UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Milan Pevec

Zmogljivostna analiza spletne PHP aplikacije z uporabo različnih tehnik predpomnjenja

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

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: 01622/2009
Datum: 15.12.2009

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

Kandidat: **MILAN PEVEC**

Naslov: **ZMOGLJIVOSTNA ANALIZA SPLETNE PHP APLIKACIJE Z UPORABO RAZLIČNIH TEHNIK PREDPOMJENJA**

PERFORMANCE ANALYSIS OF THE PHP SYSTEM BY USING DIFFERENT TECHNIQUES OF CACHING

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Čas odgovora pri spletnih aplikacijah je zelo pomemben, saj se obiskovalci kaj hitro naveličajo čakati in poiščejo drugo spletno stran. Za izboljšanje zmogljivosti spletnega strežnika obstaja več načinov in eden izmed njih je uporaba predpomnilnika.

V diplomski nalogi naj kandidat izvede zmogljivostno analizo spletne aplikacije PHP, s poudarkom na različnih tehnikah predpomnjenja. Pri izdelavi analize naj se v prvem koraku opredeli sistem ter določijo metrike in parametri. Natančno naj se opredeli postopke merjenja in izdela meritve realne spletne aplikacije. Pridobljeni rezultati naj bodo ustrezno analizirani in komentirani.

Mentor:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI •

Dekan:

prof. dr. Franc Solina

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Milan Pevec

Zmogljivostna analiza spletne PHP aplikacije z uporabo različnih tehnik predpomnjenja

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Nikolaj Zimic

Ljubljana, 2009

...ORIGINAL/IZVIRNIK IZDANE TEME...

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a MILAN PEVEC,

z vpisno številko 63980112

sem avtor/-ica diplomskega dela z naslovom:

Zmogljivostna analiza spletne aplikacije PHP z uporabo različnih tehnik predpomnjenja

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)
  prof. dr. Nikolaj Zimic
  in somentorstvom (naziv, ime in priimek)
  -
- so elektronska oblika diplomskega dela, naslov (slov., ang.), povzetek (slov., ang.) ter ključne besede (slov., ang.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice: _____

ZAHVALA

Zahvaljujem se mentorju prof. dr. Nikolaju Zimicu za strokovne nasvete in priporočila pri izdelavi diplomske naloge in direktorju Gregorju Mikliču iz podjetja Odkup terjatev d.o.o. za dovoljenje uporabe spletne aplikacije Payforcement za namene meritev diplomske naloge.

Posebna zahvala gre staršem in partnerki Urški Tkalec za vso podporo in spodbudo ter vsem, ki so mi stali ob strani v celotnem času študija.

KAZALO

SEZNAM UPORABLJENIH KRATIC

Apache (krajše za *Apache HTTP Server*)
Gre za odprtokodni spletni strežnik organizacije Apache.

CDN (*Content Delivery Network*)
Skupina strežnikov, ki so geografsko razpršeni na več lokacijah z namenom, da zmanjšajo razdalje med uporabniki in strežniki.

CSRF (*Cross-Site Request Forgery*)
Varnostni mehanizem, ki preprečuje ponarejanje zahtevkov po določenih straneh, in sicer tako, da strežnik generira žetone, ki jih pošlje odjemalcu, ta pa potem pošlje zahtevek s temi žetoni, ki potem strežniku omogočijo identifikacijo pravih (neponarejenih) zahtevkov.

Datoteke DBM
Gre za datoteke, ki omogočajo izjemno hiter dostop in sočasno branje/pisanje. Imenujejo se tudi *poor man's database*.

Front-end*, *back-end
Gre za abstrakcijo, namenjeno ločevanju delov sistema (npr. nek del spada na *back-end*).

HTML (*Hyper Text Markup Language*)
Označevalni jezik za spletne strani.

HTTP (*Hypertext Transfer Protocol*)
Protokol komunikacije na aplikacijskem nivoju sklada TCP/IP.

MPM (*Multi-Processing Modules*)
Moduli strežnika Apache, ki določajo, kako se obravnavajo prispele zahteve.

Vzorec MVC (*MVC pattern*)
Vzorec, ki definira arhitekturo spletne aplikacije, kjer se aplikacija razdeli na tri dele:

poslovno logiko (*model*), nadzorno logiko (*controller*) in predstavitveno logiko (*view*).

PHP (*hypertext preprocessor*)
Razširjen odprtokodni programski jezik.

Skripta PHP
Programska koda PHP, ki se izvaja na strežniku.

Proxy
Krajše za *proxy server* – strežnik, ki je postavljen med zahtevki odjemalcev in ostalimi strežniki.

(S)API (*Server API – application programming interface*)
Vmesna plast med spletnim strežnikom na eni strani in (v našem primeru) sistemom PHP na drugi strani. Na splošno API omogoča komunikacijo med dvema entitetama sistema.

SHM (*shared memory*)
Deljen pomnilnik, ponavadi blok v glavnem pomnilniku, katerega lastnost je ta, da si ga procesi med sabo delijo.

TCP (*transmission control protocol*)
Protokol komunikacije na transportnem nivoju sklada TCP/IP.

XML (*eXtensible Markup Language*)
Razširljivi označevalni jezik za izmenjavo strukturiranih dokumentov.

(XSL)T
Transformacija toka podatkov XML s pomočjo predloge XSL. Rezultat je ponavadi (X)HTML, lahko pa je tudi XML itd.

POVZETEK SLO

Namen pričujoče diplomske naloge je izvedba zmogljivostne analize spletne PHP aplikacije z uporabo različnih tehnik predpomnjenja. V prvem delu je predstavljen sistem PHP in nekatere ključne odločitve pri njegovi postavitvi na spletnem strežniku, ki vplivajo na zmogljivost. V drugem delu so prikazane razne tehnike predpomnjenja pri sistemih PHP. Opisani so tudi problemi, s katerimi se danes srečujemo pri predpomnjenju. Osredotočili smo se na predpomnjenje na strežniku. Tretji del vsebuje zmogljivostno analizo realne spletne aplikacije PHP z realnimi bremeni. Cilj je bil zmanjšati odzivni čas in povečati propustnost definiranega sistema. Opravljene so bile meritve glede na različne tehnike predpomnjenja, in sicer smo izvedli zmogljivostne teste in *ramp* teste. Na koncu smo podrobno predstavili rezultate.

Ključne besede: sistem PHP, spletni strežnik, predpomnjenje spletnih strani, odzivni čas, propustnost, zmogljivostni test, *ramp* test.

ABSTRACT

The purpose of the thesis is to carry out performance analysis by using different techniques of caching. The first part describes a PHP system and some key decisions related to its deployment on a web server that affect performance. The second part describes different techniques of caching in PHP systems. Additionally, problems that commonly occur in caching are described. The main attention was focused on server caching. The third part of the thesis contains performance analysis carried out in an actual PHP web application with real load. The goal of the analysis was to reduce the response time and to increase the throughput of the defined system.

Measurements were taken by considering different techniques of caching; performance tests as well as ramp tests were carried out. Finally, the results were presented in detail.

KEY WORDS: PHP system, web server, caching of web pages, response time, throughput, performance test, ramp test.

1. UVOD

Predpomnjenje je še vedno vroča tema v svetu spletnih aplikacij[1]. Ko raste popularnost neke strani, se povečuje tudi število zahtevkov, ki jim mora strežnik ustreči v določenem času. Ta čas se neprestano krajša, uporabniki pa so čedalje manj potrpežljivi. Z ustrezno zmogljivostno analizo lahko pravočasno odkrijemo morebitne težave, ki jih lahko s predpomnjenjem bistveno zmanjšamo ali celo odpravimo.

Pojem predpomnjenje se navezuje na številne tehnike, ki pa vse delujejo na podoben način: rezultat neke zahtevne operacije se shrani (na hitrejši medij) in je na voljo za kasnejšo uporabo. Pri tem je treba paziti, kajti večina strani je danes dinamičnih, torej se s časom spreminjajo. Zato je treba izbrati ustrezno metodo in strategijo predpomnjenja, da ne pride do nezaželenih rezultatov.

Zlasti pri spletnih aplikacijah lahko implementiramo predpomnjenje na različnih ravneh. Ravno tako tudi na obeh straneh komunikacije odjemalec-strežnik. Tako lahko z ustrezno nastavitvijo glav HTTP in uporabo ustreznih metaznakov HTML celo dosežemo, da določen zahtevek sploh ne pride do strežnika. V pričujoči diplomski nalogi se bomo osredotočili predvsem na tehnike predpomnjenja na strežniku.

Predpomnjenje je smiselno uporabiti le pri ustrezno zahtevnih operacijah. Zato si bomo ogledali, kako poteka analiza programske kode (*profiling*) z enim od odprtokodnih orodij. Cilj je identificirati dele kode, ki občutno zmanjšajo zmogljivost.

Kljub vsemu predpomnjenje ne prinese vedno želenih rezultatov. Namreč zgodi se lahko to, da se izvajalni čas neke skripte[2] zmanjša, vendar se tudi skalabilnost zmanjša kot posledica dejstva, da sedaj hitrejša skripta zasede več resursov. Zato si bomo pogledali, kakšni so rezultati predpomnjenja na resnični aplikaciji, ki je v produkcijski uporabi več let. Za meritev bomo uporabili najzahtevnejšo in najbolj obiskano stran uporabljene aplikacije (z največ zahtevki).

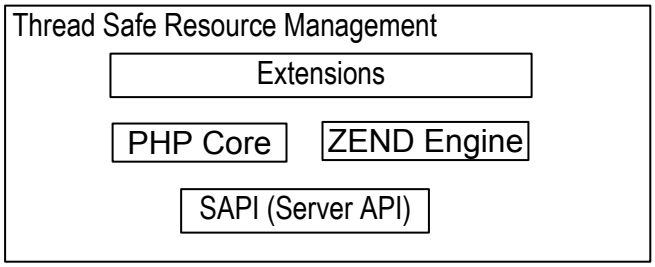
---

1 Pri spletni aplikaciji gre za sistem odjemalec – strežnik. Komunikacija poteka s protokolom HTTP(s), odjemalec (npr. brskljalnik) pošlje zahtevek na strežnik, ta ga obdela in pošlje odgovor (npr. spletno stran) nazaj k odjemalcu.

2 V svetu PHP se govori o skriptah kot programski kodi, ki se izvaja na strežniku.

2. ZGRADBA SISTEMA PHP

PHP (*Hypertext preprocessor*) je razširjen odprtokodni programski jezik, ki se uporablja predvsem za izdelavo spletnih aplikacij. Sam pojem se nanaša na celoten sistem, sestavljen iz petih delov, kot prikazuje spodnja slika:



Slika 2.1: Zgradba sistema PHP. Vir [9].

- ***SAPI***.

Koordinira življenjski cikel PHP-ja in je običajno vgrajen (*embedded*) v spletni strežnik, npr. ***Apache HTTP Server*** [21] (v nadaljevanju „Apache“). Apache sprejema zahtevke po straneh PHP (datotekah) in jih posreduje sistemu PHP. Prav tako skrbi za komunikacijo med njima.

- ***PHP Core***.

Omogoča osnovne funkcije, kot so delo z datotekami, napakami, tokovi idr., vendar te funkcije niso na voljo programerjem, ampak razširitvam (*extensions*). Skrbi tudi za njihovo inicializacijo in omogoča njihovo uporabo.

- ***Zend Engine***.

Razčlenjuje (*parse*) in prevaja skripte PHP v strojno kodo. Prav tako skrbi za njeno izvajanje. Zagotavlja tudi API za njegove razširitve, kar omogoča uporabo raznih merilnikov (*profiler*), razhroščevalnikov ipd.

- ***Extensions***.

Tu se nahajajo razne razširitve (delo s sejo, nizi, bazo itd.), katerih funkcije so na voljo programerjem.

- ***Thread Safe Resource Management***.

Zagotavlja, da lahko posamezna (ena) instanca sistema PHP pravilno obdeluje več zahtev hkrati[3].

3 Velja, da vse razširitve niso t.i. *thread-safe*, v teh primerih sistem PHP ne zagotavlja pravilnosti delovanja.

2.1 Apache in MPM (*Multi-Processing-Modules*)

Apache je zelo prilagodljiv spletni strežnik. Tako ima tudi najbolj osnovne funkcije razdeljene na module, kot je recimo komunikacija odjemalec – strežnik. Njegovo osnovno delovanje je naslednje: Apache ima ponavadi proces (odvisno od načina delovanja, lahko je tudi nit), ki čaka na določenih vratih (ponavadi 80) za morebitno novo povezavo TCP. Odjemalec pošlje zahtevo na strežnik in povezava se vzpostavi. Apache po isti povezavi pošlje odgovor. Protokol, ki definira zahteve in odgovore, je HTTP. Glede na različico protokola HTTP (in nastavitve na spletnem strežniku) lahko povezava TCP ostane vzpostavljena še nekaj časa, ki se določi vnaprej. Povezava TCP pa se lahko zaključi tudi takoj zatem, ko spletni odjemalec dobi odgovor na zahtevo.

Kako natančno se obravnajo prispele zahteve, določajo moduli MPM. Strežnik s potrebo po večji skalabilnosti lahko uporabi MPM z imenom *worker*, ki za vsako povezavo TCP uporabi nit namesto procesa. Strežnik s potrebo po veliki stabilnosti in kompatibilnosti za nazaj pa lahko uporabi MPM z imenom *prefork*. Tu se niti ne uporabljajo, za **vsako povezavo TCP se uporabi proces**.

Na voljo so tudi ostali moduli MPM, ki pa so zaenkrat v poskusni fazi [1]. PHP najbolje [2] deluje v načinu *prefork*, saj veliko njegovih razširitev ni *thread-safe*, kar pomeni, da ne deluje pravilno v načinu *worker*. Zato si bomo podrobno pogledali modul MPM *prefork*.

2.1.1 Apache MPM *prefork* [4]

Nadzorni proces (*control process*), odvisno od nastavitev, kreira nove procese (*child process*), ki čakajo na določenih vratih na zahteve po povezavi TCP. Prav tako lahko nadzorni proces te procese ubije. Kako se vse skupaj odvija, je odvisno od nastavitvenih parametrov (v nadaljevanju „direktiv“). Pogledali si bomo direktive, pomembne za kasnejšo zmogljivostno analizo.

Direktiva *MaxClients* pove, kolikšno je trenutno maksimalno število procesov. Če se pojavi zahteva po večjem številu procesov (v primeru, da hočemo imeti odprtih veliko povezav TCP), bodo zahteve čakale v vrsti. Direktiva *StartServers* pove, koliko procesov bo kreiranih

ob zagonu Apache. Direktivi *MaxSpareServers* in *MinSpareServers* pa povesta minimalno in maksimalno število procesov, ki so v določenem trenutku prosti. Velja pravilo, da mora biti *MaxSpareServers* vsaj za ena večje od *MinSpareServers*. Omenjeni nadzorni proces je odgovoren za to, da je število procesov glede na direktive ustrezno.

Direktiva *MaxRequestPerChild* pove, koliko povezav TCP lahko proces obdela, preden ga nadzorni proces reciklira (ubije in kreira novega). Vrednost 0 pove, da proces ne bo recikliran nikoli. Posledica je lahko, da v primeru velikega bremena število procesov v sistemu naraste. Ko pa breme pade, pa ti procesi ostanejo živi, kar ima svoje slabosti in prednosti [3].

2.1.2 Strežnik z veliko obremenitvijo ter direktivi *KeepAlive* in *KeepAliveTimeout*

Aktivna direktiva *KeepAlive* pomeni, da se povezava TCP ohrani vzpostavljena še za naslednje zahtevke. Ponavadi gre za zahtevke po objektih, kot so slike, ikone, datoteke css ipd. – zahtevki po statičnih vsebinah. Tipično se danes po eni povezavi TCP prenese med 5 in 20 zahtev [14].

Če analiziramo opazovano stran z orodjem Yslow (*http://developer.yahoo.com/yslow/),* nam razdelek *Components* pove, da smo poleg strani PHP zahtevali še naslednje objekte:

- css/style.css
- css/normal.css
- dsg/page_logos/logo_SI.gif
- dsg/bck_1_950.gif
- dsg/bck_template_150.gif
- dsg/menu.gif
- dsg/bck_0.gif
- dsg/button_1.gif

Število zahtevkov je torej 9.

KeepAliveTimeout določa, kako dolgo Apache na neki povezavi TCP čaka, preden prispe naslednja zahteva. Če se ta čas izteče, se povezava TCP zapre. Ampak ker želimo imeti

povezavo TCP odprto čim manj časa (ena povezava TCP pomeni en proces Apache, teh pa želimo imeti čim manj), nastane problem, kako ustrezno nastaviti *KeepAliveTimeout*. Večji kot je, več bomo imeli aktivnih procesov Apache. Tega si ne želimo. Če pa ga nastavimo na premajhno vrednost, pa tvegamo, da bomo izgubljali čas z ubijanjem in kreiranjem procesov ter vzpostavljanjem povezav TCP.

Ena možnost je, da je *KeepAlive* neaktiven. V tem primeru je treba za vsak objekt na spletni strani vzpostaviti novo povezavo TCP in tako nastane nov proces Apache. V zgornjem primeru bi to pomenilo 9 procesov – tega pa si tudi ne želimo. Kaj pa če uporabimo Apache samo za strežbo dinamične vsebine (za t.i. *back-end*, v tem primeru za skripte PHP), za statične vsebine (za t. i. *front-end*) pa ga ne uporabimo? V tem primeru lahko izklopimo *KeepAlive* in s tem zmanjšamo število aktivnih procesov Apache.

2.2 Predpomnjenje t. i. objektov *front-end*

Zgoraj smo ugotovili, da je za strežbo takih objektov primerno imeti postavljen poseben spletni strežnik. Ali je pomembno, kakšen strežnik izberemo? V podjetju Yahoo[12] so naredili analizo 10 najbolj obiskanih spletnih strani in ugotovili naslednje, citiram ([12], Steve Souders iz podjetja Yahoo):

"Optimize front-end performance first, that's where 80% or more of the end-user response time is spent."

Pomeni, da je delež strežbe t. i. objektov *front-end* v času odziva najmanj 80 %. Odgovor na naše vprašanje je zato pozitiven. Želimo si strežnik, ki streže statično vsebino hitreje kot Apache in zasede bistveno manj resursov (je "lažji"). Eden od takih je spletni strežnik Varnish. Je odprtokodni produkt, katerega glavni razvijalec je Poul-Henning Kamp, ki je eden od glavnih razvijalcev[16] operacijskega sistema freeBSD.

Dodatno lahko vpeljemo še določene izboljšave. Skupna lastnost vseh je, da z ustrezno nastavitvijo na spletnem strežniku določimo čas zapadlosti statične vsebine. Ta čas se uporabi pri predpomnjenju. Pove nam namreč, do kdaj je nek objekt še veljaven. Ker se statična

vsebina spreminja samo ob posodobitvah (ko se nova vsebina prenese na spletni strežnik), lahko za čas zapadlosti nastavimo neskončnost. Ob posodobitvah se tako prenese npr. drugačna slika, vendar bo ta imela drugo ime (ponavadi spreminjamo imena takole: style1.1.css, style1.2.css itd.).

Predpomnenje je možno v brskljalniku ali pa na t. i. proxyju. Zahtevek bo tako prispel do strežnika šele ob zgrešitvi. Prav tako pa lahko uporabimo t. i. CDN (*content delivery network*) – skupino strežnikov, ki so geografsko razpršeni na več lokacijah z namenom, da zmanjšajo razdalje med uporabniki in strežniki.

Če pa kljub vsemu zahtevek prispe do spletnega strežnika, lahko tudi strežniku določimo, naj za najpogostejše statične vsebine uporabi predpomnjenje. Prej omenjeni spletni strežnik Vanquish (v nadaljevanju: Vanquish) ima tu prednost, saj je njegov razvoj temeljil ravno na tem.

Kakšna je razlika med predpomnjenjem pri Vanquishu in tistim pri običajnih spletnih strežnikih? Pri običajnih spletnih strežnikih velja naslednje: Za predpomnjenje uporabljajo na višjem nivoju pomnilnik in na nižjem nivoju disk. Pojavi se zahtevek za nek objekt. Zahteva se postreže, objekt se shrani v pomnilnik. Tam se zadrži nekaj časa in nato ga operacijski sistem premakne v navidezni pomnilnik na disk. Tega se običajni spletni strežnik ne zaveda, saj ima še vedno informacijo, da je ta objekt v pomnilniku. Tako pride do situacije, da čez čas, glede na nastavitve, sam ugotovi, da ta isti objekt zaradi premajhne frekvence zahtevkov ne rabi biti v pomnilniku in ga zato prenese na disk. Posledično se objekt najprej prenese z navideznega pomnilnika nazaj v pomnilnik in nato nazaj na disk, kar je seveda potratno. Prednost Vanquisha je ravno v tem, da se zaveda operacij operacijskega sistema nad predpomnjenimi objekti in do opisane situacije ne pride. Poleg tega Vanquish uporablja še nekatere druge izvirne prijeme. Več o tem je dostopno na: http://varnish.projects.linpro.no/wiki/ArchitectNotes.

2.3 Življenjski cikel skript PHP v primeru Apache MPM *prefork*

Poglejmo si natančneje, kako deluje sistem PHP[9]. V prvi fazi (preden sploh pride do

zahteve), PHP kliče funkcijo MINIT (*module initialization*) na vseh razširitvah. Posledično razširitve deklarirajo konstante, registrirajo resurse ipd. To se zgodi enkrat samkrat za vse morebitne prihodnje zahtevke. Prav tako se kliče inicializacija Zend Enginea (glej Sliko 2.1) in inicializacija izhodnega vmesnika (*output buffer*), namreč vanj se shranjuje rezultat izvajanja in njegova vsebina se prenese k odjemalcu.

Ko zahtevek prispe, PHP vzpostavi okolje za izvajanje (glede na nastavitve) in se ponovno sprehodi skozi vse razširitve. Tokrat kliče funkcijo RINIT (*request initialization*). Posledično razširitve resetirajo svoje globalne vrednosti, nastavijo parametrom začetne vrednosti ipd. Aktivira se tudi Zend Engine.

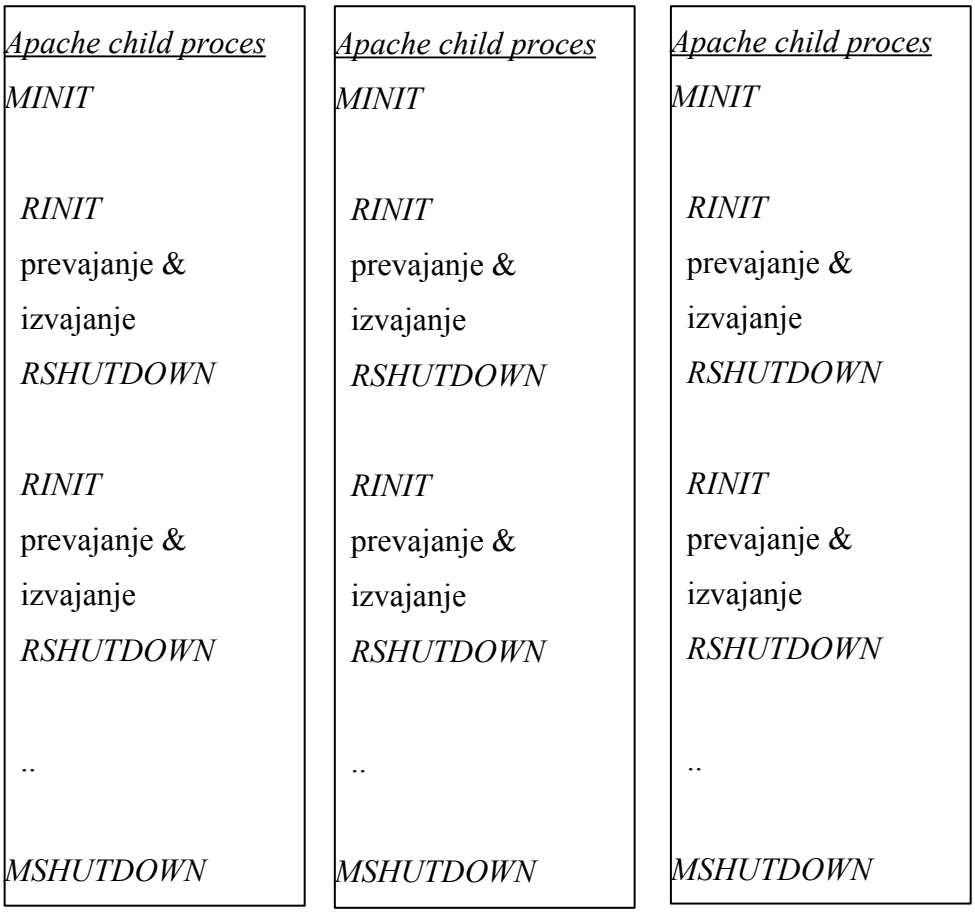
Najpomembnejše, kar se zgodi, je naslednje. Skripta PHP je pripravljena za dve fazi:

- razčlenjevanje in prevajanje
- izvajanje

Po koncu izvajanja skripte PHP ta pokliče funkcijo RSHUTDOWN (*request shutdown*) na vseh razširitvah. Posledično se sprostijo resursi, ki so jih razširitve zasedle. Tudi Zend Engine se deaktivira.

V zadnji fazi (npr. ob zaustavitvi spletnega strežnika) se sistem PHP ponovno sprehodi skozi razširitve in kliče funkcijo MSHUTDOWN (*module shutdown*). Tako se sprostijo še zadnji resursi, npr.: persistenčne povezave do baze.

Spodnja slika prikazuje zgornje faze v primeru, da je PHP vgrajen v Apache. Uporablja se MPM *prefork*, pri čemer imamo tri hkratne TCP povezave, na vsaki povezavi pa je lahko več zahtevkov:



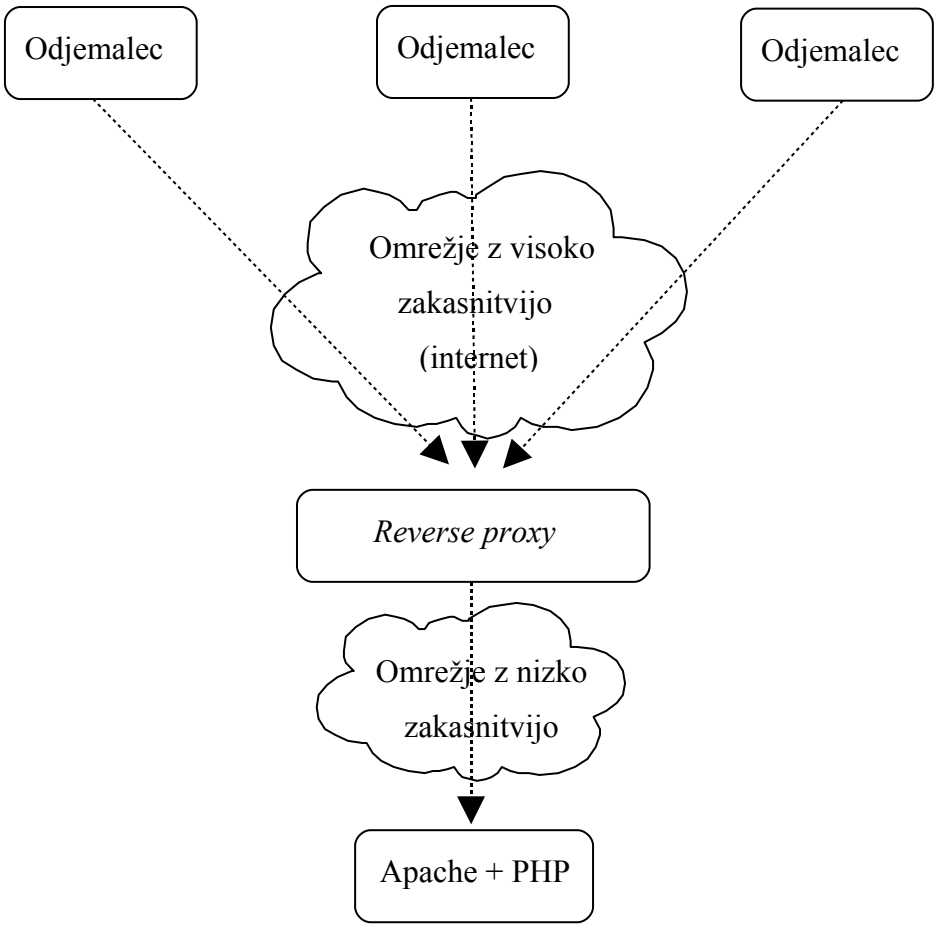
Slika 2.2: Prikaz sodelovanja sistema PHP in strežnika Apache. Vir [9]

Natančnejši opis delovanja najdemo v [10].

2.4 Zakasnitev omrežja

Obstaja omejitev glede tega, koliko podatkov se lahko prenese naenkrat, **v enem kosu**, po povezavi TCP. Če je podatek večji od omejitve, se prenese v več kosih. Velja pa, da je prenos enega podatka hitrejši kot prenos tega istega podatka v več kosih. Če je omrežje počasno (internet je lahko počasno omrežje), potem prenašanje podatkov po kosih povzroči, da spletni strežnik (pre)več časa čaka, da se ti prenosi zaključijo. Posledično živi v sistemu več procesov (kot bi sicer) in zasedenih je več resursov (kot bi bilo sicer). Rešitev je uporaba t. i. *reverse proxyja* (znanega tudi kot *HTTP accelerator*). Ta je postavljen tik pred spletnim strežnikom in na eni strani sprejema zahtevke – na strežnik pošlje celotno zahtevo, na drugi strani pa sprejema odgovore – na odjemalca pošlje celoten odgovor. Spodnja slika prikazuje ta

koncept.



Slika 2.3: Prikaz koncepta delovanja reverse proxyja

Za to rešitev sta pomembni dve lastnosti *reverse proxyja*, namreč da zasede bistveno manj resursov kot spletni strežnik (z vgrajenim PHP) in da je na hitrem omrežju (z nizko zakasnitvijo). Zato lahko spletni strežnik hitro odda odgovor na *reverse proxy* in začne s strežbo naslednjega zahtevka. *Reverse proxy* pa sam poskrbi, da ta odgovor pride do odjemalca.

Zgornjo vlogo *reverse proxyja* lahko prevzamejo tudi sistemski izravnalniki TCP (*TCP buffers*), ki morajo biti dovolj veliki, da vanj pride največji celoten odgovor strežnika. V tem primeru zgoraj omenjenega čakanja strežnika ni [20].

2.5 “Process lingering”

Zakasnitev omrežja lahko vpliva tudi na počasno zapiranje povezav TCP. Namreč Apache

pošlje zahtevo po zaprtju povezave TCP in mora predolgo čakati na potrditev s strani odjemalca. Ta pojav se imenuje *process lingering*. Posledica je enaka, kot je opisano zgoraj. Rešitev je uporaba procesa *lingerd* (ali posebnega modula na spletnem strežniku). Apache mu v celoti prepusti zapiranje povezav TCP in se raje posveti svojemu pomembnejšemu delu. Tudi *lingerd* zasede bistveno manj resursov kot proces Apache, poleg tega lahko vzporedno zapira povezave.

3. METODE PREDPOMNJENJA PRI SPLETNIH APLIKACIJAH PHP

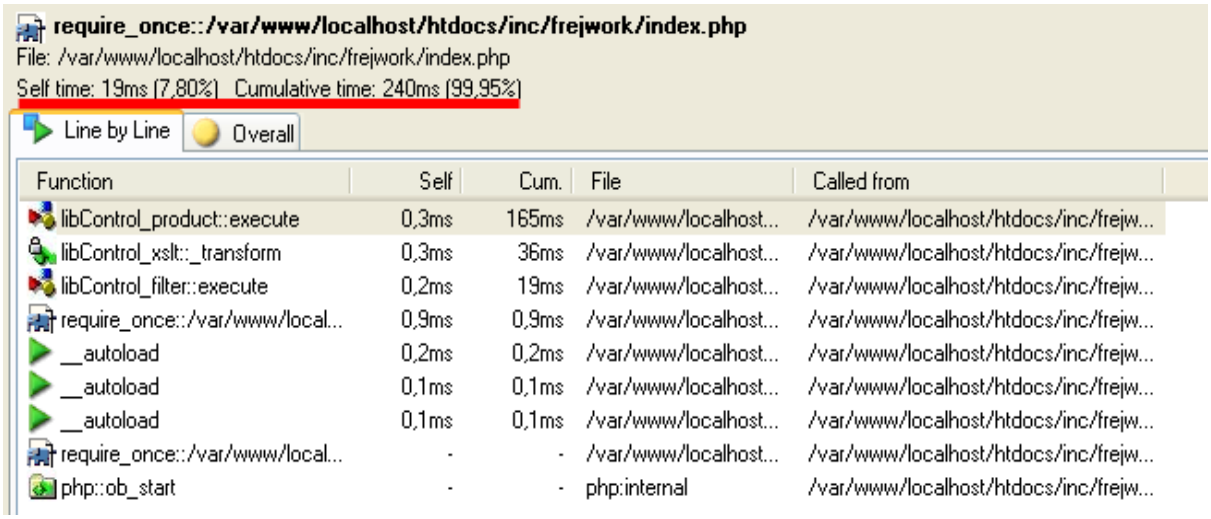
3.1 Analiza programske kode (*profiling*)

Za katere komponente spletne aplikacije je smiselno uporabiti predpomenjenje? Leta 2003[17] so se spletne strani www.php.net občutno upočasnile. Pregled dnevniških datotek spletnih strežnikov je pokazal, da lahko krivdo pripišemo iskalnikom. Brez analize programske kode, s katero bi natančno ugotovili ozka grla, so intuitivno spreminjali kodo, kjer so po nepotrebnem porabili več dni, da so odpravili težave. Pravilna analiza programske kode tako prikaže ozka grla spletne aplikacije, ki jih je potem mogoče odpraviti tudi z uporabo ustreznih metod predpomnjenja.

Na splošno lahko identifikacija komponent poteka s treh zornih kotov:

- **Princip lokalnosti.** Upoštevamo odgovor na vprašanje, katere komponente se najpogosteje uporabljajo, saj je tiste smiselno predpomniti.
- **Čas, porabljen za izvajanje komponente.** Smiselno je predpomniti komponente, ki se izvajajo dovolj časa, npr. izračuni, ki so odvisni od počasnih resursov. Zelo pomembno je dejstvo, da je treba take komponente opazovati – ne samo z vidika enega uporabnika, ampak z vidika več hkratnih uporabnikov. Tako je lahko izvrševanje določene komponente z uporabo predpomnjenja pri enem uporabniku hitrejše kot brez uporabe predpomnjenja. Vendar če se število uporabnikov poveča, lahko predpomnjenje ovira izvrševanje.
- **Statičnost komponente.** Prednost imajo komponente, ki v celoti temeljijo na statični vsebini, npr. glava strani, v kateri so zapisani podatki o podjetju, in komponente, ki so dovolj dolgo statične. Te se spremenijo zelo redko, ponavadi ob vnaprej določenem času (ob izdaji nove različice).

Za analizo programske kode (PHP) obstaja več produktov. Za namen diplomske naloge smo uporabili produkt Xdebug (http://www.xdebug.org/docs/profiler), ki ob zahtevku generira posebno datoteko s profilom. Ta se potem pregleda z namenskim programom, npr. KCacheGrind (Windows, alternativa je WinCacheGrind), kjer se izpiše t. i. celotna sled klicev (*call trace)* – vsi klici vseh funkcij in njihov čas izvrševanja. Primer je prikazan na spodnji sliki. Sama uporaba produkta je preprosta, saj ga je mogoče dinamično vklapljati in izklapljati.



Slika 3.1: Rezultat analize programske kode s produktom Xdebug

Na sliki vidimo, da je bil poslan zahtevek (in posledično ustvarjen profil) za /var/www/localhost/htdocs/inc/frejwork/index.php. Samo izvrševanje omenjene skripte PHP je trajalo 19 ms, vsi klici v tej skripti pa znašajo 240 ms (oboje je prikazano nad rdečo črto). Prav tako se na sliki vidi, katere funkcije so bile klicane in koliko časa je trajalo njihovo izvrševanje (npr.: klic libControl_xslt::_transform je trajal 16 ms). Treba je omeniti, da se dá pogrezati v globino posameznih klicev, kar omogoča natančno identifikacijo ozkih grl.

3.2 Predpomnjenje na *reverse proxyju*

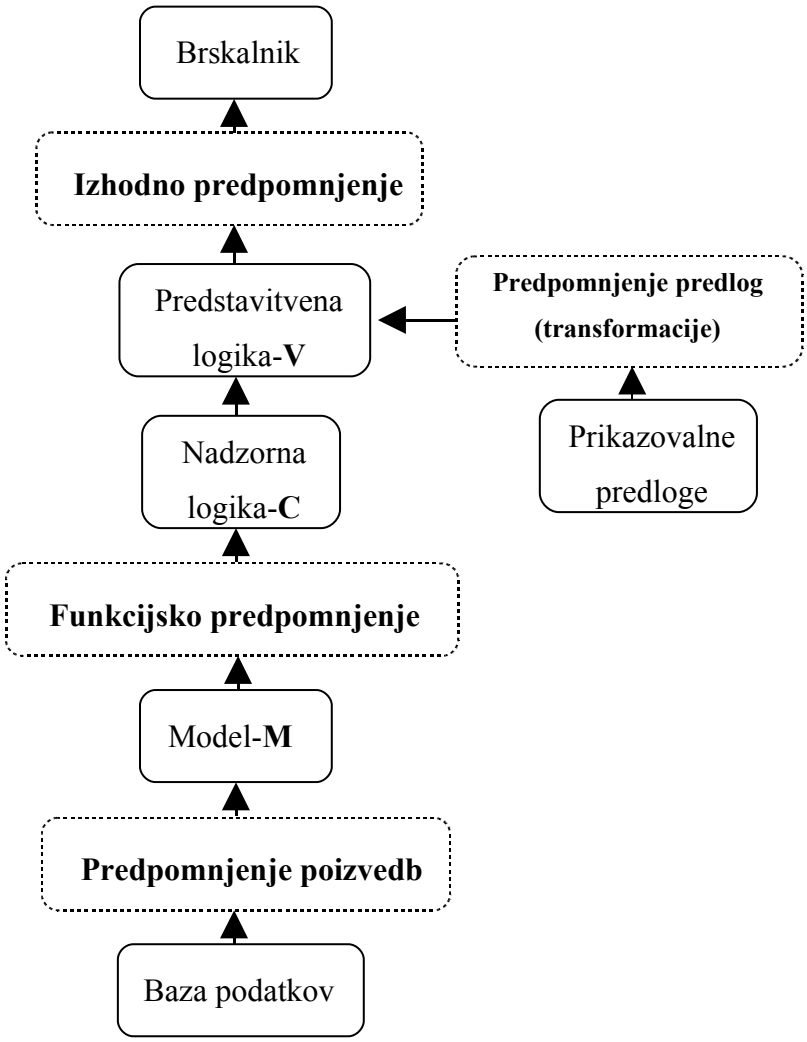
Prejle smo omenili vlogo *reverse proxyja* na počasnih omrežjih. Ponavadi v takih primerih uporabimo tudi predpomnjenje na *reverse proxyju*, saj gredo čezenj vse zahteve in odgovori. Tako se lahko sam odloča, ali bo sploh posredoval zahtevek do strežnika ali pa bo uporabil predpomnjeni odgovor.

Primer predpomnjenja na *reverse proxyju* je npr. blog. Tam večina uporabnikov le bere, nekaj

pa jih dodaja komentarje. Dokler se blog le bere, lahko uporabimo predpomnjenje na *reverse proxyju*. Ko pa se poda zahtevek za branje komentarjev ali zahtevek za njihovo vnašanje, takrat ne uporabimo predpomnjenja. Pomeni, da je mogoče **predpomnjenje celotnih dinamičnih strani**, vendar je potrebna velika mera pazljivosti in poznavanja spletne aplikacije.

3.3 Predpomnjenje z vidika dizajna

Z vidika dizajna se lahko predpomnjenje izvaja na več nivojih:



Slika 3.2: Uporaba predpomnjenja z vidika dizajna, vir [11].

Kot je razvidno z zgornje slike, lahko ob ustreznem dizajnu spletne aplikacije razdelimo predpomnjenje na štiri logične lokacije:

- izhodno predpomnjenje
- funkcijsko predpomnjenje
- predpomnjenje poizvedb
- predpomnjenje predlog (transformacije)

To si bomo natančneje ogledali v kasnejših podpoglavjih.

3.4 Predpomnjenje na strežnika

Pri dinamičnih aplikacijah imamo največ nadzora nad predpomnjenjem, če to poteka kar na strežniku (strežniku, ki izvaja kodo PHP). Predpomnjenje ne poteka za celotne strani, o čemer je bilo govora do sedaj, ampak le za določene komponente. Ponavadi so potrebni posegi v kodo in tudi dizajn mora biti ustrezen (kot prikazuje Slika 3.2).

Pomembne so tri zadeve:

- ustrezen **podatek**, ki ga želimo predpomniti
- ustrezen **ključ**, ki enolično identificira podatek
- **kontejner**, kjer je predpomnjeni podatek shranjen

Psevdokoda zgleda takole:

```
if(imamo zadetek) {
   if(podatek veljaven) {
      podatek je veljaven, preberi in preberi podatek glede na ključ...
   }
   else {
      podatek je neveljaven
      pridobi originalni podatek
      predpomni podatek glede na ključ
   }
}
else {
    pridobi originalni podatek
    predpomni podatek glede na ključ
}
```

Kot vidimo, se uporabljajo tri akcije:

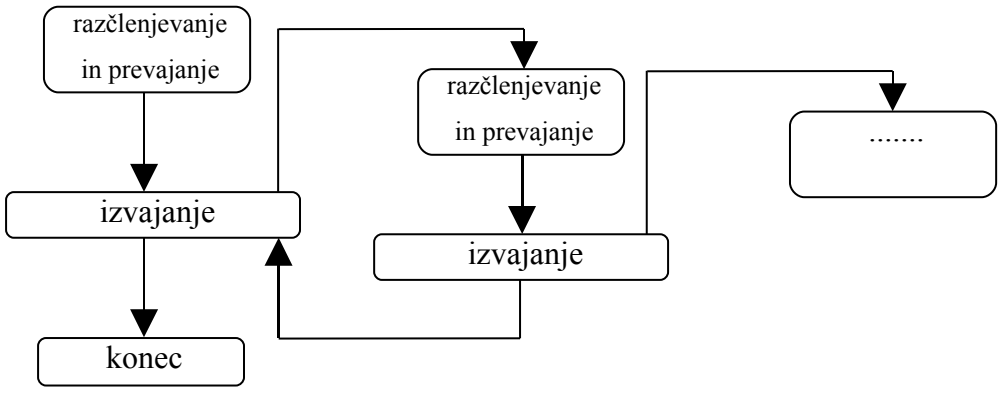
- preverjanje **zadetka**
- preverjanje **veljavnosti** podatka
- **branje ali predpomnjenje** podatka glede na ključ

Način preverjanja zadetka in veljavnosti podatka je odvisen predvsem od uporabljenega kontejnerja, branje ali predpomnjenje podatka glede na ključ pa je odvisno od tipa predpomnjenja.

3.4.1 Predpomnjenje vmesne kode PHP

V Poglavju 2.3 smo omenili, da gre vsaka skripta PHP skozi dve fazi: razčlenjevanje in prevajanje ter izvajanje. Drugače povedano, to pomeni, da v prvi fazi skripto PHP najprej prebere t. i. leksikalni analizator (*lexer*)[4], ki preveri sintakso kode in generira poseben tok žetonov (*tokens*). Ti gredo v t. i. razčlenjevalnik (*parser)*, ki žetone **prevede** v množico inštrukcij, imenovano vmesna koda (*intermediate code)*, ki jo potem **izvede** (in še pred tem jo optimizira) Zend Engine.

Znotraj faze izvajanja se ponovi enak postopek za morebitne ostale skripte PHP, ki jih glavna skripta PHP kliče, kar prikazuje spodnja slika.



Slika 3.3: Ponavljanje faz skript PHP

Velika slabost je, da se omenjene faze **ponovijo** za čisto vsako zahtevo. Prednost pa je ta, da je za vsako zahtevo vzpostavljeno novo okolje, kar pomeni, da sesutje neke zahteve ne vpliva

4 Izhaja iz zveze: *lexical analyzer*.

na drugo zahtevo.

Vendar lahko nastane problem, saj je lahko čas razčlenjevanja in prevajanja nezanemarljiv, še zlasti v primeru, ko je vključenih veliko skript PHP. Za ta namen uporabimo **predpomnjenje vmesne kode PHP**. Ko se za neko skripto PHP prvič odvije faza parsanja in prevajanja, se njen rezultat, tj. vmesna koda, shrani. Tako ta faza ob naslednji zahtevi ni več potrebna, če se skripta PHP ne spremeni[5] in če imamo na voljo dovolj prostora za predpomnjeno vmesno kodo. Obstaja več komercialnih in odprtokodnih rešitev za ta tip predpomnjenja, ki v celoti poskrbijo za pravilno delovanje. Prednost tega predpomnjenja je v tem, da se koda PHP tega sploh ne zaveda.

3.4.2 Izhodno predpomnjenje (*output buffering*)

Gre za princip, kjer predpomnimo dele kode (X)HTML, npr. kodo za prikazovanje menija. Namesto da se vedno znova obdeluje logika za njegovo prikazovanje, lahko nek del (ali celo celotni odgovor strežnika) predpomnimo. Namreč PHP svoj izhod[6] (ali pa le določene dele izhodov) hrani v svojem izhodnem vmesniku. Njegovo vsebino pa je mogoče predpomniti. Ta način je predvsem primeren za zahtevke po straneh, na katerih je vsa vsebina statična.

3.4.3 Funkcijsko predpomnjenje

Gre za princip, kjer predpomnimo rezultat klica neke funkcije, npr. izračun zakonskih zamudnih obresti pri istih vhodnih parametrih. Sem spada tudi predpomnjenje celotnih objektov[7], vendar je to smiselno le v primeru, če vemo, da bo ta objekt enak skozi več zahtevkov. Običajno se tega poslužujemo pri spletnih servisih in podobnih počasnih zadevah.

---

5 Ta se lahko spremeni samo ob nadgradnjah, ki se pa zgodijo ob načrtovanem času, ki je znan vnaprej. Do neskladja podatkov tako ne more priti.

6 Izhod je nekaj, kar je rezultat skripte PHP in se tako pošlje odjemalcu kot odgovor na zahtevo, npr. (X)HTML..

7 Pri shranjevanju objektov v PHP-ju je treba biti pozoren. Običajno shranimo objekt v sejo, vendar ga je pred tem treba serializirati – spremeniti v tekstovni tok, kar pa ni vedno mogoče (npr. če objekt vsebuje variable tipa *resource)*. Več o tem je zapisano na: http://si.php.net/manual/en/language.oop.serialization.php

Primer:

```
public function getData($param1, $param2) {
..
}
```

Če sta parametra $param1 in $param2 enaka, kot npr. velja za prejšnji zahtevek, lahko namesto klicanja telesa funkcije, vrnemo kar predpomnjeni rezultat zadnjega klica.

3.4.4 Predpomnjenje predlog (transformacij)

Gre za princip, kjer predloge vnaprej naložimo v glavni pomnilnik in jih tako predpomnimo. Predloge ponavadi vsebujejo stacionaren del odgovora strežnika in pravila, kako se naj dinamičen del odgovora strežnika predstavi odjemalcu.

Mogoče pa je tudi predpomnjenje transformacije XSLT. Gre za pretvorbo toka XML s pomočjo predloge XSL v (X)HTML. Če vedno transformiramo isti XML z isto predlogo, lahko to operacijo predpomnimo.

3.4.5 Predpomnjenje poizvedb

Gre za princip predpomnjenja poizvedb nad bazo podatkov. Tako določenih zamudnih poizvedb ni treba izvesti vedno znova, temveč lahko njihov rezultat predpomnimo. Princip je podoben funkcijskemu predpomnjenju, ki se ob ustreznem dizajnu aplikacije uporabi namesto predpomnjenja poizvedb.

3.5 Problemi pri predpomnjenju spletnih aplikacij

3.5.1 Sočasnost dostopa

Gre za situacijo, ko obstaja več zahtev za isti podatek, ki je predpomnjen. Tako lahko pride do sočasnosti branja ali celo do sočasnosti branja in pisanja: npr. ena zahteva bere podatek, druga pa hkrati piše vanj novo vsebino. To se lahko zgodi v primeru, če je podatek neveljaven in se zato poišče original, ki zamenja neveljavni podatek, med tem časom pa neka druga zahteva bere. Rešitve, ki rešujejo zgoraj opisane situacije, so odvisne predvsem od uporabe

kontejnerja – prostor, kjer so predpomnjeni podatki.

3.5.2 Preverjanje veljavnosti podatka

Nek predpomnjen podatek je neveljaven, zastrupljen, ko se njegova originalna vsebina spremeni, torej ko je prišlo do pisanja v orginal. Kako ugotoviti, da je neka vsebina predpomnjenja zastrupljena, torej neveljavna? Z dobrim poznavanjem delovanja sistema lahko uporabimo časovni parameter, ki pove, do kdaj je nek podatek veljaven.

Primer tega je naslednja situacija. S pomočjo poizvedbe nad bazo prikazujemo vsa vknjižena plačila določene osebe. Rezultat, ki ga poizvedba vrne, predpomnimo. Nato se zgodi sprememba v bazi, ker se vnese novo plačilo za isto osebo. Ker do vknjižbe pride ob točno določenem času, npr.: enkrat na uro, in sicer ob polni uri (ob 9h, 10h itd.), lahko nastavimo možnost, da je predpomnjeni podatek veljaven do polne ure, saj vemo, da do predpomnjenja pride le ob polnih urah.

Druga možnost je, da vodimo centralni imenik, kjer imamo shranjeno informacijo, kateri podatki so predpomnjeni. Tako se ob spremembi originala v tem imeniku označi, da je podatek neveljaven.

Včasih je dovolj tudi to, da rečemo, da bo nek predpomnjeni podatek veljaven npr. 15 minut. Če se pred iztekom časa original spremeni, kljub temu prikažemo neveljaven podatek, kjer to napako preprosto toleriramo. Taka situacija je npr. prikazovanje novic. Te se osvežujejo na vsakih 15 minut. Pogoj preverjanja veljavnosti bi potem bil videti takole:

```
if(čas nastanka predpomnjenega podatka + veljavnost podatka > trenuten čas) {
   ...podatek je veljaven...
}
```

3.5.3 Vzdrževanje velikosti

Ker je velikost prostora, namenjena predpomnjenju, ponavadi[8] premajhna za vse podatke, ki jih želimo predpomniti, moramo izbrati ustrezno zamenjevalno strategijo, ki bo ustrezne

8 V praksi se lahko zgodi, da velikost prostora za predpomnjenje ni premajhna, predvsem zaradi poceni glavnega pomnilnika, katerega del rezerviramo za namen predpomnjenja.

podatke odstranila. Prav tako je treba paziti na to, da ne hranimo brez potrebe neveljavnih predpomnjenih podatkov, zato jih lahko odstranimo s posebnim procesom, t. i. *garbage collection*. Tukaj imamo možnost, da preverimo veljavnost vseh predpomnjenih podatkov. Če je teh predpomnjenih podatkov veliko, je s takšnim načinom preverjanja preveč dela. V tem primeru lahko odstranimo samo tiste neveljavne podatke, za katere v tistem trenutku vemo, da so neveljavni – npr. to lahko izvemo iz centralnega imenika.

3.5.4 Skladnost predpomnjenih podatkov, koherenca

Pri distribuiranem predpomnjenju (ko predpomnimo na več strežnikih) lahko pride do neskladij. Če na enem mestu podatke zamenjamo z novimi, postanejo podatki na ostalih strežnikih neveljavni. Na tem mestu lahko predpostavimo, da se s tovrstnim prepomnjenjem ne bomo ukvarjali, oz. če imamo distribuirano okolje, bo predpomnjenje potekalo na centralnem mestu, npr. prek datotečnega sistema NFS. Obstajajo pa tudi druge namenske rešitve za ta problem[13].

3.6 Kontejnerji

Kam shraniti predpomnjene podatke? Obstaja več različnih možnosti. Od njih je odvisno reševanje zadev, opisanih v Poglavju 3.4. Zato si bomo posamezne kontejnerje pogledali v tem kontekstu.

3.6.1 Navadne datoteke

Podatek, ki ga želimo predpomniti, zapišemo v svojo datoteko, kar pomeni, da obstaja toliko datotek, kot je predpomnjenih podatkov. Ugotavljanje zadetka tako pomeni ugotavljanje obstoja datoteke. Branje iz predpomnjenega podatka tako pomeni branje celotne datoteke.

Vzdrževanje velikosti

Katero zamenjalno strategijo uporabiti? Če razmišljamo o LRU (*least recently used*), potem moramo ugotoviti, katera datoteka je bila npr. nazadnje uporabljena – ta bo zamenjana z novo. To lahko naredimo s funkcijo PHP stat(), ki poda razne informacije o datoteki. Velik problem pa je v tem, da je rezultat te funkcije predpomnjen s strani operacijskega sistema (OS), kar pomeni, da ne dobimo vedno pravega podatka – in takrat LRU odpade. Če predpomnjenje

stat() funkcije v OS izklopimo, hitro ugotovimo, zakaj je bilo sprva sploh vklopljeno. Namreč sistemski klic, ki poda informacije o datotekah, je zelo počasen. Podobne težave imamo tudi pri drugih strategijah, ki potrebujejo zgoraj omenjeni stat() – npr. strategija FIFO (*first in first out*).

Če se odločimo, da bomo imeli velik kontejner, zato da bo prej omenjenih težav z zamenjalnimi strategijami čim manj, potem pridemo do novega problema. Število datotek lahko zelo naraste, kar pa lahko določenim datotečnim sistemom povzroči težave, namreč veliko število datotek znotraj enega direktorija lahko upočasni celotni datotečni sistem. Tukaj pridejo prav rešitve, ki jih uporabljajo poštni strežniki (tam se pojavljajo podobne težave); predpomnjenje lahko izvedemo tako, da določene podatke vstavimo v zanje določen direktorij ipd.

V praksi se izkaže[18], da je najbolje, da velikost kontejnerja ni prevelika. Kot zamenjevalna strategija se uporabi ”ad-hoc” strategija, npr. taka, ki sprazni celoten kontejner ali del le-tega.

Preverjanje veljavnosti podatka

Ker vsak podatek predstavlja svojo datoteko, pomeni čas modifikacije datoteke hkrati še čas nastanka predpomnjenega podatka. Zato ga lahko uporabimo pri preverjanju.

Sočasnost dostopa

Če več procesov bere eno datoteko, ni nobenih težav. Težave nastanejo takrat, ko nek proces začne pisati v datoteko, medtem pa drug proces začne z branjem iste datoteke, namreč slednji proces bo ob preverjanju, ali datoteka obstaja, dobil pozitiven odgovor. Tega problema se lahko lotimo z uporabo dveh pristopov: *file locks* ali *file swaps*.

S prvim načinom onemogočimo hkratno branje in pisanje v datoteko. V nadaljevanju se je treba odločiti med neblokirajočim in blokirajočim načinom delovanja. Pri neblokirajočem načinu (*non-blocking*) proces, ki mu je onemogočeno branje, ne čaka na sprostitev datoteke, ampak nadaljuje z izvrševanjem (posledično pride do zgrešitve). Če uporabimo blokirajoči način (*blocking*) pa tvegamo, da bo zahtevek predolgo čakal na sprostitev datoteke in bi morebiti bilo hitreje, če v tistem trenutku sploh ne bi uporabljali predpomnjenja. Prav tako

moramo upoštevati dejstvo, da na določenih datotečnih sistemih, npr. NFS, opisana rešitev ne deluje.

Z uporabo drugega načina za rešitev problema (*file swaps*) dosežemo, da pisanje nikoli ne poteka v isto datoteko, kot se uporablja za branje. Pišemo v neko začasno datoteko, ki jo kasneje preimenujemo. Upoštevati je treba, da je začasna datoteka v istem datotečnem sistemu, saj je le takrat preimenovanje atomska operacija. Ker uporabljamo eno datoteko za en predpomnjeni podatek, je slednji način bolj primeren[9].

3.6.2 Datoteke DMB

Datoteke DBM so bile razvite z namenom, da omogočijo hiter in sočasen dostop. Velika razlika v primerjavi z navadnimi datotekami je ta, da za podatek, ki ga želimo predpomniti, ne potrebujemo svoje datoteke, ampak lahko hranimo v eni datoteki več različnih prepomnjenih podatkov. Tako bi lahko ravnali tudi pri navadnih datotekah, vendar bi v tistem primeru morali skrbeti za pravilno pomikanje datotečnega kazalca. Za sistem PHP obstaja več razširitev, ki omogočajo delo z datotekami DMB. Tako obstaja vrsta funkcij, posledično pa poteka ugotavljanje zadetka na naslednji način:

```
$gdbm = dba_popen("datoteka.dbm", "c", "gdbm");
...
if(($predpomnjenPodatek = dba_fetch($kljuc, $gdbm)) !== false) {
        ...
}
```

Tukaj lahko opazimo, da nam klic funkcije *dba_fetch(..)* vrne false v primeru zgrešitve.

Vzdrževanje velikosti

Največja slabost uporabe datotek DMB je, da prostor na disku, ki ga zasedejo datoteke DBM, lahko samo raste – nikoli se sam ne zmanjša. To pomeni, da če odstranimo kakšen podatek iz datoteke DBM, bo njena velikost na disku še vedno enaka. Preprosta rešitev je, da lahko datoteke odstranimo in ponovno kreiramo samo za dejanske podatke, in sicer znotraj t. i. procesa *garbage collection*. Ker je ta proces časovno zahteven, je treba paziti na to, kako

9 Namreč način *file locks* lahko uporabljamo tudi nad posameznimi deli datotek - v takih primerih je omenjen način bolj primeren.

pogosto ga izvajamo. Ponavadi za ta namen naredimo sistemski posel (*job*), ki pa se s pomočjo operacijskega sistema samodejno izvaja ob točno določenih časovnih intervalih.

Preverjanje veljavnosti podatka

Problematično je, da datoteke DBM ne vsebujejo podatkov o času nastanka predpomnjene vsebine, ki jo hranijo. Torej ne vemo, ali je vsebina sploh veljavna. Zato je treba čas nastanka zapisati dodatno poleg vsebine v datoteko DMB in ga uporabiti pri preverjanju veljavnosti. Konkretno lahko to pomeni naslednje: za vsak podatek, ki ga želimo predpomniti, ustvarimo tabelo z dvema elementoma: podatek in čas nastanka predpomnjenega podatka. Tabelo potem zapišemo v datoteko DBM kot dodaten predpomnjeni podatek.

Sočasnost dostopa

Datoteke DBM vsebujejo mehanizme, ki na pregleden način omogočajo sočasnost dostopa.

3.6.3 SHM

Tukaj igra vlogo kontejerja t. i. deljeni pomnilnik (*shared memory*), ki je ponavadi velik blok v glavnem pomnilniku. Ta ima sposobnost, da si ga procesi med seboj delijo, zato je takšen kontejner zelo hiter.

Obstaja več produktov, ki omogočajo uporabo SHM za namen predpomnjenja skript PHP. Tukaj si bomo pogledali produkt APC – Advanced PHP Cache, ki ima tudi API za programerje. Ugotavljanje zadetka je zelo podobno kot pri datotekah DBM:

```
if(($predpomnjenPodatek = apc_fetch($kljuc)) !== false) {
        ...
}
```

Vidimo, da nam klic funkcije *apc_fetch(..)* vrne false v primeru zgrešitve.

Vzdrževanje velikosti

Ponavadi obstaja možnost nastavitve velikosti kontejnerja, prav tako pa so na voljo funkcije, s katerimi izvajamo *garbage-collection*. Obvezno je treba spremljati fragmentacijo kontejnerja, saj gre za isto zadevo kot pri datotečnih sistemih (s tem, da je SHM bistveno hitrejši).

Fragmentacijo je pri produktu APC mogoče spremljati s posebnim pregledovalnikom, ki si ga bomo natančneje ogledali malce kasneje (pri meritvah).

Ugotavljanje veljavnosti podatka

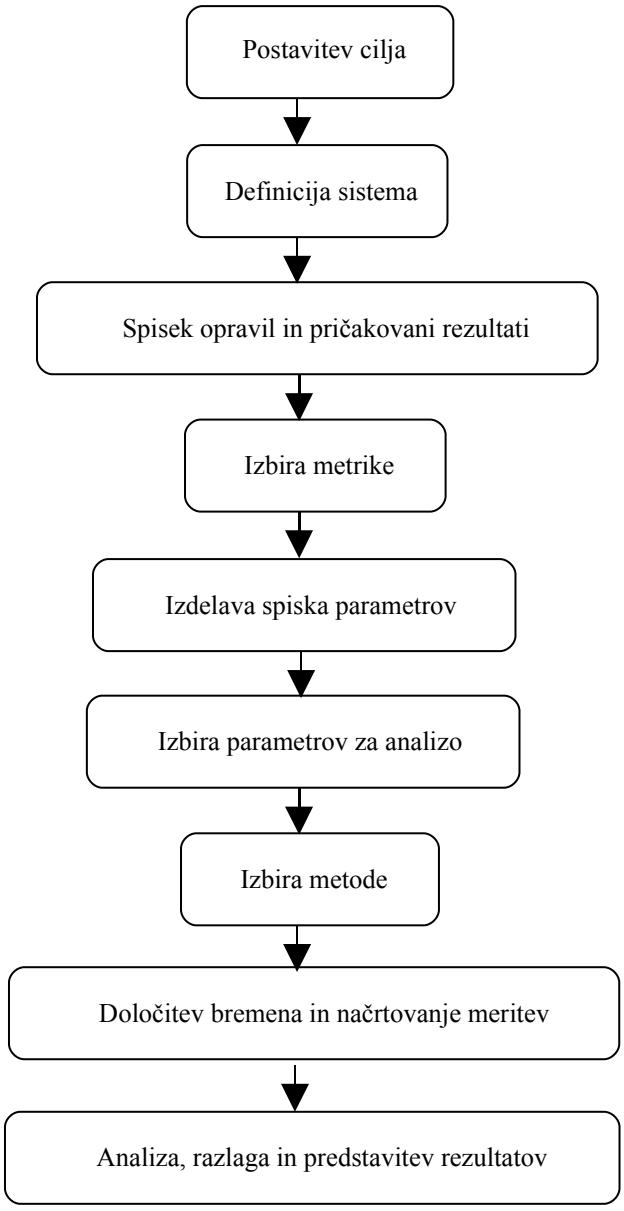
Pri ugotavljanju veljavnosti podatka se uporablja enaka strategija kot pri datotekah DBM, kjer čas nastanka predpomnjenega podatka zapišemo poleg samega podatka. Obstaja tudi druga možnost, in sicer v primeru produkta APC. Tukaj je mogoče v nastavitveni datoteki deklarativno nastaviti čase veljavnosti za celoten kontejner.

Sočasnost dostopa

Tudi področje sočasnosti dostopa je mogoče urejati deklarativno z nastavitveno datoteko. Obstaja več načinov, ki lahko bistveno vplivajo na hitrost. Več o tem na [19].

4. POSTOPEK ANALIZE ZMOGLJIVOSTI

Uporabili bomo naslednji pristop:



Slika 4.1: Pristop k zmogljivostni analizi, vir [24].

4.1 Postavitev cilja

Zelo pomembno je, da pred začetkom analize določimo cilje in da dobro poznamo opazovani sistem. V nasprotnem primeru bi analiza bila nesistematična, njeni rezultati pa bi bili

neuporabni. To še zlasti velja za primer predpomnjenja, kjer igra bistveno vlogo ocena, kdaj bo nek predpomnjeni podatek neveljaven.

Cilja sta dva:

1. **Izboljšati (zmanjšati) odzivni čas sistema pri enem uporabniku**. Odzivni čas sistema je definiran kot čas, ki ga zahteva prebije na strežniku.
2. **Povečati propustnost sistema pri več aktivnih uporabnikih**.

Pri obeh ciljih se je treba nujno zavedati, da lahko gre v praksi za kompromis med obema ciljema. Večja hitrost izvrševanja (oz. krajši čas izvrševanja) lahko povzroči manjšo propustnost pri večjem številu aktivnih uporabnikov. Poglejmo si krajši primer[10].
Spletna aplikacija mora prebrati datoteko, obdelati njeno vsebino in izpisati rezultat obdelave. Pristop 1 je takšen, da se datoteka v celoti prebere v pomnilnik, kar pomeni, da je obdelava hitrejša (krajši odzivni čas) kot pri Pristopu 2, kjer se datoteka bere in obdeluje po kosih, kajti potrebnih je več počasnih sistemskih klicev. Prav tako velja, da Pristop 1 porabi za en zahtevek odjemalca približno 10 MB pomnilnika. Torej če imamo strežnik s približno 500 MB pomnilnika (ki je na voljo za to spletno aplikacijo), lahko strežnik hkrati postreže 50 uporabnikov. Pri Pristopu 2 porabi en zahtevek odjemalca približno 5 MB pomnilnika, kar na istem strežniku pomeni hkratno strežbo 100 uporabnikov.

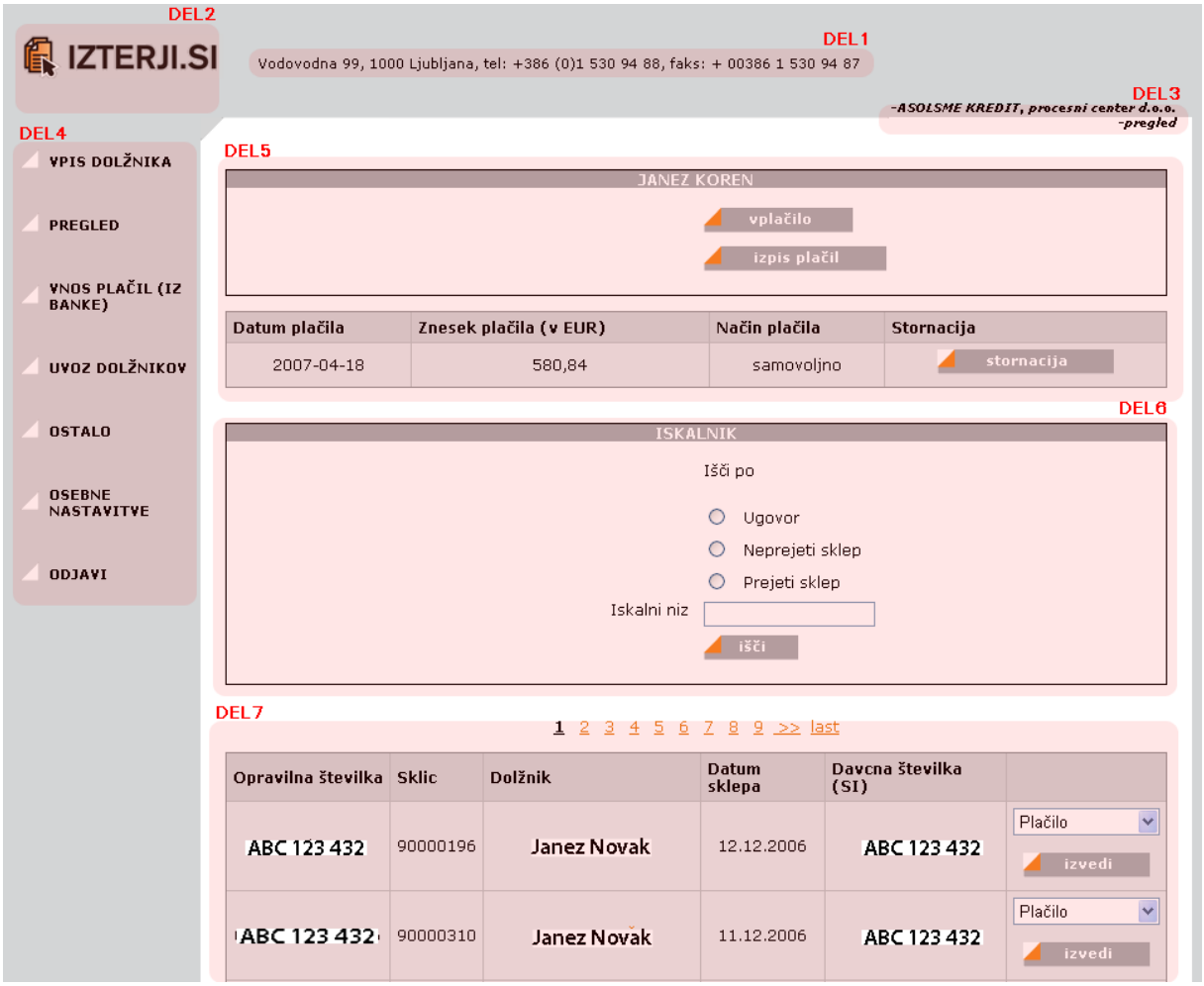
4.2 Definicija sistema

Kot sistem bomo opazovali del spletne aplikacije Payforcement®[11], in sicer tisti del, ki je najbolj kritičen (najbolj obiskan in najpomembnejši z vidika uporabnosti aplikacije). V splošnem (v realnosti) takšna definicija sistema ne zadošča, namreč naključno bi bilo treba opazovati več delov spletne aplikacije (oz. celotno aplikacijo). Vendar bomo za namen te diplomske naloge, ki želi prikazati vpliv različnih metod predpomnjenja na zmogljivost (za zgoraj postavljena cilja), zavzeli stališče, da je to dovolj dobra definicija opazovanega sistema.

10 Pri tem primeru smo zanemarili razne malenkosti pri sami strežbi zahtevkov v zvezi s porabo pomnilnika. Namen primera je samo prikazati koncept.

11 Gre za sistem za vodenje izvršilnih postopkov podjetja Odkup terjatev d.o.o.

Na kratko bi lahko sistem lahko opisali takole: gre za vodenje izvršilnih postopkov določenim dolžnikom. Postopek, ki ga opazujemo, je pregled plačil določenega dolžnika. Aplikacija je napisana v PHP-ju. Za vizualni del se uporabljata XSLT in podatkovna baza PostgreSql, kjer so shranjeni vsi dinamični podatki.



Slika 4.2: Opazovani del spletne aplikacije, definiran kot sistem pri zmogljivostni analizi.

4.2.1 Definicija sistema z vizualnega vidika

Na Sliki 4.2. vidimo, da gre za množico sedmih vizualnih delov, in sicer: pregled dolžnikov (DEL 7) z iskalnikom (DEL 6), nad katerim je prikazan postopek vnašanja plačil (DEL 5). Na strani je tudi statičen del – logotip (DEL 2) in dinamični deli, ki se redko spreminjajo: glava strani (DEL 1), podpis prijavljenega uporabnika (DEL 3) in meni (DEL 4). Popoln seznam objektov *front-end* je vključen v Poglavje 2.1.2.

4.2.2 Definicija sistema z vidika dizajna

Velja, da sistem sledi t. i. vzorcu MVC (pattern)[8]. To pomeni, da lahko uporabljamo različne tehnike predpomnjenja glede na dizajn, kot je bilo prikazano na Sliki 3.2. Sam pomen vzorca MVC: veliko problemov nastane, če (spletna) aplikacija vsebuje mešanico programske kode, namenjene uporabniški interakciji z aplikacijo, nadzoru nad izvajanjem, poslovni logiki in predstavitvi podatkov. Na tak način vnesemo mednje odvisnosti, kar pomeni, da je vzdrževanje težje, ponovna uporaba slabša, nadgradnje posameznih delov aplikacije težje, s čimer pomeni tudi težjo uporabo predpomnjenja. Vzorec MVC reši ta problem z jasno ločitvijo programske kode na poslovno logiko, predstavitveno logiko in nadzorno logiko.

Poslovna logika našega sistema

Vključuje naslednje:

- dostope do podatkovnih virov in izvajanje nad njimi

Ti dostopi potekajo preko posebnih objektov, ki predstavljajo abstrakcijo dostopa do podatkov. Ti objekti tako omogočajo, da se v modelu izvajajo abstraktni klici, npr. podajVseElementeVMenuju(..), sama implementacija je pa skrita.

- pošiljanje sprememb poslovne logike do predstavitvene logike[12] v obliki toka XML podatkov

Logika kontrole našega sistema

Vključuje naslednje:

- vzpostavitev povezave do baze
- avtorizacijo uporabnikov
- vzpostavljanje seje, kjer se seja hrani v relacijski bazi (seja po definiciji zagotavlja obstojnost podatkov čez več zahtevkov)
- vhodno in izhodna validacijo (varnostni mehanizem, kjer se preverjajo vhodni podatki v strežnik in izhodni podatki s strežnika)
- mehanizem CSRF (*cross-site request forgery*)

  Še en varnostni mehanizem, ki preprečuje ponarejanje zahtevkov po določenih straneh, in sicer tako, da strežnik generira žetone, ki jih pošlje odjemalcu, ta pa potem pošlje zahtevek s temi žetoni, ki potem strežniku omogočajo identificiranje pravih

12 Gre za t.i. *push model*, kjer poslovna logika skrbi, da ima predstavitvena logika ustrezne (in predvsem ažurne) podatke. Obstaja še t.i. *pull model*, kjer predstavitvena logika poskrbi, da iz poslovne logike pridobi ustrezne podatke.

(neponarejenih) zahtevkov.

- paginacije (možnost pregleda podatkov na “več straneh”)
- mehanizmi *autoload* (dinamično nalaganje vseh potrebnih skript PHP)
- lokalizacija (ustrezen jezik, merske enote itd.)
- prepoznavanje uporabnikovih akcij
- itd.

Pri našem sistemu bomo zanemarili čas za vzpostavo seje in čas za avtorizacijo uporabnika, kar pomeni, da se bomo postavili v točko, ko rečemo, da ima uporabnik že vzpostavljeno sejo, in da je že avtoriziran.

Predstavitvena logika našega sistema

Vključuje:

- transformacijo XSLT toka podatkov XML s pomočjo predlog XSL[23]

  Rezultat transformacije je (X)HTML, ki se prikaže v odjemalčevem brskljalniku. Predloge XSL se tako morajo naložiti v pomnilnik in se skupaj s tokom XML uporabijo pri transformaciji.

4.3 Spisek opravil in pričakovani rezultati

Glede na zgoraj razdelan sistem lahko sestavimo seznam opravil, ki jih bomo opazovali v sistemu in opišemo pričakovane rezultate predpomnjenja.

Opravilo	Opombe in pričakovani rezutati
1.) Dostopi do podatkovnih virov (relacijske baze) dao_language dao_certificate_user dao_constant dao_creditor dao_debtor dao_execution dao_menu dao_product_instance dao_payment_debtor dao_state_new	**Ozko grlo:** Ker gre za dostope do baze, predstavlja ta del ozko grlo. **Predpomnjenje:** Predpomnjenje pride v poštev v večini primerov zaradi dovolj velike statičnosti vsebine – za večino vsebine vnaprej vemo, kdaj se spremeni. **Pričakovanje:** Ker je ozko grlo veliko, se pričakuje znatna izboljšava.
2.) Pretvorba podatkov v XML libField_Caption libField_Dropdown libField_Abstract libField_Radio libField_Submit libField_Text DOMDocument->saveXML	**Ozko grlo:** Le v primeru prikaza ogromne količine podatkov na en zahtevek. Takrat je treba ustvariti ogromno takih objektov (> 500), ki generirajo XML. **Predpomnjenje:** Predpomnjenje ni smiselno, saj gre večino časa za kreiranje objektov, tega pa ni mogoče predpomniti. Predpomnjenje samih objektov ne bi prineslo pohitritev, ker so objekti zelo majhni. **Pričakovanje:** –

Tabela 4.1: Seznam opravil sistema (del: poslovna logika)

Opravilo	Opombe in pričakovani rezutati
3.) Vzpostavitev povezave do baze	**Ozko grlo:** Sama vzpostavitev lahko predstavlja ozko grlo, še zlasti v primeru oddaljene podatkovne baze. **Predpomnjenje:** Učinek predpomnjenja je mogoče doseči s t. i. persistentnimi povezavami (*persistent connection*), kjer povezava do baze ostane odprta še za naslednje zahtevke. Slednje je smiselno samo v primeru, če bi vzpostavljanje trajalo veliko časa, saj take povezave povzročajo težave glede transakcij, zaklepanja tabel ipd. [15] **Pričakovanje:** Ker bo baza postavljena na istem strežniku, bo čas za vzpostavitev povezave kratek. Persistentne povezave ne bodo potrebne.
4.) Mehanizem *autoload*	**Ozko grlo:** Ker gre za počasne operacije z datotečnim sistemom, predstavlja ta del ozko grlo. **Predpomnjenje:** Predpomnjenje je smiselno s predpomnjenjem vmesne kode PHP. **Pričakovanje:** V preteklosti, ko je bilo diskovje počasnejše, je takšno predpomnjenje bilo bistvenega pomena. Danes ne pričakujemo večjih izboljšav.
5.) Mehanizem CSRF	**Ozko grlo:** Gre za generiranje žetonov (za vsako formo na zahtevek) s pomočjo funkcije *rand* in za njihovo preverjanje. V primeru velikega števila form na en zahtevek (veliko žetonov) lahko tudi to postane ozko grlo, sicer pa ne. **Predpomnjenje:** Predpomnjenje ni smiselno, ker statičnost

Opravilo	Opombe in pričakovani rezutati
	podatkov ne obstaja. **Pričakovanje:** –
6.) Paginacija	**Ozko grlo:** Gre za operacijo deljenja in zaokroževanja (da se ugotovi, koliko podatkov se prikaže na eno stran), ki sta počasni operaciji. **Predpomnjenje:** Predpomnjenje je smiselno, saj lahko preprosto ugotovimo, če gre za enake parametre, nad katerimi je treba izvesti paginacijo. V takem primeru lahko potem predpomnimo objekt, ki skrbi za paginacijo. **Pričakovanje:** Ne pričakujemo bistvenih izboljšav, saj je računska moč CPU danes velika.
7.) Vhodna in izhodna validacija	**Ozko grlo:** Povečini gre za preverjanje vsebine podatkov s pomočjo regularnih izrazov. Če podatkov na zahtevek ni ogromno, potem to ni ozko grlo. **Predpomnjenje:** Predpomnjenje ni smiselno, ker gre za klice vgrajenih funkcij PHP, ki so zelo hitre. **Pričakovanje:** –

Tabela 4.2: Seznam opravil sistema (del: nadzorna logika)

Opravilo	Opombe
8.) XSLT (transformacija)	**Ozko grlo:** Gre za transformacijo toka podatkov XML s pomočjo predloge XSL. V primeru večjega toka podatkov in večje kompleksnosti predloge, lahko gre za ozko grlo. **Predpomnjenje:** Predpomnjenje počasne operacije XSLT je smiselno. **Pričakovanje:** Pričakuje se delna pohitritev.

Tabela 4.3: Seznam opravil sistema (del: predstavitvena logika)

Glede na opisan sistem bomo vse zahtevke po *front-endu* pri testiranju izklopili, saj lahko tako bolje [22] preizkusimo predpomnjenje na strežniku. Tako bo smiselno uporabljati predvsem funkcijsko predpomnjenje, predpomnjenje vmesne kode PHP in predpomnjenje predlog (transformacije).

Zaradi prevelike dinamičnosti vsebine ni smiselno uporabljati izhodnega predpomnjenja in predpomnjenja na *reverse proxyju*. Predpomnjenje poizvedb prav tako ni smiselno, ker do podatkovne baze dostopamo prek objektov, tako da bomo to področje pokrili s funkcijskim predpomnjenjem.

4.4 Izbira metrike

Glede na zastavljene cilje bomo izbrali naslednji metriki:

- METRIKA 1: **odzivni čas sistema pri enem uporabniku**, ki bo v našem primeru definiran kot čas, ki ga zahteva prebije na strežniku (brez upoštevanja časa pošiljanja zahtevka na strežnik in vzpostavitev povezave med odjemalcem in strežnikom).
- METRIKA 2: **maksimalna propustnost sistema** (število zahtev, postreženih v opazovanem času) **glede na število hkratnih oporabnikov** pod pogojem, da **ne** pride do nasičenja.

4.5 Izdelava spiska parametrov, ki vplivajo na zmogljivost

4.5.1 Pomožni parametri

So parametri, ki se v času izvajanja meritev ne spreminjajo, vendar kljub temu vplivajo na zmogljivost.

Pomnilnik	2 x FBDIMM DDR 2 1024 MB@ 666 MHz
CPE	Dual-Core Intel® Xeon® Processor 5130 (2 GHz) 4 x 333 Mhz – 1333 MHz 65 nm (Woodcrest) 4 MB L2
Matična plošča	Supermicro X7DAL (socket LGA 771) Chipset: 5000 X, vodilo: Intel AGTL+; 4 x 333 MHz (FSB), 2 x kanala FSB, 64 bit
Disk	WD360GD (Raptor, 37 GB, SATA150, 8 MB Cache, vklopljen DMA: hdparm – d1)
Omrežje	Mrežni vmesnik na strežniku: 2 x Intel PRO/1000 (GB) Omrežje: stikalo Netgear (100 MB)
Operacijski sistem	Gentoo Linux Profile: default-linux/amd64/2007.0 stage3 install, prevajanje izvedeno s CFLAGS: "-o2 -march=nocona -pipe" gcc4.1.2, glibc-2.6.1, kernel 2.16.17-gentoo-r8
Datotečni sistem	4x particije: /, /boot, /work_files swap
Spletni strežnik *front-end*	Varnish 2.0.4, izklopljen za namene testiranja
Spletni strežnik *back-end*	Apache 2.2.10
PHP	PHP 5.2.9.r2
Podatkovna baza	PostgreSql 8.0.15
Povezava med Apache in PHP	PHP kot Apache MTM *prefork*
Pomembne nastavitve PHP	Glej Prilogo 1: Izpis nastavitev PHP
Pomembne nastavitve Apache	KeepAlive Off[13] HostnameLookups Off[14] Options FollowSymLinks[15] AllowOverride None[16] <IfModule mpm_prefork_module>

13 Ker bomo izvajali meritve brez zahtevkov po objektih *front-end* (slike ipd.), smo direktivo *KeepAlive* nastavili na *Off*

14 Gre za poizvedovanje DNS po IP-jih odjemalcev za namene logiranja **[4]**.

15 Ta direktiva povzroči to, da se zmanjša število sistemskih klicev zaradi preverjanja, ali je nek resurs simboličen link **[4]**.

16 Ta direktiva povzroči to, da se zmanjša število sistemskih klicev zaradi preverjanja dostopov za vsak zahtevan resurs **[4]**.

	StartServers 5 MinSpareServers 5 MaxSpareServers 10 MaxClients [17] 256 MaxRequestsPerChild 10000 </IfModule> Glej Prilogo 2: Izpis nastavitev strežnika Apache
Pomembne nastavitve PostgreSQL	uporaba persistentnih povezav: NE uporaba logiranja: DA max_connections = 300 shared_buffers = 1000 (8 KB vsak) work_mem = 1024 KB maintenance_work_mem = 16384 KB max_stack_depth = 2048 KB Glej Prilogo 3: Izpis nastavitev PostgreSql
Velikost *bufferjev* TCP	128 KB
Velikost testirane spletne strani	33.3 KB[18]
Proces Lingerd	DA, kot modul strežnika Apache
Izhodna (podatki, ki potujejo s strežnika k odjemalcu) kompresija podatkov	NE

Tabela 4.4: Strežniški pomožni parametri

CPE	Intel Pentium M 1.60 GHz
Pomnilnik	2 x DDR2 1024 MB (333)
Omrežje	Broadcom NetExtreme 57 Gigabit Controller
Operacijski sistem	Windows XP SP2

Tabela 4.5: Odjemalčevi pomožni parametri

Pomembno je, da ne pride do nasičenja na odjemalcu, zato bomo pri metriki 2, opazovali tudi resurse na odjemalcu. Da se prepričamo, da odjemalec ni v nasičenju, je treba preverjati

17 Glede na **[5]** se vrednost te direktive (ta določa število hkratnih strežb – v našem primeru hkratnih procesov; če je zahtev več od tega števila, potem zahteve čakajo v vrsti) lahko izračuna, in sicer se pogleda povprečna velikost procesa. To vrednost delimo s prostim pomnilnikom, npr.: 2048 MB/ 8 MB (na proces) = 256, ampak nekaj pomnilnika mora ostati na voljo tudi za ostale stvari. Glede na naše meritve je 150 dovolj veliko število.

18 Pomeni, da ne bo blokiranja prenosov TCP, ker gre celotna stran v vmesnik TCP.

resurse odjemalca. Zato večina programske opreme, namenjene meritvam, vsebuje tudi module za merjenje resursov odjemalca. Na splošno velja, da 100-odstotna zasedenost CPE na odjemalcu še ne pomeni, da je odjemalec v nasičenju. V takem primeru se nasičenost ponavadi preverja tako, da se meritve izvedejo na več odjemalcih. Šele če pride do večjih odstopanj, lahko sumimo, da gre za nasičenje odjemalca.

4.6 Pomembni parametri in njihova izbira za analizo

To so parametri, faktorji, ki se v času meritev spreminjajo in vplivajo na zmogljivost. Uporabili jih bomo pri testiranju.

1.) Uporaba APC (predpomnjenje vmesne kode PHP). Nivo: DA/NE.
2.) Uporaba funkcijskega predpomnjenja na datotečnem sistemu. Nivo: DA/NE.
3.) Uporaba funkcijskega predpomnjenja v SHM. Nivo: DA/NE.
4.) Uporaba predpomnjenja predlog XSL v SHM. Nivo: DA/NE.

4.7 Izbira metode

Uporabljali bomo meritve v umetnem sistemu.

4.8 Načrtovanje meritev

Na splošno lahko trdimo, da se problem izvajanja meritev lahko prevede na iskanje minimuma ali maksimuma, tj. ekstrema funkcije z več spremenjivkami [24]. Pri metriki 1 (odzivni čas sistema) nas tako zanima minimum, pri metriki 2 (propustnost) pa maksimum.

Najpreprostejši način iskanja ekstrema je npr. spreminjanje prvega faktorja, pri čemer pustimo ostale faktorje nespremenjene. Na drugem koraku nastavimo prvi faktor na najbolj ugoden nivo, ob tem pa spreminjamo drugi faktor. Število potrebnih meritev je malo, vendar je doseženi rezultat zgolj lokalni ekstrem funkcije[19].

Glede na naše parametre lahko sestavimo naslednje scenarije meritev:

19 Drugi način je izvedba vseh možnih meritev.

Parameter	Scenarij 1	Scenarij 2	Scenarij 3	Scenarij 4	Scenarij 5
Uporaba APC (predpomnjenja vmesne kode PHP)	NE	**DA**	**DA**	**DA**	**DA**
Uporaba funkcijskega predpomnjenja na dat. sistemu	NE	NE	**DA**	NE	NE
Uporaba funkcijskega predpomnjenja v SHM	NE	NE	NE	**DA**	**DA**
Uporaba prepomnjenja predlog XSL (transformacij) v SHM	NE	NE	NE	NE	**DA**

Tabela 4.6: Načrt meritev

Glede na zgornje imamo 5 scenarijev meritev. Pri vsem tem je potrebno upoštevati še število ponovitev meritev zaradi statistične relevantnosti.

4.8.1 Vzdrževanje velikosti kontejnerja

Pri uporabi **predpomnjenja na datotečnem sistemu** smo rezervirali dovolj prostora (8 GB) za celotno predpomnjenje. Število datotek, ki predstavljajo predpomnjene podatke, ne bo povzročalo težav[7], saj lahko brisanje kontejnerja poteka periodično ponoči, ko je prometa zelo malo (ali ga sploh ni).

Pri **predpomnjenju v SHM** smo prav tako rezervirali dovolj prostora (30 MB), saj se celotni kontejner pobriše enkrat na uro, omenjena velikost pa je dovolj velika za potrebe aplikacije[20].

20 Zasedenost SHM se lahko spremlja periodično prek grafičnega vmesnika produkta APC, mogoče pa je tudi zapisovanje v dnevniške datoteke, ki jih lahko kasneje tudi analiziramo.

4.8.2 Sočasnost dostopa

Pri uporabi **predpomnjenja na datotečnem sistemu** smo uporabili rešitev tipa *file swaps*, ki je opisana v Poglavju 3.5.1. Pri uporabi predpomnjenja v SHM pa uporabljamo način, imenovan *pthread mutex locks*[19].

4.9 Določitev bremena

Breme po svoji definiciji predstavlja množico zahtev, ki jih mora sistem uspešno obdelati. Glede na test bomo izbrali različna bremena:

1.) Najprej bomo izvedli **zmogljivostni test** (*performance test*), in sicer bomo opravili pet različnih meritev (glede na scenarije). Posamezno meritev bomo opravili trikrat (str. 70, vir: [24]).

Breme bo predstavljala zahteva enega uporabnika, opazovali pa bomo odzivni čas sistema pri enem uporabniku, zanima nas namreč njegov minimum.

Namen tega testa je, da ugotovimo, katera opravila (definirana v Tabelah 4.1-3) se bodo izvajala hitreje (ali počasneje), glede na parametre in njihove nivoje.
Za meritev bomo uporabili *profiler*[21], ki bo izvajal analizo programske kode na strežniku.

2.) Nato bomo za vse scenarije izvedli še t. i. ***ramp test*** – gre za variacijo stresnega testa (*stress test*)[22]. Tak sistem bomo opazovali dalj časa, in sicer 10 minut. Dodatno bomo izmerili, pri kakšnem bremenu pride do nasičenja in kolikšna je takrat propustnost sistema. Nasičenje lahko v tem primeru pomeni dvoje: ali se je strežnik odzval z napako kode 404[23] ali pa je odzivni čas zahtevka presegel 10 sekund.

Opazovali bomo odzivni čas sistema glede na število aktivnih uporabnikov. Med samo meritvijo bo število hkratnih uporabnikov enakomerno naraščalo (zato *ramp test*). Zgornja meja glede števila uporabnikov bo 150, čas med posameznimi zahtevki (čas za razmišljanje)

21 Gre za produkt Xdebug, http://www.xdebug.org

22 Gre za test, kjer se ugotovi, kaj se dogaja z opazovanim sistemom, ko je breme večje kot ocena, npr. ob t.i. špicah.

23 Gre za napako kode standarda HTTP, in sicer 404 pomeni napako *Not found*.

pa bo 1 sekunda. Za merjenje bomo uporabili Webserver Stress Tool 7[24].

5.) ANALIZA, RAZLAGA IN PREDSTAVITEV REZULTATOV

5.1 Profil opazovanega sistema

Najprej bomo izvedli analizo programske kode opazovanega sistema, da vidimo, kje se pojavljajo ozka grla oz. kakšni so deleži posameznih opravil, definiranih v Tabelah 4.1-3. Uporabili bomo prej omenjeno orodje Xdebug.

Razultat je predstavljen v Tabeli 5.1:

1.) Dostopi do podatkovnih virov (relacijske baze):	
dao_language:	10 ms
dao_certificate_user:	0,2 ms
dao_constant:	0,4 ms
dao_creditor:	0,1 ms
dao_debtor:	2,2 ms
dao_execution:	25,1 ms
dao_menu:	0,3 ms
dao_product_instance:	0,3 ms
dao_payment_debtor:	0,2 ms
dao_state_new:	0,2 ms
	SKUPAJ: **39,0 ms**
2.) Pretvorba podatkov v XML	
libField_Caption:	2 ms
libField_Dropdown:	0,2 ms
libField_Hidden:	0,8 ms
libField_Radio:	0,1 ms
libField_Submit:	0,2 ms
libField_Text:	0,1 ms
libContent*	0,3 ms
DOMDocument->saveXML:	SKUPAJ**: 3,7 ms**
3.) Vzpostavitev povezave do baze	
	SKUPAJ**: 0,4 ms**
4.) Mehanizem *autoload*	
	SKUPAJ**: 8,3 ms**

24 http://www.paessler.com/webstress

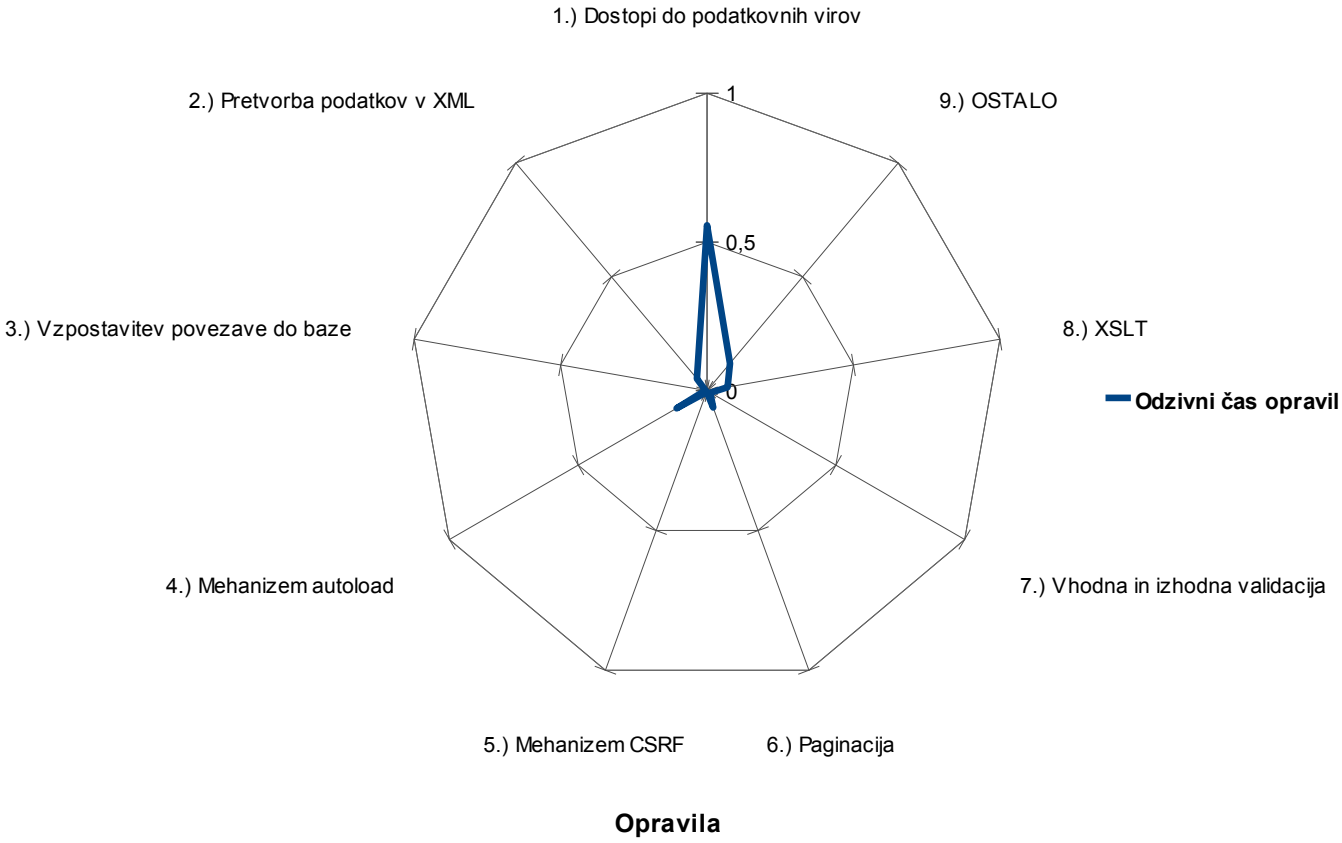
5.) Mehanizem CSRF	
	SKUPAJ**: 0,3 ms**
6.) Paginacija	
	SKUPAJ**: 4,2 ms**
7.) Vhodna in izhodna validacija	
	SKUPAJ**: 0,9 ms**
8.) XSLT	
	SKUPAJ**: 4,9 ms**
9.) OSTALO[25]	
	SKUPAJ**: 8,4 ms**

Tabela 5.1: Odzivni čas sistema pri enem uporabniku glede na opravila sistema (Scenarij 1)

Po pričakovanjih so najpočasnejša opravila tista, ki so povezana z dostopi do podatkovnih virov, kar nam prikazuje tudi naš Kiviatov diagram (Graf 5.1).

25 Gre za razne pomožne postopke, ki jih težko uvrstimo v seznam opravil, med njimi tudi ravnanje s sejo, ki se hrani v bazi.

Graf 5.1: Kiviatov diagram deležev opravil v odzivnem času sistema



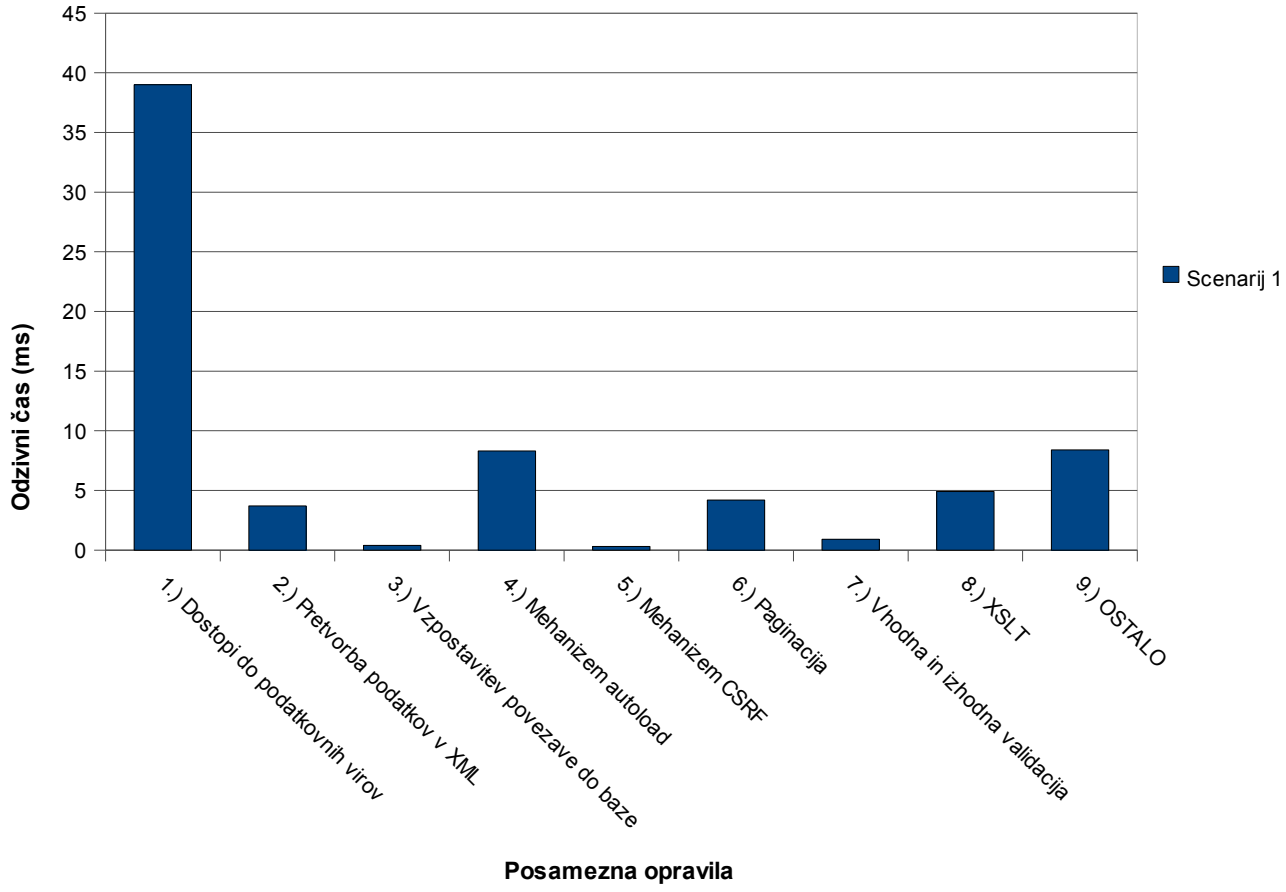
Ker je prikazani poligon neenakomeren – ima dve špici: večjo pri Opravilu 1 *(Dostopi do podatkovnih virov)* in manjšo pri Opravilu 4 *(Mehanizem autoload)*, se pri teh opravilih pojavijo ozka grla. S predpomnjenjem jih bomo poskusili odpraviti.

5.2 Scenarij 1 – perfomančni test

Scenarij 1 predstavlja situacijo, kjer nobeden od štirih parametrov iz Tabele 4.6 ni aktiven. Rezultat te meritve je prikazan v Tabeli 5.1. Opravili smo tri zaporedne meritve in upoštevali njihovo povprečje. Skupen povprečni odzivni čas je **70.1 ms**.[26] Odzivni časi so predstavljeni na Grafu 5.2. Vidimo, da je najbolj ugodno iskati rešitve, ki bodo pospešile dostope do podatkov.

26 Standardna deviacija znaša 0,37.

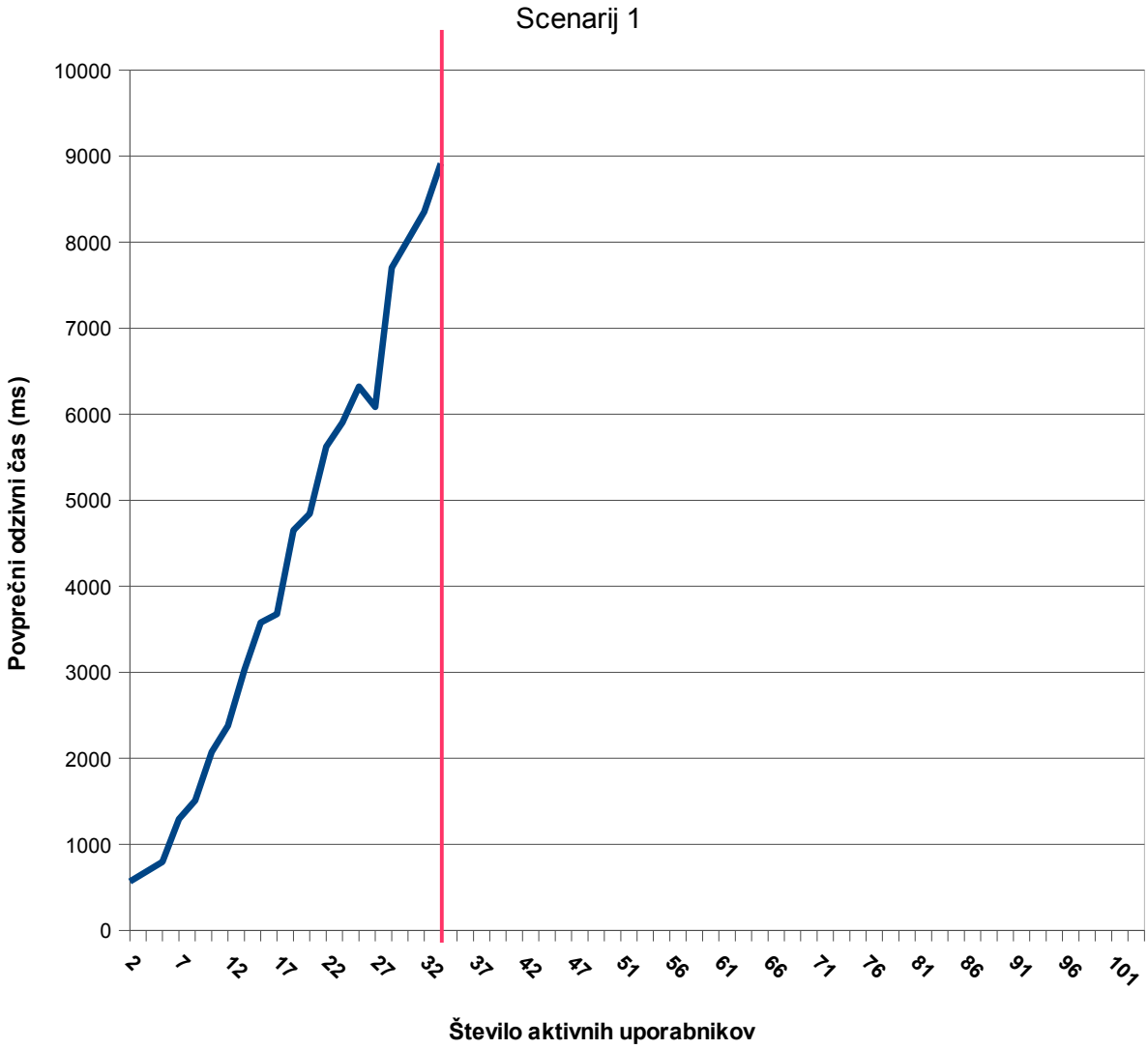
Graf 5.2: Povprečni odzivni čas opravil sistema pri enem uporabniku glede na scenarij 1



5.3 Scenarij 1 – *ramp test*

Meritev ponovimo še z *ramp testom*. Rezultat je podan na spodnjem grafu.

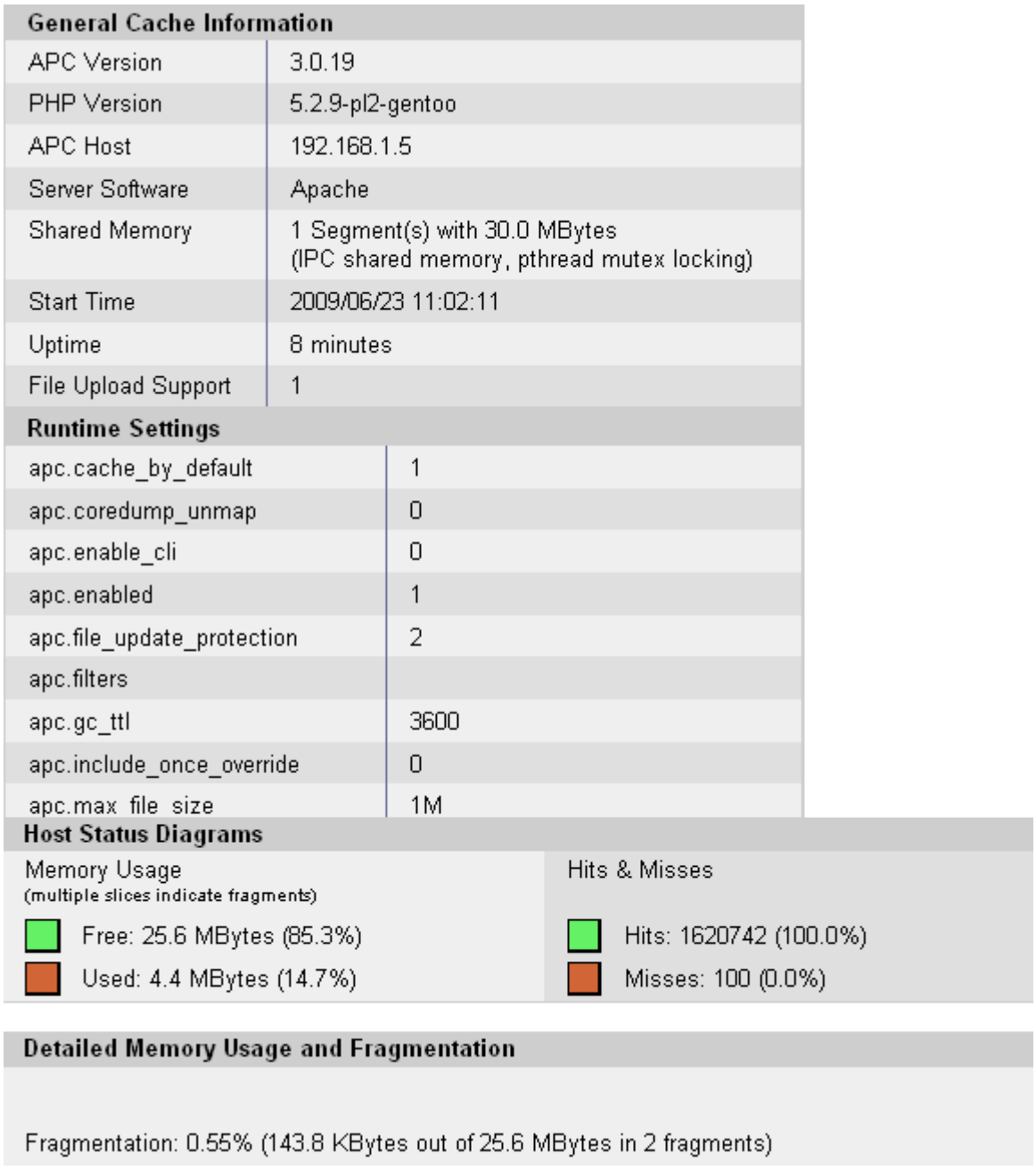
Graf 5.3: Povprečni odzivni čas sistema glede na število aktivnih uporabnikov



Z grafa je razvidno, da povprečen odzivni čas strmo raste po že 4,5 aktivnih uporabnikih. Pri 34,5 aktivnih uporabnikih pride do nasičenja (napake), kar je označeno z rdečo črto. Glede na definicijo bremena lahko ugotovimo naslednje: 34,5 aktivnih uporabnikov vsako sekundo pošlje zahtevek s povprečnim odzivnim časom 8914 ms. **Propustnost je zato enaka 34,5 zahtevkov/sekundo ali 2070 zahtevkov/minuto**.

5.4 Scenarij 2 – perfomančni test

Glede na Tabelo 4.6 je aktiven prvi parameter. Za namen predpomnjenja vmesne kode PHP je vklopljen APC. Zaradi preverjanja nastavitev, porabe SHM itd. ima APC grafični vmesnik (skripta apc.php).



General Cache Information	
APC Version	3.0.19
PHP Version	5.2.9-pl2-gentoo
APC Host	192.168.1.5
Server Software	Apache
Shared Memory	1 Segment(s) with 30.0 MBytes (IPC shared memory, pthread mutex locking)
Start Time	2009/06/23 11:02:11
Uptime	8 minutes
File Upload Support	1

Runtime Settings	
apc.cache_by_default	1
apc.coredump_unmap	0
apc.enable_cli	0
apc.enabled	1
apc.file_update_protection	2
apc.filters	
apc.gc_ttl	3600
apc.include_once_override	0
apc.max_file_size	1M

Slika 5.1: Prikaz uporabe SHM z grafičnim vmesnikom APC za opazovan sistem

S slike so razvidne nastavitve, in sicer vidimo, da je rezerviran en segment SHM v velikosti 30 MB. Od tega je zasedenega 4,4 MB, kar pomeni, da je sorazmerno malo zahtevnih skript

PHP, ki so vse predpomnjene, saj imamo 100 % zadetkov predpomnjenja. Tip zaklepanja dostopov imamo, že prej omenjen, "pthread mutex locking". Pomembna nastavitev, ki ni prikazana na zgornji sliki, je *apc.stat*. Ta je nastavljena na *Off*, kar pomeni, da se za vsak zahtevek NE preverja, ali ima original predpomnjene skripte PHP spremenjen čas modifikacije datoteke. Takšno ravnanje zmanjša število sistemskih klicev in tako pospeši predpomnjenje.

Spodaj je naveden seznam opravil in izmerjen povprečni odzivni čas opravil sistema pri enem uporabniku. Prav tako smo opravili tri zaporedne meritve in vzeli njihovo povprečje.

1.) Dostopi do podatkovnih virov (relacijske baze):	
dao_language:	9,9 ms
dao_certificate_user:	0,2 ms
dao_constant:	0,4 ms
dao_creditor:	0,1 ms
dao_debtor:	2,2 ms
dao_execution:	25,1 ms
dao_menu:	0,3 ms
dao_product_instance:	0,3 ms
dao_payment_debtor:	0,2 ms
dao_state_new:	0,2 ms
	SKUPAJ: **38,9 ms**
2.) Pretvorba podatkov v XML	
libField_Caption:	2 ms
libField_Dropdown:	0,2 ms
libField_Hidden:	0,8 ms
libField_Radio:	0,1 ms
libField_Submit:	0,2 ms
libField_Text:	0,1 ms
libContent*	0,3 ms
DOMDocument->saveXML:	SKUPAJ**: 3,7 ms**
3.) Vzpostavitev povezave do baze	
	SKUPAJ**: 0,4 ms**
4.) Mehanizem *autoload*	
	SKUPAJ**: 0,2 ms**
5.) Mehanizem CSRF	
	SKUPAJ**: 0,3 ms**
6.) Paginacija	

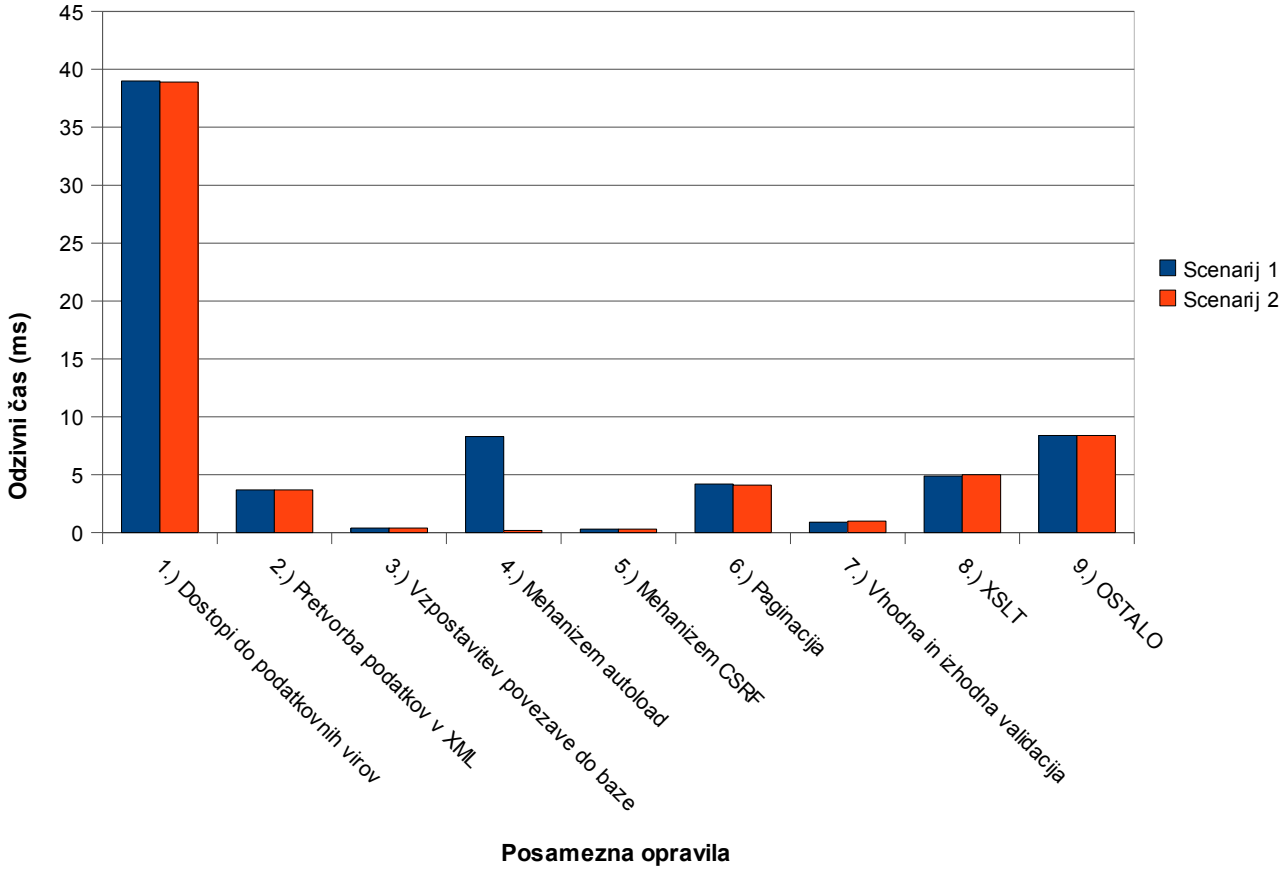
	SKUPAJ: **4,1 ms**
7.) Vhodna in izhodna validacija	
	SKUPAJ: **1,0 ms**
8.) XSLT	
	SKUPAJ: **5,0 ms**
9.) OSTALO	
	SKUPAJ: **8,4 ms**

Tabela 5.2: Odzivni čas sistema pri enem uporabniku glede na opravila sistema (scenarij 2)

Skupen povprečni odzivni čas opravil sistema pri enem uporabniku je **62 ms**.[27] Pričakovano smo zmanjšali opravilo 4 (Mehanizem autoload) zaradi predpomnjenja vmesne kode PHP. Tukaj je treba poudariti, da je prednost tega predpomnjenja bolj opazna pri sistemih, pri katerih predstavlja to opravilo večji delež v skupnem odzivnem času, npr. pri sistemih, ki vključujejo veliko število skript PHP in pri sistemih s počasnejšim diskovjem (npr. pri virtualiziranih sitemih, pri katerih je diskovje[6]) zelo obremenjeno.

Graf 5.4 prikazuje primerjavo med scenarijem 1 in scenarijem 2. Odzivni čas opravila 4 smo zmanjšali na skoraj 0 ms.

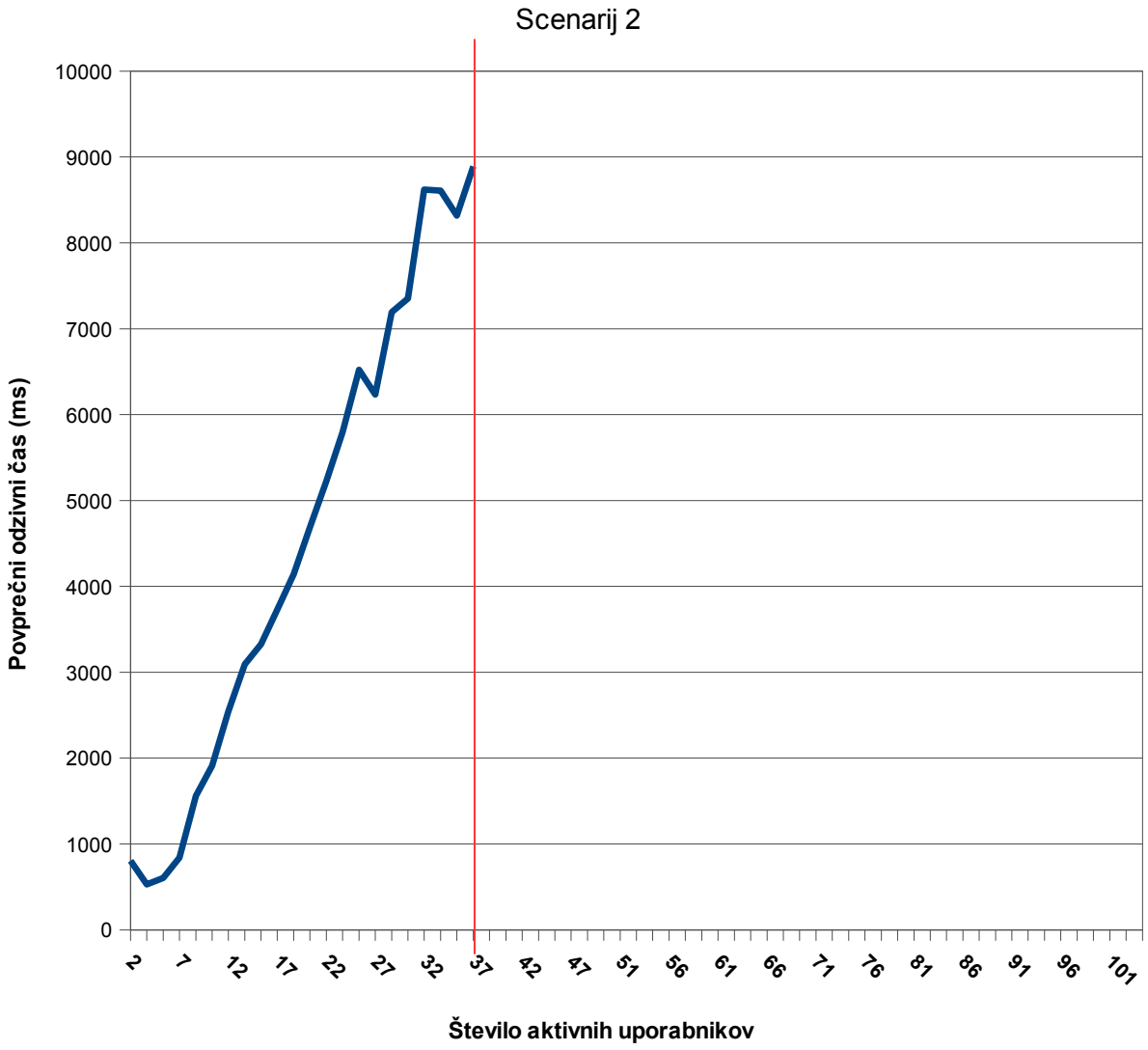
Graf 5.4: Povprečni odzivni čas opravil sistema pri enem uporabniku glede na scenarij 1, 2



5.5 Scenarij 2 – *ramp test*

Meritev smo ponovili še z *ramp testom*. Rezultat je podan na spodnjem grafu.

Graf 5.5: Povprečni odzivni čas sistema glede na število aktivnih uporabnikov



Z grafa je razvidno, da povprečni odzivni čas strmo raste po 7 aktivnih uporabnikih. Pri 37 aktivnih uporabnikih pride do nasičenja (napake), kar je označeno z rdečo črto. Glede na definicijo bremena lahko ugotovimo naslednje: 37 aktivnih uporabnikov vsako sekundo pošlje zahtevek s povprečnim odzivnim časom 9744 ms. **Propustnost je zato enaka 37 zahtevkov/sekundo ali 2220 zahtevkov/minuto**. Glede na scenarij 1 je to manjša izboljšava, ne pa bistvena.

5.6 Scenarij 3 – perfomančni test

Scenarij 3 predstavlja situacijo, v kateri sta vklopljena APC (predpomnjenje vmesne kode PHP) in funkcijsko predpomnjenje na trdem disku. Spodaj je naveden seznam opravil in izmerjen povprečni odzivni čas opravil sistema pri enem uporabniku. Prav tako smo opravili tri zaporedne meritve in vzeli njihovo povprečje.

1.) Dostopi do podatkovnih virov (relacijske baze)	
dao_language:	0,7 ms
dao_certificate_user:	0,0 ms
dao_constant:	0,1 ms
dao_creditor:	0,0 ms
dao_debtor:	0,4 ms
dao_execution:	0,1 ms
dao_menu:	0,0 ms
dao_product_instance:	0,1 ms
dao_payment_debtor:	0,0 ms
dao_state_new:	0,1 ms
	SKUPAJ: **1,6 ms**
2.) Pretvorba podatkov v XML	
libField_Caption:	2 ms
libField_Dropdown:	0,2 ms
libField_Hidden:	0,9 ms
libField_Radio:	0,1 ms
libField_Submit:	0,2 ms
libField_Text:	0,1 ms
libContent*	0,3 ms
DOMDocument->saveXML:	SKUPAJ**: 3,8 ms**
3.) Vzpostavitev povezave do baze	
	SKUPAJ**: 0,4 ms**
4.) Mehanizem *autoload*	
	SKUPAJ**: 0,4 ms**
5.) Mehanizem CSRF	
	SKUPAJ**: 0,5 ms**
6.) Paginacija	
	SKUPAJ**: 4,4 ms**
7.) Vhodna in izhodna validacija	
	SKUPAJ**: 1,1 ms**

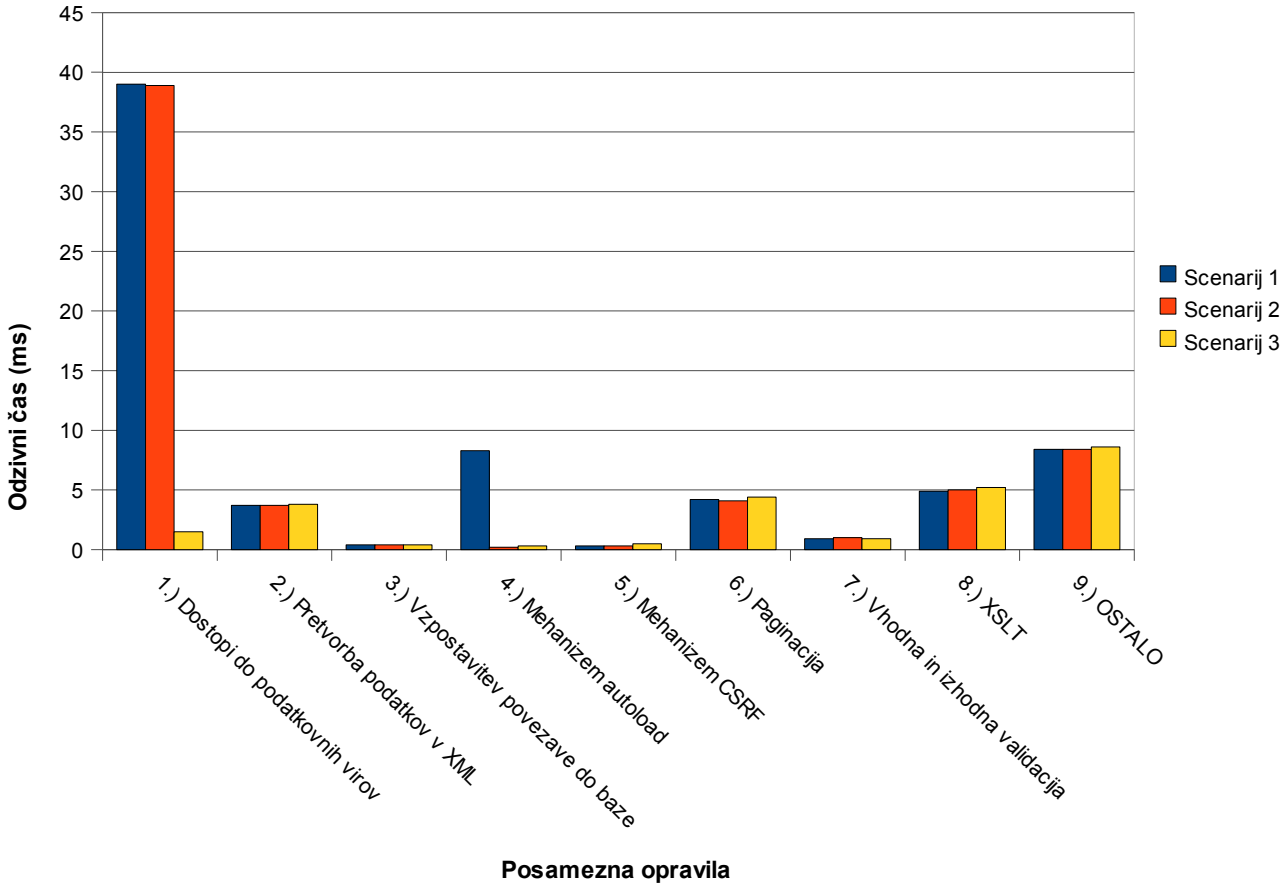
8.) XSLT	
	SKUPAJ**: 5,2 ms**
9.) OSTALO	
	SKUPAJ**: 8,6 ms**

Tabela 5.3: Odzivni čas sistema pri enem uporabniku glede na opravila sistema (scenarij 3)

Skupen povprečni odzivni čas opravil sistema pri enem uporabniku je **26 ms**.[28] Vidimo, da smo bistveno zmanjšali povprečni odzivni čas, predvsem zaradi tega, ker smo bistveno zmanjšali odzivni čas opravila 1 (Dostopi do podatkovnih virov). V tem primeru dostopov do baze v primeru zadetka sploh ni.

Spodnji graf prikazuje primerjavo med scenariji 1, 2 in 3.

Graf 5.6: Povprečnl odzivni čas opravil sistema pri enem uporabniku glede na scenarij 1, 2, 3

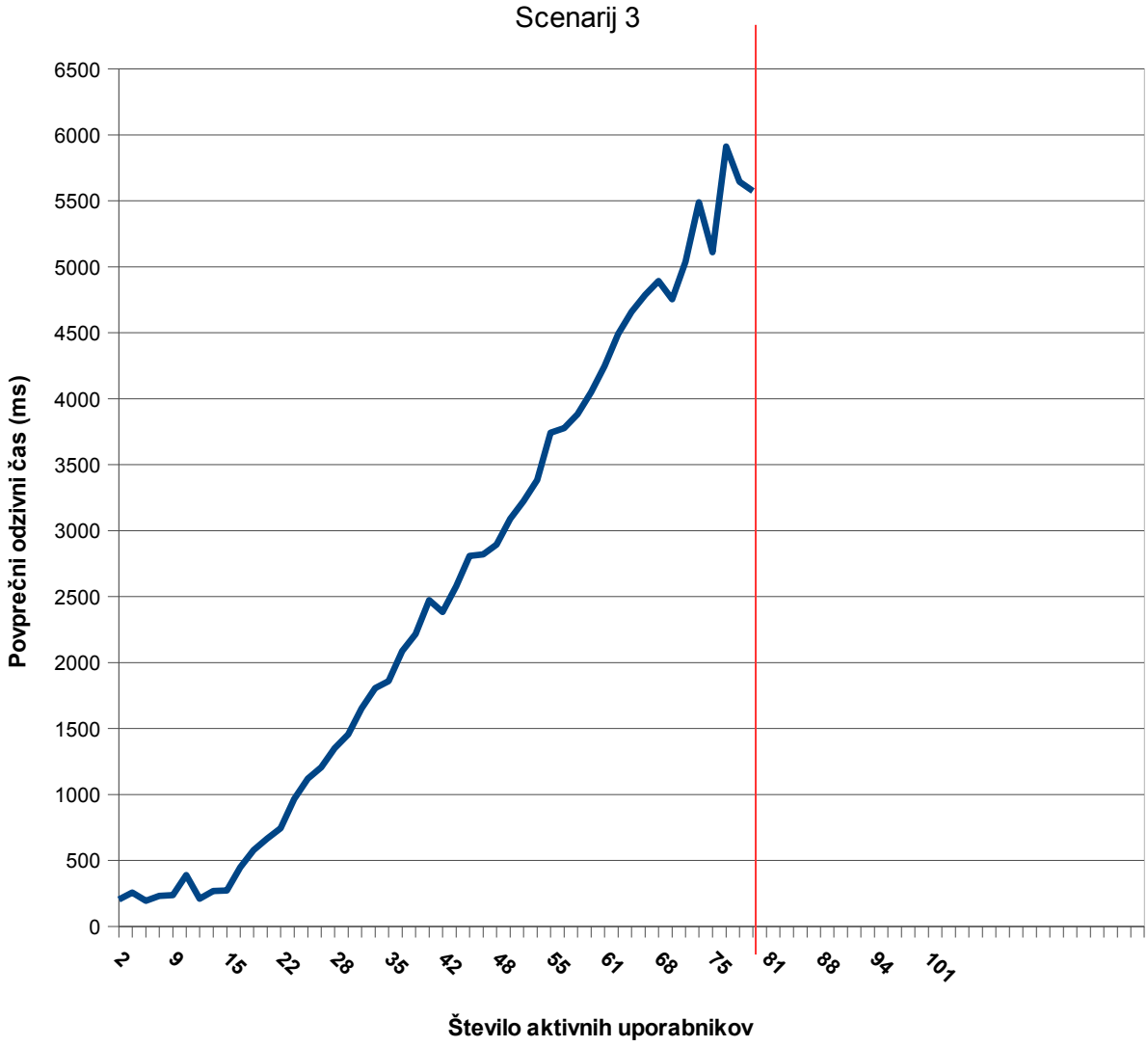


28 Standardna deviacija znaša 0,25.

5.7 Scenarij 3 – *ramp test*

Ponovimo meritev še z *ramp testom*. Rezultat je prikazan na spodnjem grafu.

Graf 5.7: Povprečni odzivni čas sistema glede na število aktivnih uporabnikov



Z grafa je razvidno, da povprečni odzivni čas strmo raste šele po 15 aktivnih uporabnikih. Pri 80 aktivnih uporabnikih pride do nasičenja (napake), kar je označeno z rdečo črto. Glede na definicijo bremena lahko ugotovimo naslednje: 80 aktivnih uporabnikov vsako sekundo pošlje zahtevek s povprečnim odzivnim časom 5574 ms. **Propustnost je zato enaka 80 zahtevkov/sekundo ali 4800 zahtevkov/minuto**. Glede na scenarij 1 in 2 je to bistvena izboljšava. Ozko grlo smo odpravili.

5.8 Scenarij 4 – perfomančni test

Scenarij 4 predstavlja situacijo, v kateri sta vklopljena APC (predpomnjenje vmesne kode PHP) in funkcijsko predpomnjenje, vendar tokrat namesto trdega diska (scenarij 3) uporabimo SHM. Spodaj je seznam opravil in izmerjeni povprečni odzivni čas opravil sistema pri enem uporabniku. Prav tako smo opravili tri zaporedne meritve in uporabili njihovo povprečje.

1.) Dostopi do podatkovnih virov (relacijske baze):	
dao_language:	0,5 ms
dao_certificate_user:	0,0 ms
dao_constant:	0,2 ms
dao_creditor:	0,0 ms
dao_debtor:	0,2 ms
dao_execution:	0,2 ms
dao_menu:	0,0 ms
dao_product_instance:	0,1 ms
dao_payment_debtor:	0,0 ms
dao_state_new:	0,1 ms
	SKUPAJ: **0,9 ms**
2.) Pretvorba podatkov v XML	
libField_Caption:	2 ms
libField_Dropdown:	0,2 ms
libField_Hidden:	0,8 ms
libField_Radio:	0,1 ms
libField_Submit:	0,2 ms
libField_Text:	0,1 ms
libContent*	0,3 ms
DOMDocument->saveXML:	SKUPAJ**: 3,7 ms**
3.) Vzpostavitev povezave do baze	
	SKUPAJ**: 0,4 ms**
4.) Mehanizem autoload	
	SKUPAJ**: 0,3 ms**
5.) Mehanizem CSRF	
	SKUPAJ**: 0,3 ms**
6.) Paginacija	
	SKUPAJ**: 4,1 ms**
7.) Vhodna in izhodna validacija	
	SKUPAJ**: 0,8 ms**
8.) XSLT	

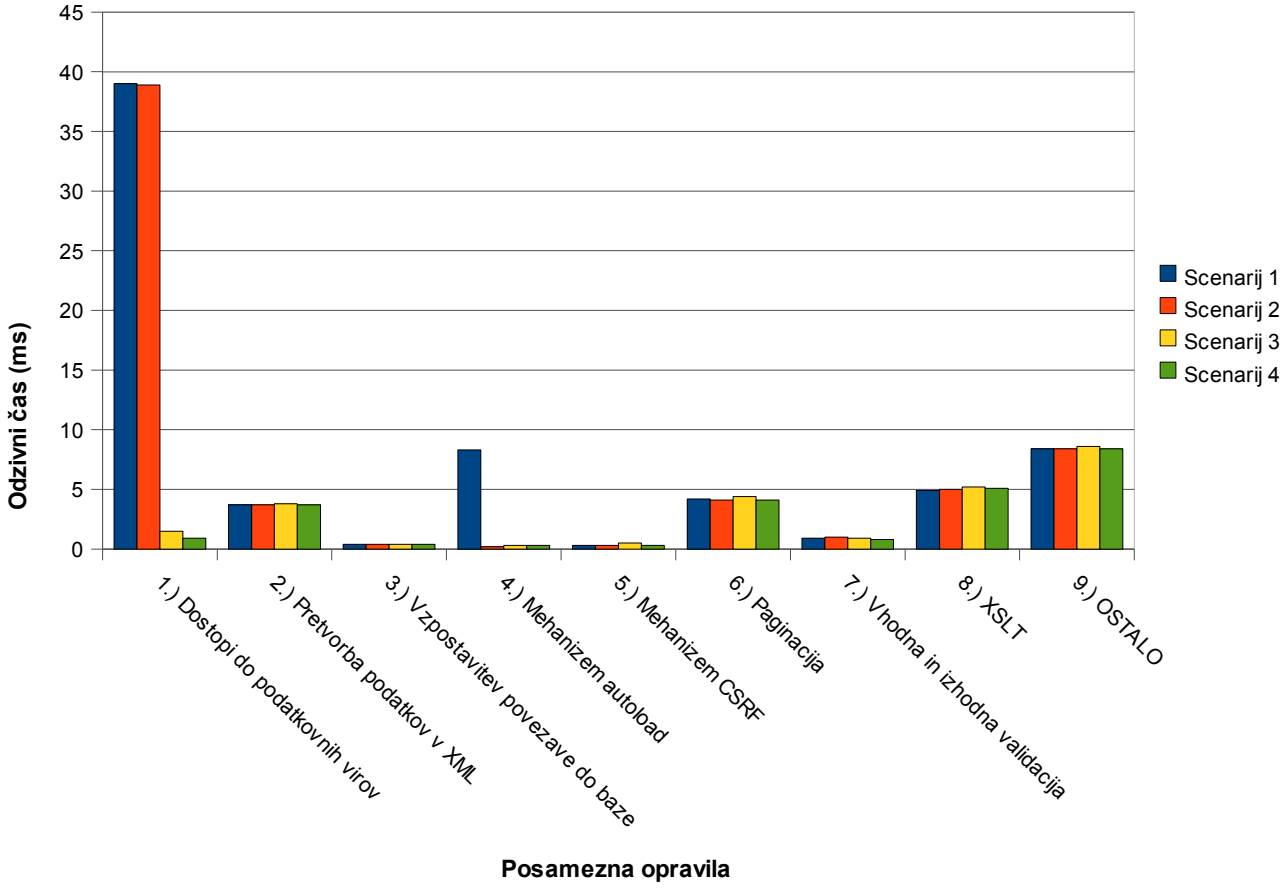
	SKUPAJ: **5,1 ms**
9.) OSTALO	
	SKUPAJ: **8,4 ms**

Tabela 5.4: Odzivni čas sistema pri enem uporabniku glede na opravila sistema (scenarij 4)

Skupen povprečni odzivni čas opravil sistema pri enem uporabniku je **24 ms**.[29] Bistvenega odstopanja od scenarija 3 ni, kljub temu, da smo za predpomnjenje uporabili SHM. Razlog tiči v tem, da so predpomnjeni podatki majhni (iz grafičnega pregleda APC lahko razberemo, da so veliki približno 4.4 MB), kar pomeni, da se ti pri predpomnjenju na disku nahajajo v hitrih vmesnikih diska in/ali operacijskega sistema (ang. buffers).

Spodnji graf prikazuje primerjavo med scenariji 1, 2, 3 in 4:

Graf 5.8: Povprečno odzivni čas opravil sistema pri enem uporabniku glede na scenarij 1, 2, 3, 4 meritev

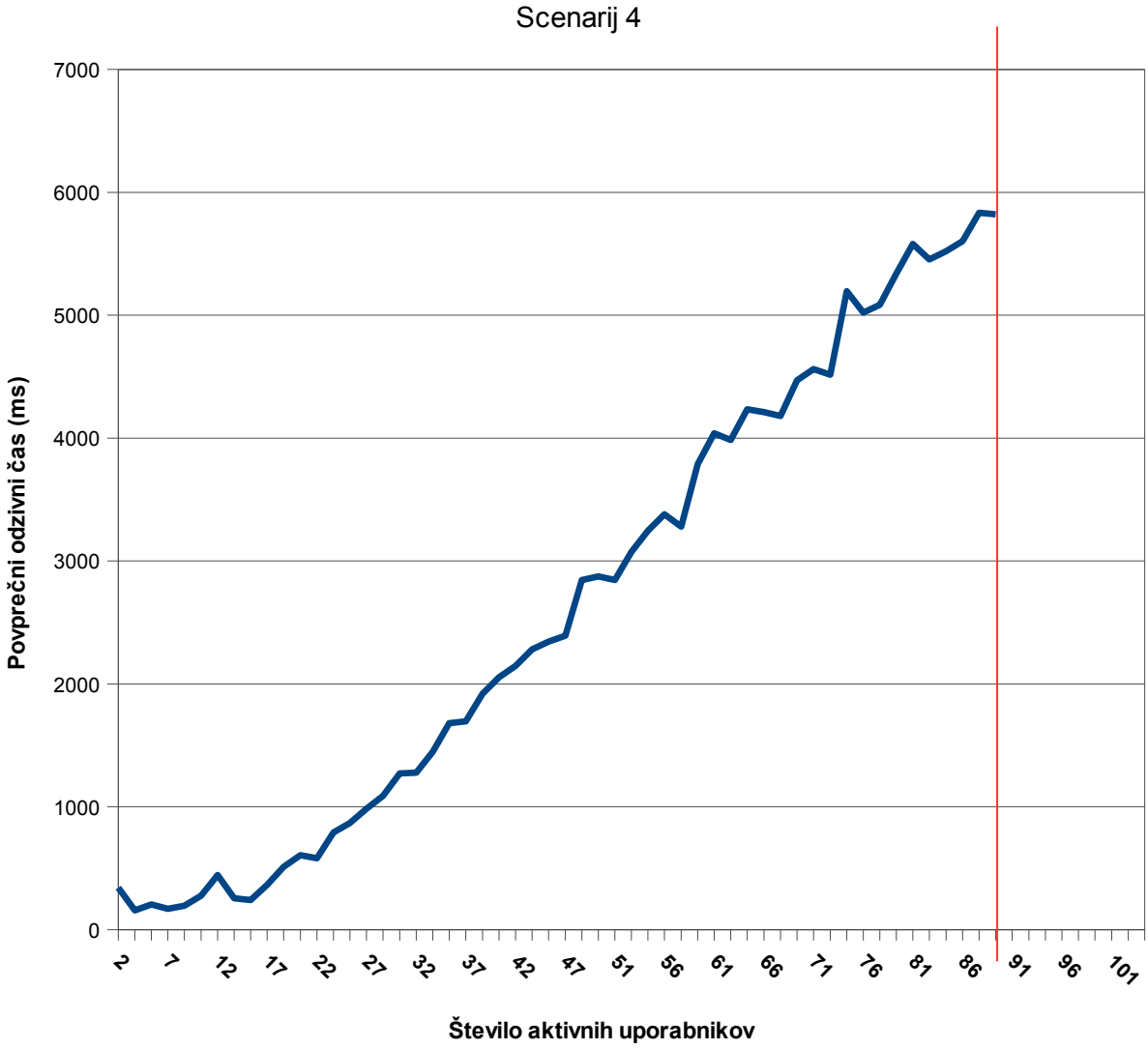


29 Standardna deviacija znaša 0,0.

5.9 Scenarij 4 – *ramp test*

Ponovimo meritev še z *ramp testom*. Rezultat je podan v spodnjem grafu.

Graf 5.9: Povprečni odzivni čas sistema glede na število aktivnih uporabnikov



Z grafa je razvidno, da so zadeve zelo podobne scenariju 3, kot smo opazili že pri perfomančnem testu. Vendar ima rezultat eno bistveno razliko: do nasičenja (rdeča črta) pride pri 89 aktivnih uporabnikih, **kar je več kot pri scenariju 3**. Glede na definicijo bremena lahko trdimo naslednje: 89 aktivnih uporabnikov vsako sekundo pošlje zahtevek s povprečnim odzivnim časom 5822 ms. **Propustnost je zato enaka 89 zahtevkov/sekundo ali 5340 zahtevkov/minuto**.

5.10 Scenarij 5 – perfomančni test

Scenarij 5 predstavlja situacijo, ko imamo vklopljen APC (predpomnjenje vmesne kode PHP), funkcijsko predpomnjenje v SHM in predpomnjenje XSLT.

Spodaj je seznam opravil in izmerjen povprečni odzivni čas opravil sistema pri enem uporabniku. Opravili smo tri zaporedne meritve in uporabili njihovo povprečje.

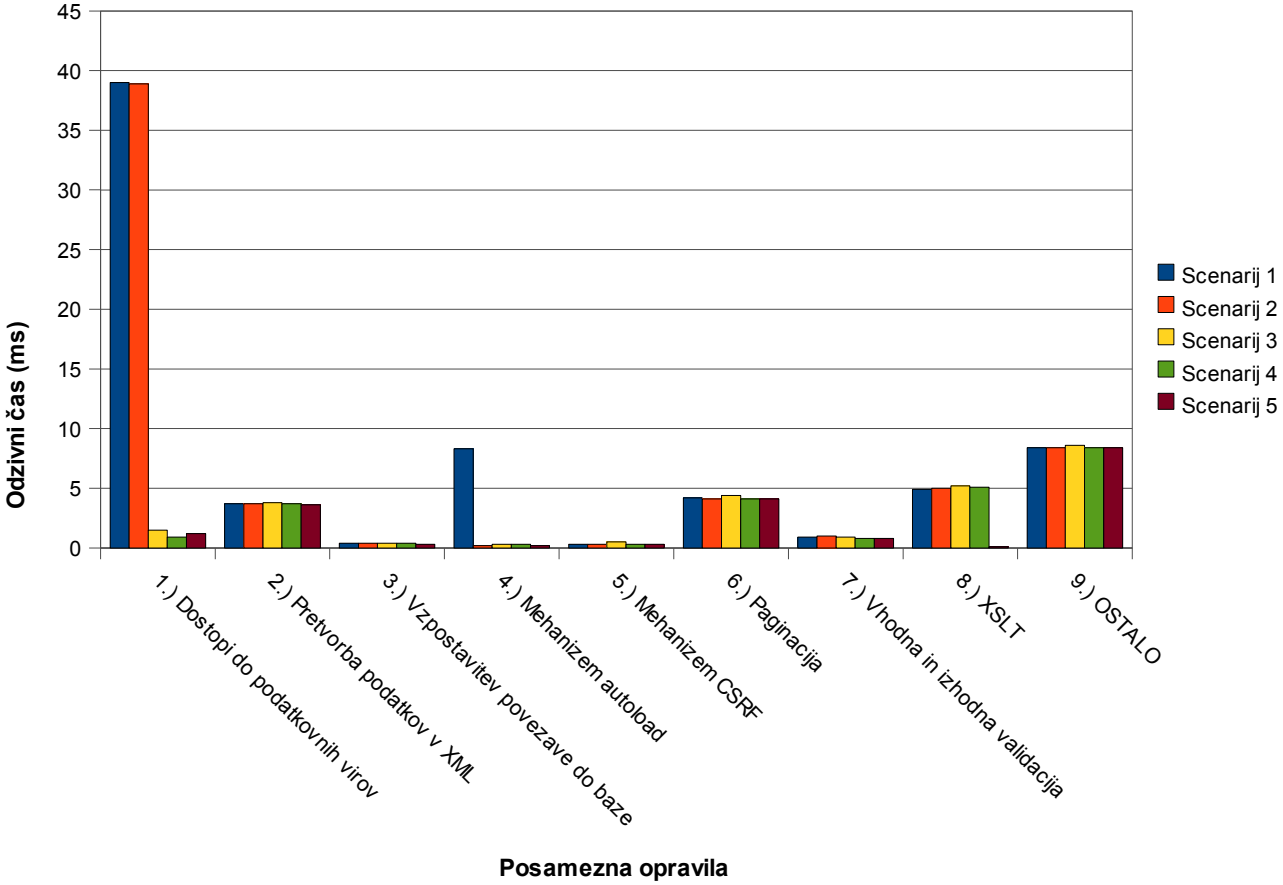
1.) Dostopi do podatkovnih virov (relacijske baze):	
dao_language:	0,4 ms
dao_certificate_user:	0,0 ms
dao_constant:	0,2 ms
dao_creditor:	0,0 ms
dao_debtor:	0,2 ms
dao_execution:	0,2 ms
dao_menu:	0,0 ms
dao_product_instance:	0,1 ms
dao_payment_debtor:	0,0 ms
dao_state_new:	0,1 ms
	SKUPAJ: **1,2 ms**
2.) Pretvorba podatkov v XML	
libField_Caption:	2 ms
libField_Dropdown:	0,2 ms
libField_Hidden:	0,8 ms
libField_Radio:	0,1 ms
libField_Submit:	0,2 ms
libField_Text:	0,1 ms
libContent*	0,2 ms
DOMDocument->saveXML:	SKUPAJ**: 3,6 ms**
3.) Vzpostavitev povezave do baze	
	SKUPAJ**: 0,3 ms**
4.) Mehanizem *autoload*	
	SKUPAJ**: 0,2 ms**
5.) Mehanizem CSRF	
	SKUPAJ**: 0,3 ms**
6.) Paginacija	
	SKUPAJ**: 4,1 ms**
7.) Vhodna in izhodna validacija	

	SKUPAJ: **0,8 ms**
8.) XSLT	
	SKUPAJ: **0,1 ms**
9.) OSTALO	
	SKUPAJ: **8,4 ms**

Tabela 5.5: Odzivni čas sistema pri enem uporabniku glede na opravila sistema (scenarij 5)

Skupen povprečni odzivni čas opravil sistema pri enem uporabniku je **19 ms**.[30] V tem primeru smo odpravili še eno ozko grlo, in sicer transformacijo XSLT. Odzivni čas tega opravila smo zmanjšali na skoraj 0 ms. Spodnji graf prikazuje še primerjavo s scenarijem 5.

Graf 5.10: Povprečno odzivni čas opravil sistema pri enem uporabniku glede na scenarij 1, 2, 3, 4, 5

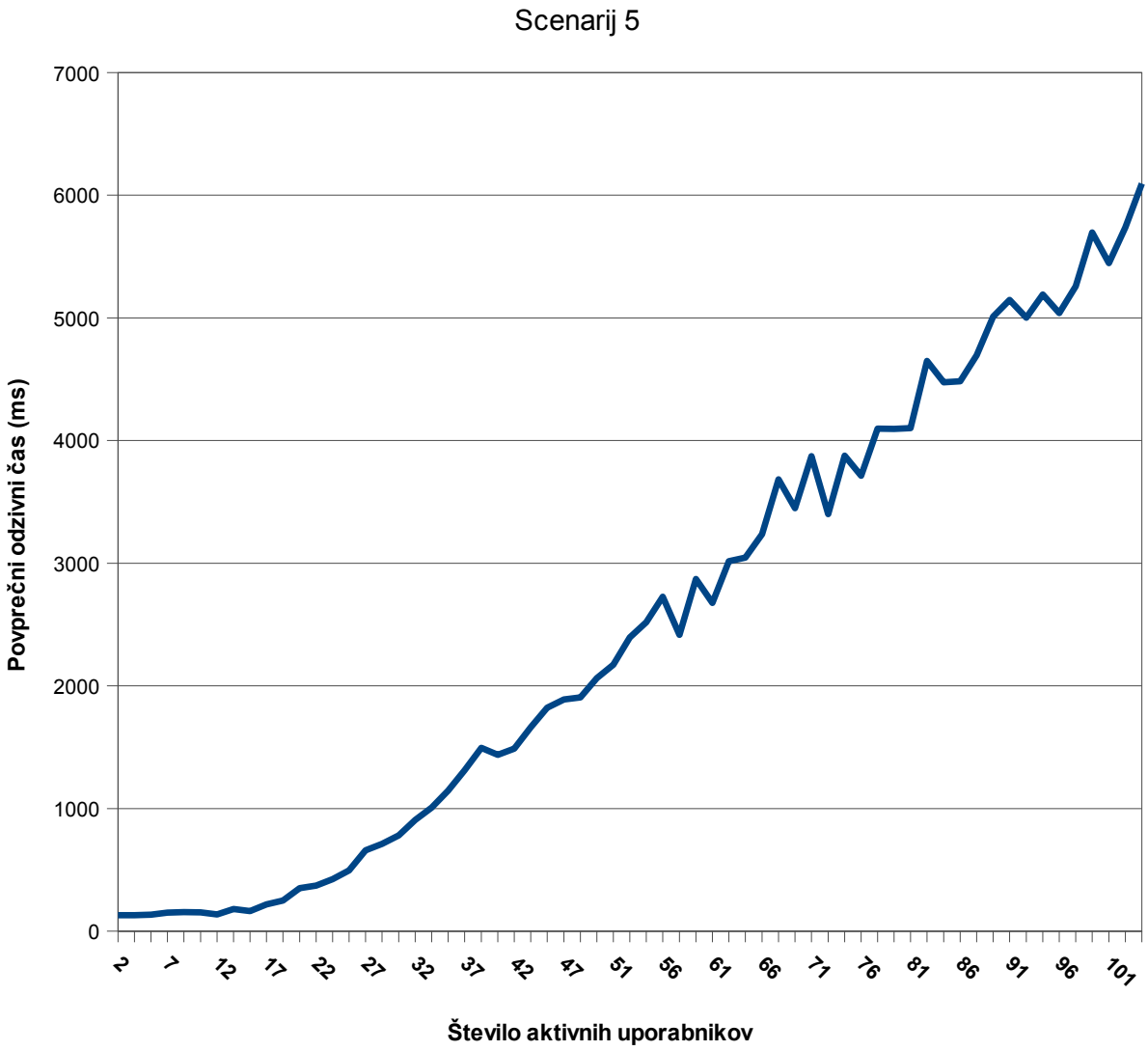


30 Standardna deviacija znaša 0,14.

5.11 Scenarij 5 – *ramp test*

Meritev ponovimo še z *ramp testom*. Rezultat je podan v spodnjem grafu.

Graf 5.11: Povprečni odzivni čas sistema glede na število aktivnih uporabnikov

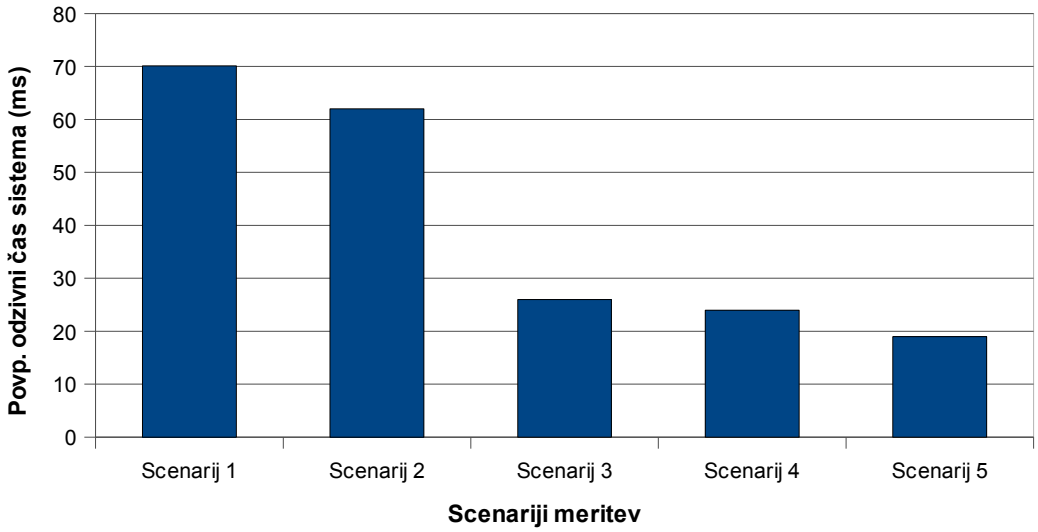


Z grafa je razvidno, da so se zadeve glede na scenarij 1 bistveno izboljšale. Povprečni odzivni čas začne rasti šele pri 13 aktivnih uporabnikih, pri čemer je rast počasna. Do nasičenja pride šele pri 104 aktivnih uporabnikih. Glede na definicijo bremena lahko trdimo naslednje: 104 aktivnih uporabnikov vsako sekundo pošlje zahtevek s povprečnim odzivnim časom 6095 ms. **Propustnost je zato enaka 104 zahtevkov/sekundo ali 6240 zahtevkov/minuto**.

6. SKLEPNE UGOTOVITVE

Z uporabo različnih tehnik predpomnjenja **smo torej povprečni odzivni čas sistema pri enem uporabniku zmanjšali s prvotnih 70,1 ms na 19 ms.** Spodnji graf prikazuje padanje odzivnega časa glede na posamezen scenarij:

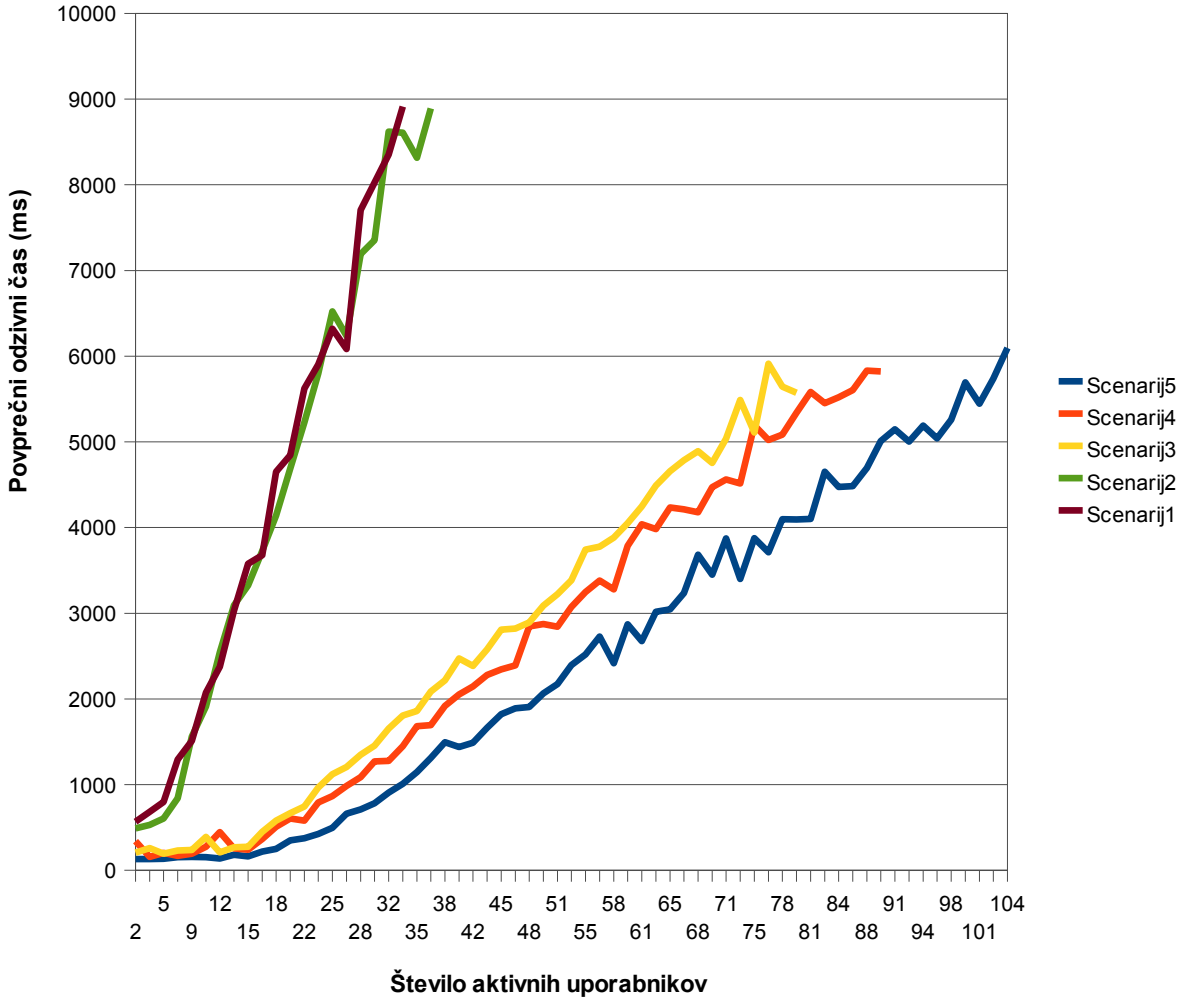
Graf 5.12: Padanje povp. odzivnega časa glede na uporabo različnih tehnik predpomnjenja



Pri sistemih s počasnim (ali bolj obremenjenem) diskovjem bi lahko pričakovali večje izboljšave pri scenariju 2. Zanimivo bi bilo videti, kako se bi se sistem obnašal v primeru virtualiziranega strežnika, kjer je diskovje, kot smo že prej ugotovili, bistveno bolj obremenjeno.

Prav tako bi pričakovali večjo razliko med scenarijem 3 in scenarijem 4, vendar lahko zaradi majhne količine predpomnjenih podatkov sklepamo, da so se pri predpomnjenju na disku uporabili vmesniki samega diska in vmesniki operacijskega sistema. Kljub temu pa je razlika med scenarijem 3 in 4 pomembna, saj scenarij 4 omogoča večjo propustnost (pri scenariju 3 je največ 89 aktivnih uporabnikov, pri scenariju 3 pa največ 80 aktivnih uporabnikov). V praksi je lahko teh 9 uporabnikov bistvenih za uspešnost izvajanja poslovne ideje. Tako Graf 5.13 prikazuje primerjavo propustnosti med scenariji.

Graf 5.13: Povprečni odzivni čas sistema glede na število aktivnih uporabnikov po posameznih scenarijih meritev



Uporaba tehnik predpomnjenja je bistveno povečala tudi propustnost, in sicer se je s 33 zahtevkov na sekundo povečala na 104 zahtevkov na sekundo. Prav tako nas je presenetil scenarij 5, kajti glede na perfomančni test je razlika med scenarijem 4 in scenarijem 5 zelo majhna, kar pa ne velja za *ramp test*, saj je predpomnjenje XSLT bistveno prispevalo k izboljšanju propustnosti.

Kot smo omenili že na začetku, bi bilo treba za celovito analizo aplikacije opraviti meritve za več različnih podstrani. Prav tako ne smemo pozabiti, da smo se osredotočili le na

predpomnjenje na strežniku, saj smo objekte *front-end* izključili iz meritev – tako bi predmet novih raziskav lahko bila tudi analiza predpomnjenja objektov *front-end*. Zanimivo bi bilo analizirati sistem tudi s kakšnim drugim spletnim strežnikom, vpliv datotečnega sistema v primeru velike količine datotek, namenjenih predpomnjenju, vpliv t. i. mehanizmov zaklepanja (*locking mechanisms*) na predpomnjenje, vpliv vzdrževanja velikosti kontejnerjev predpomnjenja ipd. Prav tako bi lahko preverili, kako se obnese predpomnjenje z datotekami DBM.

Za zaključek lahko povzamemo, da igra predpomnjenje tudi danes bistveno vlogo – kljub hitri strojni opremi in hitrim omrežjem. Glede na rezultate meritev ne gre le za majhne izboljšave, ampak tudi za to, ali bo določena aplikacija sploh delovala pri večjem številu uprabnikov. Zato se predpomnjenju pri spletnih aplikacijah (in tudi sicer) danes preprosto ni mogoče izogniti.

7. PRILOGE

7.1 Seznam slik

7.2 Seznam tabel

7.3 Seznam grafov

7.4 Izpis nastavitev sistema PHP (vključno z APC)

```
[PHP]
engine = On
zend.ze1_compatibility_mode = Off
short_open_tag = On
asp_tags = Off
precision    =  12
y2k_compliance = On
output_buffering = Off
zlib.output_compression = Off
implicit_flush = Off
unserialize_callback_func=
serialize_precision = 100
allow_call_time_pass_reference = On
safe_mode = Off
safe_mode_gid = Off
safe_mode_include_dir =
safe_mode_exec_dir =
```

```
safe_mode_allowed_env_vars = PHP_
safe_mode_protected_env_vars = LD_LIBRARY_PATH
disable_functions =
disable_classes =
expose_php = On
max_execution_time = 30     ; Maximum execution time of each script, in seconds
max_input_time = 60        ; Maximum amount of time each script may spend parsing request data
memory_limit = 16M      ; Maximum amount of memory a script may consume (128MB)
error_reporting = E_ALL & ~E_NOTICE
display_errors = Off
display_startup_errors = Off
log_errors = On
log_errors_max_len = 1024
ignore_repeated_errors = Off
ignore_repeated_source = Off
report_memleaks = On
track_errors = Off
error_log = /var/www/localhost/htdocs/phplog
variables_order = "EGPCS"
register_globals = Off
register_long_arrays = On
register_argc_argv = On
auto_globals_jit = On
post_max_size = 8M
magic_quotes_gpc = On
magic_quotes_runtime = Off
magic_quotes_sybase = Off
auto_prepend_file =
auto_append_file =
default_mimetype = "text/html"
include_path = ".:/usr/share/php5:/usr/share/php"
doc_root =
user_dir =
extension_dir = /usr/lib64/php5/lib/php/extensions/no-debug-non-zts-20060613
enable_dl = On
file_uploads = On
upload_max_filesize = 2M
allow_url_fopen = Off
```

```
allow_url_include = Off
default_socket_timeout = 60

[Syslog]
define_syslog_variables  = Off

[mail function]
SMTP = localhost
smtp_port = 25

[SQL]
sql.safe_mode = Off

[PostgresSQL]
pgsql.allow_persistent = On
pgsql.auto_reset_persistent = Off
pgsql.max_persistent = -1
pgsql.max_links = -1
pgsql.ignore_notice = 0
pgsql.log_notice = 0

[bcmath]
bcmath.scale = 0

[Session]
session.save_handler = files
session.use_cookies = 1
session.name = PHPSESSID
session.auto_start = 0
session.cookie_lifetime = 0
session.cookie_path = /
session.cookie_domain =
session.cookie_httponly =
session.serialize_handler = php
session.gc_probability = 1
session.gc_divisor     = 100
session.bug_compat_42 = 1
session.bug_compat_warn = 1
```

```
session.referer_check =
session.entropy_length = 0
session.entropy_file =
session.cache_limiter = nocache
session.cache_expire = 180
session.use_trans_sid = 0
session.hash_function = 0
session.hash_bits_per_character = 4
url_rewriter.tags = "a=href,area=href,frame=src,input=src,form=,fieldset="

[soap]
soap.wsdl_cache_enabled=1
soap.wsdl_cache_dir="/tmp"
soap.wsdl_cache_ttl=86400
```

7.5 Izpis nastavitev Apache spletnega strežnika

```
ServerRoot "/usr/lib64/apache2"
LoadModule actions_module modules/mod_actions.so
LoadModule alias_module modules/mod_alias.so
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule authn_alias_module modules/mod_authn_alias.so
LoadModule authn_anon_module modules/mod_authn_anon.so
LoadModule authn_dbm_module modules/mod_authn_dbm.so
LoadModule authn_default_module modules/mod_authn_default.so
LoadModule authn_file_module modules/mod_authn_file.so
LoadModule authz_dbm_module modules/mod_authz_dbm.so
LoadModule authz_default_module modules/mod_authz_default.so
LoadModule authz_groupfile_module modules/mod_authz_groupfile.so
LoadModule authz_host_module modules/mod_authz_host.so
LoadModule authz_owner_module modules/mod_authz_owner.so
LoadModule authz_user_module modules/mod_authz_user.so
LoadModule autoindex_module modules/mod_autoindex.so
<IfDefine CACHE>
LoadModule cache_module modules/mod_cache.so
</IfDefine>
LoadModule cgi_module modules/mod_cgi.so
<IfDefine DAV>
LoadModule dav_module modules/mod_dav.so
```

```
</IfDefine>
<IfDefine DAV>
LoadModule dav_fs_module modules/mod_dav_fs.so
</IfDefine>
<IfDefine DAV>
LoadModule dav_lock_module modules/mod_dav_lock.so
</IfDefine>
LoadModule deflate_module modules/mod_deflate.so
LoadModule dir_module modules/mod_dir.so
<IfDefine CACHE>
LoadModule disk_cache_module modules/mod_disk_cache.so
</IfDefine>
LoadModule env_module modules/mod_env.so
LoadModule expires_module modules/mod_expires.so
LoadModule ext_filter_module modules/mod_ext_filter.so
<IfDefine CACHE>
LoadModule file_cache_module modules/mod_file_cache.so
</IfDefine>
LoadModule filter_module modules/mod_filter.so
LoadModule headers_module modules/mod_headers.so
LoadModule include_module modules/mod_include.so
<IfDefine INFO>
LoadModule info_module modules/mod_info.so
</IfDefine>
LoadModule log_config_module modules/mod_log_config.so
LoadModule logio_module modules/mod_logio.so
<IfDefine CACHE>
LoadModule mem_cache_module modules/mod_mem_cache.so
</IfDefine>
LoadModule mime_module modules/mod_mime.so
LoadModule mime_magic_module modules/mod_mime_magic.so
LoadModule negotiation_module modules/mod_negotiation.so
LoadModule rewrite_module modules/mod_rewrite.so
LoadModule setenvif_module modules/mod_setenvif.so
LoadModule speling_module modules/mod_speling.so
<IfDefine SSL>
LoadModule ssl_module modules/mod_ssl.so
</IfDefine>
```

```
<IfDefine STATUS>
LoadModule status_module modules/mod_status.so
</IfDefine>
LoadModule unique_id_module modules/mod_unique_id.so
<IfDefine USERDIR>
LoadModule userdir_module modules/mod_userdir.so
</IfDefine>
LoadModule usertrack_module modules/mod_usertrack.so
LoadModule vhost_alias_module modules/mod_vhost_alias.so
User apache
Group apache
Include /etc/apache2/modules.d/*.conf
Include /etc/apache2/vhosts.d/*.conf

PidFile /var/run/apache2.pid
<IfModule mpm_prefork_module>
        StartServers            5
        MinSpareServers         5
        MaxSpareServers         10
        MaxClients              150
        MaxRequestsPerChild     10000
</IfModule>

<IfModule mpm_worker_module>
        StartServers            2
        MinSpareThreads         25
        MaxSpareThreads         75
        ThreadsPerChild         25
        MaxClients              150
        MaxRequestsPerChild     10000
</IfModule>

<IfModule mpm_event_module>
        StartServers            2
        MinSpareThreads         25
        MaxSpareThreads         75
        ThreadsPerChild         25
        MaxClients              150
```

```
        MaxRequestsPerChild     10000
</IfModule>

<IfModule mpm_peruser_module>
        MinSpareProcessors      2
        MinProcessors           2
        MaxProcessors           10
        MaxClients                      150
        MaxRequestsPerChild     1000
        ExpireTimeout           1800
        Multiplexer nobody nobody
        Processor apache apache
</IfModule>

<IfModule mpm_itk_module>
        StartServers            5
        MinSpareServers         5
        MaxSpareServers         10
        MaxClients              150
        MaxRequestsPerChild     10000
</IfModule>

Listen 80
Listen 8443
ServerName 127.0.0.1:80
NameVirtualHost *:80
<VirtualHost *:80>
        DocumentRoot "/var/www/localhost/htdocs/http"
</VirtualHost>

<VirtualHost *:8443>
        DocumentRoot "/var/www/localhost/htdocs/https"
        HostnameLookups off
        <Directory "/var/www/localhost/htdocs/https">
                Options Indexes FollowSymLinks
                AllowOverride None
                Allow From All
        </Directory>
```

```
</VirtualHost>
```

7.6 Izpis nastavitev relacijske baze PostgreSql

```
listen_addresses = '*'                  # what IP interface(s) to listen on;
max_connections = 100
shared_buffers = 1000                   # min 16, at least max_connections*2, 8KB each
#work_mem = 1024                        # min 64, size in KB
#maintenance_work_mem = 16384# min 1024, size in KB
#max_stack_depth = 2048                 # min 100, size in KB
log_destination = 'stderr'              # Valid values are combinations of stderr,
lc_messages = 'C'                       # locale for system error message strings
lc_monetary = 'C'                       # locale for monetary formatting
lc_numeric = 'C'                        # locale for number formatting
lc_time = 'C'                           # locale for time formatting
```

7.7 CD s diplomsko nalogo skupaj z nastavitvenimi datotekami

8. Viri

[1] (2009) Apache HTTP Server Version 2.0. Dostopno na:
http://httpd.apache.org/docs/2.0/mpm.html

[2] (2009) Apache 2.0 on Unix systems. Dostopno na:
http://si.php.net/manual/en/install.unix.apache2.php

[3] (2009) Apache MPM Common Directives. Dostopno na: http://httpd.apache.org/docs/2.0/mod/mpm_common.html#maxrequestsperchild

[4] (2009) Apache MPM Prefork. Dostopno na:
http://httpd.apache.org/docs/2.0/mod/prefork.html

[5] (2009) Apache Performance Tuning. Dostopno na:
http://httpd.apache.org/docs/2.2/misc/perf-tuning.html

[6] (2009) Best practices for using server virtualization in your storage environment. Dostopno na: http://searchstorage.techtarget.com/tip/0,289483,sid5_gci1359389,00.html

[7] (2002) File System Limitations RedHat 7.3, ext3. Dostopna na:
http://answers.google.com/answers/threadview?id=122241

[8] Freeman et al., *Head First Design Patterns*, O'Reilly, 2004, Pogl. 12.

[9] S. Golemon, *Extending and Embedding PHP*, Sams, 2006, Pogl. 1.

[10] S. Golemon, *Extending and Embedding PHP*, Sams, 2006, Pogl. 20.

[11] A. Gutmans, S.S. Bakken, D. Rethans, *PHP 5 Power Programming*, Prentice Hall, str. 452, 2005

[12] (2007) High Performance Web Sites: The Importance of Front-End Performance. Dostopno na: http://developer.yahoo.net/blog/archives/2007/03/high_performanc.html

[13] (2009) Memcached. Dostopno na: http://www.danga.com/memcached/

[14] J.T.J. Midgley, *Principles of HTTP benchmarking*, Zeus Tehnology, 2002, str. 7.

[15] (2009) Persistent Database Connections. Dostopno na: http://si2.php.net/manual/en/features.persistent-connections.php

[16] (2009) Poul-Henning Kamp, Unix guru at large. Dostopno na: http://people.freebsd.org/~phk/

[17] G. Schlossnagle, "Profiling PHP Applications", *PHP Architect*, št. 2, zv. III, str.41-50, 2004

[18] G. Schlossnagle, *Advanced PHP Programming*, Sams, 2004, Pogl 10.

[19] B. M. Shire, "apc@facebook", *PHP Works Atlanta*, Atlanta, 2007

[20] (2009) Tuning hints, Iagora. Dostopno na: http://www.iagora.com/about/software/lingerd/TUNING.txt

[21] (2009) The Apache Software Foundation. Dostopno na: http://httpd.apache.org/

[22] Webserver Stress Tool 7, *User manual*, Paessler GmbH, 2005, Pogl. 2

[23] (2009) What is XSL? Dostopno na: http://www.w3.org/Style/XSL/WhatIsXSL.html

[24] N. Zimic, *Temelji zmogljivosti računalniških sistemov*, Ljubljana: Fakulteta za računalništvo in informatiko, 2006