UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Marko Kotar

Realizacija knjižnice regex na specializirani strojni opremi

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01587/2009
Datum: 01.09.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MARKO KOTAR**

Naslov: **REALIZACIJA KNJIŽNICE REGEX NA SPECIALIZIRANI STROJNI OPREMI**

IMPLEMENTATION OF REGEX LIBRARY ON A SPECIAL PURPOSE HARDWARE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Procesor Octeon 3860 podjetja Cavium nudi strojno podporo za delo s končnimi avtomati. Za ta procesor realizirajte knjižnico regex za delo z regularnimi izrazi, ki za hitrejše delovanje izkoristi to strojno podporo. Natančno opišite morebitne razlike med vašo in standardno realizacijo knjižnice regex.

Mentor:
B. Slivnik
doc. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
Franc Solina
prof. dr. Franc Solina

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Marko Kotar,

z vpisno številko 63030102,

sem avtor/-ica diplomskega dela z naslovom:

Realizacija knjižnice regex na specializirani strojni opremi

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom
  doc. dr. Boštjana Slivnika
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 14. december 2009 Podpis avtorja/-ice:

Zahvala

Zahvaljujem se mojemu očetu Martinu, bratu Robertu in mami Darji, da so mi stali ob strani. Zahvaljujem se tudi podjetju Quarad, d. o. o., ki mi je omogočilo testiranje knjižnice na njihovi opremi. Prav tako se moram zahvaliti tudi mentorju doc. dr. Boštjanu Slivniku za dobro mentorstvo.

Kazalo

Seznam uporabljenih kratic in simbolov

- DKA – deterministični končni avtomat
- ϵNKA – nedeterministični končni avtomat z epsilon prehodi
- NKA – nedeterministični končni avtomat
- MIPS – (Million Instructions Per Second – milijon ukazov na sekundo) arhitektura z RISC naborom ukazov
- RISC – računalnik z reduciranim naborom ukazov (Reduced Instruction Set Computer)
- PCRE – regularni izrazi združljivi s Perlom (Perl Comapatible Regular Expressions)
- GPL – Splošna javna licenca (General Public Licence)
- MMU – enota za upravljanje pomnilnika (Memory Management Unit)
- CN3680 – Tip procesorja Octeon (Glej Opis strojne opreme.)
- PHP – trenutno tričrkovni rekurzivni akronim za PHP Hypertext Preprocessor, starejši izraz pa je Personal Home Page Tools
- UTM – Unified Threat Management
- VPN – navidezno zasebno omrežje (Virtual Private Network)
- IPsec – Internet Protocol Security
- SSL – Secure Sockets Layer

- IDS – Intrusion Detection System
- IPS – Intrusion Prevention System
- L4+ – nivo 4 (transportni nivo) ali več v OSI modelu (Open System Interconnection Reference Model)
- TCP – Transmission Control Protocol
- iSCSI – Small Computer System Interface
- RDMA – Remote Direct Memory Access
- SRTP – Secure Real-time Transport Protocol
- DES – Data Encryption Standard
- 3DES – trojni DES (Triple DES)
- AES – Advanced Encryption Standard
- SHA1 – družina Secure Hash Algorithm-1
- SHA-2 – družina Secure Hash Algorithm-2: SHA-224, SHA-256, SHA-384 in SHA-512.
- SHA-512 – funkcija Secure Hash Algorithm-512
- RSA – algoritem za javno kodiranje
- DH – Diffie–Hellmanova izmenjava ključev (Diffie–Hellman Key Exchange)
- DDR2 – Double Data Rate synchronous dynamic random access memory
- RLDRAM2 – Reduced Latency DRAM(Dynamic Random Access Memory)
- IP – internetni protokol (Internet Protocol)
- QoS – kvaliteta storitve (Quality of Service)
- ECC – koda za popravljanje napak (Error Correction Code)
- MIPS64 – MIPS 64-bitni standardni nabor ukazov

- SPI – System Packet Interface
- SPI-4.2 – System Packet Interface Level 4
- RGMII – Reduced Gigabit Media Independent Interface
- GPIO – splošno namenski vhod/izhod (General Purpose Input/Output)
- I2C – Inter-Integrated Circuit (tudi I^2C)
- UART – Universal Asynchronous Receiver/Transmitter
- FAU – Fetch and Add Unit
- LLM – nizko latentni pomnilnik (Low Latency Memory)
- ϵ – epsilon prehod
- M – avtomat
- Σ – vhodna abeceda avtomata
- δ – funkcija prehodov avtomata
- F – končna stanja avtomata
- q_0 – začetno stanje avtomata
- Q – stanja avtomata
- POSIX – Portable Operating System Interface
- ASCII – American Standard Code for Information Interchange
- ISO – International Organization for Standardization

Povzetek

Implementirali smo strojno pospešeno knjižnico, ki je delno združljiva s knjižnico PCRE (Perl Compatible Regular Expression Library).

Naša knjižnica teče na procesorju Octeon CN3680 podjetja Cavium. Za delovanje uporablja njegovo posebno enoto za razširjene diskretne končne avtomate. Implementacijo smo močno poenostavili. Podpira le osnovne funkcije in oblike regularnih izrazov originale knjižnice. Prav tako smo uporabili drugačen način iskanja. Zato dobimo tudi drugačen rezultat pri iskanju. Naša knjižnica najde najkrajši niz, ki ustreza regularnemu izrazu. Rezultat je niz, ki je lahko skrajšan na začetku in/ali na koncu niza rezultata originalne knjižnice. Tudi razčlembe rezultata na dele, ki jih predstavljajo regularni izrazi v oklepajih, nismo implementirali.

Opisali smo osnovni dostop do enote za deterministične končne avtomate preko Simple Executive Library, ki je pod GPL licenco. Knjižnica teče pod operacijskim sistemom Linux.

Zaradi modularnosti smo prevedli regularne izraze preko vmesne strukture v avtomate. Nato smo jih zapisali v nizko latentni pomnilnik, kjer do njih dostopa enota. Izmerili smo hitrosti iskanja različnih regularnih izrazov in ugotovili, da je občutno večja hitrost glede na originalno nepospešeno knjižnico najbolj zanimiva pri alternativah. Tu časovna zahtevnost originalne knjižnice raste s številom alternativ, pri naši pa je konstantna.

Ključne besede:

PCRE, strojna pospešitev regex, Octeon DKA

Abstract

We implemented hardware accelerated library, which is partialy compatible with PCRE (Perl Compatible Regular Exression Library).

Our library runs on Cavium Octeon CN3680 processor. For execution a special purpose unit for extended discrete finite automata is used. Our implementation is very simplified. Only basic functions and regular expressions of original library are supported. A different way of search was also used. That is why a different result of seach is returned by library, too. It finds the shortest string which corresponds to the regular expresssion. The result is the string which can be a shortened string in the beginning and/or in the end of the string of the reusult returned by the original library. The capturing parens are also not implemented.

We described a basic access to the special unit for extended discrete finite automata through Simple Executive Library. It is under GPL licence. It can be used with or without Linux operation system.

Due to modularity regular expressions are translated through an inmediate structure to the format of the automata, which are then saved to low latency memory, where they can be accessed by the unit. The search speed of the hardware accelerated library was measured with different regular expressions. The measurements show an interesting speed gain on alternatives as compared to original unaccelerated library. The time complexity of the original library is linear depending on a number of alternatives. But our library has a constant time complexity.

Key words:

PCRE, regex hardware acceleration, Octeon DKA

Poglavje 1

Uvod

Knjižnice za regularne izraze se uporabljajo za različne naloge v zvezi z nizi. Nekateri naloge so iskanje vzorca v tekstu, preverjanje formata (npr. datuma), zamenjevanje delov niza ali spreminjanje formata zapisa, izvajanje določene funkcije ob najdenem vzorcu v nizu ...

Procesorji Octeon podjetja Cavium vsebujejo posebno ločeno enoto za razširjene deterministične končne avtomate. Njihova moč je nekoliko večja od običajnih determinističnih končnih avtomatov. Še najbolj so podobni avtomatom z izhodi, in sicer Mealyjevim avtomatom. Vendar se razlikujejo, ker ima enota izhod samo pri določenih prehodih, kar je več kot Mealyjev avtomat. Enota vrača tudi število prebranih znakov vhoda. Manjša moč pa je zaradi fiksno določene informacije na izhodu. Avtomat vrača namreč le notranje stanje avtomata in število prebranih bajtov na vhodu.

Na spletu obstaja veliko knjižnic za regularne izraze. Nekatere omogočajo samo iskanje, druge pa tudi zamenjevanje najdenih delov. Različne so tudi po številu podprtih posebnih znakov pa tudi po moči. Nekatere knjižnice namreč uporabljajo tudi dodatne funkcionalnosti, ki presežejo moč regularnih izrazov. Vendar še vedno delujejo, saj se izvajajo na močnejšem pomnilniško omejenem Turingovem stroju – računalniku in ne na avtomatu. Večina knjižnic poenostavi sintakso regularnih izrazov kot so teoretično predstavljeni v literaturi [5] z dodatnimi metaznaki, ki skrajšajo zapis regularnega izraza. Namesto sprejemanja besed večinoma privzeto iščejo besede v nizu, ki so v jeziku regularnega izraza. Ker na procesorju Octeon teče operacijski sistem Linux, smo se osredotočili na knjižnice pisane za ta operacijski sistem. Izbrali smo knjižnico Perl Compatible Regular Expression Library ali na kratko PCRE. Knjižnica je razširjena med odprtokodnimi programi(npr. PHP je dodal podporo te knjižnice poleg že uveljavljene lastne). Nekateri drugi programi, ki

uporabljajo to knjižnico, so: Privoxy, Wireshark, xgrep, Apache, Winefish, tin, snort ...

Kot že samo ime knjižnice PCRE pove, je knjižnica v veliki meri združljiva s Perl-ovimi regularnimi izrazi. Knjižnica podpira samo iskanje za razliko od Perl-ovih regularnih izrazov, ki podpirajo tudi zamenjavo. Razširi se jo lahko tudi na zamenjevanje. To je na primer že narejeno v Privoxy-ju in PHP-ju. Zato smo se odločili, da bomo implementirali to knjižnico.

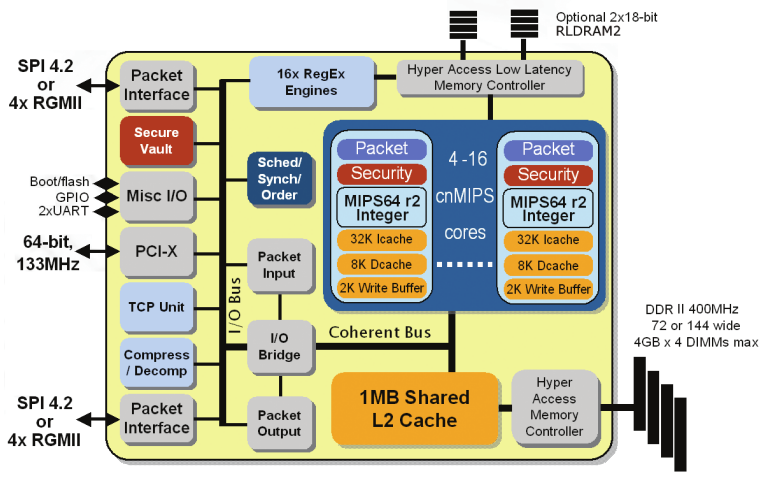
Poglavje 2

Opis strojne opreme

To poglavje je povzeto po [4].

Uporabili smo Octeon CN3860 64–bitni 16–jedrni MIPS procesor s 5 stopenjskim cevovodom. Frekvenca ure jedra je 500 MHz. Procesor lahko izvaja tudi 32–bitne MIPS ukaze. Prav tako ima tudi MMU enoto. Ponuja ga podjetje Cavium Networks. Na tržišču je že vsaj 3 leta. Namenjen je za inteligentne mrežne storitve, podatkovne strežnike, kontrolo mrežnega prometa. Uporaben je za:

- integrirane ali samostojne usmerjevalnike in naprave naslednje generacije,
- naprave UTM s požarnim zidom, VPN(IPsec, SSL), IDS, IPS, skeniranje Anti-virus,
- vsebinsko procesiranje L4+ in preklapljanje,
- pospešene mrežne kartice za varnost, TCP, procesiranje vsebine in kompresijo,
- integrirane upravljalne in usmerjevalne procesorske kartice,
- preklopne in/ali usmerjevalne linijske kartice, servisne kartice za kontrolo in procesiranje podatkovnih poti,
- procesiranje TCP, iSCSI, RDMA, kompresijo za aplikacijo shranjevanja,
- varnost in procesiranje pri brezžičnih mrežnih stikalih in napravah.



Slika 2.1: Shema procesorja

Propustnost sistema se giblje od 1Gb/s do 10Gb/s. Ima strojno podporo za: kompresijo in dekompresijo podatkov (zlib, gzip), vhodno/izhodno paketno procesiranje, QoS (Quality of Service – kakovost storitve), pospeševanje TCP, IPsec, SSL, SRTP, varnostne algoritme (DES, 3DES, AES do 256 bitov, SHA1, SHA-2 do SHA-512, RSA, DH) ter enoto namenjeno regularnim izrazom. Za delovanje enote namenjene regularnim izrazom (enota za razširjene deterministične končne avtomate – enota za DKA) uporablja poleg običajnega DDR2 pomnilnika še nizko latentni dinamični pomnilnik (angl. *Low Latency Memory* – LLM) RLDRAM2 za zapis avtomata.

Procesorski čip ima porabo električne energije le od 14W do 30W. Sposoben je izvesti do 19,2 milijard ukazov na sekundo, 20 tisoč posredovanj IP (angl. *IP Forwarding*) paketov velikosti 64B na sekundo, obdelati podatke s hitrostjo do 4Gb/s z enoto za DKA ter z enoto za kompresijo in dekompresijo. Strojna QoS podpira razvrščanje v čakalne vrste in razporejanje. Doseže tudi zelo nizko zakasnitev prometa v realnem času. Pomnilniški sistem podpira 1 MB 8–smernega set asociativnega predpomnilnika z zaklepanjem, ki je zaščiten proti napakam z ECC (Error Correction Code – koda za popravljanje napak). Integriran ima krmilnik za 128–bitni ali 144–bitni DDR2 pomnilnik(vse do DDR2-800). Jedra uporabljajo poleg standardnih MIPS64 ukazov tudi razširjen nabor ukazov. Med njimi so nekateri namenjeni mrežnemu procesiranju. Čip ima tudi dva vmesnika SPI-4.2 ali 2–krat po 4 RGMII vmesnike za ethernet.

Na sistemu lahko tečeta operacijski sistem Linux ali pa samostojna aplika-

cija, ki je statično povezana s knjižnico Simple Executive Library. Ta knjižnica deluje tudi pod operacijskim sistemom Linux. Na voljo je pod GPL licenco verzije 2.

2.1 Enota za regularne izraze – enota za DKA

Ta del je prav tako povzet po [6].

Enota vsebuje 16 neodvisnih podenot. Sodeč po Simple Executive Library uporablja za izvajanje nekoliko razširjen deterministični končni avtomat.

Strukturo avtomata najprej naložimo v LLM (RLDRAM2). Možni sta dve različici strukture avtomata (mala ali velika). Mala zasede manj prostora. Velika pa ima večjo odpornost na napake zapisa podatkov v nizko latentnem pomnilniku (podpira tudi ECC, mala ima samo pariteto) in možnost podajanja vrste stanja (razen začetnega) v prehodih.

Začetno stanje podamo pred izvajanjem. Prav tako podamo vhodni niz (besedo), na kateri izvajamo avtomat. Poleg običajnih, začetnega ter končnih stanj enota pozna še "označevalna" stanja (angl. *marked states*).

Za rezultat izvajanja avtomata na enoti dobimo strukturo, v kateri so podani razlog za končanje izvajanja avtomata, podatki o vseh obiskih označevalnih stanj s pozicijami v nizu ter zadnjega obiskanega stanja skupaj s pozicijo v nizu. Pozicije v nizu predstavljajo položaj pri branju vhodnega niza, ko avtomat doseže posamezno stanje (označevalno ali končno).

Poglavje 3

Knjižnica za delo s strojno opremo

To poglavje je napisano na podlagi [6].

Knjižnica za delo s strojno opremo (Simple Executive Library) vsebuje funkcije za lažje programiranje samostojne aplikacije brez operacijskega sistema. Prav tako omogoča lažji dostop do dodatnih enot procesorja skozi operacijski sistem Linux, ki niso podprte preko gonilnikov.

Programerju pomaga pri:

- klicu ukazov zbirnika iz programskega jezika C/C++,
- inicializaciji procesorja ter ostalih dodatnih enot,
- delu s pomnilnikom,
- medjedrni sinhronizaciji,
- dostopu do nizko latentnega pomnilnika,
- upravljanju z enoto za izvajanje determinističnih končnih avtomatov,
- pisanju in branju pomnilnika flash,
- upravljanju z GPIO,
- uporabi prekinitev,
- protokolu I2C,
- logiranju,

- dostopu do ure realnega časa,
- SPI (System Packet Interface – sistemski paketni vmesnik),
- informacijah o sistemu (tip procesorja),
- časovniku (timerju),
- UART–u,
- enoti zip – enoti za kompresijo in dekompresijo,
- FAU(Fetch and Add Unit – enota *sprejmi vrednost in dodaj* – uporablja se pri sinhronizaciji dostopov).

3.1 Inicializacija knjižnice

Za operacijski sistem Linux je funkcija *main* že definirana v knjižnici. Taka zgradba je zaradi možnosti, da lahko aplikacija, ki teče brez operacijskega sistema, teče tudi pod operacijskim sistemom Linux brez spremembe izvorne kode.

Funkcija *main* kliče funkcijo *appmain*, ki je naš program. Vendar, če imamo knjižnico, je funkcijo *main* nesmiselno uporabiti v knjižnici, kar tudi prevajalnik ne dopušča. Inicializacija nizko latentnega pomnilnika ni možna brez vključitve funkcije *main*, tako da knjižnice ne spremenimo. Rešitev je v tem, da prej poženemo Linux program, ki uporablja funkcijo *main* v knjižnici in inicializira nizko latentni pomnilnik, saj dostop normalno deluje tudi pri vseh naslednjih dostopih programov brez ponovne inicializacije. Na ta način se med drugim izognemo večkratni inicializaciji strojne opreme. Zato bomo po vsakem ponovnem zagonu operacijskega sistema zagnali program za inicializacijo nizko latentnega pomnilnika.

3.2 Funkcije knjižnice za delo z običajnim pomnilnikom

Za delo s pomnilnikom lahko uporabljamo:

- *int cvmx_bootmem_init(void *mem_desc_ptr)*

  Inicializira pomnilnik.

- *void *cvmx_bootmem_alloc(uint64_t size, uint64_t alignment)*

  Vrne del pomnilnika s poravnavo *alignment* in velikostjo *size*.

- *static inline uint64_t cvmx_ptr_to_phys(void ptr)*

  Prevede navidezni naslov *ptr* v fizičnega.

- *static inline void *cvmx_phys_to_ptr(uint64_t physical_address)*

  Prevede fizični naslov *physical_address* v navideznega.

Ostale funkcije omogočajo še dodeljevanje in sproščanje pomnilnika z imenovanimi pomnilniškimi bloki. Vsak blok lahko poimenujemo z zaporedjem znakov. Na ta način lahko dostopamo do istega bloka tudi z drugim procesom oziroma programom.

3.3 Funkcije za dostop in delo z nizko latentnim pomnilnikom

Za dostop nizkolatentnega pomnilnika (LLM) lahko uporabljamo naslednje funkcije:

- *int cvmx_llm_initialize(void)*

  Inicializira LLM; inicializacija je potrebna za branje in pisanje v LLM. Pred tem mora biti klicana funkcija *main* knjižnice, ki poskrbi, da je določen pravilen način dostopa.

- *static inline uint64_t cvmx_llm_parity(uint64_t value)*

  Izračun paritete; uporablja se za mali format avtomata (glej naprej). Zapis prehoda se lahko v LLM nezaželeno spremeni.

- *static inline int cvmx_llm_ecc(uint64_t value)*

  Izračun ECC (Error Correction Code – koda za popravljanje napak); uporablja se za veliki format avtomata (glej naprej). Zapis prehoda se lahko nezaželeno spremeni v LLM. Vendar se napake največkrat same popravijo (enojne napake se popravljajo same).

- *static inline void cvmx_llm_write_narrow(cvmx_llm_address_t address, uint64_t value, int set)*

Zapis v LLM 36 bitov dolge vrednosti; LLM pomnilnik mora biti pred tem ukazom inicializiran. Da so podatki uspešno shranjeni, lahko preverimo z branjem. Na ta način se lahko prepričamo, ali imamo dostop do LLM, ker smo zapisane podatke tudi prebrali.

- *static inline void cvmx_llm_write_wide(cvmx_llm_address_t address, uint64_t value, int set)*

  Zapis v LLM 64 bitov dolge vrednosti; LLM pomnilnik mora biti prav tako inicializiran. Preverjanje lahko izvedemo prav tako z branjem.

- *static inline cvmx_llm_data_t cvmx_llm_read_narrow(cvmx_llm_address_t address, int set)*

  Branje 36–bitne vrednosti iz LLM.

- *static inline uint64_t cvmx_llm_read_wide(cvmx_llm_address_t address, int set)*

  Branje 64–bitne vrednosti iz LLM.

3.4 Funkcije za upravljanje z enoto za DKA

Za delo z enoto za DKA lahko uporabljamo naslednje funkcije:

- *static inline void cvmx_dfa_write_edge_sm(const cvmx_dfa_graph_t *graph, uint64_t source_node, uint64_t match_index, uint64_t destination_node0, uint64_t destination_node1)*

  V LLM zapiše prehoda v formatu malega avtomata pri vhodnima črkama *match_index*(natančneje cela polovica numerične vrednosti vhodne črke), ki sta po numerični vrednosti zaporedni; Prehod v stanje *destination_node0* je pri vhodni črki s sodo numerično vrednostjo, prehod v stanje *destination_node1* pa je pri vhodni črki z liho numerično vrednostjo. Pomnilniški naslov se avtomatsko izračuna iz začetnega naslova avtomata, številke stanja *soruce_node* in vhodnih črk *match_index*.

- *static inline void cvmx_dfa_write_node_lg(const cvmx_dfa_graph_t *graph, uint64_t source_node, unsigned char match, uint64_t destination_node, cvmx_dfa_node_type_t destination_type)*

  V LLM zapiše prehod v formatu velikega avtomata pri vhodni črki *match*; Pomnilniški naslov se avtomatsko izračuna iz začetnega naslova avtomata, številke stanja *soruce_node* in vhodne črke *match*.

- *static inline void cvmx_dfa_write_doorbell(uint64_t num_commands)*

  Sporoči (pozvoni), da mora enota za DKA izvesti določeno število ukazov.

- *static inline void cvmx_dfa_write_command(cvmx_dfa_command_t *command)*

  V ustrezno lokacijo zapiše ukaz *command* za enoto za DKA in ji sporoči, da je na voljo nov ukaz.

- *static inline void cvmx_dfa_submit(const cvmx_dfa_graph_t *graph, int start_node, void *input, int input_length, int use_gather, int is_little_endian, cvmx_dfa_result0_t *result, int max_results, cvmx_wqe_t *work)*

  Pošlje ukaz za začetek izvajanja avtomata na enoti za DKA nad podatki *input* z avtomatom *graph*.

- *static inline uint64_t cvmx_dfa_is_done(cvmx_dfa_result0_t *result_ptr)*

  Preveri, če je enota za DKA že zaključila z izvajanjem avtomata.

- *int cvmx_dfa_initialize(void)*

  Incializira DKA enoto in podatkovne strukture za upravljanje z njo.

- *void cvmx_dfa_shutdown(void)*

  Sprosti podatkovne strukture za upravljanje z DKA enoto.

3.5 Struktura podatkov v LLM za enoto za DKA

V strukturi so zapisani le prehodi avtomata. Za vsako stanje (ni nujno, da za vsa končna) in za vsako vhodno črko je definiran prehod. Prehodi so sortirani po vrsti glede na stanje, iz katerega prehajajo, in vhodne črke. Torej sta izhodiščno stanje in vhodna črka prehoda že definirana implicitno s položajem v pomnilniku.

Na voljo sta dva formata DKA v LLM (mali in veliki). V velikem je en zapis na naslednje stanje v 64–bitni besedi, v malen pa sta zapisani dve ciljni stanji v 36–bitni besedi.

3.5.1 Mali format DKA

Mali format ima za preverjanje pravilnosti le kodo za detekcijo napake, ne pa tudi za odpravljanje. Zasede manj prostora. Število označevalnih in končnih stanj navedemo na začetku izvajanja. Na podlagi teh informacij enota ugotovi, katero je označevalno, končno in katero je navadno stanje.

```
typedef union
{
 uint64_t u64;
 struct
 {
   uint64_t mbz        :28;//neuporabljeno
   uint64_t p1         : 1;//paritetni bit
   uint64_t next_node1 :17;//naslednje stanje
   uint64_t p0         : 1;// paritetni bit
   uint64_t next_node0 :17;//naslednje stanje
 };
} cvmx_dfa_node_next_sm_t;
```

3.5.2 Veliki format DKA

Veliki format ima v eni 64–bitni besedi tudi tip naslednjega stanja. Namesto paritetnega bita ima kodo za popravljanje napak.

```
typedef union
{
 uint64_t u64;
 struct
 {
  //neuporabljeno:
  uint64_t              mbz           :28;
  //koda za popravljanje napak:
  uint64_t              ecc           : 7;
  //tip naslednjega stanja:
  cvmx_dfa_node_type_t type           : 2;
  //dodatni biti:
  uint64_t              extra_bits    : 5;
  //stevilo replikacij:
  uint64_t              next_node_repl: 2;
  uint64_t              next_node     :20;
```

```
  };
} cvmx_dfa_node_next_lg_t;
```

3.5.3 Tipi stanj

Vsak veliki format avtomata ima tri vrste stanj. Ta so definirana kar pri prehodih. Poleg običajnih stanj DKA ima še „označevalna“ stanja. To so stanja, kjer se avtomat ne ustavi, ampak vseeno zabeleži kot del rezultata informacijo o tem, v katerem stanju je in koliko vhodnega niza je že prebral.

```
typedef enum
{
//navadno stanje:
 CVMX_DFA_NODE_TYPE_NORMAL   =0,
//oznacevalno stanje:
 CVMX_DFA_NODE_TYPE_MARKED   =1,
//koncno stanje:
 CVMX_DFA_NODE_TYPE_TERMINAL=2
} cvmx_dfa_node_type_t;
```

3.5.4 Struktura rezultata

Rezultat enote za DKA je sestavljen iz osnovnih podatkov o rezultatu (prvi rezultat) in seznama rezultatov označitev iz označevalnih stanj (drugi rezultat).

Prvi rezultat

Prvi rezultat nam pove, kdaj se je avtomat ustavil. Ko enota za DKA zaključi z izvajanjem, postavi zastavico *done*. Prav tako nam ta struktura pove število rezultatov pa tudi, zakaj se je enota za DKA ustavila. Poleg tega imamo še bit *last_marked*, ki nam pove, če je zadnje stanje označevalno. Enota se namreč lahko ustavi v katerem koli stanju, če je vzrok za zaustavitev napaka v LLM, če je zmanjkalo vhodnega niza ali prostora za rezultate.

```
typedef union
{
 uint64_t u64;
 struct
 {
 //razlog ustavitve:
```

```
  cvmx_dfa_stop_reason_t  reas         : 2;
  uint64_t                mbz          :44;
  //zadnje stanje oznacevalno? :
  uint64_t                last_marked  : 1;
  //izvajanje zakljuceno? :
  uint64_t                done         : 1;
  //stevilo zapisov oznacitvev:
  uint64_t                num_entries  :16;
 } s;
} cvmx_dfa_result0_t;
```

Razlog ustavitve

Razlog za ustavitev enote za DKA je lahko to, da je konec vhodnega niza, da je napaka v LLM, da je pomnilnik za rezultate poln ali da je enota za DKA dosegla končno stanje avtomata. Ti razlogi nam skupaj z zastavico *last_marked* omogočajo, da program ugotovi, kaj mora narediti, ali nadaljevati izvajanje nad naslednjim blokom podatkov ali pa samo obdelati rezultate in ponovno pognati enoto DKA.

```
typedef enum
{
//konec vhodnega niza:
 CVMX_DFA_STOP_REASON_DATA_GONE     =0,
//napaka v LLM:
 CVMX_DFA_STOP_REASON_PARITY_ERROR=1,
//stevilo zapisov 2. rezultata je preseglo doloceno mejo:
 CVMX_DFA_STOP_REASON_FULL          =2,
//DKA enota je prisla v koncno stanje:
 CVMX_DFA_STOP_REASON_TERMINAL      =3
} cvmx_dfa_stop_reason_t;
```

Drugi rezultat

Pri drugem rezultatu dobimo podatke iz označevalnega stanja oziroma podatke o stanju, v katerem se je avtomat ustavil (zadnji zapis v drugem rezultatu). To so: predhodno stanje in trenutno stanje, število sprejetih bajtov pri označitvi oziroma ustavitvi. Število sprejetih bajtov in trenutno stanje nam pomagata pri iskanju določene besede (začetek, konec besede).

Podatki o tem, v katerem stanju se je avtomat ustavil in koliko bajtov je sprejel, nam omogočajo nadaljevanje izvajanja v primeru, da smo avtomatu podali le del vhodnega niza ali da je zmanjkalo pomnilnika za rezultate.

```
typedef union
{
 uint64_t u64;
 struct
 {
  //stevilo sprejetih bajtov:
  uint64_t byte_offset     : 16;
  uint64_t extra_bits_high:  4;
  //predhodno stanje:
  uint64_t prev_node       : 20;
  uint64_t extra_bits_low :  2;
  uint64_t next_node_repl :  2;
  //trenutno stanje:
  uint64_t current_node    : 20;
 } s;
 struct
 {
  //stevilo sprejetih bajtov:
  uint64_t byte_offset     : 16;
  uint64_t extra_bits_high:  4;
  //predhodno stanje:
  uint64_t prev_node       : 20;
  uint64_t extra_bits_low :  2;
  //trenutno stanje z replikacijami:
  uint64_t curr_id_and_repl:22;
 } s2;
} cvmx_dfa_result1_t;
```

3.6 Podatki o avtomatu

Struktura vsebuje informacije o avtomatu, kot so: začetna pozicija v LLM, začetno stanje in tip avtomata (veliki ali mali format). Mali format avtomata vsebuje tudi število začetnih stanj in število označevalnih stanj. Iz teh dveh števil lahko enota razbere tip stanja.

```
typedef struct
```

```
{
 cvmx_llm_replication_t replication;
 cvmx_dfa_graph_type_t  type;
 uint64_t               base_address;
 union {
  struct {
   uint64_t             gxor          : 8;
   uint64_t             nxoren        : 1;
   uint64_t             nreplen       : 1;
   uint64_t             snrepl        : 2;
   uint64_t             start_node_id :20;
  };
  uint32_t              start_node;
 };
 int                    num_terminal_nodes;
 int                    num_marked_nodes;
} cvmx_dfa_graph_t;
```

Poglavje 4

Realizacija strojno pospešene knjižnice

Avtomat moramo najprej zgraditi iz regularnega izraza. Nato ga preoblikujemo tako, da bo namesto ugotavljanja pripadanja besede jeziku izvajal iskanje. Preoblikovan avtomat je nedeterminstičen in ima ϵ–prehode. Torej potrebujemo algoritem za prevajanje nedeterminističnih končnih avtomatov z ϵ prehodi v deterministične končne avtomate. Zato je algoritem, s katerim zgradimo avtomat iz regularnega izraza, bolj preprost. Zadostuje že algoritem, ki prevede regularni izraz v nedeterminsitični končni avtomat z ϵ–prehodi. Če ne bi bilo potrebno preoblikovati avtomata v iskalnega, bi bilo smiselno prevesti regularni izraz direktno v deterministični končni avtomat.

Minimizacije avtomata, ki izvaja iskanje, se med drugim zaradi morebitnega nastanka problema pri iskanju nizov, ki ustrezajo delom regularnih izrazov, ločenih z oklepaji, nismo lotili.

4.1 Preoblikovanje avtomata v iskalni avtomat

Narejenemu avtomatu (M) dodamo novo začetno stanje (q_0'). Dodamo mu en ϵ–prehod v staro začetno stanje (q_0) in en prehod iz novega začetnega stanja za vse možne vhodne črke nazaj v to novo začetno stanje. Dobljeni novi avtomat (M') sprejema vse besede, ki se končajo z besedo v jeziku prvotnega avtomata

(4.1).

$$\begin{array}{lcl} M & = & < Q,\Sigma,\delta,q_0,F > \\ M' & = & < Q',\Sigma,\delta',q_0',F > \\ Q' & = & Q \bigcup \{\ q_0'\} \\ \delta'(q,\sigma) & = & \left\{ \begin{array}{lcc} \{q_0'\} & ; & q = q_0' \wedge \sigma \neq \epsilon \\ \{q_0\} & ; & q = q_0' \wedge \sigma\ = \epsilon \\ \delta(q,\sigma) & ; & sicer \end{array} \right\} \end{array} \tag{4.1}$$

Avtomat enote za DKA prekine z izvajanjem, ko pride do končnega stanja. S pomočjo te enote zato lahko v taki obliki avtomata ugotovimo, ali je beseda, ki pripada jeziku regularnega izraza, v določenem nizu ali ne in kje se ta beseda konča.

4.2 Problem začetka najdenega niza

Pri iskanju določenega niza v nekem drugem nizu s pomočjo enote za DKA ni težko najti začetne pozicije najdenega niza. Potrebno je samo odšteti dolžino iskanega niza od pozicije konca najdenega niza.

Ker regularni izrazi predstavljajo tudi nize različnih dolžin, tega ne moremo vedno uporabiti. Zato spremenimo v označevalna vsa tista stanja, ki so najbolj oddaljena od začetnega in v katere ne pridemo po kateri koli drugi alternativni poti različne dolžine (tudi po ciklu). Poleg spremembe zapišemo še število bajtov sprejetih znakov pred začetkom v strukturo, ki smo jo poimenovali *označitev* in jo vsebuje vsako stanje. Vse to naredimo še, ko nimamo iskalnega avtomata, ker imamo po prevedbi v iskalni avtomat cikel že v začetnem stanju.

Po tem postopku lahko dobimo tudi več začetkov najdenega vzorca. Zadnji najdeni začetek zagotovo ustreza. Zato vzamemo kar ta začetek. Predhodni morebitni začetki niza so prav tako lahko pravilni. Da bi dobili enak rezultat kot originalna knjižnica PCRE, moramo vsak tak začetek preveriti z avtomatom za ujemanje ali predelati algoritem za prevedbo iz ϵNKA v DKA, ki bi posebej obravnaval prehod katere koli črke v začetnem stanju iskalnega avtomata. Implementacijo smo poenostavili in upoštevali samo zadnji dobljeni začetek najdenega niza. Zaradi tega dobimo drugačen rezultat kot originalna knjižnica.

4.3 Požrešno iskanje

Uporabili smo tudi samo nepožrešno iskanje. Z izvajanjem končamo, ko najdeni niz prvič ustreza regularnemu izrazu. Na ta način poenostavimo knjižnico

med tem ko originalna knjižnica poskuša sprejeti najdaljši niz, ki ustreza. Implementacija požrešnega iskanja je bolj zahtevna. Vse prehode v končna stanja iskalnega avtomata moramo povezati z enakovrednimi končnimi stanji običajnega avtomata. Iskanje se zaključi, ko ne obstaja prehod za vhodno črko ali ko zmanjka podatkov. Običajni in iskalni avtomat je potrebno najprej prevesti v DKA in ju šele potem združiti.

4.4 Vmesna struktura

Celotno prevajanje naredimo preko vmesne strukture – grafa stanj in prehodov. Vmesna struktura je povezan seznam stanj. Vsako stanje lahko vsebuje povezan seznam prehodov, seznam označitev, seznam stanj (za prevajanje iz ϵNKA v DKA), tip stanja (začetno, končno ali običajno), zaporedno številko stanja ali označitev prevajalnih algoritmov za reševanje problema ciklov.

Vsak prehod vsebuje vhodno črko (za interval so zgornja in spodnja črka), pri kateri le-ta obstaja, tip prehoda (neobčutljiv na velike ali male črke, ϵ-prehod, katerakoli vhodna črka, številka, črka, kateri koli znak, razen nova vrstica, interval ...), ciljno stanje in bitno masko za znake (4 po 64–bitna ne predznačena števila). Bitna maska mora imeti nastavljene bite pri posameznih indeksih glede na numerično vrednost vhodne črke in tipa prehoda. Le ta se določi kasneje, ko imamo že vmesno strukturo. Na ta način lahko poenostavimo implementacijo prevajanja v vmesno strukturo in tudi zmanjšamo število struktur prehodov.

Označitve so lahko tipa *začetek*, *konec*. Tip lahko razširimo tako, da označitev vsebuje tudi druge informacije, na primer za naslove funkcij, ki jih mora program klicati, če najde določeno besedo.

4.4.1 Prevajanje v vmesno strukturo

Prevajanje v vmesno strukturo se delno nanaša na [5].

Prevajalni algoritem v vmesno strukturo prevede niz, ki definira regularni izraz, v avtomat. Zaradi oklepajev uporablja sklad. Velikost sklada je nastavljena na 1000, kar zadostuje za 1000 stopenj oklepajev in s tem za običajen regularni izraz. Najprej naredimo začetno in končno stanje avtomata. Prav tako naredimo za vsak oklepaj končno in začetno stanje podavtomata, ki je ekvivalenten regularnemu izrazu v oklepaju. Narejena končna stanja podavtomatov potrebujemo, če imamo v izrazu znak za alternative. Zadnje stanje moramo v tem primeru povezati s končnim. Povežemo ga kar z ϵ–prehodom.

Začetno stanje pa potrebujemo zato, da ga lahko povežemo z avtomatom, ki je ekvivalenten delu regularnega izraza za znakom za alternativo.

Za zvezdico in vprašaj povežemo začetno stanje, ki ustreza regularnemu izrazu v oklepaju ali stanju regularnega izraza zadnjega znaka, s trenutnim stanjem. Začetno stanje določimo s spremenljivko *beforeState*, ki jo nastavimo za vsakim zaklepajem in za znakom za vhodno črko. Za zaklepajem spremenljivko *beforeState* nastavimo na začetno stanje, ki ustreza regularnemu izrazu v oklepaju, za znakom za vhodno črko pa spremenljivko *beforeState* nastavimo na predhodno stanje.

Prevajalni algoritem v vmesno strukturo doda prehodom samo znak in tip znaka. Na koncu ločena funkcija zgradi bitne maske za vse prehode.

Številnih posebnih znakov knjižnice PCRE nismo implementirali. Večina neimplementiranih literalov je trivialno dodati. Izstopa samo imenovana referenca nazaj. S pomočjo nje iščemo niz, ki ima dva ali več delov enakih. Tako iskanje pa preseže moč enote za DKA. Realizacija reference nazaj je kljub temu možna. Vendar pa se zaradi nje zmanjša hitrost iskanja, ker je ni mogoče realizirati direktno na enoti za DKA.

4.5 Prevedba iz ϵNKA v DKA

Ta del se delno nanaša na [5].

Algoritem ustvari nov avtomat. Najprej inicializira vse bite bitne maske znakov, ki je spremenljivka v algoritmu, na 1. Iz začetnega stanja poišče vsa stanja, v katera lahko pride samo z ϵ–prehodi. Ta stanja tvorijo novo začetno stanje DKA. Vsa najdena stanja zapišemo v seznam stanj novega začetnega stanja. Potem poiščemo največjo skupno bitno masko znakov. Največja skupna bitna maska se izračunana z ekskluzivnim ali iz bitov bitne maske največ prehodov, ki imajo nastavljen vsaj en skupen bit. Ta skupen bit mora biti nastavljen tudi v bitni maski algoritma. Nato ustvarimo novo stanje DKA iz vseh ciljnih stanj ϵNKA prehodov, ki so bili vključeni v izračun največje skupne bitne maske, in pogledamo, če stanje že obstaja na seznamu stanj. V primeru da ne obstaja, ga dodamo. Nato dodamo prehod s to največjo skupno bitno masko iz začetnega stanja v sedanje novo stanje. Na 0 postavimo tiste bite bitne maske algoritma, ki so v največji skupni bitni maski. Ta postopek ponavljamo, dokler nam ne zmanjka znakov. Do tega pride takrat, ko ima bitna maska v algoritmu vse bite na 0. Na koncu za vsak naslednji znak rekurzivno izvedemo algoritem za vsako dodano stanje.

Za vsako stanje DKA iz stanj ϵNKA v seznamih stanj je potrebno prepisati

še vse označitve. To naredimo tako, da pogledamo pri vsakem stanju DKA v seznam stanj tega stanja (stanja ϵNKA, ki definirajo to novo stanje DKA) in prepišemo označitve.

4.6 Pretvorba iz vmesne strukture v končno strukturo

Vsa stanja avtomata oštevilčimo. Tako dobimo številke ciljnih stanj prehodov. Oštevilčenim stanjem dodamo novo nedefinirano stanje, v katerega avtomat zaide, če za vhodno črko ne obstaja prehod. Za vsako stanje najprej za vse možne prehode zapišemo prehod v to nedefinirano stanje, potem pa za vsak prehod s pomočjo bitne maske in številke ciljnega stanja ustrezno popravimo zapisane prehode.

Na koncu naredimo še tabelo kazalcev na označitve. V tej tabeli je indeks številka stanja. Na ta način bomo hitro našli označitve pri izvajanju iskanja s tem avtomatom.

4.7 Izvajalni algoritem

Najprej prekopiramo niz, v katerem iščemo, v poseben prostor pomnilnika, kjer je znan fizični naslov. Prav tako potrebujemo prostor za rezultate. Vse te prostore dodelimo s klicem funkcije *cvmx_bootmem_alloc*. Iz kazalca, ki ga vrne funkcija *malloc*, namreč ne moremo dobiti fizičnega naslova. Prav tako ni nujno, da so podatki v pomnilniku. Lahko so na disku, pa tudi dodeljeni dejanski fizični prostor s klicem funkcije *malloc* ni nujno zvezen. Ker nimamo funkcije za sproščanje pomnilnika dodeljenega z funkcijo *cvmx_bootmem_alloc*, smo napisali funkcije, ki upravljajo z dodeljevanjem pomnilnika z znanim fizičnim naslovom.

Po kopiranju kličemo ukaze za zagon avtomata. Po končanju izvajanja iz rezultatov preberemo stanja in položaje v nizu. Za vsako stanje dobimo označitve iz tabele. S pomočjo označitev, položajev v nizu, stanj in tipa označitve dobimo začetek in konec niza. Če je označitev več, upoštevamo zadnjo. Če so na vhodu še ostali znaki in avtomat še ni prišel v končno stanje, ponovno izvedemo algoritem.

4.8 Funkcije za upravljanje s pomnilnikom

Potrebujemo dvakrat toliko mest za rezultate in nize, po katerih iščemo, kot je možnih izvajalnih niti enote za DKA. Potrebno je namreč kopirati vhodni niz v dodeljen prostor ali analizirati rezultate, medtem pa lahko izvajamo tudi nit na enoti za DKA.

Uporaba ene niti programa za dostop do enote za DKA je nesmiseln za strojno pospešitev, ker imamo več možnih izvajalnih niti enote za DKA. Zato do enote za DKA dostopa več niti programa hkrati. Torej moramo nitim tudi učinkovito dodeljevati vire. Dodeljevanje realiziramo s pomočjo tabele in dveh kazalcev. Prvi je za branje kazalcev podatkovnih blokov, drugi pa za pisanje. Za velikost dodeljenega bloka smo izbrali 4kB.

4.9 Dodeljevanje virov

Za sinhronizacijo niti smo uporabili knjižnico Pthread[7]. Dodeljevanje blokov pomnilnika posamezni izvajalni niti programa, ki je v klicu funkcije za iskanje naše knjižnice, smo implementirali s pomočjo semaforja in muteksa. S semaforjem smo namreč rešili problem, kako knjižnica ugotovi, kdaj ji zmanjka pomnilniških blokov. Dodeljevanje dela pomnilnika pa je kritična sekcija, ker lahko več hkratnih dodeljevanj pokvari tabelo prostih lokacij. Zato smo uporabili muteks.

Za zagon avtomata smo potrebovali semafor. Tako smo bolje določili čas, pri katerem avtomat konča z izvajanjem. Ker funkcija za zagon avtomata ne blokira procesa oziroma niti, dokler se enota za DKA izvaja, smo čakanje rešili z zaposlenim čakanjem. Vendar je bil realni in procesorski čas izvajanja knjižnice še daljši kot pri originalni knjižnici. Zato smo poskusili s preklopom niti po klicu funkcije za zagon izvajanja avtomata. Vendar ni bilo razlike v času izvajanja. Šele, ko smo uporabili funkcijo *usleep*[8], ki zamrzne nit za določen čas v mikrosekundah podan v argumentu, je bilo izvajanje glede na procesorski čas hitrejše kot pri originalni knjižnici.

Ker je maksimalna možna hitrost iskanja 4Gb/s, izvajamo lahko do 16 vzporednih iskanj, in ker imamo pomnilniški prostor za eno iskanje velikosti 4kB, smo najmanjši čas enega izvajanja avtomata lahko izračunali (enačba 4.2).

$$\begin{aligned}
\textit{propustnost} &= 4\tfrac{Gb}{s} = 0{,}5\tfrac{GB}{s} = 500\tfrac{MB}{s} \\
\textit{propustnost na enoto} &= \tfrac{\textit{propustnost}}{\textit{število enot}} \\
&= \tfrac{500\frac{MB}{s}}{16} = 31{,}25\tfrac{MB}{s} \\
&= 31250\tfrac{kB}{s} = 31{,}25\tfrac{B}{\mu s} \\
\textit{čas na enoto} &= \tfrac{\textit{velikost bloka}}{\textit{propustnost na enoto}} \\
&= \tfrac{4kB}{31{,}25\frac{B}{\mu s}} = \tfrac{4000B}{31{,}25\frac{B}{\mu s}} \\
&= 128\mu s
\end{aligned} \tag{4.2}$$

Rezultate izračuna smo uporabili za argument pri klicu funkcije *usleep*.

4.10 Metode naše knjižnice za regularne izraze

Ta del se delno nanaša na [2] in [1].

Notranjo zgradbo knjižnice smo razdelili na 3 nivoje:

- združljivi vmesnik za regularne izraze,
- vmesno strukturo in
- upravljanje s strojno opremo.

Zaradi take zgradbe lahko na primer zamenjamo strojno enoto DKA za kakšno drugo, napišemo npr. vmesnik za drugo knjižnico za regularne izraze ali pa vmesnik za iskanje samih besed.

Knjižnica se prav tako lahko razdeli na 3 podknjižnice. S tem je dosežena še bolj fleksibilna zgradba. Tako lahko namreč omogočamo menjavo strojnega dela knjižnice med emulacijo in strojno enoto brez naknadnega prevajanja knjižnice.

4.10.1 Združljivi vmesnik za regularne izraze

V vmesniku so vse metode, ki jih program neposredno kliče. Ta nivo potem kliče druga dva nivoja knjižnice. V vmesniku je prevajalnik za prevajanje regularnega izraza v avtomat in združljive metode poljubne knjižnice za regularne izraze. V diplomski nalogi smo v tem nivoju implementirali prevajalnik v vmesno strukturo in metode, ki so delno združljive s knjižnico PCRE.

4.10.2 Metode naše knjižnice in knjižnice PCRE

Implementirali smo dve metodi *pcre_compile2* (prevajanje regularnega izraza v obliko za izvajanje) in *pcre_exec* (izvajanje regularnega izraza).

Ostale metode originalne knjižnice PCRE, ki jih nismo implementirali so:

- *pcre_compile* (trivialna sprememba *pcre_compile2*),
- *pcre_study* (za analizo regularnega izraza po prevajanju, ki omogoča hitrejše izvajanje iskanja),
- *pcre_dfa_exec*(najde vse možne nize regularnega izraza, ne samo enega),
- metode za delo z deli nizov, ki ustrezajo regularnim izrazom v oklepajih, in njihovimi morebitnimi referencami,
- *pcre_fullinfo*,
- *pcre_info*,
- *pcre_refcount*,
- *pcre_config*,
- *pcre_version*,
- *pcre_malloc*,
- *pcre_free*,
- *pcre_stack_malloc*,
- *pcre_stack_free*,
- *pcre_callout*.

pcre_compile2

```
pcre* pcre_compile2(
      const char *pattern //regularni izraz,
      int options //opcije regularnega izraza,
      int *errorcodeptr //kazalec na kodo napake,
      const char **errptr //kazalec na opis napake,
      int *erroffset,
      const unsigned char *tableptr
      )
```

Funkcija preveri, če je prvi znak v regularnem izrazu enak znaku ^. Če je, ga izbriše iz regularnega izraza in postavi zastavico *PCRE_ANCHORED*.

Po prevajanju (klicu funkcije *pcre_complile_internal*) doda označitve v najbolj oddaljena stanja od začetnega, v katera ni mogoče priti po nobeni drugi poti. V označitve zapiše, da so tipa začetek in koliko že sprejetih bajtov bo potrebno odšteti od števila sprejetih bajtov rezultata označevalnega stanja, da dobimo začetek niza. Prav tako označi končna stanja, ki pa niso premaknjena. Vendar zaradi enotnosti vseeno vsebujejo informacijo o številu bajtov, ki jih bo potrebno odšteti. Označitve so v povezani strukturi zato, da jih je lahko več. Že pri navadnem nizu se zgodi, da so označitve začetkov in koncev v istem stanju.

Nato funkcija preveri, če je prisotna opcija *PCRE_ANCHORED*. Če ni, doda novo začetno stanje, ki ima povraten prehod za katero koli vhodno črko in ϵ–prehod v staro začetno stanje (glej prevedbo v iskalni avtomat).

Na koncu se avtomat prevede v DKA. Po tej prevedbi se avtomat zapiše v obliko, primerno za izvajanje na strojni enoti za DKA.

pcre_internal_compile

```
pcre *pcre_complile_internal(
      const char *pattern //regularni izraz,
      int flags //opcije regularnega izraza)
```

Metoda prevede regularni izraz po principu skladovnega avtomata v vmesno strukturo. Sklad namreč potrebuje zaradi oklepajev. Nastavljenega imamo na velikost 1000, kar nedvomno zadostuje za kateri koli običajni regularni izraz knjižnice PCRE. Prevajanje smo podrobneje opisali že v razdelku **Prevajanje v vmesno strukturo**.

Prevajalni algoritem prehodom v vmesno strukturo doda samo znak in opis znaka. Na koncu ločena funkcija *make_bitmask* zgradi bitno masko za vsak znak posebej.

pcre_exec

```
int pcre_exec(
    const pcre *code,//preveden regularni izraz
    void *extra,//dodatni podatki
    void *subject, //niz, v katerem iscemo
    int length, //dolzina
    int startoffset,//zacetni odmik v nizu
```

```
    int options, //nastavitve iskanja
    int *ovector, //vektor rezultatov
    int ovecsize //velikost vektorja);
```

Funkcija poišče začetek in konec niza, ki ustreza prevedenemu regularnemu izrazu *code*, v nizu *subject*.

Najprej kliče funkcijo *dfaMatch*. Nato obdela vse označitve po številkah in po tipu začetek ali konec. Število bajtov prenese v tabelo *ovector*. Če avtomat še ni zaključil z izvajanjem, potem ponovno kliče funkcijo *dfaMatch*.

4.10.3 Vmesna struktura

Vmesna struktura nam olajša preoblikovanje avtomata v iskalni in prevedbo ϵNKA v DKA. V vmesni strukturi imamo tri glavne podatkovne tipe:

- označitev,
- stanje,
- prehod.

Označitve potrebujemo za obdelavo rezultatov označevalnih stanj. Lahko so tipa *začetek*, *konec* ali pa *posebna dodatna informacija*. Vsebujejo tudi zapis, za koliko znakov so premaknjene *bytes*. Označitve tipa posebne dodatne informacije vsebujejo tudi podatke označitve *data*. Posebnih dodatnih informacij v diplomski nalogi nismo uporabili.

```
struct mark //oznacitev
{
    //tip oznacitve:
        int type;
    //stevika oznacitve:
        int id;
    //premaknjenost oznacitve:
        int bytes;
    //podatki oznacitve:
    void * data;
    //naslednja oznacitev:
        struct mark * next;
};
```

Stanja so lahko tipa *začetno*, *končno* ali *normalno stanje*. Vsebujejo seznam prehodov *transitions*, zaporedno številko stanja oziroma označitev za razrešitev

problema ciklov nekaterih algoritmov *num*, morebitno ekvivalentno množico stanj ϵNKA *ls* in označitve *mark*.

```
struct state //stanje
{
    //tip stanja:
        char type;
    //seznam prehodov:
        struct transition* transitions;
    //naslednje stanje:
        struct state* next;
    //oznacitev za cikle
    //ali stevilka stanja:
        int num;
        //seznam oznacitev:
        struct mark * mark;
    //seznam stanj epsilonNKA
        struct statelist* ls;
};
```

Prehodi vsebujejo vhodno črko ali interval *val*, tip prehoda *type*, ciljno stanje *dst* in bitno masko *bitmask*. Prav tako imajo okrajšavo za večkratni prehod istega znaka *times*. Tako za več enakih znakov ni potrebno narediti novih prehodov in stanj. Te okrajšave nismo uporabili. Okrajšava tudi ni podprta pri prevajanju v ciljni avtomat.

```
struct transition //prehod
{
    //tip prehoda:
        char type;
    //crka ali interval prehoda:
        int val;
    //minimalno stevilo enakih
    //znakov potrebnih za prehod
    //neuporabljeno:
        char times;
    //ciljno stanje:
        struct state* dst;
    //naslednji prehod:
        struct transition* next;
    //bitna maska crk prehoda:
```

```
        long long bitmask[4];
};
```

Funkcije za delo z vmesno strukturo so:

- makeDFA_from_eNFA

  Prevede ϵNKA v DKA.

- makeBitmask

  Zgradi bitno masko iz tipa in vhodne črke prehoda za vse prehode v avtomatu.

- addTransition

  Doda prehod v povezan seznam prehodov.

- newState

  Naredi in vrne novo stanje.

- freeStateList

  Sprosti seznam stanj, ki so v vsakem stanju kot seznam stanj: Vsako stanje DKA namreč po prevedbi iz ϵNKA v DKA vsebuje seznam ekvivalentnih stanj ϵNKA temu stanju iz DKA.

- printGraph

  Izpiše strukturo avtomata.

makeDFA_from_eNFA

```
STATE makeDFA_from_eNFA(STATE graph);
```

Algoritem ustvari nov DKA iz ϵNKA . Iz začetnega stanja, poišče vsa stanja s katerimi lahko pridemo preko ϵ–prehodov. Ta stanja tvorijo novo začetno stanje DKA. Zapišemo jih v seznam stanj začetnega stanja.

Na koncu za vsako stanje DKA iz stanj ϵNKA v seznamih stanj prepišemo še vse označitve.

Funkcije, ki jih ta funkcija uporablja, so:

- nextState,
- stateSame,
- getAllEpsilon in
- setAllMarks.

nextState

```
STATE* nextState(
        STATE states //seznam stanj,
        STATE state //trenutno stanje,
        int *mark //oznacitev za odpravo ciklov,
        STATE* tail //konec seznama stanj)
```

Ta funkcija izvaja rekurzivni del algoritma za prevedbo ϵNKA v DKA.

stateSame

```
STATE stateSame(
       STATE states, //seznam stanj
       STATE state //stanje)
```

S pomočjo funkcije *state_eq* ta funkcija poišče že znano stanje DKA v povezani strukturi stanj pri prevajanju ϵNKA v DKA. Če stanje funkcija najde, vrne staro obstoječe stanje, novo pa sprosti.

state_eq

```
int state_eq(
    STATE s1 //stanje,
    STATE s2 //stanje)
```

Ugotovi, če sta dve stanji DKA enaki glede na stanja ϵNKA. Torej preveri, če vsebujeta isti stanji ϵNKA v seznamu stanj. Ker je seznam stanj lahko poljubno urejen, razbere funkcija enakost tako, da pogleda, če je seznam enega stanja podmnožica drugega in obratno.

setAllMarks

```
void setAllMarks(STATE st)
```

Za vsako stanje v DKA *st* ta funkcija prenese vse označitve iz stanj, ki so na seznamu ekvivalentnih stanj ϵNKA zapisane pri vsakem stanju.

getAllEpsilon

```
int getAllEpsilon(
    STATE state,//stanje
```

```
int* mark,//oznacitev za odpravo cikla
struct statelist*** tail //kazalec na kazalec
//na zadnji kazalec seznama stanj
)
```

Funkcija poišče vsa stanja, v katera pridemo preko ϵ–prehodov.

Cikle rešuje s pomočjo vrednosti *num*, ki je v strukturi stanja. Vsa stanja zapše na konec seznama *tail*. Funkcija pogleda tudi, če je kakšno stanje končno in vrne rezultat, da je kot 1 in ne kot 0.

4.10.4 Funkcije za delo s strojno opremo

Za delo s strojno opremo imamo dve funkciji. Ena prevede vmesno strukturo v strukturo, primerno za izvajanje na strojni opremi, druga pa je namenjena izvajanju avtomata:

- makeHardwareDFA,
- dfaMatch.

makeHardwareDFA

```
int makeHardwareDFA(
    STATE start,//seznam stanj DKA
    void **data1 //prevedeni avtomat
     )
```

Ta funkcija zapiše avtomat v LLM. Za format zapisa smo izbrali veliki format. Funkcija najprej oštevilči stanja. Prvo prosto številko stanja uporabi za posebno, nedefinirano stanje. Pri vsakem stanju zapiše vse prehode v nedefinirano stanje. Nedefinirano stanje pri vsakem prehodu označi kot končno. Za vsak definiran prehod potem ustrezno popravi ciljno stanje.

Funkcija si tudi sproti zapomni naslov, do katerega je že zapisala avtomat. Tako vemo, kateri naslovi v LLM so še prosti za nadaljnje pisanje. Funkcija prepiše označitve stanj v tabelo, zato jih lahko pri iskanju hitro najdemo. Tabela označitev prav tako vsebuje dodano nedefinirano stanje, da pri branju označitev ne pride do napake.

dfaMatch

```
int dfaMatch(void *automata,//prevedeni avtomat
        void** input,//vhodni niz
        int *datalen, //velikost niza
        struct mark * matches[],//oznacitve
        int offsets[],//pozicije oznacitev
        int* nresults //stevilo rezultatov
        )//vrne zadnje stanje
```

Ta funkcija začne izvajati avtomat. Če je vhodni niz večji od 4kB, kolikor je veliko eno mesto za vhodni niz, potem v zanki nadaljuje z izvajanjem.

Ko enota za DKA opravi svoje delo, funkcija kot rezultat zapiše označitve v seznam označitev *matches* skupaj s pozicijami *offsets* in število označitev v *nresults*. Popravi tudi kazalec vhodnega niza *input* in velikost niza *datalen* ter vrne številko zadnjega stanja. Tako lahko funkcijo ponovno zaženemo, če zmanjka prostora za označitve.

4.11 Razširjeni regularni izrazi knjižnice PCRE in naše knjižnice

Ta del povzema regularne izraze knjižnice in se nanaša večinoma na [2].

Regularni izrazi knjižnice PCRE so razširjeni regularni izrazi. Ime pove, da so združljivi z regularnimi izrazi v programskem jeziku Perl. Vendar regularni izrazi knjižnice PCRE niso enaki Perl-ovim, temveč se nekoliko razlikujejo.

Poleg običajnih metaznakov teoretičnih regularnih izrazov, kot so $+, *, ^n$, ima knjižnica pa tudi Perl znake, ki omogočajo krajši zapis. To so npr.: $+, ?, \{m,n\}$.

Regularni izraz prav tako vsebuje spremenljivke, s katerimi preverjamo, če je določeni del že najdenega niza enak drugemu delu. V tem knjižnica PCRE in Perl presežeta moč regularnih izrazov in tudi moč enote za DKA. Zato te razširitve nismo implementirali. Implementacija takega regularnega izraza je možna na način, da to primerjavo izvedemo kar na centralni procesni enoti namesto na enoti za DKA.

Prav tako nismo implementirali vseh metaznakov. Implementacija le-teh je za večino trivialna.

4.11.1 Določila

Ta del se tudi nanaša na [1].

Določila (angl. *modifiers*) so znaki, ki povedo dodatne lastnosti regularnega izraza. Programski jezik Perl jih pozna v samem delu regularnega izraza. Knjižnica PCRE pa jih uporablja kot dodatne opcije pri prevajanju regularnega izraza (tabela 4.1). Te opcije so v nekaterih primerih drugačne od Perl-ovih.

Implementirali smo samo določilo *PCRE_CASELESS* za iskanje, neobčutljivo na male in velike črke.

4.11.2 Običajni metaznaki

Metaznaki so znaki, ki ne predstavljajo vhodnih črk avtomata ali črk jezika regularnega izraza, temveč imajo nek poseben pomen. Standardni metaznaki knjižnice, ki imajo enak pomen kot pri ukazu *egrep*, so:

- \

  Naslednji metaznak obravnavaj kot navaden znak; to smo implementirali.

- ^

  Najdi na začetku vrstice oziroma na začetku niza; to smo tudi implementirali.

- .

  Poljuben znak, vendar glede na določilo *s* ne znak za novo vrstico ali pa tudi; implementirali smo ga samo kot poljuben znak.

- *$*

  Najdi na koncu vrstice oziroma niza; tega metaznaka nismo implementirali.

- |

  Alternativa; alternativo smo implementirali.

- *()*

  Oklepaji; oklepaje smo implementirali, vendar brez razčlembe niza v podnize, ki jih predstavljajo deli regularnih izrazov v oklepajih.

- *[]*

  Razredi znakov; razrede smo implementirali (črke številke, intervali znakov).

Določilo v Perl-u	Konstanta PCRE	Opis
m	PCRE_MULTILINE	išči v več vrsticah pri metaznakih ^ in $
s	PCRE_DOTALL	obravnavaj večvrstiči niz kot enovrstični. Metaznak . pomeni kateri koli znak
i	PCRE_CASELESS	išči neobčutljivo na male in velike črke
x	PCRE_EXTENDED	razširitev, ki dovoljuje presledke in komentarje
p		ohrani niz
g in c		išči globalno in ohrani trenutni položaj po neuspelem iskanju
	PCRE_DOLLAR_ENDONLY	pri $ išči samo na koncu
	PCRE_EXTRA	striktno upoštevaj ubežne sekvence
	PCRE_UTF8	rokuj z znaki UTF8
	PCRE_UNGREEDY	zamenjaj pomen * in *?
	PCRE_NO_AUTO_CAPTURE	onemogoči zajem podnizov, ki jih predstavljajo deli regexa v oklepajih

Tabela 4.1: Določila

Privzeto Perl garantira, da pri uporabi metaznaka ^ najde niz le v prvi vrstici na začetku niza, za znak *$* pa na koncu niza ali pa pred morebitnim znakom za novo vrstico, ki je na koncu niza. Za pravilno delovanje pa ta dva metaznaka potrebujeta posebno določilo. Naša knjižnica podpira samo osnovni znak za začetek vrstice in garantira njegovo pravilno delovanje samo v prvi vrstici.

Kvantifikatorji (angl. *quantifiers*) so:

- ***

  Najdi ujemanje večkrat ali pa preskoči; to smo implementirali.

- *+*

  Najdi ujemanje enkrat ali večkrat; to smo implementirali.

- *?*

  Najdi ujemanje ali pa preskoči; to smo implementirali.

- {*n*}

  Najdi ujemanje natanko n-krat; to nismo implementirali.

- {*n*,}

  Najdi ujemanje vsaj n-krat; to nismo implementirali.

- {*n*,*m*}

  Najdi ujemanje vsaj n-krat do največ m-krat; to nismo implementirali.

Pri originalni knjižnici in Perl-u so zaviti oklepaji obravnavani kot navadni znaki, če se pojavijo v drugem kontekstu. Spodnjo mejo v zavitih oklepajih je potrebno obvezno navesti, vendar je dovoljena tudi *0* (npr. {*0,5*}). Naša knjižnica zavitih oklepajev ne podpira.

Za originalno knjižnico je kvantificiran (pod)niz privzeto „požrešen“. To pomeni, da bo knjižnica za regularne izraze poskušala najti najdaljše ujemanje, preden bo nadaljevala z ostalim delom regularnega izraza. Če nadaljnji del ne ustreza, se originalna knjižnica vrača nazaj. Naša knjižnica pa sproti preverja možen začetek niza. Zato ne potrebuje vračanja. Najde namreč zadnji možen začetek in prvi možen konec najdenega niza. S tem najde le prvi niz, ki se ustreza regularnemu izrazu. Najdeni niz je z začetne in končne strani najkrajši, ki ustreza regularnemu izrazu.

Za nepožrešno iskanje na koncu niza imata originalna knjižnica in Perl metaznake, ki jim sledi znak *?*:

- *?*

  Najdi ujemanje večkrat ali pa preskoči, ampak nepožrešno.

- *+?*

  Najdi ujemanje enkrat ali večkrat, ampak nepožrešno.

- *??*

  Najdi ujemanje ali pa preskoči ta del, ampak nepožrešno.

- {*n*}*?*

  Najdi ujemanje natanko n-krat, ampak nepožrešno.

- {*n,*}*?*

  Najdi ujemanje vsaj n-krat, ampak nepožrešno.

- {*n,m*}*?*

  Najdi ujemanje vsaj n-krat in največ do m-krat, ampak nepožrešno

Teh metaznakov nismo implementirali.

Knjižnica pozna tudi „posesivne“ regularne izraze, ki ne dovoljujejo vračanja. Ti so označeni tako, da jim sledi znak *+*:

- **+*

  Najdi ujemanje večkrat ali pa preskoči, vendar se ne vračaj.

- *++*

  Najdi ujemanje enkrat ali večkrat, vendar se ne vračaj.

- *?+*

  Najdi ujemanje ali pa preskoči ta del, vendar se ne vračaj.

- {*n*}*+*

  Najdi ujemanje natanko n-krat, vendar se ne vračaj.

- {*n,*}*+*

  Najdi ujemanje vsaj n-krat, vendar se ne vračaj.

- {*n,m*}*+*

  Najdi ujemanje vsaj n-krat, vendar največ m-krat vendar se ne vračaj.

Teh metaznakov tudi nismo implementirali, ker so zaradi omenjenih razlik pri požrešnosti neuporabni.

Implementirali pa smo naslednje ubežne sekvence:

- $\backslash t$
  Tabulator.
- $\backslash n$
  Nova vrstica.
- $\backslash r$
  Vrnitev.
- $\backslash f$
  Pomik navzdol.
- $\backslash a$
  Alarm (zvonec).
- $\backslash e$
  Escape (pobeg).
- $\backslash 033$
  Osmiški znak.
- $\backslash x1B$
  Šestnajstiški znak.
- $\backslash cK$
  Kontrolni znak.

Originalna knjižnica in Perl podpirata še:

- $\backslash x\{263a\}$
  Dolgi šestnajstiški znak.
- $\backslash N\{ime\}$
  Imenovani Unicode znak.

- $\backslash l$
  Naslednji znak je mala črka.
- $\backslash u$
  Naslednji znak je velika črka.
- $\backslash L$
  Naslednji znaki so male črke, dokler ni sekvence $\backslash E$.
- $\backslash U$
  Naslednji znaki so velike črke, dokler ni sekvence $\backslash E$.
- $\backslash E$
  Konec določila črk.
- $\backslash Q$
  Metaznaki so navadni znaki, dokler ni sekvence $\backslash E$.
- $\backslash w$
  Besedni znak; alfanumerični znak ali znak _.
- $\backslash W$
  Nebesedni znak.
- $\backslash s$
  Presledek.
- $\backslash S$
  Vse, razen presledka.
- $\backslash d$
  Števka.
- $\backslash D$
  Vse, razen števke.
- $\backslash pP$
  Imenovana lastnost; za daljša imena lahko uporabimo $\backslash p\{Prop\}$.

- $\backslash PP$

  Vse, razen P.

- $\backslash X$

  Najdi razširjeno Unicode (kombinirano zaporedje znakov) ekvivalento *?:\PM\pM**.

- $\backslash C$

  En znak C(oktet) tudi, če iščemo po Unicode; ta sekvenca razbije znake v bajte in lahko povzroči napačna UTF-8 zaporedja bajtov. Znak ni podprt pri iskanju nazaj.

- $\backslash 1$

  Klicanje nazaj na določeno skupino; lahko je poljubno pozitivno celo število.

- $\backslash g1$

  Klicanje nazaj na določeno skupino ali prejšnjo skupino.

- $\backslash g$ {*-1*}

  Skupina iz prejšnjega klica; velja za poljubno negativno celo število. Lahko je tudi brez zavitih oklepajev. Vendar je uporaba varnejša z oklepaji.

- $\backslash g\{name\}$

  Imenovana referenca nazaj.

- $\backslash k<name>$

  Imenovana referenca nazaj.

- $\backslash K$

  Iz najdenih nizov in podnizov izvzemi vse, kar je na levi.

- $\backslash v$

  Navpični presledek.

- $\backslash V$

  Vse, razen navpični presledek.

- $\backslash h$

  Vodoravni presledek.

- $\backslash H$

  Vse, razen vodoravni presledek.

- $\backslash R$

  Nova vrstica; $\backslash R$ avtomatično najde novo vrstico tudi *\x0D\x0A*. Ekvivalenten je *(?>\x0D\x0A?|[\x0A-\x0C\x85\x{2028}\x{2029}])*.

4.11.3 Razredi znakov

Za razrede znakov uporabljata Perl in originalna knjižnica POSIX sintakso (*[:razred:]*). To podpira tudi naša knjižnica.

Oglati oklepaji skupaj z dvopičjem so literali in morajo biti vedno uporabljeni v izrazu razreda znakov. Torej je regularni izraz *[[:alpha:]]* pravilen, medtem ko *[:alpha:]* ni pravilen.

Na voljo so naslednji razredi:

- *alpha*,
- *alnum*,
- *blank*,
- *cntrl*,
- *digit*,
- *graph*,
- *lower*,
- *print*,
- *punct*,
- *space*,
- *upper*,
- *word*,

- *xdigit*.

Primer: *[:upper:]* bo našel vsak znak, ki je velika črka. Znaki *[]* so del konstrukta *[::]* in ne celotnega razreda znakov. Torej *[01[:alpha:]%]* bo našel 0, 1, katero koli črko abecede in znak za procent.

Ekvivalence z Unicode $\backslash p\{\}$ konstrukti prikazuje Tabela 4.2. Unicode kon-

Razred [[: ... :]]	Unicode konstrukt $\backslash p\{...\}$	Ubežna sekvenca \
alpha	IsAlpha	
alnum	IsAlnum	
ascii	IsASCII	
blank		
cntrl	IsCntrl	
digit	IsDigit	\d
graph	IsGraph	
lower	IsLower	
print	IsPrint	
punct	IsPunct	
space	IsSpace, IsSpacePerl	\s
upper	IsUpper	
word	IsWord	
xdigit	IsXDigit	

Tabela 4.2: Ekvivalence razredov znakov z Unicode konstrukti in nekaterimi ubežnimi sekvencami

struktov $\backslash p\{\}$ nismo implementirali.

Če je uporabljena pragma locale in ne pragma UTF–8, potem so razredi konstrukta *[::]* v skladju s C funkcijo *isalpha()*, razen za razreda *word* in *blank*. Naša knjižnica je samo v skladju s C funkcijo *isalpha()*. Prav tako naša knjižnica podpira samo znake dolžine en bajt.

Ostali imenovani razredi so še:

- cntrl

  Poljuben kontrolni znak; običajno znaki ne dajo nobenega izhoda, ampak upravljajo terminal. Na primer nova vrstica in vračalka sta kontrolna znaka. Vsi znaki katerih numerična vrednost ne presega 32, so običajno kontrolni (torej prvih 32 znakov). Kontrolni je običajno še znak DEL (127 po numerični vrednosti). To velja za ASCII, latinski ISO znakovni nabor in Unicode.

- graph

  Poljubna črka, števka ali ločilo.

- print

  Poljubna črka, števka, ločilo ali presledek.

- punct

  Poljubno ločilo.

- xdigit

  Poljubna šestnajstiška števka.

Znakovni razredi s konstruktom *[::]* se lahko pred imenom negirajo z znakom ^. To je Perl-ova razširitev POSIX standarda pri znakovnih razredih. Tako negiranje smo implementirali tudi pri naši knjižnici.

Knjižnica podpira POSIX znakovne razrede v znakovnem razredu. Znakovni razredi [.cc.] in [=cc=] so prepoznani, vendar nepodprti (povzročijo napako) s strani Perl–a. V naši knjižnici jih ne prepoznamo.

4.11.4 Trditve

Perl podpira tudi trditve brez dolžine, ki jih nismo implementirali:

- $\backslash b$

  Najdi na robu besede.

- $\backslash B$

  Najdi povsod, samo na robu besede ne.

- $\backslash A$

  Najdi samo na začetku niza.

- $\backslash Z$

  Najdi samo na koncu niza ali koncu niza pred novo vrstico.

- $\backslash z$

  Najdi samo na koncu niza.

- $\backslash G$

  Najdi samo na *pos()*, torej na koncu prejšnje najdene pozicije.

Končni rob besede ($\backslash b$) je položaj med dvema znakoma. Prvi je besedni ($\backslash w$), drugi pa je nebesedni ($\backslash W$) ali obratno.

Poglavje 5

Meritve

Meritve hitrosti smo izvedli na omenjenem procesorju pod operacijskim sistemom Linux, ki je porabljal vseh 16 jeder.

Testni program je napisan večnitno. Vsaka nit izvaja iskanje nad danim nizom. Za niti smo uporabili knjižnico *Pthread*. Vse niti uporabljajo svoj naslovni prostor nizov (nizi so prekopirani), v katerem iščemo. Program je dinamično povezan s knjižnico. Zato lahko to knjižnico tudi zamenjamo brez prevajanja.

Zanimalo nas je, koliko procesorskega in realnega časa porabi originalna knjižnica in koliko naša. Preverili smo tudi, pri katerih regularnih izrazih je naša knjižnica boljša, pri katerih pa je po obeh kriterijih(po realnem in procesorskem času) slabša od originalne.

Izvedli smo več meritev. Merili smo čas izvajanja naše knjižnice s spanjem, knjižnice z zasedenim čakanjem, knjižnice, ki ne izvaja iskanja, in originalne knjižnice. Knjižnico, ki ne izvaja iskanja, smo uporabili za meritev, koliko časa izvaja testni program druge operacije brez iskanja.

Čas izvajanja testnega programa smo izmerili s pomočjo ukaza *time*.

5.1 Postopek

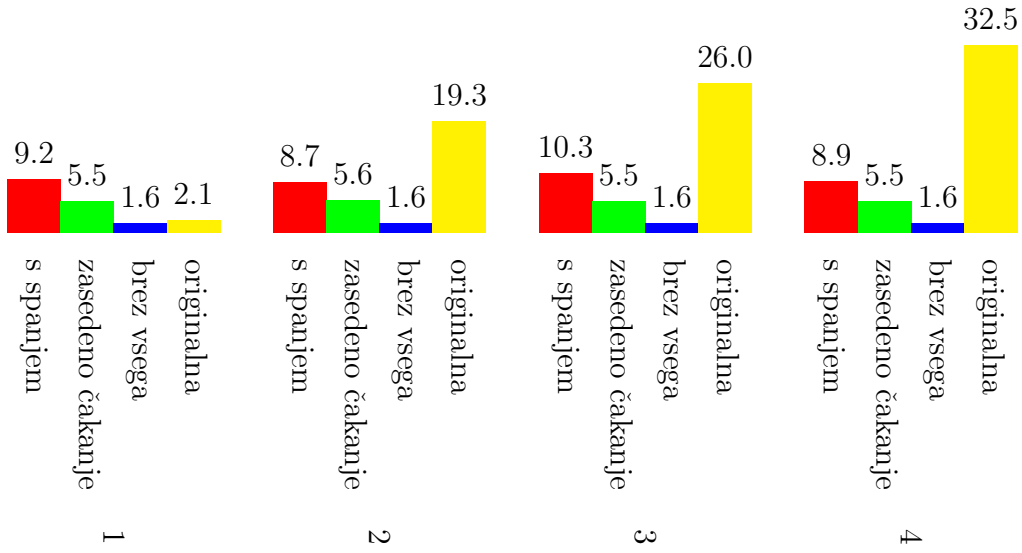
Testni program smo napisali tako, da preko ukazne vrstice prejme velikost vhodnega niza, v katerem izvaja iskanje z regularnim izrazom. Za vhodne nize smo uporabili kar nize, ki vsebujejo vse znake NULL (*\0*). Če bi uporabili poljuben znak, bi knjižnica lahko niz našla. Zato pa ne bi mogli izmeriti časa, ki ga knjižnica porabi, da preišče celoten niz.

Testni program najprej dinamično dodeli del pomnilnika za niz in ga inicializira. Regularni izraz prevede glavna nit programa. Na koncu ta glavna nit

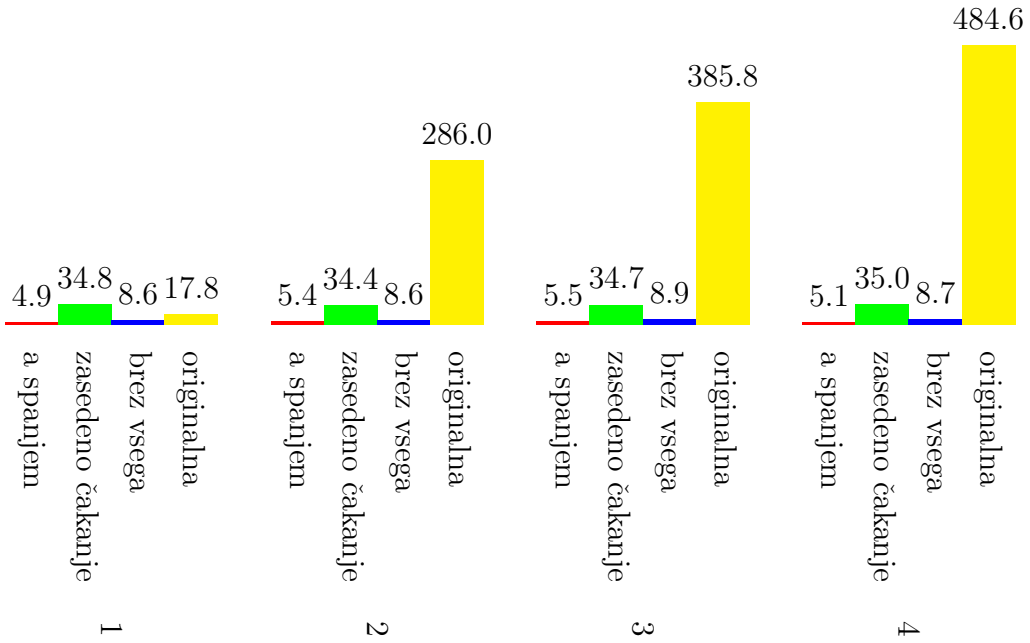
zažene iskalne niti, ki prekopirajo vsebino niza in iščejo v kopiji s prevedenim regularnim izrazom.

Določanje časa in povezovanje posameznih različic knjižnic s testnim programom nadzira skripta, napisana za lupino sh (glej Dodatek C). Ta začasno premakne originalno knjižnico na drugo lokacijo, na njeno mesto pa prekopira našo. Tako povezovalnik poveže testni program z našo knjižnico. Nato skripta prepiše našo knjižnico s še drugimi različicami naše knjižnice in izvede testni program. Na koncu prekopira nazaj originalno knjižnico in še zadnjič izvede program. Testni program se vsakič izvede s programom *time* ter standardni izhod testnega programa preusmeri v */dev/null*. Na ta način ne pride do zakasnitve zaradi čakanja pri pisanju na standardni izhod in tudi ne odvečnih informacij, ki so izpisi v testnem program za potrebe razhroščevanja knjižnice.

Razlike v hitrosti smo preverili na različnih lastnostih regularnih izrazov, kot so alternative, dolgi niz, oklepaji ... Nato smo se odločili za natančnejše meritve alternativ.



Slika 5.1: Odvisnost realnega časa izvajanja testnega programa glede na število alternativ (Številke spodaj pomenijo število alternativ, nad stolpci pa je realni čas izvajanja testnega programa v sekundah.)



Slika 5.2: Odvisnost procesorskega časa izvajanja testnega programa glede na število alternativ (Številke spodaj pomenijo število alternativ, nad stolpci pa je procesorski čas izvajanja testnega programa v sekundah.)

5.2 Razlaga rezultatov

Rezultati kažejo, da naša strojno pospešena knjižnica najbolje deluje na alternativah. Pri ostalih primerih je čas izvajanja enak času enega znaka (glej Dodatek A) in zato ni zanimiv.

S časom privarčujemo, če v regularnem izrazu potrebujemo alternative. V ostalih primerih glede na originalno knjižnico PCRE naša strojno pospešena knjižnica ni pokazala nobene presenetljivo večje pospešitve. Obremenitev procesorja je manjša za več kot polovico pri knjižnici, ki ima čakanje realizirano s spanjem. Ampak zasedeno čakanje izdaja, da je tudi realni čas čakanja na rezultat iskanja velik. Taka pospešitev je edino smiselna, če izvajamo regularne izraze z drugimi procesorsko intenzivnimi operacijami.

Alternative morajo biti že pri prvi črki, ki jo sprejema jezik regularnega izraza, drugače originalna knjižnica deluje enako kot brez alternativ. Ena alternativa na grafih (sliki 4.1 in 4.2) je samo niz brez metaznaka |. Grafi kažejo, da je tu časovna zahtevnost originalne knjižnice približno *O(n)* glede na število alternativ. Časovna zahtevnost naše strojno pospešene knjižnice pa je *O(1)*.

Zanimivo je, da je knjižnica, ki je za test hitrosti celotnega programa brez iskanja, počasnejša glede na porabljen procesorski čas. Na to najverjetneje vpliva število aktivnih niti (niso blokirane, so v pripravljenosti ali pa tečejo) testnega programa. Pri knjižnici brez iskanja so vse niti aktivne. Zaradi take aktivnosti se lahko podaljša čas zaradi preklopa konteksta. Za razliko od knjižnice brez iskanje je pri nitih optimizirane knjižnice s spanjem velik del časa aktivnih le malo niti.

Poglavje 6

Zaključek

Naša implementacija strojno pospešene knjižnice za regularne izraze je v primerjavi z originalno knjižnico PCRE hitrejša. Najbolj se to pozna na alternativah. Delna združljivost knjižnice omogoča poganjanje programov s pomočjo te knjižnice in to brez dodatnega prevajanja programov. Delovanje večine programov bi bilo zato nespremenjeno. Vendar problem predstavljajo neimplementirane funkcije. Te je potrebno definirati tako, da vračajo vsaj pričakovane vrednosti. Razlike med regularnimi izrazi lahko tudi predstavljajo probleme združljivosti, vendar ima večina programov regularne izraze zapisane v dodatnih nastavitvenih datotekah. S pomočjo tega lahko razlike morda omilimo.

Implementacija strojno pospešene knjižnice je smiselna glede na pridobljeni čas, če ne upoštevamo morebitne večje cene strojne opreme in časa razvoja implementacije.

Dobro bi bilo ugotoviti, kako učinkovito združiti iskanja regularnih izrazov na problemih, kjer iščemo po istih nizih z velikim številom regularnih izrazov. S tem bi močno pridobili na hitrosti. Mogoče je potrebno napisati algoritme, ki bodo učinkovito razrešili konflikte, če bi to knjižnico uporabili za zamenjevanje, in/ali dodati opcije, s katerim bi zamenjevanju nastavili različne razrešitve nastalih konfliktov.

Dobro bi bilo, narediti modul za jedro Linux, ki prevede fizični naslov iz navideznega. Izvajanje regularnega izraza bi tudi lahko prestavili v ta modul. Tako bi imeli probleme z navideznim naslavljanjem in dostopom enote rešene v jedru. Na ta način bi lahko tudi več programov hkrati dostopalo do enote za regularne izraze.

Za boljšo združljivost bi bilo dobro implementirati še ostale funkcionalnosti knjižnice. Knjižnico bi bilo prav tako potrebno opremiti z različnimi načini delovanja (različni prevajalni in izvajalni algoritmi), kar bi pripomoglo k večji

fleksibilnosti med uporabnostjo in hitrostjo.

Dodatek A

Rezultati meritev

Meritve so prikazane v tabelah na naslednjih straneh. Povprečje meritev smo izračunali tako, da smo od vsote vseh meritev odšteli maksimalno in minimalno meritev ter rezultat delili s številom, ki je za dva manjše od števila meritev. Torej smo ovrgli maksimalno in minimalno meritev. Iz preostanka meritev pa smo izračunali povprečje.

sleep				bw				stub				unaccelerated			
real time	cpu time1	cputime2	cputime	real time	cpu time1	cputime2	cputime	real time	cpu time1	cputime2	cputime	real time	cpu time1	cputime1	cputime
a															
8,74	2,54	3,08	5,62	5,51	25,49	8,65	34,14	1,58	2,46	5,88	8,34	2,10	11,29	6,69	17,98
8,55	2,33	3,03	5,36	5,50	26,05	8,34	34,39	1,57	2,54	6,05	8,59	2,14	11,25	6,91	18,16
9,64	2,37	2,86	5,23	5,77	24,32	10,96	35,28	1,57	2,55	6,02	8,57	2,05	11,20	6,69	17,89
8,54	2,15	2,54	4,69	5,51	26,19	8,43	34,62	1,58	2,58	5,97	8,55	2,09	11,34	6,68	18,02
7,46	2,06	2,82	4,88	5,55	25,68	8,60	34,28	1,58	2,57	6,12	8,69	2,04	11,28	6,65	17,93
8,15	2,33	3,00	5,33	5,55	26,70	8,48	35,18	1,58	2,53	6,03	8,56	2,07	11,26	6,61	17,87
9,92	2,29	2,66	4,95	5,52	26,45	8,70	35,15	1,59	2,59	6,06	8,65	2,11	11,04	6,87	17,91
8,72	2,29	2,87	5,15	5,53	25,97	8,57	34,72	1,58	2,55	6,03	8,58	2,08	11,26	6,72	17,95
aaaaaaaaa															
9,29	2,48	3,14	5,62	5,53	24,68	8,72	33,40	1,59	2,52	5,98	8,50	2,03	11,00	6,65	17,65
7,67	2,07	2,41	4,48	6,11	25,84	14,28	40,12	1,59	2,56	6,07	8,63	2,06	11,12	6,68	17,80
8,89	2,31	2,98	5,29	5,63	24,78	8,70	33,48	1,58	2,58	6,01	8,59	2,86	11,01	6,76	17,77
9,96	2,27	2,81	5,08	5,60	25,47	8,83	34,30	1,59	2,56	6,29	8,85	2,01	10,97	6,83	17,80
8,64	2,13	2,77	4,90	5,60	26,20	8,60	34,80	1,58	2,62	6,25	8,87	2,05	11,23	6,68	17,91
12,08	1,90	2,34	4,24	5,58	25,49	8,56	34,05	1,59	2,51	6,05	8,56	2,03	10,87	6,40	17,27
10,69	2,49	2,95	5,44	5,58	26,52	8,45	34,97	1,59	2,58	6,01	8,59	2,02	11,00	6,64	17,64
9,49	2,25	2,78	5,04	5,60	25,56	8,68	34,32	1,59	2,56	6,08	8,64	2,04	11,02	6,68	17,73
aaaaaaaaaaaaaaa															
12,17	2,19	2,47	4,66	5,62	25,96	8,54	34,50	1,60	2,57	6,07	8,64	2,08	11,13	6,78	17,91
8,79	2,19	2,58	4,77	5,63	24,89	8,97	33,86	1,70	2,48	7,22	9,70	2,02	11,08	6,52	17,60
9,13	2,31	2,94	5,25	5,61	26,37	8,47	34,84	1,60	2,62	6,16	8,78	2,06	11,10	6,69	17,79
9,47	2,00	2,41	4,41	5,54	27,07	8,31	35,38	1,60	2,57	6,08	8,65	2,01	10,93	6,47	17,40
11,15	2,78	3,45	6,23	5,62	15,55	8,74	24,29	1,60	2,53	6,26	8,79	2,03	10,87	6,66	17,53
6,99	2,21	3,29	5,50	5,60	26,41	8,45	34,86	1,60	2,55	5,89	8,44	2,11	10,88	6,74	17,62
8,33	2,46	3,05	5,51	5,64	25,60	8,71	34,31	1,60	2,57	6,49	9,06	2,03	10,91	6,76	17,67
9,37	2,27	2,87	5,14	5,62	25,85	8,58	34,47	1,60	2,56	6,21	8,78	2,04	10,98	6,67	17,64

sleep					bw				stub				unaccelerated			
real time	cpu time1	cputime2	cputime		real time	cpu time1	cputime2	cputime	real time	cpu time1	cputime2	cputime	real time	cpu time1	cputime1	cputime
a																
8,88	2,49	3,04	5,53		5,56	25,85	8,59	34,44	1,58	2,57	5,98	8,55	2,09	11,29	6,78	18,07
7,35	2,58	3,46	6,04		5,57	26,32	8,63	34,95	1,58	2,56	6,12	8,68	2,03	11,15	6,68	17,83
8,55	1,99	2,51	4,50		5,56	25,34	8,80	34,14	1,58	2,51	6,01	8,52	2,07	11,20	6,69	17,89
10,61	2,09	2,59	4,68		5,52	26,51	8,63	35,14	1,58	2,59	6,04	8,63	2,05	11,18	6,60	17,78
9,08	2,10	2,40	4,50		5,52	26,99	8,32	35,31	1,58	2,63	6,08	8,71	2,04	11,18	6,43	17,61
8,95	2,08	2,58	4,66		5,52	26,64	8,39	35,03	1,58	2,60	5,96	8,56	2,07	11,13	6,69	17,82
28,35	2,33	2,77	5,10		5,58	25,97	8,65	34,62	1,59	2,56	6,09	8,65	2,03	11,25	6,60	17,85
9,21	2,22	2,70	4,89		5,55	26,26	8,58	34,84	1,58	2,58	6,04	8,61	2,05	11,19	6,65	17,83
a|b																
9,82	2,58	3,18	5,76		5,60	26,00	8,54	34,54	1,58	2,56	6,02	8,58	19,31	285,57	0,37	285,94
8,16	2,22	2,69	4,91		5,59	25,58	8,63	34,21	1,58	2,56	6,05	8,61	19,28	285,70	0,34	286,04
8,43	2,60	3,30	5,90		5,54	26,27	8,74	35,01	1,58	2,53	6,04	8,57	19,27	285,66	0,33	285,99
11,02	2,47	2,90	5,37		5,52	25,78	8,40	34,18	1,59	2,60	6,06	8,66	19,26	285,69	0,36	286,05
8,16	2,58	3,38	5,96		5,55	26,44	8,24	34,68	1,59	2,54	6,13	8,67	19,31	285,57	0,40	285,97
8,69	2,29	2,73	5,02		5,57	25,44	8,68	34,12	1,58	2,53	6,16	8,69	19,36	285,59	0,38	285,97
7,87	2,24	2,88	5,12		5,57	25,77	8,53	34,30	1,58	2,60	6,03	8,63	19,31	285,55	0,39	285,94
8,65	2,43	3,00	5,43		5,56	25,88	8,56	34,38	1,58	2,56	6,06	8,63	19,30	285,62	0,37	285,98
a|b|c																
8,83	2,48	3,38	5,86		5,57	26,50	8,60	35,10	1,68	2,61	7,20	9,81	25,96	385,59	0,26	385,85
10,03	2,50	3,05	5,55		5,53	26,69	8,44	35,13	1,59	2,58	5,91	8,49	25,94	385,40	0,30	385,70
11,39	2,39	3,00	5,39		5,53	25,45	8,64	34,09	1,58	2,48	6,07	8,55	25,97	385,37	0,35	385,72
10,48	2,25	2,67	4,92		5,52	25,83	8,71	34,54	1,69	2,57	6,99	9,56	25,90	385,49	0,25	385,74
9,48	2,43	2,91	5,34		5,57	26,56	8,42	34,98	1,59	2,47	6,11	8,58	25,96	385,56	0,26	385,82
10,14	2,58	2,95	5,53		5,54	26,76	8,20	34,96	1,58	2,57	6,08	8,65	25,95	385,54	0,27	385,81
11,60	2,90	3,41	6,31		5,60	25,31	8,74	34,05	1,59	2,59	6,45	9,04	25,95	385,44	0,28	385,72
10,30	2,48	3,06	5,53		5,55	26,21	8,56	34,73	1,61	2,56	6,34	8,88	25,95	385,49	0,27	385,76
a|b|c|d																
9,38	2,21	2,60	4,81		5,46	23,75	8,98	32,73	1,58	2,57	6,16	8,73	32,51	484,34	0,27	484,61
9,30	2,44	2,86	5,30		5,58	25,94	8,54	34,48	1,58	2,54	6,09	8,63	32,55	484,37	0,25	484,62
7,42	2,16	2,93	5,09		5,55	26,88	8,38	35,26	1,59	2,65	6,12	8,77	32,51	484,21	0,34	484,55
8,28	1,97	2,61	4,58		5,52	26,38	8,38	34,76	1,58	2,64	6,03	8,67	32,51	484,28	0,27	484,55
10,13	2,58	3,22	5,80		5,52	26,79	8,45	35,24	1,57	2,50	5,94	8,44	32,59	484,24	0,30	484,54
7,42	2,03	2,70	4,73		5,48	27,07	8,11	35,18	1,58	2,54	6,22	8,76	32,49	484,27	0,33	484,60
11,72	2,69	3,19	5,88		5,54	28,11	8,46	36,57	1,59	2,55	6,47	9,02	32,67	484,29	0,26	484,55
8,90	2,28	2,86	5,15		5,52	26,61	8,44	34,98	1,58	2,57	6,12	8,71	32,53	484,28	0,29	484,57

Dodatek B

Testni program

test.c:

```
#include "pcre.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <limits.h>
#define NTH 1000
#define NIT 100
#define CI  0
#define DCHAR '\0'
sem_t threadwait;
sem_t main_exit;
pcre* regex1;
size_t size;
char * data;
int threadNum;
void * thread(void * something)
{
int i;
for(i=0;i<NIT;i++)
{
        pcre*reg=regex1;
        int res[2*3];
        char * str=(char*) malloc(size+100);
```

```
        memcpy(str,data,size);
        //str[0]='a';
        printf("exec\n");
        int ret=pcre_exec(reg,NULL,str,size,0,0,res,1*3);
        printf("n of substrings and string=%d\n",ret);
        if (ret>0)
        {
        str[res[1]]='\0';
        printf("substring is=%s\n",str+res[0]);
        }
        int a=0;
        int b=1;
        int c=1;
        for(a=0;a<CI;a++)//INT_MAX;a++)
        {
//              printf("%d\n",b);
          b=b+b+1+2*b+a+c*c;
        }
        printf("%d\n",b);
        free(str);
}
      sem_wait(&threadwait);
        threadNum--;
        printf("%d threads to finish.\n",threadNum);
        if (threadNum<=0)
        {
        sem_post(&threadwait);
        printf("exit\n");
        exit(0);
        }
      sem_post(&threadwait);
    return 0;

}
int main(int argc, char *argv[])
{
    if (argc!=3)
    {
       printf("Usage:\n\t%s <data length> <regex>\n",
```

```
        argv[0]);
        return;
    }
        size=(size_t) atoi(argv[1]);
        data=(char*) malloc(size);
        memset(data,'0',size);
        const char *error;
    int erroffset;
    int errorcode;
        pthread_t threads[NTH];
        int i;
        printf("compile\n");

        pcre* reg=pcre_compile2(argv[2],0&
    PCRE_CASELESS,&errorcode,&error,&erroffset,NULL);

        regex1=reg;
        sem_init(&threadwait,1,1);
        threadNum=NTH;
        sem_init(&main_exit,1,0);
        i=0;
        for(i=0;i<NTH;i++)
        {
                pthread_create(threads+i,NULL,thread,NULL);
        }
/*
        while (1)
        {
                sem_getvalue(&threadwait, &i);
                if (i<=0) break;
                sched_yield();
        }
*/
        sem_wait(&main_exit);


}
```

Dodatek C

Testna skripta z merjenjem časa in menjavo knjižnic

run.sh:

```
#!/bin/sh
mkdir -p /lib64/pcreorig/
mv /lib64/libpcre* /lib64/pcreorig/.

cp /libpcre.so /lib64/libpcre.so
export dir=$(pwd)
cd /lib64/
ln -s libpcre.so libpcre.so.0
cd $dir
echo "test  with acceleration sleep thread:"
time /test "$1" "$2" > /dev/null
rm /lib64/libpcre*

cp /libpcrebw.so /lib64/libpcre.so
export dir=$(pwd)
cd /lib64
ln -s libpcre.so libpcre.so.0
cd $dir
echo "test  with acceleration bussy waiting:"
time /test "$1" "$2" > /dev/null
rm /lib64/libpcre*

cp /libpcrefake.so /lib64/libpcre.so
```

```
export dir=$(pwd)
cd /lib64
ln -s libpcre.so libpcre.so.0
cd $dir
echo "test library without anything:"
time /test "$1" "$2" > /dev/null
rm /lib64/libpcre*


echo "test without acceleration:"
mv /lib64/pcreorig/* /lib64/.
time /test "$1" "$2" > /dev/null
```

Dodatek D

Testna skripta za izvajanje zaporedja meritev

testing.sh:

```
#!/bin/sh
echo "a"
./run.sh 4096 "a" 2>&1
echo "a|b"
./run.sh 4096 "a|b" 2>&1
echo "a|b|c"
./run.sh 4096 "a|b|c" 2>&1
echo "a|b|c|d"
./run.sh 4096 "a|b|c|d" 2>&1
echo "a"
./run.sh 4096 "a" 2>&1
echo "aaaaaaaaa"
./run.sh 4096 "aaaaaaaaa" 2>&1
echo "aaaaaaaaaaaaaaaaa"
./run.sh 4096 "aaaaaaaaaaaaaaaaa" 2>&1
echo "a*"
./run.sh 4096 "a*" 2>&1
echo "((((a)a)a)a)"
./run.sh 4096 "((((a)a)a)a)" 2>&1
echo \
"((((((((((((((((((a)a)a)a)a)a)a)a)a)a)a)a)a)a)a)a)a)"
./run.sh 4096 \
"((((((((((((((((((a)a)a)a)a)a)a)a)a)a)a)a)a)a)a)a)a)" \
```

```
 2>&1
echo "(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)"
./run.sh 4096 \
"(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)(a)" 2>&1
```

Literatura

[1] PCRE(3) Man pages. Dostopno na:
http://www.pcre.org/pcre.txt

[2] Perl 5 version 10.1 documentation perlre. Dostopno na:
http://perldoc.perl.org/perlre.html

[3] Summary of Cavium Networks-Specific cnMIPS Core Instructions. Dostopno na:
http://www.cnusers.org/index.php?Itemid=32&func=startdown&id=48

[4] OCTEON CN3800/CN3600 4 to 16-Core MIPS64 Based SoCs ™ Product Brief. Dostopno na:
http://www.caviumnetworks.com/pdfFiles/
OcteonCN38XX_CN36XX_PB-Jan29-06-web-v1.pdf

[5] John E. Hopcroft, Jeffrey D. Ullman, prevod: Boštjan Vilfan *Uvod v teorijo avtomatov, jezikov in izračunov*, 3., pregledana in dopolnjena izd., Ljubljana: Fakulteta za elektrotehniko in računalništvo, 1990.

[6] izvorna koda Simple Executive Library for Octeon, Cavium.

[7] POSIX Threads Programming, Blaise Barney, Lawrence Livermore National Laboratory. Dostopno na:
https://computing.llnl.gov/tutorials/pthreads/

[8] The Single UNIX ® Specification, Version 2: usleep, Copyright ©1997 The Open Group. Dostopno na:
http://www.opengroup.org/onlinepubs/007908799/xsh/usleep.html