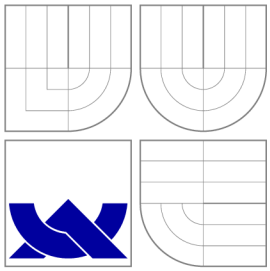
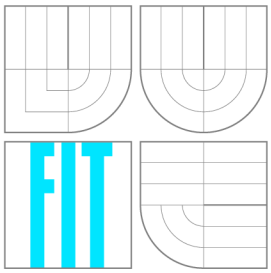


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ANALÝZY HTTP PROVOZU

ACCELERATION OF HTTP TRAFFIC ANALYSIS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAKUB BUDISKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KOŘENEK, Ph.D.

BRNO 2014

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2013/2014

Zadání bakalářské práce

Řešitel: **Budiský Jakub**
Obor: Informační technologie
Téma: **Akcelerace analýzy HTTP provozu**
Acceleration of HTTTP Traffic Analysis
Kategorie: Počítačové sítě

Pokyny:

1. Seznamte se s HTTP protokolem a nástroji pro efektivní analýzu HTTP provozu.
2. Nastudujte nástroje Catapult C a Vivado HLS, které slouží pro syntézu zdrojových kódů v jayzku C a C++ do technologie FPGA.
3. Navrhněte systém pro získávání URL a dalších zajímavých položek z HTTP provozu a navržený systém implementujte v jazyku C.
4. Ověřte korektnost implementace na dostupných vzorcích dat síťového provozu a změřte výkonnost takto vytvořené implementace na běžném procesoru.
5. Proveďte syntézu vytvořené implementace do technologie FPGA s využitím technologie Vivado HLS. Snažte se dosáhnout maximální možnou propustnost.
6. V závěru diskutujte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kořenek Jan, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2013
Datum odevzdání: 21. května 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Tato bakalářská práce se zabývá hardwarovou akcelerací analýzy nejrozšířenějšího protokolu na internetu, HTTP. Cílem je získat zajímavé položky z HTTP hlaviček a dosáhnout propustnosti potřebné k monitorování provozu na vysokorychlostních sítích. Navrhnutá softwarová implementace v jazyce C je optimalizována pro paralelní prostředí a poté převedena do hardwarové architektury s využitím vysokoúrovňové syntézy. Obě řešení jsou otestovány na vzorku dat z reálného provozu a je změřena jejich propustnost. Dosažené výsledky jsou diskutovány a na základě výsledků je navrhnuté další řešení.

Abstract

This bachelor thesis addresses hardware accelerated analysis of HTTP, the most used protocol on the Internet. The goal is to extract substantial information from the HTTP headers and to achieve throughput needed for monitoring high-speed networks. The C language is used to create a software implementation which is then optimized for parallel environment and transformed into a hardware architecture using High Level Synthesis. Both solutions, software and hardware one, are tested on real traffic samples and their throughput is measured. Achieved results are discussed and new solution is proposed on their basis.

Klíčová slova

HTTP, analýza, softwarově definované monitorování, vysokoúrovňová syntéza

Keywords

HTTP, analysis, Software Defined Monitoring, High Level Synthesis

Citace

Jakub Budiský: Akcelerace analýzy HTTP provozu, bakalářská práce, Brno, FIT VUT v Brně, 2014

Akcelerace analýzy HTTP provozu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kořenka, Ph.D.

.....
Jakub Budiský
20. května 2014

Poděkování

Rád bych poděkoval mému vedoucímu práce Ing. Janu Kořenkovi, Ph.D., za ochotnou pomoc, cenné rady a věcné připomínky.

Obsah

Kapitola 1

Úvod

Informačné technológie zohrávajú dôležitú úlohu v dnešnom živote. Denne ich výhody využívajú milióny ľudí po celom svete, či už v práci, školstve, alebo v súkromnom živote. Poskytujú prostriedky pre zvyšovanie efektivity v prakticky všetkých oblastiach ľudskej činnosti. Zjednodušujú množštvo bežných úloh, poskytujú priestor pre zábavu, prinášajú nové možnosti oddychu a umožňujú komunikovať novými spôsobmi.

Väčšina riešení a služieb, ktoré informačné technológie ponúkajú, je závislá na počítačových sietiach. Najvýznamnejšou a najväčšou z nich je celosvetová sieť internet. Neustále pribúda množstvo zariadení schopných internetového pripojenia a zvyšuje sa množstvo ľudí využívajúcich ich služby. Tiež je zaznamenávaný nárast času, ktorý ľudia strávia on-line.

Všetky tieto faktory vedú k zvyšujúcim sa nárokom na robustnosť a rýchlosť sieťovej komunikácie a podnecujú k vzniku nových sieťových technológií. Ďalším vedľajším faktorom je prudký nárast kriminality v tejto oblasti. Je to spôsobené množstvom príležitostí, ktoré vznikajú ako následok interakcie veľkého počtu ľudí. Tieto fakty vyúsťujú k potrebe lepších a účinnejších metód monitorovania sieťovej prevádzky.

HTTP tvorí dominantnú časť dát prenášaných internetom. Okrem toho, že poskytuje flexibilný a jednoduchý spôsob pre výmenu informácií, je prostriedkom mnohých druhov bezpečnostných hrozieb. Pre zložitosť jeho analýzy je problémovým protokolom pre spracovanie firewallmi a uniká mnohým monitorovacím riešeniam. Tieto riešenia sú prevažne softvérové a nemajú dostatočnú kapacitu na spracovanie množstva dátových tokov využívajúcich tento protokol. Cieľom tejto práce je pokúsiť sa adresovať tento problém hardvérovou akceleráciou monitorovania HTTP protokolu využitím technológie FPGA a nového prístupu k tvorbe obvodov, použitím vysokoúrovňovej syntézy.

Práca je rozdelená na 4 kapitoly. Kapitola 2 sa venuje popisu internetových protokolov so zameraním na HTTP, technikám pre monitorovanie vysokorýchlostných sietí a vysokoúrovňovou syntézou. V kapitole 3 je navrhnutý softvérový HTTP parser, ktorý je následne optimalizovaný za účelom prevedenia do logického popisu nástrojom vysokoúrovňovej syntézy. Kapitola sa ďalej zaoberá simuláciou vzniknutého hardvérového návrhu, diskutuje dosiahnuté výsledky a prezentuje možné smerovanie ďalšieho vývoja monitrovania HTTP protokolu. V poslednej kapitole 4 sa nachádza zhrnutie obsahu práce a dosiahnutých výsledkov.

Kapitola 2

Teoretický rozbor

2.1 Balík internetových protokolov TCP/IP

Balík intenretových protokolov (angl. *Internet protocol suite*) [1] je sieťový model pôvodne vyvíjaný agentúrou amerického ministerstva obrany DARPA. Dnes je celosvetovo najpoužívanejším sieťovým modelom použitým v rámci siete Internet. Na rozdiel od modelu OSI, ktorý je čiste konceptuálnym modelom [2], TCP/IP vznikol ako praktické riešenie a jeho súčasťou sú definície jednotlivých protokolov. Názov TCP/IP vznikol zo skratiek dvoch najdôležitejších a zároveň prvých definovaných protokolov v tomto štandarde. Sú to TCP (*Transmission Control Protocol*) a IP (*Internet Protocol*).

Účelom TCP/IP je sprostredkovať komukikáciu v rámci počítaťových sietí. Pre svoju zložitosť bol navrhnutý ako vrstevnatý model, kde každá vrstva využívajúc služby nižšej vrstvy poskytuje služby vrstve nad ňou. Protokoly týchto vrstiev ďalej obsahujú špecifikácu formátovania, adresovania, odosielania, smerovania a prijímania dát.

2.1.1 Architektúra TCP/IP

Požiadavky pre systémy implementujúce TCP/IP sú popísané v RFC 1122 [3] a RFC 1123 [4], pričom jednotlivé funkcie sú rozdelené do štyroch abstraktných vrstiev. V niektorých intepretáciách štandardu sa nachádza vrstiev päť, pričom za najnižšiu vrstvu sa považuje vrstva fyzická (hardvérová), tzn. vrstva prenosového média [5, 6]. Pôvodný model neobsahuje túto vrstvu a bol navrhnutý ako hardvérovo nezávislý.

Pre oddelenie jednotlivých abstraktných vrstiev používa TCP/IP princípy zapúzdrenia, čo zvyšuje modularitu a vytvára priestor pre rôzne kombinácie protokolov. Protokol nižšej úrovne nepozná protokol použitý vo vrstve vyššej, tento obsah je abstrahovaný ako ľubovoľné dáta a sú k nemu pripojené štruktúry typické pre daný protokol.

Linková vrstva

Linková vrstva zabezpečuje komunikáciu zariadení na lokálnej sieti, tzv. linke. Hranice takejto siete sú tvorené smerovačmi. Je najnižšou vrstvou modelu a môže byť implementovaná na ľubovoľnom hardvéri.

Funckie protokolov linkovej vrstvy:

- prenos jednotiek sieťovej vrstvy, paketov
- adresovanie zariadení na linke

- príprava dát na prenos fyzickým médiom
- determinizácia hraníc rámcov
- detekcia chýb

Špecifikácia TCP/IP popisuje princípy prekladu sieťovej adresy na adresu linkovú, akou je napríklad adresa MAC. Ostatné aspekty prenosu nie sú explicitne definované.

Na tejto vrstve môžu byť vybrané rámce posielané prostredníctvom virtuálnych sietí (VPN) alebo prostredníctvom iných tunelov. V tomto prípade sú rámce považované za aplikačné dáta, pričom sú posielané prostredníctvom inej siete. Použitým tunelovacím protokolom môže byť v prípade použitia TCP/IP siete protokol transportnej alebo aplikačnej vrstvy.

Sieťová vrstva

Sieťová, alebo aj internetová vrstva je zodpovedná za proces smerovania, čo je prenos dátových jednotiek, na tejto úrovni tzv. paketov, z počiatočnej do cieľovej siete. V balíku protokolov TCP/IP sa dnes jedná prevažne o protokoly IPv4 a IPv6.

Funckie IP protokolov:

- adresovanie klientov a sietí tzv. IP adresou
- smerovanie segmentov/datagramov medzi sieťami

Smerovače postupne preposielajú pakety ďalším smerovačom (tzv. *next-hop router*), bližším cieľovej sieti. Sieťová vrstva neposkytuje spoľahlivý prenos dát, ale tzv. doručenie s najlepším úsilím. V prípade potreby musí spoľahlivý prenos dát zabezpečiť niektorý z protokolov vyššej vrstvy.

Transportná vrstva

Protokoly transportnej vrstvy vytvárajú základný komunikačný kanál medzi procesmi komunikujúcich aplikácií. Bývajú implementované až na koncových zariadeniach, firewalloch, prípadne na zariadeniach zabezpečujúcich monitorovanie vyšších vrstiev. Najčastejšie využívanými protokolmi sú TCP a UDP. Prenášané jednotky sa nazývajú segmenty v prípade TCP, alebo datagramy v prípade UDP.

TCP je spojovo orientovaný protokol ktorý okrem identifikácie procesov, pomocou tzv. portov, zabezpečuje spoľahlivosť spojenia. To zahŕňa zoraďovanie, potvrdzovanie, retransmisiu a riadenie toku paketov podľa stavu sieťového spojenia. Tiež kontroluje integritu posielaných dát.

UDP je naproti tomu nespojovaný protokol ktorý okrem kontrolného súčtu a identifikácie procesov neposkytuje ďalšie služby. Má nižšiu réžiu, ale ak aplikácia vyžaduje spoľahlivý prenos, musí ho implementovať v rámci svojho aplikačného protokolu.

Aplikačná vrstva

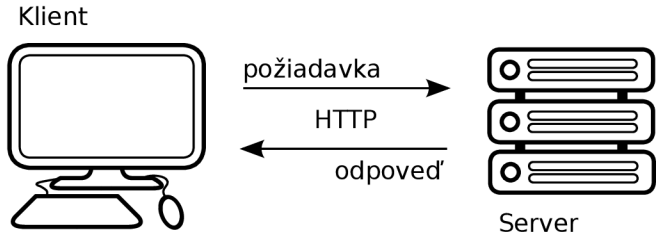
Najvyššia vrstva modelu TCP/IP, aplikačná, zahŕňa protokoly používané aplikáciami poskytujúcimi služby ich užívateľom. Používajú služby nižších vrstiev pre prenos užívateľských dát, zabezpečujú identifikáciu užívateľov aplikácie. Okrem toho môžu implementovať služby zabezpečujúce funkciu nižších vrstiev, ako napríklad smerovacie alebo konfiguračné protokoly.

2.2 Protokol HTTP

Hypertext Transfer Protocol (HTTP) [7, 8] je protokol aplikačnej vrstvy pre distribuované, kolaboratívne, hypermediálne informačné systémy. Pôvodne bol navrhnutý pre výmenu hypertextových dokumentov (Hypertext Markup Language, HTML). Je používaný v rámci World-Wide Web od roku 1990.

Prvá verzia, HTTP/0.9, bola jednoduchým protokolom pre prenos surových dát prostredníctvom Internetu [9], ktorá sa dnes už nevyužíva. HTTP/1.0 je vylepšená verzia protokolu podporujúca hlavičky vo formáte podobnom MIME (*Multipurpose Internet Mail Extensions*), obsahujúce metainformácie o prenášaných dátach a modifikátoroch sémantiky dotazov a odpovedí. Verzia HTTP/1.1 ďalej rozširuje tieto metainformácie, pričom zohľadňuje hierarchické proxy, kešovanie, potrebu perzistentných spojení a virtuálne servery. Tieto dve verzie prinášajú tiež niekoľko nových metód pre požiadavky.

HTTP protokol je protokol typu dotaz – odpoveď v architektúre klient – server (obr. 2.1). Klient použitím niektorej s dostupných metód odošle požiadavku pre získanie alebo modifikáciu zdroja umiestneného na serveri. Server odpovedá informáciou o stave spracovania požiadavky klienta, za ktorým typicky nasleduje telo s vyžiadanou informáciou. V rámci hlavičky sú tiež prenášané modifikátory požiadavky, metainformácie o tele správy či informácie o klientovi alebo o serveri.



Obr. 2.1: Princíp činnosti HTTP

Komunikácia je vo väčšine prípadov iniciovaná klientom. Od verzie HTTP/1.1 je v rámci jedného vytvoreného spojenia možné odoslať a prijať viac požiadavkov či odpovedí, čím sa znižuje réžia potrebná k vytváraniu spojení.

HTTP komunikácia najčastejšie prebieha nad sadou protokolov TCP/IP. Rezervovaným portom je TCP 80, ale môže používať aj iné porty, ako napríklad 8080 alebo 8008 [10]. Vo všeobecnosti môže byť HTTP obsah prenášaný nad akýmkoľvek sieťovým protokolom poskytujúcim spoľahlivý prenos, nakoľko tento vo vlastnej réžii neposkytuje.

V rámci tela požiadavku môžu byť odosielané prakticky ľubovoľné dáta, čo umožňuje podstatne širšie uplatnenie protokolu ako posielanie hypertextových dokumentov. Špecifikácia tiež umožňuje zadefinovať si vlastné metódy a vlastné položky hlavičiek, čo tiež zovšeobecňuje jeho použitie. HTTP server musí podporovať minimálne metódy `GET` a `HEAD`. V prípade, že server implementuje niektorú štandardnú metódu, tá musí byť implementovaná v súlade so štandardom.

Požiadavka na server začína prvým riadkom označovaným ako `Request-Line`. Tento riadok obsahuje metódu, URL (*Uniform Resource Locator*) oddelenú medzerami a reťazec obsahujúci verziu protokolu. Ako symbol nového riadku sa používa dvojica znakov `\r\n` (ich ASCII hodnota zodpovedá číslam 10 a 13).

Za týmto riadkom nasleduje niekoľko riadkov hlavičky. Tieto obsahujú názov položky nasledovaný dvojbodkou, ľubovoľným počtom tabulátorov alebo medzier a hodnotou. Hodnota je obvykle ukončená koncom riadku, ale úplná špecifikácia definuje syntax aj pre viacriadkové hodnoty. Obsahom hlavičky sú modifikátory požiadavky a metainformácie obsahu tela správy, ako napríklad hostiteľ dokumentu (doména), identifikácia klienta, požiadavky na kódovanie odpovede, kódovanie tela požiadavky, dĺžka obsahu v tele správy a ďalšie.

Hlavička končí prázdnym riadkom, teda sekvenciou `\r\n\r\n`. Po hlavičke nasleduje voliteľne telo správy, napríklad v prípade metódy `POST`, kedy sa dáta prenášajú od klienta na server. Na obrázku 2.1 sa nachádza príklad platnej HTTP požiadavky.

```
GET / HTTP/1.1\r\n
Host: example.com\r\n
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Firefox/31.0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
Accept-Language: sk,cs;q=0.8,en-us;q=0.5,en;q=0.3\r\n
Accept-Encoding: gzip, deflate\r\n
DNT: 1\r\n
Connection: keep-alive\r\n
\r\n
```

Výpis 2.1: Príklad HTTP požiadavky

Prvý riadok odpovede servera sa označuje ako `Reponse-Line`. Obsahuje verziu protokolu, stavový kód a jeho slovný popis. Syntax ani účel hlavičky odpovede sa nelíši od požiadavky, požiadavka a odpoveď sa však odlišujú množinou definovaných hlavičiek. V našom zjednodušenom príklade na obrázku 2.2 tiež vidno, že obsahuje prevažne metainformácie tela správy, ktoré nasleduje za hlavičkou. Telom odpovede je v našom prípade HTML dokument, v skrátenom príklade sú uvedené len jeho prvé dva riadky.

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
Date: Tue, 06 May 2014 20:24:19 GMT\r\n
Expires: Tue, 13 May 2014 20:24:19 GMT\r\n
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT\r\n
Server: ECS (iad/19AB)\r\n
X-Cache: HIT\r\n
Content-Length: 1270\r\n
\r\n
<!doctype html>\n
<html>\n
```

Výpis 2.2: Príklad HTTP odpovede

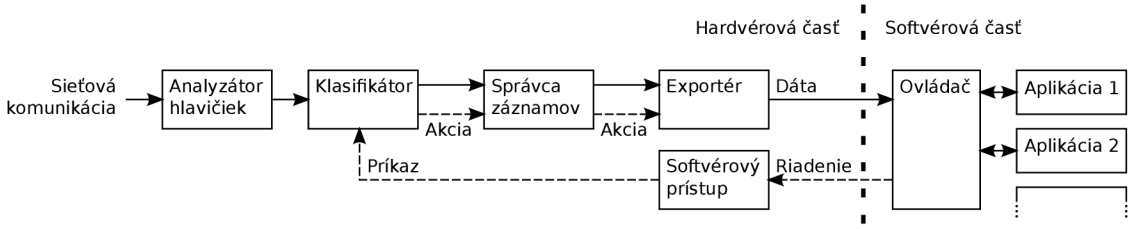
Implementácia HTTP je dnes dostupná na všetky bežné platformy a HTTP komunikácia je povolená vo väčšine sietí. Tieto fakty vyústili do významného podielu HTTP na prenose dát Internetom, pričom tvorí až 60% všetkých prenášaných dát [11]. HTTP protokol sa začal využívať v množstve aplikácií ako transportný protokol aplikačnej vrstvy, preto je protokol HTTP označovaný za nové TCP [12]. Tiež nie je zanedbateľné jeho využívanie

pre tunelovanie, tzn. zapúzdrenie inej komunikácie a jej prenos prostredníctvom HTTP protokolu.

Dominantné postavenie HTTP so sebou prináša bezpečnostné riziká. Vzhľadom na obmedzené zdroje a narastajúce množstvo prenesených dát Internetom je obtiažne komunikáciu HTTP protokolu monitorovať pomocou softvérových riešení. Preto sa treba zaoberať hardvérovou akceleráciou monitorovania tohoto protokolu.

2.3 Softvérovo definované monitorovanie

Softvérovo definované monitorovanie (angl. *Software defined monitoring*, SDM) [11] je systém softvérovo riadenej hardvérovej akcelerácie monitorovacích a bezpečnostných aplikácií zameraných na vysokorýchlostné siete. Funkcionalitu rozdeľuje medzi funkčnú hardvérovú a inteligentnú softvérovú časť, ktoré spolu spolupracujú. Firmvér SDM využíva pre klasifikáciu paketov tzv. sieťové toky – postupnosti paketov so spoločmými vlastnosťami prechádzajúce daným bodom siete za určitý interval.



Obr. 2.2: Schéma činnosti SDM

Na obrázku 2.2 sa nachádza zjednodušená verzia architektúry SDM. Analyzátor hlavičiek sa stará o extrakciu informácie z hlavičiek paketu. Získané informácie posiela v dohodnutom formáte klasifikátoru, ktorý ich využije na zaradenie do toku. Tiež vyberie zodpovedajúce pravidlá aplikovateľné na daný paket, kotrými sa ďalej riadi správca záznamov a exportér. Správca záznamov v prípade potreby zaznamenáva štatistiky o tokoch, akými sú počet paketov a ich trvanie. Exportér posiela vyžiadané pakety v požadovanom formáte ďalej pre softvérové spracovanie a v prípade, že nie sú potrebné, ich zahodí. Aplikácie využívajú dostupné aplikačné rozhranie ovládača pre pridávanie či modifikáciu pravidiel a pre získavanie potrebných tokov z hardvéru.

Cieľom SDM je znížiť nároky na softvér a to tvorbou niektorých štatistík v hardvéri a aplikačne riadenou stratou informácie. Nezaujímavé informácie sú vyfiltrované z dát určených pre spracovanie softvérom, čím sa zvýši celková efektivita a výkonnosť spracovania.

Jednou z testovaných aplikácií bola analýza aplikačného protokolu HTTP. Použitým analyzátorom bol HTTP plugin pre systém `FlowMon` [13] a úlohou hardvéru bolo odfiltrovať dáta nezaujímavé z pohľadu tejto aplikácie. Získavanými informáciami sú niektoré zaujímavé položky z HTTP hlavičiek. Redukcia dát bola dosiahnutá pomocou dvoch praktík. Prvou bola filtrácia založená na použitej protokolovej sade, kedy väčšina HTTP prevádzky prebieha s využitím transportného protokolu TCP na porte 80. Druhou bola filtrácia tokov, pre ktoré už bola zachytená HTTP hlavička, tzn. informácie boli extrahované len z prvej hlavičky daného toku.

Popísaným spôsobom bola dosiahnutá redukcia 75% počtu paketov a 73% objemu dát. Extrakciou požadovaných dát priamo pomocou hardvéru je len ďalším krokom k zníženiu

množstva dát, ktoré je nutné spracovať softvérovo. Výsledná architektúra by mohla byť použitá ako modul do hardvérovej časti – analyzátora – systému SDM.

2.4 Vysokoúrovňová syntéza

Vysokoúrovňová syntéza obvodov (angl. *High Level Synthesis*, HLS) [14, 15] pre ASIC a FPGA čipy je novým prístupom k návrhu, syntéze a verifikácii týchto obvodov. Cieľom je vytvoriť RTL (*Register Transfer Level*) implementácie z vysokoúrovňového algoritmického popisu. Tento býva dostupný vo forme implementácie programu, pre ktorý sa často využíva niektorý z dostupných jazykov nižšej úrovne (napríklad jazyk C).

Postupom času sa začala komplexnosť potrebných obvodov výrazne zvyšovať a návrhári čelia stále obtiažnejším úlohám pri vytváraní a hľadaní optimálnej architekúry. S narastajúcou zložitosťou tiež stúpa množstvo chýb, ktorých sa návrhári dopúšťajú. Tieto fakty výrazne prispievajú k nárastu dĺžky vývojového cyklu.

HLS rieši tieto problémy automatizáciou procesu plánovania a mapovania operácií, čím premosťuje fázu manuálnej tvorby logického popisu zo zadaného algoritmu. Odpadajú problémy týkajúce sa cieľovej technológie, časovania, procesov či ich hierarchie. Implementácia je prenositeľná, ľahšie udržiavateľná a modifikovateľná, nakoľko programovanie v známych a rozšírených algoritmických jazykoch je výrazne jednoduchšie. Úroveň paralelizmu je ľahko nastaviteľná prostredníctvom užívateľského rozhrania. HLS sa tiež snaží zjednodušiť proces verifikácie a poskytuje prostriedky pre verifikáciu vzniknutej architektúry voči preloženému kódu bežiacemu na procesore. Najpoužívanejšími jazykmi používané syntetizačnými nástrojmi tohoto typu sú rozšírené C, C++ alebo SystemC [16].

2.4.1 Obmedzenia a aspekty HLS

Pre dosiahnutie lepších výsledkov (rýchlosť, plocha na čipe) je potrebné upraviť zdrojové kódy tak, aby použité konštrukcie nebránili syntetizačnému nástroju paralelizovať jednotlivé operácie. Vzhľadom na to, že plánovanie operácií vychádza z dátových závislostí, programátor musí venovať zvýšenú pozornosť tomu, aby ich zbytočne nevytváral. Ďalej je nutné zaobísť sa bez podpory dynamickej alokácie či ďalších služieb operačného systému, akými sú vlákna alebo semafory.

V tejto práci je popisovaný a použitý nástroj `Catapult C`, ktorý je k dispozícii k použitiu v univerzitnej verzii.

Dátové typy

Základné dátové typy jazyka C nepostačujú pre popis algoritmu v hardvéri, a to hneď z dvoch dôvodov. V technológii FPGA je možné si zvoliť dátové šírky signálov s bitovou presnosťou. Umožňuje to implementovať operácie pracujúce s týmito dátami čo najefektívnejšie, nakoľko operácie s menšími dátovými šírkami sú častokrát rýchlejšie a zaberajú menšie množstvo prostriedkov. Napríklad na popis štyroch hodnôt stačia v závislosti na zvolenom kódovaní dva až štyri bity informácie, v softvéri by tieto bity boli uložené v dátovom type širokom typicky 32 alebo 64b.

Ďalej je nutné definovať operácie nad dátami s bitovou presnosťou, ktoré budú ekvivalentné možnostiam signálov v RTL. Toto zahŕňa prácu s ľubovoľnými bitmi informácie a možnosť vytvárať nové signály ich delením či konkatenáciou. Tiež je potrebné nad takýmito dátovými typmi vykonávať logicko matematické operácie.

Použitý nástroj, `Catapult C`, používa na pokrytie týchto požiadaviek jazyk C++ a šablóny tried zvané `AC Datatypes`[1], ktoré implementujú funkčnosť dátových typov s bitovou presnosťou.

Bloky a cykly

V nástrojoch využívajúcich HLS je možné stupeň paralelizácie veľmi dobre nastaviť na úrovni blokov a cyklov. Bloky sú samostatné jednotky, ktoré vykonávajú svoju činnosť nezávisle na svojom okolí, a so svojím okolím, ostatnými blokmi, komunikujú pomocou rozhraní. Každý blok je reprezentovaný jeho funkciou. Všetky funkcie, ktoré sú súčasťou jedného bloku, sú spracované ako `inline` funkcie, tzn. ich telo je rozbalené na miesto, z ktorého je funkcia volaná. Ktoré funkcie budú syntetizované ako samostatné bloky a aké rozhranie bude pre ne použité sa určuje v rámci nastavení obmedzení architektúry. V dostupnej verzii použitého nástroja funkcia vytvárania blokov nie je dostupná, a zo zdrojového kódu je z určenej funkcie vytvorený jeden, hlavný blok.

Cykly sú ďalším dôležitým nástrojom pre zvyšovanie paralelizmu. K dispozícii sú voľby pre rozbalenie cyklu, úplné alebo čiastočné, ako aj možnosť spracovanie cyklu zreťaziť s predom špecifikovanou priepustnosťou. Okrem explicitných cyklov sa za cyklus považuje aj hlavná syntetizovaná funkcia, pretože po skončení výpočtu v jej tele je automaticky zahájený nový výpočet.

Pamäť a jej adresovanie

Realizácie pamäte v technológii FPGA závisí na jej návrhárovi, ktorý má na výber z niekoľkých typov pamätí, či už externých alebo realizovaných priamo v FPGA čipe. Túto pamäť je ale potrebné nejakým spôsobom adresovať.

Pamäť je z pohľadu HLS realizovaná pomocou poľa a je adresovaná pomocou indexov. Indexy teda nahrádzajú ukazatele a šírka použitého dátového typu určuje šírku adresovaného slova pamäte. V rámci `AC Datatypes` je dostupná šablóna pre výpočet logartimu so základom dva v čase prekladu. To umožňuje správne určiť dátové typy potrebné pre adresovanie vytvorených pamätí. Technológia použitej pamäte sa určuje v rámci nastavení obmedzení architektúry.

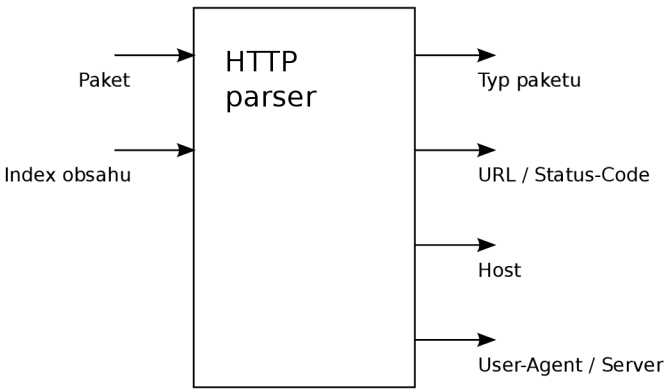
[1] http://calypto.com/en/page/leadform/38, dostupné po registrácii

Kapitola 3

Architektúra pre analýzu HTTP

Analýza HTTP ako protokolu aplikačnej vrstvy je kľúčová z pohľadu bezpečnosti sietí. Tento protokol tvorí nadpolovičnú väčšinu dátovej prevádzky prenášanej prostredníctvom celosvetovej siete internet a treba sa zaoberať akceleráciou jeho analýzy. Analyzátor v rámci architektúry SDM už disponuje nástrojmi pre analýzu protokolov nižších vrstiev. Vďaka jeho modulárnosti sa stáva vhodným prostredím pre rozšírenie tejto funkcionality a implementáciu parsera HTTP ako novú komponentu jeho architektúry. Je teda možné sústrediť sa výlučne na HTTP protokol, čo tiež prispieva k prenositeľnosti implementácie.

V rámci akcelerácie analýzy HTTP prevádzky nás zaujíma obsah hlavičky paketu, resp. jeho najdôležitejšie časti. Je potrebné ich lokalizovať a extrahovať pre budúcu hlbšiu analýzu (či už softvérovú alebo hadvérovú) a prípadné použitie ako kľúča pre filtráciu nežiadúcej HTTP prevádzky.



Obr. 3.1: Rozhranie parsera

Na obrázku 3.1 je popísané rozhranie navrhovanej architektúry. Vstupom je paket samotný a index na obsah paketu. Ten ukazuje na prvý oktet nachádzajúci sa za TCP halvičkou. Tento prístup je zvolený, pretože kopírovanie obsahu do samostatnej pamäte je časovo náročná operácia. Tiež je možné vstup dopredu predspracovať, napríklad obmedziť parsovanie len na špecifické porty protokolu TCP. O to sa môže postarať existujúci analzyátor TCP, ktorý tiež môže poskytnúť potrebný index obsahu.

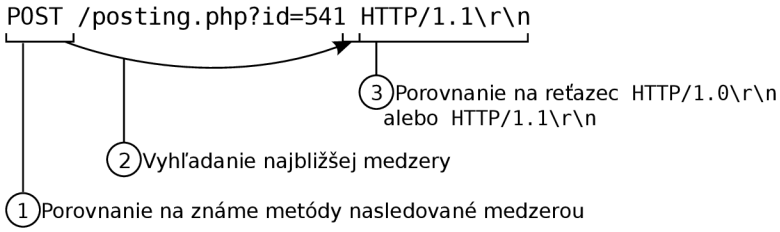
Medzi monitorované položky hlavičiek vybraných pre túto prácu patrí metóda, URL spolu s položkami `User-Agent` a `Host` pre požiadavku, stavový kód a položku `Server` pre odpoveď servera. Či sa jedná o HTTP požiadavku, odpoveď, alebo nerozpoznaný paket, je

zakódované vo výstupe „Typ paketu". Tiež je ním daná použitá metóda v prípade požiadavky. V rámci URL je možné odhaliť množstvo útokov na server, akými sú zakódované skripty či cielené dotazy na databázový server [17]. Položky User-Agent a Server nám prezrádzajú identitu komunikujúcich strán, respektíve identitu za ktorú sa vydávajú. User-Agent v prípade robotov často používa rozpoznateľný opakujúci sa vzor. Položka Host nám v prípade HTTP/1.1 prezradí informáciu o serveri, na ktorý je potenciálny útok cielený. Tieto položky je potrebné v pakete lokalizovať, a ich polohu alebo obsah vrátiť v rozumnom formáte.

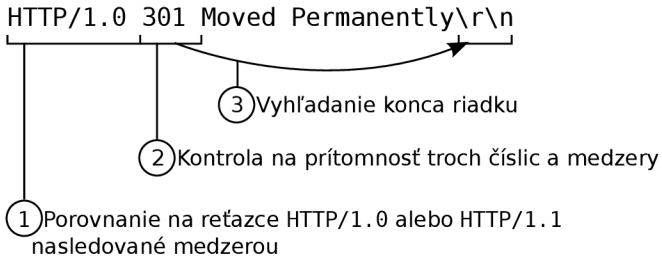
3.1 Softvérová implementácia

Aby bolo možné využiť HLS k návrhu hardvérovej architektúry, je nutné vytvoriť jej ekvivalentnú softvérovú implementáciu. Tá bola vytvorená v jazyku C s ohľadom na nasledujúce mapovanie do technológie FPGA.

Funkcia parsera sa dá rozdeliť do dvoch častí. Prvá časť sa stará o parsovanie prvého riadku, a rozhoduje o tom, či paket obsahuje HTTP hlavičku. Postupy možno vidieť na obrázkoch 3.2 a 3.3. Výstup pre URL a stavový kód je generovaný vždy po kroku 1 (začiatok) a 2 (koniec). Druhá časť parsuje ďalšie riadky hlavičky, a v prípade nájdenia požadovanej položky zaznamená začiatok a koniec hodnoty. Princíp možno nájsť na obrázku 3.4. Výstupy sú generované po kroku 2 a 3, nájdením prázdneho riadku parsovanie končí (koniec halvičky).



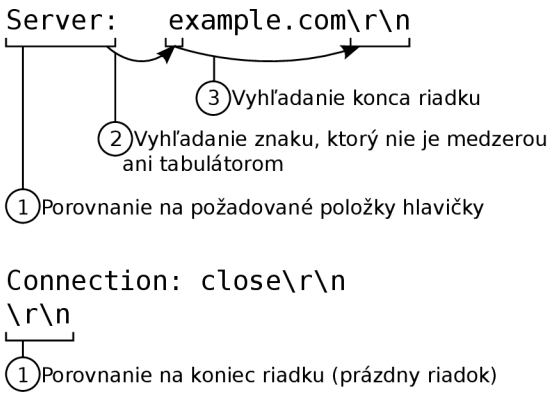
Obr. 3.2: Princíp parsovania `Request-Line`



Obr. 3.3: Princíp parsovania `Response-Line`

Pre testovanie softvérovej implementácie bol použitý záznam reálnej prevádzky, z ktorého boli pomocou programu `Wireshark`[1] vyfiltrované HTTP pakety. Vstup zabezpečuje knižnica `libpcap`[2] z uloženého `pcap` súboru. Výsledky sú vrátené ako ukazatele na začiatok

[1] http://www.wireshark.org
[2] http://www.tcpdump.org



Obr. 3.4: Princíp parsovania ďalších položiek hlavičky

a koniec požadovaných položiek, pričom za účelom verifikácie sú vypisované na štandardný výstup. Pre meranie času výpočtov boli použité funkcie `getrusage`[3] a `gettimeofday`[4] dostupné ako súčasť knižnice GNU C.

Ďalším vstupom, ktorý bolo treba zabezpečiť, sú idexy na začiatok obsahu paketu, tzn. index prvého oktetu HTTP protokolu. Táto informácia bola získaná z detailov paketov, ktoré je možné z programu `Wireshark` vyextrahovať vo formáte xml. Pre spracovanie XML boli použité dostupné nástroje interpretu bash (výpis 3.1). Vstupom je xml súbor `http.xml`, výstupom je textový súbor `http.idx`, ktorý na každom riadku obsahuje index na obsah zodpovedajúceho paketu. Tento index je zistený súčtom indexu začiatku hlavičky TCP a jej dĺžkou. Pre porovnanie získaných výsledkov pri verifikácii funkčnosti bol taktiež použitý detailný výstup programu `Wireshark`.

```
grep '<proto name="tcp"' http.xml | \
sed -E 's/^.+size="([0-9]+)".+pos="([0-9]+)".+$/\1 + \2/g' | \
bc > http.idx
```

Výpis 3.1: Príkaz pre extrakciu indexu obsahu z xml súboru

Po otestovaní funkčnosti nasledoval test priepustnosti takto naimplementovaného parsera. Meraným časom bol čas strávený vo funkcii pre parsovanie HTTP paketu. Vstupom boli výlučne pakety obsahujúce HTTP hlavičku (najhorší prípad). Parser bol testovaný v prostredí operačného systému `GNU/Linux`, verzia jadra 3.14. Použitým prekladačom bol `gcc` verzie 4.8.2. Výpočty boli vykonávané na mobilnom procesore strednej triedy `Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz`. Výsledky merania na piatich vzorkách dát je možné nájsť v tabuľke 3.1.

Priemerná priepustnosť parsera bola 963,85 kp/s (kilopaketov za sekundu), resp. 6,88 Gb/s. Nakoľko sa SDM snaží zamerať na ethernetové siete s rýchlosťou 100 Gb/s, nejedná sa o postačujúci výsledok napriek faktu, že HTTP hlavičky sa nachádzajú len v časti HTTP paketov. Ak by sa, napríklad vplyvom distribuovaného útoku, zvýšilo množstvo hlavičiek, nebola by rýchlosť spracovania dostatočná. Okrem toho je procesor využívaný inými aplikáciami potrebnými pre beh systému.

[3] http://pubs.opengroup.org/onlinepubs/9699919799/functions/getrusage.html
[4] http://pubs.opengroup.org/onlinepubs/9699919799/functions/gettimeofday.html

Číslo vzorky	Počet paketov [kp]	Veľkosť [MB]	Čas výpočtu [ms]	Priepustnosť [kp/s]	Priepustnosť [Gb/s]
1	81,77	71,28	83,16	983,32	6,86
2	78,56	68,84	81,24	966,95	6,80
3	78,01	69,77	82,13	949,80	6,80
4	79,94	71,60	81,68	978,72	7,01
5	71,24	63,95	75,75	940,48	6,75

Tabuľka 3.1: Priepustnosť softvérového parsera

3.2 Syntéza do technológie FPGA

Softvérová implementácia bola od začiatku písaná s ohľadom na neskoršiu snahu o jej automatizovaný prevod do RTL. Využíva minimálne množstvo knižníc a nespolieha sa na dynamickú alokáciu pamäte. Napriek tomu bolo nutné vykonať niekoľko úprav, aby bol kód syntetizovateľný.

Knižnica `libpcap` je stále prítomná, ale ako súčasť verifikačného procesu a nie je syntetizovaná do výslednej architektúry. Funkcie pre meranie času boli z kódu vylúčené, nakoľko čas výpočtu bude získaný zo simulácie.

Jazyk bol zmenený na C++ z dôvodu použitia potrebných šablón. Nebolo však použité objektovo orientované paradigma a štruktúra programu bola zachovaná. Tiež nie sú využívané žiadne ďalšie pokročilé funkcie jazyka C++.

3.2.1 Paralelizácia procesu

Hlavnou výhodou hardvérového spracovania spočíva vo využití obvodov špecificky navrhnutých pre daný účel a možnosť paralelizácie jednotlivých výpočtov. Paralelizácia pritom tvorí hlavný ovplyvniteľný faktor akcelerácie, okrem nej sú vlastnosti výsledného obvodu závislé už len na cieľovej technológii. Nakoľko sa jedná o parser textového protokolu, budeme sa zameriavať hlavne na porovnávanie reťazcov a vyhľadávanie v reťazci. Zvážiť musíme ale aj technologický faktor ovplyvňujúci možnosti paralelizácie, a tým je prístup k potrebným dátam.

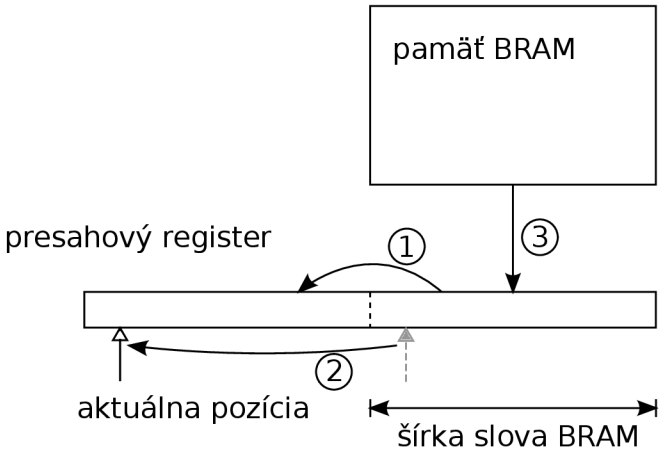
Prístup k pamäti

Keďže veľkosť paketu sa bežne pohybuje rádovo v kB až MB, nie je možné umiestniť ho do registru. Pre vstup musí byť teda použitá niektorá z dostupných foriem RAM pamäťí.

Z pamäte je možné čítať a zapisovať dáta len prostredníctvom jej rozhrania. Týmto je výrazne ovplyvnená mieru paralelizmu, pretože je z nej možné v jednom okamihu čítať a zapisovať len toľko slov, koľko je dostupných rozhraní. Toto obmedzenie by zamedzilo väčšine paralelných procesov popísaných nižšie, a preto je nutné ho obísť.

Výhodou BRAM (*Block Random Access Memory*), ktorá býva súčasťou FPGA čipov, je fakt, že je možné si určiť šírku dátového slova [18]. Je teda možné použiť širšie slovo obsahujúce niekoľko oktetov, ktoré by umožnilo jednoduché spracovanie. Použitie šírky slova menšej ako porovnávané reťazce by mohlo viesť k nutnosti načítať z pamäte predchádzajúce slová, čomu je vhodné sa vyhnúť. Šírku slova tiež ovplyvňujú potreby požadovaných paralelných procesov a tým požiadavky na cieľovú priepustnosť.

BRAM typicky obsahuje dve rozhrania, ale jedno z nich je vhodné nechať k dispozícii pre časť architektúry zodpovednej za zápis. Druhé z rozhraní využijeme pre načítanie slov z pamäte do registru, ku ktorému je, na rozdiel od pamäte, možné pristupovať paralelne v celej jeho šírke. Problémom takejto pamäťovej architektúry je komplikovanosť operácií vykonávaných na hranici slov pamäte. To sa dá vyriešiť načítaním istého presahu, a nakoľko je možné z pamäte čítať len po slovách, vhodnou veľkosťou sú práve dve slová.



Obr. 3.5: Princíp presahového registra

Na obrázku 3.5 sa nachádza návrh funkčnosti presahového registra. Jeho šírka je ekvivalentná dvom pamäťovým slovám. Postupnosť oktetov je z ľava doprava, a teda v tomto smere je postupne spracovávaná. V kóde je na potrebných miestach vložená rutina, ktorá kontroluje, aký je index aktuálne spracovaného oktetu v rámci registru. Ak je zistené, že sa spracúvajú dáta v druhej polovici registra, je zabezpečené jeho doplnenie.

Doplnenie sa skladá z troch krokov:

1. presunutie slova z vrchnej polovice registra do spodnej
2. korekcia indexu do registra jeho znížením o dĺžku slova
3. načítanie nového slova do vrchnej polovice registra

V prípade, že sa v pamäti už nenachádza ďalšie slovo paketu, je vrchná polovica registra inicializovaná nulami. Pri ďalšej snahe o načítanie nasledujúceho slova parsovanie paketu končí.

Porovnávanie reťazca

V rámci parsera je nutné rozpoznať metódy, verziu protokolu či jednotlivé názvy hlavičiek. K tomu je nutné porovnávať zachytené dáta s predom nadefinovanými reťazcami. Jedná sa teda o porovnanie konštantného reťazca s obsahom pamäte na zadanom mieste. Výpis 3.2 obsahuje možný spôsob zápisu takejto funkcie v jazyku C++.

V cykle dochádza k postupnému porovnaniu jednotlivých znakov reťazca, `ri` je počiatočná adresa presahového registra `cache`, od ktorej chceme porovnávať. Dátový typ `regCache` je celočíselný bezznamienkový typ s bitovou prensosťou o veľkosti dvoch slov pamäte, `rcIndex` je typ dostatočne široký na adresovanie oktetov v rámci `regCache`. Funckia `o2b` prepočítava index oktetu na index bitu.

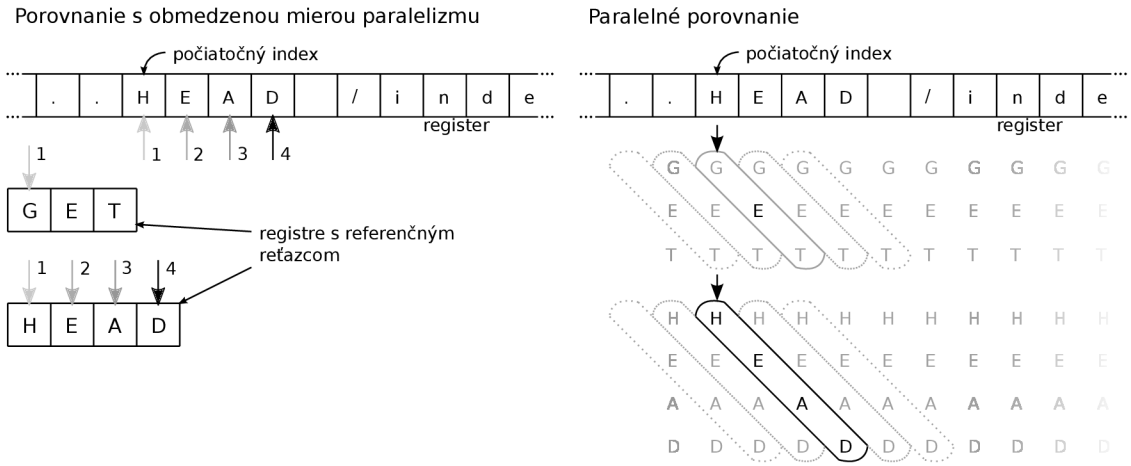
```
static inline bool checkForSubstr(rcIndex ri, regCache& cache,
                                  const char* str, const int size)
{
    for (int i = 0; i < size; i++) {
        if (cache.slc<8>(o2b(ri + i)) != str[i])
            return false;
    }
    return true;
}
```

Výpis 3.2: Porovnanie reťazca

V rámci parsera je táto funkcia volaná s konštantnými parametrami `str` a `size` a v rámci plánovania bude považovaná za inline funkciu aj v prípade, že by takto nebola deklarovaná. Cyklus má teda známy počet opakovaní už v čase prekladu, čo zohráva dôležitú úlohu v rámci možností paralelizácie.

V hardvéri môže byť tento cyklus vykonaný sekvenčne, podobne ako v softvéri, ale pre dosiahnutie vyššej rýchlosti je lepšie využiť možnosti paralelného spracovania a operácie porovnania sa pokúsiť vykonať naraz. Predpokladom pre úplné rozbalenie cyklu je okrem známeho počtu opakovaní nutnosť paralelného prístupu ku všetkým prvkom, ako reťazca, tak porovnávanej pamäte. Napriek tomu, že je táto podmienka splnená a rozbalenie cyklu je vynútené v rámci nastavení obmedzení architektúry, dostupný nástroj operáciu porovnania síce rozbalí, ale nenaplánuje plne paralelne.

Na obrázku 3.6 je vľavo znázornený spôsob porovnania, ako ho použitý nástroj naplánoval. To prebieha od počiatočného indexu jeho postupnou inkrementáciou a porovnaním znaku na ktorý ukazuje so znakom na zospovedajúcej pozícii v referenčnom reťazci. Istú mieru paralelizmu možno pozorovať na porovnaní prvého znaku, kedy sú porovnané znaky z oboch referenčných reťazcov. V pravej časti obrázku je znázornené riešenie paralelného porovnania všetkých znakov. Tento prístup si vzhľadom na dopredu neznámu pozíciu začiatku vyžaduje veľké množstvo zdrojov (komparátorov a logických ciest), pretože znaky na všetkých pozíciách treba porovnať so všetkými znakmi hľadaných reťazcov.



Obr. 3.6: Možné spôsoby porovnávania reťazcov

Pre otestovanie paralelného prístupu bola skúšobne naimplementovaná verzia parsera, ktorá porovnáva len na začiatku presahového registra. Výsledok bol napriek plne paralelnému porovnaniu výrazne pomalší z dôvodu réžie spôsobenej častou a komplikovanou operáciou posunu.

V použitej verzii funkcie bola pridaná možnosť pre porovnanie bez ohľadu na veľkosť písmen. Táto požiadavka vyplýva z faktu, že jednotlivé názvy položiek hlavičky HTTP môžu byť vo forme malých, veľkých písmen alebo ľubovoľnej ich kombinácie. Táto funkčnosť bola zabezpečená funkciou logického súčtu s číslom 32, ktorý využíva vlastnosti ASCII tabuľky a transformuje text na malé písmená. Okrem písmen je v takejto transformácii bezpečné používať ASCII znaky 32 – 63, čo pokrýva aj potrebné znaky pomlčky a dvojbodky.

Vyhľadávanie v reťazci

Pre parsovanie HTTP je kritické vyhľadávanie niektorých znakov resp. postupností, a to hlavne

- znaku medzery, pre nájdenie konca URL
- znaku, ktorý nie je medzerou ani horizontálnym tabulátorom, pre nájdenie začiatku hodnoty
- postupnosti znakov konca riadku `\r\n`, pre nájdenie konca hodnoty

Všetky tieto vyhľadávania majú veľmi podobný charakter, a preto je vhodné zamerať sa na najjednoduchší z nich, a to hľadanie medzery. Vyhľadávanie konca riadku a „nebieleho" znaku bude prebiehať podobným spôsobom s mierne odlišnou podmienkou.

Paralelizácia vyhľadávania je založená na možnosti porovnávania viacerých oktetov (znakov) naraz. Vo všeobecnosti je možné úlohu vyhľadávania popísať nasledovne:

- vyhľadávame postupnosť znakov (napríklad medzeru)
- vyhľadávame paralelne v registri širokom n znakov
- požadujeme prvý výskyt za aktuálnym indexom k ($k < n$)

```
rcIndex findSP1(rcIndex ri, regCache cache)
{
    FIND_SP: for (; ri < wordWidth; ri++)
    {
        if (cache.slc<8>(o2b(ri)) == '␣')
            return ri;
    }
    return wordWidth;
}
```

Výpis 3.3: Vyhľadávanie v reťazci I

Výpis 3.3 obsahuje jednoduchý kód pre hľadanie medzery, ktorý je variáciou algoritmu použitého v softvérovej implementácii. Funkcia parametrami prijíma aktuálnu pozíciu `ri` a register `cache`. Použité dátové typy sú bezznamienkové celé čísla s dátovou šírkou zodpovedajúcou ich funkcii. V prípade neúspechu je vrátená hodnota mimo rozsah slova. Tento

kód však nie je optimálnym riešením pre HLS. Asi najväžnejší problém je fakt, že cyklus nemá spodnú hranicu. Má síce maximálny počet iterácií vyplývajúci z použitého dátového typu, ale takýto cyklus sa nedá úplne rozbaliť. To tvorí pre proces plánovania len ťažko prekonateľnú prekážku. Pri snahe rozbaliť ho 64-krát sme čelili problémom s dĺžkou extrakcie konečného automatu pre riadenie obvodu. Hlavným problémom je však nekvalitný výsledok, pretože jednotlivé iterácie rozbaleného cyklu neboli naplánované paralelne. Použitý nástroj naplánoval cyklus do siedmych hodinových taktov, pričom 6 taktov je latencia obvodu a jeden takt bol využitý pre uloženie výsledku.

```
rcIndex findSP2(rcIndex ri, regCache cache)
{
    FIND_SP: for (unsigned int i = 0; i < wordWidth; i++)
    {
        if (i < ri)
            continue;
        if (cache.slc<8>(o2b(i)) == '␣')
            return i;
    }
    return wordWidth;
}
```

Výpis 3.4: Vyhľadávanie v reťazci II

Problému s rozbalením sme sa snažili vyhnúť použitím upravenej implementácie s rovnakou sémantikou (výpis 3.4). Výsledok plánovania operácií z takéhoto kódu sa však výrazne nelíši od prvého pokusu. Cyklus síce je možné úplne rozbaliť, použitý nástroj ho ale nenaplánoval paralelne. Naopak, pribudli zdroje za použitie samostatného čítača a prejavil sa pokles priepustnosti za dodatočné priradenie. Cyklus bol po rozbalení naplánovaný do 17-tich taktov, z čoho jeden opäť slúžil pre uloženie výsledku.

Po niekoľkých ďalších pokusoch bol použitý popis v jazyku C++, ktorý zodpovedá presnej predstave o výslednom obvode a nachádza sa vo výpise 3.5. Algoritmus pracuje v štyroch krokoch. V cykle `FIND_SP` sú nasjkôr nájdené výskyty medzier, ktoré sú zaznamenané v príznakovom registri `flags`. V druhom kroku `MASK_GEN_SP` je z aktuálnej pozície vytvorená maska, ktorou je príznakový register v nasledujúcom cykle `MASK_SP` vymaskovaný. Týmto si zaručíme hľadanie len od požadovanej pozície. Posledným krokom je `ENC_SP`, cyklus, ktorý plní funkciu prioritného kodéra.

Kód je na rozdiel od predchádzajúcich prípadov spracovaný rýchlo, vrátane verzie s rozbalenými cyklami, a je vykonávaný paralelne. Napriek jeho neefektivite v softvéri, priepustnosť vzniknutého riešenia dosahuje požadovaných výsledkov. Latencia výsledného obvodu je jeden takt a ďalší takt zaberá uloženie výsledku. Odhadované množstvo zdrojov je asi 40-krát nižšie, ako v predchádzajúcich dvoch prípadoch.

Uvedené príklady sú zamerané na proces plánovania operácií. Výsledný kód je rozšírený o prácu s presahovým registrom (v prípade neúspechu v aktuálnom slove sa načíta ďalšie) a s hodnotou indexu manipuluje priamo. Na výsledok plánovania to nemá vplyv a zjednodušený kód je vhodnejší pre ilustráciu. Podobným spôsobom je naimplementované vyhľadávanie nového riadku a preskakovanie bielych znakov, líšia sa len podmienkou pre nastavenie premennej `flags`.

```
rcIndex findSP3(rcIndex ri, regCache cache)
{
    bool flags[wordWidth] = {false};
    bool mask[wordWidth] = {false};

    FIND_SP: for (unsigned int i = 0; i < wordWidth; i++)
    {
        if (cache.slc<8>(o2b(i)) == '␣')
            flags[i] = true;
    }
    mask[ri] = true;
    MASK_GEN_SP: for (unsigned int i = 1; i < wordWidth; i++)
    {
        mask[i] = mask[i] || mask[i-1];
    }
    MASK_SP: for (unsigned int i = 0; i < wordWidth; i++)
    {
        flags[i] = flags[i] && mask[i];
    }

    rcIndex i;
    ENC_SP: for (i = 0; i < wordWidth; i++)
    {
        if (flags[i])
            break;
    }
    return i;
}
```

Výpis 3.5: Vyhľadávanie v reťazci III

3.2.2 Výsledok syntézy

Použitím vyššie popísaných postupov pre zabezpečenie paralelizácie bola softvérová implementácia transformovaná do kódu, ktorý bol následne naplánovaný a namapovaný nástrojom `Catapult C`. Funkčnosť parsera v C++ bola overená, podobne ako v softvérovej verzii, na dostupnej vzorke dát. Samotná verifikácia RTL prebiehala automatizovane počas simulácie programom `ModelSIM`. Syntézu do FPGA a presnejší odhad zdrojov zabezpečil nástroj `Precision RTL`.

Simulované boli dve varianty s rôznou šírkou slova, a to 64-bajtová a 32-bajtová verzia. Výsledky sa nachádzajú v tabuľkách 3.2 a 3.3.

Číslo vzorky	Počet paketov [kp]	Veľkosť [MB]	Čas výpočtu [ms]	Priepustnosť [kp/s]	Priepustnosť [Gb/s]
1	81,77	71,28	113,89	717,98	5,01
2	78,56	68,84	104,60	750,98	5,27
3	78,01	69,77	103,93	750,61	5,30
4	79,94	71,60	112,67	709,55	5,08
5	71,24	63,95	106,66	667,95	4,80

Tabuľka 3.2: Priepustnosť parsera so šírkou slova 64 bajtov

Číslo vzorky	Počet paketov [kp]	Veľkosť [MB]	Čas výpočtu [ms]	Priepustnosť [kp/s]	Priepustnosť [Gb/s]
1	81,77	71,28	114,16	716,29	5,00
2	78,56	68,84	104,80	749,58	5,26
3	78,01	69,77	104,07	749,54	5,29
4	79,94	71,60	112,85	708,42	5,08
5	71,24	63,95	106,71	667,61	4,80

Tabuľka 3.3: Priepustnosť parsera so šírkou slova 32 bajtov

Priemerná priepustnosť hardvérového riešenia s použitím HLS bola teda 719,42 kp/s resp. 5,09 Gb/s v prípade slova dlhého 64 bajtov. Verzia s 32-bajtovým slovom zaostáva len mierne s priepustnosťou 718,29 kp/s a 5,08 Gb/s. Tabuľka 3.4 obsahuje informácie o potrebných zdrojoch získané syntézou vygenerovaného RTL programom `Precision RTL`.

	Šírka slova [B]	
	32	64
LUTs	13912	26319
CLB Slices	3478	6580
Dffs or Latches	8000	16160

Tabuľka 3.4: Potrebné zdroje pre syntetizovaný parser

Napriek poklesu priepustnosti v 32-bajtovej verzii o zanedbateľných 0,16% je potrebných len asi 52,86% zdrojov FPGA. Z týchto výsledkov možno konštatovať, že šírka slova má výrazný dopad na výslednú spotrebu prostriedkov, ale len mierny dopad na výslednú rýchlosť. Nakoľko parser používajúci širšie slovo nie je rýchlejší, znamená to, že faktorom obmedzujúcim priepustnosť nie je samotné paralelné vyhľadávanie a porovnávanie, ale automat, ktorý parsovanie riadi. Napriek tomu, že RTL bolo generované programovo a s najväčšou pravdepodobnosťou nie úplne optimálne, netriviálny problém s rýchlosťou automatu vzniká aj pri snahe navrhnúť RTL priamo.

3.3 Hardvérovo-sotvérové riešenie

V predchádzajúcich sekciách sme sa venovali najskôr čiste softvérovému a následne čiste hardvérovému riešeniu. Z výsledkov však možno konštatovať, že spracovanie HTTP protokolu patrí medzi protokoly umožňujúce len nízky stupeň paralelného spracovania, a softvér sa javí ako lepšie riešenie už aj na mobilnom procesore strednej triedy.

Problémom hardvérového riešenia sú závislosti vyplývajúce z popisu protokolu. Hlavičky HTTP protokolu, ako aj mnohých ďalších textových aplikačných protokolov, je nutné parsovať sekvenčne. Ak chceme vedieť, aká časť hlavičky nasleduje (hodnota, položka, URI,...), musíme mať znalosti o tom, čo predchádza aktuálnej pozícii. Táto sekvenčnosť obmedzuje paralelizáciu v mieste konečného automatu, ktorý riadi jednotlivé paralelné operácie vyhľadávania či porovnávania.

Softvér, okrem lepšej priepustnosti oproti jednej hardvérovej jednotke, poskytuje výrazne vyššiu flexibilitu. Tá nám oproti hardvéru ponúka možnosť rýchleho rozšírenia analýzy o ďalšie metódy či hlavičky a tým sa prispôsobiť flexibilnému HTTP protokolu. Tiež je

jednoduchšie riešiť situácie zahŕňajúce pakety, ktoré sa odchyľujú od štandardu. Obmedzením softvérového riešenia je ale prístup k dátam. Tie musia byť prenášané zo sieťovej karty cez zbernicu do pamäte servera, ktorý sa stará o monitorovanie danej sieťovej prevádzky. Následne takýto výpočet využíva časť systémových prostriedkov, čím vplýva na výkonnosť celého systému. Kritickými sú práve pamätové prenosy.

Z rovnakého dôvodu sa koncept SDM snaží o redukciu dát, ktoré je nutné spravcovávať softvérom. V prípade HTTP však redukcia dosiahla len 75% paketov, pričom v každom toku je zaznamenaná len prvá hlavička. Z analýzy zachytenej komunikácie je však možné konštatovať, že HTTP paketov obsahujúcich hlavičku je výrazne menej – asi 1,69% (viď. tabuľka 3.5). Preto by sa naša snaha mala upriamiť smerom k efektívnejšej hardvérovej redukcie HTTP paketov, ktoré neobsahujú hlavičku. O hĺbkovú analýzu sa postará softvér. Výslednou architektúrou je tiež možné priamo doplniť architektúru SDM a využiť tak už existujúce riešenie pre spoluprácu hardvéru so softvérom.

	Všetky pakety		Pakety s HTTP hlavičkou			
Číslo vzorky	Počet [kp]	Veľkosť [B]	Počet [kp]	Počet [%]	Veľkosť [MB]	Veľkosť [%]
1	4864,63	4447,08	81,77	1,68	71,28	1,60
2	4332,34	3954,11	78,56	1,81	68,84	1,74
3	4168,07	3808,09	78,01	1,87	69,77	1,83
4	4707,58	4200,41	79,94	1,70	71,60	1,71
5	5146,38	4821,05	71,24	1,38	63,95	1,37

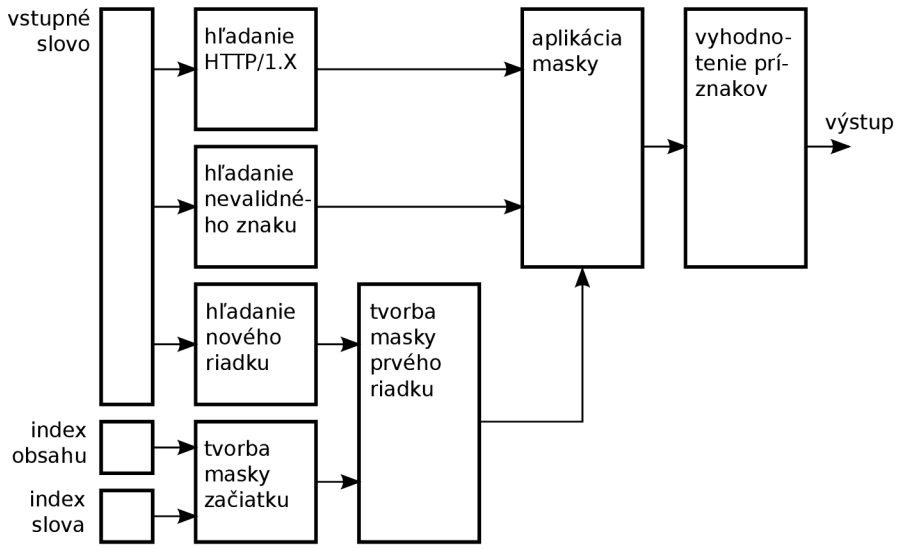
Tabuľka 3.5: Podiel paketov obsahujúcich HTTP hlavičku

3.3.1 Kritériá pre rozpoznanie HTTP hlavičky

V záujme zachovať jednoduchosť obvodu a vyhnúť sa akýmkoľvek závislostiam je nutné zvoliť čo najjednoduchší prístup s prihliadnutím na potrebné zdroje. Priepustnosť takéhoto obvodu by mala byť mnohonásobne vyššia ako priepustnosť celého HTTP parsera. Druhým aspektom pri výbere podmienok je snaha minimalizovať falošné nálezy. Podmienky prehlásenia paketu za HTTP paket obsahujúci hlavičku musia byť dostatočné na to, aby sa minimalizovalo riziko, že paket neobsahujúci hlavičku bude preposlaný na analýzu do softvéru. Tretím aspektom je snaha vylúčiť pakety neobsahujúce hlavičku čo najrýchlejšie tak, aby sme nevylúčili žiadne pakety, ktoré naopak hlavičku obsahujú.

Na základe týchto kritérií boli vybrané nasledujúce podmienky:

- prehľadávame prvý riadok obsahu paketu
- hľadáme prítomnosť reťazca `HTTP/1.0` alebo `HTTP/1.1`
- hľadáme prítomnosť nevalidných znakov, a to
  - znakov, ktoré nie sú ASCII (hodnota vyššia ako 127)
  - znakov, ktoré sú kontrolnými ASCII znakmi (hodnota nižšia ako 32)



Obr. 3.7: Návrh obvodu pre rozpoznávanie http paketov obsahujúcich hlavičku

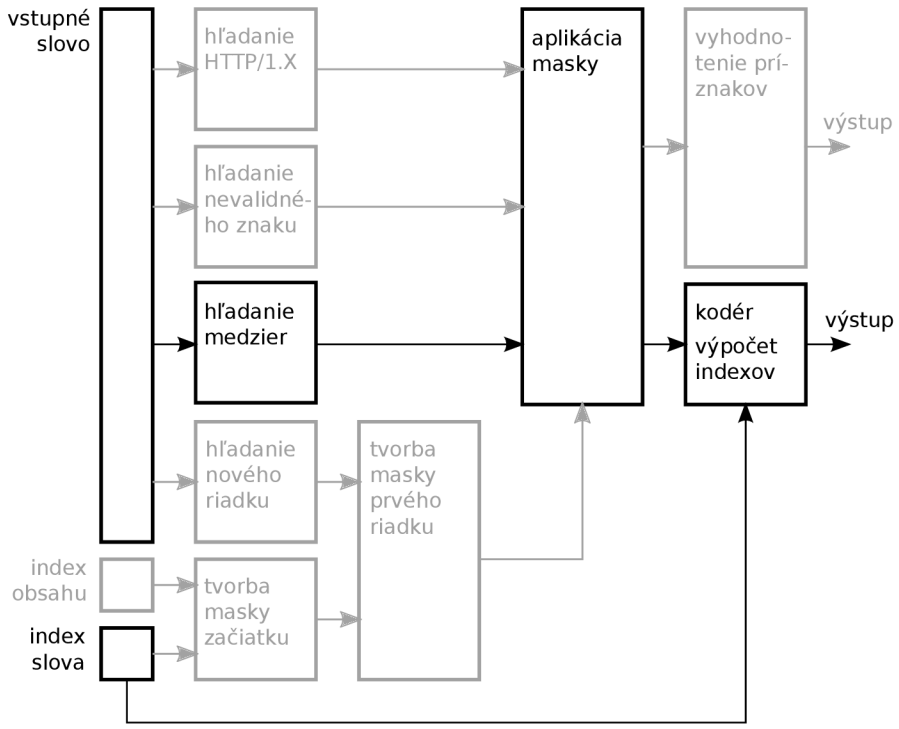
Navrhnutá architektúra obvodu sa nachádza na obrázku 3.7. Vstupom je slovo pamäte spolu s potrebným presahom. V prvej úrovni sú vyhľadávané potrebné sekvencie, a to sekvencia nového riadku a reťazec `HTTP/1.X`. Nájdené výskyty sú vymaskované, pričom maska pre začiatok sa počíta zo vstupu určujúceho počiatočný index a zakončujúca maska je vypočítaná z prvého nálezu nového riadku. Získané príznaky sa ukladajú a sú perzistenté po dobu spracovania jedného paketu. Výstup je generovaný logickými operáciami nad takto získanými príznakmi.

Napriek tomu, že návrh je na nízkej úrovni, využijeme možnosti HLS pre konštrukciu kľúčových prvkov s variabilnou dátovou šírkou. Okrem toho je v rámci HLS možné implementáciu zreťaziť bez námahy a spracovať vstupné slovo každý takt hodinového signálu. Tiež sa na výsledku môžu odzrkadliť optimalizácie niektorých operácií s ohľadom na cieľovú architektúru.

3.3.2 Rozšírenie o parsovanie URL

V prípade, že bude potrebné URL parsovať automatom implementovanom v hardvéri, je žiadúce ho priamo v hardvéri získať. Toho sa dá docieliť rozšírením návrhu o hľadanie prvých dvoch medzier na riadku, o ktorých lokalizovanie sa postará vzhľadávacia jednotka a dva prioritné kodéry. V priestore medzi týmito medzerami sa nachádza URL v prípade HTTP požiadavky a stavový kód v prípade odpovede. Rozšírenie architektúry je naznačené v obrázku 3.8.

Obvod využíva maskovaný výstup z komparátora hľadajúceho znaky medzery a využíva dva prioritné kodéry. Kodéry sa po nájdení svojho prvého výskytu deaktivujú, pričom prvý kodér zamaskuje výskyt, ktorý sám našiel. Tým za zabezpečí, že prvý kodér nájde prvú a druhý kodér druhú medzeru. Návrh je jednoducho zreťaziteľný, pretože sa v ňom nevyskytujú žiadne cyklické závislosti. Podobne ako v základnom návrhu, príznaky sa nulujú príchodom nového paketu. K indexu prvej medzery je na výstupe nutné pričítať jedničku, čím získame index začiatku URL resp. stavového kódu.



Obr. 3.8: Návrh rozšírenia pre hľadanie prvých dvoch medzier

3.3.3 Výsledky

S využitím HLS boli naimplementované obidve uvedené verzie architektúry starajúce sa o označovanie HTTP paketov. Jednodcuhšia verzia prijíma slovo pamäte rozšírené o presah, index na obsah a index aktuálneho slova. Posledným vstupom je príznak nového paketu, ktorý, v prípade že je nastavený, spôsobí reinicializáciu príznakov pre nasledujúci paket. Výstupom je príznak HTTP hlavičky a príznak ukončenia operácie, tzn. príznak určujúci, či je k rozhodnutiu nutné poskytnúť nasledujúce slovo. Použitý nástroj funkciu naplánoval do jedného hodinového taktu pri nastavenej frekvencii 200 Mhz, čo zodpovedá priepustnosti 102,4 Gbps a latencii 5 ns.

Rozšírená verzia navyše vracia indexy na medzery obklopujúce URL resp. stavový kód v prípade, že sa jedná o paket obsahujúci HTTP hlavičku. Naplánovaná architektúra po zreťazení dosahuje rovnakú priepustnosť, 12,4 Gbps, pri latencii dvoch hodinových cyklov, tzn. 10 ns. V tabuľke 3.6 sú uvedené potrebné zdroje oboch verzií zistené nástrojom `Precision RTL`.

	Jednoduchá verzia	Rozšírená verzia
LUTs	1243	1784
CLB Slices	311	446
Dffs or Latches	6	144

Tabuľka 3.6: Potrebné zdroje pre detekciu HTTP hlavičky

Pre funkčnosť takto získanej architektúry je nutné doimplementovať obvod pre plnenie vstupov a konečný automat, ktorý bude výslednú architektúru riadiť. Pretože použitý nástroj neumožňuje v dostupnej verzii syntézu viacerých blokov, nie je ním možné pomocou HLS spojiť a testovať implementáciu tohoto obvodu. Z predbežných výsledkov je ale možné konštatovať, že takáto architektúra zaberá zlomok zdrojov celého parsera a poskytuje priaznivé predpoklady výsledkov celkovej implementácie.

Kapitola 4

Záver

Táto práca je zameraná na návrh a implementáciu hardvérovo akcelerovanej analýzy HTTP prevádzky použitím prístupu nazývaného vysokoúrovňová syntéza. Cieľom bolo dosiahnuť priepustnosti dostačujúcej pre vysokorýchlostné siete s rýchlosťou 100 Gb/s.

Po zoznámení sa s protokolom HTTP a existujúcim monitorovacím riešením v podobe softvérovo definovaného monitoringu som naštudoval princípy vysokoúrovňovej syntézy a k nej určený nástroj `Catapult C`. Získané znalosti som najskôr využil pre implementáciu softvérového parsera v jazyku C, ktorého úlohou je extrahovať zaujímavé položky z HTTP hlavičiek. Vybrané boli URL, stavový kód, hostiteľ, identifikácia klienta a servera. Implementácia parsera bola overená na vzorke dát z reálnej prevádzky. Priepustnosť v najhoršom prípade dosahuje 963,85 kilopaketov za sekundu.

Po úprave zdrojových kódov pre maximálne využitie možného paralelizmu bola prevedená syntéza parsera do technológie FPGA. Boli simulované dve verzie výsledného RTL s rôznou šírkou slova pamäte a rôznou úrovňou paralelizmu pri vyhľadávaní a porovnávaní reťazcov. Maximálna dosiahnutá priepustnosť bola 719,42 kilopaketov za sekundu pre 64-bajtovú šírku slova a 718,29 kilopaketov za sekundu pre 32-bajtovú šírku slova.

Vzhľadom na dosiahnuté výsledky bola ďalej navrhnutá architektúra pre rozpoznanie HTTP hlavičky v pakete a získanie URL s teoretickou priepustnosťou 102,4 Gb/s, ktorej cieľom je redukcia dát určených pre hlbšiu softvérovú analýzu. Architektúra využíva výhody hardvérového a softvérového prístupu (rýchlosť filtrovania vs. rýchlosť analýzy) a snaží sa vhodne rozdeliť úlohu pre dosiahnutie maximálnej priepustnosti s ohľadom na minimalizáciu spotreby hardvérových zdrojov. Táto architektúra je kombináciou hardvérového a softvérového prístupu a je použiteľná ako rozšírenie analyzátora v architektúre SDM.

V budúcnosti je možné prácu rozšíriť o hardvérové vyhľadávanie vzorov v rámci získanej URL, prípadné rozšírenie analýzy pre ďalšie protokoly.

Literatúra

[1] RUFI, Antoon W. *Network fundamentals: CCNA exploration labs and study guide*. Indianapolis, Ind.: Cisco Press, c2008, xxvi, 370 p. Cisco Networking Academy Program series. ISBN 15-871-3203-6.

[2] ISO/IEC 7498-1:1994(E). *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. Second edition. Geneva, Switzerland: International Organization for Standardization, 1994.

[3] RFC 1122. *Requirements for Internet Hosts – Communication Layers*. Internet Engineering Task Force, 1989. Dostupné z: http://www.ietf.org/rfc/rfc1122.txt.

[4] RFC 1123. *Requirements for Internet Hosts – Application and Support*. Internet Engineering Task Force, 1989. Dostupné z: http://www.ietf.org/rfc/rfc1123.txt.

[5] TANENBAUM, Andrew S. *Computer networks*. 4th ed. New Jersey: Prentice-Hall, c2003, xx, 891 s. ISBN 01-306-6102-3.

[6] KUROSE, James F a Keith W ROSS. *Computer networking: a top-down approach*. 4th ed. Boston: Pearson ; Addison-Wesley, 2007, xxiv, 852 s. Cisco Networking Academy Program series. ISBN 978-0-321-49770-3.

[7] RFC 1945. *Hypertext Transfer Protocol – HTTP/1.0*. Internet Engineering Task Force, 1996. Dostupné z: http://www.ietf.org/rfc/rfc1945.txt.

[8] RFC 2616. *Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, 1999. Dostupné z: http://www.ietf.org/rfc/rfc2616.txt.

[9] The Original HTTP as defined in 1991. BERNERS-LEE, Tim. W3C. *World Wide Web Consortium* [online]. [cit. 2014-05-17]. Dostupné z: http://www.w3.org/Protocols/HTTP/AsImplemented.html.

[10] Service Name and Transport Protocol Port Number Registry. IANA. *Internet Assigned Numbers Authority* [online]. 2014-05-14 [cit. 2014-05-17]. Dostupné z: http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

[11] KEKELY, Lukáš. *Hardwarová akcelerace aplikací pro monitorování a bezpečnost vysokorychlostních sítí*. Brno, 2013. Diplomová práce. FIT VUT v Brně.

[12] POWERS, Adam. Your network is changing – are you ready for it?. *Computer Fraud*. 2011, vol. 2011, issue 3, s. 11-13. DOI: 10.1016/S1361-3723(11)70030-6. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S1361372311700306.

[13] ŠÍMA, Tomáš, Petr VELAN a Pavel ČELEDA. *FlowMon - Plugins for HTTP Monitoring* [online]. 2012-12-20 [cit. 2014-05-17]. Dostupné z: http://dior.ics.muni.cz/~velan/flowmon-input-http/.

[14] BACON, David F., Rodric RABBAH a Sunil SHUKLA. FPGA programming for the masses. *Communications of the ACM*. 2013-04-01, vol. 56, issue 4, s. 56-. DOI: 10.1145/2436256.2436271. Dostupné z: http://dl.acm.org/citation.cfm?doid=2436256.2436271.

[15] FINGEROFF, Michael. *High-level synthesis: blue book*. United States: Mentor Graphics Corporation, 2010, 330 p. ISBN 14-500-9723-5.

[16] MEEUS, Wim, Kristof VAN BEECK, Toon GOEDEMÉ, Jan MEEL a Dirk STROOBANDT. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*. 2012, vol. 16, issue 3, s. 31-51. DOI: 10.1007/s10617-012-9096-8. Dostupné z: http://link.springer.com/10.1007/s10617-012-9096-8.

[17] SCOTT, David a Richard SHARP. *Abstracting Application-Level Web Security* [online]. 2002 [cit. 2014-05-17]. Dostupné z: http://www2002.org/CDROM/refereed/48/.

[18] XILINX. *Virtex-5 FPGA Configuration User Guide* [online]. UG191 (v3.11). 2006, 2012-10-19 [cit. 2014-05-17]. Dostupné z: http://www.xilinx.com/support/documentation/user_guides/ug191.pdf.

Dodatok A

Obsah CD

`/Parser_C`
: Priečinok obsahujúci zdrojový kód HTTP parsera v jazyku C

`/Parser_HLS`
: Priečinok obsahujúci zdrojový kód HTTP parsera v jazyku C++ s použitím `AC Datatypes`, určený pre vysokoúrovňovú syntézu

`/HTTP_flagger`
: Priečinok obsahujúci zdrojový kód pre označovanie paketov obsahujúcich HTTP hlavičku v jazyku C++ s použitím `AC Datatypes`, určený pre vysokoúrovňovú syntézu

`/HTTP_flagger_ex`
: Priečinok obsahujúci zdrojový kód pre označovanie paketov obsahujúcich HTTP hlavičku rozšírený o získavanie pozície URL v jazyku C++ s použitím `AC Datatypes`, určený pre vysokoúrovňovú syntézu

`/tex`
: Priečinok obsahujúci zdrojový kód tohoto dokumentu

`/16124.pdf`
: Súbor obsahujúci elektronickú verziu tohoto dokumentu