VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

BEZEZTRÁTOVÁ KOMPRESE DAT V IP SÍTÍCH

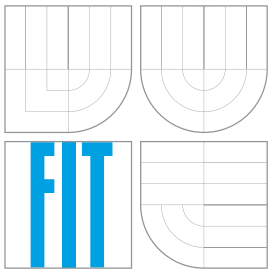
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RICHARD PÁNEK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

BEZEZTRÁTOVÁ KOMPRESE DAT V IP SÍTÍCH

LOSSLESS DATA COMPRESSION FOR IP NETWORKS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RICHARD PÁNEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PAVOL KORČEK

BRNO 2013

Abstrakt

Tato bakalářská práce shrnuje metody použitelné při komprimací dat v IP sítích. Popisuje vývoj a funkcionalitu kompresního algoritmu LZW. Dále popisuje testování komprimace paketů IP sítě pomoci algoritmu LZW, kde bylo dosaženo snížení velikosti na 60 % až 70 % původní velikosti u páteřního provozu. Je zde uveden postup při vysokoúrovňové syntéze z jazyka C do jazyka VHDL algoritmu LZW.

Abstract

This bachelor's thesis deals with data compression methods in IP networks. The LZW compression algorithm and its history is described more in detail. This algorithm is tested on the different types of IP traffic. It is shown that depending on the traffic type it is possible to reduce data to 70 % of its original size. As the final implementation of the LZW algorithm is intented for use in the FPGA the results from high level synthesis (from C to VHDL language) are finally described.

Klíčová slova

Komprese dat, IP pakety, LZW, vysokoúrovňová syntéza

Keywords

Data compression, IP packets, LZW, high level synthesis

Citace

Richard Pánek: Bezeztrátová komprese dat v IP sítích, bakalářská práce, Brno, FIT VUT v Brně, 2013

Bezeztrátová komprese dat v IP sítích

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana inženýra Pavola Korčeka a uvedl jsem všechny použité zdroje informací.

.....
Richard Pánek
15. května 2013

Poděkování

Děkuji vedoucímu mojí práce panu inženýrovi Pavolu Korčekovi za odborné vedení mojí práce a zajímavé podněty, které mi poskytl k řešení.

Obsah

1 Úvod **3**

2 Komprese dat **4**

3 IP Pakety **5**
3.1 IPv4 . . . 5
3.2 IPv6 . . . 6
3.3 TCP a UDP . . . 7

4 Komprese IP paketů **9**
4.1 Komprimace celých paketů . . . 9
4.2 Komprimace hlaviček (Header Compression) . . . 9
4.2.1 Van Jacobson Header Compression (VJHC) . . . 10
4.2.2 IP Header Compression (IPHC) . . . 10
4.2.3 RObust Header Compression (ROHC) . . . 11
4.3 Shrnutí uvedených komprimačních algoritmů . . . 12

5 LZW algoritmus **13**
5.1 Vývoj LZW algoritmu . . . 13
5.1.1 LZ77 . . . 13
5.1.2 LZ78 . . . 15
5.1.3 LZW . . . 17
5.2 Použitá implementace . . . 18
5.3 Testování LZW komprese na síťovém provoze . . . 18
5.3.1 Popis jednotlivých testovacích sad . . . 18
5.3.2 Výsledky testů . . . 19

6 Imlementace pro FPGA **22**
6.1 O programu Vivado HLS . . . 22
6.2 Implementace pomoci Vivado HLS . . . 23
6.2.1 Implementace pro Spartan3 . . . 23
6.2.2 Optimalizace . . . 23
6.2.3 Implementace pro Virtex5 . . . 24
6.2.4 Implementace pro Virtex7 . . . 24

7 Závěr **25**

Seznam obrázků

3.1 IPv4 paket. 6
3.2 IPv6 paket. 7
3.3 TCP paket. 8

4.1 Operační módy ROHC. 11

5.1 Pohyblivé okno LZ77. 14
5.2 Pohyblivé okno LZ77 po posunu. 14
5.3 Pro odvození velikosti pole délka shody značky LZ77. 14
5.4 Slovník LZ78. 16
5.5 Graf zobrazující kompresní poměr. 20
5.6 Graf srovnání velikosti dat z koncového zařízení před a po kompresi. 21
5.7 Graf srovnání velikosti dat z páteřní sítě před a po kompresi. 21

Kapitola 1

Úvod

Tato práce se zabývá komprimací dat pocházejících z IP sítí. Důvodem této práce je zrychlující se přenos informací po sítích a zvětšující se objem přenášených dat. U koncových zařízení sítě tento problém není příliš znát, ale projeví se na páteřních vysokorychlostních sítích, kde komunikuje velké množství uživatelů, kteří si vyměňují nejrůznější informace.

Přístup k internetu je v současné době snadný. Proto je postupně využíván pro stále více činností, při niž si mohou jednotlivé strany vyměňovat i citlivé údaje. Protože takové údaje jsou cenné, snaží se je nepoctivci získat nelegální cestou a vydělat na nich. Nebo mají snahu zablokovat některé severy a ochromit tím jejich fungování. Proto se množí nejrůznější útoky po sítích a je snaha těmto útokům předcházet, popřípadě jim zabránit v napáchání velkých škod. Proto je síťový provoz potřeba analyzovat, buď v reálném čase nebo odchytávat a ukládat pro pozdější analýzu. Při odchytávání vyvstává problém, protože sběrnice ze síťových karet nezvládají přenášet tak velký objem dat v potřebném čase. Z tohoto důvodu se tato práce zabývá zmenšením objemu přenášených dat, ale zároveň zabráněním ztráty důležitých dat.

Ovšem je možné mít na síťové kartě před sběrnicí modul s FPGA, kde by byl implementován algoritmus pro kompresi dat. Potřebujeme v rámci možností obecný algoritmus, protože budeme komprimovat síťový provoz, který na páteřní síti může být jakýkoliv. Po této kompresi by byl objem snížen na velikost, kterou už by byla sběrnice schopna přenést. Po přenosu na externí médium už není problém provést dekompresi a příslušná data analyzovat.

Struktura této práce je shrnuta v následujícím odstavci. V druhé kapitole je obecný úvod do komprese dat. Třetí kapitola se zabývá IP pakety, jsou zde rozebrány jednotlivá pole hlaviček paketů pro účely komprimace. Čtvrtá kapitola se zabývá kompresi v IP sítích. Jsou zde popsány metody, které je možné použít při komprimování IP provozu. Ty jsou rozděleny do dvou skupin podle zaměření, kdy v první skupině jsou komprimovány pakety jako celek a ve druhé je snaha o komprimaci hlaviček paketů. V páté kapitole je popsán vývoj kompresního algoritmu LZW a také je zde uvedeno, jak algoritmus pracuje. Je zde zpracováno testování tohoto algoritmu na různých sadách IP paketů a také jsou zde popsány výsledky jednotlivých testů. V šesté kapitole je popsána implementace algoritmu LZW pomocí nástroje Xilinx®Vivado HLS. Tento nástroj provádí vysokoúrovňovou syntézu například z jazyka C do jazyka VHDL pro popis hardware.

Kapitola 2

Komprese dat

Základem této kapitoly je studijní opora [2].

Komprese dat je jedním z případů kódování. Kódování se zabývá vzájemnému jednoznačnému přidělení množiny symbolů jiné množině symbolů. Kódování můžeme rozdělit do tří skupin podle redundance, která vyjadřuje míru nadbytečných informací pro význam:

- nemění redundanci, například některé druhy šifrování,
- zvětšuje redundanci, kódy pro opravu chyb v datech,
- zmenšují redundanci, komprese dat.

My se budeme dále zabývat třetí možností, kompresí dat. Komprese dat je proces, kdy ze vstupního toku dat vytváříme výstupní s menší velikostí. Toho dosáhneme snížením redundance ve vstupních datech. Redundance je abecední a kontextová. U abecední nás zajímá četnost výskytů symbolů ve vstupních datech. Potom se přidělí kratší kód četnějším výskytům. U kontextové redundance nás zajímají posloupnosti symbolů. Tato redundance je častá u multimediálních souboru, kdy například následující obraz videa má jen nepatrný rozdíl od předchozího.

Důvodem pro použití komprese je nedostatek úložného prostoru nebo malé přenosové pásmo linky. S tím souvisí cena a technické možnosti. V dnešní době, kdy je velký objem dat přesouvám přes internet, je zmenšení tohoto objemu důležité.

Kompresní metody jsou ztrátové a bezeztrátové. Ztrátové znamená, že zmenšení velikosti je na úkor zachování všech informací. To nám nemusí vadit například při kompresi obrázku, videa nebo zvuku, kdy lidské oko nebo ucho stejně nepostřehnou všechny detaily. Oproti tomu při komprimaci textu by mám ztráta několika bitů přinejmenším ztížila pochopení obsahu, nebo by se stal text nečitelný úplně. Při bezeztrátové komprimaci jsou veškeré informace zachovány a po dekompresi dostaneme vždy původní data.

Pro měření míry komprese použijeme veličinu *kompresní poměr*, který je definován následujícím vzorcem:

$$kp = \frac{|out|}{|in|}$$

kde kp je kompresní poměr, $|out|$ je velikost výstupního toku dat a $|in|$ velikost vstupního toku dat. Tato hodnota vyjadřuje, jakou část z původní velikosti zabírají data po provedení komprese. Je zřejmé, že čím je kompresní poměr nižší, tím větší komprese dat je dosaženo. Pokud by ale vyšla hodnota vyšší než jedna, znamenalo by to, že komprimovaná data zabírají více prostoru než původní. Taková komprese, kdy se ani o kompresi nejedná, je samozřejmě zcela k ničemu.

Kapitola 3

IP Pakety

IP paket je blok dat přenášený po síti pod Internet protokolem. Skládá se z hlavičky a vlastních dat. V hlavičce jsou informace potřebné pro přenos po síti. V současné době se používají dvě verze IP protokolu a to IPv4 a IPv6. Obě verze protokolu mají odlišné hlavičky. IPv6 verze byla zavedena kvůli nedostatku adresového prostoru, který bylo možné obsáhnout verzí IPv4. Oba protokoly budou blíže popsány v následujících sekcích [7].

Po IP sítích se přenáší velké množství dat různých velikostí v nejrůznějších podprotokolech. Nejpoužívanější jsou dva protokoly transportní vrstvy a to TCP pro spolehlivý přenos a UDP pro nespolehlivý přenos. U spolehlivého přenosu je zajištěno doručení paketu, nebo je odesílatel informován o nedoručení. Ovšem tato spolehlivost je vykoupena větší režií. Na druhé straně při použití nespolehlivého přenosu, kde tato režie není, není ani odesílatel informován o případném nedoručení paketu. Dále zde je velké množství aplikačních protokolů jako: HTTP (internetové stránky), SMTP (email), FTP (přenos souborů), DNS (informace o adresách), SIP (internetová telefonie) a další. IP sítě sjednotily množství dříve využívaných sítí například IP telefony využívají spojování okruhu. Avšak tím, že jsou dnes řešeny pomocí IP paketů, má každý paket svoji hlavičku a ta se se stejnou informací musí přenést navíc. Vyvstává otázka, do jaké míry je daný přenos efektivní. Efektivitu můžeme definovat následujícím vzorcem podílu počtu bytů, které nesou potřebné informace, a počtem celkově přenesených bytů:

$$efektivita = \frac{|info|}{|celek|}$$

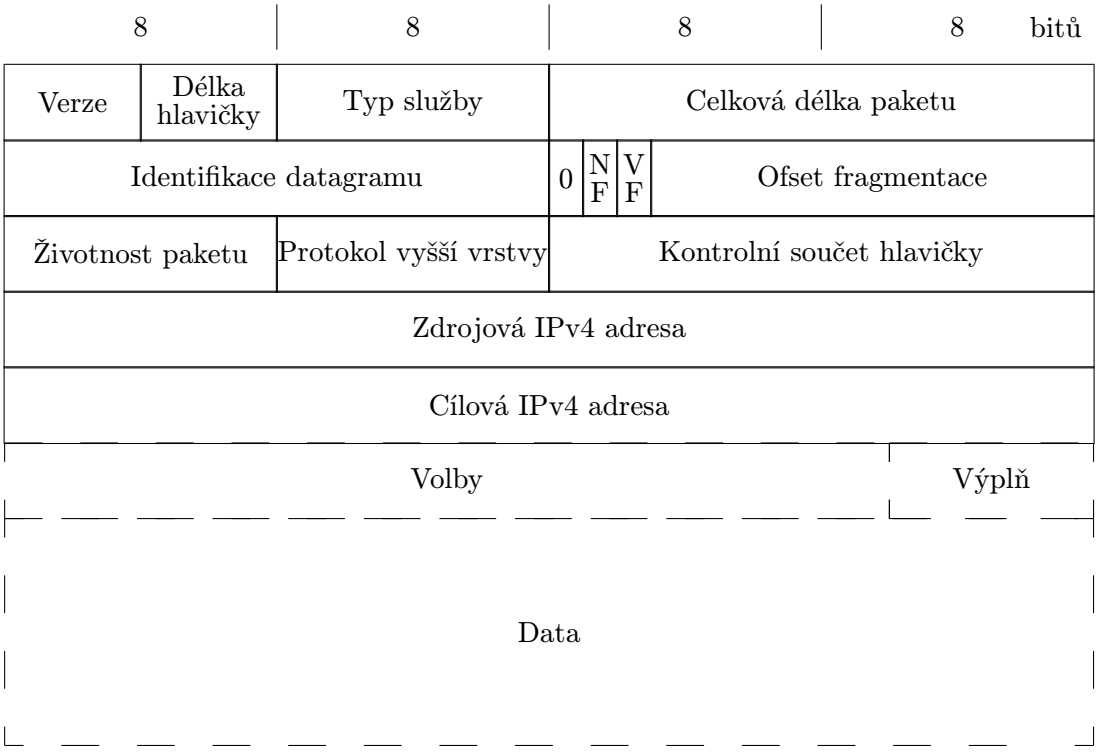
kde $|info|$ je velikost užitečné informace v bytech a $|celek|$ je celkový počet přenesených bytů. Efektivita je důležitá, když je vysoká cena přenosu. Pokud je omezená kapacita přenosové linky dochází ke zpožďování paketů nebo k jejich zahazování [6].

3.1 IPv4

Tato sekce vzchází z přednášky [4].

Na obrázku 3.1 je znázorněn paket protokolu IPv4, jehož hlavička má díky volbám proměnnou velikost. Pole verze je pro všechny pakety IPv4 konstantní. Protože hlavička nemá konstantní velikost, proto i pole délka hlavičky je proměnné. Avšak pro daný tok, pakety se stejnou zdrojovou a cílovou adresou a také stejným zdrojovým a cílovým portem (na transportní vrstvě) v daném čase, by hlavičky mohly mít stejnou délku. Stejně tak pole typ služby by mělo být stejné pro daný tok. Celková délka paketu záleží na velikosti odesílaných dat, tudíž nejde blíže specifikovat. Pole identifikace datagramu je unikátní náhodné číslo,

ale bývá generováno globálním čítačem v operačním systému, tudíž může mít pro daný tok jen malý rozdíl mezi jednotlivými pakety. Následující pole NF (nefragmentovat) a VF (více fragmentů) jsou příznaky k fragmentaci a pokud je paket fragmentován, je vyplněno i pole ofset fragmentace. Pole ofset fragmentace má také jen malý rozdíl mezi jednotlivými pakety daného toku. Pole životnost paketu, které udává přes kolik ještě směrovačů může paket projít, je v ideálním případě v jednom místě sítě stejné. Avšak protože dochází k zacyklení a pakety nemusí procházet stejnou trasou, může tato hodnota být blíže nespecifikovatelná. Pole protokol vyšší vrstvy je pro daný tok konstantní, určuje například protokoly: TCP, UDP, ICMP, ... Pole kontrolní součet se přepočítává na každém směrovači, tudíž je jeho hodnota také blíže nespecifikovatelná. Pole zdrojová a cílová adresa jsou pro daný tok konstantní. Pole volby blíže specifikuje například směrování nebo bezpečnost. Pro daný tok by měly být stejné. Pole výplň je nadbytečné, slouží pouze pro zarovnání dat na celé byty.



Obrázek 3.1: IPv4 paket.

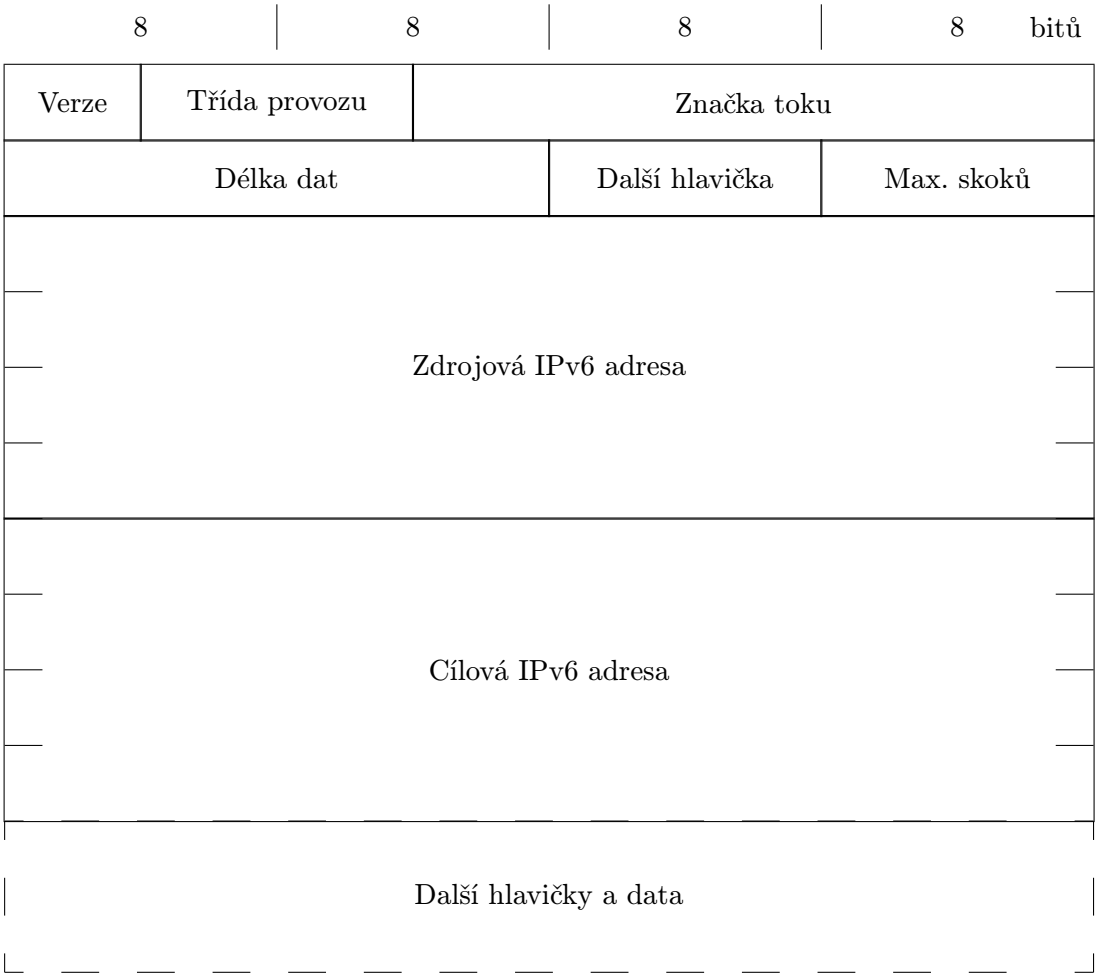
Jak je zmíněno výše, v daných tocích se zbytečně odesílá množství redundantních informací. Omezením této redundance můžeme snížit datový tok po síti.

3.2 IPv6

Tato sekce vychází z přednášky [5].

Na obrázku 3.2 je znázorněn paket protokolu IPv6. Jeho hlavička má pevnou velikost a to 40 B. Pole verze je pro každý paket IPv6 stejné. Pole značka toku identifikuje tok, proto je pro daný tok stejné. Pole délka dat závisí na dalších rozšiřujících hlavičkách a velikosti přenášených dat. Pole další hlavička umožňuje přidání rozšiřující hlavičky do řetězce

hlaviček pro další specifikaci jako jsou směrování, fragmentace, ... Tyto rozšiřující hlavičky jsou za hlavní hlavičkou před daty. Je jich jen omezený počet a musí být v určitém pořadí. Teoreticky by mohly být pro daný tok stejné. Pole maximální počet skoků udává, kolika směrovači může ještě paket projít, stejně jako u IPv4. Pole se zdrojovou a cílovou adresou jsou pro daný tok shodné.



Obrázek 3.2: IPv6 paket.

I u rozšiřujících hlaviček je možné určit, které jsou pro daný tok konstantní, které mají jen malý rozdíl v rámci paketů daného toku a o kterých se nedá určit nic bližšího. V každé rozšiřující hlavičce je pole další hlavička. U poslední hlavičky v řetězci hlaviček, tou může být i hlavní hlavička, je na tomto místě příslušný identifikátor, který má význam, že už žádná rozšiřující hlavička nenásleduje.

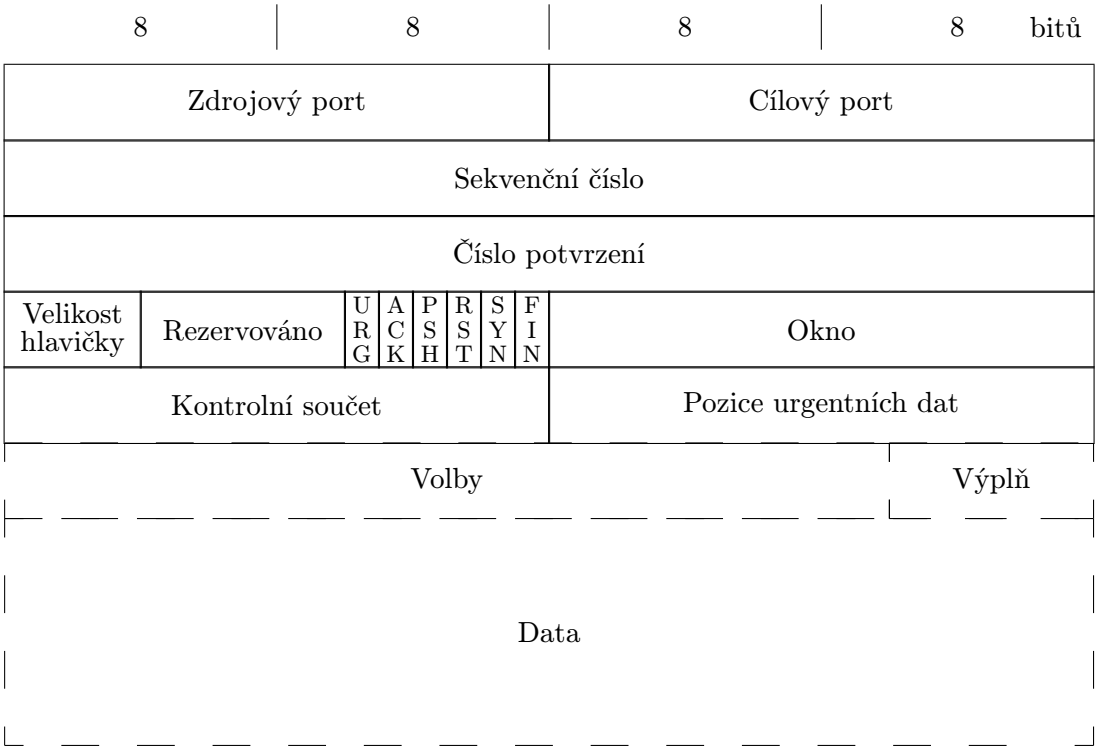
3.3 TCP a UDP

Takto sekce vychází z přednášky [3].

U nespolehlivého přenosu na transportní vrstvě má hlavička paketu UDP pouze čtyři pole. Těmito poli jsou zdrojový a cílový port, které jsou pro daný tok stejné. Dále pak

obsahuje délku hlavičky s daty, která závisí na velikosti těchto dat, a kontrolní součet. Tyto pole neobsahují redundantní informace, ale například při přenosu streamovaného videa mohou mít pakety jednotnou velikost a potom by bylo i příslušné pole délka konstantní.

Oproti UDP paketu je kvůli zajištění spolehlivého přenosu TCP paket složitější a proto je zobrazen na obrázku 3.3. Stejně jako u UDP jsou pole pro zdrojový a cílový port u daného



Obrázek 3.3: TCP paket.

toku stejná. Pole se sekvenčním číslem se postupně inkrementuje a tudíž nabývá jen malého rozdílu mezi jednotlivými pakety. To stejné platí i pro číslo potvrzeného paketu, zde sice může být rozdíl větší než u sekvenčního čísla, ale stále je výhodný. Pole velikost hlavičky je závislé na upřesňujících volbách. Pole rezervováno je v současné době nadbytečné. Dále je zde šest polí s příznaky, z nichž SYN (požadavek na vytvoření spojení), FIN (požadavek na ukončení spojení) a RST (reset spojení) se v rámci daného toku odesílají jen minimálně. Příznaky URG (urgentní data) a PSH (požadavek co nejrychlejšího doručení) se využívají u důležitých přenosů. Nejvíce používaný je příznak ACK (indikuje platné číslo potvrzení). Pole okno udává aktuální velikost okna s posílanými pakety. Tato hodnota se mění v případě zahlcení sítě. Je tedy do značné míry stejná a mění se jen zřídka. Pole s kontrolním součtem neobsahuje redundantní informace. Pole pozice urgentních dat je definována jen s příslušným příznakem, jinak je nadbytečné. Pole volby by mohlo být pro daný tok konstantní. Pole výplň nenese žádnou důležitou informaci, protože slouží pouze k zarovnání následujících dat.

Kapitola 4

Komprese IP paketů

Tato kapitola vychází z článku [6].

Kompresi IP paketů můžeme rozdělit do dvou základních směrů. Jedním je komprimování celých paketů pomoci „univerzálního“ komprimujícího algoritmu. Druhou možností je zaměřit se na toky po síti a nepřenášet redundantní informace z hlaviček.

4.1 Komprimace celých paketů

Při komprimaci celých paketů je brán paket jako blok dat, který se zkomprimuje pomocí vybraného algoritmu. Nejčastěji se používají slovníkové algoritmy. Zde jsou dvě možnosti, pevně definovaný slovník a slovník vytvářený za běhu. U pevně definovaného slovníku musí mít kompresor a dekompresor stejný slovník, který se za běhu nemění. Jelikož se po IP síti posílají nejrůznější data, je prakticky nemožné vytvořit obecný efektivní pevně definovaný slovník. Druhou variantou je slovník vytvářený algoritmem za běhu. Ten se lépe přizpůsobuje aktuálně komprimovaným datům. Kompresor a dekompresor spolu musí neustále komunikovat, aby udržely synchronizaci. Pokud by došlo ke ztrátě synchronizace, musí se zajistit její opětovné ustálení. Avšak po tuto dobu, od ztráty po znovu ustálení, budou posílaná data neplatná a budou se muset zahodit. Druhou nevýhodou je paměťová náročnost, protože slovníky jsou rozsáhlé.

Problémy s malou pamětí a kvalitou linky řeší například algoritmus *packet by packet dictionary*. Základem je komprimování každého paketu samostatně a spolu s ním je posílán i kontext. Tím se předchází ztrátě synchronizace a také je použit jen malý slovník. Tyto výhody jsou vykoupeny menší efektivitou.

4.2 Komprimace hlaviček (Header Compression)

Hlavičky paketů mají přesně definovanou strukturu, která se skládá z polí předem definované velikosti. Každé pole má také definované hodnoty, jakých může nabývat. Dalším podstatným aspektem je využití IP toků. Hlavičky paketů ze stejného toku mají podstatnou část polí totožnou jako například IP adresy zdroje a cíle. Další pole jako například pořadové číslo paketu se mění postupně, jeho diference je proto malá. Je zřejmé, že se zbytečně posílají redundantní data. Odstranění této redundance je základem pro následující přístupy.

4.2.1 Van Jacobson Header Compression (VJHC)

Van Jacobsonovo schéma komprimace hlaviček využívá znalostí o TCP/IPv4 hlavičkách. Nejdříve rozdělí provoz na jednotlivé toky. Každému toku přidělí unikátní identifikátor kontextu CID. Ten slouží kompresoru i dekompresoru. Pole v hlavičkách, která jsou pro daný tok stejná, se již nemusí posílat v každém paketu, protože stačí poslat CID, které tyto informace nahrazuje. Jsou to IP adresy, IP typ protokolu a TCP čísla portů.

VJHC komprese spojuje hlavičky IP a TCP do jednoho celku. Více jak polovina polí se nemění. Oproti tomu některá pole jako kontrolní součet se musí u příjemce znovu přepočítat. O pole celková délka a identifikace se nestará algoritmus, ale jsou nechána na režii linkové vrstvy. Jiná pole se nemusí měnit pravidelně, ale jen jednou za čas. Taková pole je možné z komprimované hlavičky vynechávat a posílat jen když ke změně došlo. Některá pole se mění jen o malou hodnotu, v takovém případě je možné odesílat pouze diferenci této hodnoty. Je zřejmé, že VJHC je efektivní pro TCP/IPv4 hlavičky.

U VJHC může dojít při přenosu ke dvěma typům chyb. K chybě CRC (kontrolní součet) na linkové vrstvě, to znamená, že je chybný komprimovaný paket nebo je špatný TCP kontrolní součet, a to znamená, že je chyba v paketu, který byl komprimován. Při detekování chyby je příslušný paket zahozen. Ovšem toto zahození způsobí další problém. Protože po zahození paketu může dojít ke ztrátě synchronizace a další paket nemusí být dekomprimován správně. Tudíž se musí zahodit i tento paket. I další pakety musí být zahazovány dokud se synchronizace neobnoví. Obnovení synchronizace je možné například doručením nekomprimovaného paketu. Této chybě lze předcházet zavedením rozdílu sekvenčního čísla u příjemce mezi posledním správně přijatým paketem a současným paketem. Když je paket zahozen, u dalšího paketu se s pomoci tohoto rozdílu spočítá správné sekvenční číslo a poté je paket správně dekomprimován. Při špatném TCP kontrolním součtu není odesláno potvrzení o přijetí paketu a odesílatel po určité době pošle paket znovu. Ovšem po tomto odeslání musí kompresor zajistit znovu ustálení synchronizace.

U TCP/IPv4 při použití VJHC lze dosáhnout kompresního poměru 10 % (z původních 40 B před kompresí se redukuje na 4 B).

Hlavní nevýhodou je omezené použití a náchylnost na ztrátu synchronizace. VJHC se hodí na spolehlivé linky s převahou TCP/IPv4. I Když tento protokol používá řada služeb jako HTTP, FTP, SSH, ... není jediný. Například pro, v dnešní době rozšiřovaný z důvodu nedostatku adres, IPv6 protokol je tento přístup nepoužitelný. Nebo pro transportní protokol nespolehlivého přenosu UDP je taktéž nepoužitelný.

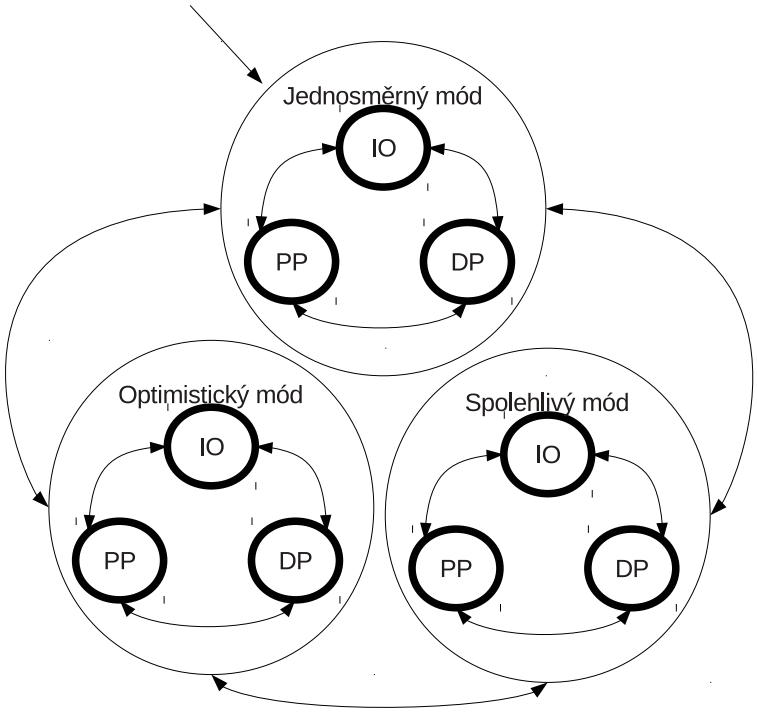
4.2.2 IP Header Compression (IPHC)

Komprimace hlaviček IP je podobné jako již zmíněné VJHC, protože je využito snížení redundance v polích hlaviček. Hlavním rozdílem je, že při IPHC se komprimuje pouze IP hlavička, avšak podporuje obě verze IP protokolu (IPv4 i IPv6). Pro identifikaci toků se opět využívá CID. I chyby jsou podobné jako u VJHC. Jsou zavedeny metody pro zotavení se z chyby TCP kontrolního součtu. Pro pakety mimo TCP je periodicky posílán nekomprimovaný paket, který zajistí kontrolu kontextu. Tato opatření ovšem sníží efektivitu, ale zabrání zbytečnému zahazování paketů. IPHC redukují IP hlavičku na 2 B pro mimo TCP a na 4 B pro TCP přenos. Hlavní výhodou je nezávislost na transportní vrstvě, která je vykoupena menší efektivitou u TCP přenosu oproti VJHC.

4.2.3 RObust Header Compression (ROHC)

Odolná komprese hlaviček je nový projekt vyvíjený v pracovní skupině ROHC ve IETF (Internet Engineering Task Force). V porovnání s IPHC má být odolnější a poskytovat větší efektivitu. ROHC je komplexní řešení komprimace hlaviček. Je flexibilnější než VJHC, protože se zaměřuje na větší počet hlaviček, ne pouze na TCP/IPv4. Proto nemá problém s IPv6 provozem, tunelováním a bezpečnostními protokoly. Tato flexibilita je možná jen díky komplikovanému technickému řešení. ROHC má v sobě zabudovány předchozí známé komprimační algoritmy a navíc je přidáno mnoho nových sofistikovaných mechanizmů. Tím je zajištěna také spolehlivost.

Kompresor a dekompresor jsou konečné stavové automaty. Stavy kompresoru jsou: *Inicializace a Obnovení* (IO, Inicializaion and Refresh), *První Příkaz* (PP, First Order) a *Druhý Příkaz* (DP, Second Order). Pro dekompresor jsou také tři stavy a to: *Bez Kontextu* (BK, No Context), *Statický Kontext* (SK, Static Context) a *Celý Kontext* (CK, Full Context). Přenos začíná ve stavu bez kontextu pro daný tok (CID). Zde je komprimace omezena na minimum. Teprve když se kompresoru podaří zavést kontext pro dekompresor, může začít vysílat data v dalším stavu. V tomto stavu je možné díky kontextu posílat data zkomprimovaná více než v předchozím stavu.



Obrázek 4.1: Operační módy ROHC.

ROHC zavádí tři operační módy: *Jednosměrný* (J-mód, Unidirectional), *Obousměrný Optimistický* (O-mód, Bidirectional Optimistic) a *Obousměrný Spolehlivý* (S-mód, Bidirectional Reliable). ROHC začíná v J-módu a podle zpětné vazby se může přesunout buď do O-módu nebo do S-módu. J-mód se stará o pravidelné obnovování kontextu, což je podobné jako u IPHC pro UDP přenos dat. O-mód využívá zpětnou vazbu pro opravu chyb

a S-mód využívá všech možností, aby předcházel ztrátě kontextu nebo synchronizace. Každý s těchto operačních módů zahrnuje všechny tři stavy IO, PP a DP, což je vidět na obrázku 4.1. Při detekci chyb se ROHC automaticky přesune do stavu s menší kompresí dat nebo do nižšího operačního módu, pokud je zaznamenáno velké množství chyb.

ROHC využívá pro přenos vlastní pakety, které se v základu skládají ze čtyř částí:

- Vyplnění (Padding) je na začátku komprimované hlavičky a slouží k zarovnání na celý byte.
- Zpětná vazba (Feedback) slouží k uchování synchronizace kontextového stavu.
- Hlavička (Header), zde jsou zkomprimované informace původní hlavičky
- Data (Payload), na tomto místě jsou data přenášená původním paketem

Dekompresor po přijetí tohoto paketu okamžitě generuje zpětnou vazbu pro kompresor. Stejně jako ostatní přístupy pro komprimování hlaviček i ROHC využívá kontrolního součtu CRC na linkové vrstvě. Je tím zajištěno správné přenesení komprimovaných dat. Aby bylo dosaženo všech výhod, ROHC dokáže ze zpětné vazby odhalit chybu a také ji napravit.

Hlavní výhodou ROHC je, že je možné navrhnout kompresi jakéhokoliv typu hlavičky. Je více odolný na nespolehlivých linkách než předchozí přístupy IPHC a VJHC. Je také výhodný pro malé pakety jako jsou potvrzení u TCP přenosu nebo VoIP pakety na linkách s malou kapacitou nebo pomalou přenosovou rychlostí.

4.3 Shrnutí uvedených komprimačních algoritmů

Je zřejmé, že pro provoz s převahou malých paketů mají výhodu algoritmy komprimace hlaviček. U malých paketů zabírá hlavička velkou část velikosti paketu. S rostoucí velikostí paketů tato výhoda pomíjí. U velkých paketů má hlavička zanedbatelnou velikost, tudíž její komprimace nepřinese moc velký užitek. Na rozdíl od komprimace celých paketů, kde s rostoucí velikostí efektivita stoupá. Při komprimovaní malých paketů, kde značnou část zabírá hlavička, která se tímto přístupem špatně komprimuje, můžeme dosáhnout i zvětšení velikosti. Avšak ani u velkých paketů není zaručeno, že bude komprimace úspěšná. Zde záleží i na datech, která paket nese pod hlavičkou. Pokud již tato data byla komprimována (například programem ZIP) nebo se jedná o data z multimediálního kodeku, je velká šance, že další komprimací se už moc jejich velikost nezmenší.

Kapitola 5

LZW algoritmus

V mé práci se nepočítá s možností komunikace kompresoru a dekompresoru, protože v určitém bodě na páteřní síti bude pouze kompresor. Algoritmy komprese hlaviček počítají s provozem na linkové vrstvě, což náš případ není. Proto se zaměříme na komprimaci celých paketů. Protože síťový provoz na páteřní sítí je málo specifický, zvolil jsem algoritmus, který nemá pevně definovaný slovník, ale dynamicky vytvářený za chodu kompresoru podle vstupních dat. Takovým algoritmem je právě algoritmus LZW, který jsem použil a je popsán dále v této kapitole. Je zde uveden jeho vývoj a testování na datových sadách, které jsou odchycené z počítačové komunikace po síti.

5.1 Vývoj LZW algoritmu

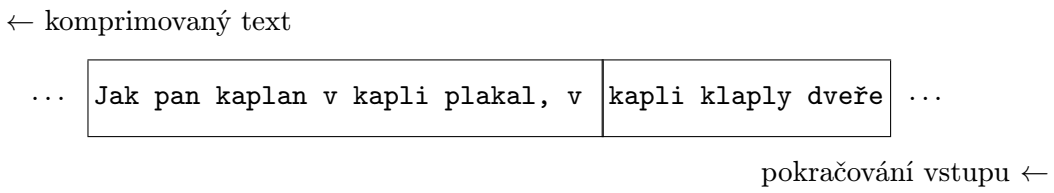
Tato sekce vychází ze studijní opory [2].

Než si popíšeme algoritmus LZW, musíme nejdříve popsat jeho vývoj. Před tímto algoritmem byly totiž ještě postupně dva algoritmy a to: LZ77 a LZ78. Jména nesou po svých tvůrcích jimiž jsou Jacob Zip a Abraham Lempel. V zásadě se jedná o slovníkové metody se slovníkem vytvářeným za běhu. Tyto metody se staly základem pro mnoho bezeztrátových kompresních algoritmů, my se budeme zabývat pouze touto jednou řadou.

5.1.1 LZ77

Primární ideou metody LZ77 (někdy také LZ1) je použít již dříve načtený vstup jako slovník. Toho docílíme zavedením pohyblivého okna, které má pevnou, předem definovanou velikost a které se postupně posouvá nad vstupními daty. Toto okno rozdělíme na dvě části. Levá část, již načtená a zkomprimovaná data, která slouží jako slovník, se nazývá *vyhledávací buffer* (search buffer). Pravá část je *předvídací buffer* (look-ahead buffer), zde je vstup, který teprve bude komprimován. Toto rozdělení je vidět na obrázku 5.1. V praxi se velikost vyhledávacího bufferu pohybuje okolo tisíce bytů a velikost předvídacího bufferu je jen v řádu desítek bytů.

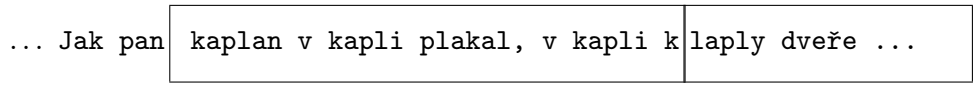
Činnost kompresoru si ukážeme na příkladu z obrázku 5.1. Kompresor začne vyhledávat první znak z předvídacího bufferu `k` ze slova `kapli␣` od konce (zprava doleva) ve vyhledávacím bufferu. První shodu najde ve stejném slově `kapli␣`. Toto slovo je s ofsetem 16 a velikost shody je 6 symbolů včetně mezery za slovem. Kompresor dál hledá jestli nenajde další shodu ve vyhledávacím bufferu, která by byla delší než ta předchozí. Další shoda s písmenem `k` začíná ve slově `kaplan`. Jenže tato shoda má délku pouze 4 symboly, což je méně



Obrázek 5.1: Pohyblivé okno LZ77.

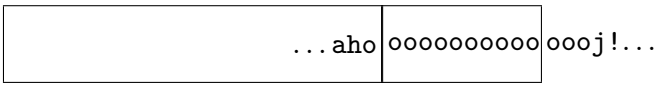
než předchozí. Proto bude odeslána značka (16, 6, `k`). Kdybychom našli více stejně dlouhých shod, je jedno, kterou pošleme, ale zpravidla to bývá kvůli jednoduchosti ta poslední. Dekodér příslušnou shodu najde a použije pro dekódování, ať je kdekoli ve vyhledávacím bufferu.

Pro značku LZ77 platí, že je složena ze tří částí: ofset, délka shody a následující symbol za shodou. V našem příkladu byl symbol `k` první znak následujícího slova `klaply`. Nyní se posune okno o velikost shody plus jedno místo doprava, nebo vstupní text doleva. Tento případ je na následujícím obrázku 5.2. Na začátku předvídacího bufferu se tedy objeví první znak za již odeslaným znakem v poslední značce. V našem případě první `l` slova `klaply`.



Obrázek 5.2: Pohyblivé okno LZ77 po posunu.

Hlavně na začátku, když je vyhledávací buffer prázdný, ale i později se může stát, že není nalezena žádná shoda. V takovém případě je odeslána značka s nulovým ofsetem i nulovou délkou a symbolem, pro který nebyla nalezena shoda. Právě tento případ je hlavním důvodem proč má značka třetí část, symbol. Je zřejmé, že v tomto případě o kompresi moc nejde, protože místo jednoho symbolu je vysláno ještě nemálo bitů v polích značky ofset a délka. Pole pro ofset musí být schopné obsáhnou délku celého vyhledávacího bufferu. Pokud má v praxi velikost tisíc bytů, bude potřeba ve značce 10 bitů. U pole délka shody nám stačí velikost taková, abychom obsáhli délku předvídacího bufferu mínus jedna. Tato situace je znázorněna na obrázku 5.3. Tento příklad je uměle vytvořený pro ukázku. V našem příkladu má předvídací buffer délku deset symbolů. Shoda je nalezena s ofsetem



Obrázek 5.3: Pro odvození velikosti pole délka shody značky LZ77.

jedna. Velikost této shody je devět, protože se jedná o devět kopií tohoto symbolu. Je tedy odeslána značka (1,9,o). I když se zdá, že značka na začátku ukazuje do prázdna, dekompresor s touto situací nemá žádný problém. Když dekóduje přijatou značku, tak na výstup postupně, symbol po symbolu, vypisuje dekódované symboly. S tím se mu zároveň posouvá vyhledávací buffer a tudíž další symbol už není neznámý, je kopií nově přijatého symbolu.

Dekompresor je mnohem jednodušší něž kompresor. Stačí mu pouze vyhledávací buffer. V něm najde příslušnou shodu. Tu zkopíruje a přidá na konec vyhledávacího bufferu a ještě přidá symbol ze třetího pole značky. Neustále udržuje vyhledávací buffer tak, aby končil s již dekódovaným vstupem.

5.1.2 LZ78

Metoda LZ78 bývá také označovaná LZ2. Na rozdíl od LZ77 nemá žádné posuvné okno, tudíž ani vyhledávací a předvídací buffer. Zato má slovník dříve nalezených řetězců. Tento slovník začíná prázdný nebo téměř prázdný. Jediným omezením jeho rozsahu je velikost paměti. Kompresor posílá dekompresoru značku, v níž je ukazatel do slovníku a následující nový symbol. Velikost shody je implicitně dána záznamem ve slovníku, tudíž ji není potřeba odesílat ve značce. Výhodou oproti LZ77 je, že se ze slovníku nic nevymazalo, a tak je větší pravděpodobnost, že bude nalezena shoda. Avšak má i nevýhodu a to, že slovník značně narůstá a proto je vyhledávaní náročnější.

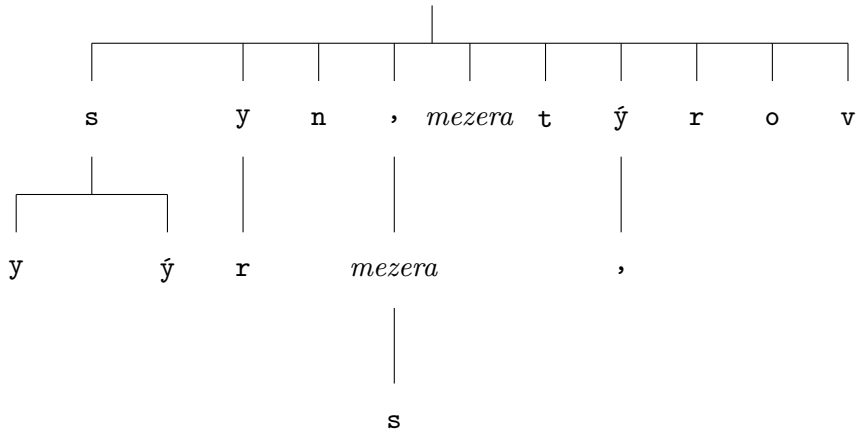
Kompresor načítá symbol ze vstupu, například: `x`. Pokud symbol není nalezen mezi jednoprvkovými řetězci ve slovníku, je přidán na první volnou pozici a je odeslána značka (0, `x`). Druhou možností je, že symbol `x` byl nalezen ve slovníku na pozici `p`. V tomto případě se ze vstupu načte další symbol, řekněme například `y`. Nyní se opět hledá ve slovníku, ale už dvouprvkový řetězec `xy`. Pokud není nalezen, je odespána značka (`p`, `y`), kde `p` je pozice prvního symbolu ve slovníku, a řetězec `xy` je přidán na první volnou pozici do slovníku. Pokud i řetězec `xy` byl nalezen, pokračuje se načítáním dalších symbolů ze vstupu, dokud je řetězec nacházen ve slovníku. Jakmile nalezen není, přidá se do slovníku a odešle se značka. Takto je postupně odeslán celý vstup. V následující tabulce 5.1 je příklad kompresoru pro řetězec: `syn, sytý, sýr, syrový,...`.

Vstupní symbol(y)	Slovník		Značka
	0	prázdný	
s	1	s	(0, s)
y	2	y	(0, y)
n	3	n	(0, n)
,	4	,	(0, ,)
␣	5	␣	(0, ␣)
sy	6	sy	(1, y)
t	7	t	(0, t)
ý	8	ý	(0, ý)
,␣	9	,␣	(4, ␣)
sý	10	sý	(1, ý)
r	11	r	(0, r)
,␣s	12	,␣s	(9, s)
yr	13	yr	(2, r)
o	14	o	(0, o)
v	15	v	(0, v)
ý,	15	ý,	(8, ,)

Tabulka 5.1: Komprimace pomoci LZ78

Je zřejmé, že metoda LZ78 začne přinášet větší užitek, až je slovník naplněn delšími

řetězci. Aby se zjednodušilo vyhledávání, je slovník řešen pomoci stromové struktury. Kořenem tohoto stromu je prázdné místo, jeho potomky jsou jednoprvkové řetězce v našem příkladě to byly `s; y; n; ,; ␣; t; ý; r; o; v`. Dále jejich potomky jsou jejich následníci. V našem příkladě pro `s` to jsou `y` a `ý`. Podle tohoto principu je dále vytvářen celý strom. Strom našeho příkladu je na obrázku 5.4. Pokud vezmeme abecedu 8-bitových symbolů (256 symbolů celkem), tak každý symbol by mohl mít za potomka znovu 256 symbolů. Je tudíž potřeba dobře hospodařit s pamětí. To zajistíme dynamickým přidáváním listů stromové struktury teprve tehdy, až to bude potřeba. Vyhledávání v této stromové struktuře je jednoduché. Začínáme u kořene a postupně hledáme potomky s odpovídající hodnotou až k listům.



Obrázek 5.4: Slovník LZ78.

Jelikož nemáme nekonečnou paměť, je výška stromu omezena. To znamená, že i slovník se nemůže vytvářet do nekonečna. V původní metodě není specifikováno, co v takovém případě dělat. Existuje ovšem několik možností:

1. Nejsnadnější je slovník zmrazit. Další listy se už nadále nebudou přidávat, ale bude možné slovník dále používat, nyní ovšem už jen staticky. To je nevýhodné, pokud bude další vstup výrazně rozdílný od předchozího, protože se na tuto změnu nebude reagovat.

2. Při vyčerpání paměti se celý slovník odstraní a začne se s novým od začátku. Toto řešení je výhodné pokud se vstupní data výrazně mění, protože nejsou zachovány zastaralé fráze, které by se stejně už nikdy nepoužily. Je to podobný předpoklad jako u metody LZ77.

3. Sofistikovaný algoritmus pro odstranění některých frází. Například mohou být odstraňovány nejdéle nepoužité fráze. Ovšem je problém, jak tyto fráze určit. Musely by být zavedeny příznaky, které by s výběrem pomáhaly. To samozřejmě zesložití celou kompresi.

Dekompresor je složitější než u LZ77. Musí si stejně jako kompresor budovat vlastní slovník, podle něhož dekóduje komprimovaná data.

5.1.3 LZW

LZW modifikoval Terry Welch z LZ78 v roce 1984. Nejvýraznější změnou je zjednodušení značky pouze na jedinou položku a tou je ukazatel do slovníku. Posílání symbolu je nahrazeno inicializací slovníku na všechny možné jednoprvkové řetězce. Typické je 256 položek, totiž všechny 8-bitové symboly. Tím je zajištěno, že se každá značka ve slovníku vždy najde.

Kompresor načítá postupně symboly ze vstupu do řetězce R. Na začátku je tento řetězec prázdný. Po načtení symbolu x je hledána shoda řetězce $R + x$ se záznamy ve slovníku. Operátor $+$ značí konkatenaci řetězců, v našem případě řetězce R s jednoprvkovým řetězcem se symbolem x. V případě nalezení této shody, je tato konkatenace uložena do řetězce R ($R = R+x$). Pokračuje se načtením dalšího symbolu. V případě nenalezení shody je odeslán ukazatel do slovníku na řetězec R, na první volnou pozici je do slovníku přidán řetězec $R+x$ a řetězec R je vymazán a je do něj vložen symbol x. Opět se pokračuje načtením dalšího symbolu. V následující tabulce 5.2 je ukázán příklad kompresoru pro řetězec `máma má mák`. Pro zjednodušení počítáme se slovníkem, který má pouze symboly obsažené v tomto řetězci.

Slovník (řetězec R)		Značka
0	m	
1	á	
2	a	
3	k	
4	␣	
5	má	(0)
6	ám	(1)
7	ma	(0)
8	a␣	(2)
9	␣m	(4)
10	má␣	(5)
11	␣má	(4)
12	ák	(1)
13	k*(EOF)*	(3)

Tabulka 5.2: Komprimace pomoci LZW

Slovník má už od začátku zaplněno 256 položek. Je tudíž potřeba mít ukazatel větší než 8-bitový. Jednoduchá implementace by použila 16 bitů, čemuž odpovídá 64 K položek. Ale i tento počet se zaplní poměrně rychle. Problém se zaplněním slovníku se dá řešit podobně jako u LZ78. I na příkladu si můžeme všimnout, že řetězce se ve slovníku prodlužují velice pomalu, vždy jen o jeden znak. Je tudíž značně zdlouhavé, než je možné dosáhnout lepší komprese. LZW se tedy vstupním datům přizpůsobuje pomalu.

Dekompresor začíná se stejným slovníkem jako kompresor (typicky 256 symbolů). Po přijetí ukazatele vyhledá záznam ve slovníku, zkopíruje ho na výstup a také si rozšíří slovník o tento záznam s tím, že k němu ještě přidá první symbol z následujícího řetězce. Nyní načte nový ukazatel a vyhledá ho ve slovníku. Zkopíruje první symbol a přidá ho k poslednímu záznamu ve slovníku. Pokud neměnil aktuálně nalezený řetězec, může jej vypsat na výstup. V opačném případě vyhledá znovu podle ukazatele položku ve slovníku a tu zkopíruje na výstup. Znovu tento řetězec také přidá na konec slovníku s tím, že se v následujícím kroku k němu přidá ještě jeden symbol. Takto pokračuje až do konce vstupu.

5.2 Použitá implementace

Jelikož se jedná o poměrně starý algoritmus, je volně dostupná řada implementací. K testování a následně jako zdroj pro syntézu do hardware jsem využil implementaci LZW [1]. Jedná se o implementaci LZW, která při naplnění slovníku tento slovník resetuje do výchozího stavu. Výchozím stavem jsou všechny značky délky 8 bitů (256 symbolů).

5.3 Testování LZW komprese na síťovém provoze

Testování proběhlo na osobním počítači s dvoujádrovým procesorem (1,4 GHz) a operační pamětí DDR3 o velikoti 4 GiB. K testování byly vytvořeny datové sady odchytnutím paketů síťového provozu. Jednotlivé sady byly zaměřeny na určitý druh provozu s tím, že je v nich ponechána i provozní komunikace jednotlivých prvků sítě. Jedná se tedy o reálná data, která byla odesílána a přijímána na síti.

5.3.1 Popis jednotlivých testovacích sad

sada1 a sada2 Malé testovací sady zaměřené pouze na režijní komunikaci počítače s okolím. Velké množství různých protokolů (ICMPv6, RTMP, TCP, SSDP, ARP, DHCPv6, ...).

sada3 Sada vytvořená při procházení různých internetových stránek v internetovém prohlížeči. Sada obsahuje převážně pakety protokolu TCP. Nejpočetnějším protokolem aplikační vrstvy je pak protokol HTTP.

sada4 Sada zaměřená na komunikaci tzv. prohlížečové hry (online web browser game) se svým serverem. V této sadě je převaha paketů protokolu TCP.

sada5 a sada6 Sady vytvořené při komunikaci přes program Skype mezi dvěma počítači. Bylo přenášeno video (web–kamera), audio i text. Sada obsahuje hlavně pakety protokolu UDP.

sada7 Sada obsahuje komunikaci při sledování iVysílání České televize v internetovém prohlížeči v počítači. V sadě je převaha paketů aplikačního protokolu HTTP a transportního protokolu TCP.

sada8 a sada9 Sady vytvořené při sledování digitálního televizního vysílání na počítači v programu VLC Media Player. Sada je složená převážně z paketů protokolu UDP a na druhém místě z hlediska četnosti jsou pakety protokolu MPEG PES.

sada10 a sada11 Sady připravené při komunikací RPG hry se svým serverem. Sady jsou složeny s převahou paketů obou transportních protokolů TCP a UDP.

sada12 a sada13 Sady vytvořené při sledování videí přes internetový prohlížeč na portálu YouTube. V sadách je převaha paketů protokolu TCP.

sada14 a sada15 Sady z páteřní sítě. Tyto sady obsahují na rozdíl od předešlých sad také více zdrojových a cílových IP adres. Je zde také velké množství protokolů. Tyto sady mají i řádově větší velikost než předešlé sady.

5.3.2 Výsledky testů

V následující tabulce 5.3 jsou shrnuty výsledky komprimace jednotlivých testovacích datových sad. Je zde uvedena původní velikost datové sady, velikost sady po komprimaci pomoci algoritmu LZW a kompresní poměr příslušné sady. Velikosti jsou uvedeny v bitech. Kompresní poměry při testovaní jednotlivých sad jsou kromě tabulky znázorněny také v grafu 5.5, kde je navíc vyznačena červenou čarou hranice, pod kterou má komprese pozitivní účinek a nad níž negativní. Tím je myšleno, že pod hranicí je velikost komprimovaných dat snížena, ale nad touto hranicí je velikost dat po kompresi větší. V takovém případě je pak použití této komprese nevhodné. Velikosti datových sad před a po kompresi na konečném zařízení jsou znázorněny v grafu 5.6 a na páteřní síti v grafu 5.7. Sada1 a sada2 mají tak malé datové sady, že jsou v grafu prakticky neviditelné.

Číslo sady	Původní velikost [b]	Velikost po kompresi [b]	Kompresní poměr
1	117 800	92 129	0,782
2	86 456	66 255	0,766
3	42 925 480	41 416 225	0,965
4	32 849 460	30 760 286	0,936
5	71 661 100	76 808 163	1,072
6	72 358 744	77 184 483	1,067
7	163 572 628	182 174 590	1,114
8	264 193 452	266 262 535	1,008
9	432 545 752	439 806 265	1,017
10	55 112 540	43 162 863	0,783
11	53 016 404	31 547 683	0,595
12	158 312 864	177 556 039	1,122
13	77 149 840	86 590 030	1,122
14	1 393 688 662	846 750 547	0,608
15	2 478 019 885	1 717 344 843	0,693

Tabulka 5.3: Výsledky testování komprimace datových sad.

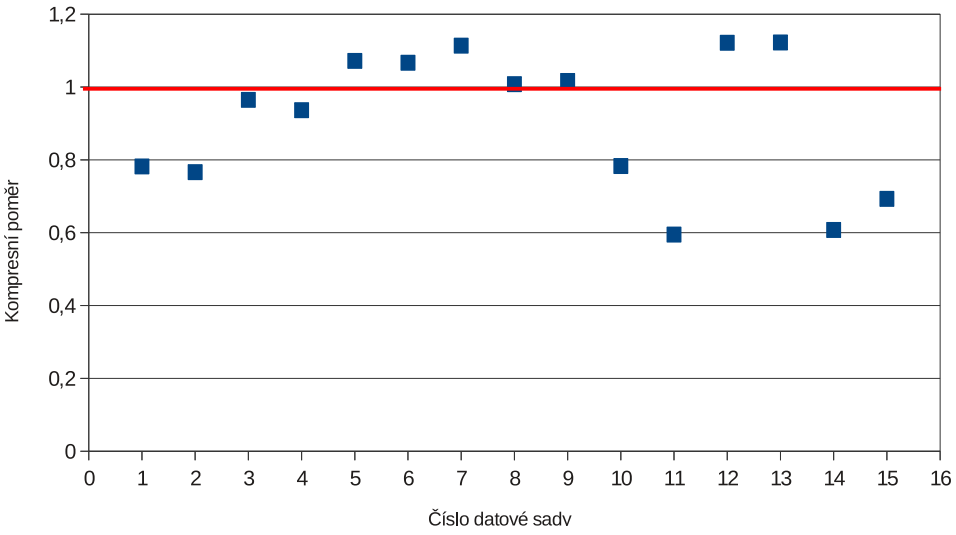
Z provedených testů vyplývá, že nejlepší komprese bylo dosaženo při komprimování testovacích datových sad 14 a 15, hned následovanými datovými sadami 10 a 11, poté třetí v pořadí jsou datové sady 1 a 2. Zvolený algoritmus LZW se tedy osvědčil pro kompresi různorodého přenosu paketů na páteřovém prvku sítě, kde komunikovalo velké množství uživatelů, kteří si vyměňovali různé druhy informací. Dále také u komunikace RPG hry se svým serverem, kde předpokládám komunikace probíhala strukturovaně, tzn. v předem definovaném formátu. V poslední řadě se algoritmus osvědčil u komunikace počítače s ostatními síťovými prvky, kdy jednotlivé pakety mají přesnou strukturu a jejich rozdíl v čase je minimální.

Při komprimaci datových sad 3 a 4 je minimální úspora velikosti oproti původní. Z příslušných testů vyplývá, že komunikace internetového prohlížeče se servery je pro zvolený algoritmus spíše nevhodná. Na internetových stránkách je velké množství multimediálního obsahu, který je pro bezeztrátovou komprimaci nevhodný. Ovšem tím, že není jediným obsahem, bylo dosaženo alespoň malé komprese.

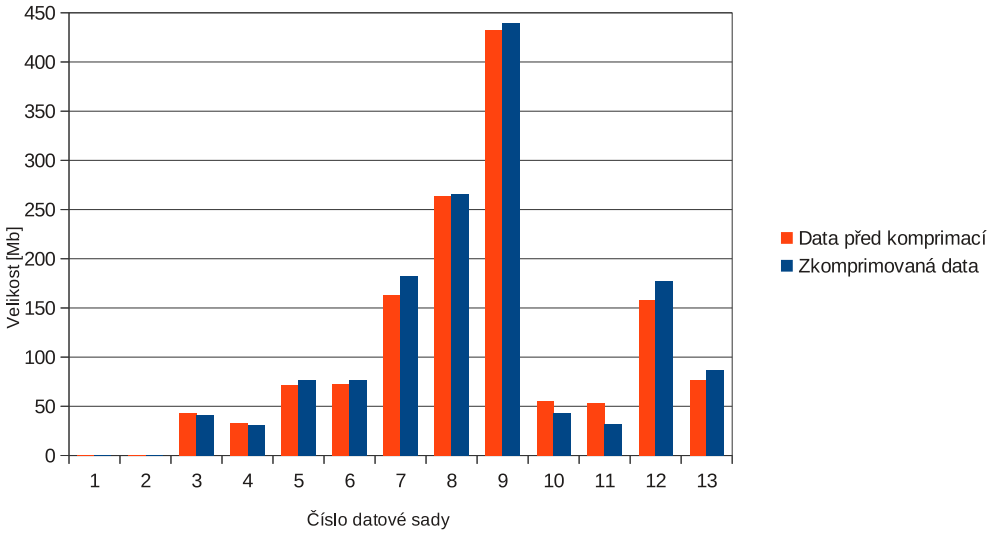
Při kompresi zbylých testovacích datových sad (5, 6, 7, 8, 9, 12 a 13) byl přenášen multimediální obsah. Z příslušných testů vyplynulo, že komprese nebylo dosaženo, ba naopak

velikost výsledné sady byla větší než původní. Potvrdil se tedy předpoklad, že v těchto datech je minimum redundantních informací a tudíž jsou ke kompresi nevhodné.

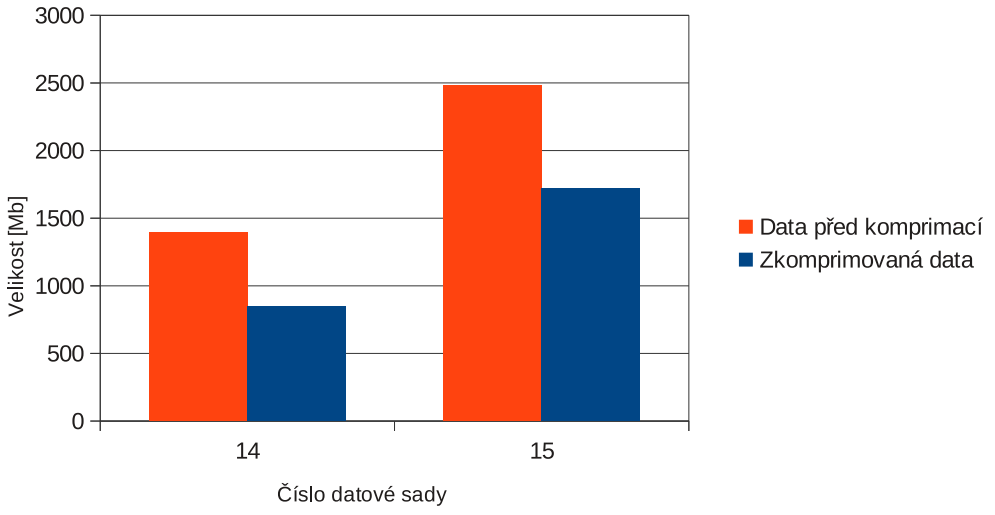
Algoritmus se tedy nejvíce osvědčil na datových sadách z páteřní sítě, což je dobře, protože je to místo, pro které má být v budoucnu použit.



Obrázek 5.5: Graf zobrazující kompresní poměr.



Obrázek 5.6: Graf srovnání velikosti dat z koncového zařízení před a po kompresi.



Obrázek 5.7: Graf srovnání velikosti dat z páteřní sítě před a po kompresi.

Kapitola 6

Imlementace pro FPGA

V této kapitole je popsána vysokoúrovňová syntéza implementace algoritmu LZW v jazyce C pro procesor do jazyka VHDL pro implementaci v FPGA. Pro tuto syntézu jsem využil program Xilinx®Vivado HLS (High Level Synthesis). Práci s tímto programem jsem se naučil díky cvičení [8] vytvořených společností Xilinx®. Program Vivado HLS je popsán v sekci 6.1. Vlastní práci a zkušenosti s tímto programem jsem shrnul v sekci 6.2.

6.1 O programu Vivado HLS

Tato sekce vychází z uživatelské příručky [9].

Vysokoúrovňová syntéza převádí C, C++ nebo SystemC specifikaci do logických obvodů pro FPGA. Usnadňuje tak programování pro hardware, protože se nemusí vytvářet obvod ručně, ale je automaticky převede a také optimalizován. Pro vstup je možné použít standardizovanou implementaci jednoho ze tří již zmíněných jazyků a to s minimální modifikací. Vysokoúrovňová syntéza funguje ve dvou odlišných částech. Těmi jsou syntéza algoritmu a syntéza rozhraní. Při syntéze algoritmu je převedena funkcionalita do logických obvodů a je zavedeno časování. Syntéza rozhraní převádí parametry funkcí na porty logických obvodů. Těmi mohou být například registr, sběrnice nebo FIFO fronta. Také zajišťuje potřebnou komunikaci s dalšími prvky systému. Stejně jako u manuálního navrhování obvodů existuje mnoho možností, jak implementovat a optimalizovat dané řešení, a jen správná kombinace nám zaručí ideální řešení, tak i vysokoúrovňová syntéza produkuje v krátkém čase nejlepší návrh.

Vysokoúrovňová syntéza se dá rozdělit na několik částí. V první řadě se zjišťuje, jakými cestami může program procházet, kdy a jak se větví a jak je to s daty. V druhém části je plánováno, jaké operace budou v jakém čase provedeny. Jestli například všechny současně v jednom taktu nebo postupně. Také je v této části rozhodováno, jaké prvky budou použity a v jakém pořadí, například se plánuje využití zřetězení. Na rozdíl od jazyka C, kde jsou typy proměnných v osmibitových násobcích, u hardware je možné efektivně využít přesně tolik místa, kolik je potřeba. Pro poměřování jednotlivých implementací slouží hodnoty oblast (Area), zpoždění (Latency), inicializační interval (Initiation Interval, II). Oblast udává, kolik hardware komponentů je použito. Zpoždění udává, kolik cyklů hodinového signálu proběhne mezi vstupem dat a vrácením výsledných dat. Inicializační interval udává dobu v cyklech hodinového signálu mezi vstupními daty a vyžádáním si dalších dat. Tyto údaje jsou konečným výstupem vysokoúrovňové syntézy celého systému.

Pro ověření výstupů z výsledného obvodu stačí napsat testovací program v jazyce C, není

tedy potřeba vytvářet testovaní v logických obvodech. Vysokoúrovňová syntéza podporuje tři druhy výstupních jazyků pro popis hardware a to VHDL, Verilog a SystemC. Dále pak také umožňuje vytvořit script pro simulaci logického obvodu pomoci jednoho z programů: ModelSim, VCS. Open SystemC Initiative (OSCI), NCSim, XSim, ISim nebo Riviera.

6.2 Implementace pomoci Vivado HLS

Při vlastní implementaci jsem vycházel z implementace pro procesor [1]. Tato implementace je v jazyce C. Mým cílem bylo vytvořit implementaci pro FPGA. Proto jsem zvolil následující tři typy FPGA:

Spartan3 (xc3s400pq208-3) ten je například na školním přípravku FitKit,

Virtex5 (xc5vlx155tff1136-3) který se používá na síťových kartách,

Virtex7 (xc7vx1140tflg1930-1) což je zástupce nejvýkonnější rodiny FPGA.

6.2.1 Implementace pro Spartan3

Začal jsem s implementací pro Spartan3. V první řadě jsem vytvořil nový projekt a připojil k němu potřebné soubory. Tato implementace je v projektu v „solution1“. Pro ověření Vivado HLS umožňuje analýzu při níž ověří správnost programu ve vstupním jazyce, v našem případě v C. Ta proběhla úspěšně a tak dalším krokem je analýza syntézy z jazyka C. Zde vyvstal problém, protože funkce pro práci se soubory ze standardní knihovny jazyka C nelze syntetizovat. Proto jsem byl nucen upravit funkčnost programu a místo do souboru vracet výstup na standardní výstup. Dále jsem přesunul vstup ze souboru do testovací části a syntetizoval pouze funkce komprese dat. Je pravda, že výsledkem bude prvek pro komprimaci dat a ten tudíž bude obecnější, protože ho bude možné zakomponovat do většího systému. Dále byl problém s předáváním dat komprimační funkci. Tato data byla předávána odkazem na řetězec a proto nebyla v době překladu známa jejich velikost. Bylo proto nutné v této funkci definovat maximální možný řetězec, do kterého se vstupní řetězec zkopíroval. Po těchto úpravách se mi podařilo úspěšně provést syntézu pro implementaci v FPGA.

6.2.2 Optimalizace

V druhém kroku jsem se pokoušel o optimalizaci. Pro větší přehlednost jsem tuto část oddělil v projektu do „solution2“. V reportu ze syntézy mi ale chyběly údaje o zpoždění. Předpokládám, že je to způsobeno tím, že se nedá vyhodnotit přesný počet průchodů některými cykly v době překladu. Chyběla mi tedy zpětná vazba a tak nevím, jak moc jsem byl úspěšný. Po prostudování výstupů výsledné syntézy jsem postupně doplňoval direktivy pro zřetězení cyklů. I další syntézy už proběhly v pořádku. Z reportu ze syntézy vyplývá, že vstup dat pro komprimaci je řešen přes paměť. Dále velikost dat je předávána přes port. Struktura potřebná pro komprimaci je z části řešena pomoci registru a z druhé části pomocí portů. Pomocí registru jsou také řešeny pomocné proměnné potřebné pro činnost kompresoru. Nakonec jsem provedl export do RTL. Ten ovšem skončil s varováním ohledně časování. Zde jsem nepřišel na způsob jak toto varování napravit. Prvky výsledné implementace jsou shrnuty v tabulce 6.1.

6.2.3 Implementace pro Virtex5

V dalším kroku jsem vyšel v projektu z části „solution2“ a vytvořil „solution3“. Znamená to, že jsou využity optimalizace z předešlé části. Syntéza proběhla úspěšně, ale následující export do RTL se mi nepodařilo realizovat. Chybové hlášení naznačovalo, že vybraný typ FPGA není podporován.

6.2.4 Implementace pro Virtex7

I při vytváření implementace pro Virtex7 jsem vyšel v projektu z části „solution2“. Tato část je v projektu v „solution4“. Syntéza proběhla úspěšně. Zde bylo v pořádku i exportování do RTL. Prvky výsledné syntézy jsou shrnuty v tabulce 6.1.

	Spartan3	Virtex7
SLICE	678	273
LUT	999	741
FF	760	647
BRAM	1	1

Tabulka 6.1: Prvky výsledných implementací.

Vyexportované zdrojové soubory v jazyce VHDL s výsledným řešení jsou na přiloženém CD disku. Stejně tak je na něm celý vytvářený projekt v programu Vivado HLS.

Kapitola 7

Závěr

Tato práce se zabývala komprimací odchyceného síťového provozu z páteřních sítí pro pozdější analýzu. Pro tento účel byl vybrán komprimační algoritmus LZW, protože se při komprimaci přizpůsobuje aktuálním datům. Při testování bylo zjištěno, že lze dosáhnout snížení velikosti odchyceného síťového provozu na 60 % až 70 % původní velikosti. Pokud ale síťový provoz bude obsahovat velké množství multimediálních dat, bude efektivita nižší. Komprimace se dokonce může i nevyplatit, protože výsledná data budou mít větší velikost. Tato negativní komprimace byla pozorována u datových sad připravených při sledování videí v internetovém prohlížeči.

Vivado HLS je velice rozsáhlé prostředí a myslím, že by bylo potřeba více času ke zdokonalení se s ním lépe pracovat. Při této práci jsem zjistil, že není jednoduché využít libovolný program v jazyce C a implementovat jej do hardware. Z mé zkušenosti vyplynulo, že je potřeba na tuto problematiku myslet už při vytváření programu v jazyce C. Tím by bylo možné se vyhnout například standardním funkcím, které nelze syntetizovat. I přesto se mi podařilo připravit tři experimentální implementace pro tři různá FPGA firmy Xilinx. Těmi jsou Spartan3 (xc3s400pq208-3), který je například na přípravku Fit-Kit, Virtex5 (xc5vlx155tff1136-3), který se používá například na síťových kartách a Virtex7 (xc7vx1140tflg1930-1), který patří do nejmodernější rodiny FPGA.

Dalším pokračováním práce by bylo tuto implementaci dále optimalizovat a následné ji využít při zachytávání síťových paketů na páteřní síti pro pozdější analýzu.

Literatura

[1] Antonenko, V.: clzw [online]. Dostupné z: `https://code.google.com/p/clzw/`, [cit. 2013-03-15].

[2] Drábek, V.: *Kódování a komprese dat KKO, Studijní opora*. Brno: FIT VUT v Brně, 2008.

[3] Jaroslav, R.; Ondřej, R.: *Přednáška 4: Spolehlivý přenos dat*. FIT VUT v Brně, 2011/12.

[4] Jaroslav, R.; Ondřej, R.: *Přednáška 5: Síťová vrstva (IPv4)*. FIT VUT v Brně, 2011/12.

[5] Jaroslav, R.; Ondřej, R.: *Přednáška 5: Síťová vrstva (IPv6)*. FIT VUT v Brně, 2011/12.

[6] Tye, C. S.; Fairhurt, G.: *A Review of IP Packet Compression Technique*. Scotland: PGNet, 2003, iSBN 1-9025-6009-4.

[7] Wikipedie: Paket [online]. Dostupné z: `http://cs.wikipedia.org/wiki/Paket`, 2013-04-07 [cit. 2013-04-09].

[8] Xilinx®: *Vivado Design Suite Tutorial: High–Level Synthesis*. Xilinx®, 2013, uG902 (v2013.1).

[9] Xilinx®: *Vivado Design Suite User Guide: High–Level Synthesis*. Xilinx®, 2013, uG910 (v2013.1).