SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2870

ANALIZA UČINKOVITOSTI IZVOĐENJA PROGRAMA U OKOLINI PYPY

Mihaela Svetec

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2870

ANALIZA UČINKOVITOSTI IZVOĐENJA PROGRAMA U OKOLINI PYPY

Mihaela Svetec

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 2870

Pristupnica:	**Mihaela Svetec (0036500791)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	izv. prof. dr. sc. Klemo Vladimir

Zadatak: **Analiza učinkovitosti izvođenja programa u okolini PyPy**

Opis zadatka:

Proučiti i opisati sustav PyPy. Posebno opisati arhitekturu sustava s naglaskom na podsustav za dinamičko prevođenje u trenutku izvođenja kompilatora (Just-in-Time). Usporediti PyPy sa standardnom Python implementacijom CPython. Implementirati skup programa različitih obilježja u programskom jeziku Python te prikupiti rezultate izvođenja programa u okolini s PyPy i CPython interpreterom. Provesti analizu učinkovitosti izvođenja ovisno o obilježjima programa i okolini. Opisati rezultate provedene analize te navesti zaključke.

Rok za predaju rada: 27. lipnja 2022.

Zahvaljujem se svom mentoru, izv. prof. dr. sc. Klemi Vladimiru, na pruženom strpljenju i vodstvu tijekom izrade ovog diplomskog rada.

Veliko hvala mojim prijateljima i dečku koji su put do završnog koraka studiranja učinili lakšim i zabavnijim.

Posebno se zahvaljujem svojim roditeljima na kontinuiranoj podršci i ohrabrenju koje su mi pružili tijekom svih mojih godina studija. Bez njih ovo postignuće ne bi bilo moguće.

Sadržaj

Uvod

Učinkovita komunikacija pomaže u izgradnji kvalitetnih međuljudskih odnosa, poboljšava produktivnost i štedi vrijeme, a podržana je postojanjem jezika. Modernizacija društva i tehnološki napredak uveli su nove izazove u proces razmjene informacija. Pojavilo se pitanje: kako omogućiti prijenos znanja između osobe i računala te je li moguće koristiti jezike koji su poznati? Stvoreni su programski jezici, kojih danas postoji mnoštvo, a među njima se istaknuo Python.

Python je programski jezik opće namjene, što znači da se može koristiti za razvoj programa različitih funkcionalnosti. Veliku popularnost stekao je u posljednjih nekoliko godina, a od preostalih jezika ponajviše se ističe svojom jednostavnošću upotrebe i širokim spektrom mogućnosti. Mnoštvo prednosti često padne u drugi plan zbog ograničenja na brzinu izvođenja. Ograničenje proizlazi iz činjenice da je Python dinamički programski jezik – tijekom izvršavanja programa izvorni kôd se kompilira u međukôd (eng. *byte code*), koji se zatim tumači koristeći Python prevoditelj (eng. *interpreter*). U usporedbi sa statičkim programskim jezicima, koji provode kompilaciju prije izvođenja programa i na razinu binarnog strojnog kôda, Python implementacija će programe određenih karakteristika izvoditi sporije. Opisani nedostatak potaknuo je razvoj različitih Python prevoditelja s ciljem optimizacije performansi. Standardna i najraširenija Python implementacija naziva se CPython, ali pozornost se sve više počela usmjeravati i na rješenje PyPy. Projekt PyPy rezultirao je prevoditeljem pisanim u razvojnom okviru RPython koji koristi prateći kompilator za dinamičko prevođenje u trenutku izvođenja (eng. *tracing just-in-time compiler*; u nastavku prateći JIT kompilator). Prateći JIT kompilator osnovna je komponenta zaslužna za učinkovitije izvođenje Python programa u usporedbi s CPython implementacijom. Međutim, iako je PyPy često predstavljen kao brži prevoditelj, istraživanje je pokazalo kako to nije uvijek slučaj.

Naglasak diplomskog rada stavljen je na arhitekturu sustava PyPy, njegove komponente i funkcionalnosti. Posebna pažnja pridaje se pratećem JIT kompilatoru te je dan detaljan pregled njegove arhitekture. Kako bi motivacija za izgradnju PyPy sustava dobila puni smisao, opisane su značajke standardne Python implementacije i njena izvedba. Nadalje, u okviru rada provedena je analiza učinkovitosti izvođenja u okolinama PyPy i CPython koristeći

reprezentativni skup ispitnih programa. U konačnici, na temelju prikupljenih rezultata, provedena je usporedba i doneseni su zaključci o prikladnosti upotrebe svakog od navedenih Python prevoditelja.

1 Programski jezik Python

Python je programski jezik visoke razine i opće namjene. Nastao je 1990. godine te se vrlo brzo istaknuo i postao jednim od najkorištenijih programskih jezika. Popularnost je stekao zahvaljujući kvalitetnom dizajnu i mnoštvu ugrađenih alata koji omogućavaju brz i učinkoviti razvoj programskih rješenja. Zbog spoja jednostavnosti, stabilnosti i agilnosti, Python je široko primjenjiv u područjima analize podataka, strojnog učenja, razvoja web aplikacija, automatizacije, skriptnog programiranja itd. U nastavku je dan pregled osnovnih značajki programskog jezika, nakon čega je naglasak stavljen na opis standardne Python implementacije i načina izvršavanja programa.

1.1 Osnovne značajke

Python paradigma usmjerena je na razvoj programskog jezika koji je jednostavan, snažan i fleksibilan.

- **Jednostavnost**: Jasna i čitljiva sintaksa čini učenje programskog jezika Python i snalaženje u njegovom kôdu znatno lakšim.
- **Bogatstvo ugrađenih funkcionalnosti**: Dostupnost mnoštva ugrađenih knjižnica i rješenja trećih strana omogućuje učinkovit razvoj programskih rješenja u svim područjima primjene.
- **Prenosivost**: Python programe moguće je izvoditi na različitim platformama – nije potrebno raditi izmjene nad izvornim kôdom s ciljem uspješnog izvršavanja na drugoj platformi.
- **Proširivost i integracija**: Izvorni kôd moguće je proširiti implementacijama izvedenim u drugim programskim jezicima (najčešće C), čime se obogaćuje skup dostupnih funkcionalnosti i postiže učinkovitije izvođenje. Također, Python program može se lako integrirati u okoline koje sadrže komponente pisane drugim programskim jezicima.
- **Poticanje produktivnosti**: Python kôd često je kraći u usporedbi s jezicima kao što su C, Java,… Nadalje, postupak izvršavanja programa ne zahtijeva prethodnu kompilaciju u strojni kôd. Istaknuta obilježja potiču učinkovitije programiranje i brži razvojni proces.

- **Napredne tehnike**: Python koristi napredne alate poput objektno-orijentirane paradigme, automatskog upravljanja memorijom, ugrađenih tipova i funkcija, podrške ugrađenih knjižnica i modula trećih strana itd.
- **Besplatan**: Korištenje programskog jezika i njegovih implementacija je besplatno te u obliku otvorenog kôda (eng. *open-source*). Prednost jednostavnog pristupa izvornom kôdu i ostalim komponentama je mogućnost raznih modifikacija radi prilagodbe vlastitih rješenja.
- **Aktivno podržan**: Važnu ulogu u razvoju Python-a ima velika zajednica koja aktivno radi na unaprjeđenju jezika i njegovih implementacija.

1.2 Standardna implementacija – CPython

CPython je Python prevoditelj pisan u programskom jeziku C, a predstavlja najpoznatiju i najkorišteniju Python implementaciju. Istovremeno se definira kao prevoditelj i kompilator – kompilira izvorni kôd programa u međukôdove koje zatim prevodi u strojni kôd. Postupak izvršavanja Python programa sastoji se od većeg broja koraka i složeniji je od navedenog. Cilj je stvoriti općenitu sliku o načinu izvođenja CPython prevoditelja sažetim opisom osnovnih koraka (detalji nisu u opsegu ovog rada).

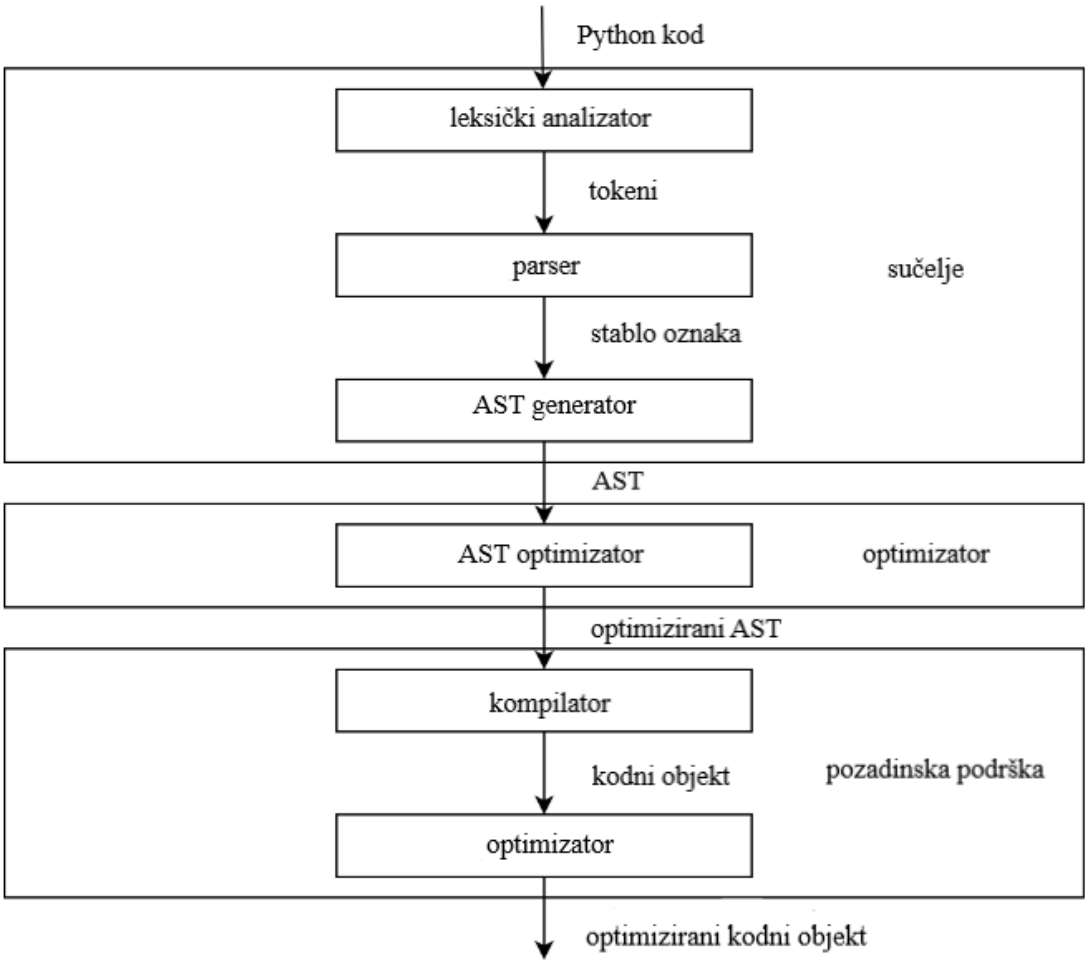
1.2.1 Proces izvođenja programa

Proces izvođenja programa s CPython prevoditeljem može se podijeliti u tri osnovna koraka:

1. inicijalizacija
2. kompiliranje
3. prevođenje

U prvom koraku provodi se inicijalizacija podatkovnih struktura potrebnih za pokretanje Python programa. Postupak inicijalizacije uključuje pripremu ugrađenih tipova podataka, konfiguraciju i učitavanje ugrađenih funkcionalnosti, pripremu sustava za učitavanje knjižnica i sl.

Postupak kompiliranja izvorni kôd Python programa pretvara u novu reprezentaciju programa koja se može učinkovito izvoditi na virtualnom stroju. Nova reprezentacija programa opisana je međukôdovima – nizovima operacija koji opisuju strojne instrukcije i pripadne argumente. Slikom 1.1 [17] prikazana je arhitektura komponente kompilatora. CPython kompilator sadrži dvije glavne komponente: sučelje (eng. *frontend*) i podršku (eng. *backend*).



Slika 1.1 Arhitektura kompilatora CPython prevoditelja [17]

Osnovna uloga sučelja je da iz izvornog kôda Python programa stvori stablo apstraktne sintakse (eng. *Abstract Syntax Tree*, u nastavku AST). AST opisuje izvorni kôd podatkovnom strukturom visoke razine apstrakcije kojom se može lako upravljati. Postupak generiranja AST strukture započinje označavanjem programskog kôda (eng. *tokenization*). Nadalje, rezultat označavanja se obrađuje (eng. *parse*) prikladnim tehnikama te pretvara u AST. Prije nego se proces nastavi komponentom podrške, provodi se dodatna optimizacija AST strukture metodom

otkrivanja konstanti (eng. *constant folding*). Virtualni stroj CPython prevoditelja ne razumije AST strukture i na temelju njih ne može izvršiti program. Podrška kompilatora zaslužna je za pretvorbu AST strukture u skup kôdnih objekata koje može izvršiti CPython prevoditelj. Kôdni objekt pohranjuje dio programa u obliku međukôda, uz ostale ključne informacije poput korištenih varijabli i sl. Postupak izgradnje kôdnih objekata uključuje oblikovanje tablice s informacijama o objektima, izgradnju grafova toka kontrole (eng. *Control Flow Graph*, u nastavku CFG) i optimizaciju međukôdova. Strukture CFG reprezentacija su izvornog programa i prikazuju moguće putanje koje program može slijediti tijekom izvršavanja. Konačno, korak kompiliranja stvara datoteke s kompiliranim međukôd instrukcijama koje se mogu koristiti u sljedećem izvršavanju programa (ako u međuvremenu nije bilo izmjena u izvornom kôdu).

U posljednjem koraku provodi se izvršavanje međukôda, pohranjenog u stvorenim kôdnim objektima, koristeći virtualni stroj CPython prevoditelja. Funkcionalnost virtualnog stroja implementirana je glavnom petljom čije se iteracije nastavljaju sve dok postoje instrukcije za izvršavanje.

1.2.2 Snage i slabosti

Kako je CPython standardni prevoditelj programskog jezika Python, njegova osnovna snaga proizlazi iz bogatog skupa ugrađenih funkcionalnosti. Podržane funkcionalnosti redovito su održavane i osvježavane novim rješenjima. Nadalje, arhitektura CPython prevoditelja i automatsko upravljanje memorijom olakšavaju implementaciju programskih rješenja te omogućavaju učinkovitiji razvojni proces. Konačno, instalacijski postupak CPython prevoditelja je vrlo jednostavan što dodatno potiče njegovu upotrebu (većina Linux operacijskih sustava dolazi s prethodno instaliranim Python prevoditeljem).

Iako struktura CPython prevoditelja nudi mnoge prednosti, prisutan je negativan utjecaj na učinkovitost prilikom izvršavanja Python programa. CPython implementacija često usporava izvođenje programa, što je posljedica obilježja interpretiranih, dinamičkih jezika. Također, neizbježna je i visoka potrošnja memorije. Odgovor na probleme standardne Python implementacije pokušao se pronaći razvojem PyPy paradigme.

2 Projekt PyPy

Pojmom PyPy izvorno je opisana okolina za implementaciju dinamičkih jezika [6]. Proces implementacije dinamičkih jezika započinje oblikovanjem odgovarajućeg prevoditelja koristeći programski jezik RPython. U sljedećem koraku primjenjuje se niz operacija, tzv. lanac alata za prevođenje, koje na temelju RPython kôda kreiraju skup C programa i *makefile* datoteku. Konačno, pokretanjem *makefile* skripte stvara se binarna izvršna datoteka, tj. prevoditelj za ciljano okruženje. Razvojna okolina nudi bogati skup optimizacijskih koraka te omogućava prilagodbu strukture ovisno o zahtjevima i potrebama. Snaga PyPy okoline sadržana je u mogućnosti upotrebe pratećeg JIT kompilatora koji se, na relativno jednostavan način, može dodati u odabrani prevoditelj. Detalji izvedbe i način korištenja pratećeg JIT kompilatora bit će opisani kasnije (poglavlje 2.2.2). PyPy prevoditelj razvijen je u opisanoj okolini, a predstavlja implementaciju programskog jezika Python koja koristi prateći JIT kompilator i postepeno, generacijsko upravljanje memorijom.

U nastavku poglavlja dan je detaljniji opis razvojnog procesa PyPy prevoditelja, uključujući jezik RPython i lanac alata za prevođenje (poglavlje 2.1). Nadalje, opisane su komponente PyPy prevoditelja, pri čemu je naglasak stavljen na obilježja i način rada pratećeg JIT kompilatora (poglavlje 2.2). Konačno, provedena je usporedba CPython i PyPy prevoditelja iznošenjem razlika i sličnosti u njihovoj izvedbi (poglavlje 2.3).

2.1 Izgradnja PyPy prevoditelja

2.1.1 Oblikovanje prevoditelja – RPython

Postupak izgradnje izvršne datoteke PyPy prevoditelja započinje pisanjem prevoditelja u programskom jeziku RPython. RPython (skraćenica, eng. *Restricted Python*) definiran je kao podskup jezika Python. Jezici imaju jednaku sintaksu, međutim RPython uvodi dodatna ograničenja na korištenje varijabli, objekata, ugrađenih funkcija i tipova podataka, petlji, iznimki i sl. Zbog svoje visoke razine i prilagodljivosti, RPython omogućava učinkovitiju implementaciju i eksperimentiranje s obilježjima prevoditelja.

U sljedećem koraku RPython program se kompilira prolazeći kroz niz alata za prevođenje. Ovaj postupak definiran je mnoštvom Python modula, a detalji izvođenja prikazani su u nastavku.

2.1.2 Lanac alata za prevođenje RPython programa

Osnovna uloga alata za prevođenje jest pretvoriti RPython program u učinkoviti Python virtualni stroj umetanjem odgovarajućih elemenata niže razine apstrakcije. Postupak prevođenja [14] sastoji se od sljedećih koraka:

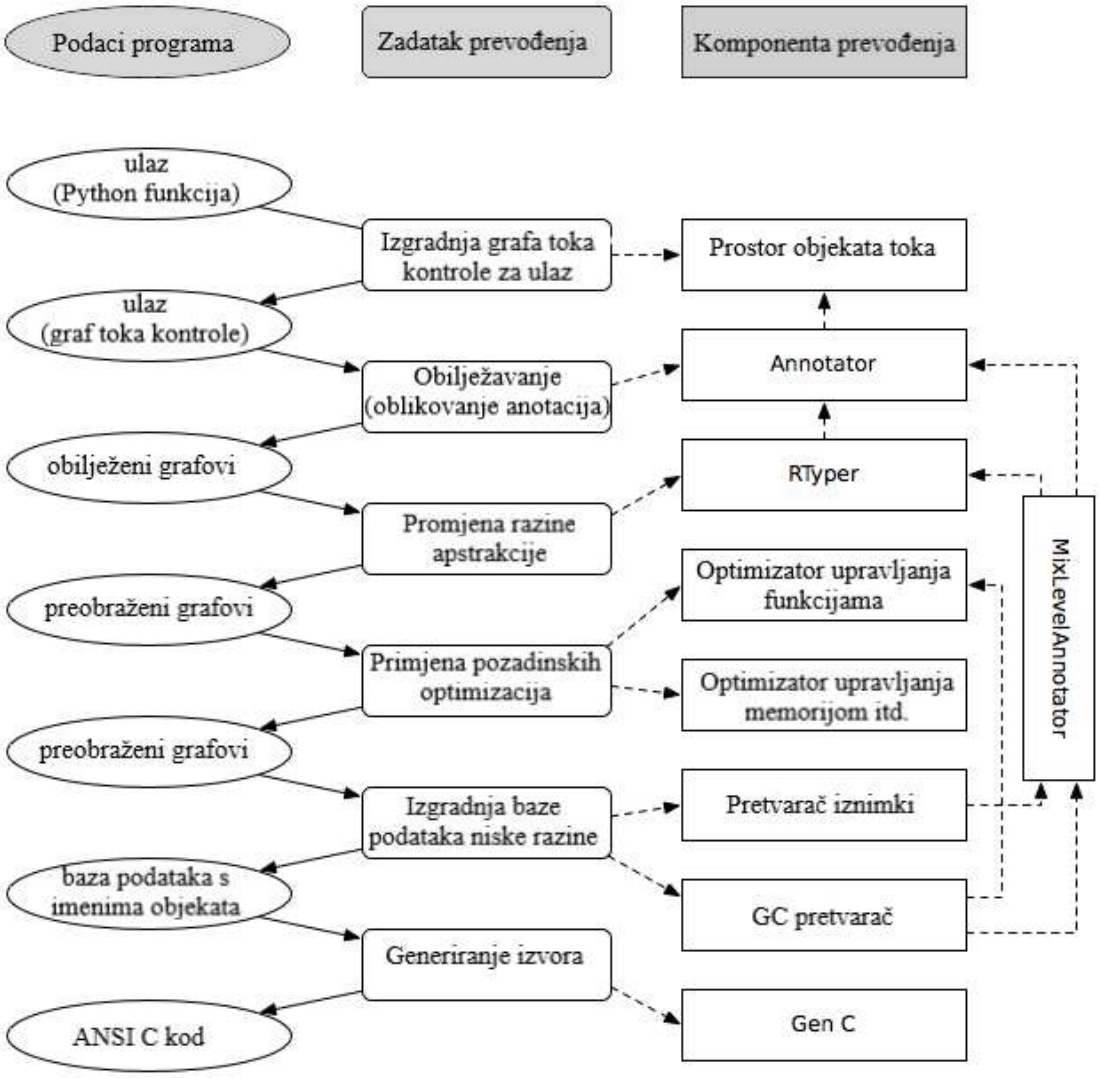
1. Učitavanje RPython programa.
2. Izgradnja CFG struktura na temelju Python objekata kôda.
3. Obilježavanje varijabli CFG struktura. Otkrivanje svih tipova podataka koje varijabla može poprimiti tijekom izvođenja programa, uz modeliranje grafova po potrebi.
4. Pretvaranje operacija označenih CFG struktura, prikazanih visokom razinom apstrakcije, u operacije bliže razini apstrakcije ciljane platforme (C).
5. Dodatne optimizacije programske izvedbe (eng. *backend optimizations*) s ciljem bržeg izvršavanja rezultantnog programa.
6. Priprema CFG struktura za proces kreiranja izvora – preslikavanje imena varijabli i funkcija, primjena transformacija za dodavanje eksplicitnog upravljanja iznimkama i memorijom.
7. Kreiranje izvornih C datoteka na temelju konačnih verzija CFG struktura.
8. Kompiliranje dobivenih C datoteka u izvršni program prevoditelja.

Izgradnja CFG struktura

Komponente postupka prevođenja RPython programa nikada ne vide izvorni kôd programa već djeluju nad Python kôdnim objektima. Graditelj grafova toka (eng. *flow graph builder*) koristi apstraktnu interpretaciju nad Python objektima kako bi izgradio strukture CFG (jedan graf odgovara jednoj funkciji programa). CFG predstavlja temeljnu podatkovnu strukturu korištenu u gotovo svakom koraku postupka prevođenja – izvorni program je na taj način u obliku koji je

prikladan za primjenu tehnika korištenih u nastavku, što omogućava učinkovitiju analizu i kvalitetnije rezultate.

Apstraktna interpretacija provodi se nad međukôdom funkcija za koje se bilježe sve izvršene operacije nad Python objektima i oblikuju strukture osnovnog bloka. Svaki osnovni blok sadrži listu zabilježenih operacija, listu ulaznih varijabli i listu veza s drugim osnovnim blokovima. Zabilježena operacija opisana je imenom operacije, listom argumenata (varijable i konstante) i varijablom u kojoj je pohranjen rezultat operacije. Konačno, oblikovan je kontejner s referencom na skup povezanih osnovnih blokova, čime je stvoren jedan graf toka kontrole.



Slika 2.1 Postupak prevođenja RPython programa i kompiliranja prevoditelja [14]

Obilježavanje varijabli

Nad stvorenim CFG strukturama poziva se komponenta opažača (eng. *annotator*). Glavna uloga opažača je utvrditi sve moguće vrijednosti koje varijabla CFG strukture može poprimiti tijekom izvođenja, analizirajući pri tome cjelokupni program, tj. svaki CFG. Rezultat obilježavanja varijabli može se opisati oblikom rječnika – ključ predstavlja varijablu, a vrijednost skup Python objekata koje varijabla može sadržavati (npr. cijeli broj, znakovni niz, lista, rječnik). Zbog dinamičke strukture grafova i povezanosti osnovnih blokova, u postupku obilježavanja varijabli često dolazi do nepredvidljivih kretanja po blokovima te unošenja promjena kako bi se postigao konstantan i potpuni rezultat.

Pretvaranje u nižu razinu apstrakcije

Obilježene CFG strukture opisane su visokom razinom apstrakcije, uključujući operacije i vrijednosti varijabli. Kako bi prevođenje u izvršni kôd bilo učinkovitije i jednostavnije, potrebno je zamijeniti visoku razinu s modelom niske razine apstrakcije (odgovarajuće za jezik C). Korak izmjene CFG struktura provodi komponenta RPython pisača (eng. *RPython Typer*, u nastavku RTyper). RTyper prolazi kroz svaki osnovni blok jednom te, pomoću vlastite implementacije preslikavanje, zamjenjuje vrijednosti varijabli s odgovarajućim tipovima podataka niske razine, a operacije s jednom ili više operacija niske razine.

Dodatne optimizacije

Optimizacija pojedinih funkcionalnosti provodi se s ciljem bržeg izvršavanja konačne verzije programa. Istaknuta su dva optimizacijska koraka: kopiranje funkcija heurističkom metodom (eng. *function inlining*) i uklanjanje nepotrebnog zauzeća memorije. Ovisno o zahtjevima i potrebama, moguće je proširiti optimizacijski skup s vlastitim implementacijama za učinkovitije izvođenje.

Preslikavanje i transformacija grafova

Prethodno generiranju izvornog kôda izvršava se konačno donošenje odluka važnih za implementaciju dinamičkog jezika. U okviru ovog postupka obavljaju se sljedeće aktivnosti:

- Definiranje eksplicitnog načina upravljanja iznimkama.
- Definiranje eksplicitnih operacija za upravljanje memorijom.

- Dodjela imena koje će funkcije i varijable imati u konačnom programu, koristeći „bazu podataka niske razine“ (eng. „*low-level database*“).

Kreiranje C datoteka i izvršnog programa prevoditelja

Nakon što su prethodno opisani koraci izvršeni, poziva se generator C kôda (Gen C) nad konačnim verzijama CFG struktura. Rezultat izvođenja generatora uključuje skup C programa i *makefile* skriptu kojom se oblikuje binarna izvršna datoteka PyPy prevoditelja.

Slika 2.1 nudi vizualni prikaz tijeka izvršavanja opisanog postupka za prevođenje RPython programa. Vrijedno je istaknuti postojeću ovisnost komponenti. Optimizacijski koraci pojedinih komponenti utječu na korištene strukture, što rezultira potrebom za ponovnim prolaskom kroz prethodne korake radi prikladnog oblikovanja unesenih izmjena. Primjerice, komponenta RTyper uvodi pozive funkcionalnosti niže razine koje je potrebno pravilno obilježiti, zbog čega se ponovno aktivira komponenta opažača.

2.2 Arhitektura PyPy prevoditelja

Postupkom opisanim u prethodnom poglavlju stvoren je PyPy prevoditelj. PyPy [13] potpuno implementira programski jezik Python i predstavlja učinkovitu zamjenu za standardno rješenje, CPython. Izgradnja PyPy prevoditelja motivirana je problemom sporijeg izvršavanja s kojim dolazi CPython – osnovni cilj bio je stvoriti implementaciju koja brzinom izvođenja, uz mogućnosti korištenja bogatog skupa Python knjižnica, može steći prednost nad CPython prevoditeljem. Za postizanje ubrzanja korišten je prateći JIT kompilator u kombinaciji s metodom generacijskog upravljanja memorijom s postepenim označavanjem (eng. *incremental generational garbage collection*). Obje komponente razvijene su u kontekstu okoline PyPy, a ugrađene su u prevoditelj postupkom prevođenja, tj. prilikom njegove izgradnje.

U nastavku je dan pregled osnovnih komponenti PyPy prevoditelja. Nadalje, prateći JIT kompilator ključan je element PyPy okoline te je detaljno opisana njegova arhitektura i utjecaj na izvršavanje korisničkih programa. Predstavljanje arhitekture PyPy prevoditelja zaokruženo je opisom metode upravljanja memorijom.

2.2.1 Osnovne komponente prevoditelja

PyPy prevoditelj oblikovan je koristeći tri komponente [13]:

1. kompilator međukôda (eng. *bytecode compiler*)
2. prevoditelj međukôda (eng. *bytecode interpreter*)
3. standardni prostor objekata (eng. *standard object space*)

Kompilator djeluje nad izvornim kôdom korisničkog Python programa [1] – iz izvornog kôda generira skup Python kôdnih objekata. Kreirani objekti su precizno oblikovane strukture koje reprezentiraju izvorni program, a njihov glavni sadržaj je međukôd. Koraci kompilatora uključuju označavanje i parsiranje programskog kôda, izgradnju i optimizaciju AST struktura te generiranje međukôdova i njihovo oblikovanje u strukture objekata.

Prevoditelj je sličan virtualnom stroju implementacije CPython. Njegova osnovna uloga je upravljanje tijekom izvođenja programa koristeći objekte kôda i strukturu stoga. Objekti se postavljaju i uzimaju sa stoga ovisno o zahtjevima programa, međutim prevoditelj ne zna izvoditi operacije nad objektima te taj zadatak dodjeljuje prostoru objekata.

Prostor objekata sadrži implementacije Python objekata i definira operacije koje se nad njima mogu izvršavati. Jedna od inačica prostora objekata je standardni prostor objekata, a sadrži implementacije ugrađenih Python objekata i funkcija na nižoj razini apstrakcije po uzoru na CPython. Standardni prostor objekata i prevoditelj međukôda čine temelj strukture PyPy prevoditelja.

2.2.2 Implementacija i aktivnosti pratećeg JIT kompilatora

Općeniti pogled

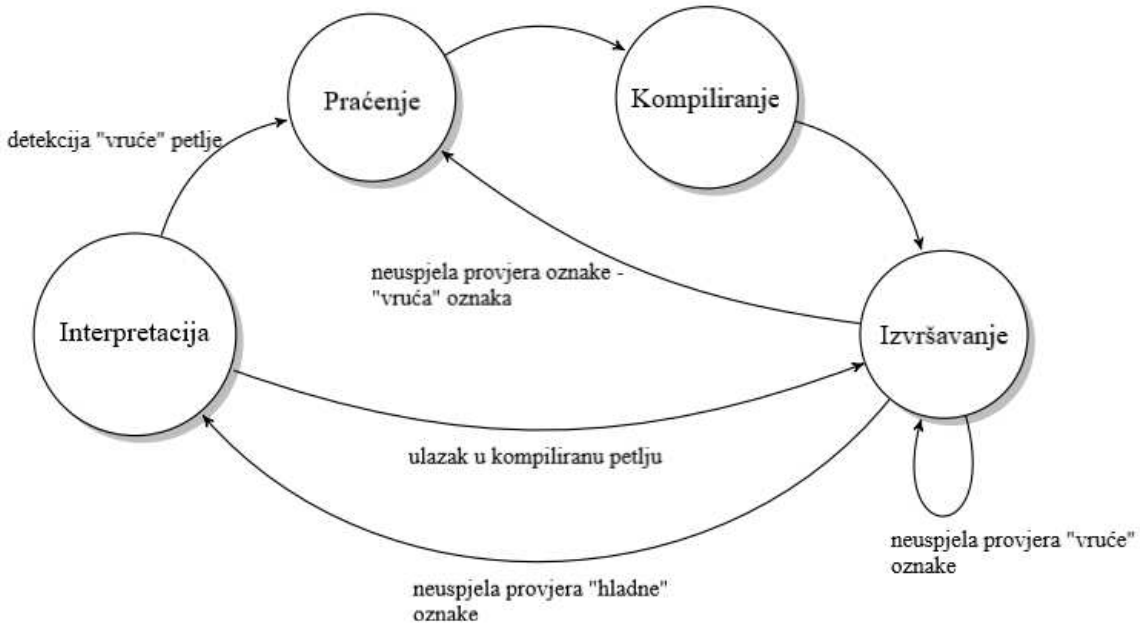
Jedno od istaknutih obilježja okoline PyPy je što omogućuje ugradnju pratećeg JIT kompilatora uz izvršni program prevoditelja. Prateći JIT kompilatori predstavljaju naprednu tehniku

[1] Pojam korisničkog programa odnosi se na Python program pisan od strane programera koji koristi prevoditelj PyPy za izvršavanje implementiranih funkcionalnosti.

optimizacije i unaprjeđenja implementacija dinamičkih jezika, a njihova pravilna upotreba rezultira ubrzanjem procesa izvršavanja programa. Funkcionalnosti pratećih JIT kompilatora zasnivaju se na sljedećim pretpostavkama [3]:

- programi tijekom izvršavanja većinu vremena provedu u petljama
- nekoliko iteracija iste petlje će vrlo vjerojatno rezultirati sličnim putanjama kroz programski kôd

Prevoditelji s ugrađenim pratećim JIT kompilatorom dijele proces izvršavanja programa u nekoliko faza (slika 2.2 [7]).



Slika 2.2 Faze prevoditelja s pratećim JIT kompilatorom [7]

Pokretanjem programa započinje korak standardne interpretacije (eng. *interpretation*) uz koji se provodi dodatno profiliranje radi utvrđivanja često izvođenih petlji, tzv. „vrućih“ petlji. Detekcija „vrućih“ petlji temelji se na uvođenju brojača unazadnih programskih skokova (eng. *backward jumps*). Kada brojač dostigne unaprijed definiranu vrijednost, započinje faza praćenja (eng. *tracing*).

Metoda praćenja temelji se na bilježenju svih operacija prevoditelja do trenutka kada je uočena „vruća“ petlja. Lista zabilježenih operacija, zajedno s vrijednostima varijabli i rezultata, naziva se trag (eng. *trace*) i predaje se JIT kompilatoru, čime započinje faza kompiliranja (eng.

compilation). Za provjeru pojave „vruće“ petlje koristi se tzv. pozicijski ključ. Pozicijski ključ sadrži vrijednosti koje opisuju trenutnu poziciju programa u procesu izvršavanja. Pozicijski ključ provjerava se samo kod instrukcija koje mogu ponoviti izvršavanje prethodnih instrukcija (npr. unazadni programski skokovi). Ako je ustanovljeno da je vrijednost pozicijskog ključa već viđena, otkrivena je „vruća“ petlja. Nadalje, tragom je predstavljena jedna od mogućih putanji prolaska kroz kôd, zbog čega je uveden element za provjeru njegove valjanosti (eng. *guard*) kako bi se održala učinkovitost izvođenja. Ako provjera elementa ne uspije, tada se trag ne prevodi u strojni kôd te se nastavlja standardna interpretacija prevoditeljem.

Osnovna uloga JIT kompilatora je prevesti dobiveni trag u strojni kôd prikladan za izvršavanje na procesoru. Generirani strojni kôd izvršava se odmah i moguće ga je koristiti u sljedećoj iteraciji petlje.

Izvršavanje (eng. *running*) strojnog kôda odvija se do trenutka kada je zadovoljen izlazni uvjet. Ako dođe do izlaska iz petlje, tada se nastavlja standardno izvođenje prevoditelja ili se izvodi sljedeći kompilirani kôd.

PyPy prateći JIT kompilator

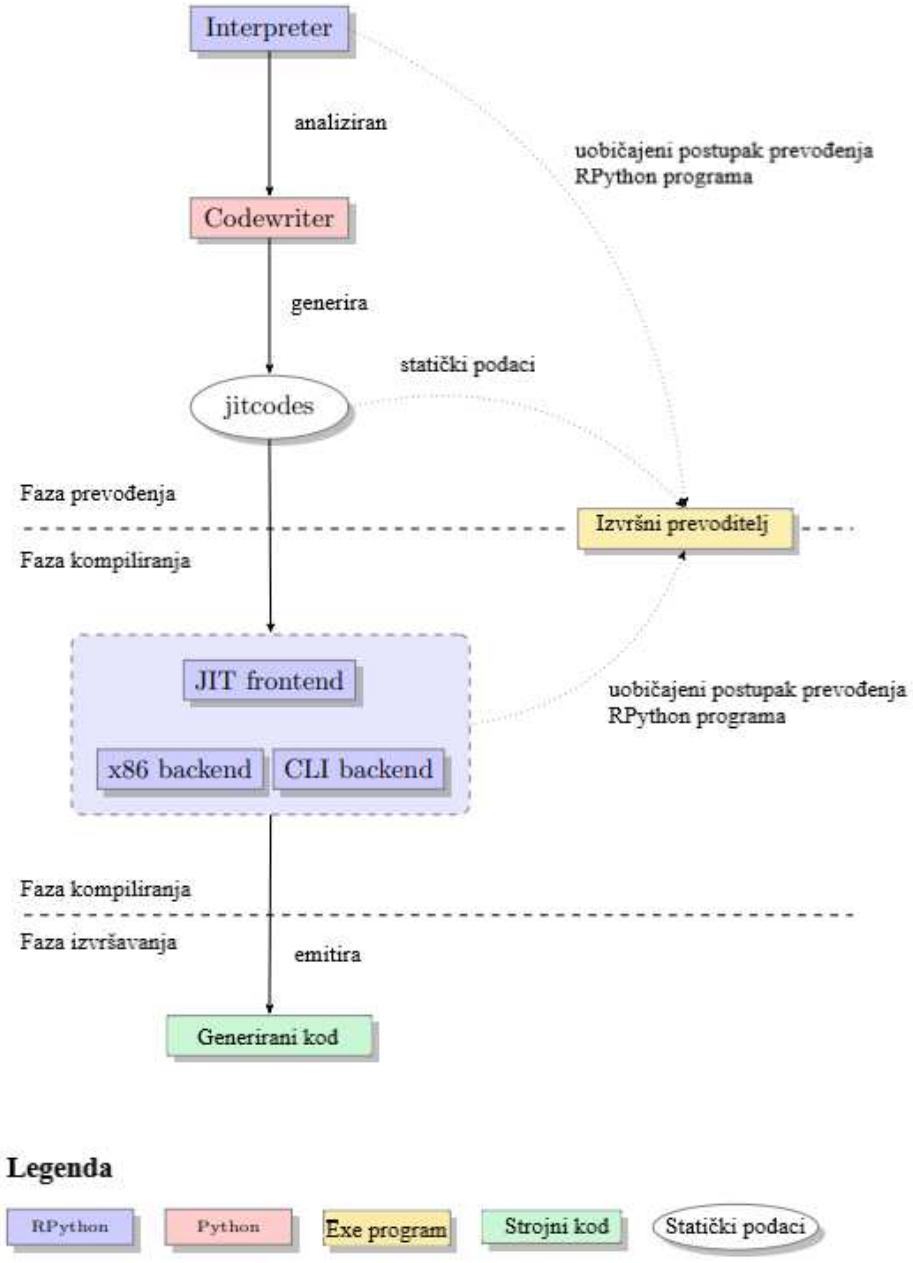
PyPy predstavlja drugačiju implementaciju i integraciju pratećeg JIT kompilatora. Model je sličan prethodno opisanome, s nekoliko ključnih izmjena, a njegova arhitektura je prikazana slikom 2.3 [7].

Kako bi PyPy prevoditelj koristio JIT kompilator, njegov izvorni RPython kôd mora proći kroz dvije različite putanje tijekom procesa prevođenja. Prva putanja definirana je uobičajenim prolaskom kroz RPython lanac (poglavlje 2.1.2) i rezultira konačnom verzijom prevoditelja koja je uključena u izvršni program. Aktivacija JIT kompilatora provedena je prolaskom kroz drugu putanju koja sadrži tzv. šifrant komponentu (eng. *codewriter*). Šifrant uzima izvorni program prevoditelja i stvara novu reprezentaciju kôda koja je razumljiva sučelju JIT generatora, a pohranjena je u obliku jit-kôdova (eng. *jitcodes*). Jit-kôdovi se ugrađuju u izvršni program u obliku statičkih podataka i koristi ih isključivo JIT sučelje za detekciju „vrućih“ petlji.

Komponente JIT sučelja (eng. *JIT frontend*) i podrške (eng. *JIT backend*) također su pisane RPython jezikom i kompilirane su u izvršni program prevoditelja. Osnovna uloga sučelja je izvršavati pristigle jit-kôdove tijekom izvršavanja programa i detektirati „vruće“ petlje

bilježenjem tragova. JIT podrška preuzima oblikovane tragove, kompilira ih u strojni kôd i šalje na izvršavanje.

Iz opisanog modela vidljivo je kako JIT kompilator PyPy prevoditelja nije razvijen kao zasebna komponenta, već djelomično nastaje na temelju izvornog kôda prevoditelja. Preciznije, PyPy okolinom razvijen je generator JIT kompilatora koji ovisi o obilježjima prevoditelja.



Slika 2.3 Arhitekutra JIT generatora PyPy prevoditelja [7]

Prateći JIT kompilator PyPy prevoditelja razlikuje se od ostalih pristupa jer implementira praćenje prevoditelja tijekom izvršavanja korisničkog programa, tj. ne prati korisnički program izravno.

Pronalaženje „vrućih“ petlji i njihovo kompiliranje odvija se nad programom prevoditelja. Glavna petlja prevoditelja (eng. *bytecode dispatch loop*) zaslužna je za upravljanje tijekom izvođenja programa te prevoditelj u njoj provodi najviše vremena, što je motiviralo upotrebu pratećeg JIT kompilatora. Međutim, obilježja glavne petlje ne odgovaraju potpuno pretpostavkama pratećeg JIT kompilatora, točnije pretpostavci da će nekoliko iteracija petlje rezultirati sličnim prolaskom kroz kôd. Jedna iteracija petlje odgovara izvođenju jedne strojne instrukcije, ali šanse za uzastopno izvođenje iste operacije su vrlo male. Istovremeno, jednoj iteraciji petlje korisničkog programa odgovara nekoliko iteracija glavne petlje prevoditelja.

Prateći JIT kompilator PyPy prevoditelja pronalazi rješenje u praćenju petlji korisničkog programa pomoću glavne petlje prevoditelja. Ideja je bilježiti izvođenje glavne petlje prevoditelja do trenutka kada je uočena jedna iteracija petlje korisničkog programa. Pojava petlji u korisničkom programu definirana je ponavljanjem vrijednosti programskog brojača (eng. *program counter*). Programski brojač se sastoji od skupa vrijednosti koje definiraju poziciju sljedeće instrukcije i koriste se za oblikovanje pozicijskog ključa, a pohranjenje su u izvornom kôdu prevoditelja. Prateći JIT kompilator nije upoznat s načinom na koji prevoditelj pohranjuje vrijednosti programskog brojača, stoga ih je potrebno označiti u izvornom kôdu kako bi kompilator dodao vrijednosti u pozicijski ključ. Kada pozicijski ključ poprimi vrijednost jednaku prethodno viđenoj, tada je uočena jedna iteracija petlje korisničkog programa. U izvornom kôdu PyPy prevoditelja, označavanje programskog brojača nalazi se na početku glavne petlje (funkcija *dispatch*) i implementirano je funkcijom *jit_merge_point* (slika 2.4 [11]).

Kako bi izvršavanje PyPy prevoditelja bilo učinkovito, provjera vrijednosti pozicijskog ključa treba se provoditi samo na mjestima gdje je uočen unazadni programski skok. Prateći JIT kompilator ne zna koji dijelovi izvornog kôda prevoditelja implementiraju takav skok, zbog čega ih je potrebno dodatno označiti. Označavanje je provedeno na mjestu gdje je definiran unazadni skok (funkcija *jump_absolute*), a koristi se funkcija *can_enter_jit* (slika 2.5 [11]).

```
def dispatch(self, pycode, next_instr, ec):
    self = hint(self, access_directly=True)
    next_instr = r_uint(next_instr)
    is_being_profiled = self.get_is_being_profiled()
    try:
        while True:
            pypyjitdriver.jit_merge_point(ec=ec,
                frame=self, next_instr=next_instr, pycode=pycode,
                is_being_profiled=is_being_profiled)
            co_code = pycode.co_code
            self.valuestackdepth = hint(self.valuestackdepth, promote=True)
            next_instr = self.handle_bytecode(co_code, next_instr, ec)
            is_being_profiled = self.get_is_being_profiled()
    except Yield:
        self.last_exception = None
        w_result = self.popvalue()
        jit.hint(self, force_virtualizable=True)
        return w_result
    except ExitFrame:
        self.last_exception = None
        return self.popvalue()
```

Slika 2.4 Označavanje vrijednosti programskog brojača [11]

```
def jump_absolute(self, jumpto, ec):
    if we_are_jitted():
        #
        # assume that only threads are using the bytecode counter
        decr_by = 0
        if self.space.actionflag.has_bytecode_counter:    # constant-folded
            if self.space.threadlocals.gil_ready:    # quasi-immutable field
                decr_by = _get_adapted_tick_counter()
        #
        self.last_instr = intmask(jumpto)
        ec.bytecode_trace(self, decr_by)
        jumpto = r_uint(self.last_instr)
    #
    pypyjitdriver.can_enter_jit(frame=self, ec=ec, next_instr=jumpto,
                                pycode=self.getcode(),
                                is_being_profiled=self.get_is_being_profiled())
    return jumpto
```

Slika 2.5 Označavanje unazadnih programskih skokova [11]

Ključan korak u primjeni opisanih oznaka (eng. *hints*) je definiranje varijabli korištenih u glavnoj petlji prevoditelja. Varijable su podijeljene u dvije skupine:

- zelene varijable – predstavljaju pozicijski ključ
- crvene varijable – ostale

Konačno, tragovi proizvedeni sučeljem pratećeg JIT kompilatora nisu spremni za izravno prevođenje u strojni kôd. Nad tragovima je potrebno provesti odgovarajuće korake optimizacije s ciljem učinkovitijeg izvršavanja. Tragovi se prenose kroz uzastopan niz optimizacijskih metoda koje uključuju uklanjanje nepotrebnih operacija te pretvorbu operacija u oblik koji je manje vremenski ili memorijski zahtjevan.

2.2.3 Upravljanje memorijom – postepeno i generacijski

PyPy prevoditelj koristi automatsko upravljanje memorijom pomoću metode generacijskog sakupljanja s postepenim označavanjem [10]. Komponenta za upravljanje memorijskim objektima (eng. *garbage collector*) izvorno je napisana u RPython jeziku te je RPython lancem integrirana u izvršni program prevoditelja. Službeni naziv sakupljača je incminimark.

Osnovna ideja generacijskog upravljanja memorijom temelji se na pretpostavci da se većina objekata odbacuje ubrzo nakon upotrebe. Objekti koji nisu odbačeni kroz određeni period smatraju se dugovječnima i dijelom strukture koja se često koristi. Novostvoreni objekti smještaju se u poseban dio memorije, tzv. jaslice (eng. *nursery*), i nazivaju se mladim objektima. Manje sakupljanje (eng. *minor collection*) provodi se nakon što dođe do potpunog zauzeća jaslica. Mladi objekti koji nisu uništeni tijekom manjeg sakupljanja premještaju se u generaciju starih objekata – stari objekti nalaze se u posebnom, većem dijelu memorije. Zbog veličine jaslica i broja mladih objekata koji su u njima sadržani, postupak manjeg sakupljanja vrlo je brz. Povremeno je potrebno izvršiti veće sakupljanje (eng. *major collection*) koje uključuje sve objekte programa prisutne u memoriji. Veće sakupljanje je zahtjevniji postupak i često dovodi do zaustavljanja izvođenja programa (memorijske pauze) dok se ne riješi problem zauzeća. Sakupljač incminimark dijeli postupak većeg sakupljanja na manje postupke označavanja i brisanja, a raspoređuje njihovo izvođenje neposredno nakon manjih sakupljanja. Cilj ovog

pristupa je umanjiti utjecaj memorijskih pauza na izvršavanje programa. Primjerice, umjesto da se veće sakupljanje izvršava 100 milisekundi u cijelosti, ono se izvršava nekoliko milisekundi nakon svakog manjeg sakupljanja.

U opisanom modelu program se izvršava u intervalima između koraka većeg sakupljanja i može prouzročiti promjenu objekata i njihovih pokazivača, što dovodi do problema prilikom označavanja. U kontekstu postepenog sakupljača koristi se metoda trobojnog označavanja (eng. *tri-color marking*) koja pridaje objektu jednu od tri boje:

1. bijeli objekt – nije pregledan sakupljačem
2. sivi objekt – sakupljač je započeo pregledavanje, ali nije provjerio sve njegove pokazivače
3. crni objekt – sakupljač je završio njegovo pregledavanje

Proces označavanja završen je kada više ne postoje sivi objekti. Svi bijeli objekti koji u tom trenu još postoje nisu dohvatljivi i memorija koju zauzimaju se može osloboditi. Važno je istaknuti činjenicu da crni objekt nikada ne smije sadržavati pokazivač na bijeli objekt. Incminimark implementira posebnu funkcionalnost zaduženu za praćenje promjena nad starim objektima. Zabilježeni stari objekti te mladi objekti izbačeni iz jaslica u prethodnom manjem sakupljanju dodaju se u listu sivih objekata za sljedeći korak većeg sakupljanja. Opisanim pristupom spriječena je pojava nekonzistentnosti prilikom označavanja, tj. pojava zavisnih crnih i bijelih objekata.

2.3 Pregled razlika u odnosu na CPython

Iako su CPython i PyPy prevoditelji implementacije istog programskog jezika, postoje značajne razlike u njihovoj izvedbi.

- Za razliku od CPython prevoditelja, pisanog u jeziku C, izvorni kôd PyPy prevoditelja pisan je u podskupu jezika Python. Odabir jezika Python uveo je prednosti, ali i ograničenja na razvojni proces te konačno rješenje prevoditelja.
- CPython ne koristi naprednu optimizacijsku metodu pratećeg JIT kompilatora već se oslanja na ugrađene tehnike za optimizaciju međukôdova.

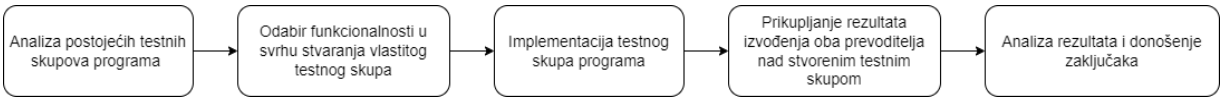
- Umjesto metoda brojanja referenci i otkrivanja ciklusa referenci, PyPy za automatsko upravljanje memorijom koristi generacijski sakupljač objekata s postepenim označavanjem.
- CPython nudi bogatiji skup mogućnosti prilikom razvoja rješenja s C proširenjima. PyPy podrška za C funkcionalnosti postoji, ali je skromnija u odnosu na CPython. Jedan od glavnih razloga sporijeg izvršavanja C proširenja je što PyPy mora imitirati brojanje referenci prilikom upravljanja memorijom.
- Obilježja PyPy prevoditelja mogu se lako prilagoditi zahtjevima korisnika. Moguće je deaktivirati JIT kompilator ili podesiti njegove značajke radi postizanja boljih performansi.

Uz navedene razlike postoje i implementacijske razlike na razini ugrađenih tipova i funkcija. Njihovo iznošenje izlazi iz okvira ovog diplomskog rada. Jedan od ciljeva rada je analizirati učinkovitost izvršavanja Python programa u različitim okolinama te se smatralo važnim predstaviti samo značajne razlike između prevoditelja, na temelju kojih se mogu oblikovati pretpostavke o ishodu eksperimenta.

3 Opis eksperimenta

Poželjno je, često i zahtijevano, da se program odabranih funkcionalnosti izvodi učinkovito u kontekstu vremena i resursa. Važnu ulogu na putu do kvalitetnog izvođenja Python programa može imati odabir prevoditelja. Promatrajući CPython i PyPy arhitekture, lako je za zaključiti kako će njihove performanse dodatno ovisiti o značajkama programa nad kojim se pozivaju. Razlike u izvedbama promatranih prevoditelja potiču pitanje: s obzirom na funkcionalnosti odabranog programa, upotrebom kojeg prevoditelja će se program izvesti učinkovitije?

Praktični dio rada usmjeren je na analizu izvođenja Python programa u dvije različite okoline. Okoline se razlikuju u definiranom prevoditelju – prva koristi CPython, dok je u drugoj okolini postavljen PyPy. Proces izvedbe eksperimenta podijeljen je u nekoliko koraka, a njihov slijed je prikazan na slici 3.1.



Slika 3.1 Faze proces izvedbe eksperimenta

Početna faza uključivala je istraživanje postojećih skupova programa koji su korišteni u projektima slične tematike. Uvid u pronađene primjere potaknuo je stvaranje vlastitog testnog skupa koji će se u nastavku koristiti za prikupljanje rezultata izvođenja u postavljenim okolinama. Nakon provedene programske implementacije programi su pokretani u obje okoline te su prikupljene vrijednosti mjera ključnih za analizu učinkovitosti. Završnim korakom učinjena je obrada i evaluacija rezultata. Ishod uključuje donošenje zaključaka o prikladnosti upotrebe pojedinog prevoditelja te kako obilježja programa utječu na njihove performanse.

U nastavku poglavlja predstavljeni su osnovni izazovi koji su utjecali na pripremu i implementaciju projekta. Nadalje, opisano je razvojno okruženje te je dan detaljan pregled odabranog ispitnog skupa programa, uključujući opis njihovih funkcionalnosti i ključnih karakteristika. Konačno, predstavljeni su glavni ciljevi eksperimenta i pretpostavke o ishodu.

3.1 Izazovi

U procesu definiranja eksperimenta istaknula su se dva zahtjevna izazova – donošenje odluke o obliku testnog skupa i odabir odgovarajućeg alata za praćenje performansi.

Kako bi ishod bio zadovoljavajuć i u skladu s očekivanjima, ključno je koristiti kvalitetan ispitni skup programa. Odabir reprezentativnog skupa dovodi do jasnijih rezultata, a donošenje zaključaka vezanih uz učinkovitost pojedinog prevoditelja je preciznije te se može smatrati općenito primjenjivim. Međutim, izbor prikladnih programa za izgradnju testnog skupa pokazao se prilično izazovnim. Cilj postupka bio je odabrati programe koji, u kontekstu primjene Python programskog jezika, pokrivaju dovoljno širok spektar funkcionalnosti i uključuju često korištena rješenja. Na oblikovanje prihvatljivog skupa utjecalo je i postavljeno vremensko ograničenje praktične izvedbe – programi moraju pokrivati što veći dio spektra funkcionalnosti, ali ne smiju biti previše vremenski zahtjevni (kako bi eksperiment bio odrađen u zadanom roku). U poglavlju 3.3 detaljno je opisan izabrani ispitni skup programa i način njihovog izvršavanja u postavljenim okolinama.

Drugi izazov uključivao je odabir prikladnog alata za analizu performansi. Python omogućava korištenje postojećih modula (iz standardne knjižnice ili kao rješenja treće strane) koji omogućavaju analizu izvođenja programa. Neki od istaknutijih primjera su cPython, line_profiler, memory_profiler, py-spy, Scalene itd. Spomenuti alati zahtijevaju prilagodbu samog kôda te mogu značajno utjecati na performanse programa. Jedan od ciljeva je prikupiti podatke koji ovise o karakteristikama programa, bez utjecaja vanjskih komponenti. Također, problemi prilikom usklađivanja verzija alata na CPython i PyPy prevoditeljima dodatno su obeshrabrili njihovo korištenje. Konačno, odabran je alat perf – naredba operacijskog sustava Linux kojom je omogućeno praćenje performansi sustava tijekom izvođenja odabrane naredbe. Istraživanje je pokazalo kako perf ima vrlo mali ili gotovo nikakav utjecaj na performanse programa koji se izvodi. Alat je stekao prednost nad navedenim rješenjima zbog svoje jednostavnosti i bogatog skupa mogućnosti. Način korištenja alata perf u kontekstu eksperimenta opisan je detaljnije u poglavlju 3.3.

3.2 Okolina izvođenja

Prikupljanje rezultata izvođenja programa ostvareno je na prijenosnom računalu sa specifikacijama prikazanim u tablici 3.1. Korišten je operacijski sustav Ubuntu 22.04 LTS koji je pokretan s vanjskog tvrdog diska Seagate Backup Plus Slim (model SRD00F1, veličine memorije od 2 TB).

Tablica 3.1 Specifikacije uređaja korištenog za izvedbu eksperimenta

Model	Lenovo Ideapad Gaming 3
Arhitektura	x64
Procesor	Intel (R) Core (TM) i5 – 10300H, 4 jezgre, 2.50 GHz radni takt
Radna memorija (RAM)	8 GB
Tvrdi disk (SSD)	512 GB
Grafička kartica	NVIDIA GeForce GTX 1650

Projekt je oblikovan koristeći conda virtualno okruženje radi jednostavnijeg upravljanja Python paketima, s primarnim kanalom za dohvat paketa conda-forge. Za upravljanje projektom korišteno je integrirano razvojno okruženje PyCharm Professional 2021.3.1.

Odabrane su sljedeće verzije prevoditelja [2]:

- CPython 3.9.7 (datum izlaska: 30.08.2021.)
- PyPy 7.3.7 (datum izlaska:)

3.3 Ispitni skup i praktična izvedba

U nastavku je dan detaljan pregled karakteristika korištenog ispitnog skupa, nakon čega je pobliže opisan način izvedbe eksperimenta.

[2] Navedene verzije prevoditelja nisu zadnje dostupne verzije. Prilikom instalacije i konfiguracije okoline koristeći najnovije Python implementacije došlo je do značajnih nepodudaranja u verzijama potrebnih paketa. Probleme neusklađenosti nije bilo moguće riješiti u odgovarajućem roku, zbog čega su odabrane ranije verzije prevoditelja. Konačno, podaci prikupljeni eksperimentom i zaključci doneseni u okviru ovog rada su i dalje validni te se mogu smatrati kvalitetnim rezultatima.

3.3.1 Karakteristike ispitnog skupa

Za izgradnju ispitnog skupa korištena su 22 programa. Tablica 3.2 nudi prikaz svih programa i opise njihovih funkcionalnosti.

Tablica 3.2 Ispitni programi s opisom funkcionalnosti

	Naziv programa	Opis funkcionalnosti
1	bubble_sort	sortiranje liste cijelih brojeva
2	classification	linearna i kvadratna diskriminantna analiza s kovarijantnim elipsoidom (klasifikacijski algoritam)
3	clustering	usporedba različitih hijerarhijskih metoda povezivanje (algoritam grupiranja)
4	encryption_decryption	enkripcija i dekripcija tekstualne poruke koristeći RSA
5	fannkuch	rješenje problema sortiranja u minimalnom broju koraka
6	fibonnaci	izračun Fibonnacijevog niza korištenjem rekurzivnih poziva
7	function_calls	ponavljajući pozivi ugniježđenih funkcija
8	girvan_newman	implementacija algoritma za otkrivanje zajednica u velikim skupovima podataka
9	large_sum	ispis prvih 10 znamenki zbroja skupa velikih brojeva (Project Euler, problem 13)
10	largest_prime_factor	pronalazak najvećeg prostog faktora velikog broja (Project Euler, problem 3)
11	lexicographic_permutations	pronalazak milijunte leksikografske permutacije zadanog niza cijelih brojeva
12	nbody	simulacija planetarnih orbita
13	nqueens_solver	rješenje problema N kraljica
14	outlier_detection	otkrivanje odstupanja na stvarnom skupu podataka
15	park_chen_yu	algoritam za pronalaženje čestih skupova predmeta u velikim skupovima podataka
16	pi_digits	izračun broja Pi
17	raytracing	algoritam praćenja zraka svjetlosti
18	regex_web_page	pretraga sadržaja web stranice regularnim izrazom
19	regression	regresija algoritmom K najbližih susjeda
20	shared_database_mp	višeprocesni pristup zajedničkom resursu
21	special_pythagorean_triplet	pronalazak Pitagorine trojke (Project Euler, problem 9)
22	str_list_sort	sortiranje velike liste znakovnih nizova

Programi ispitnog skupa nude rješenja korištena u različitim područjima primjene programskog jezika Python. Ukupni ispitni skup, korišten u praktičnoj izvedbi, sastoji se od 25 testnih primjera – svaki program naveden u tablici 3.2 predstavlja jednu ispitnu komponentu, osim programa fannkuch, fibonnaci i pi_digits koji su pokretani s dvije različite vrijednosti ulaznog parametra. Nadalje, pojedini programi implementiraju funkcionalnosti poželjne za ispitni skup, ali zbog zahtjevnosti izvedbe nisu dio vlastite implementacije već su preuzeti s vanjskih izvora

[19,20,21]. Istraživanje literature, vezane uz arhitekture i svojstva oba prevoditelja, je usmjerilo pažnju na potencijalno važan element koji bi mogao dovesti do preciznijih zaključaka. Naime, rijetko kada je Python implementacija odabrane funkcionalnosti izvedena koristeći tzv. „čisti“ Python – često su korištene postojeće knjižnice s modulima djelomično pisanim programskim jezikom C. Upotrebom navedenih knjižnica se pojednostavljuje i ubrzava razvojni ciklus. Vrijedno je analizirati kako na učinkovitost izvođenja programa prevoditeljem utječe i prisutnost knjižnica koje koriste pozive C funkcija.

Tablica 3.3 pobliže opisuje karakteristike svih programa ispitnog skupa. Simbol * označava da program koristi knjižnice (module) djelomično napisane u jeziku C, međutim njihovi pozivi unutar programa su zanemarivi te se program može smatrati „čistom“ Python implementacijom.

Tablica 3.3 Karakteristike ispitnih programa

	Naziv programa	Vlastita implementacija	Ulazni parametar	Vrijednosti ulaznog parametra	Prisutna C proširenja
1	bubble_sort	✓	×	-	× *
2	classification	×	×	-	✓
3	clustering	×	×	-	✓
4	encryption_decryption	✓	×	-	✓
5	fannkuch	✓	✓	10, 11	× *
6	fibonnaci	✓	✓	30, 40	× *
7	function_calls	✓	×	-	×
8	girvan_newman	✓	×	-	✓
9	large_sum	✓	×	-	×
10	largest_prime_factor	✓	×	-	✓
11	lexicographic_permutations	✓	×	-	✓
12	nbody	×	×	-	✓
13	nqueens_solver	✓	×	-	×
14	outlier_detection	×	×	-	✓
15	park_chen_yu	✓	×	-	✓
16	pi_digits	✓	✓	1000, 20000	× *
17	raytracing	×	×	-	× *
18	regex_web_page	✓	×	-	✓
19	regression	×	×	-	✓
20	shared_database_mp	✓	×	-	✓
21	special_pythagorean_triplet	✓	×	-	×
22	str_list_sort	✓	×	-	✓

3.3.2 Izvođenje programa i metoda prikupljanja rezultata

U kontekstu diplomskog rada definiraju se sljedeća dva izraza, po uzoru na [4]:

1. kratkotrajno izvođenje – funkcionalnost programa ispitnog skupa pokreće se jednom unutar jednog poziva prevoditelja
2. dugotrajno izvođenje – funkcionalnost programa ispitnog skupa pokreće se više puta unutar jednog poziva prevoditelja

Na temelju opisanih načina izvođenja programa, prikupljanje podataka o performansama podijeljeno je u dvije faze. U prvoj fazi provodi se poziv prevoditelja nad programom, pri čemu se funkcionalnost programa izvodi jednom. Postupak se ponavlja 20 puta, čime se dobivaju prosječne vrijednosti promatranih performansnih mjera. Cilj prve faze je prikupiti rezultate izvođenja za programe relativno kratkog vremenskog trajanja. Druga faza izvedena je na način kojim se pokušalo oponašati izvođenje dugotrajnijih Python aplikacija – prevoditelj se poziva jednom nad programom ispitnog skupa čija funkcionalnost se izvršava ponavljajući, 20 puta uzastopno. Opisana podjela napravljena je s ciljem dobivanja uvida u uspješnosti prevoditelja CPython i PyPy ovisno o trajanju izvođenja odabranog programa. Također, zanimljiva je i informacija o učinkovitosti PyPy prevoditelja u istom kontekstu.

Kako bi daljnja analiza izvršavanja programa bila precizna i kvalitetna, korišten je alat perf i njegova naredba stat za prikupljanje sljedećih mjera:

- ukupno vrijeme izvođenja
- ukupan broj instrukcija po ciklusu (eng. *instruction per cycle*; u nastavku IPC)
- ukupan broj instrukcija procesora izvršenih tijekom izvršavanja programa (u nastavku ukupan broj dinamičkih instrukcija)
- ukupan broj provedenih logičkih grananja
- ukupan broj promašaja prilikom prolaska po logičkim granama
- ukupan broj čitanja iz priručne (eng. *cache*) memorije zadnje razine
- ukupan broj promašaja prilikom čitanja iz priručne memorije zadnje razine

U nastavku se definira mjera ubrzanja kao omjer vremena izvođenja programa koristeći CPython prevoditelj i vremena izvođenja programa koristeći PyPy prevoditelj. Trajanje

izvođenja pod prevoditeljem CPython predstavlja referentnu vrijednost, npr. ako je vrijednost ubrzanja jednaka 10, to znači da korištenje prevoditelja PyPy rezultira 10 puta bržim izvođenjem nad odabranim programom. Nad mjerom ubrzanja provodi se izračun prosjeka koristeći formulu za harmonijsku sredinu. Harmonijska sredina smatra se prikladnim odabirom – smanjuje mogućnost pogrešne interpretacije i pridaje smislenije značenje dobivenim rezultatima [4,5].

3.4 Ciljevi i pretpostavke

Osnovni cilj je utvrditi ovisnost učinkovitosti korištenja CPython i PyPy prevoditelja o obilježjima programa nad kojima su pozivani. Rezultati eksperimenta mogu pomoći u budućem donošenju odluka vezanih uz odabir prevoditelja za specifični Python program. Između ostalog, nakon provedene analize očekuje se potvrda karakteristika koje su navedene u literaturi i dokumentaciji prevoditelja PyPy, ali i odgovor na pitanje: je li PyPy prevoditelj kvalitetna konkurencija CPython implementaciji?

U nastavku su navedene ključne pretpostavke o ishodu eksperimenta:

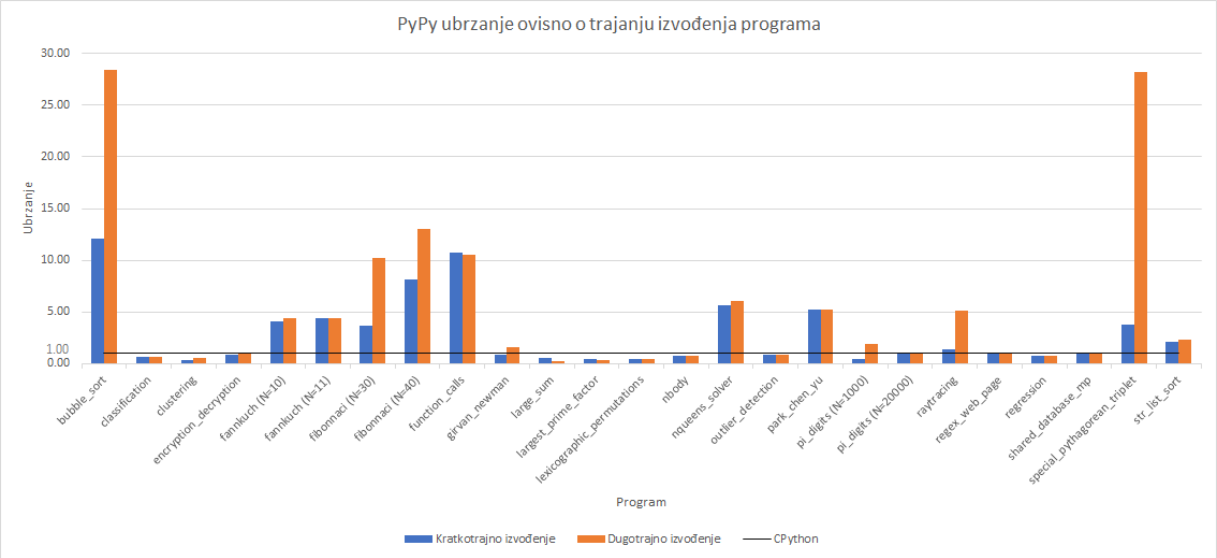
- PyPy će brže izvršiti programe koji su pisani „čistim“ Python jezikom
- programi koji koriste knjižnice djelomično pisane u jeziku C učinkovitije će se izvršavati koristeći CPython
- kod kratkotrajnih programa, koji nisu funkcionalno zahtjevni, odabir prevoditelja neće igrati veliku ulogu
- PyPy će biti učinkovitiji nad programima koji koriste velik broj petlji i/ili ponavljajuće pozive istih funkcija
- općenito, PyPy će postići veće ubrzanje nad dugotrajnim programima u usporedbi s kratkotrajnim programima

4 Rezultati

Analiza učinkovitosti započeta je statističkom obradom prikupljenih podataka i kreiranjem prikladnih vizualizacija, s ciljem stjecanja preglednijeg uvida u razlike i trendove kretanja vrijednosti odabranih mjera učinkovitosti s obzirom na obilježja programa. Tumačenje rezultata provedeno je u kontekstu dva različita stajališta. Prvo stajalište usmjereno je na analizu učinkovitosti s obzirom na različite okoline izvođenja (poglavlje 4.1). Drugim stajalištem provodi se usporedba rezultata dobivenih izvođenjem u okolini PyPy, s naglaskom na obilježja programa (poglavlje 4.2).

4.1 Usporedna analiza prevoditelja

Na slici 4.1 prikazano je ubrzanje dobiveno korištenjem PyPy prevoditelja za kratkotrajan i dugotrajan način izvođenja programa.



Slika 4.1 Ubrzanje PyPy prevoditelja

Izračunom harmonijske sredine ubrzanja dolazi se do srednje vrijednosti ubrzanja od 1.008 za skup kratkotrajnog izvođenja te 1.122 za skup dugotrajnog izvođenja. Srednja vrijednost pokazuje kako izbor prevoditelja ne igra veliku ulogu kod programa koji se izvršavaju u relativno kratkom periodu. Međutim, povećanje ubrzanja opaženo kod dugotrajnijih programa

potvrđuje činjenicu da JIT kompilator otkriva ponavljajući kôd i time optimizira izvođenje programa. Maksimalno postignuto ubrzanje je 12.061 za skupinu kratkotrajnih programa te 28.401 za skupinu dugotrajnih programa. Istovremeno, PyPy pojedine programe značajno sporije izvodi, s minimalnim ubrzanjem od 0.344 za skupinu kratkotrajnih programa i 0.306 za skupinu dugotrajnih programa.

Primijećeno je kako ubrzanje mnogo ovisi i o samim obilježjima programa. Tablicom 4.1 prikazane su prosječne vrijednosti mjera učinkovitosti (trajanje izvođenja i mjera IPC) ovisno o načinu na koji je program pisan, tj. je li programski kôd „čisti" Python ili nije [3].

Tablica 4.1 Prosječne vrijednosti mjera učinkovitosti ovisno o stilu izvedbe programa

		„Čisti" Python		Python s C proširenjima	
		CPython	PyPy	CPython	PyPy
Vrijeme [s]	Kratkotrajno	16.14 (8.67)	3.10 (1.69)	5.69 (13.39)	6.27 (7.33)
	Dugotrajno	321.57 (169.72)	58.28 (29.33)	98.56 (255.89)	100.30 (123.80)
IPC	Kratkotrajno	2.83 (2.81)	1.94 (1.86)	1.63 (1.74)	1.37 (1.49)
	Dugotrajno	2.93 (2.91)	2.19 (2.14)	1.93 (2.02)	1.61 (1.70)

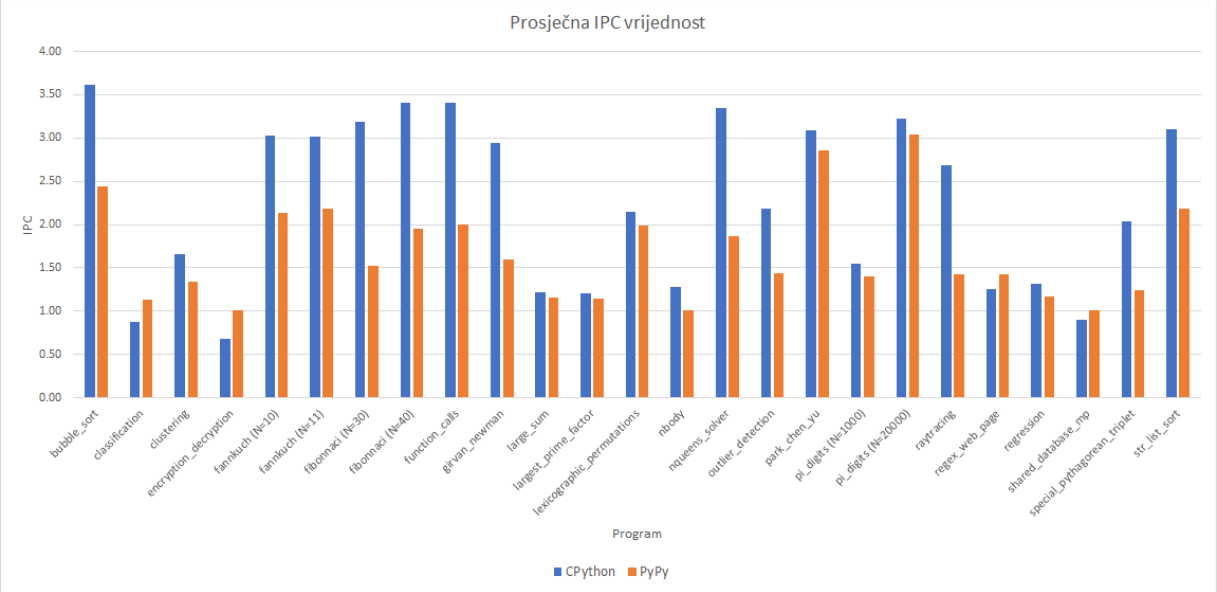
Iz rezultata prikazanih u tablici jasno je vidljivo kako se „čisti" Python programi osjetno brže izvode u okolini s PyPy prevoditeljem, neovisno o načinu izvođenja. Za dugotrajne programe razlika prosječnih vrijednosti je približno 265 sekundi, što se ne može smatrati zanemarivim. Nadalje, CPython prevoditelj je učinkovitiji nad programima koji koriste postojeće module s pozivima C funkcionalnosti [4] – PyPy okolina nudi slabiju podršku za korištenje C proširenja što rezultira sporijim izvođenjem programa koji ih koriste. Preciznije, za programe koji se značajno oslanjaju na upotrebu C proširenja, s naglaskom na korištenje numpy, scikit-learn i matplotlib knjižnica, okolina PyPy nije odgovarajući izbor jer znatno narušava brzinu izvršavanja. Među njima se ističe program clustering koji se u kratkotrajnom načinu izvođenja izvodi gotovo 3 puta sporije u odnosu na okolinu s CPython prevoditeljem.

[3] Program park_chen_yu početno je svrstan u kategoriju programa koji koriste knjižnice proširene jezikom C, što zaista i je slučaj. Međutim, rezultati su pokazali kako program ukupno provodi vrlo malo vremena u pozivima C funkcija i mogao bi se smatrati „čistim" Python programom. Vrijednosti u zagradama tablice 4.1 označavaju rezultate dobivene u slučaju kada je park_chen_yu promatran kao program koji koristi pozive prema C funkcijama.
[4] Razlika u vremenu izvođenja bila bi veća kada bi programi provodili više vremena izvršavajući funkcije prožete C proširenjima. Odabrani programi koriste module s C podrškom, no ispostavilo se da nisu dovoljno funkcionalno zahtjevni u tom kontekstu. Bez obzira na zapažanje, rezultati su dovoljno jasni i u skladu s očekivanjima.

CPython postiže bolje rezultate nad programima koji su vrlo kratki i nisu funkcionalno zahtjevni, npr. large_sum, largest_prime_factor, lexicographic_permutations. Uzrok tome je što JIT kompilator PyPy prevoditelja unosi nepotrebne troškove u proces izvođenja programa. Ako se program ne izvodi dovoljno dugo, JIT kompilator nema dovoljno vremena za optimizaciju te dodatno usporava izvršavanje programa.

Kod programa koji koriste ugniježđene petlje i/ili ponavljajuće pozive funkcija (npr. bubble_sort, fibonnaci, function_calls) dolazi do znatnog ubrzanja prilikom izvršavanja u PyPy okolini. Važno je istaknuti kako se radi o „čistim“ Python programima – kada bi se većinom oslanjali na C proširenja, bez obzira na prisustvo ponavljajućeg kôda, moguće je da ubrzanja ne bi bilo ili bi ono bilo manje.

Proučavajući trend vrijednosti na slici 4.1 uočeno je kako izvođenje programa girvan_newman i pi_digits (N=1000) rezultira različitim ubrzanjem ovisno o načinu izvođenja. U kontekstu kratkotrajnog izvođenja poželjnije je odabrati CPython prevoditelj za oba programa, a kod dugotrajnog izvođenja se brže izvršavanje postiže prevoditeljem PyPy. Za rezultate preostalih programa može se reći da su jednoznačni u oba slučaja – učinkovitiji je ili CPython, ili PyPy, ili odabir nema utjecaja. Iz istaknutog zapažanja vidljivo je koliko zaista vrijeme izvođenja ovisi o obilježjima programa i da proces odabira prikladnog prevoditelja može znatno utjecati na buduće performanse Python aplikacija.

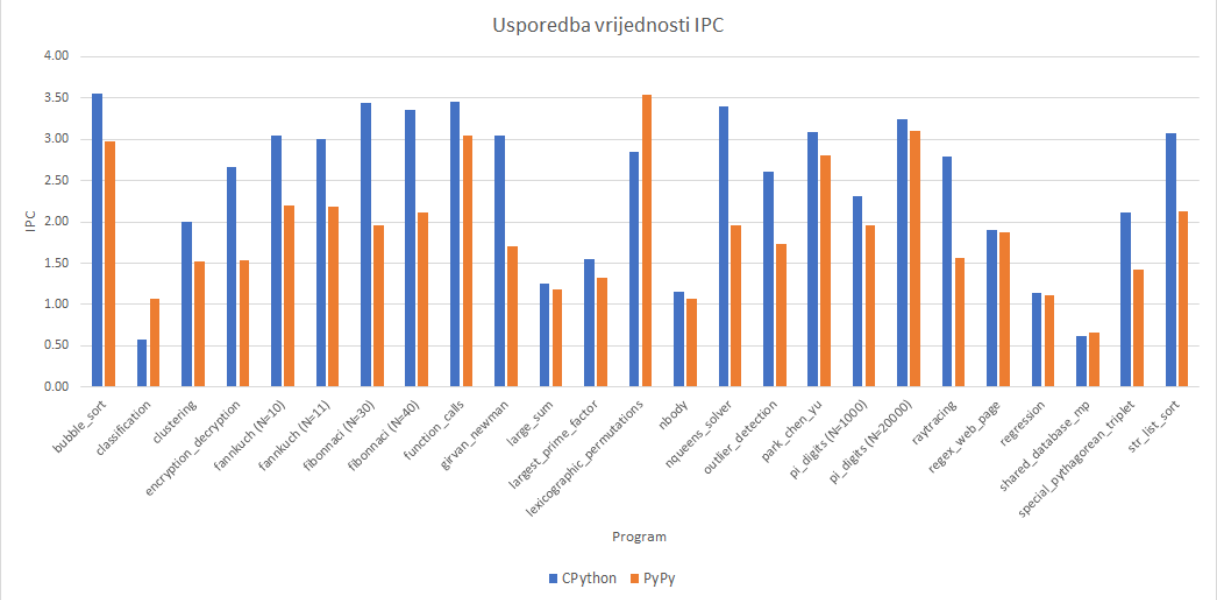


Slika 4.2 Vrijednost IPC mjere kratkotrajnih programa

Proces izvođenja programa može se analizirati i na razini mjere IPC i ukupnog broja dinamičkih instrukcija. Učinkovitost procesora prilikom izvršavanja odabranog programa definirana je sljedećom formulom:

$$Vrijeme\ procesora\ za\ program\ = ukupan\ broj\ dinamičkih\ \ instrukcija \times \frac{1}{IPC} \times C \quad (1)$$

, gdje C predstavlja trajanje ciklusa sata procesora. U okviru eksperimenta, vrijednost varijable C je konstantna, zbog čega je poželjno da ukupan broj dinamičkih instrukcija bude što manji, a mjera IPC što veća. Na slikama 4.2 i 4.3 prikazana je usporedba IPC vrijednosti ovisno o okolini izvođenja. Slike 4.4 i 4.5 vizualiziraju stopu smanjenja ukupnog broja dinamičkih instrukcija postignutu prevoditeljem PyPy [5].



Slika 4.3 Vrijednost IPC mjere dugotrajnih programa

Na temelju danih vizualizacija i rezultata tablice 4.1 lako se zaključuje da CPython, u većini slučajeva, postiže veću vrijednost mjere IPC. Za „čiste“ Python programe prosječna vrijednost IPC mjere je čak 1.46 puta veća od one postignute prevoditeljem PyPy, dok za ostale programe

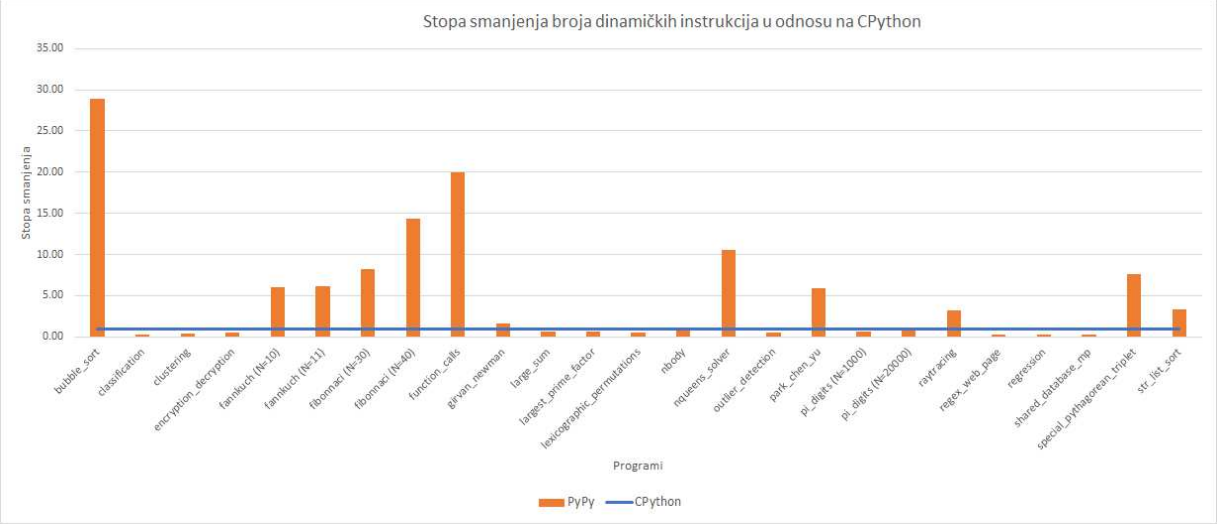
[5] Iz vizualizacije na slici 4.5 uklonjena je vrijednost programa special_pythagorean_triplet. Stopa smanjenja za navedeni program je 116.14, što je puno veće u odnosu na preostale vrijednosti i dovodilo je do znatno slabije preglednosti grafa.

vrijedi povećanje od 1.19 puta. Također, primijećen je dodatan pad vrijednosti IPC mjere, u obje okoline, za programe koji koriste C proširenja u odnosu na „čiste“ Python programe. Prilikom dugotrajnijeg izvođenja programa dolazi do povećanja vrijednost IPC mjere bez obzira na odabir prevoditelja.

Tablica 4.2 Smanjenje prosječnog broja dinamičkih instrukcija u okolini PyPy

	„Čisti“ Python	Python s C proširenjima
Kratkotrajno	6.66 x	0.82 x
Dugotrajno	6.85 x	1.15 x

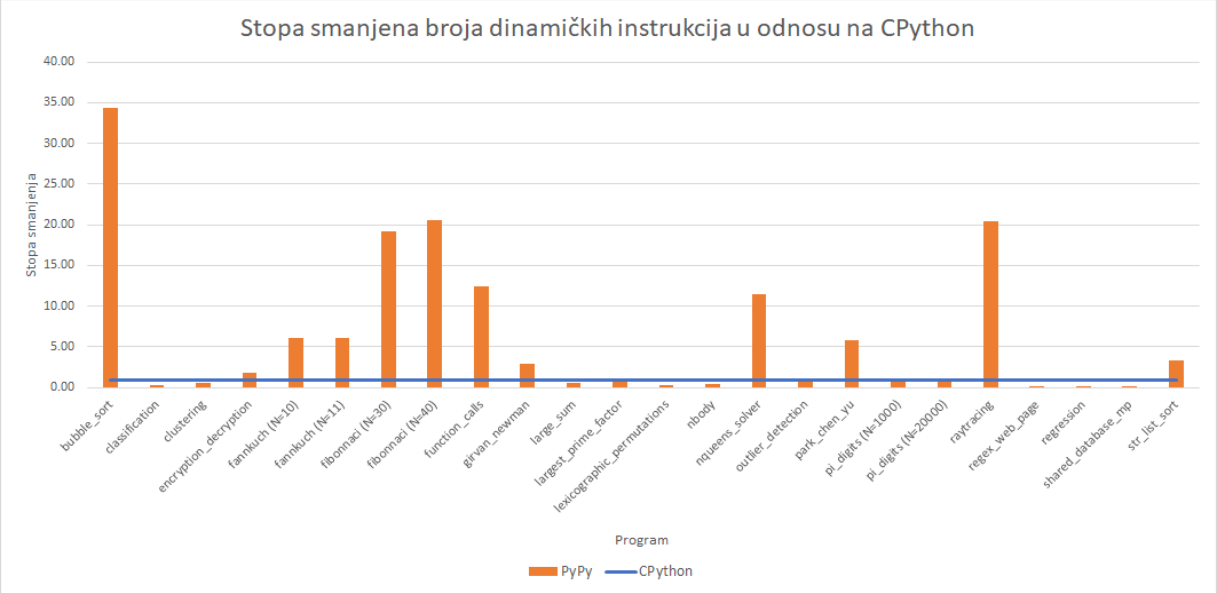
Prosječan broj dinamičkih instrukcija, prilikom izvođenja programa s PyPy prevoditeljem, 4.95 puta je manji za kratkotrajne programe i 5.71 puta manji za dugotrajne programe u odnosu na CPython. Nadalje, primijećena je i ovisnost broja instrukcija o upotrebi modula s C proširenjima – PyPy prevoditelj značajno smanjuje broj instrukcija prilikom izvođenja „čistih“ Python programa, u prosjeku za 6.76 puta. Kod programa s C podrškom broj instrukcija ili nije manji (kratkotrajno izvođenje) ili je neprimjetno smanjen (dugotrajno izvođenje).



Slika 4.4 Stopa smanjenja broja dinamičkih instrukcija kratkotrajnih programa

Zanimljivo je istaknuti kako do većeg smanjenja dolazi izvođenjem dugotrajnijih programa, što ukazuje na bolju prilagodbu JIT kompilatora i time učinkovitiju optimizaciju. Primjerice, izvršavanjem programa bubble_sort u PyPy okolini postiže se 28.94 puta manje dinamičkih

instrukcija u kontekstu kratkotrajnog izvođenja (slika 4.4) dok je za dugotrajno izvođenje broj instrukcija 34.40 puta manji u odnosu na CPython (slika 4.5).



Slika 4.5 Stopa smanjenja broja dinamičkih instrukcija dugotrajnih programa

Za programe bubble_sort, function_calls i park_chen_yu postiže se veliko smanjenje broja dinamičkih instrukcija te je vrijednost IPC mjere veća od prosjeka. Istaknuto zapažanje dodatno potvrđuje činjenicu da PyPy prevoditelj dovodi do znatno veće učinkovitosti nad „čistim“ Python programima koji sadrže veliku količinu petlji, tj. ponavljajućeg kôda.

Konačno, uočeno je kako učinkovitost izvršavanja ovisi i o vrijednosti ulaznog parametra kojim se definira programska zahtjevnost. Zaključak proizlazi iz analize rezultata programa fibonnaci, s ulaznim vrijednostima 30 i 40, te programa pi_digits, s ulaznim vrijednostima 1000 i 20000. PyPy postiže veće ubrzanje tijekom izvršavanja programa fibonnaci s većom ulaznom vrijednošću, u oba načina izvođenja. Međutim, prilikom izvođenja programa pi_digits nije uočen isti trend. U kontekstu kratkotrajnog izvođenja prednost ima veća ulazna vrijednost, a kod dugotrajnog izvođenja do izražaja dolazi manja ulazna vrijednost. Iako oba programa pripadaju skupini „čistih“ Python programa, pi_digits ne sadrži ugniježđene petlje (nema dovoljno ponavljajućeg kôda), zbog čega dolazi do nedosljednih rezultata i slabijih performansi JIT kompilatora.

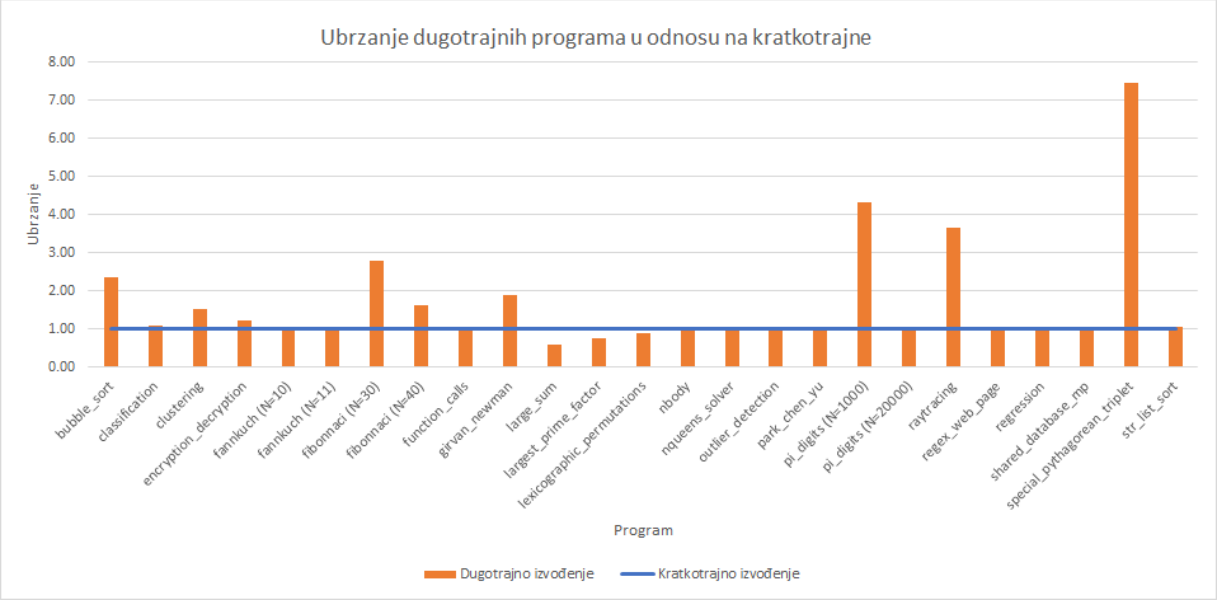
4.2 Utjecaj obilježja programa na izvođenje u PyPy okolini

Nakon provedene usporedbe prevoditelja detaljnije je istražen utjecaj karakteristika programa na performanse PyPy prevoditelja. Do sada opisani rezultati istaknuli su povećanje učinkovitosti prilikom izvođenja dugotrajnijih programa u okolini PyPy. Jedan od pokazatelja je porast u srednjoj vrijednosti ubrzanja – ono je u prosjeku veće za 1.113 puta u odnosu na ubrzanje postignuto tijekom izvršavanja kratkotrajnih programa. Tablicom 4.3 dodatno je prikazan odnos dobivenog ubrzanja, pri tome uzimajući u obzir podjelu ispitnog skupa na temelju obilježja programa. Izvršavanje „čistih" Python programa rezultira povećanjem srednje vrijednosti ubrzanja u odnosu na programe koji koriste knjižnice s C podrškom, neovisno o duljini trajanja programa. Nadalje, dugotrajno izvođenje „čistih" Python programa dovodi do 1.114 puta većeg prosječnog ubrzanja u odnosu na kratkotrajno izvođenje, dok je za programe s C proširenjima primijećen porast od 1.110 puta. Slikom 4.6 dan je pregledniji prikaz dobivenog ubrzanja za svaki program ispitnog skupa, čime se dodatno potvrđuju uočena zapažanja.

Tablica 4.3 Srednja vrijednost ubrzanja ovisno o podjeli ispitnog skup

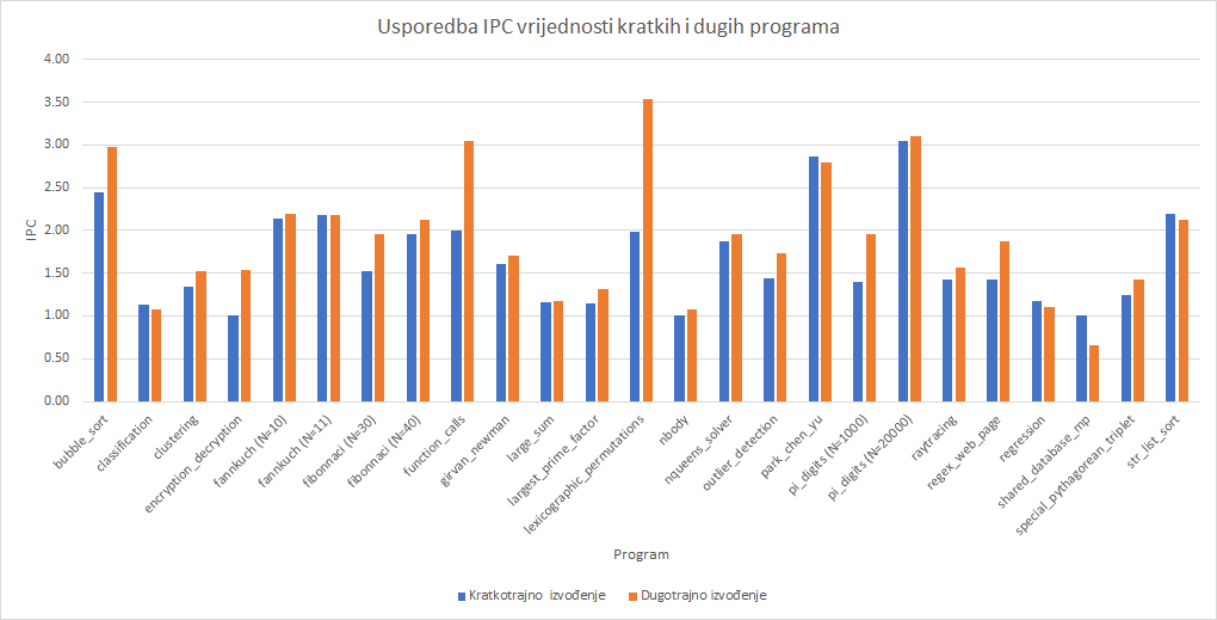
	„Čisti" Python	Python s C proširenjima
Kratkotrajno	0.986	0.971
Dugotrajno	1.098	1.078

Iako programi clustering i girvan_newman koriste proširene knjižnice, primijećen je porast u ubrzanju tijekom dugotrajnijeg izvođenja programa. Iz istaknutih rezultata dodatno je vidljiv utjecaj trajanja izvođenja na učinkovitost izvršavanja. Ako program koristi C proširenja, ali izvodi se dovoljno dugo, JIT kompilator (i ostale komponente PyPy prevoditelja) dovodi do uspješnije optimizacije puta izvršavanja. Istovremeno, za pojedine programe (npr. fannkuch, function_calls, nqueens_solver, outlier_detection itd.) postignuto ubrzanje gotovo je jednako u oba načina izvođenja. Istaknuti rezultati upućuju na postojanje optimizacijske granice PyPy prevoditelja koja ovisi o složenosti implementacije. Ako se programske funkcionalnosti lako otkriju i optimiziraju JIT kompilatorom, tada dugotrajnije izvršavanje neće dovesti do značajnijeg ubrzanja – postignuta je maksimalna učinkovitost.



Slika 4.6 Ubrzanje tijekom dugotrajnog izvođenja u odnosu na kratkotrajno izvođenje

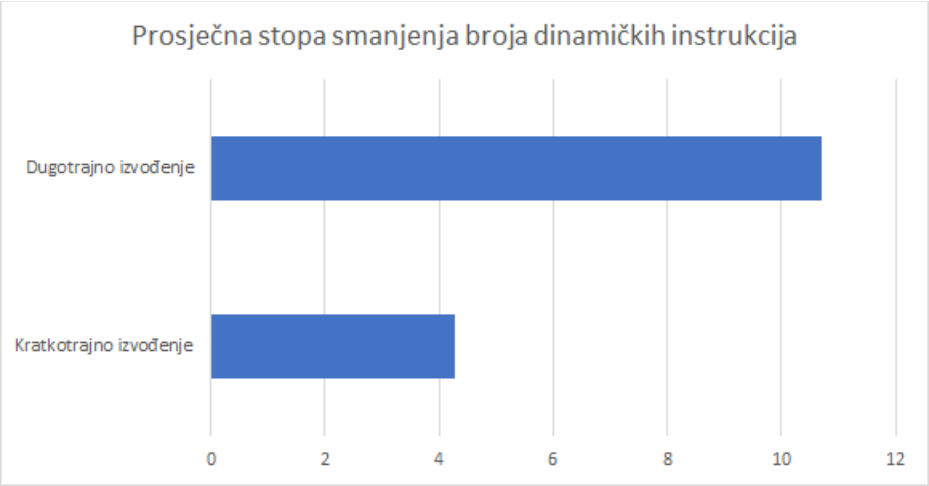
Slikom 4.7 prikazana je usporedba vrijednosti IPC mjere kratkotrajnog i dugotrajnog načina izvođenja programa. Izvršavanje dužih programa u okolini s PyPy prevoditeljem u prosjeku dovodi do povećanja IPC vrijednosti (iznimka su programi classification, regression, shared_database_mp i str_list_sort).



Slika 4.7 Odnos IPC mjere kratkotrajnog i dugotrajnog izvođenja

Vrijednost IPC mjere u dugotrajnom načinu izvođenja „čistih“ Python programa prosječno je veća za 1.13 puta u odnosu na kratkotrajno izvođenje, a za programe s C proširenjima stopa povećanja jednaka je 1.18 (tablica 4.1).

Uz povećanje vrijednosti IPC mjere primijećeno je i značajno smanjenje broja instrukcija kod dugotrajnog izvođenja programa. Prosječna stopa smanjenja broja instrukcija za dugotrajno izvođenje jednaka je 10.70, a za kratkotrajno izvođenje 4.28 (slika 4.8). Broj instrukcija znatno je smanjen za „čiste“ Python programe u odnosu na programe s C proširenjima: prosječna stopa smanjenja veća je za 11.20 puta kod kratkotrajnog izvođenja te čak 19.72 puta kod dugotrajnog izvođenja. Povećanje mjere IPC i smanjenje broja instrukcija kod dugotrajnog izvođenja potvrđuju kako je JIT kompilatoru PyPy prevoditelja potrebno određeno vrijeme za postizanje veće učinkovitosti.



Slika 4.8 Usporedba načina izvođenja na temelju prosječne stope smanjenja broja instrukcija

Prilikom analize rezultata proučen je i utjecaj PyPy prevoditelja na promašaje prilikom čitanja iz zadnje razine priručne memorije te donošenja odluke kod logičkog grananja. Kvalitetno predviđanje sljedećeg koraka može značno utjecati na performanse programa. Također, zadnja razina priručne memorije je ujedno i najsporija te njeni promašaji vode do čitanja iz radne memorije, koje je vremenski skupo, što može dodatno usporiti izvođenje programa. Tablicom 4.4 prikazane su prosječne vrijednosti promašaja kod logičkih grananja i zadnje razine priručne memorije ovisno o načinu izvođenja programa i njihovim obilježjima. Korištena je mjerna jedinica promašaja po kilo-instrukciji (eng. *Misses Per Kilo Instruction*, u nastavku MPKI).

Tablica 4.4 Prosječni promašaji na logičkim granama i zadnjoj razini priručne memorije

	Logičke grane – MPKI		Zadnja razina priručne memorije – MPKI	
	„Čisti“ Python	Python s C proširenjima	„Čisti“ Python	Python s C proširenjima
Kratkotrajno	1.89	3.22	0.14	0.44
Dugotrajno	1.64	2.92	0.12	0.63

Dugotrajnim izvođenjem programa dolazi do smanjenja promašaja u logičkim granama za obje skupine programa. PyPy prevoditelj kvalitetnije pridonosi predviđanju sljedećeg koraka pri logičkom grananju tijekom dužeg izvršavanja, što je i očekivano s obzirom na optimizacijski proces PyPy prevoditelja. Također, primijećena je uzajamna veza s preostalim mjerama učinka – kod dugotrajnog izvođenja rastu IPC i stopa smanjenja broja dinamičkih instrukcija, a MPKI logičkog grananja pada. Uočeni odnos dodatno potvrđuje porast učinkovitosti PyPy prevoditelja prilikom izvođenja zahtjevnijih programa.

Promašaji na zadnjoj razini priručne memorije ne prate jednaki trend kao promašaji logičkog granjanja. U odnosu na kratkotrajno izvođenje, dugotrajnim izvođenjem „čistih“ Python programa smanjuje se mjera MPKI, ali se povećava kod programa koji koriste C proširenja. Porast promašaja zabilježen je za manji skup programa koji uključuje encryption_decryption, lexicographic_permutations, outlier_detection i regex_web_page. Korištenje PyPy prevoditelja rezultiralo je povećanjem IPC mjere istaknutih programa, međutim uočen je značajan porast broja dinamičkih instrukcija. S obzirom na to da do nekonzistentnosti dolazi za mali dio ispitnog skupa, općenito se može zaključiti da PyPy postiže i smanjenje promašaja na zadnjoj razini priručne memorije.

4.3 Donošenje odluke prilikom izbora prevoditelja

Rezultati su pokazali da odabir prikladne Python implementacije može značajno utjecati na učinkovitost izvođenja programa. Analiza provedena nad prikupljenim podacima omogućila je oblikovanje smjernica koje mogu pomoći prilikom odabira prevoditelja za Python programe. Iznesene smjernice oblikovane su na temelju rezultata izvođenja odabranog skupa ispitnih

programa. Iako definirani skup pokriva ograničeni spektar funkcionalnosti i obilježja, smatra se da je skup dovoljno reprezentativan te da su doneseni zaključci široko primjenjivi.

U nastavku su opisani prijedlozi korištenja CPython i PyPy prevoditelja ovisno o obilježjima ciljanog programa.

1. Ako se implementacija programa značajno oslanja na upotrebu knjižnica djelomično pisanih u programskom jeziku C, tada je bolje koristiti CPython. Upotreba PyPy prevoditelja rezultira smanjenjem učinkovitosti izvođenja zbog skromnije podrške za C proširenja, što dovodi do nemogućnosti kvalitetne optimizacije.
2. Ako je program implementiran koristeći „čisti" Python, tada je vrijedno razmotriti upotrebu PyPy prevoditelja. PyPy u većini slučajeva postiže bolje performanse, međutim velik utjecaj ima struktura programskog kôda. Ako je navedeni uvjet zadovoljen, potrebno je nastaviti procjenu uzimajući u obzir način izvedbe funkcionalnosti programa.
3. Ako program zadovoljava uvjet iz točke 2, a istodobno sadrži velik broj (ugniježđenih) petlji i/ili ponavljajuće pozive istih funkcija (općenito ponavljajućeg kôda), tada je prikladnije izabrati PyPy prevoditelj. PyPy omogućuje puno učinkovitije izvršavanje programa s navedenim karakteristikama u odnosu na CPython zahvaljujući optimizaciji JIT kompilatorom.
4. Ako program implementira funkcionalnost koja nije zahtjevna i sadržana je u poprilično kratkom kôdu, tada prednost ima CPython. Često će oba prevoditelja postići gotovo jednaku učinkovitost izvođenja, međutim PyPy unosi optimizacijski trošak koji nije isplativ, a ponekad može nepotrebno usporiti program. Upotreba CPython prevoditelja za opisani program dodatno je podržana ako program koristi podršku programskog jezika C.
5. Ako program zadovoljava uvjet iz točke 2, a zahtijeva duže izvršavanje, tada je prikladnije koristiti PyPy. Dugotrajniji programi često sadrže velik broj ponavljajućih petlji koje PyPy prevoditelj značajno optimizira, što u konačnici dovodi do bržeg izvršavanja programa.
6. Ako se program izvršava duže, sadrži velik broj petlji i koristi knjižnice s pozivima C funkcija, tada odabir prevoditelja ovisi o tijeku izvršavanja programske funkcionalnosti. Preciznije, potrebno je istražiti koliko vremena program zaista provede izvršavajući korištena C proširenja. Ako se pokaže da je istaknuto vrijeme gotovo zanemarivo u usporedbi s ostatkom Python kôda, tada će se program učinkovitije izvršavati u PyPy

okolini. U suprotnome, poželjniji je odabir CPython prevoditelja zbog boljeg upravljanja pozivanim C funkcionalnostima.

7. Ako program implementira funkcionalnost koja se općenito izvršava u vrlo kratkom periodu, tada odabir prevoditelja ne igra veliku ulogu, ali prikladniji izbor je CPython. PyPy prevoditeljem ne može se postići dodatno ubrzanje zbog vremenskog ograničenja, a CPython omogućava bezbrižnije korištenje C knjižnica radi jednostavnije implementacije i bogatijeg skupa mogućnosti.

Odabir prikladnog Python prevoditelja može biti izazovan zadatak jer zahtijeva popratnu analizu strukture i funkcionalnosti programa. Analiza velikih i složenih programa može nepotrebno usporiti razvojni ciklus aplikacija. Predložen je sljedeći model procesa odluke o upotrebi Python prevoditelja:

1) Ciljanu implementaciju programa početno izvršiti koristeći CPython prevoditelj.
2) Analizirati performanse izvođenja. Ustanoviti zadovoljava li vrijeme izvođenja tražene uvjete. Dodatno optimizirati Python kôd ako je to moguće.
3) Ako vrijeme izvođenja programa nije odgovarajuće ni nakon koraka 2, razmotriti upotrebu PyPy prevoditelja. U nastavku analizirati strukturu programskog kôda i tijek izvršavanja programa prema opisanim smjernicama.
4) Upotrijebiti PyPy prevoditelj ako su zadovoljeni uvjeti opisani u točkama 2, 3 i/ili 5. U suprotnome, koristiti CPython prevoditelj i pokušati provesti dodatnu optimizaciju izvedene funkcionalnosti.

5 Zaključak

Upotreba dinamičkih programskih jezika doživjela je značajan porast tijekom posljednjih godina. Njihove implementacije često rezultiraju negativnim utjecajem na učinkovitost izvođenja programa zbog korištenja tradicionalnih tehnika prevođenja međukôdova. Navedeni nedostatak karakterističan je i za standardnu implementaciju programskog jezika Python, CPython. CPython je najkorišteniji Python prevoditelj, a popularnost navedenog programskog jezika potaknula je razvoj novih implementacija s ciljem optimizacije procesa izvršavanja programa. Jedno od rješenja istaknulo se svojim načinom izvedbe, a riječ je o projektu PyPy.

U okviru diplomskog rada predstavljena je okolina PyPy i njena implementacija programskog jezika Python. Detaljno je proučen pristup izgradnji Python prevoditelja i pripadne okoline, s naglaskom na prateći JIT kompilator. Prateći JIT kompilator glavna je komponenta PyPy prevoditelja, a optimizaciju provodi detekcijom često izvođenih petlji koje kompilira u strojni kôd. Ključna razlika u odnosu na druge implementacije je što se praćenje često izvođenih petlji odvija na razini glavne petlje prevoditelja, a ne izravno na petljama korisničkih programa. Ugradnja napredne tehnike kompiliranja omogućila je kvalitetnu optimizaciju i unaprjeđenje implementacije prevoditelja PyPy.

Osnovni zadatak diplomskog rada bio je utvrditi prikladnost upotrebe prevoditelja CPython i PyPy postupkom analize učinkovitosti izvršavanja programa u različitim okolinama. Korišten je ispitni skup programa različitih funkcionalnosti i obilježja s ciljem oblikovanja preciznih zaključaka koji su općenito primjenjivi. PyPy prevoditelj pokazao se učinkovitijim u izvršavanju „čistih“ Python programa, odnosno programi ne koriste C proširenja ili provode jako malo vremena u njihovim pozivima, koji koriste mnogo programskih petlji. Također, za stvaranje prednosti nad CPython prevoditeljem ključno je da se programi s navedenim obilježjima izvode kroz duži period. Upotreba PyPy prevoditelja nad kratkotrajnim programima neće rezultirati učinkovitijim izvođenjem jer prateći JIT kompilator zahtijeva određenu količinu vremena za postizanje kvalitetne optimizacije. Ako su programi relativno kratki ili ako koriste bogati skup C funkcionalnosti, tada je poželjnije koristiti CPython.

Ustanovljeno je kako karakteristike programske izvedbe mogu značajno utjecati na performanse CPython i PyPy prevoditelja. Na temelju navedenog zapažanja oblikovan je model procesa

odluke odabira prikladne Python implementacije. Razmatranje upotrebe PyPy prevoditelja trebalo bi započeti nakon ustanovljenog negativnog utjecaja CPython prevoditelja te popratiti detaljnim profiliranjem strukture i obilježja programa. Konačno, ishod analize je u skladu s očekivanjima i potvrdio je pretpostavke stvorene na temelju tehničkih izvedbi oba prevoditelja.

_____

Mihaela Svetec

Literatura

[1] Lutz, M. „Learning Python“. Četvrto izdanje. O'Reilly Media, 2009.

[2] Jones, R., Hosking A., Moss, E. „The Garbage Collection Handbook: The Art of Automatic Memory Management“. CRC Press, 2012.

[3] Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A. „Tracing the Meta-Level: PyPy's Tracing JIT Compiler“. ICOOOLPS ’09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems. 2009., str. 18-25

[4] Eechhout, L., Crapé, A. „A Rigorous Benchmarking and Performance Analysis Methodology for Python Workloads“. 2020 IEEE International Symposium on Workload Characterization (IISWC). 2020, str. 83-93

[5] Fleming, P.J., Wallace, J.J. „How not to lie with statistics: the correct way to summarize benchmark results“. Communications of the ACM. Svezak 29, treće izdanje. 1986., str. 218-221

[6] Rigo, A., Pedroni, S. „PyPy's Approach to Virtual Machine Construction“. Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. 2006., str. 944-953

[7] Cuni, A. „High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages“. Doktorski rad. Universit`a degli Studi di Genova. 2010.

[8] Deutel, M. „Virtual Machines for Dynamic Languages“. 2021. Poveznica: https://www4.cs.fau.de/Lehre/WS20/MS_AKSS/arbeiten/paper_deutel_final.pdf

[9] Galindo Salgado, P. „Design of CPython's Garbage Collector“. Datum pristupa: 07.06.2022. Poveznica: https://devguide.python.org/garbage_collector/

[10] Fijalkowski, M., Rigo, A. „Incremental Garbage Collector in PyPy“. Datum pristupa: 14.05.2022. Poveznica: https://morepypy.blogspot.com/2013/10/incremental-garbage-collector-in-pypy.html

[11] „PyPy: Python in Python Implementation“. Datum pristupa: 10.06.2022. Poveznica: https://foss.heptapod.net/pypy/pypy/-/tree/branch/default/

[12] Ozsvald, I., Gorelick, M. „High Performance Python: Practical Performant Programming for Humans“. Drugo izdanje. O'Reilly Media. 2020.

[13] PyPy dokumentacija. Poveznica: https://doc.pypy.org/en/latest/#

[14] RPython dokumentacija. Poveznica: https://rpython.readthedocs.io/en/latest/

[15] Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D. „RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages“. In Proceedings of the 2007 symposium on Dynamic languages (DLS '07). 2007., str. 53-64

[16] Bolz, C.F. „Meta-Tracing Just-in-Time Compilation for RPython“. Disertacija. 2014. Poveznica: https://docserv.uni-duesseldorf.de/servlets/DerivateServlet/Derivate-32715/meta-tracing-archived.pdf

[17] Skvortsov, V. „Python Behind the Scenes #2: How the CPython Compiler Works“. Datum pristupa: 03.06.2022. Poveznica: https://tenthousandmeters.com/blog/python-behind-the-scenes-2-how-the-cpython-compiler-works/

[18] Skvortsov, V. „Python Behind the Scenes #1: How the CPython VM works“. Datum pristupa: 03.06.2022. Poveznica: https://tenthousandmeters.com/blog/python-behind-the-scenes-1-how-the-cpython-vm-works/

[19] PyPy benchmarks. Poveznica: https://foss.heptapod.net/pypy/benchmarks/-/tree/branch/default/. Datum pristupa: 11.05.2022.

[20] Scikit-Learn examples. Poveznica: https://scikit-learn.org/stable/auto_examples/index.html. Datum pristupa: 12.05.2022.

[21] N-body simulacija. Poveznica: https://github.com/brandones/n-body/blob/master/sim_mpl.py. Datum pristupa: 14.05.2022.

Sažetak

Zahvaljujući svojoj jednostavnosti i prilagodljivosti, programski jezik Python stekao je veliku popularnost tijekom posljednjih godina. Mnoštvo vrlina utjecalo je na širenje njegovog područja primjene, što ga je svrstalo u skupinu najkorištenijih programskih jezika. Međutim, zbog načina izvršavanja programa i dinamičke prirode, Python je sporiji u odnosu na jezike kao što su C, C++, Java i sl. Problem sporijeg izvršavanja potaknuo je razvoj mnogih rješenja, s naglaskom na razvoj novih Python prevoditelja među kojima se istaknuo PyPy. U okviru ovog rada predstavljene su arhitektura i obilježja prevoditelja PyPy. Nadalje, provedena je analiza učinkovitosti izvođenja programa koristeći standardnu Python implementaciju (CPython) i implementaciju PyPy. Na temelju zapažanja oblikovani su zaključci o upotrebi pojedinog prevoditelja ovisno o obilježjima programa.

Ključne riječi: Python, CPython, PyPy, analiza učinkovitosti, kompilator u trenutku izvođenja

Summary

Thanks to its simplicity and adaptability, the Python programming language has gained great popularity in recent years. Many virtues influenced the expansion of its field of application, which made it one of the most used programming languages. However, due to program execution and dynamic nature, Python is slower compared to languages such as C, C ++, Java, etc. The problem of slower execution has prompted the development of many solutions, with emphasis on the development of new Python interpreters, including PyPy. In this thesis, the architecture and features of the PyPy interpreter are presented. Furthermore, performance analysis of program execution was performed using the standard Python implementation (CPython) and the PyPy implementation. Based on the observations, conclusions were drawn about the use of each interpreter depending on the characteristics of the program.

Key words: Python, CPython, PyPy, performance analysis, just-in-time compiler

Popis tablica

Popis slika