УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Марко Поповић

Напредни алгоритам за распоређивање трансакција са избегавањем конфликата

МАСТЕР РАД

Нови Сад, 2017. година

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски – мастер рад
Аутор, **АУ**:		**Марко Поповић**
Ментор, **МН**:		**проф. др Илија Башичевић**
Наслов рада, **НР**:		**Напредни алгоритам за распоређивање трансакција са избегавањем конфликата**
Језик публикације, **ЈП**:		Српски / латиница
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		**2017**
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		**6/55/14/2/15/0/1**
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кqучне речи, **ПО**:		**паралелно програмирање, трансакционе меморије, алгоритми, распоређивање трансакција**
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		**У овом раду је развијен напредни алгоритам за распоређивање TM трансакција са избегавањем конфликата (тзв. AAC алгоритам), који превазилази недостатке предходно развијеног AC алгоритма, и који је у стању да увек пронађе распоред трансакција без конфликта са минималним распоном. Додатно, нови AAC алгоритам је валидиран, и његова перформанса је упоређена са перформансом RR, ETLB и AC алгоритама на процесору са четири језгара, и на бази тест оптерећења разних нивоа конкуренције.**
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	**Растислав Струхарик**
	Члан:	**Миодраг Ђукић**
	Члан, ментор:	**Илија Башичевић**

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		**Marko Popović**	
Mentor, **MN**:		**Prof. Ilija Bašičević, PhD**	
Title, **TI**:		**Advanced algorithm for scheduling TM transactions with conflict avoidance**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		**2017**	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		**6/55/14/2/15/0/1**	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S**/**KW**:		**parallel programming, transactional memories, algorithms, transaction scheduling**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**In this paper we developed the AAC algorithm, which surpasses the limitations of the previously developed AC algorithm, and which always finds the transaction schedule without conflicts and with minimal span. Additionally, new AAC algorithm was validated, and its performance was compared with the performance of RR, ETLB, and AC algorithms on a processor with four cores, and by using the test loads of various levels of concurrency.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	**Rastislav Struharik**	
	Member:	**Miodrag Đukić**	Menthor's sign
	Member, Mentor:	**Ilija Bašičević**	

Zahvalnost

Zahvaljujem se svojoj porodici, malobrojnoj rodbini i malobrojnim prijateljima koji su bili uz mene mnogo pre, pa i u pisanju ovog rada.

Posebno se zahvaljujem mome ocu, prof. Popoviću i asistentu Branislavu Kordiću za svesrdnu pomoć, bez koje ovo sve ne bi bilo moguće.

SADRŽAJ

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

Skraćenica	Opis
STM	- *Software Transaction Memory*, Softverska transakciona memorija
PSTM	- *Python Software Transaction Memory*, Python softverska transakciona memorija
RR	- *Round Robin algoritam*, Algoritam redom u krug
ETLB	- *Execution Time based Load Balancing algorithm,* Algoritam na bazi procene vremena izvršenja
AC	- *Avoid Conflicts algorithm*, Algoritam za izbegavanje konflikata
AAC	- *Advanced Avoid Conflicts algorithm*, Napredni algoritam za izbegavanje konflikata
RPC	- *Remote Procedure Call*, Poziv udaljene procedure
API	- *Application Programming Interface*, Sprega na aplikativnom nivou

1. Uvod

U ovom radu je razvijen napredni algoritam za raspoređivanje TM transakcija sa izbegavanjem konflikata (tzv. AAC algoritam), koji prevazilazi nedostatke prethodno razvijenog AC algoritma, i koji je u stanju da uvek pronađe raspored transakcija bez konflikta sa minimalnim rasponom. Naime, prethodno razvijeni AC algoritam proveri da li je moguće sledeću transakciju rasporediti na radnika sa najmanjim opterećenjem. Ako to ne dovodi konflikta, onda rasporedi sledeću transakciju na radnika sa najmanjim opterećenjem. U suprotom slučaju, odmah odustaje od dalje analize i konzervativno rasporedi sledeću transakciju na radnika sa najvećim opterećenjem.

Za razliku AC algoritma, novi AAC algoritam ne odustaje od dalje analize, već je nastavlja, tako što redom proverava sve sledeće radnike, u rastućem redosledu njihovih opterećenja (od radnika sa najmanjim opterećenjem do radnika sa najvećim opterećenjem), i pronalazi radnika sa najmanjim opterećenjem na kog je moguće rasporediti sledeću transakciju bez konflikta sa već raspoređenim transakcijama. Na taj način AAC algoritam uvek pronalazi raspored bez konflikta sa minimalnim rasponom.

Dodatno, u ovom radu je validiran novi AAC algoritam, i njegova performansa je upoređena sa performansom RR, ETLB i AC algoritama na procesoru sa četiri jezgra, i na bazi test opterećenja raznih nivoa konkurencije. Početni eksperimentalni rezultati prikazani u ovom radu ukazuju da je, u slučaju male konkurencije i tri procesa radnika, relativno ubrzanje pojedinih algoritama u odnosu na RR algoritam sledeće: za ETLB algoritma je u proseku 0.84x, za AC algoritam je u proseku 1.11x, a za AAC algoritam je 1.30x. Alternativno, eksperimenti pokazuju da u slučaju veoma kratkih transakcija bez konkurencije sva četiri algoritma imaju slične performanse.

Tekst ovog master rada je organizovan na sledeći način. U drugom poglavlju su date teorijske osnove rada. U trećem poglavlju je uveden novi AAC algoritam za raspoređivanje

transakcija. U četvrtom poglavlju je opisana implementacija novog AAC algortima. U petom poglavlju je data eksperimentalna evaluacija novog AAC algortima. U šestom poglavlju je dat zaključak rada. U sedom poglavlju je naveden spisak korišćene literature. Na kraju, u dodatku A, su dati primeri PSTM-zasnovanih programa.

.

2. Teorijske osnove rada

Osmišljena na temelju koncepta transakcija u bazama podataka, Transakciona Memorija (TM) je mehanizam koji zamenjuje konvencionalne brave (locks) sa transakcijama na višejezgrnim procesorima [1]. Iako su neke od teorijskih osnova TM već utemeljene [2], TM je i dalje oblast koja privlači veliki istraživački interes i u nauci i u industriji. Vodeće firme iz industrije, poput IBM i Intel su već obezbedile sinhronizaciju baziranu na TM na svojim savremenim višejezgarnim procesorima, npr. na arhitekturama IMB Blue Gene/Q, zEnterprise, EC12 i Power 8 arhitekture i arhitekturi Intel Haswell. Pored hardverskih TM, istraživači su razvili mnoge Softverske TM (STM) sa različitim API i različitim semantikama. Ipak, glavno otvoreno pitanje za sve vrste TM je njihova performansa koja može da se dramatično smanji pri velikoj konkurenciji između transakcija.

Raspoređivanje transakcija ili rukovanje konkurencijom (contention management) su problemi koji se naširoko izučavaju u literaturi koja se tiče višejezgarnih sistema. Pored nekoliko algoritama za raspoređivanje sa dokazanim gornjim i donjim granicama, kao i nekim nemogućim rezultatima datim u [3], [4], [5], postoje i algoritami, koji su evaluirani isključivo eksperimentalno [6] i [7]. Ipak, pored svih ovih i drugih doprinosa, raspoređivanje transakcija i dalje predstavlja aktuelnu oblast istraživanja.

Osmisliti dobar algoritam raspoređivanja za STM je izazov zato što on mora da zadovolji sledeće uslove: (i) brzina uporediva sa brzinom algoritma redom-u-krug (ii) dobro uravnoteženje opterećenja procesnih elemenata, i (iii) mala konkurencija između transakcija. Kao odgovor na ovaj izazov, u radu [8] su razvijena tri algoritma za raspoređivanje transakcija za Python STM (PSTM) [9], tačnije: Algoritam redom u krug (Round Robin algorithm, RR), (ii) Algoritam na bazi procene vremena izvršenja transakcije (Execution Time Based Load Balancing algorithm, ETLB) i Algoritam za izbegavanje konflikata (Avoid Conflicts algorithm, AC).

Međutim, iako se AC algoritam pokazao kao najbolji, on nije u stanju da uvek pronađe optimalan raspored transakcija bez konflikta. Naime, AC algoritam najpre pokuša da sledeću transakciju rasporedi na proces radnika sa najmanjim opterećenjem, ali ako to dovodi do konflikta, AC algoritam odmah odustaje i raspoređuje sledeću transakciju na radnika sa najvećim opterećenjem.

U ovom radu je razvijen napredni AC algoritam (Advanced Avoid Conflict algoritm, AAC) koji prevazilazi ovo ograničenje AC algoritma, i koji je u stanju da uvek pronađe raspored transakcija bez konflikta sa minimalnim rasponom.

U nastavku ovog poglavlja opisana je PSTM i predhodno razvijeni algoritmi raspoređivanja transakcija.

2.1 Python Softverska Transakciona Memorija

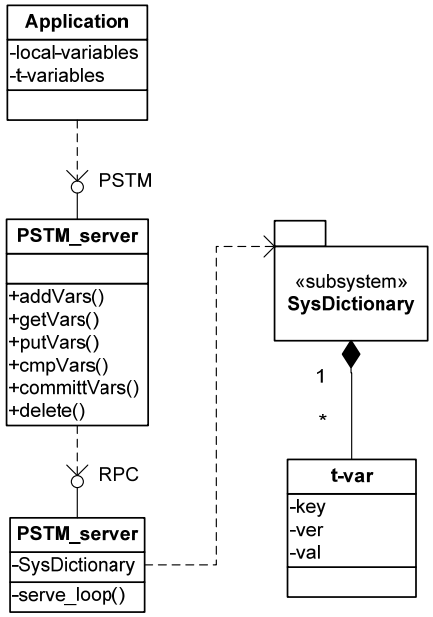
U ovom odeljku predstavljena je arhitektura, aplikativna programska sprega (eng. Application Programming Interface - API) i implementacija Python Softverske Transakcione Memorije (PSTM).

2.1.1 PSTM arhitektura

Pregled PSTM arhitekture dat je na Slici 2.1 [9]. STM-zasnovana aplikacija sadrži skup transakcija koje rade nad lokalnim promenljivama, tj. kopijama transakcionih promenljivih (t-promenljive) koje su smeštene u softverskoj transakcionoj memoriji. Transakcije dobavljaju svoje kopije t-promenljivih na početku izvršavanja, obavljaju obradu nad njima i na kraju izvršavanja mogu da ažuriraju neke od njih. Komunikacija između transakcija i PSTM se obavlja preko API sprege PSTM. API sprega PSTM je skup javnih funkcija definisanih u PSTM modulu. API funkcije zahtevaju servis od PSTM servera preko sprege za poziv udaljene procedure (eng. Remote Procedure Call - RPC).

PSTM server je definisan u istom PSTM modulu, zajedno sa API funkcijama, i vlasnik je sistemskog rečnika, koji sadrži trenutne stavke za t-promenljive, koje su korišćene od strane svih transakcija. Stavka rečnika je trojka (*key*, *ver*, *val*), gde je *key* ključ t-promenljive, *ver* je trenutna verzija t-promenljive i *val* je trenutna vrednost t-promenljive. PSTM server opslužuje nadolazeće zahteve automatski – prima poruku, odredi zahtevanu obradu, i šalje odgovarajući odgovor. To je ključna ideja iza ove arhitekture.

PSTM arhitektura je tipičan predstavnik arhitekture tipa klijent - server. Klijent se sastoji iz dva dela. Prvi deo je sama STM-zasnovana aplikacija. Drugi deo je skup API funkcija definisanih u PSTM modulu, koji sa arhitektonskog gledišta može da se gleda kao klijentov zastupnik (eng. proxy). PSTM server je proces koji izvršava beskonačnu petlju sa odgovarajućim slučajevima za razne vrste zahteva, gde svaki zahtev odgovara određenoj PSTM API funkciji.



Slika 2.1: PSTM arhitektura

U suštini RPC sprega je mehanizam koji serializuje asinhrone zahteve koji dolaze od skupa istovremenih transakcija. Ova sprega treba da minimizira korišćenje kritičnih sekcija da bi maksimizizovao performansu sistema. Detalji implementacije RPC srege za dve varijante implementacije PSTM opisane su u odeljku 2.1.3.

2.1.2 PSTM API

PSTM API obezbeđuje sledeće funkcije:

1. *addVars*(*q*, *keys*)
2. *getVars*(*q*, *keys*)
3. *putVars*(*q*, *vars*)
4. *cmpVars*(*q*, *vars*)
5. *commitVars*(*q*, *read_write*)

U listi gore, *q* je signalizirajuća putanja, koja se koristi za prenos poruka između klijenta (tj. transakcije) i PSTM servera, *keys* je lista ključeva t-promenljivih, *vars* je lista stavki rečnika i *read_write* je lista parova <*read*, *write*>, gde je *read* lista stavki rečnika koje je klijent čitao, a *write* je lista stavki rečnika u koje je klijent i upisivao. Radi jednostavnosti, u ovom radu se termini *stavka rečnika* i *t*-promenljiva smatraju za sinonime (zato što svaka stavka rečnika sadrži trenutnu verziju i trenutnu vrednost od odgovarajuće t-promenljive), a radi sažetijeg izlaganja, nekada se jednostavno nazivaju t-promenljive.

Funkcija *addVars* uvodi novu t-promenljivu u sistemski rečnik. Svaka t-promenljiva mora biti uvedena u rečnik pre nego što se može koristiti. Ova funkcija je uvek uspešna, zato što

uvođenje t-promenljivih koje su već uvedene se ne smatra za grešku (takvi zahtevi se jednostavno ignorišu i odgovarajuće t-promenljive zadržavaju svoje verzije).

Funkcija *getVars* vraća listu parova (*present*, *data*) za zadatu listu ključeva, gde *present* ima vrednost **true** ako je stavka rečnika nađena u rečniku (inače je **false**), a *data* je torka (*ver*, val) koja odgovara zadatom ključu *key*. Poziv ove funkcije zapravo predstavlja početak transakcije.

Funkcija *putVars* pokušava da ažurira (upiše) listu t-promenljivih. Pokušaj će biti uspešan samo ako su sve t-promenljive u *vars* još uivek ažurne, tj. ako su njihove verzije (*ver*) jednake verzijama odgovarajućih stavki PSTM rečnika, i u tom slučaju funkcija vraća 'yes' (inače 'no'). Poziv ove funkcije može biti korišćen za realizaciju operacije *vidljivog upisa* (eng. visible write) u t-promenljivu, kao npr. u DSTM [12] i TinySTM [13], i/ili samo kao operacija ažuriranja (eng. update only commit) u slučaju da transakcija samo ažurira neke t-promenljive. U ovom drugom slučaju, poziv ove funkcije zapravo predstavlja kraj transakcije.

Funkcija *cmpVars* je pomoćna funkcija koja vraća 'yes' ako su sve t-promenljive u zadatoj listi još uvek ažurne. Ova pomoćna funkcija može biti korisna za realizaciju nekih strategija za izbegavanje konflikata na nivou aplikacije.

Funkcija *commitVars* pokušava da ažurira transakciju koja je radila nad skupom t-promenljivih koje su čitane (koji može biti prazan) i skupom t-promenljivih u koje je upisivano (koji može biti prazan). Slično funkciji *putVars*, i ova funkcija će biti uspešna samo ako sve verzije, svih t-promenljivih (u oba skupa) imaju istu verziju kao odgovarajuće stavke u PSTM rečniku. Uspeh se opet prikazuje sa povratnom vrednošću 'yes', dok je neuspeh prikazan sa vrednošću 'no'. Poziv funkcje *commitVars* uvek predstavlja kraj transakcije.

Napomena: PSTM API bez funkcije *putVars* je kompatibilan sa vrstom STM kod koje je operacija upisa u t-promenljivu nevidljiva između transakcija (eng. invisible write), dok je celokupan PSTM API poredliv sa vrstom STM kod koje je operacija upisa u t-promenljivu vidljiva između transakcija (eng. visible write).

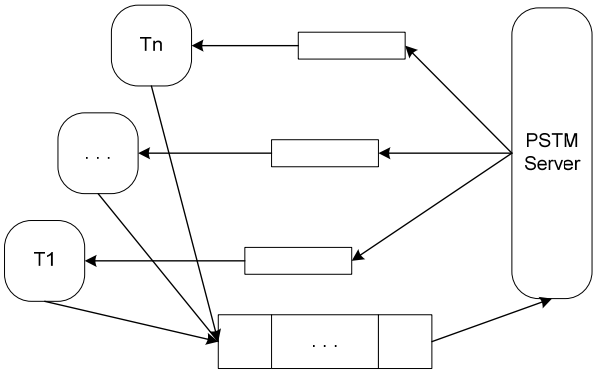
2.1.3 PSTM Implementacija

RPC sprega se pokazala kao glavni izazov u implementaciji PSTM projekta, koji je predstavljen u predhodnom odeljku. Implementacija ove sprege je zahtevna zato što bi idealno zaključavanje trebalo izbegavati, ili makar minimizirati. Najpre je bila implementirana varijanta PSTM prototipa (PSTM-PT) koja je bila zasnovana na Python *Queue* i *Pipe* mehanizmima.

Glavna svrha PSTM-PT bila je da potvrdi koncept PSTM i da može biti korišćen kao osnovna verzija (eng. baseline) za sve buduće optimizovane verzije PSTM. Pored toga, kao prva

raspoloživa implementacija PSTM arhitekture, omogućila je programerima test programa (eng. benchmarks) i PSTM-zasnovanih aplikacija da sprovedu svoja rana istraživanja i pokušaje.

PSTM-PT je zasnovana na Python 3.0 apstrakcijama reda (eng. queue) i cevi (eng. pipe), vidi sliku 2.2. Jedan red poruka se koristi za serijalizaciju i prebacivanje svih poruka (zahteva) od transakcija ka PSTM serveru, preko veze tipa više-tačaka-jedna-tačka (eng. multipoint to point). Odgovori od servera ka klijentima se šalju koristeći skup cevi, tako da svaki klijent ima svoju cev za dobijanje odgovora. Tako da, svaki odgovor se šalje preko individualne veze tipa tačka-tačka (eng. point to point).



Slika 2.2: RPC sprega u PSTM-PT implementaciji.

Očigledno, jedan red je ključ za atomičnost transakcije, ali po ceni uvođenja dodatne sistemske režije vezane za rukovanje redom poruka (zahteva). Ipak, glavno usko grlo za PSTM arhitekturu je definitivno sam PSTM server zato što mora da opsluži svakog klijenta u datoj iteraciji.

2.2 Arhitektura i algoritmi raspoređivača transakcija

U ovom odeljku opisana je arhitektura raspoređivača transakcija i prethodno razvijeni algoritmi za raspoređivanje transakcija: RR algoritam, ETLB algoritam i AC algoritam - vidi [8].

2.2.1 Arhitektura raspoređivača i njene operacije

Neka je T transakcija koja je definisana od strane aplikacije**:**

$$T = (f, V). \tag{1}$$

gde je f funkcija koja je dodeljena transakciji, a V definiše t-promenljive korišćene od strane aplikacije:

$$T = (R, W). \tag{2}$$

gde je R niz t-promenljivih koje T samo čita, a W je niz t-promenljivih u koje T piše (a možda ih nekad i čita).

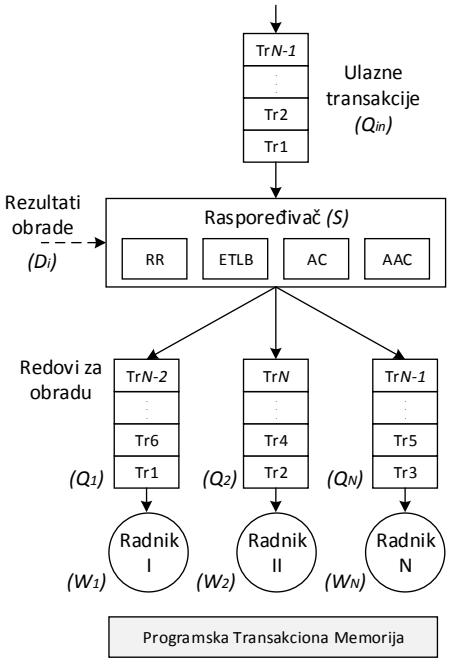
Arhitektura sistema sadrži proces raspoređivača S i n procesa radnika W_i, koji su dodeljeni na n raspoloživih procesnih elemenata, vidi Sliku 2.3. Nove transakcije se iz aplikacije ulančavaju u ulazni red Q_{in}. Raspoređivač S radi u fazama, pri čemu se svaka faza sastoji od dve runde.

Tokom prve runde, raspoređivač S ponavlja sledeće korake dok se ulazni red Q_{in} ne isprazni:

1. Izlančaj transakciju sa početka ulaznog reda Q_{in}.
2. Rasporedi izlančanu transakciju korišćenjem konfigurisanog algoritma raspoređivanja. Povratna vrednost algoritma raspoređivaja je indeks i radnika W_i kome treba da se ulančana raspoređivana transakcija.
3. Ulančaj transakciju na kraj reda Q_i pridruženog radniku W_i.

Tokom druge runde, raspoređivač S čeka da sve raspoređene transakcije budu izvršene od radnika. Ovo radi tako što čeka na signale *done* koji radnici W_i šalju u svoje izlazne redove D_i, gde je $i = 1, \ldots, n$.

Dok raspoređivač čeka, svaki radnik W_i izvršava redom transakcije $Tj = (f_j, V)$ iz njegovog reda Q_i, tako što za svaku T_j poziva odgovarajuću funkciju f_j. Ako je transakcija T_j neuspešno završena (abort), radnik W_i rasporedi tu T_j na kraj ulaznog reda Q_{in}. Inače, uzima sledeću transakciju iz reda Q_i. Kad ulazni red Q_i postane prazan, radnik W_i ulanča signal *done* da je završio obradu transakcija u izlazni red Q_i.



Slika 2.3: Arhitektura sistema za raspoređivanje i obradu transakcija.

Na kraju ovog kratkog opisa uopštene arhitekture raspoređivača transakcija, dobro je pokazati i njena ograničenja. Na način kako je objašnjen rad arhitekture, čini se da se arhitektura raspoređivača transakcija nalazi direktno iznad fizičkog sloja mašine (hardver), zato što smo rekli da je *n* procesa radnika pridruženo sa *n* procesnih elemenata. Raspoređivački algoritmi opisani u ovom radu su razvijeni na osnovima te apstrakcije.

Iako ova apstrakcija pomaže kad se radi sa raspoređivačkim algoritmima, ona je u stvari implementirana na aplikacionom nivou programa, koji se implementira u Python programskom jeziku, a koji u stvarnosti radi na operativnom sistemu (npr. OS Windows). Zbog ovoga realno vreme izvršavanja može malo da varira od idealnog vremena izvršenja (bez smetnji OS i Python izvršne biblioteke). Ako dođe do smetnji, ove varijacije mogu biti viđene kao varijacije vremena početka izvršenja transakcija i vremena njihovog trajanja.

2.2.2 RR algoritam

U suštini, RR algoritam ne zna karakteristike transakcija kao što su dužina izvršenja transakcija ili informacije o transakcionim promenljivama. Pošto ne poseduje nikakve informacije, RR algoritam koristi pristup crne kutije i tretira sve transakcije podjednako.

Cilj RR algoritma je da za dati radni teret (workload) obezbedi uravnoteženje opterećenja raspoloživih procesnih elemenata. Pošto nema informacije o ulaznim transakcijama, on optimistično pretpostavlja da sve imaju isto trajanje i da su bez konflikata. Na ove pretpostavke, on jednostavno koristi tradicionalno raspoređivanje redom u krug da bi se ostvario zadati cilj.

Implementacija RR algoritma je prilično jednostavna – on koristi promenljive *n* i *i* koje smo već uveli. Algoritam računa novu vrednost od *i* tako što je poveća za 1, po modulu *n*, i vrati prethodnu verziju od *i*. Raspoređivač *S* onda koristi vraćenu verziju od *i* da rasporedi sledeću transakciju, videti korak 3, u tački 2.2.1.

2.2.3 ETLB algoritam

ETLB algoritam je pohlepan raspoređivački algoritam, koji koristi metodu procenjivanja vremena izvršenja na bazi log-normalne distribucije [10]. Metoda za procenu vremena izvršenja transakcije koristi jednostavan pristup mašinskog učenja. Operacija radi na klizećem prozoru odbiraka od prethodnih izvršenja, koji se mere tokom izvršenja aplikacije. Za svaki sledeći odbirak, metoda ažurira klizeći prozor odbiraka i pridružene interne podatke, tako da može da proceni vrednost sledećeg vremena izvršenja kada je to potrebno.

ETLB algoritam pretpostavlja da će aplikacija izvesti inicijalnu kalibraciju mereći vremena izvršenja svakog tipa transakcije, popunjavajući inicijalne klizajuće prozore sa inicijalnim odbircima. Broj inicijalnih odbiraka je jednak širini klizećeg prozora. Radeći inicijalnu

kalibraciju, metoda uči inicijalno ponašanje svih tipova transakcija, i može trenutno obezbedi procenjeno vreme izvršavanja kada je zatraženo od procesa radnika W_i.

Na dalje, tokom normalnog rada algoritma, procesi radnici mere vremena izvršenja transakcija i ažuriraju odgovarajući klizajući prozor novim odbircima. Tako da arhitekruta raspoređivača transakcija predstavlja adaptivan algoritam. Kada se pravo vreme izvršenja za određenu transakciju menja tokom vremena (zbog promena u izvršenjom okruženju), procenjeno vreme izvršenja prati ove promene.

Neka L_i bude teret (kumulativno procenjeno vreme izvršenja) dodeljen procesu radniku W_i. ETLB algoritam koristi trenutne vrednosti L_i gde je $i = 1,\ldots, n$, i procenjeno vreme izvršenja sledeće transakcije t_e iz ulaznog reda Q_{in} da bi raporedio sledeću transakciju na najmanje opterećenog radnika. Na taj način ETLB algoritam minimizira raspon (span), odnosno maksimizira propusnost.

Implementacija ETLB algoritma koristi promenljive n, i, t_e i L_i, gde je $i = 1,\ldots, n$, koje su već uvedene. Algoritam pronalazi indeks i sa minimalnom vrednošću, doda t_e na L_i i vrati vrednost i. Raspoređivač S onda koristi vraćenu verziju od i da rasporedi sledeću transakciju, videti korak 3, u tački 2.2.1

2.2.4 AC algoritam

Na AC algoritam se može gledati kao prošireni ETLB algoritam. AC algoritam kao i ETLB algoritam koristi metodu za određivanje vremena izvršenja transakcija. Za razliku od ETLB algoritma, AC algoritam proverava mogući konflikt između sledeće transakcije iz ulaznog reda Q_{in} i trenutno raspoređene transakcije.

Neka F bude raspoređena transakcija:

$$F = (t_b, t_e, V). \tag{3}$$

gde je t_b vreme početka izvršenja F, t_e je vreme završetka izvršenja F, a V je informacija o t-promenljivama koje F koristi kao što je definisano u (1). Neka je $F_1 = (tb_1, te_1, V_1)$ već raspoređena transakcija i neka je $F_2 = (tb_2, te_2, V_2)$ sledeća transakcija iz Q_{in} koju treba rasporediti.

U ovom radu se pretpostavlja da svaka transakcija F_i počinje sa čitanjem neke od svojih t-promenljive i završava sa operacijom ažuriranja (eng. commit), tako da Fi koristi svoje t-promenljive tokom celog svog životnog ciklusa $[t_{bi}, t_{ei}]$. Dakle, za STM, poput PSTM, nema potrebe da se prati vreme izvršenja za svaku promenljivu. Ukidanje ove pretpostavke i prilagođavanje AC algoritma za druge STM memorije je predmet našeg budućeg rada.

Generalno, kad se F_i i F_j izvršavaju na optimističnoj STM, sa odloženom detekcijom konflikta (eng. lazy conflict detection) poput PSTM, kažemo da F_i napada F_j, ako se njihovo

izvršenje preklopi u vremenu i ako se F_i završi pre F_j, zato što će se F_i završiti uspešno (*commited*), dok će se F_j završiti neuspešno (*abort*). Ovaj napad je uvek uspešan, ali F_j otkriva da je neuspešna (eng. abort) u fazi ažuriranja. Teorijski, i F_i i F_j mogu napasti jedna drugu, ako se završe u isto vreme, ali u realnosti PSTM serijalizuje sve zahteve za ažuriranjem, tako da će jedna od ovih transakcija biti uspešna, na suprot drugoj, koja će biti neuspešna.

Transakcije F_1 i F_2 su u konfliktu ako: (i) F_1 napada F_2, ili obrnuto, i ako (ii) Bernštajnovi uslovi [11] nisu zadovoljeni (npr. postoji trka za podacima između F_1 i F_2). Preciznije, F_1 napada F_2, ukoliko:

$$t_{b2} < t_{e1} \leq t_{e2}. \tag{4}$$

F_2 napada F_1 ako:

$$t_{b1} < t_{e2} \leq t_{e1}. \tag{5}$$

Bernštajnovi uslovi za F_1 i F_2 su sledeći:

$$R_1 \cap W_2 = \{\} \tag{6a}$$

$$R_2 \cap W_1 = \{\}. \tag{6b}$$

$$W_1 \cap W_2 = \{\} \tag{6c}$$

Sledeće, definišemo neuravnoteženost opterećenja L_u. Neka L_{min} i L_{max} budu minimalni L_i i maksimalan L_i za $i = 1,..., n$, respektivno. Neuravnoteženost je definisana kao:

$$L_u = (L_{\max} - L_{\min}) / L_{\max}. \tag{7}$$

Primarni cilj AC algoritma je da pokuša da izbegne konflikt između raspoređenih transakcija, ali izbegavanje konflikata može da rezultuje serijalizacijom transakcija koje bi dovele od značajne neuravnoteženosti opterećenja. Dokle god je L_u manji od L_{ut}, algoritam nastoji da izbegava konflikte. U suprotnom slučaju, algoritam odustaje od izbegavanja konflikata i dalje se ponaša kao ETLB algoritam.

Kada je L_u manje od L_{ut}, AC algoritam izbegava konflikte tako što raspoređuje sledeću transakciju na kraj tekućeg rasporeda, tj. dodeljuje sledeću transakciju radniku sa L_{max}. Ovaj konzervativan pristup je najjednostavnije rešenje, ali ima svoja ograničenja. Novi AAC algoritam predstavljen u ovom radu je projektovan upravo sa ciljem da se ta ograničenja eliminišu.

Implentacija AC algoritma koristi promenljive koje su već uvedene. Uvodi se nova promenljiva t_d – trajanje transakcije, kao i redovi raspoređenih transakcija F za svaki proces radnika, QF_i. Kada je $L_u < L_{ut}$, algoritam izvršava sledeće korake da izračuna ciljni indeks i:

1. Nađe indekse i_{min} i i_{max} koji odgovaraju L_{min} i L_{max}, respektivno.
2. Postavi T_2 na sledeću transakciju iz Q_{in}.

3. Odredi procenjeno vreme izvršenja transakcije tipa f_2 i dodeli ga vrednosti t_{d2}.
4. Postavi $i = i_{min}$ i $F_2 = (L_{min}, L_{min} + t_{d2}, V2)$.
5. U petlji, za svaki F_1 od već raspoređenih transakcija, proveri da li postoji konflikt između F_1 i F_2. Prekini petlju na prvom detektovanom konfliktu.
6. Ako je konflikt detektovan u koraku 5, namesti i da je $i = i_{max}$ i $F_2 = (L_{max}, L_{min} + t_{d2}, V_2)$.
7. Ulančaj F_2 na kraj reda QF_i.
8. Vrati i.

Raspoređivač koristi povratnu vrednost od i, da rasporedi trenutnu transakciju, videti korak broj 3, u tački 2.2.1.

3. Novi AAC algoritam

U ovom poglavlju je opisan novi AAC algoritam, kao prirodno proširenje predhodno razvijenog AC algoritma. Shodno tome i sam tekst opisa AAC algortima je napravljen kao nastavak teksta opisa AC algoritma iz predhodnog poglavja, tako da sve uvedene apstrakcije, kroz jednačine u predhodnom poglavlju, važe i ovde.

3.1 AAC algoritam

U prethodnom odeljku je opisan AC algoritam. U suštini, AC algoritam odredi proces radnika sa najmanjim opterećenjem, i rasporedi sledeću transakciju na tog radnika (koraci 1 do 4). Zatim u koraku 5 proveri da li ta nova tako raspoređena transakcija ima konflikt sa već raspoređenim transakcijama na drugim radnicima. Ako postoji konflikt, onda prerasporedi novu transakciju na radnika sa najvećim opterećenjem.

Nedostatak ovako konzervativnog raspoređivanja je u tome što u slučaju da postoji više od dva radnika, neki radnik koji ima opterećenje između minimalnog i maksimalnog, neće biti upošljen iako nova transakcija može biti na njega raspoređena bez konflikata sa već prethodno raspoređenim transakcijama.

Ideja novog AAC (Advanced Avoid Conflicts) algoritma je da se prevaziđe taj nedostatak konzervativnog raspoređivanja u AC algoritmu. Ukoliko bi raspoređivanje nove transakcije na radnika sa najmanjim opterećenjem dovelo do konflikata, AAC algoritam, za razliku od AC algoritma, ne odustaje odmah, i ne raspoređuje tu transakciju na radnika sa najvećim opterećenjem, već ispituje redom sve radnike od radnika sa najmanjim opterećenjem do radnika sa najvećim opterećenjem. Kada pronađe radnika na kome se transakcija može rasporediti tako da ne izaziva konflikte, AAC algoritam rasporedi tu transakciju na tog radnika.

Prednost ovakvog načina raspoređivanja u AAC algoritmu, u odnosu na AC algoritam, je što u opštem slučaju AAC algoritam pronalazi raspored sa kraćim rasponom (eng. span; vidi

[14]), odnosno većom propusnošću (eng. throughput). U stvari, AAC algoritam pronalazi raspored sa najkraćim mogućim rasponom (teorijski dokaz za ovu tvrdnju izlazi iz okvira ovog rada).

Da bi se ova strategija raspoređivanja realizovala, potrebno je uvesti pojam mape, koja preslikava indeks radnika na njegovo opterećenje:

$$M : i \rightarrow L_i. \tag{8}$$

ili alternativno:

$$M(i) = L_i. \tag{9}$$

Mapa se konstruiše kao rečnik:

$$M = \{(0, L_0), (1, L_1), ..., (n-1, L_{n-1})\}. \tag{10}$$

Algoritam AAC se onda može opisati kroz sledeće korake:

1. Pronađi indekse i_{min} i i_{max} koji odgovaraju L_{min} i L_{max}, respektivno.
2. Postavi T_2 na sledeću transakciju iz Q_{in}.
3. Odredi procenjeno vreme izvršenja za transakciju tipa f_2 i dodeliti tu vrednost promenljivoj t_{d2}.
4. Konstruiši M.
5. Postavi $i = i_{min}$, $F_2 = (L_{min}, L_{min} + t_{d2}, V_2)$, c*onflictFound* = False
6. Iteriraj kroz M u redosledu sortiranih opterećenja radnika, koristeći iterirajući indeks i. Ako indeks i dostigne i_{max}, postavi indikator *conflictFound* na True i izađi iz petlje. Inače, za svaku već raspoređenu transakciju F_1, proveri da li postoji konflikt između F_1 i F_2. Ako nema konflikta, postavi $F_2 = (L_i, L_i + t_{d2}, V_2)$ i izađi iz petlji, inače nastavi sa izvršavanjem petlje.
7. Ako je *conflictFound* postavljen na True, postavi $i = i_{max}$ i $F_2 = (L_{max}, L_{max} + t_{d2}, V_2)$.
8. Ulančaj F_2 na kraj reda QF_i.
9. Vrati i.

3.2 Teorijska analiza rasporeda transakcija

U ovom odeljku je data analiza očekivanih rezultata sva četiri algoritama za raspoređivanje transakcija (RR algoritam, ETLB algoritam, AC algoritam i AAC algoritam), za tri vrste zadatih ulaznih radnih optetrećenja, odnosno paketa transakcija (RDW, CFW i WDW), u obliku očekivanih rasporeda transakcija, na zadati broj radnika (dva radnika, tri radnika i četiri radnika). U naredna tri pododeljka analiziraju se očekivani rezultati za tri vrste zadatih ulaznih radnih optetrećenja (RDW, CFW i WDW).

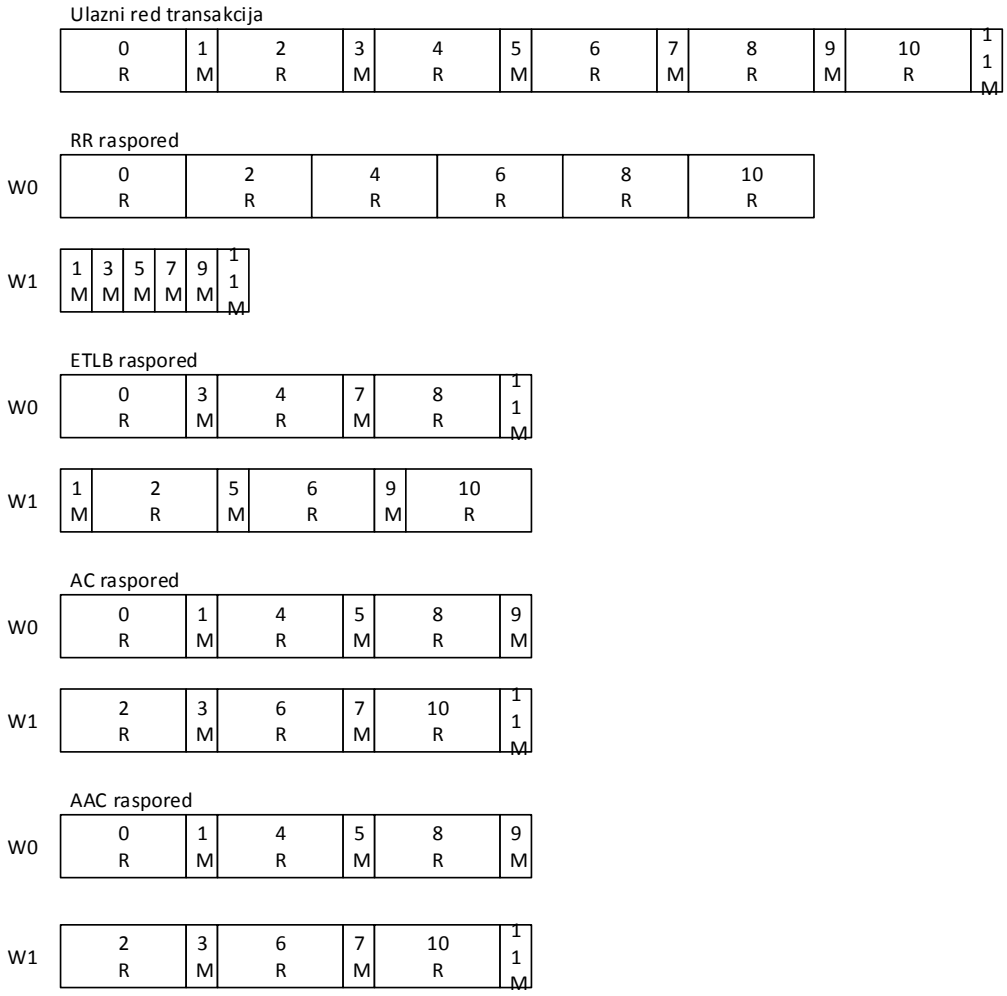
3.2.1 Analiza rasporeda za RDW radno opterećenje

Na slici 3.1 su prikazani očekivani rasporedi transakcija za sva četiri algoritma raspoređivanja transakcija (RR, ETLB, AC i AAC), za ulazno opterećenje RDW, na dva radnika. Na vrhu slike je prikazan ulazni red transakcija, koji se sastoji od niza RAA (Read All Accounts) transakcija, na slici označenih sa R, i niza MT (Money Transfer) transakcija, na slici označenih sa M, koje su poređane tako da su parne transakcije R (počevši od nulte) a neparne su transakcije M (počevši od prve). Na vrhu reda je nulta R transakcija, a na kraju reda je M transakcija.

Ispod ulaznog reda transakcija prikazan je RR raspored (to je očekivani rezultat RR algoritma). RR algoritam ne zna nista o vrsti transakcija niti o njihovom trajanju, i jednostavno raspoređuje ulazne transakcije (iz ulaznog reda) na raspoložive radnike, redom u krug. Za zadato RDW radno opterećenje, RR algoritam najpre uzme nultu R transakciju i dodeli je radniku W_0, zatim sledeću (prvu) M transakciju dodeli radniku W_1. Dalje se ovaj postupak ponavlja, drugu R transakciju dodeli radniku W_0, treću M transakciju radniku W_1, itd., desetu R transakciju radnik W_0 i jedanaestu M transakciju radniku W_1.

ETLB algoritmu je poznata procena trajanja pojedinih transakcija, ali ne i njihova vrsta (R i M), a njegov cilj je da pohlepno minimizira raspored, tako što za svaku ulaznu transakciju dodeli trenutno najmanje opterećenom radniku. Za zadato RDW radno opterećenje, ETLB algoritam najpre rasporedi nultu R transakciju na radnika W_0, pošto oba radnika imaju isto opterećenje ali radnik W_0 ima manji indeks, zatim prvu M transakciju rasporedi na radnika W_1 jer on u tom trenutku ima manje opterećenje, potom drugu R transakciju ponovo rasporedi na radnika W_1 jer on i dalje ima manje opterećenje, zatim treću M transakciju rasporedi na radnika W_0 jer on tada ima manje opterećenje. Dalje se ovaj postupak ponavlja jednom, tako da četvrta R transakcija ide na W_0, peta M transakcija ide na W_1, šesta R transakcija takođe na W_1, a sedma M transakcija na W_0, i na kraju se ponavlja još jednom (osma transakcija na radnika W_0, deveta transakcija na radnika W_1, deseta transakcija takođe na W_1, i jedanaesta transakcija na W_0).

AC i AAC algoritmi prave isti raspored u slučaju dva radnika. Oba algoritma najpre rasporede nultu R transakciju na radnika W_0. Transakcija broj jedan ne može biti raspoređena na radnika W_1 zbog konflikta sa već raspoređenom nultom R transakcijom, pa oba algoritma prvu M transakciju rasporede takođe na radnika W_0. Pošto druga R transakcija nije u konfliktu sa nultom R transakcijom, oba algoritma je rasporede na radnika W_1. Slično, pošto treća M transakcija nije u konfliktu sa prvo M transakcijom, oba algoritma rasporede treću M transakciju na radnika W_1. Dalje se ovaj postupak periodično ponavlja jednom, četvrta i peta transakcija idu na radnika W_0, a šesta i sedma transakcija idu na radnik W_1, i na kraju se ponavlja još jednom, tako da osma i deveta transakcija idu na radnika W_0, a deseta i jedanaesta transakcija idu na radnika W_1.



Slika 3.1: Rasporedi transakcija za RDW opterećenje na dva radnika.

Na slici 3.2 je prikazano raspoređivanje transakcija na 3 radnika za sva 4 algoritma raspoređivanja.

RR raspored u ovom slučaju radi po modulu tri, tj. nulta R transakcija se dodeljuje radniku W_0, prva M transakcija se dodeljuje radniku W_1, druga R transakcija se dodeljuje radniku W_2, i tako u krug; treća M transakcija se dodeljuje radniku W_0, četvrta R radniku W_1 i peta M transakcija radniku W_2, itd.

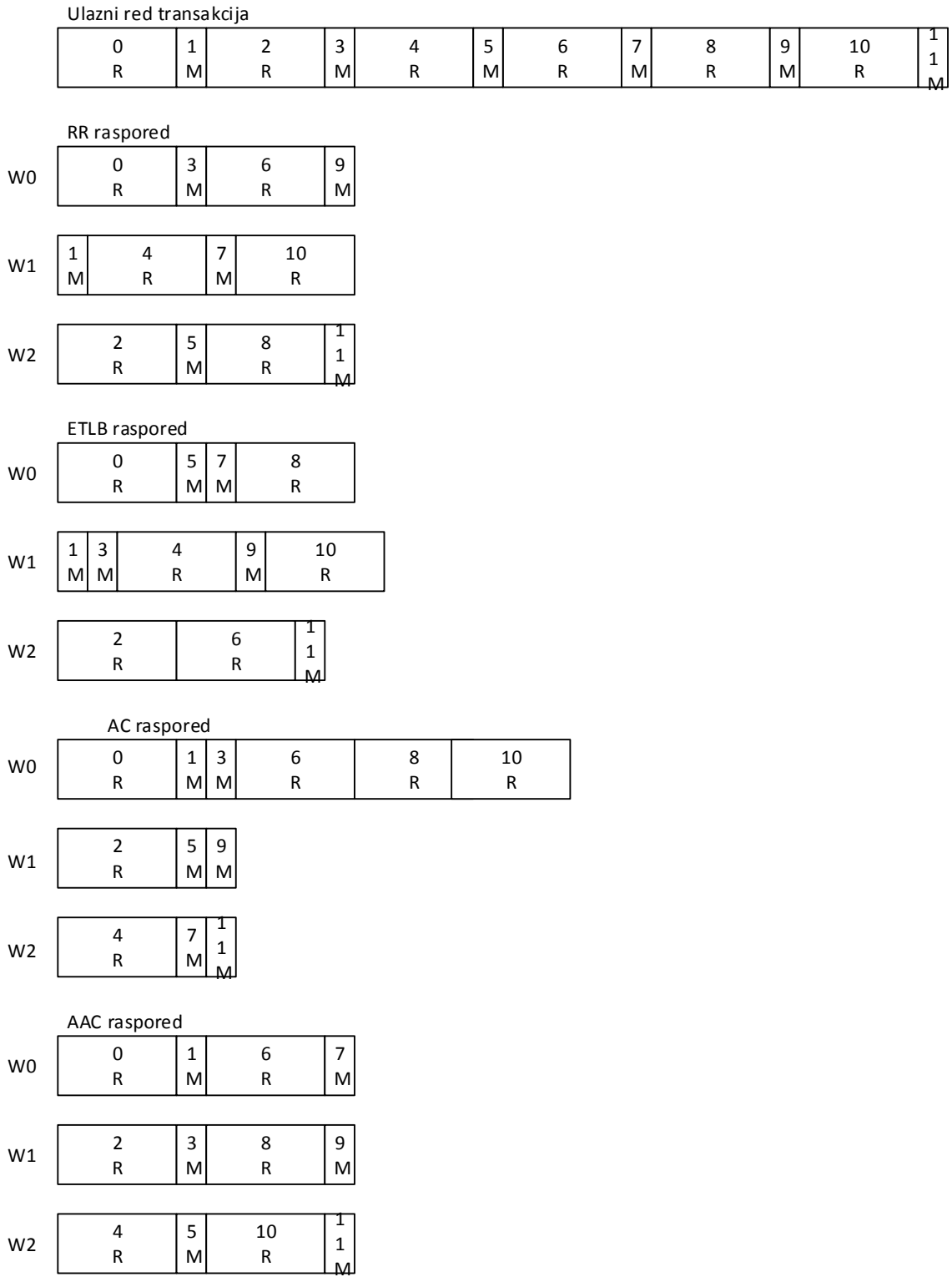
ETLB algoritmu je poznata procena trajanja pojedinih transakcija, ali ne i njihova vrsta (R i M), a njegov cilj je da pohlepno minimizira rasporeda rasporeda, tako što svaku ulaznu transakciju dodeli trenutno najmanje opterećenom radniku. Tako, u prvom krugu, nulta R transakcija se dodeljuje radniku W_0, prva M transakcija radniku W_1, druga R transakcija radniku W_2, zatim W_1 postane najmanje opterećen, pa se i njemu dodeljuje treća M transakcija, a zatim i četvrta R transakcija. Posle toga radnici W_0 i W_1 postaju najmanje opterećeni. Pošto radnik W_0 ima manji indeks, njemu se dodeli peta M transakcija. Iza toga, radink W_2 postaje najmanje opterećen, pa se njemu dodeli šesta R transakcija. Po istom principu, sedma M i osma R

transakcija idu na radnika W_0, deveta M i deseta R transakcija na radnika W_1, a jedanaesta M transakcija na radnika W_2.

AC algoritam deluje tako što traži najmanje opterećenog radnika, kada ga nađe, i dodeljuje mu tekuću transakciju ukoliko to ne dovodi do konflikta sa već raspoređenim transakcijama. Ako bi raspoređivanje tekuće transakcije na radnika sa najmanjim opterećenjem dovelo do konflikta, onda istu dodeljuje na radnika sa najvećim opterećenjem. U skladu s tim, najpre rasporedi nultu R transakciju na radnika W_0. Nakon toga, najmanje opterećen radnik sa najmanjim indeksom je radnik W_1. Pošto bi raspoređivanje prve M transakcije na tog radnika (W_1) dovelo do konflikta sa već raspoređenom nultom R transakcijom (na radniku W_0), AC algoritam rasporedi prvu M transakciju na najopterećenijeg radnika, a to je radnik W_0. Dalje, druga R transakcija ide na najmanje opterećenog radnika (sa najmanjim indeksom) W_1, pošto druga R transakcija nije u konfliktu sa već raspoređenom nultom R transakcijom (koja se istovremeno izvršava na radniku W_0). Na suprot tome, treća M transakcija ne može biti raspoređena u tom trenutku na najmanje opterećenog radnika W_2, pošto bi ona bila u konfliktu sa nultom R transakcijom na radniku W_0 i drugom R transakcijom na radniku W_1. Zbog toga, ona ide najopterećenijeg radnika, a to je u tom trenutku radnik W_0.

Po istom pricipu, četvrta R transakcija ide na radnika W_2 (jer nije u konfliktu sa nultom R i drugom R transakcijom), peta M transakcija ide na radnika W_1 (jer nije u konfliktu sa prvom M transakcijom), šesta R transakcija ne može biti raspoređena na radnika W_2 (zbog konflikta sa prvom i petom M transakcijom) pa ide na najopterećenijeg radnika W_0, sedma M transakcija ide na radnika W_2 (jer nije u konfliktu sa prvom M i petom M transakcijom), itd. Konačan rezultat je dosta nepovoljan jer su dugačke R transakcije većinom raspoređene na radnikaW_0.

AAC algoritam svaku ulaznu transakciju rasporedi na radnika sa najmanjim opterećenjem na kome to raspoređivanje ne dovodi do konflikta. Prema tome, nultu R transakciju dodeljuje radniku W_0, prvu M transakciju ne može da rasporedi ni na radnika W_1 niti na radnika W_2 jer bi to dovelo do konflikta (sa nultom R transakcijom na radnik W_0) pa je dodeljuje radniku W_0, drugu R transakciju dodeljuje radniku W_1 (jer nije u konfliktu sa već raspoređenom nultom R transakcijom), treću M transakciju dodeljuje radniku W_1 (jer nije u konfliktu sa već raspoređenom prvom M transakcijom). U ovom koraku AAC je došao do boljeg rasporeda od AC algoritma, koji nako što utvrdi da bi raspoređivanje na najmanje opterećenog radnika W_2 dovelo do konflikta, odmah odustaje od dalje pretrage radnika i treću M transakciju (pogrešno) rasporedi na radnika W_0. Na suprot tome, AAC algoritam, nakon što proveri rasporedi na radnika W_2, ne odustaje, već proverava sledećeg najmanje opterećenog radnika W_1 i utvrđuje da tamo nema konflikta. Dalje AAC algoritam, dodeljuje četvrtu R transakciju radniku W_2, petu M transakciju takođe dodeljuje radniku W_2, itd.

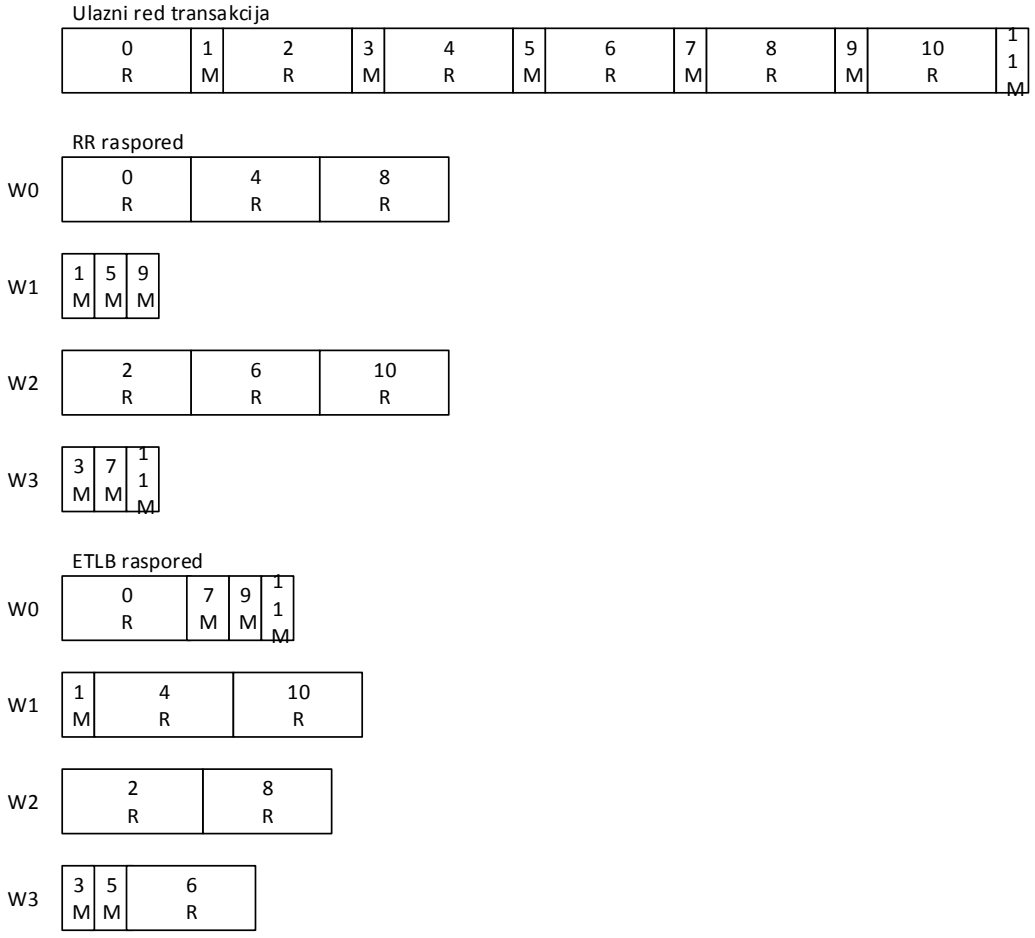


Slika 3.2: Rasporedi transakcija za RDW opterećenje na tri radnika.

Na slici 3.3 predstavljeno je raspoređivanje po RR i ETLB algoritmima za četiri radnika. RR algoritam, u ovom slučaju, radi sa modulom četiri, tako da će, u prvom krugu, nulta R transakcija biti dodeljena radniku W_0, prva M transakcija će biti dodeljena radniku W_1, druga R

transakcija će biti dodeljena W_2 i treća M transakcija će biti dodeljena radniku W_3. Zatim se postupak ponavlja, u drugom krugu, četvrta R transakcija se dodeljuje radniku W_0, peta M transakcija se dodeljuje radniku W_1 itd.

ETLB algoritam u ovom slučaju, ima više mesta za raspodelu transakcija jer ima više raspoloživih radnika. Slično kao i u prethodnim slučajevima sa dva i tri radnika, nulta R transakcija se dodeljuje radniku W_0, prva M transakcija se dodeljuje radniku W_1, druga R transakcija radniku W_2, treća M transakcija radniku W_3, zato što su na početku svi radnici bez opterećenja. Nakon toga radnici W_1 i W_3 su najmanje opterećeni, pa četvrta R transakcija ide na radnika W_1, a peta M transakcija na radnika W_3. Posle toga radnik W_3 je najmanje opterećen, pa se njemu dodeljuje šesta R transakcija, sedma M transakcija ide na radnika W_0, osma R transakcija na radnika W_2, deveta M transakcija na radnika W_0, itd.



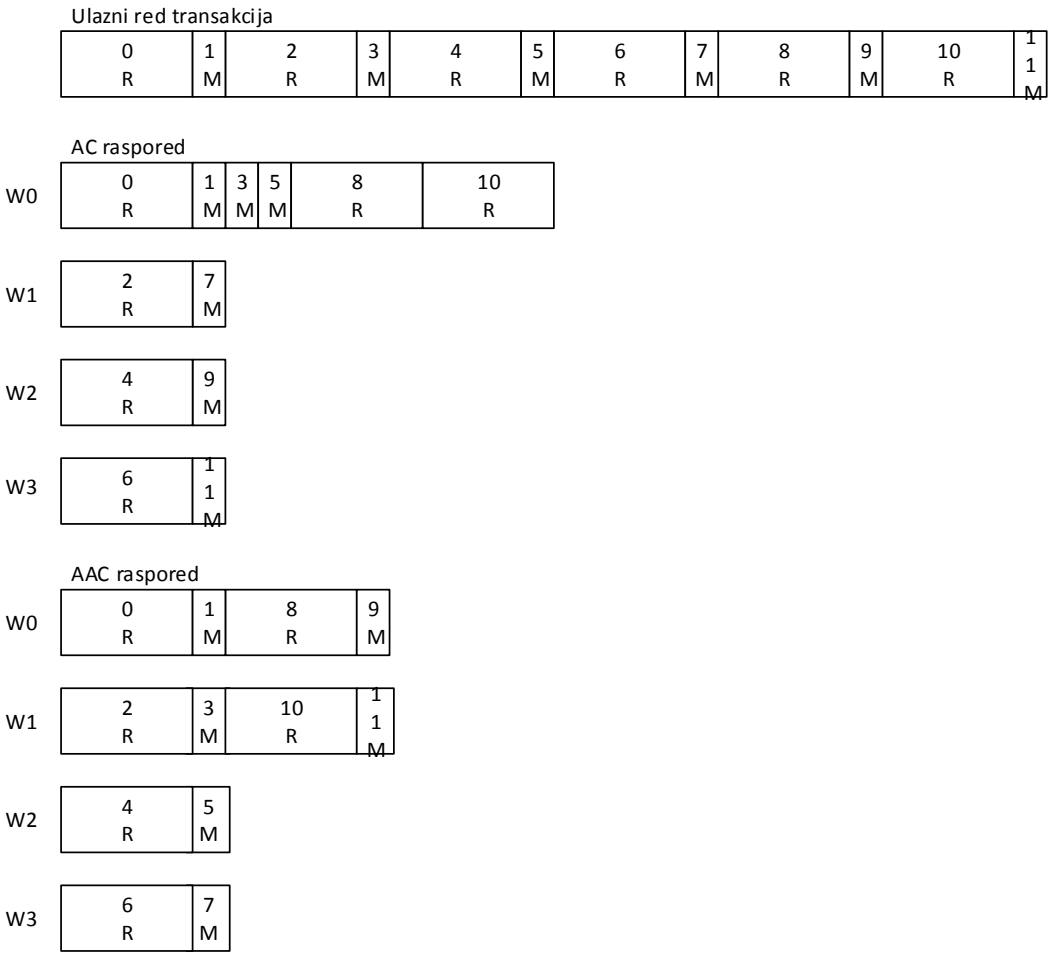
Slika 3.3: Rasporedi transakcija za RDW opterećenje na četiri radnika (RR i ETLB).

Na slici 3.4 je prikazan raspored transakcija za AC i AAC algoritme na četiri radnika.

Algoritam AC, funkcioniše slično kao u predhodnim slučajevima za dva i tri radnika: nulta R transakcija se dodeljuje radniku W_0, prva M transakcija ne može biti dodeljena radnicima W_1, W_2 i W_3, jer bi to dovelo do konflikta, pa se ona dodeljuje ponovo radniku W_0. Na suprot

tome, druga R transakcija nema konflikt sa već raspoređenom nultom transakcijom, pa se dodeljuje radniku W_1. Dalje, pošto bi raspoređivanje treće M transakcije na najmanje opterećenog radnika W_2 dovelo do konflikta sa već raspoređenom nultom R transakcijom (na radniku W_0), AC algoritam odmah dodeljuje treću M transakciju najopterećenijem radniku W_0, i time propušta prilku da napravi bolji raspored, jer treća M transakcija nema konflikt sa prvom M transakcijom (na radniku W_0). Ovu priliku će AAC algoritam iskoristiti i napraviti bolji raspored.

Sličan propust AC algoritam pravi i prilikom raspoređivanja pete M transakcije, koju on zbog konflikta na najmanje opterećenom radniku W_3, raspoređuje na najopterećenijeg radnika W_0, a mogla bi biti raspoređena na radnika W_1 ili W_2.

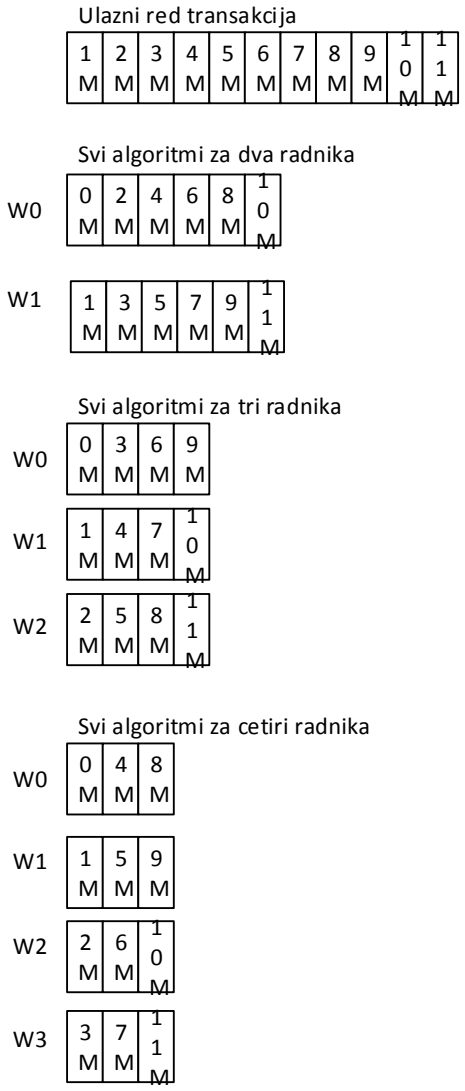


Slika 3.4: Rasporedi transakcija za RDW opterećenje na četiri radnika (AC i AAC).

AAC algoritam funkcioniše po modulu od osam i pravi raspored koji omogućava najbrže izvršnje potpuno bez konflikata. Tako, nulta R i prva M transakcija se dodeljuju radniku W_0, druga R i treća M transakcija se dodeljuju radniku W_1, četvrta R i peta M transakcija se dodeljuju radniku W_2, a šesta R i sedma M transakcija radniku W_3. Zatim, sve funkcioniše na isti način: osma i deveta transakcija se dodeljuju radniku W_0 itd.

3.2.2 Analiza rasporeda za CFW radno opterećenje

Na slici 3.5 je prikazan raspored M transakcija, koje nisu u konfliktu, i to za sve algoritme (RR algoriotam, ETLB algoritam, AC algoritam i AAC algortiam) na dva, tri i četiri radnika. S obzirom da, ako nema konflikata između transakcija, nema ni konkurencije između radnika, odnosno transakcije se mogu izvršavati paralelno – po dve na dva radnika, po tri na tri radnika i po četiri na četiri radnika.



Slika 3.5: Rasporedi transakcija za CFW opterećenje za sva četiri algoritma.

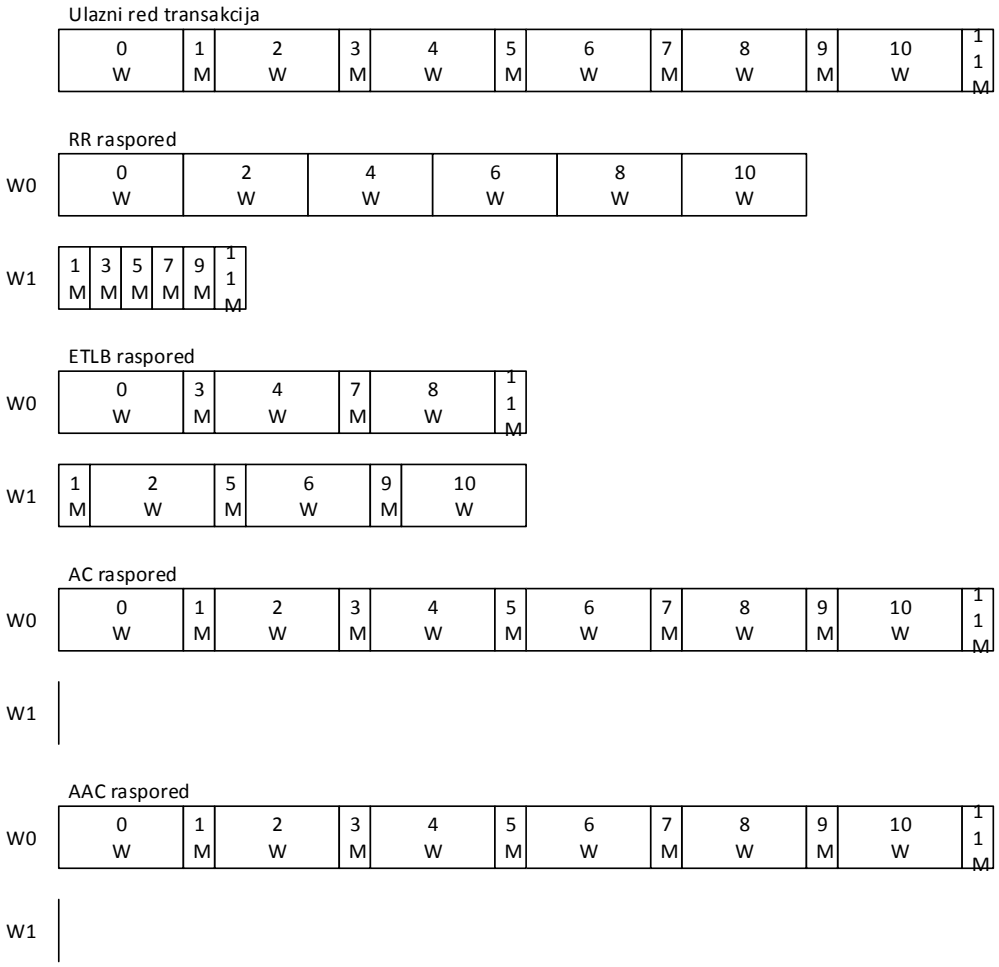
3.2.3 Analiza rasporeda za WDW radno opterećenje

Na slici 3.6 su prikazani rasporedi transakcija za sva četiri algoritma (RR algoritam, ETLB algortam, AC algoritam i AAC algoritam) za ulazno opterećenje WDW, na dva radnika.

Pošto se ulazna opterećenja WDW i RDW razlikuju samo u tome što su kod WDW parne transakcije tipa W a kod RDW parne transakcije su tipa R, a s obzirom da RR algoritam ne

razlikuje tipove transakcija (W i R su za njega ravnopravne), on pravi isti raspored za WDW ulaz kao i za RDW ulaz. Izlazni raspored transakcija je isti kao na slici 3.1, samo su sada na radnika W_0 raspoređene parne W transakcije (dok su na slici 3.1 bile u pitanju R transakcije).

Pošto ni ETLB algoritam ne razlikuje tip transakcija, on takođe pravi isti raspored za WDW ulaz kao i za RDW ulaz. Izlazni raspored transakcija je isti kao na slici 3.1, samo što su sada R transakcije zamenjene W transakcijama.



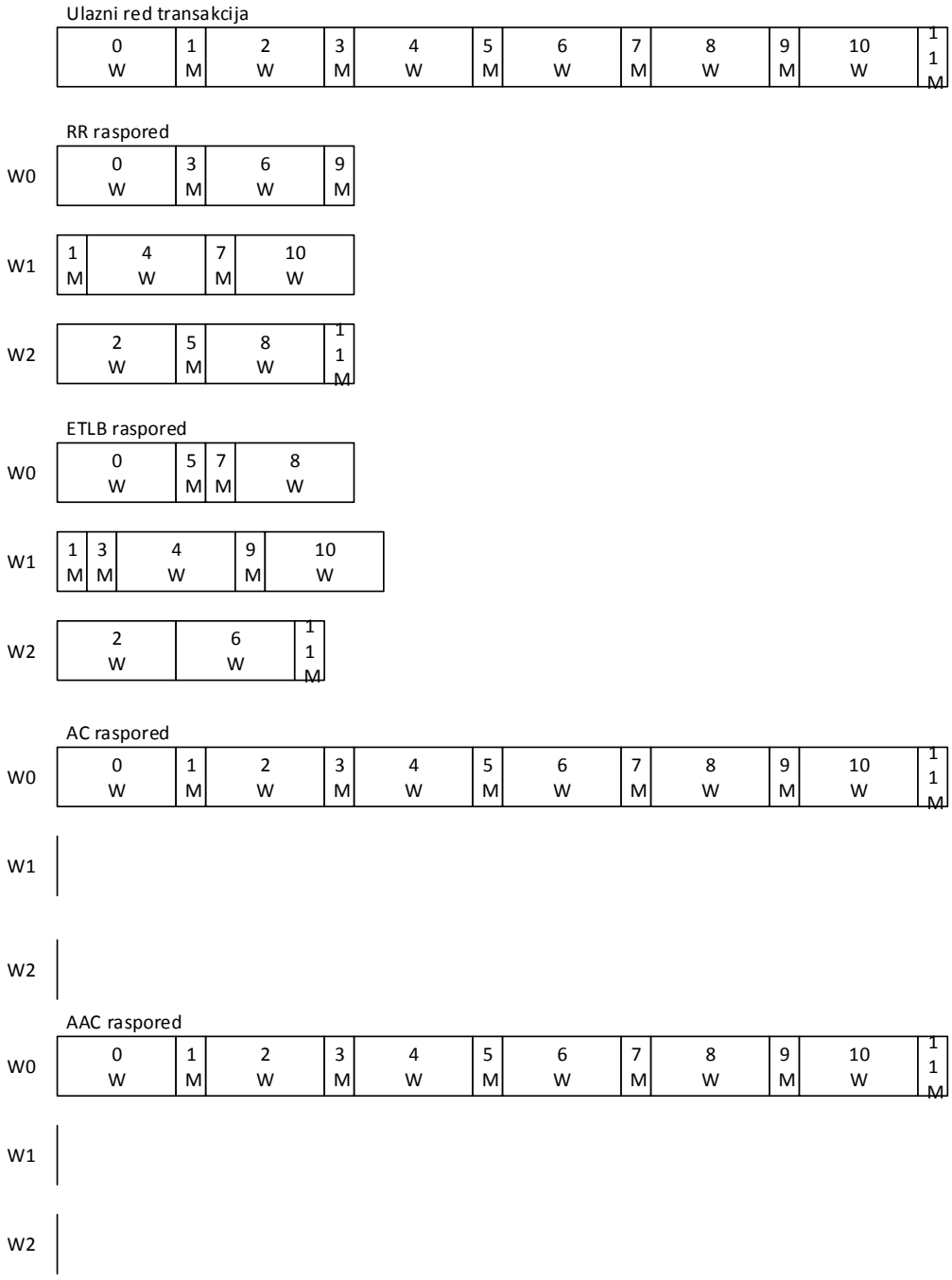
Slika 3.6 Rasporedi transakcija za WDW opterećenje na dva radnika.

AC i AAC algoritmi imaju loše performanse, jer nakon što je nulta W transakcija raspoređena na radnika W_0, svaka sledeća M ili W transakcija ima konflikt sa njom, pa ne može biti raspoređena na slobodnog radnika W_1, već mora biti serijalizovana sa nultom W transakcijom, odnosno raspoređuje se na kraj reda transakcija za radnika W_0.

Na slici 3.7 su prikazani rasporedi transakcija za sva četiri algoritma (RR algoritam, ETLB algortam, AC algoritam i AAC algoritam) za ulazno opterećenje WDW, na tri radnika.

S obzirom da RR i ETLB algoritmi ne razlikuju tipove transakcija (W i R su za njih ravnopravne), on prave isti raspored za WDW ulaz kao i za RDW ulaz. Izlazni rasporedi

transakcija na slici 3.7 su isti kao i izlazni rasporedi za RDW opterećenje na slici 3.2, samo su R transakcije na slici 3.2 zamenjene W transakcijama na slici 3.7.

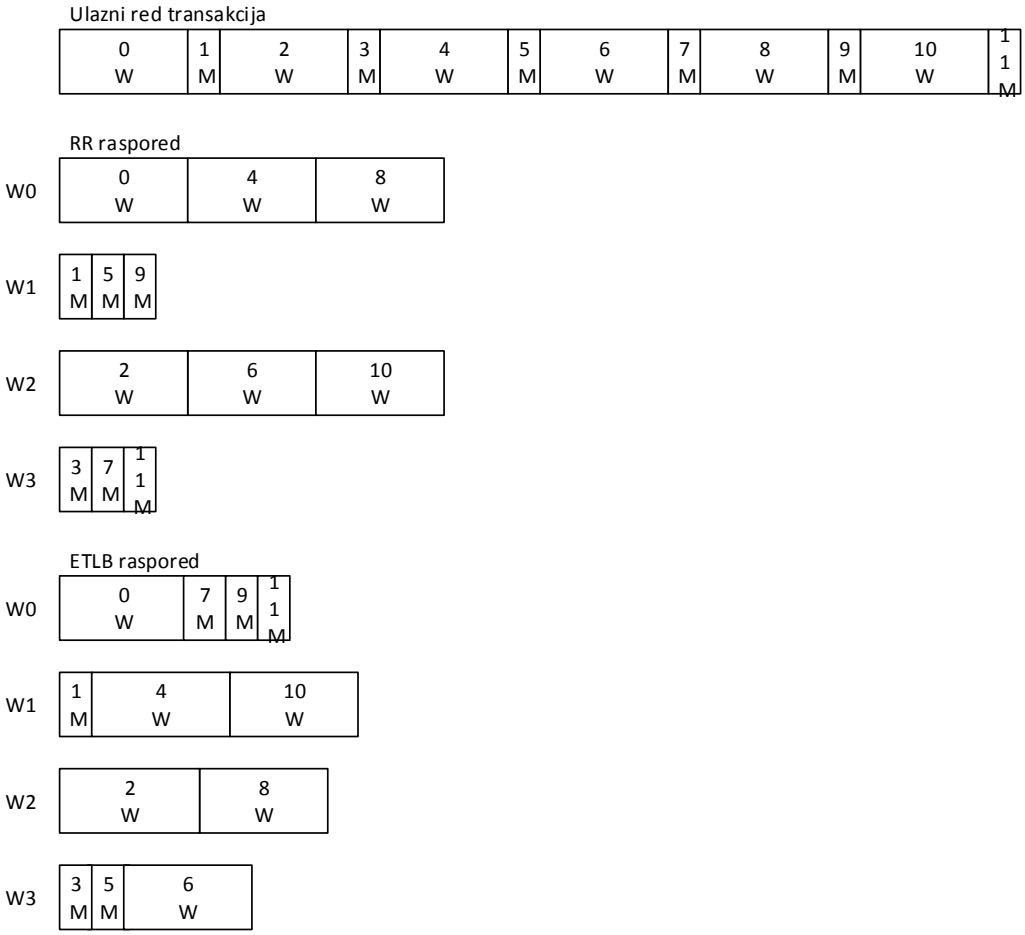


Slika 3.7: Rasporedi transakcija za WDW opterećenje na tri radnika.

S druge strane, AC i AAC algoritmi prave rasporede vrlo slične kao u slučaju dva radnika na predhodnoj slici 3.6, s tim što su na slici 3.7 ponovo sve transakcije serijalizovane na radnika W_0, dok su radnici W_1 i W_2 ostali neupošljeni (na predhodnoj slici postojao je jedan neupošljen radnik W_1). Razlog je isti, nakon što je nulta W transakcija raspoređena na radnika W_0, ni jedna

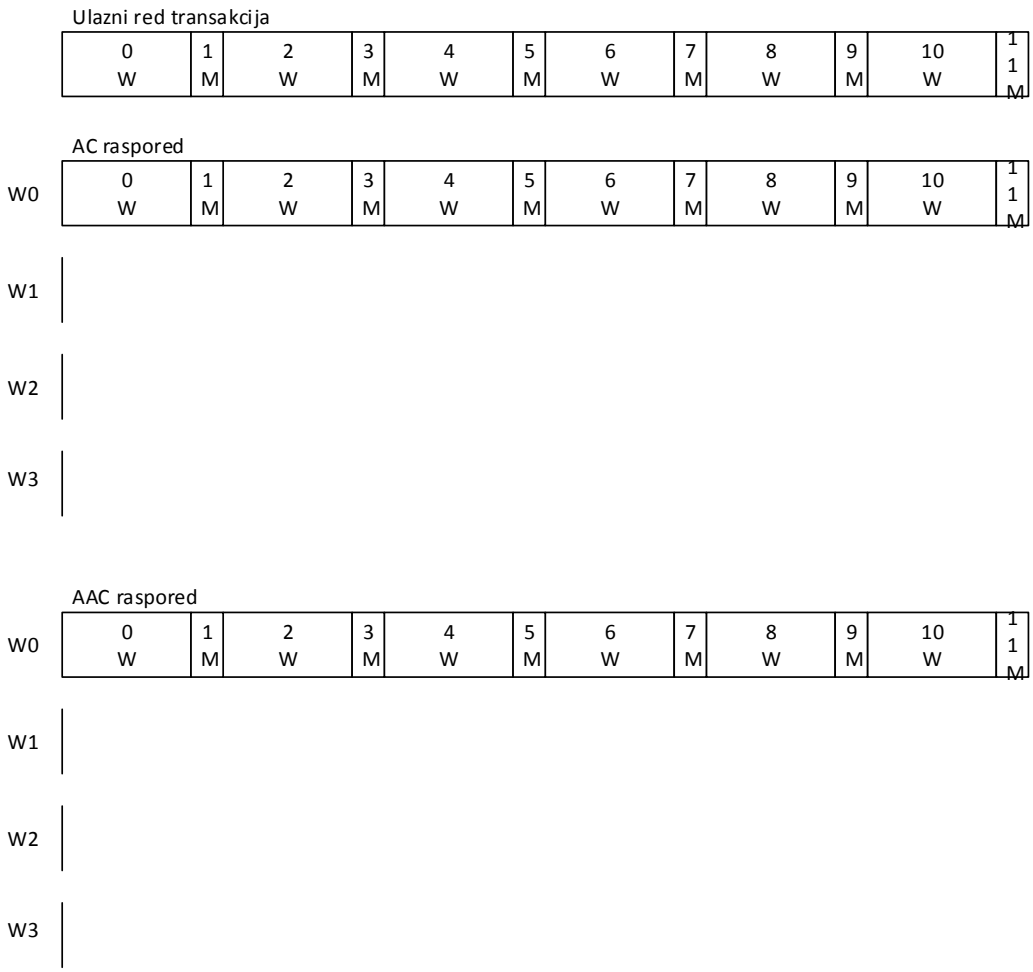
sledeća M ili W transakcija ne može bit raspoređena ni na radnika W_1 niti na radnika W_2, jer bi to dovelo do konflikta sa već raspoređenom nultom W transakcijom.

Na slici 3.8 su prikazani rasporedi transakcija za RR algoritam i ETLB algortam, za ulazno opterećenje WDW, na četiri radnika. S obzirom da RR i ETLB algoritmi ne razlikuju tipove transakcija (W i R su za njih ravnopravne), on prave isti raspored za WDW ulaz kao i za RDW ulaz. Izlazni rasporedi transakcija na slici 3.8 su isti kao i izlazni rasporedi za RDW opterećenje na slici 3.3, samo su R transakcije na slici 3.3 zamenjene W transakcijama na slici 3.8.



Slika 3.8: Rasporedi transakcija za WDW opterećenje na četiri radnika (RR i ETLB).

Na slici 3.9 su prikazani rasporedi transakcija za AC algoritam i AAC algortam, za ulazno opterećenje WDW, na četiri radnika. AC i AAC algoritmi prave rasporede vrlo slične kao u slučaju tri radnika na slici 3.7, s tim što su na slici 3.9 ponovo sve transakcije serijalizovane na radnika W_0, dok su tri radnika (W_1, W_2 i W_3) ostali neupošljeni (dok su na slici 3.7 postojala dva neupošljena radnika – radnici W_1 i W_2). Ponovo je razlog isti, nakon što je nulta W transakcija raspoređena na radnika W_0, ni jedna sledeća M ili W transakcija ne može bit raspoređena ni na radnika W_1, niti na radnika W_2, niti na radnika W_3, jer bi to dovelo do konflikta sa već raspoređenom nultom W transakcijom.



Slika 3.9: Rasporedi transakcija za WDW opterećenje na četiri radnika (AC i AAC).

4. Implementacija AAC algoritma

U ovom poglavlju je opisan ciljni programski paket i implementacija novog AAC algortima.

4.1 Ciljni programski paket

AAC algoritam je implementiran u programskom paketu koji se sastoji od sledećih programskih modula:

- Modul *ts_bank.py* u kome je implementirana PSTM-zasnovana aplikacija Banka.
- Modul *globals.py* u kome su definisane globalne promenljive.
- Modul *stm.py* u kome je realizovana PSTM.
- Modul *el_stat.py* u kome je realizovana metoda za procenu trajanja zadotog tipa transakcije.
- Modul *txn_scheduler.py* u kome su realizovani svi algoritmi za raspoređivanje transakcija (RR algoritam, ETLB algoritam, AC algortiam i novi AAC algoritam); algoritmi su realizovani kao blokovi kodova u funkciji *execute_txns*.

4.2 Python implementacija AAC algoritma

U ovom odeljku je opisana Python implementacija AAC algoritma. U pododeljku 4.2.1 su opisane važnije promenljive koje se koriste u implementaciji.

Sam AAC algoritam je implementiran u funkciji *execute_txns* u modulu *txn_scheduler.py*, kao programski blok od linje 188 do linije 297. Ovaj programski blok se može podeliti po sledećim logičkim celinama:

- Inicijalizacija (linije 188-219), koja odgovara koracima algoritma od 1 do 5.

- Centralni deo algoritma, koji pronalazi radnika sa najmanjim opterećenjem na koga je moguće rasporediti sledeću transakciju bez konflikata (linije 220-277), koji odgovara koraku 6.
- Finalizacija (linije 278-297), koja odgovara koracima algoritma od 7 do 9.

Ove tri logičke celine AAC implementacije su opisane u narednim pododeljcima.

4.2.1 Važnije promenljive

U sledećoj tabeli je za svaku promenljivu iz pseudo koda, kao i važnije promenljive uvedene prilikom opisa algoritama a koje se ne pojavljuju u pseudo kodu, data odgovarajuća Python promenljiva u modulu *txn_scheduler*, funkcija *execute_txns* (linije 188-297), kao i linija u kodu u kojoj se ta Python promenljiva prvi put pojavljuje.

Tabela 4.1: Važnije promenljive korišćene u implementaciji

Promenljiva u pseudo kodu	Python promenljiva	Relevantne linije u kodu
i_{min}	ixminload	207
i_{max}	ixmaxload	208
L (L_i, $i = 1, \ldots, n$)	w_load	190
L_{min}	min(w_load)	207
L_{max}	max(w_load)	208
T_2	txn	204
Q_{in}	q_txns	201
f_2	funTxn	203
t_{d2}	t2len	204
V_2	t2vars	205
M	map_windex_to_wload	216
i	targetWorkerIndex	211
conflictFound	conflictFound	210
F_2	t2	214

F_1	t1	242
Q (Q_i, i = 1, …, n)	w_q_txns	286
QF (QF_i, i = 1, …, n)	w_q_sch_txns	195
QF_i	w_q_sch_txns[targetWorkerIndex]	289

4.2.2 Inicijalizacija

Na slici 4.1 je deo AAC implementacije za inicijalizaciju. U linijama 190-192 se pravi niz *L* (lista *w_load*) i njegovi elementi se postavljaju na 0.0. U linijama 194-197 se pravi niz Q (lista *w_q_sch_txns*) i njegovi elementi se postavljaju na prazne liste. U liniji 201 započinje glavna petlja za raspoređivanje ulaznih transakcija (brojač prolaza kroz tu petlju, *j*, se inicijalizuje u liniji 200).

U liniji 202 se izlančava sledeća transakcija T_2 (promenljiva *txn*) iz ulaznog reda Q_{in} (red *q_txns*). U linijama 203-205, izlančana transakcija se rasčlanjuje i postavljaju se t_{d2} (promenljiva *t2len*) i V_2 (promenljiva *t2vars*). U liniama 207 i 208 se postavljaju i_{min} (promenljiva *ixminload*) i i_{max} (promenljiva *ixmaxload*). U liniji 210 se promenljiva *conflictFound* (*conflictFound*) inicijalizuje na vrednost netačno (False).

U liniji 211 se brojač *i* (*targetWorkerIndex*) postavlja na minimalnu vrednost i_{min} (*ixminload*). Zatim se u liniji 212 iz niza *L* (*w_load*) dobavlja element L_{min} sa indeksom i_{min} *(ixminload*) i upisuje u promenljivu *t2b*. U liniji 213 se sabiraju *t2b* i *t2len* i rezultat se upisuje u promenljivu *t2e*, a u liniji 214 se formira torka *t2*, koja predstavlja raspoređenu transakciju.

U liniji 216 se mapa *M* (*map_windex_to_wload*) postavlja na inicijalnu vrednost, {}. U linijama 217 i 218 se u petlji formiraju stavke mape *M* (*map_windex_to_wload*) za pojedine radnike, koje preslikavaju indeks radnika (*windex*) u vrednost opterećenja tog radnika (*w_load*[*windex*]). Konkretno to se radi tako što se iterira preko promenljive *windex* i odgovarajući po redu element iz liste *L* (*w_load*) se pridružuje novom elementu mape *M* sa ključem *windex*.

```
188   if SCH_ALG == SCH_ALG_AAC:
189     # AC schedule txns to workers
190     w_load = []
191     for i in range(0, NO_SCHEDULER_WORKERS_):
192       w_load.append(0.0)
193
```

```
194    # Create worker-scheduled-txns-queues
195    w_q_sch_txns = []
196    for i in range(0, NO_SCHEDULER_WORKERS_):
197      w_q_sch_txns.append( [] )
198
199    # Schedule txns
200    j = 0
201    while not q_txns.empty():
202      txn = q_txns.get()
203      (funTxn, tvars, (priority, retries)) = txn
204      t2len = globals.execStats[hash(funTxn)].mode()
205      t2vars = tvars
206
207      ixminload = w_load.index(min(w_load))
208      ixmaxload = w_load.index(max(w_load))
209
210      conflictFound = False
211      targetWorkerIndex = ixminload
212      t2b = w_load[ixminload]
213      t2e = t2b + t2len
214      t2 = (t2b, t2e, t2vars)  # t2 is scheduling infor for the next txn to be scheduled
215
216      map_windex_to_wload = {}
217      for windex in range(0, NO_SCHEDULER_WORKERS_):
218        map_windex_to_wload[windex] = w_load[windex]
219
```

Slika 4.1: AAC kod – inicijalizacija.

4.2.3 Pronalaženje optimalne pozicije transakcije u rasporedu

Na slici 4.2 je deo AAC implementacije za pronalaženje optimalne pozicije transakcije u rasporedu transakcija. U liniji 220 se koristi dobro poznat idiom, pomoću koga se pretražuje mapa (tj. iterira kroz mapu) u sortiranom redosledu vrednosti iz stavki mape, a to su u ovom slučaju opterećenja radnika. Tako se pretražuju svi radnici od radnika sa najmanjim opterećenjem ka radniku sa najvećim opterećenjem. U narednoj liniji 221 se proveri da li je ključ radnika *key* dostigao vrednost *ixmaxload*, odnosno da li se u pretrazi stiglo do radnika sa najvećim opterećenjem (L_{max}). Ako jeste, onda se ovaj slučaj svodi na slučaj da je okriven

konflikt, tako što se u liniji 222 promenljiva *conflictFound* postavi na vrednost tačno i u liniji 223 se izlazi iz petlje započete u liniji 220 (break).

U suprotnom slučaju, u liniji 224 promenljiva i_{min} (*ixminload*) postavi na vrednost *key* (indeks tekućeg radnika), a zatim se u liniji 225 promenljiva *i* (*targetWorkerIndex*) postavi na i_{min} (*ixminload*). Dalje sledi provera u dve ugnježdene for petlje da li postoji konflikt između nove transakcije *t2* raspoređene na radnika W_i i već raspoređenih transakcija na drugim radnicima.

Spoljna for petlja, koja počinje u liniji 234, ide po svim radnicima (Wj, j = 1,..., n). Kada se naiđe na ranika čiji indeks je jednak indeksu *i* (*targetWorkerIndex*), vidi liniju 236, on se preskoči u liniji 237 (continue), zato što sigurno nema konlikata između transakcije *t2* i već raspoređenih transakcija na tog radnika.

U unutrašnjoj for petlji, koja počinje u liniji 239, se proverava da li transakcija *t2* ima konflikt sa već raspoređenim transakcijama na tekućeg radnika spoljne for petlje W_j. Ta unutrašnja for petlja je napisana korišćenjem dobro poznatog idioma za pretragu niza unazad, od poslednjeg elementa ka prvom (u nazad u vremenu). U liniji 242 se promenljiva *t1* redom, u svakom prolazu, postavlja na sve te već raspoređene transakcije na radniku W_j. U liniji 244 se promenljiva *t1* rasčlanjuje na njene konstitutivne promenljive *t1b*, *t1e* i *t1vars*.

U liniji 246 se proverava da li je ispunjen uslov (*t2b* >= *t1e*), jer ako jeste, onda niti transakcija *t1* može da napadne transakciju *t2*, niti obrnuto, pa s obzirom da sigurno nema konflikta, unutrašnja for petlja se napušta u linij 248 (break).

Zatim se u liniji 250 pozove funkcija *dataRaceCondition*, koja proveri da li su za transakcije *t1* i *t2* zadovoljeni Bernštajnovi uslovi, odnosno da ne postoji trka do podataka između njih. Ako su Bernštajnovi uslovi zadovoljeni, unutrašnja for petlja se napušta u liniji 252 (break).

Ako se prođu gornje linije, onda postoji trka do podataka između transakcija *t1* i *t2* i mogućnost da *t1* napadne *t2* ili obratno. Dalje se u liniji 255 proverava da li transakcija *t1* napada transakciju *t2*, i ako da, pošto je otkriven konflikt, u liniji 257 se promenljiva *conflictFound* postavlja na tačno i u liniji 258 se napušta unutrašnja for petlja (break).

Slično tome, u liniji 260 se proverava da li transakcija *t2* napada transakciju *t1*, i ako da, pošto je otkriven konflikt, u liniji 262 se promenljiva *conflictFound* postavlja na tačno (True) i u liniji 263 se napušta unutrašnja for petlja (break).

U liniji 265 se proverava da li je unutar unutrašnje for petlje otkriven konflikt, i ako jeste, u liniji 266 se napušta spoljna for petlja (break).

Ako u dve ugnježdene for petlje nije otkriven konflikt (*conflictFound* ima vrednost netačno), ažuriraju se promenljive *targetWorkerIndex*, *t2b*, *t2e*, *t2* (linije 269-272), i napušta se

glavna petlja (break u liniji 273) koja je započela u linij 220. U suprotnom slučaju (linija 274), resetuje se promenljiva *conflictFound* (linija 275) i prelazi na sledeću iteraciju glavne petlje (*continue* u liniji 276), koja je započela u liniji 220.

```
220         for key, value in sorted(map_windex_to_wload.items(), key=lambda item: (item[1], item[0])):
221         if key == ixmaxload:
222           conflictFound = True
223           break
224         ixminload = key
225         targetWorkerIndex = ixminload
226
227-231  Linije od 227 do 231 su komenentarisane, pa su ovde namerno izostavljene.
232
233         # Check for conflicts - the next two nested for loops
234         for i in range(0, len(w_q_sch_txns)):
235
236           if i == targetWorkerIndex:
237             continue
238
239           for k in range(len(w_q_sch_txns[i])-1, -1, -1):  # range(startix, endix+step, step)
240             #print('i=', i, 'k=', k)
241
242             t1 = w_q_sch_txns[i][k]
243             #print('t1=', t1)
244             (t1b, t1e, t1vars) = t1
245
246             if t2b >= t1e:    # neither t1 can attack t2 nor t2 can attack t1
247               #print('The End of conflict checking: t1e=', t1e, 't2b=', t2b)
248               break
249
250             if not dataRaceCondition(t1vars, t2vars):
251               #print('data race condition is False')
252               break
253
254             #print('data race condition is True')
```

```
255           if (t2b < t1e <= t2e) :
256             #print('conflict 1 found, t2b=', t2b, 't1e=', t1e, 't2e=', t2e)
257             conflictFound = True
258             break
259
260           if (t1b < t2e <= t1e):
261             #print('conflict 2 found, t1b=', t1b, 't2e=', t2e, 't1e=', t1e)
262             conflictFound = True
263             break
264
265         if conflictFound:
266           break
267
268       if not conflictFound:
269         targetWorkerIndex = ixminload
270         t2b = w_load[ixminload]
271         t2e = t2b + t2len
272         t2 = (t2b, t2e, t2vars)  # t2 is scheduling infor for the next txn to be scheduled
273         break
274       else:
275         conflictFound = False
276         continue
277
```

Slika 4.2: AAC kod – pronalaženje optimalne pozicije transakcije u rasporedu.

4.2.4 Završetak algoritma

Na slici 4.3 je deo AAC implementacije za finalizaciju algoritma. Na početku ovog dela, u linij 279, proverava se da li je promenljiva *conflictFound* postavljena na vrednost tačno. Ako nije, onda je transakcija *t2* uspešno raspoređena u predhodnom delu (opisanom u predhodnom odeljku), i prelazi se na liniju 286. U suprotnom slučaju, u predhodnom delu se došlo do kraja pretrage radnika, odnosno do radnika sa najvećim opterećenjem (tj. nije bilo moguće rasporediti transakciju ni na jednog radnika sa manjim opterećenjem). Zbog toga se u liniji 281 promenljiva *i* (*targetWorkerIndex*) postavlja na indeks radnika sa najvećim opterećenjem i_{max} (ixmaxload), a u liniji 283 transakcija *t2* podešava za izvršenje na kraju reda pridruženog tom radniku: vreme početka *t2* se postavlja na L_{max} (*w_load*[*ixmaxload*]), vreme završetka *t2* na L_{max} + *t2len*, a transakcione promenljive *t2var* ostaju nepromenjene.

Zatim se u liniji 286 neraspoređena transakcija T_2 (*txn*) ulanča u red Q_i (*w_q_txns* [*targetWorkerIndex*], u liniji 289 raspoređena transakcija F_2 (*t*2) ulanča u red QF_i (*w_q_sch_txns* [*targetWorkerIndex*]), i u liniji 292 se ažurira opterećenje L_i (*w_load*[*targetWorkerIndex*]) radnika W_i na koga je raspoređena transakcije F_2 (*t*2).

```
278     # Check for conflict and change the traget worker's queue if conflict was found
279     if conflictFound:
280       #print('Conflict found')
281       targetWorkerIndex = ixmaxload
282       # Update t2
283       t2 = (w_load[ixmaxload], w_load[ixmaxload] + t2len, t2vars)
284
285     # Put txn at the tail of the target worker's queue
286     w_q_txns[targetWorkerIndex].put(txn)
287
288     # Update w_q_sch_txns[targetWorkerIndex]
289     w_q_sch_txns[targetWorkerIndex].append(t2)
290
291     # Update worker's load
292     w_load[targetWorkerIndex] = w_load[targetWorkerIndex] + t2len
293
294     # Print schedule
295     #print('j, targetWorkerIndex, tvars =', j, targetWorkerIndex, tvars)
296     j = j + 1
297
```

Slika 4.3: AAC kod – završetak algoritma.

.

5. Eksperimentalna evaluacija

U ovom odeljku predstavljeni su eksperimentalni rezultati evaluacije novog AAC algoritma. Takođe, analizana je performansa AAC algoritma u odnosu na tri već postojeća algoritma – RR, ETLB i AC algoritam.

5.1 Test program za evaluaciju performanse Banka

Evaluacija i merenje efikasnosti transakcionih sistema nije jednostavan zadatak. U cilju dobijanja što relevantnijih rezultata evaluacije potrebno je odabrati adekvatan testni program. Za merenje performanse algoritama za raspoređivanje TM transakcija koristi se program Banka.

Banka predstavlja jednostavnu transakcionu aplikaciju koju čine transakcije i softverska transakciona memorija (STM). Aplikacija definiše skup transakcija, koje mogu biti sa i bez sinhronizacije na barijeri (u ovom radu su korišćene transakcije bez sinhronizacije na barijeri). Na početku izvršavanja svaka transakcija dobavlja skup t-promenljivih iz STM i obrađuje ih. Kada transakcija završi sa obradom ona onda pokušava da ažurira vrednosti t-promenljivih. Ako se ažuriranje uspešno izvrši onda se smatra da je transakcija uspešno završena, inače transakcija počinje ponovno izvršavanje. Svaka transakcija mora se izvršiti uspešno, što znači da transakcije mogu više puta da ponove svoje izvršavanje. Transakcija može izvršiti sledeće operacije: (i) čitanje, (ii) pisanje i (iii) par operacija čitaj-piši. Transakcije čitanja i pisanja čitaju i pišu na određeni bankovni račun, respektivno. Transakcije čitaj-piši obavljaju obe operacije, na primer: čita stanje sa jednog i prebacuje pare na drugi bankovni račun. Odabir bankovnih računa može da se definiše na slučajan ili unapred definisan način.

Transakcija koja izvršava operaciju čitanja, odnosno pisanja naziva se transakcija čitanja (read), odnosno transakcija pisanja (write) respektivno. Transakcija koja čita stanje jednog bankovnog računa i menja stanje drugog, odnosno transakcija koja prebacuje pare sa jednog na drugi bankovni račun naziva se transakcija za prenos novca (money transfer).

5.2 Opis merenja

U radu [8] su validirana tri prethodno razvijana algoritma raspoređivanja, RR, ETLB i AC, i upoređene su njihove performanse pomoću eksperimenata korišćenjem TM aplikacije zasnovane na PSTM pod nazivom Banka, koja se izvršavala na procesoru sa dva jezgra. U ovom radu je validiran novi AAC algoritam i upoređena je performansa sva četiri algoritma ponavljanjem eksperimenata na isitim radnim opterećenjima, kao u radu [8], ali ovaj put na procesoru sa četiri jezgra. Preciznije, korištena je Intel Core i7-3770@3.40GHz mašina sa četiri jezgra i 16GB radne memorije, Windows 64 operativnim sistemom i Python interpreterom verzije 3.4.

Kao u radu [1], i ovde koristimo relativno ubrzanje kao meru poređenja performanse dva algoritma raspoređivanja, *A1* i *A2*. Neka dva algoritma, *A1* i *A2* obrađuju isto opterećenje *L*, i neka t_{e1} i t_{e2} budu odgovarajuća srednja vremena izvršenja *A1* i *A2*, respektivno. Onda je relativno ubrzanje *S* od *A2* u odnosu na *A1* dato kao:

$$S = t_{e1} / t_{e2}. \tag{11}$$

Pošto je RR algoritam najjednostavniji od sva četiri algoritma raspoređivanja posmatrana u ovom radu, koristili smo ga kao osnovu za dalje analiziranje performansi. Relativna ubrzanja ETLB, AC i AAC algoritma su računata u odnosu na RR algoritam, tj. u jednačini 11 t_{e1} predstavlja vreme izvršenja RR algoritma.

Da bi smanjili smetnje sa lokalnim OS, eksperimente smo sproveli koristeći paketsku obradu transakcija, umesto online obrade transakcija. Jedina razlika između paketske i online obrade, koja je opisana u odeljku 2.1.1 je da celokupni paket transakcija (tj. radno opterećenje) je ulančan u red Q_{in} pre početka izvršavanja, i tokom obrade ne dodaju se nove transakcije.

U sprovedenim eksperimentima, kao i u radu [8], korišćena su tri različita paketa transakcija (radna opterećenja): (i) paket transakcija sa dominantnim čitanjem (eng. *Read-Dominated Workload - RDW*), (ii) paket tansakcija bez konflikata (eng. *Conflict-Free Workload – CFW*) i paket transakcija sa dominatnim pisanjem (eng. *Write-Dominated Workload - WDW*). *RDW* je paket transakcija napravljen od 100 transakcija koje očitavaju sve račune (*RAA – Read All Accounts*) i 100 transakcija za prenos novca sa računa na račun (*MT – Money Transfer*) koje su bez konflikta tako da su unutar *RDW* paketa *RAA* transakcije uvek neparne a MT transakcije uvek parne. *CFW* je samo skup od 100 MT transakcija za prenos novca, bez međusobnog konflikta. *WDW* je paket transakcija napravljen od 100 transakcija koje upisuju zadatu vrednost u sve račune (*WAA – Write All Accounts*) i 100 transakcija za prenos novca sa računa na račun (*MT – Money Transfer*) koje su bez konflikta tako da su unutar *WDW* paketa *WAA* transakcije uvek neparne a MT transakcije uvek parne.

Za razliku od rada [8] gde je prag L_{ut} bio $L_{ut} = 0.3$ u svim eksperimentima, u ovom radu prag L_{ut} je postavljen na beskonačno, tj. nije ograničavana neravnoteža opterećenja po procesnim elemetima.

Eksperimenti su organizovani na sledeći način. Formirane su tri grupe eksperimenata sa po tri gorepomenuta paketa transakcija. Dalje, u svakoj grupi eksperimenata napravljene su po četiri pod-grupe eksperimenata, jedna podgrupa za svaki od četiri algoritma raspoređivanja. Na kraju, u svakoj podgrupi eksperimenata izvršeno je po 10 merenja, kako bi se dobila objektivna srednja vrednost vremena izvršavanja.

Zbog nesavršenosti sistema na kome su obavljenja merenja i nemogućnosti apsolutnog izolovanja ispitivanog sistema od spoljnih uticaja (u našem slučaju operativnog sistema) granične vrednosti merenja nisu razmatrane, odnosno krajnje niska i visoka vremena izvršenja su zanemarena. Takođe, treba napomenti da se PSTM izvršava kao poseban Python proces koji zajedno sa radnicima konkuriše za izvršavanje na fizičkim jezgrima. Predstavljeni rezultati su preliminarni i daljem radu biće evaluirani na mnogojezgarnoj arhitekturi (eng. manycore).

5.3 Analiza rezultata

Eksperimentalni rezultati su dati u Tabeli 5.1. Kolone tabele sadrže rezultate za sva četiri posmatrana algoritma, RR, ETLB, AC i AAC, respektivno. Svi rezultati u kolonama su dati za dva i tri radnika. Vrste tabele predstavljaju rezultate ostvarene za različite pakete opterećenja. U preseku vrste i kolone dati su sledeći rezultati: vreme izvršenja (eng. *Time – T*) konkretnog paketa transakcija, broj neuspešnih, odnosno ponovljenih transakcija (eng. *Abort – A*) i ostvareno relativno ubrzanje (eng. *Speedup – S*) u odnosu na RR algoritam. U nastavku sledi analiza rezultata za svaki tip radnog radnog operećenja u zavisnosti od tipa algoritma i broja radnika.

5.3.1 Analiza rezultata za RDW radno opterećenje

U slučaju paketa transakcija sa dominantnim čitanjem i dva radnika za obradu transakcija, (vidi Tabelu 5.1, red *RDW*, kolona za dva radnika) performansa RR algoritma je najlošija i njegovo prosečno vreme izvršenja je 4.43s, što je najveće od svih vrednosti. Performansa ETLB algoritma je bolja, prosečno vreme izvršenja je 3.81s. Performansa AC i AAC algoritma je najbolja, njihovo prosečno vreme izvršenja je 3.11s i 3.09s, respektivno. Relativno ubrzanje od ETLB algoritma u odnosu na RR algoritam je 1.16, u proseku, dok su ubrzanja za AC i AAC algoritam u proseku 1.42 i 1.43, respektivno. AAC algoritam ispoljava isto ponašanje kao AC algoritam, što se može proveriti kroz ostvareno ubrzanje – ovo je tako jer AAC algoritam nema prostora da razvije raspored koji bi dominirao u odnosu na AC zbog nedovoljnog broja radnika (samo dva).

Veoma je važno primetiti da AC i AAC algoritmi obično proizvode osetno bolje rasporede od RR algoritma – u najboljem slučaju, vreme izvršenja AC i AAC algoritma je bilo samo 2.2s, koje kad bi se uporedilo sa srednjom vrednošću vremena izvršenja RR algoritma, daje savršeno ubrzanje od 2.0x (na dva radnika i mašini sa četiri jezgra). Inače, visoko kvalitetan raspored napravljen od strane AC i AAC algoritma, može biti narušen od strane drugih procesa na operativnom sistemu, koji se možda ponovno pokrenu sporadično i unesu neželjeno odstupanje u vremenu početka transakcije i drastično degradirati performanse. Na primer, u najgorem slučaju, vreme izvršenja AC i AAC algoritma raste do 3.6s, što odgovara relativnom ubrzanju AC i AAC algoritma u odnosu na RR algoritam na 1.22x.

	Dva Radnika								***Tri Radnika***							
	RR		***ETLB***		***AC***		***AAC***		***RR***		***ETLB***		***AC***		***AAC***	
	T S	A	T S	A	T S	A	T S	A	T S	A	T S	A	T S	A	T S	A
RDW	4.43 -	1	3.81 1.16	35.67	3.11 1.42	9.33	3.09 1.43	5.67	2.49 -	1.33	2.96 0.84	70	2.23 1.11	9.33	1.91 1.30	9.33
CFW	0.135 -	0	0.135 0.99	0	0.135 0.99	0	0.136 0.99	0	0.126 -	0	0.127 0.99	0	0.127 0.99	0	0.127 0.99	0
WDW	4.40 -	1	5.27 0.83	97.67	4.59 0.95	0	4.59 0.95	0	6.67 -	88,67	6.44 1.03	97.33	4.52 1.47	0	4.51 1.47	0

Tabela 5.1: Rezultati merenja izvršenja transakcija za dva i tri radnika.

Gledajući Tabelu 5.1, kolona „A“, primećujemo da je broj neuspešnih transakcija (abort) za RR algoritam izuzetno nizak – u proseku jedna transakcija se neuspešno izvrši. Na prvi pogled, ovo može da izgleda neintuitivno. S jedne strane srednje vreme izvršenja RR algoritma je najveće, a sa druge strane, broj neuspešnih transakcija je najmanji. Normalno bismo očekivali proporciju između vremena izvršenja i broja neuspešnih transakcija. Zašto onda to ovde nije slučaj?

Odgovor na dato pitanje je relativno jednostavan. RR algoritam raspoređuje sve RAA transakcije na process radnika W_1 i sve MT transakcije na proces radnika W_2. RAA transakcije su dugačke (reda 45ms), dok su MT transakcije relativno kratke (reda veličine 0.65ms). Tako da, iako je rezultujuća raspodela skoro bez konflikata, ona je relativno duga, zato što W_1 izvršava sve RAA sekvencijalno.

Sa druge strane, ETLB algoritam uzima u obzir procenjeno vreme izvršenja i tako pravi kraće rasporede, koje su u isto vreme konkurentnije, što za uzvrat daje povećan broj neuspešnih transakcija – tačnije za ovaj algoritam, srednja vrednost neuspešnih transakcija je 35.67, što je najveća vrednost za sve algoritme. AC algoritam, a pogotovo AAC algoritam, dodatno izbegava konflikte, tako da može da proizvede najbolje rasporede, što se i pored ostvarenog ubrzanja može proveriti i kroz neuspešnih transakcija. U proseku, unutar rasporeda transakcija dobijenih

AC i AAC algoritmom ima 9,33 i 5,67 neuspešnih transakcija, respektivno. Bez obzira na povećan broj neuspešnih transakcija u odnosu na RR algoritam, AC i AAC algoritmi naprave bolji raspored jer poseduju informaciju o konfliktima između transakcija što rezultuje povoljnijim vremenom izvršenja. Kad su smetanje od strane OS-a manje, ovi veoma kvalitetni rasporedi vode ka izvršenjima radnih opterećenja sa malim brojem neuspešnih transakicja, npr. 1 ili 2 (u nekim eksperimentima). Najinteresantniji su eksperimenati u kojima su svih 200 transakcija bile uspešne, i nije bilo neuspešnih uopšte. U ovim eksperimentima, veoma mala smetnja od strane operativnog sistema je zapravo uzrokovala savršeno izvršenje bez konflikata, na radnom opterećenju.

Uporedna analiza rezultata algoritama za tri radnika je donekle slična analizi za dva radnika ali ipak nije u potpunosti ista.

Za tri radnika performansa ETLB algoritma je najlošija i njegovo prosečno vreme izvršenja je 2,96s, što je najveće od svih vrednosti, odnosno relativno ubrzanje za ETLB algoritam u odnosu na RR algoritam je 0.84x, što je suštinski usporenje od 0.16x. Degradacija performanse ETLB algoritma je uzrokovana prevelikim broj neuspešnih transakcija koja u proseku iznosi 70. Kako ETLB algoritam poseduje samo informaciju o opterećenosti radnika onda nije u stanju da izbegne konflikte. Štaviše, povećanje broja radnika ne pogoduje ETLB algoritmu jer se povećava konkurentnost transakcija, odnosno verovatnoću da se neka od MT transakcija preklopi sa nekom od RAA transakcija što neposredno dovodi do ponovljenog izvršavanja transakcije koja duže traje, a to je u ovom slučaju RAA transakcija. RR algoritam ima drugo vreme izvršavanja, 2.49s, što je manje od vremena potrebno ETLB algoritmu a veće od AC i AAC algoritma. Uprkos većem broju ponovljenih transakcija od RR algoritma, AC i AAC algoritam očekivano ostvaruju najbolja vremena izvršenja, 2.23s i 1.91s, odnosno ubrzanja, 1.11x i 1.30x, respektivno. Za razliku od prethodnog slučaja sa dva radnika gde su AC i AAC algoritmi ispoljavali isto ponašanje (vreme izvršavanja i ubrzanje) ovde se jasno vidi dominacija AAC algoritma u odnosu AC algoritam što je posledica povećanog broja radnika. Naime, AAC algoritam je u stanju da optimalnije iskoristi nove radnike i ostvari bolji raspored transakcija što se odražava na ostvareno vreme izvršenja, odnosno ubrzanje. Za razliku od ETLB algoritma, AAC algoritmu pogoduje povećanje broja radnika.

Interesantno je uporedno analizirati vrednost ostvarenih ubrzanja i vremena izvršenja za dva i tri radnika. Iako su apsolutne vrednosti ubrzanja veće za dva radnika to nije slučaj i sa vremenima izvršenja. U svim slučajevima vremena izvršenja za tri radnika su manja u odnosu na vremena izvršenja za dva radnika. Ovo je direktan uticaj povećanog broja radnika na propusnost sistema.

Takođe, interesantno je analizirati zašto se broj neuspešnih transakcija povećao za AAC algoritam? Razlog treba tražiti u tome kakav raspored napravi AAC algoritam za tri radnika. Naime, raspored dobijen AAC algoritmom je veoma osetljiv na smetnje, tj. pomeranja uzrokovana operativnim sistemom što dovodi do povećanog broja neuspešnih transakcija te nešto manje ostvarenog ubrzanja. Za AAC algoritam zanimljivi su eksperimenti za tokom kojih se dogodi i preko 40 neuspešnih transakcija. I pored toga, dominanti su rezultati tokom kojih se desi jedan ili ni jedan neuspešan pokušaj ažuriranja tj. ponavljanja transakcije.

5.3.2 Analiza rezultata za CFW radno opterećenje

Eksperimentalni rezultati, za izvršenje paketa transakcija bez konflikata, su prezentovani u Tabeli 5.1, red *CFW*. Iz ovog reda vidimo da je performansa sva četiri algoritma približno ista kako za dva tako i za tri radnika. Relativno ubrzanje ETLB, AC i AAC algoritma u odnosu na RR algoritam je neznatno manje, tačnije ono umesto 1x iznosi 0.99x. Ovo odstupanje se može pripisati sitnim smetnjama zbog nesavršenosti sistema za merenje i mogu se zanemariti, tj. može se smatrati da su sva četiri algoritma, sa stanovišta vremena izvršenja, ostvarila ekvivalentan raspored transakcija.

Kolona "A" potvrđuje da je radno opterećenje uvek bez konflikata. Sa druge strane, ovi rezultati isto potvrđuju da se celokupna arhiktetura raspoređivača ponašala kao što se očekuje i to u oba slučaja (za dva i tri radnika) – radno opterećenje bez konflikata se izvršilo bez konflikata.

5.3.3 Analiza rezultata za WDW radno opterećenje

Eksperimentalni rezultati za paket transakcija sa dominatnim pisanjem i dva radnika su prikazani levo u Tabeli 5.1, red *WDW*. Iz ovog reda vidimo da je RR algoritam najbolji (prosečno vreme izvršenja je 4.40s). Kao što je očekivano, ETLB, AC i AAC algoritmi se ponašaju lošije u odnosu na RR algoritam – prosečna vremena izvršenja su 5.27s, 4.59s i 4.59, respektivno. Ovo je očekivani rezultat, zato što RR algoritam u svom jednostavnom pristupu rasporedi sve WAA transakcije na proces radnika W_1 i sve MT transakcije na proces radnika W_2. WAA transakcije su duge (reda veličine 45ms), dok su MT transakcije kratke (reda 0.65ms). Dakle, sve duge WAA transakcije, su već serijalizovane na procesu W_1. Sve kratke MT transakcije su takođe serijalizovane na W_2 – one će sve biti uspešne u svom prvom izvršenju i izazvaće da 1 ili 2 duge WAA transakcije na W_1 budu neuspešne i ponovo izvršene.

Složeniji ETLB algoritam proizvodi konkurentnije rasporede, što za uzvrat može da proizvede više neuspešnih ažuriranja i, shodno tome, više ponovnih izvršenja. Efekat toga se upravo oslikava u broju neuspešnih transakcija koje je kod ETLB algoritma izrazito veliko (97.68) i koje se direktno odražava na performansu (praktično se dobije usporenje). AC i AAC

algoritmi daju veoma slične rezultate što je posledica toga da se AAC algoritam za dva radnika praktično ponaša kao AC algoritam. AC i AAC algoritam ostvaruju ubrzanje od 0.95x što predstavlja gubitak peformanse u odnosu na RR algoritam koji se pokazao veoma efikasan za raporeĎivanje WDW paketa transakcija. Razlog usporenja AC i AAC algoritma u odnosu na RR algoritam je u tome što su praktično svi zadaci serijalizovani na jednom radniku W_1 te RR algoritam koji raspoređuje na dva radnika dominira. U ovoj analizi za dva radnika bitno je uočiti da je broj ponovnih izvršenja za RR algoritam veoma mali.

Rezultati algoritama ostvareni za dva radnika ipak nisu pravi pokazatelj efikasnosti jer RR algoritmu pogoduje raspoređivanje na dva radnika i tu je u stanju da uspešno učešlja WAA i MT transakcije tako da se dobiju veoma dobri rezultati, odnosno da broj neuspešnih transakcija bude minimalan.

Za tri radnika RR algoritam je ostvario najlošije rezultate – njegovo vreme izvršavanja je 6.67s. Nakon toga sledi ETLB algoritam koji je uspeo da ostvari neznatno ubrzanje od koje iznosi 1.03x, ali uz veliki broj neuspešnih transakcija, 97.33 prosečno. Slično kao u analizi algoritma za RDW paket transakcija, za ETLB algoritam se očekuje da bi se ovaj broj smanjivao sa povećanjem broja radnika jer ETLB kao ni RR algoritam nije u stanju da izbegne koflikte. Ovde se treba osvrnuti na broj ponovljenih izvršenja RR algoritma koji je veoma visok, čak 88,67, što potvrđuje da je raspored koji pravi na dva radnika samo izuzetak. Zbog mogućnosti analize konflikata između nove, neraspoređene, i već raspoređenih transakcija AC i AAC algoritmi obezbeđuju najbolje performase sa ostvarenim ubrzanjima od 1.47x, njh oboje. U ovom slučaju sa povećanim brojem radnika broj ponovnih izvršenja za RR i ETLB algoritme znatno utiče na njihova izvršenja i u prvi plan stavlja AC i AAC algoritme.

.

6. Zaključak

U radu je razvijen AAC algoritam za raspoređivanje TM transakcija sa izbegavanjem konflikata, koji uvek pronalazi raspored bez konflikata i sa minimalnim rasponom. Dodatno, u ovom radu je validiran novi AAC algoritam, i njegova performansa je upoređena sa performansom RR, ETLB i AC algoritama na procesoru sa četiri jezgara, i na bazi test opterećenja raznih nivoa konkurencije i različitog broja radnika.

Početni eksperimentalni rezultati prikazani u ovom radu ukazuju da je, u slučaju male konkurencije i tri procesa radnika, relativno ubrzanje pojedinih algoritama u odnosu na RR algoritam sledeće: za ETLB algoritma je u proseku 0.84x, za AC algoritam je u proseku 1.11x, a za AAC algoritam je 1.30x. U slučaju veoma kratkih transakcija bez konkurencije svi algoritmi imaju slične performanse. U slučaju visoke konkurencije i tri procesa radnika, relativno ubrzanje pojedinih algoritama u odnosu na RR algoritam sledeće: za ETLB algoritma je u proseku 1.03x, za AC i AAC algoritam je u proseku 1.47x puta.

Pravci budućeg rad su istraživanja na nadolazećim mnogojezgarnim sistemima i u oblasti distribuiranih sistema.

.

Literatura

[1] M. Herlihy and J.E.B. Moss, “Transactional memory: architectural support for lockfree data structures,” In *Proc. of the 20th Annual International Symposium on Computer Architecture, ISCA ’93, ACM,* New York, NY, pp. 289–300, 1993.

[2] T. Harris, J.R. Larus, and R. Rajwar, Transactional Memory, 2nd edition, Morgan and Claypool, 2010.

[3] R. Guerraoui, M. Herlihy, and B. Pochon, “Toward a theory of transactional contention managers”, In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pp. 258-264, 2005.

[4] H. Attiya, L. Epstein, H. Shachnai, and Tami Tamir, “Transactional contention management as a non-clairvoyant scheduling problem”, *Algorithmica*, vol. 57, no. 1, pp. 44-61, 2010.

[5] H. Attiya and A. Milani, “Transactional scheduling for read-dominated workloads”, *Journal of Parallel and Distributed Computing*, vol. 72, no. 10, pp. 1386-1396, 2012.

[6] W.N. Scherer and M.L. Scott, “Advanced contention management for dynamic software transactional memory”, In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pp. 240-248, 2005.

[7] R.M. Yoo and H-H.S. Lee, “Adaptive transaction scheduling for transactional memory systems”, In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 169-178, 2008.

[8] M. Popovic, B. Kordic, and I. Basicevic, “Transaction Scheduling for Software Transactional Memory,” In *Proceedings of the 2nd IEEE International Conference on Cloud Computing and Big Data Analysis*, pp. 191-195, 2017.

[9] M. Popovic, and B. Kordic, “PSTM: Python Software Transactional Memory,” In *Proceedings of the 22nd IEEE Telecommunications Forum*, pp. 1106-1109, 2014.

[10] M. Popovic, B. Kordic, and I. Basicevic, “Estimating Transaction Execution Times for a Software Transactional Memory,” In *Proceedings of the 6th IEEE International Conference on Information Science and Technology*, pp. 137-141, 2016.

[11] A. J. Bernstein, “Analysis of Programs for Parallel Processing”, *IEEE Transactions on Electronic Computers*, vol. EC-15, no. 5, pp. 757–763, 1966.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, "Software transactional memory for dynamic-sized data structures," *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC '03)*, pp. 92-101, 2003.

[13] P. Felber, T. Riegel, C. Fetzer, "Dynamic performance tuning of word-based software transactional memory," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pp. 237-246, 2008.

[14] M. Popovic, B. Kordic, and I. Basicevic, “Work, Span, and Parallelism of Transactional Memory Programs,” In *Proceedings of the 4th IEEE Eastern European Regional Conference on the Engineering of Computer Based Systems*, pp. 59-66, 2015.

.

Dodatak A: Primeri PSTM programa

U ovom dodatku su data dva primera PSTM-baziranih programa. Privi primer je klijent-server arhitekturu sa dva servera (radnika) i jednim klijentom, i u njemu se transakcije konkurentno izvršavaju na dva servera (radnika). Dugi primer je prototip raspoređivača transakcija redom u krug. Ova dva primera nemaju neposrednu vezu sa ovim diplomskim master radom, autor ih je napravio u toku upoznavanja sa PSTM i pripreme za razvoj novog AAC algoritma, a u ovaj rad su uključeni jer mogu biti korisni sledićim kolegama koji će možda raditi na PSTM.

A.1 Primer klijent-radnik arhitekture

U glavnom programu se inicijalizuju redovi za slanje zahteva (*qreq1* i *qreq2*) kao i redovi za prijem odgovora (*qrsp1* i *qrsp2*), kao i lista readova za slanje zahteva *lqreq* (čiji elementi su *qreq1*i *qreq2*) i lista redova za prijem odgovora *lqrsp* (čiji elementi su *qrsp1* i *qrsp2*). Potom se prave dve instance klase *Process*, *worker1* i *worker2*, koje izvršavaju funkciju *WorkerServer*. Objekat *worker1*/*worker2* se pravi tako što se funkciji *WorkerServer* prosleđuju kao argumenti red *qreq1*/*qreq2*, red *qrsp1*/*qrsp2*, identifikacija 0/1, i objekat reda za komunikaciju sa PSTM serverom *stm_queue*. Zatim se pravi instanca klase *Process*, *client1*, koja izvršava funkciju *ClientReq*. Prilikom pravljenja objekta *client1*, funkciji *ClientReq* se kao argumenti prosleđuju liste *lqreq* i *lqrsp*.

Server funkcioniše na taj način što se vrti u beskonačnoj petlji i proverava da li mu je poslata poruka *m* = 'terminate' (*Msg*) i ako jeste, upisuje krajnju naznaku u otvorenoj datoteci i izađe iz petlje. U suprotnom, iz primljenog zahteva, server (*worker1*/*worker2*) preuzima funkciju koju transakcija treba da izvrši i t-promeljivu nad kojom ta transakcija treba da radi. Postoje dve funkcije: *tx_inc* koja inkrementira zadatu t-promenljivu i *tx_dec* koja dekrementira zadatu t-

promenljivu. Povratnu vrednost iz pozvane funkcije, server (*worker1*/*worker2*) upisuje u poruku *m*, koju kao odgovor vraća klijentu (*client1*) preko odgovarajućeg reda (*qrsp1*/*qrsp2*).

Klijent u spoljnoj for petlji pravi *num_messages* (10) poruka zahteva, u koje upisuje instance klase *Transaction* (polja ova klase su funkcija koju transakcija treba da izvrši i t-promenljiva nad kojom on radi). U parnim porukama, koje se upućuju radniku *worker1* (preko reda *qreq1*), funkcija je *tx_inc* a t-promenljiva je 'tvar1', a u neparnim porukama koje se upućuju radniku *worker2* (preko reda *qreq2*), funkcija je *tx_dec* a t-promenljiva je 'tvar2'. Poruke zahteva klijent *client1* šalje radnicima, preko redova *qreq1*/*qreq2*, a nakon toga prima od njih odgovore preko redova *qrsp1*/*qrsp2*. Na kraju, klijent šalje posebne 'terminate' poruke radnicima da bi oni završili sa radom.

```
from multiprocessing import  Queue, Process
import logging
from stm import *

# constants
num_workers = 2
num_messages = 10

class Transaction:
    def __init__(self, function, l_tvar):
        self.function = function
        self.l_tvar = l_tvar

class Msg():
    def __init__(self, client_id, data, response, transaction):
        self.client_id = client_id
        self.data = data
        self.response = response
        self.transaction = transaction

def send(queue, message):
    queue.put(message)

def recieve(queue):
    return queue.get()
```

```
def tx_inc(stm_queue, l_tvar):
    tvar_name = l_tvar[0]

    tvarTuple = getVars(stm_queue, l_tvar)
    (tvarExistence, (tvarVersion, tvarValue)) = tvarTuple[0]
    tvarValue = tvarValue + 1

    R = []
    R.append((tvar_name, tvarVersion, tvarValue))
    W = []
    W.append((tvar_name, tvarVersion, tvarValue))
    ret_status = commitVars(stm_queue, [R, W])

    return (tvarValue, ret_status)

def tx_dec(stm_queue, l_tvar):
    tvar_name = l_tvar[0]

    tvarTuple = getVars(stm_queue, l_tvar)
    (tvarExistence, (tvarVersion, tvarValue)) = tvarTuple[0]
    tvarValue = tvarValue - 1

    R = []
    R.append((tvar_name, tvarVersion, tvarValue))
    W = []
    W.append((tvar_name, tvarVersion, tvarValue))
    ret_status = commitVars(stm_queue, [R, W])

    return (tvarValue, ret_status)

def WorkerServer(qreq, qrsp, id, stm_queue):
    ret_val = 0
    ret_status = None
    logging.basicConfig(filename='log' + str(id) + '.txt', filemode='w', level=logging.DEBUG)
    FORMAT = 'ID = %d, data = %s, response = %s, function = %s, l_tvar = %s'
```

```
    while True:
        m = recieve(qreq)
        #logging.info(FORMAT, m.client_id, m.data, m.response, m.function, m.l_tvar)
        if m.data == 'terminate':
            logging.info(FORMAT, m.client_id, m.data, m.response, '', '')
            m.response = 'terminate'
            send(qrsp, m)
            break
        else:
            logging.info(FORMAT, m.client_id, m.data, m.response, m.transaction.function,
                         m.transaction.l_tvar)
            # (ret_val, ret_status) = m.transaction.function(stm_queue, m.transaction.l_tvar)
            transaction = m.transaction
            function = transaction.function
            l_tvar = transaction.l_tvar
            (ret_val, ret_status) = function(stm_queue, l_tvar)
            m.response = 'status = ' + str(ret_status) + ', ' + str(m.transaction.l_tvar) + ' = ' +
                          str(ret_val)
            send(qrsp, m)

def RequestClient(lqreq, lqrsp):

    for i in range(num_messages):
        for j in range(num_workers):
            if j%2 == 0:
              l_tvar = ['tvar1']
              function = tx_inc
              transaction = Transaction(tx_inc, l_tvar)
            else:
               l_tvar = ['tvar2']
               function = tx_dec
               transaction = Transaction(tx_dec, l_tvar)

            m = Msg(1, 'message' + str(i) + ' ID: ' + str(j), '', transaction)
            send(lqreq[j], m)
```

```
            m = recieve(lqrsp[j])
            print(m.client_id, m.data, m.response)

    for j in range(num_workers):
        m = Msg(1, 'terminate', '', '')
        send(lqreq[j], m)
        m = recieve(lqrsp[j])
        print(m.client_id, m.data, m.response, '', '')

def InitializeSTM():
    q = Queue()
    ptm = new(q)

    addVars(q, ['tvar1', 'tvar2'])

    tvarVersion = 0
    tvarValue = 77
    putVars(q, [('tvar1', tvarVersion, tvarValue)])
    tvarValue = 88
    putVars(q, [('tvar2', tvarVersion, tvarValue)])

    return (q, ptm)

def DeinitializeSTM(q, ptm):
    print(delete(q))
    ptm.join()

def main():
    qreq1 = Queue()
    qrsp1 = Queue()
    qreq2 = Queue()
    qrsp2 = Queue()

    lqreq = []
    lqreq.append(qreq1)
    lqreq.append(qreq2)
```

```
    lqrsp = []
    lqrsp.append(qrsp1)
    lqrsp.append(qrsp2)

    #initialize STM
    (stm_queue, ptm) = InitializeSTM()

    worker1 = Process(target=WorkerServer, args=(qreq1, qrsp1, 0, stm_queue))
    worker2 = Process(target=WorkerServer, args=(qreq2, qrsp2, 1, stm_queue))

    client1 = Process(target=RequestClient, args=(lqreq, lqrsp))

    worker1.start()
    client1.start()
    worker2.start()

    worker1.join()
    client1.join()
    worker2.join()

    DeinitializeSTM(stm_queue, ptm)

if __name__ == '__main__':
    main()
```

A.2 Primer raspoređivača TM transakcija

Ovaj primer je veoma sličan predhodnom primeru. Osnovna razlika između njih je što u ovom primeru, glavni proces *main* u osnovi radi ono što je radio klijent *client1* u predhodnom primeru, odnosno formira *num_messages* (10) poruka zahteva za transakcijama (parne transakcije treba da izvrše funkciju *tx_inc* nad t-promeljivom 'tvar1', a neparne transakcije treba da izvrše funkciju *tx_dec* nad ovoga puta istom t-promeljivom 'tvar1'). Glavni proces *main* ove poruke zahteva šalje klijentu *scheduler* preko reda *txs_queue*.

Klijent *scheduler* izvršava funkciju *simpleRRScheduler* u kojoj zahteve raspoređuje u krug, tako što vadi zahteve iz reda *txs_queue*, parne zahteve šalje radniku *worker1* preko reda *lqreq*[0]

(a to je red *qreq1*), a neparne zahteve šalje radniku *worker2* preko reda *lqreq*[1] (a to je red *qreq2*). Radnici *worker1* i *worker2* izvršavaju zadate transakcije nad istom t-promenljivom konkurentno, tako postoje uslovi da dođe do konflikta, i u tom slučaju jedna od dve konkurentne transakcije će biti uspešna a druga neuspešna.

Nakon što klijent *scheduler* pošalje sve zahteve radnicima *worker1* i *worker2*, on čeka sve odgovore od njih (koje dobija preko radova *lqrsp*[0] i *lqrsp*[1]) i ispisuje rezultate na monitor. Na kraju klijent *scheduler* šalje posebne posebne 'terminate' poruke radnicima da bi oni završili sa radom.

```
from multiprocessing import  Queue, Process
import logging
from stm import *

# constants
num_workers = 2
num_messages = 10

class Transaction:
    def __init__(self, function, l_tvar, worker_id):
        self.function = function
        self.l_tvar = l_tvar
        self.worker_id = worker_id

class Msg():
    def __init__(self, client_id, data, response, transaction):
        self.client_id = client_id
        self.data = data
        self.response = response
        self.transaction = transaction

def send(queue, message):
    queue.put(message)

def recieve(queue):
    return queue.get()
```

```
def tx_inc(stm_queue, l_tvar):
    tvar_name = l_tvar[0]
    tvarTuple = getVars(stm_queue, l_tvar)
    (tvarExistence, (tvarVersion, tvarValue)) = tvarTuple[0]
    tvarValue = tvarValue + 1

    R = []
    R.append((tvar_name, tvarVersion, tvarValue))
    W = []
    W.append((tvar_name, tvarVersion, tvarValue))
    ret_status = commitVars(stm_queue, [R, W])

    return (tvarValue, ret_status)

def tx_dec(stm_queue, l_tvar):
    tvar_name = l_tvar[0]

    tvarTuple = getVars(stm_queue, l_tvar)
    (tvarExistence, (tvarVersion, tvarValue)) = tvarTuple[0]
    tvarValue = tvarValue - 1

    R = []
    R.append((tvar_name, tvarVersion, tvarValue))
    W = []
    W.append((tvar_name, tvarVersion, tvarValue))
    ret_status = commitVars(stm_queue, [R, W])

    return (tvarValue, ret_status)

def WorkerServer(qreq, qrsp, id, stm_queue):
    ret_val = 0
    ret_status = None
    logging.basicConfig(filename='log' + str(id) + '.txt', filemode='w', level=logging.DEBUG)
    FORMAT = 'ID = %d, data = %s, response = %s, function = %s, l_tvar = %s'

    while True:
```

```
        m = recieve(qreq)
        #logging.info(FORMAT, m.client_id, m.data, m.response, m.function, m.l_tvar)
        if m.data == 'terminate':
            logging.info(FORMAT, m.client_id, m.data, m.response, '', '')
            m.response = 'terminate'
            send(qrsp, m)
            break
        else:
            logging.info(FORMAT, m.client_id, m.data, m.response, m.transaction.function,
                         m.transaction.l_tvar)
            # (ret_val, ret_status) = m.transaction.function(stm_queue, m.transaction.l_tvar)
            transaction = m.transaction
            function = transaction.function
            l_tvar = transaction.l_tvar
            (ret_val, ret_status) = function(stm_queue, l_tvar)
            #m.response = 'status = ' + str(ret_status) + ', ' + str(m.transaction.l_tvar) + ' = ' +
                          str(ret_val)
            m.response = str(ret_status) + ', ' + str(m.transaction.l_tvar) + ' = ' + str(ret_val)
            send(qrsp, m)

def simpleRRScheduler(lqreq, lqrsp, txs_queue):

    # outmost loop
    while not txs_queue.empty():
        print('start execution round')
        # j is the index of the current worker
        j = 0
        num_tx = []

        for k in range(num_workers):
            num_tx.append(0)

        while not txs_queue.empty():
            m = recieve(txs_queue)
            send(lqreq[j], m)
            num_tx[j] = num_tx[j] + 1
```

```
            j = (j + 1) % num_workers

        #print('num_tx =', num_tx)

        for k in range(num_workers):
            #print('k = ', k)
            for j in range(num_tx[k]):
                #print('j = ', j)
                m = recieve(lqrsp[k])
                print(m.response, 'worker = ' + str(m.transaction.worker_id), 'j = ', j)
                yesno = m.response[0:6]
                #print('yesno = ', yesno)
                if yesno == "['no']":
                    print('put aborted txn into txs_queue')
                    m.response = ''
                    send(txs_queue, m)

    # end of outmost loop
    for j in range(num_workers):
        m = Msg(1, 'terminate', '', '')
        send(lqreq[j], m)
        m = recieve(lqrsp[j])
        print(m.client_id, m.data, m.response, '', '')

def InitializeSTM():
    q = Queue()
    ptm = new(q)

    addVars(q, ['tvar1', 'tvar2'])

    tvarVersion = 0
    tvarValue = 77
    putVars(q, [('tvar1', tvarVersion, tvarValue)])
    tvarValue = 88
    putVars(q, [('tvar2', tvarVersion, tvarValue)])
```

```
    return (q, ptm)

def DeinitializeSTM(q, ptm):
    print(delete(q))
    ptm.join()

def main():
    # Queues between tha scadulers and the workers
    qreq1 = Queue()
    qrsp1 = Queue()
    qreq2 = Queue()
    qrsp2 = Queue()

    lqreq = []
    lqreq.append(qreq1)
    lqreq.append(qreq2)

    lqrsp = []
    lqrsp.append(qrsp1)
    lqrsp.append(qrsp2)

    txs_queue = Queue()

    #initialize STM
    (stm_queue, ptm) = InitializeSTM()

    worker1 = Process(target=WorkerServer, args=(qreq1, qrsp1, 0, stm_queue))
    worker2 = Process(target=WorkerServer, args=(qreq2, qrsp2, 1, stm_queue))

    scheduler = Process(target=simpleRRScheduler, args=(lqreq, lqrsp, txs_queue))

    worker1.start()
    worker2.start()
    scheduler.start()

    # Create transaction and send them to txs_queue
```

```
    for i in range(num_messages):
        for j in range(num_workers):
            if j%2 == 0:
                l_tvar = ['tvar1']
                function = tx_inc
                transaction = Transaction(tx_inc, l_tvar, j)
            else:
                # l_tvar = ['tvar2']
                l_tvar = ['tvar1']
                function = tx_dec
                transaction = Transaction(tx_dec, l_tvar, j)

            m = Msg(1, 'message' + str(i), '', transaction)
            send(txs_queue, m)

    # Scheduler will not return a message for terminate, it will just terminate!
    #m = Msg(1, 'terminate', '', '')
    #send(txs_queue, m)

    worker1.join()
    worker2.join()
    scheduler.join()

    DeinitializeSTM(stm_queue, ptm)

if __name__ == '__main__':
    main()
```