor: Jan Ondruš
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Jan Lánský
e-mail vedoucího: Jan.Lansky@mff.cuni.cz

Abstrakt: LZP algoritmus je jedním z algoritmů navržených pro kompresi textu. Vychází ze základní myšlenky, kterou publikovali Ziv a Lempel ve své práci "A Universal Algorithm for Sequential Data Compression". Autorem vlastní metody LZP je Charles Bloom, který navrhl obecné schéma a několik vlastních implementací. Cílem práce je navázat na existující implementace LZP algoritmu. Jsou zde zapracovány vlastní myšlenky s cílem dosáhnout efektivní komprese pro různé druhy dat. Program LZPXj se zaměřuje na kompresi běžných souborů jako jsou textové, multimediální nebo spustitelné soubory. Součástí práce jsou kromě popisu vylepšeného algoritmu také otestování a srovnání s existujícími kompresními programy.

Klíčová slova: komprese dat, LZP algoritmus, srovnání komprese

Title: LZPXj - improvement of LZP algorithm
Author: Jan Ondruš
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský
Supervisor's e-mail address: Jan.Lansky@mff.cuni.cz

Abstract: LZP algorithm is one of algorithms designed for text compression. It is based on ideas published in paper "A Universal Algorithm for Sequential Data Compression" by Ziv and Lempel. The author of method LZP itself is Charles Bloom who proposed general scheme and his implementations. The aim of present work is continue progress of existing implementations of LZP algorithm and designing software for file compression. Some original ideas are proposed and included helping achieve effecient compression for various kind of data. LZPXj compressor is better for common files containing text, multimedia or executables. Part of this work is apart from description of improved algorithm also tests and comprehensions with some existing compression programs.

Keywords: data compression, LZP algorithm, compression comprehension

Kapitola 1

Úvod

1.1 Komprese dat

Komprese dat je proces kódování informace pomocí menšího počtu bitů (nebo jiné jednotky informace) než měla původní reprezentace. Dosahuje toho použitím různých schémat. Komunikace pomocí komprese dat funguje pouze, pokud odesílatel i příjemce znají schéma, které v jejich případě použili. V opačném případě by příjemce nebyl schopen dekódovat obsaženou informaci. Komprese je užitečná, protože pomáhá ušetřit zdroje jako je místo na úložném médiu nebo přenosovou kapacitu na síti. Na druhou stranu je nevýhodou, že je potřeba čas na kompresi a dekompresi dat a výpočetní kapacita schopná ji provést.

Kompresní algoritmy můžeme rozdělit na dvě základní skupiny. Těmi jsou bezztrátová a ztrátová komprese. Bezztrátové algoritmy se snaží využít statistické redundance dat k lepší reprezentaci odesílaných dat. V tomoto případě je příjemce schopen rekonstruovat původní data naprosto přesně, tedy sekvence bitů si přesně odpovídají. Dalším druhem komprese je ztrátová komprese, při které je možná ztráta "kvality" přijatelná. Obvykle se používá například pro kompresi obrázků, videa nebo zvuku.

Bezztrátové kompresní metody můžeme rozdělit podle typu dat, pro který jsou navrženy. Těmi jsou hlavně text, obrázky a zvuk. V principu lze obecný kompresní algoritmus použít na jakákoliv binární data. Některé algoritmy ale mohou mít problémy s výkonem komprese pro jiný druh dat než pro ten, pro který byly navrženy. Například zvuková data nelze efektivně komprimovat textovými algoritmy. Většina bezztrátových programů používá dva druhy algoritmů - první provádí statistické modelování vstupních dat a

druhý kóduje vstupní data do bitových řetězců, tak aby častější data dávala kratší výstup. Příkladem algoritmů první skupiny jsou Burrows-Wheelerova transformace (BWT), LZ77 nebo LZW. Naopak kódovacími algoritmy jsou například Huffmanovo kódování nebo aritmetické kódování.

1.2 Algoritmus LZP

Metodu LZP představil v roce 1996 Charles Bloom v [1]. Řadíme ji mezi slovníkové metody komprese dat. Je odvozena z metody LZ77 komprese popsané v [6]. Hlavním rozdílem je, že při hledání opakujícího se úseku se nedíváme na celé okno, ale jen na jediné místo, na kterém je jeho výskyt nejpravděpodobnější. To umožňuje snížit náklady na místo pro uložení odkazu na nalezený podřetězec. Zlepšení kompresního poměru se dosáhne, protože zmenšení délek řetězců je dostatečně kompenzováno tím, že není třeba zapisovat přesnou polohu na výstup. Algoritmus je vhodný pro širokou škálu využití na poli datové komprese, jako je archivace distribuce nebo on-line komprese. Algoritmus lze implementovat v mnoha variantách, které mohou dosahovat jak velmi vysokých rychlostí, tak i výkonné komprese.

Algoritmus LZ77 a další varianty si ve výstupním proudu dat rezervují prostor pro umístění pozice zdrojového podřetězce. Koncept LZP ho umožňuje ušetřit a využít ho efektivněji. Pro nalezení vhodného místa je ale třeba použít jinou techniku. Tou je kontextové modelování. Je dána pevná délka kontextu N (order-N kontexty). Ukazatel se vybere podle pozice posledního nalezeného výskytu kontextu. Použitím kontextů vyšších řádů můžeme dosáhnout větší úspěšnosti - častěji budou nalezeny delší úseky symbolů. Porostou ovšem také paměťové nároky na modelování order-n kontextů.

Obecné schéma kompresního procesu algoritmu: Pro každý byte vstupního zdroje dat se předcházejícím N bytům je vytvřen order-N konečný kontext. Tento kontext se použije pro vyhledání v "indexové" tabulce (až už je implementována jakkoli). Výsledek tohoto kroku nám dodá ukazatel někam do proudu už přečtených a zpracovaných dat P, které se nachází před aktuální pozicí. Řetězec znaků S na aktuální pozici se porovná s řetezcem na pozici P. Dostaneme délku shody L. Do "indexové" tabulky se vloží ukazatel na řetězec S na místo, kde byl předtím ukazatel P. Pokud je použito ověřování kontextu je hodnota kontextu přímo uložena také do indexové tabulky na stejné místo. Pokud je nalezená délka shody L nulová (nebo menší než stanovená minimální délka) je zakódována značka "literál" a jednotlivý

znak je pak zapsán na výstup. Pokud je délka L nenulová (aspoň velká jako minimální délka) je kódována značka "match" a číslo L. Protože kódujeme dva druhy entit - délky úseků a hodnoty jednotlivých bytů, je předřazena rozlišující značka, která je umožňuje při dokódování rozlišit. Jak se vyřeší následné kódování jednotlivých entit závisí už na konkrétních implementacích.

1.3 Implementace LZP algoritmu - verze od Charlese Blooma

Charles Bloom navrhl čtyři implementace LZP1 - LZP4, kde LZP1 je nejrychlejší s nejhorší kompresí a LZP 4 poskytuje nejvýkonnější kompresi, ale je nejpomalejší. Verze se liší ve způsobech jakými je naprogramována indexová tabulka a jakým způsobem jsou zapisovány prvky(tj. značky, délky shodujících se řetězců a literály) na výstup.

U LZP1 a LZP2 je indexová tabulka implementována hašovací tabulkou, která používá hašovací funkci:

$H = ((C >> 11)\ XOR\ C)\ \&\ 0xFFF$

V tomto případě je C pevný order-3 kontext. Ta dává 12-bitovou hodnotu, která se používá jako index do pole ukazatelů do už zakódované části souboru. Pokud není hodnota ukazatele P inicializována (tedy nebyl nalezen výskyt podřetězce), je zakódován na výstup literál bez použití značky. Kolize se při hašování neřeší. Verze LZP3 používá obdobnou hašovací funkci, která je však 16 bitová a parametrem je tentokrát order-4 kontext (4 poslední byty). Navíc používá potvrzování kontextu. To znamená, že je v indexové tabulce kromě ukazatele P také uložen samotná hodnota order-4 kontextu, která umožní ověřit, zda je ukazatel platný. Při neúspěchu se nezapisuje literál, ale jsou použity ještě alternativní tabulky pro order-3 kontexty a order-2 kontexty. LZP4 používá nejnáročnější strategii - indexace se provádí pro order-5 kontexty a ve dvou fázích. Při neúspěchu nepoužívá nižší kontexty jako LZP3.

Pro kódování výstupních prvků bylo použito také rozdílných technik. U LZP1 jsou literály kódovány nekomprimované v 8 bitech. Značky pro shodující se řetězce a jejich délky jsou kódovány ve zvláštních kontrolních bytech. Napevno přiřazené sekvence bitů jsou pevně přiřazeny jednotlivým značkám i hodnotám délek. LZP2 pracuje stejně až na to, že se používá sta-

tické Huffmanovo kódování pro literály. U LZP3 je kódování o něco složitější. Značky a délky shodujících se úseků jsou obě kódovány pomocí order-1 Huffmanova kódování. Kontextem je v tomto případě číslo použitého řádu pro LZP kontext (pouze tři možnosti 4, 3 nebo 2). Na tyto tři pole se jednoduše aplikuje statické Huffmanovo kódování. Značky jsou kódovány po čtveřicích. Literály také kóduje Huffmanem, konkrétně ve variantě order-1-0. Dalším rozdílem je, že v případě nalezení shodujícího se řetězce, se následně po něm vynechává značka a předpokládá se, že bude následovat literál (stejně jako u původní LZ77 metody). LZP4 pro dosažení lepší komprese obsahuje již aritmetický kodér a pokročilejší techniky kontextového modelování (context modelling). U každého kontextu má zapsány relativní počty nalezených podřetězců a literálů. Z nich se spočítá kodóvací kontext a použije se pro aritmetické kódování délek shodných podřetězců. Pro literály se kóduje nulová délka. Ty jsou dále transformovány tak, že se vypočítá rozdíl od minulé hodnoty. Pokud je délka menší než minulá hodnota, kóduje se již literál a aktuální hodnota se zmenší na jedničku. Poté se kódují hodnoty s pomocí order-1-0 modelu s tím, že jako order-1 kontext se použije kódovací kontext. Literály se zakódují pomocí order-4-3-2-1-0 PPMC metody a použije se "full exclusion". Znak predikovaný pomocí LZP (tedy ten, který následuje za posledním výskytem) se odstraní z modelu také. Literál se zapisuje automaticky hned po odkazu na minulý řetězec stejně jako u LZP3.

1.4 Předchozí práce - program LZPX

Nyní budu popisovat jednu z dalších implementací LZP algoritmu, ze které jsem vycházel při své práci a kterou jsem se snažil vylepšit hlavně po stránce míry komprese a obecné použitelnosti na různé typy dat. LZPX je kompresor s volně dostupnými zdrojovými kódy jehož autorem je Ilia Muraviev. Pokud porovnáváme tuto implementaci s původními verzemi od Charlese Blooma, je ve své poslední verzi nejpodobnější LZP3. Nicméně první z verzí LZPX jsou zaměřeny více na rychlost a podobají se spíše LZP1 nebo LZP2. Nyní se zaměřím na poslední verzi 1.5b, která tvoří předchůdce programu LZPXj.

LZP algoritmus pracuje s těmito parametry: Bloky jsou délky 16 MB. Úseky se hledejí jen v rámci aktuálního bloku. Indexová tabulka pro hledání LZP kontextu délky 4 znaky je implementována pomocí hašovací tabulky s ověřováním platnosti kontextu. Hašovací tabulka má velikost 8 MB a obsahuje 220 položek po 8 bytech. Čtyři byty jsou použity jako ukazatel do aktuálního bloku a zbývající čtyři jako kontext pro ověření shody. Pokud

není nalezen kontext je použita tabulka, která obsahuje odkazy do datového bloku pro kontexty délky 2 (indexována přímo 2-bytovou hodnotou kontextu). V další fázi jsou kódovány značky, délky nalezených úseků a literály.

Při kódování značek je možno použít tyto tři - úsek, literál nebo konec souboru. Program si zapamatuje poslední čtyři zakódované značky a tato čtveřice je použita jako order-1 kontext aritmetického kodéru. Tato čtveřice značek zřejmě nikdy neobsahuje značku konec souboru a proto máme $2^4 = 16$ možných kontextů pro kódování značek. Po zakódování aritmetickým kodérem je kontext aktualizován pro kódování značky v příštím kroku.

Pokud je nalezen úsek je třeba zakódovat jeho délku. Ta se kóduje pomocí 256 možných hodnot. Pro hodnoty větší než 254 je zakódována speciální hodnota 255 a délka se o 255 sníží. To je opakováno dokud není dosaženo hodnoty pod 255 a tato hodnota je již zakódována přímo. Dále se využije znalosti z předchozí fáze komprese o tom, zda byl index úseku nalezen v hašovací tabulce nebo ne. Každá z obou možností má vlastní rozdělení pravděpodobnosti tj. existuje dvakrát 256 čítačů (odpovídající dvěma kontextům) aritmetického kodéru přiřazené každé z obou možností. Aritmetickým kodérem je pak zakódována hodnota 0 až 255.

Při kódování literálů se jako kontext se bere v úvahu předchozí znak (order-1). Každému ze 256-ti znaků je přiřazen jeden čítač v každém z 256-ti kontextů. Z modelu se navíc na začátku odebere znak, který předpověděla LZP predikce. Čítače se aktualizují přičtením konstatní hodnoty 128. Přesáhne-li celkový součet pro všechny symboly v rámci jednoho bloku čítačů pro jeden kontext určitou hodnotu, všechny čítače jsou v tomto bloku vyděleny dvěma. Tato dynamická aktualizace umožňuje přizpůsobování v případě změny charakteru dat při kompresi nehomogeního souboru.

Update:
```
freq[char] += 128
```
Rescale:
```
if total > 2^16 then { Vyděl všechny freq[i] dvěma }
```

Kapitola 2

Vylepšení LZPXj

2.1 LZPXj - úvod

Implementace LZPX je efektivní hlavně pro textová nebo jiná lineární data uspořádaná jako sekvence bytů. Ale moderní aplikace používají mnoho dalších druhů dat, které je třeba více či méně efektivně komprimovat. Například jsou to multimediální formáty jako obrázky a zvukové soubory. Dále také komprimované formáty dat ať už jde o ztrátovou kompresi nebo bezztrátovou. A v neposlední řadě jsou velmi časté také spustitelné soubory. Mnoho datových souborů obsahuje mnoho z těchto i jiných druhů dat často smíchány dohromady (aplikace, hry , dokumenty).

LZPXj se snaží vylepšit výkon komprese hlavně pro tyto druhy, o kterých se dá předpokládat, že budou často komprimována koncovými uživateli. Cílem je ukázat, jaký výkon může poskytnout metoda založená na LZP na širokém spektru datových souborů.

V LZPXj jsem navrhl a implementoval několik vylepšení, které přinášejí zvýšení výkonu algoritmu. Za cenu zvýšení složitost celého procesu komprese a dekomprese a časových i paměťových nároků se mi podařilo dosáhnout v průměru lepších výsledků. Tato metoda ukázala schopnost konkurovat většině současných komprimačních či archivačních programů a nástrojů. V dalších kapitolách jsou podrobněji popsány a vysvětleny jednotlivé postupy a techniky, které jsem do algoritmu v jeho implementaci LZPXj zařadil.

2.2 Vylepšení výkonu pro špatně komprimovatelná data

Při komprimování souborů obsahující již nějakým způsobem zkomprimovaná data, ať už jde o ztrátovou nebo bezztrátovou kompresi, dochází v LZPX implementaci k expanzi dat. Důvodem tohoto jevu je, že čítače použité při kódování nedávají dostatečně přesnou informaci o pravděpodobnostech symbolů. Tato nedokonalost vede k tomu, že místo toho, aby byla všem symbolům přiřazena stená relativní pravděpodobnost (data vypadají náhodně - hodnoty jednotlivých symbolů jsou na sobě nezávislé), je rozložena nerovnoměrně. V závislosti na míře této nerovnoměrnosti to pak vede ke ztrátám při kompresi. Nepřesné hodnoty jsou pak zasílány do aritmetického kodéru. Ten poté přiřazuje jednotlivým symbolům větší počet bitů než by bylo nutné.

Tuto ztrátu je možné vyčíslit na testovacích souborech obsahující a) pseudonáhodná data vygenerovaná nějakým náhodným generátorem; b) komprimovaná data (obrázky JPG, soubory ZIP apod.). V následující tabulce 2.1 je vidět v procentech o kolik je výkon komprese horší. Pro porovnání je uvedeny původní hodnoty a hodnoty po aplikaci mnou navržené metody, která je implementována ve verzi programu LZPXj 1.0e. Tato metoda je zařazena také ve všech dalších verzích s tím, že k ní jsou přidány další způsoby vylepšení, které jsou popsány v této práci v dalších kapitolách.

Soubor	**JPG**		**ZIP**	
Původní velikost	449638	100,0%	861435	100,0%
Po „kompresi“ LZPX 1.5b	486144	108,1%	931215	108,1%
LZPXj 1.0e	441887	98,3%	860491	99,9%

Tabulka 2.1: Porovnání komprese LZPX 1.5b a LZPXj 1.0e na souborech JPG a ZIP.

První nápad, který mnoho lidí asi napadne je průběžně zjišťovat jestli nejsou data expandována. V případě, že je to detekováno pozastavit aritmetické kódování a místo toho jen kopírovat na výstup nekomprimovaná data resp. symboly či literály z předchozí LZP kompresní fáze. To je ale také ekvivalentní tomu, když nahradíme hodnoty všech čítačů tak, aby měly stejnou hodnotu. Tato stejná hodnota přiřazená všem čítačům zajistí, že aritmetické kódování v další fázi bude používat pro svou práci rovnoměrné rozdělení pravděpodobnosti, kde každý symbol bude kódován stejným počtem bitů. V

případě, že symboly jsou jednotlivé byty souboru, bude to přesně 8 bitů.

Nabízí se otázka, zda by nebylo možné a z hlediska dosažení vyšší komprese výhodnější použít nějaký mezistupeň. Pod tím si představme, že pravděpodobnosti odpovídající jednotlivým čítačům nebudou nahrazeny jedinou hodnotou, ale jejich velikosti se vyrovnají jen o něco méně. To znamená, že vypočítané rozdělení pravděpodobnosti pomocí systému čítačů se nenahradí rovnoměrným, ale provede se kombinace původního a rovnoměrného rozdělení v nějakém vhodném poměru.

Aby se toho dosáhlo v programu LZPXj, je ke všem čítačům přičtena hodnota H, která má záviset na jakési odhadnuté míře náhodnosti dat. Jak je počítána hodnota H je popsáno podrobněji níže. Obecně se dá říci, že pokud se data daří komprimovat dobře, je H snižována, a naopak, pokud by byl výstup větší než původní data, je zvyšována.

K výpočtu H jsem použil jednoduchou heuristiku. Principem práce je snaha zjistit P - podíl celkové frekvence všech symbolů ku frekvenci konkrétního čítače pro jediný právě kódovaný symbol. Čím je tento podíl vyšší, tím je výstup z následného kódovaní pomocí aritmetického kodéru větší. Dále máme proměnnou C, jejíž hodnota nějakým způsobem zohledňuje, jak dobře se daří data komprimovat. C je snižována pokud podíl P klesne pod dolní mez udanou konstatou K1. Naopak podku P přeroste horní mez K3 je hodnota proměnné C zvyšována. Konkrétně pokud nastává $P < K1$ je C vynásobeno konstantou K3, která je menší než jedna, pro $P > K3$ se provede přičtení konstanty K4 k proměnné C. V ostatních případech je-li $P >= K1$ a zároveň $P <= K3$ přičte se k proměnné C konstanta K5, jejíž hodnota je menší než K4.

Aktualizace proměnné C:

```
P = total / freq[char]
if P < K1 then C = C * K2
if P > K3 then C = C + K4
if (P >= K1 AND P <= K3) then C = C + K5
```

Určení konstant K1 až K5 jsem provedl pouze přibližně ($K1 = 185$, $K2 = 9/10$, $K3 = 2048$, $K4 = 20$, $K5 = 9$). Experimenty ukázaly, že tyto hodnoty dobře postačují pro účely tohoto postupu. Také platí, že na jejich přesných velikostech kriticky nezáleží, protože hodnota C je dále transformována na H a tato transformace je z hlediska dosažení větší přesnosti důležitější. Kromě proměnné C je počítána také C‘ - a to stejným způsobem jako C, jen pro každý jednoznakový kontext (tj. poslední kódovaný znak) zvlášť. Ukázalo

se, že součet C + C‘ dobře vystihuje chování různých druhů dat.

Klíčem k dosažení optimálního působení je však dostatečně přesně určit závislost proměnné H a součtu C + C‘. Provedl jsem proto řadu měření. Interval možných hodnot (C + C‘) jsem si rozdělil na několik úseků a měřil jsem výsledek komprese pro různá H. Nakonec jsem do tabulky poznamenal ten nejlepší zjištěný výsledek. Testovací data obsahovala jak textová, binární i pseudonáhodná data. Vynesené hodnoty jsou vidět na grafu 2.1 jako vodorovné čáry. Výsledkem je zjištění, jak má proměnná H záviset na proměnné C - body grafu můžeme proložit parabolu a určit i přesné koeficienty kvadratické závislost. Je možné použít i exponencielní závislost, ta se však ukázala jako přibližně stejně vhodná, přičemž z hlediska rychlosti a jednoduchosti dopadne lépe kvadratická závislost - pro výpočet stačí jediné násobení.

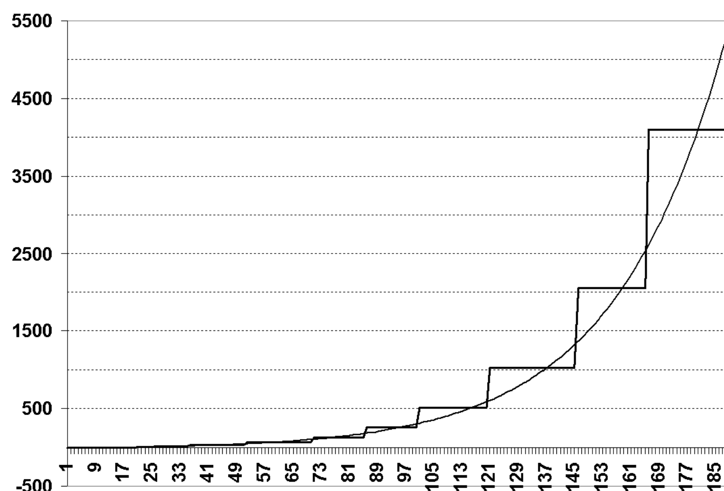
Určení hodnoty H:

```
Z = ( C + C' )
if Z > 1024 then Z = 1024
H = ( Z * Z / 48 * total ) >> 18
```

2.3 Komprese pro 16 / 24 / 32 bitová data

Mnoho souborů obsahuje struktury a data zarovnané do skupin. Významovou jednotku tam pak netvoří jediný byte, ale přímo skupiny více než jednoho bytu. To je v případě pokud jsou ukládána 16-bitová čísla nebo 32-bitová čísla poměrně častá v binárních souborech nebo také v multimediální formáty. Například 24-bitové RGB obrázky obsahují 24-bitové zarovnání záznamů. Dále nekomprimová 16-bitová audio data ve formátu WAV jsou zarovnány po 32 bitech (stereo - 2 kanály) nebo 16 bitech (mono - jediný kanál). Ve velkém počtu binárních formátech je možné takovéto struktury záznámů nalézt a spustitelné soubory nejsou výjimkou.

Jeden z možných postupů jak této vlastnosti datových souborů využít pro vylepšení komprese je předzpracovat vstup pomocí filtrů. Tento způsob se snaží buď vhodně přeuspořádat data nebo v případě delta filtrů jsou místo původních záznamů kódovány rozdíly mezi aktuální a předchozí hodnotou. Tato fáze je pak plně předřazena kompresnímu algoritmu a nijak jej není třeba modifikovat. Nevýhodou je v tomoto případě nutnost složitě detekovat, kdy a jaké transformace je možné provádět. Dalším záporem může být menší obecnost tohoto postupu, protože se dá požít jen na určitou malou



Obrázek 2.1: Proložení naměřené závislosti proměnných H a C křivkou.

množinu formátů (ty které se dají spolehlivě detekovat a vhodně transformovat). V LZPXj jsem zvolil jinou cestu, tou je provedení změn přímo uvnitř kompresního algoritmu. Takové změny, které vedou k vylepšení komprese těchto struktur, spočívají v tomto případě ve změně způsobu kódování literálů.

V algoritmus LZP jsem nahradil jednoduché kontextové modelování, které pro kódování literálů používá jako kontext jeden poslední symbol ze zakódovaných dat. Pokud bereme jako kontext jediný symbol je lepší při zacházení s 16/24/32-bitovými záznamy použít jako kontext znak na odpovídající pozici v předešlém záznamu. Odpovídá to tomu, že se odkazujeme na druhý, třetí nebo čtvrtý znak od konce. Snahou je, aby bylo možné toto nahrazení provést bez újmy na výkonu pro běžné textové soubory, ve kterých je samozřejmě nejvhodnější volbou použití posledního symbolu. Konkrétně v metodě LZPXj je modelování použito více-násobně, konkrétně

čtyřnásobně. Pro každý znak jsou k dispozici čtyři sady čítačů, které odpovídají pravděpodobnostním rozdložením vhodným pro jednotlivé druhy dat. V každém kroku se vybírá právě jeden kontext z těchto čtyř, který se jeví jako nejvhodnější. Model se dynamicky přizpůsobuje přicházejícím datům a dokáže si vybírat správně ten nejvhodnější. Vyběr se provádí podle odhadnutého výsledku na nedávných datech. Ze čtyř možností se bere nejlepší výsledek.

Ke každému z kontextů je přiřazena proměnná, která určuje jí příslušný počet bodů B. Tento počet udává relativně kvalitu tohoto kontextu - méně bodů znamená lepší možnou kompresi. Pro kompresi je vybírán ten s nejnižším bodovým kontem. Aktualizace se provádí na základě hodnoty P poměru čítače pro kódovaný symbol ku celkovému součtu čítačů v daném kontextu. Pokud je tato P = hodnota total / freq[char] větší než v některém z ostatních kontextů dostane za každý z nich bod a naopak druhá strana jeden bod ztrácí. To přesně znamená, že jsou prověřeny všechny dvojice (je jich šest). Tomu z dvojice, který má menší P je snížen bodový součet B o jedna. Tomu z dvojice, který má součet P větší je bodový součet naopak zvýšen. Dynamické přizpůsobení je zajištěno občasným vydělením všech B dvěma.

Výběr a aktualizace:

```
P1 = total1 / freq1[char]
P2 = total2 / freq2[char]
P3 = total3 / freq3[char]
P4 = total4 / freq4[char]
if (P1 > P2) then B1++, B2-- else B1--, B2++
if (P1 > P3) then B1++, B3-- else B1--, B3++
if (P1 > P4) then B1++, B4-- else B1--, B4++
if (P2 > P3) then B2++, B3-- else B2--, B3++
if (P2 > P4) then B2++, B4-- else B2--, B4++
if (P3 > P4) then B3++, B4-- else B3--, B4++
if (T++ >= 256) then T=0, B1/=2, B2/=2, B3/=2, B4/=2
Vyber kontext s nejnižší hodnotou B
```

2.4 Filtr pro instrukční kód x86

K dosažení lepší komprese na spustitelných souborech je třeba předřadit vlastnímu kompresnímu algoritmu minimálně filtr pro CALL a JMP in-

strukce. Ty jsou v instrukčním kódu pro běžné procesory řady x86 velice časté a znamenají instrukci skoku na určenou adresu. Adresa je uložna relativně k pozici dané insturkce. To způsobuje, že všechny skoky na jediné místo v souboru jsou kódovány každý jiným číslem a to vede k horšímu kompresnímu poměru. Toto čtyřbajtové číslo znamenající relativní adresu skoku by proto bylo výhodné nahradit absolutní adresou nebo jednoznačným záznamem určujícím jediné místo v souboru. Smyslem jednoduchého filtrování je jednoduché nahrazení relativní adresy za absolutní. Cílem je aby se stejné absolutní adresy mnohokrát opakovaly.

CALL / JMP instrukce jsou v x86 kódu zapsány byty 0xe8h / 0xe9h. Hledáním těchto hodnot můžeme nalézt všechna místa, kde se má transformace uskutečnit. Nicméně najdeme také spoustu dalších výskytů, která nekódují přímo číslo instrukce, ale například se nacházejí jinde v kódu jako operandy instrukcí nebo jako jiná data napevno uložená v souboru apod. Dále o datech předem nemůžeme s jistotou vědět zda opravdu obsahují instrukční kód a tedy jestli se nejedená o jiná binární či textová data, kde by pokus o transformování bytů způsobil jistě zhoršení komprese. Proto je třeba s dostatečnou spolehlivostí detekovat, kdy jde o spustitelný kód a kdy jde v tomto kódu o instrukci a má tedy dojít k nahrazování.

U relativní adresy je nutno analyzovat, do kterého místa ukazuje. Pokud jde o skutečnou CALL / JMP instrukci je cíl umístěn v naprosté většině případů poměrně blízko výskytu instrukce. Toho se využívá pro zjištění falešných výskytů bytů 0xe8h a 0xe9h. Konkrétně stačí testovat hodnotu nevyššího byte z adresy, není-li to 0 nebo 255, pak jde s největší pravděpodobností o falešný výskyt. V opačném případě si tak jistí být ale nemůžeme, protože například pro náhodná data by došlo ke zbytečné transformaci průměrně v každém z 128 případů (testována hodnota jediného bytu). Aby se tomu předešlo LZPXj používá jednoduchou heuristiku pro detekci instrukčního kódu.

Transformace:

```
if C == 0 then { nedělej nic, nebyl detekován instukční kód }
else { if (char[pos] == 0xe8h OR char[pos] == 0xe9h)
AND (char[pos+4] == 255 || char[pos+4] == 0)) then {
   X = (char[pos + 1], char[pos + 2], char[pos + 3])
   X = X + pos + block_offset
   (char[pos + 1], char[pos + 2], char[pos + 3]) = X
} }
```

Detekce sleduje opakující se znaky v instrukčním kódu. Znaky jsou rozděleny do dvou kategorií. Jednou z nich jsou znaky 0x83h a 0xe8h, které se ve spustitelném kódu vyskytují velmi pravidelně. Druhou část tvoří další vybrané často se vyskytující znaky, které už nemají při detekci tak velkou váhu. Při detekci jsou jako čítače použity dvě proměnné A0 a A1, které udávají jak často se v nedávných datech objevil znak 0x83h resp. 0xe8h. Vyšší hodnota znamená menší četnost. A0 či A1 jsou inkrementovány pouze v případě znaku, který nepadne ani do jedné z kategorií (tj. znaků, které se v EXE kódu běžně nevyskytují). Pokud součet A0 + A1 přesáhne danou konstantu 256 je čítač C snížen o jedna. Přesáhne-li hodnotu 2^{14} sníží se na nulu. jinak se nastaví na 2^{14}. Případ C = 0 znamená negativní detekci a transformace není prováděna.

Detekce:

```
switch (char[pos]) {
   case 0x83h: A0 = A0 * 3 / 5
   case 0xe8h: A1 = A1 * 3 / 5
   case 137, 139, 8, 192, 4, 133, 141:
   case 0, 1, 69, 80, 116, 117, 255: nedělej nic
   default:
   if A0 < 2^14 then A0++
   if A1 < 2^14 then A1++
}
if A0+A1 < 256 then C = 2^14
else if A0 + A1 > 2^14 then C = 0
else if C > 0 then C--
```

Kapitola 3

Kódování v LZPXj

3.1 Kódování - 2 modely

Poslední verze LZPXj 1.2h obsahuje dva modely pro kódovýní výstupu z LZP fáze. LZP produkuje tři druhy entit - literály, délky opakujících se řetězců a rozlišující značky. Ty je třeba efektivně zakódovat. Ke zvýšení efektivity jsem použil dva modely, z nichž každý se více hodí pro jiný druh dat. Model 1 je verze PPMC algoritmu, která je vhodná pro textová a lineární data. Model 2 naopak implementuje vylepšení popsané v kapitole Komprese pro 16 / 24 / 32 bitová data a Vylepšení výkonu pro špatně komprimovatelná data, takže je vhodná pro binární, bloková nebo předem komprimovaná data. Z tohoto důvodu jsou v činnosti oba modely a podle průběžně aktualizovaného ukazatele účinnosti je vybírán ten lepší a vhodnější model, který se v ten daný okamžik použije.

3.2 Model 1 - PPM

První model je jednoduchým order 3-1-0 PPM. PPM je algoritmus často používaný a velmi efektivní pro kompresi textů. Jedna z možných implementací je popsána také [3]. Já jsem zvolil implementaci pomocí hašovací tabulky pro všechny order-3 kontexty. Hašovací funkce H1 použije hodnotu posledních tří bytů (X) ke konstrukci indexu v hašovací tabulce. Při kolizi, tedy pokud je dané místo v hašovací tabulce obsazeno informacemi o jiném kontextu, se použije náhradní část tabulky určená pro order-2 kontexty. Jiný rozsah indexů je pro ně vyhrazen. V tomto případě je pak použito

sekundární hašovací funkce H2. Jako parametr dostává hodnotu posledních dvou bytů (Y). Hodnota mem nastavení programu, které udává množství použité paměti. Je to číslo v rozsahu 1 až 9.

Hodnota získaná jednou z těchto dvou funkcí odpovídá indexu nejvyššího řádu pro následné PPM. Dalším indexem prvního řádu je přímo hodnota posledního byte (order-1 kontext) v rozsahu 1..256. Pro order-0 kontext je použit index s hodnotou 0. V následném kódování se pak sestupuje zeshora dolů a použije se nejdelší kontext, ve kterém je symbol nalezen. Pokud není v některém nalezen je zakódován ESC symbol.

Hašovací funkce:

$H_1(X) = 257 + ((X/3737 + X*3737 + X/523 + X) \,\&\, (2^{10+mem} - 1))$

$H_2(Y) = 257 + ((Y*11 + Y/11 + Y) \,\&\, (2^{10+mem} - 1)) + 2^{10+mem}$

Znaky, které se kódují v tomto modelu jsou vhodně přetransformovány. Používá se abeceda s 504 znaky. Znak s číslem 0 je použit jako ESC symbol. Znaky s čísly 1 až 256 kódují literály. Znaky 257 až 503 kódují délky úseků. Není třeba kódovat značky, protože ty jsou tímto systémem zakódovány implicitně. Rozlišení literálů a řetězců je totiž už provedeno. To dohromady znamená, že v hašovací tabulce je uloženo pro každý index 504 čítačů. V tomto případě jsou jednobytové, aby se ušetřila paměť. Celkový počet indexů je $2^{(11+mem)}$, tedy celkem zabere model $2^{(20+mem)}$ bytů.

3.3 Model 2

Tento model využívá techniky popsané v předcházejících kapitolách. To znamená, že se používá komprese pro 16/24/32 bitová data. Zde se pravděpodobnosti pro aritmetické kódování zjišťují v závislosti na jediném znaku - ten předchozí, druhý, třetí nebo čtrvtý znak od konce dosud zakódovaných dat. Rozdělení pravděpodobnosti je udržováno tak, že je každému symbolu použité 256-ti znakové abecedy přiřazen jeden čítač. Jeho velikost relativně určuje, jak je daný znak pravděpodobný. Následně se použije postup popsaný v kapitole Vylepšení výkonu pro špatně komprimovatelná data. Výsledné pravděpodobnostní rozdělení je poté upraveno přičtením hodnoty H příslušným způsobem vypočítané, tak aby se co nejvíce zlepšila komprese pro špatně kompimovatelné úseky dat. Nevýhodou tohoto modelu je, že používá pouze velmi krátké jednoznakové kontexty a proto dosahuje poměrně špatných výsledků na textových datech. Tuto situaci lépe řeší Model1 pomocí jednoduché implementace PPM algoritmu.

3.4 Výběr modelu

Po celou dobu komprese jsou v činnosti oba modely zároveň. Jsou aktualizovány příslušné čítače v obou modelech. Protože pro různá data je vhodný různý model, je třeba rozhodnout, kdy je lépe použít, který z nich. K tomu, abychom dostali výslednou predikci pro následující symbol, je vybrán jeden z modelů podle heuristiky, která počítá cenu na zakódování symbolu. Určí se pro poslední už zakódovaný úsek, který z modelá je na tomto úseku výhodnější co se týče délky generovaného výstupu.

Udržováno je 257 dvojic čítačů O1 a O2. Z nich 256 je používáno podle toho, jaký byl poslední zakódovaný znak (O1[0..255], O2[0..255]) a jedna dvojice je globální bez ohledu na poslední znak (O1[256], O2[256]). Při kódování i dekódování se nejprve podle hodnot O1[ch], O2[ch], O1[256] a O2[256] určí lepší z modelů. Následně se model použije pro kódování resp. dekódování aktuálního znaku. Nakonec se čítače aktualizují, tak aby se v následujících datech dosáhlo co nejlepšího výsledku. Při aktualizaci se přitom spočítá kolik bitů by zabralo kódování pomocí modelu1 a výsledek se přičte k proměnným O1[ch] a O1[256]. Aby se preferoval význam proměnné O1[ch] oproti O1[256] přičítá se k ní dvojnásobek. Stejným způsobem se aktualizují O2. Dynamická povaha rozhodování se zachovává tím, že se čítače vynásobí 3 / 4, pokud součet O1[x] + O2[x] přesáhne určitou konstantu (konkrétně 1024).

Rozhodovací kritérium:

```
if O1[ch] + O1[256] > O2[ch] + O2[256] then { použij model1 }
else { použij model2 }
```

Aktualizace čítačů:

```
for (x = ch, x = 256) {
   if O1[x] + O2[x] > 1024 then {
      O1[x] = O1[x] * 3 / 4, O2[x] = O2[x] * 3 / 4
} }
O1[ch] = O1[ch] + cost1 * 2, O2[ch] = O2[ch] + cost2 * 2
O1[256] = O1[256] + cost1, O2[256] = O2[256] + cost2
```

Kapitola 4

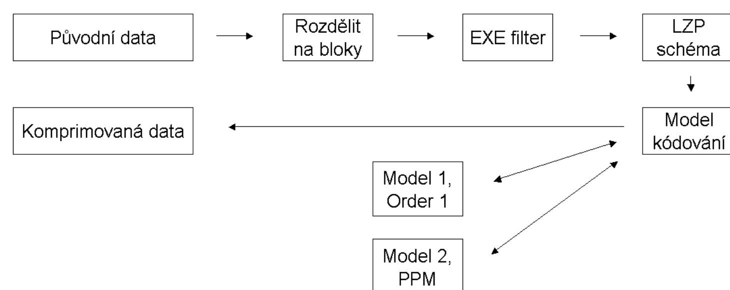
Shrnutí a závěr

4.1 Celkový postup komprese

Poslední verze LZPXj 1.2h obsahuje fitr pro spustitelné soubory jako preprocesor. Dále používá LZP schéma odpovídající přibližně tomu jak je využito v LZPX. To co se nejvíce liší a také nejvíce ovlivňuje výkon komprese je způsob kódování literálů a také možnost využít více paměti pro většinu komponent využitých v algoritmu. Kódování literálů je řešeno pomocí dvou modelů. Schéma celkového průběhu komprese i dekomprese je vidět na obrázcích 4.1 a 4.2

Záverečnou fázi tvoří aritmetické kódování. Tuto fázi jsem neupravoval a zůstává tedy stejná jako v LZPX. Použité aritmetické kódování je verze popsaná Schindlerem v schindler, která se názývá range coder. Její vlastností je, že přeškálování aktivního rozsahu se provádí po celých bytech, nikoliv po jednotlivých bitech.

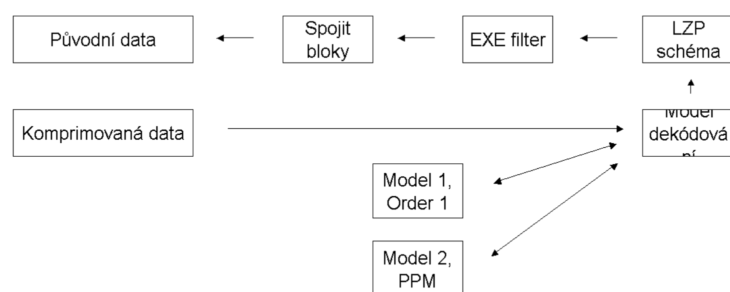
Implementace LZP fáze algoritmu, se liší od té v LZPX poměrně málo. Hlavním rozdílem je, že umožňuje nastavení množství použité paměti pro jednotlivé komponenty kompresního procesu. Proto je možné použít větší velikost bloku nebo velikost hašovacích tabulek. Paměťové nastavení ovlivňuje také velikost hašovací tabulky použité pro PPM model. Dalším menším rozdílem je možnost použití kromě dvou a čtyřznakových LZP kontextů také osmiznakový. Kódovací fáze se liší velmi významně - dva použité modely umožňují vylepšit kompresi pro nejrůznější datové typy.



Obrázek 4.1: Kódování v LZPXj 1.2h.

4.2 Srovnání komprese

Abych otestoval, jak jednotlivé části programu ovlivňují výkon po stránce kompresního poměru, provedl jsem srovnání komprese při vypnutí některých částí kompresoru nebo některých jejich vylepšení. V tomto případě jsem použil tři testovací soubory. Prvním z nich byl spustitelný soubor WINWORD.EXE z kancelářského balíku Microsoft Office. Zastupuje binární soubory a také obsahuje x86 instrukční kód. Zde můžeme vidět, kolik ušetří transformace relativních adres. Druhý soubor je archiv, který obsahuje několik souborů ve formátu html. Je to několik html souborů obsahující nápovědu k programu Visual Studio. Dohromady jsou sloučeny do jednoho TAR archivu. Třetím souborem jsem jako zástupce multimediálních dat zvolil BMP soubor ve běžném 24-bitovém RGB formátu. Obsahuje fotografii z digitálního fotoaparátu zmenšenou na menší rozlišení a převedenou do formátu BMP.



Obrázek 4.2: Dekódování v LZPXj 1.2h.

Výsledky jsou prezentovány v tabulce 4.1. Relativní změny, tedy kompresní ztráty v procentech, pro jednotlivé soubory jsou vidět na grafu 4.3 spolu s průměrným výsledkem.

Můžeme si všimnout, že model2 a komprese pro 16/24/32 bitová data, která je v něm obsažena má velký vliv při kompresi obrázku BMP. Zlepšení se pohybuje kolem 9%. Naopak model1 s PPM je výhradně použit pro kompresi textových dat a u souboru HTML.TAR se neobjevuje žádná ztráta při vypnutí součásti model2. Soubor WINWORD.EXE, což je spustitelná aplikace, je dobře komprimován kombinací obou modelů. Přitom při použití, každého z nich samostatně přináší ztrátu (2% resp. 5,5%). Filtr pro spustitelné soubory dokázal ušetřit na EXE souboru kolem 3,5%. Zajímavé je, že komprese pro 16/24/32 bitová data zlepšuje kompresi i u TAR archivu s textovými html soubory. V průměru se všechna vylepšení ukázala jako opodstatněná. Pouze optimalizace délek řetězců se na těchto testovacích datech

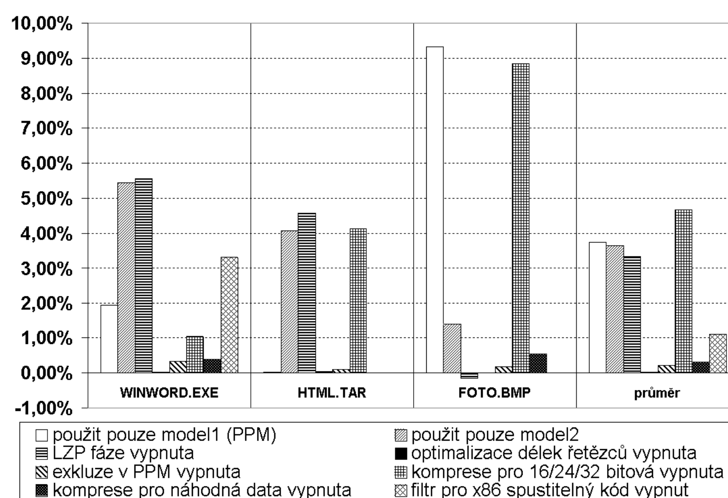
Testovaný soubor / nastavení	EXE	TAR	BMP
LZPXj 1.2h (vše zapnuto)	40,40% (+0,00%)	16,77% (+0,00%)	54,75% (+0,00%)
použit pouze model1 (PPM)	42,34% (+1,93%)	16,80% (+0,02%)	64,07% (+9,32%)
použit pouze model2	45,84% (+5,44%)	20,86% (+4,08%)	56,13% (+1,39%)
LZP fáze vypnuta	45,96% (+5,56%)	21,35% (+4,58%)	54,59% (−0,16%)
optimalizace délek řetězců vypnuta	40,43% (+0,03%)	16,81% (+0,04%)	54,75% (+0,00%)
exkluze v PPM vypnuta	40,74% (+0,34%)	16,87% (+0,10%)	54,93% (+0,18%)
komprese pro 16/24/32-bitová vypnuta	41,45% (+1,05%)	20,90% (+4,13%)	63,59% (+8,84%)
komprese pro náhodná data vypnuta	40,80% (+0,40%)	16,78% (+0,00%)	55,29% (+0,55%)
filtr pro x86 spustitelný kód vypnut	43,71% (+3,31%)	16,77% (+0,00%)	54,75% (+0,00%)

Tabulka 4.1: Výsledky srovnání komprese při vypnutí některých částí kompresoru.

neprojevila. Došlo k tomu, protože tato optimalizace je vhodná pro delší texty obsahující mnoho dlouhých opakujících se úseků a také její vliv na kompresi není tak velký jako například vypnutí celého jednoho modelu.

Program LZPXj umožňuje pracovat v devíti režimech, které se liší množstvím použité paměti. Množství paměti je možné nastavit volbou od 1 do 9. Standardním nastavením je 6, při kterém se používá kolem 163 MB. Nejnižší možnou hodnotou je 7 MB a nejvyšší 1315 MB. Každým zvýšením o jedničku je možné množství použité paměť přibližně zdvojnásobit a naopak. Na grafu 4.4 můžeme vidět, jak se mění kompresní poměr na Calgary Corpus pro nastavení 1 až 7. Protože tata standardní testovací data mají poměrně malou velikost (kolem 3 MB), větší paměť něž 40 MB už nepomáhá k lepší kompresi. Na řádově větších datech se dá očekávat zlepšování komprese i při vyšších nastaveních.

Na internetu existuje několik testů, které se zabývají testováním komprimačních nástrojů. Tou nejdůležitější je stránka Maximumcompression.com [2], která se zaměřuje hlavně na srovnání kompresního poměru. Jejím auto-



Obrázek 4.3: Relativní zhoršení komprese při vypnutí některých částí kompresoru.

rem je Werner Bergmans. Obsahuje deset testů, které reprezentují soubory několika široce rozšířených formátů. Těmito formáty jsou bmp, doc, dic, dll, exe, hlp, jpg, log, pdf, txt. Celková velikost všech souborů je něco přes 50 MB.

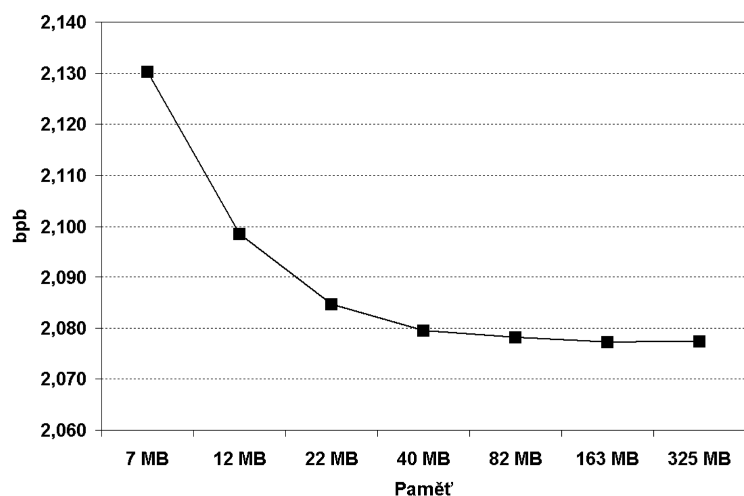
Rozsáhlejším testem je Squeeze Chart (autor Stephan Busch). Velikost testovacích dat přesahuje 3 GB. Testovací data jsou rozdělena do několika skupin. Obsahují mnoho souborů v mnoha různých formátech. Pro srovnáni jsem vybral několik známějších nástrojů. Těmi jsou WinZIP, PKZIP, WinRAR, 7-ZIP, GZIP, BZIP2. Jejich výsledky v obou testech jsou uvedeny v tabulce 4.3.

Soubor	**Velikost**	**LZPXj** 162MB	**LZPXj** 22MB	**LZPXj** 7MB	**LZPX 1.5b** 25MB	**LZP4** 2MB
	Bytů	bpb	bpb	bpb	bpb	bpb
bib	111261	1,971	1,974	1,995	2,336	1,915
book1	768771	2,469	2,482	2,562	3,075	2,350
book2	610856	2,091	2,101	2,157	2,580	2,011
geo	102400	4,050	4,049	4,044	5,092	4,740
news	377109	2,390	2,399	2,466	2,758	2,350
obj1	21504	3,721	3,723	3,728	4,343	3,744
obj2	246814	2,328	2,337	2,396	2,711	2,388
paper1	53161	2,449	2,452	2,468	2,789	2,378
paper2	82199	2,442	2,445	2,471	2,826	2,389
pic	513216	0,804	0,804	0,803	0,797	0,814
progc	39611	2,456	2,456	2,464	2,765	2,392
progl	71646	1,601	1,603	1,608	1,782	1,589
progp	49379	1,622	1,620	1,620	1,843	1,585
trans	93695	1,347	1,347	1,365	1,558	1,343
average	**224402**	**2,267**	**2,271**	**2,296**	**2,661**	**2,291**

Tabulka 4.2: Kompresní poměr pro LZPX 1.5b, LZPXj 1.2h a LZP4 naměřený na Calgary Corpus.

4.3 Závěr

V implementaci LZP algoritmu LZPXj, jsem navrhl v zásadě 4 hlavní vylepšení oproti LZPX. Těmi jsou a) efektivnější zacházení s předem komprimovanými daty (zip, jpg), b) efektivnější komprese pro data ukládaná v 16, 24 nebo 32-bitových záznamech (wav, bmp), c) filtrování a detekce spustitelného kódu (exe), d) 2 modely - jeden vhodnější pro textová data (txt, html) a druhý pro binární data - s možností automatického výběru. Další změnou je možnost nastavení použité paměti pro kompresi. Při porovnání s ostatními metodami jsem dosáhl v průměru dobrého kompresnímu poměru. Co se týče rychlosti je o něco pomalejší něž běžně používané metody, hlavně díky tomu, že celou dobu byla primární snahou optimalizace na lepší kompresní poměr. Práce tak navázala na existující implementace hlavně ve způsobech kódování výstupu LZP algoritmu, jehož výkon je, jak se ukázalo, velmi důležitý.



Obrázek 4.4: Závislost použité paměti a celkového kompresního poměru naměřená na Calgary Corpus.

Test Datum a velikost	**Maximumcompression** 21.5.2007 53134726			**Squeeze Chart** 21.5.2007 3569857215		
Název programu	Výsledek	Procent	Pozice	Výsledek	Procent	Pozice
LZPX 1.5b	14183310	26,69%	110	1565914282	43,86%	175
LZPXj 1.2h	**11977759**	**22,54%**	**35**	**1219059043**	**34,15%**	**56**
WinZIP	12296985	23,14%	45	1525647892	42,74%	160
PKZIP	14868451	27,98%	126	2048935998	57,40%	259
WinRAR	11600943	21,83%	25	1401921481	39,27%	96
7-ZIP	11290176	21,25%	19	1183548458	33,15%	39
GZIP	14948376	28,13%	134	1745506325	48,90%	208
BZIP2	13379118	25,18%	90	1604765959	44,95%	180

Tabulka 4.3: Srovnání některých výsledků testů na internetu.

Literatura

[1] C. Bloom. *LZP — a new data compression algorithm*, Proceedings of the IEEE Data Compression Conference (DCC'96), 1996.

[2] W. Bergmans. *Lossless data compression software benchmarks / comparisons*, http://www.maximumcompression.com/, 2007.

[3] A. Moffat. *Implementing the PPM data compression scheme*, Proceedings of the IEEE Transactions on Communications, 38 (11), pp. 1917–1921, 1990.

[4] M. Schindler. *Range Encoder version 1.3*, http://www.compressconsult.com/rangecoder/, 2000.

[5] I. H. Witten, R. M. Neal, and J. G. Cleary. *Arithmetic coding for data compression*, Communications of the ACM, 30(6):520–540, 1987.

[6] J. Ziv and A. Lempel. *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory 23 (3), pp. 337–342, 1977.