UNIVERZITET U BEOGRADU
ELEKTROTEHNIČKI FAKULTET

Živojin B. Šuštran

POBOLJŠANJE PERFORMANSI ASIMETRIČNIH VIŠEJEZGARNIH PROCESORA KROZ MIGRACIJU TRANSAKCIJA I PRILAGOĐENJE PODSISTEMA KEŠ MEMORIJA

doktorska disertacija

Beograd 2021.

UNIVERSITY OF BELGRADE
SCHOOL OF ELECTRICAL ENGINEERING

Živojin B. Šuštran

IMPROVING THE PERFORMANCE OF ASYMMETRIC MULTICORE PROCESSORS BY TRANSACTIONS' MIGRATION AND THE ADAPTATION OF THE CACHE SUBSYSTEM

Doctoral Dissertation

Belgrade, 2021.

Mentori:

dr Jelica Protić, redovni profesor
Univerzitet u Beogradu – Elektrotehnički fakultet

dr Milo Tomašević, redovni profesor
Univerzitet u Beogradu – Elektrotehnički fakultet

Članovi komisije:

dr Jelica Protić, redovni profesor
Univerzitet u Beogradu – Elektrotehnički fakultet

dr Milo Tomašević, redovni profesor
Univerzitet u Beogradu – Elektrotehnički fakultet

dr Miloš Cvetanović, vanredni profesor
Univerzitet u Beogradu – Elektrotehnički fakultet

dr Miloš Kovačević, redovni profesor
Univerzitet u Beogradu – Građevinski fakultet

dr Zaharije Radivojević, vanredni profesor
Univerzitet u Beogradu – Elektrotehnički fakultet

Datum odbrane: _____

Zahvalnica

Zahvaljujem se mojim mentorima, profesorima dr Jelici Protić, dr Milu Tomaševiću i dr Veljku Milutinoviću koji su uvek verovali u mene i koji su mi svojom stalnom podrškom pomogli da istrajem na ovom dugom putom.

Zahvalnost dugujem i svim učiteljima, nastavnicima i profesorima koji su me podučavali od osnovne škole do doktorskih studija. Tu uključujem i sve bivše i sadašnje članove Katedre za računarsku tehniku i informatiku koji su me uveli u svet računarske nauke.

Ovom istraživanju su mnogo doprineli i kolege sa kojima sam pisao naučne radove Dražen Drašković, Goran Rakočević, Marko Mišić, Saša Stojanović, Sofija Purić, Marko Mićović, Uroš Radenković i Vladimir Jocović. Velika je čast i zadovoljstvo bilo sarađivati, putovati i družiti se sa vama.

Moja porodica je bila moj najveći oslonac tokom celog mog života. Supruzi Jeleni i kćerki Teodori hvala na ljubavi, strpljenju i motivaciji koju su mi dale. Na kraju, najviše se zahvaljujem mami i tati za bezuslovnu ljubav i sve što su uložili u moje obrazovanje. Hvala za nebrojane dane vežbanja zadataka i učenja lekcija.

Naslov teze: Poboljšanje performansi asimetričnih višejezgarnih procesora kroz migraciju transakcija i prilagođenje podsistema keš memorija

Rezime: Postojeći pravci razvoja računarstva imaju za cilj da se performanse računarskih sistema podignu na što viši nivo, da bi se zadovoljile potrebe za obradom velike količine podataka. Pažnja je usmerena na razvoj procesora kao glavne karike u procesu obrade podataka. Trendovi poboljšanja performansi procesora predviđeni Murovim zakonom u poslednje vreme se usporavaju usled fizičkih ograničenja poluprovodničke tehnologije, te poboljšanje performansi postaje sve teže i teže. Taj problem se pokušava nadomestiti raznim tehnikama koje imaju za cilj poboljšanje performansi bez povećanja broja tranzistora i potrošnje energije.

U ovoj disertaciji su razmatrani asimetrični višejezgarni procesori sa podrškom za transakcionu memoriju. Predložene su dve nove tehnike za poboljšanje performansi takvih procesora. Jedna tehnika služi za smanjenje zagušenja transakcija usled velikog paralelizma migracijom transakcija na brže jezgro. Transakcije koje najviše doprinose pojavi zagušenja, se biraju za migraciju. Njihovim izvršavanjem na bržem jezgru se smanjuje verovatnoća da će doći u konflikt sa drugim transakcijama. Na taj način se povećava šansa za izbegavanja zagušenja. Druga tehnika prilagođava podsistem keš memorija tako da se smanji vreme pristupa do keš memorije i da se smanji šansa za pojavu lažnih konflikata, uz smanjenje broja tranzistora koji su potrebni za implementaciju keš memorije. To se može postići korišćenjem malih i jednostavnih keš memorija.

Za obe tehnike dati su detaljni predlozi implementacije. Napravljeni su prototipovi tih tehnika u simulatoru Gem5, koji detaljno modeluje procesorski sistem. Pomoću prototipova je izvršena evaluacija predloženih tehnika simuliranjem velikog broja aplikacija iz standardnog skupa aplikacija za testiranje transakcione memorije. Analizom rezultata simulacije preporučeno je na koji način i kada treba koristiti predložene tehnike.

Ključne reči: asimetrični višejezgarni procesori, homogeni instrukcijski set, transakciona memorija, podsistem keš memorija, Gem5

Naučna oblast: Elektrotehnika i računarstvo

Uža naučna oblast: Računarska tehnika i informatika

UDK broj: 621.3:004.2

Dissertation title: Improving the performance of asymmetric multicore processors by transactions' migration and the adaptation of the cache subsystem

Abstract: Existing trends in computer design aim to raise the performance of computer systems to the highest possible level in order to meet the needs for processing large amounts of data. Attention is focused on the design of a processor as the main actor in the data processing process. Improvement trends in processor performance predicted by Moore's Law has been slowing down recently due to physical limitations of semiconductor technology and increasing performance is getting harder and harder. This problem is attempted to be compensated by various techniques aimed at improving performance without increasing transistor and power consumption.

In this thesis, asymmetric multicore processors with support for transactional memory are considered. Two new techniques have been proposed to increase the performance of such processors. One technique aims to reduce transaction congestion due to high parallelism by migrating transactions to a faster core. The transactions that contribute the most to an occurrence of congestion are selected for migration. Executing them on a faster core reduces their chances of conflict with other transactions and thus increases the chance of avoiding congestion. Another technique adjusts the cache subsystem to reduce caches' access latency and to reduce the chances of false conflicts while reducing the number of transistors required to implement the cache. This can be achieved by using small and simple caches.

Detailed implementation proposals are given for both techniques. Prototypes of these techniques were made in the Gem5 simulator, which models processor's system in detail. Using prototypes, the proposed techniques were evaluated by simulating a large number of applications from a standard benchmark set for transactional memory. The analysis of the simulation results gave suggestions on how and when the proposed techniques should be used.

Keywords: asymmetric multiprocessors, homogeneous instruction set, transactional memory, cache subsystem, Gem5

Research area: Electrical Engineering and Computing

Research sub-area: Computer Engineering and Informatics

UDC number: 621.3:004.2

Sadržaj

Poglavlje 1

Uvod

U poslednjoj dekadi evidentni su trendovi tehnologije i aplikacija za prevazilaženje ograničenja u pogledu energije i disipacije (eng. *power wall*), sve većih razlika u brzinama procesora i memorije (eng. *memory wall*), kao i sve manjih mogućnosti korišćenja paralelizma na nivou instrukcije (eng. *ILP wall*). Poboljšanje performansi procesora postalo je težak zadatak i stalno se istražuju nova rešenja u arhitekturi i organizaciji procesora da bi se taj problem prevazišao. Pored arhitekturalnih poboljšanja, poboljšanje performansi procesora se do sada najviše zasnivalo na povećanju frekvencije signala takta. Povećanje frekvencije rezultuje povećanjem dinamičke snage. Zbog tog se prešlo na smeštanje više prostijih procesora na jedan čip i razvoj višejezgarnih procesora (eng. *multicore processors*). Povećanjem broja jezgara na jednom čipu postižu se izvesna ubrzanja, ali ne više tako drastična kao ranije.

Najveći problem je što su se aplikacije dominantno razvijale za izvršavanje na jednom procesoru (sekvencijalne), pa se ubrzanje dobijalo samo ako postoji više programa koji mogu da se izvršavaju u paraleli. U nedostatku pravih paralelnih aplikacija, resursi višejezgarnog procesora često ostaju neiskorišćeni. Potreba da se aplikacije paralelizuju radi poboljšanja performansi i boljeg iskorišćenja resursa znatno otežava njihovo programiranje i testiranje.

Pojava višejezgarnih procesora nameće veće zahteve u pogledu propusne moći memorijskog sistema i efektivnog vremena pristupa podacima. Osim toga, transfer podataka između memorije i procesora traje nekoliko redova veličine duže nego što traje obrada tih podataka. Taj problem se rešavao stavljanjem malih lokalnih memorija, tzv. keš memorija (eng. *cache memory*), u sam procesor, koje služe da se podaci koji se aktivno koriste nalaze na samom čipu i na taj način im se pristupa mnogo brže. One se obično organizuju u hijerarhije sa više nivoa. Povećanjem broja jezgara raste i potreba za podacima koji se aktivno koriste, što znači da kapacitet keš memorije treba da se srazmerno povećava. To povećanje nije lako postići, jer veći kapacitet keš memorije znači i povećanje vremena pristupa, što degradira performanse samog procesora.

Razni pristupi rešavanju ovih problema se mogu naći u otvorenoj literaturi, što uključuje inovativne tehnike za projektovanje keš memorija. Jedna grupa rešenja se bazira na korišćenju više različitih vrsta keš memorija koje služe za prilagođenje keš memorija zbog različitih vrsta podataka [1], [2]. Problem otežanog programiranja i testiranja aplikacija se može rešiti prebacivanjem odgovornosti za ispravno funkcionisanje paralelnih aplikacija sa programera na sam višejezgarni procesor primenom programskog modela transakcione memorije (eng. *transactional memory*) [3]. Osnovna ideja je da programer samo označi koji delovi aplikacije moraju atomično da se izvrše u odnosu na ostale delove koji se izvršavaju na drugim jezgrima. Ovi atomični delovi se nazivaju transakcijama. Jezgro detektuje kada se neka transakcija nije atomično izvršila, pa je ponavlja sve dok se ne izvrši atomično. Postojeća istraživanja na temu transakcione memorije su aktuelna i ogromna i pokrivaju rešenja koja se mogu implementirati u hardver, u softver ili u nekoj hibridnoj varijanti koja uključuje i hardver i softver. Ovo istraživanje se bavi hardverskom transakcionom memorijom (HTM) koja je zastupljena i u komercijalno dostupnim procesorima.

Jedna arhitektura koja omogućava efikasnije iskorišćenje višejezgarnog procesora sa stanovišta potrošnje energije su asimetrični višejezgarni procesori [4]. Ovi procesori se sastoje od više jezgara koja nisu istovetne arhitekture, pa su neka brža, a neka sporija za određene namene. Ideja je da se na bržim jezgrima izvršavaju sekvencijalni delovi aplikacija koji ne mogu da iskoriste postojanje više jezgara, a da se sporija jezgra koriste za paralelne delove aplikacija. Jedan deo postojećih rešenja predlaže asimetrične višejezgarne procesore čija jezgra imaju identičan instrukcijski skup (eng. *instruction set*). Jezgra u tim rešenjima se razlikuju u organizaciji. Kod nekih instrukcije mogu da se izvršavaju van programskog poretka, a kod nekih ne, neka jezgra mogu da istovremeno izvršavaju više instrukcija u paraleli, neka jezgra imaju veće keš memorije, itd. Takvi procesori se nazivaju asimetrični višejezgarni procesori sa jednim instrukcijskim setom (eng. *Single-ISA asymmetric multicore processors*) [4]. Postojanje ovakve asimetričnosti može da se iskoristi za poboljšanje performansi tako što će izvršavanje jedne niti prebacivati, odnosno migrirati, na pogodno jezgro [5]. Prebacivanje može da se radi na nivou niti [6]–[12] ili na nivou dela niti [5], [13]–[15]. Obično se premeštanje cele niti radi pomoću raspoređivača operativnog sistema, dok se deo nit prebacuje pomoću hardvera.

Cilj ovog istraživanja je da se kombinacijom gore navedenih tehnika utvrdi da li se može dobiti ubrzanje izvršavanja aplikacija na asimetričnim višejezgarnim procesorima. S obzirom na to da su tehnike ortogonalne, potrebno je osmisliti efikasan način za dobijanje sinergije u njihovom kombinovanju. Specifična organizacija podsistema keš memorija treba da omogući veću dostupnost podataka procesoru i da doprinese da se deo aplikacije na jednom procesoru brže izvršava, bez značajnog povećanja samog podsistema keš memorija. Pored toga, cilj je da se utvrdi kako će taj novi način organizacije uticati na primenu tehnika transakcione memorije i asimetričnih višejezgarnih procesora. S obzirom na to da performanse aplikacija, koje koriste transakcionu memoriju mogu da se pogoršaju, ako se transakcije često ponavljaju, treba da se ispita i da li novom metodom, prilagođenja keš memorija i migracijom transakcija koje su kritične, može da se smanji njihov broj ponavljanja. Namera istraživanja je da se formuliše algoritam koji vrši migraciju transakcija po potrebi i ispita njegov način realizacije u višejezgarnom procesoru. Migracijom transakcija na brže jezgro smanjio bi se broj njihovog ponavljanja, jer je manja verovatnoća da će se izvršavati uporedo sa drugim transakcijama, ako se transakcija brže izvrši.

S obzirom na to da su neke od navedenih tehnika već primenjene u proizvodima najboljih svetskih proizvođača procesora, a neke su predložene u istraživačkim studijama, ovaj rad će težiti sinergističkoj primeni modernih koncepata u cilju postizanja boljih performansi. Značaj istraživanja je u tome što će se osmisliti novi način projektovanja višejezgarnih procesora koji će omogućiti brže izvršavanje aplikacija.

Iako su oblasti transakcione memorije i asimetričnih višejezgarnih procesora veoma aktuelne i zastupljene u otvorenoj literaturi, nedostaju detaljnija istraživanja mogućnosti njihovog kombinovanja, koje je u fokusu ove disertacije. Međutim, neka prethodna istraživanja mogu da se kombinuju sa rešenjima prikazanim u ovoj disertaciji, jer poboljšavaju ili samo asimetrične procesore ili samo transakcionu memoriju. Rešenja kojima se bavi ova disertacija su namenjena za implementiranje isključivo u hardver i u potpunosti su transparentna za programera, drugim rečima, nije potrebno menjati instrukcijski skup i kod aplikacija, što znači da postojeće aplikacije mogu da rade bez bilo kakvih modifikacija.

Ovaj rad sadrži osam poglavlja.

Tekuće poglavlje (Poglavlje 1) daje osnovne informacije o aktuelnim pravcima istraživanja vezanim za arhitekturu procesora i tehnikama koje se koriste da bi se poboljšale performanse procesora i da bi se olakšalo višenitno programiranje. Pored toga navodi osnovni cilj ovog istraživanja.

U Poglavlju 2, naveden je detaljan opis arhitekturalnih tehnika koje se istražuju. Definisani su pojmovi koji se koriste kasnije u ostalim poglavljima.

Detaljan pregled postojećih rešenja dat je u Poglavlju 3. Opisani su doprinosi najznačajnih radova. Razmatrani su prednosti i nedostaci pojedinih rešenja. Izvršena je klasifikacija postojećih rešenja. Na kraju, navedeno je kako se predloženo rešenje u ovoj disertaciji uklapa u kategorije postojećih rešenja

i koje novine donosi.

U Poglavlju 4, opisani su problemi koji se žele rešiti u ovoj disertaciji. Dati su motivi za njihovo rešavanje i mogući načini za njihovo rešavanje. Navedene su osnovne hipoteze ove disertacije i koji su osnovi za njihovo formulisanje. Poglavlje je podeljeno u dva dela. Prvi deo razmatra problem migracije transakcija, dok drugi deo poglavlja razmatra prilagođenje podsistema keš memorija.

Detaljan opis predloženih rešenja dat je u Poglavlju 5. Prvo se opisuje algoritam predloženih rešenja na višem nivou apstrakcije. Nakon toga se razmatra kako je moguće implementirati predložena rešenja. Prikazana je detaljna arhitektura predloženih rešenja i cena implementacije. Poglavlje je podeljeno u dva dela. Prvi deo prikazuje rešenje za migriranje transakcija, dok drugi deo poglavlja prikazuje rešenje za prilagođeni podsistem keš memorija.

Postupak koji je korišćen za eksperimentalnu analizu naveden je u Poglavlju 6. Koristi se simulator koji simulira procesore zajedno sa memorijom na nivou procesorskih ciklusa. Detaljno su navedeni načini na koje se vrše simulacije. Opisani su detalji implementacije simulatora. Na kraju, dat je postupak za pronalaženje grešaka u toku implementiranja simulatora i navedeno je koji uslovi treba da budu ispunjeni da bi se smatralo da simulator ispravno radi.

U Poglavlju 7, opisani su sprovedeni eksperimenti koji služe za potvrđivanje hipoteza. Prvo je navedena eksperimentalna postavka. U okviru nje navedeni su detaljno parametri simuliranog sistema i koje aplikacije su korišćene za evaluaciju predloženih rešenja. Nakon toga su navedeni rezultati eksperimenata. Na kraju je data diskusija o dobijenim rezultatima i šta se iz njih može zaključiti. Poglavlje je podeljeno u dva dela. Prvi deo prikazuje eksperimente sa migriranjem transakcija, dok drugi deo poglavlja pokazuje eksperimente sa prilagođenim podsistemom keš memorija.

Na kraju, Poglavlje 8 sumira rezultate prikazane u okviru ove disertacije i prikazuje glavne elemente disertacije, zaključke i pravce daljeg istraživanja.

Poglavlje 2

Arhitekturalne tehnike za poboljšanje performansi procesora

U ovom poglavlju opisana su tri elementa značajna za unapređenje performansi savremenih višejezgarnih procesora: podsistem keš memorija, transakciona memorija i asimetrični višejezgarni procesori. Za svaku od njih opisano je čemu služe, kako se prave i kako se koriste.

2.1 O podsistemu keš memorija

Keš memorija služi da se premosti jaz koji nastaje zbog razlike u brzini između procesora i operativne memorije [16], [17]. Procesor je oko sto puta brži [17] i da bi iskoristio tu svoju brzinu na raspolaganju mora imati podatke kojima može brzo da pristupi. Keš memorija se stavlja na sam čip pored procesora i u njoj se nalaze podaci koje je procesor najskorije koristio. Veličina keš memorije ne može biti velika, da bi brzina pristupa bila usaglašena sa brzinom procesora.

U početku stavljala se samo jedna keš memorija. Kako se jaz u brzini između procesora i operativne memorije povećavao, bilo je potrebno sve više i više podataka čuvati na samom čipu da bi procesor mogao da radi maksimalno brzo. Iz tog razloga dodavane su različite keš memorije na čip. Procesor pristupa najmanjoj keš memorija. U tu keš memoriju se dohvataju podaci iz veće keš memorije, pa ta keš memorija iz sledeće veće, itd. Najveća keš memorija dohvata podatke iz operativne memorije. Drugim rečima, dodato je nekoliko nivoa keš memorija da bi moglo više podataka da se čuva na čipu, a u isto vreme da se brzina pristupa ne smanji. Najmanja keš memorija, koja je ujedno i najbliža procesoru, nalazi se u najnižem nivou.

Radi lakše implementacije takvog podsistema keš memorija uveden je princip da podatak, koji se nalazi u jednoj keš memoriji, mora da se nalazi i u svim keš memorijama koji se nalaze u višim nivoima. Taj princip naziva se princip inkluzije (eng. *inclusion policy*). Podsistem keš memorija može da se implementira i bez poštovanja principa inkluzije, radi boljih performansi ili boljeg iskorišćenja keš memorija.

U keš memoriju ne mogu da stanu svi podaci. Iz tog razloga mora da se zna gde se podatak smešta i šta se radi kada potreban podatak nije u kešu, a mesto gde bi ga trebalo smestiti je zauzeto. Podatak se smešta u ulaz keš memorije na osnovu njegove adrese u operativnoj memoriji (ili na osnovu virtualne adrese). Mapiranje adrese u ulaz je moguće uraditi na više načina. Najjednostavniji način za implementaciju je direktno mapiranje (eng. *direct mapping*). Podatak se može smestiti samo u jedan ulaz keš memorije. Taj ulaz se određuje na osnovu dela adrese podatka. Mana tog pristupa je što je dovoljno da postoji jedan podatak u memoriji da nema mesta za drugi podatak, ako ta dva podatka imaju isti deo adrese na osnovu koga se određuje u koji ulaz će biti smešten podatak. Dobre strane tog pristupa su što je dovoljno pored podatka pamtiti samo deo adrese, tzv. tag, koji ne učestvuje u određivanju ulaza i što se provera da li se podatak nalazi u keš memoriji radi samo jednostavnim poređenjem jednog taga.

Drugi način za mapiranje je asocijativno mapiranje (eng. *associative mapping*). Podatak može da se nađe u bilo kom ulazu keš memorije. Dobra strana tog mapiranja je što tek kada su svi ulazi u keš memoriji zauzeti, nema mesta za sledeći podatak. Loše strane su što za tag mora da se pamti cela adresa i što provera da li se podatak nalazi u keš memoriji mora da se radi poređenjem svih tagova u keš memoriji sa adresom podatka koji se traži. To može značajno da poveća vreme pristupa keš memoriji.

Mapiranje koje pokušava da reši mane i jednog i drugog mapiranja, a iskoristi njihove prednosti naziva se set-asocijativno mapiranje (eng. *set associative mapping*). Ideja je da podatak može da bude u skupu od nekoliko ulaza. Što je veličina tog skupa veća, mapiranje se približava asocijativnom mapiranju (veličina skupa je jednaka broju ulaza u keš memoriji kod asocijativnog mapiranja). Što je veličina tog skupa manja, mapiranje se približava direktnom mapiranju.

Kada u keš memoriji nema mesta da se smesti podatak, mora se odabrati ulaz iz koga će se izbaciti podatak da bi se napravilo mesto za novi podatak. Kod direktnog mapiranja postoji samo jedan ulaz kao kandidat. Kod ostalih načina mapiranja, bira se jedan ulaz iz skupa ulaza u kojima podatak može da se nađe. Postoje razni algoritmi za odabir jednog ulaza za zamenu. Najčešće se koristi algoritam najdavnije korišćen (eng. *least recently used*). Taj algoritam odabira onaj ulaz kome se najdavnije pristupalo, računajući na to da taj podatak više neće biti potreban. Što je veći skup ulaza iz koga algoritam treba da odabere jedan, to je teže implementirati algoritam (ponekad čak i nemoguće). Iz tog razloga koriste se algoritmi koji imaju istu ideju, ali nju sprovode aproksimativno [18], a ne potpuno tačno.

U keš memoriji podaci mogu da se čuvaju na proizvoljnoj širini. Obično se čuvaju na širini od nekoliko reči. Taj niz reči naziva se blok. Time se postiže da keš memorija ima manje informacija za čuvanje, jer je širina adrese jednog bloka manja od širine adrese jedne reči, te je broj ulaza u keš memoriji manji.

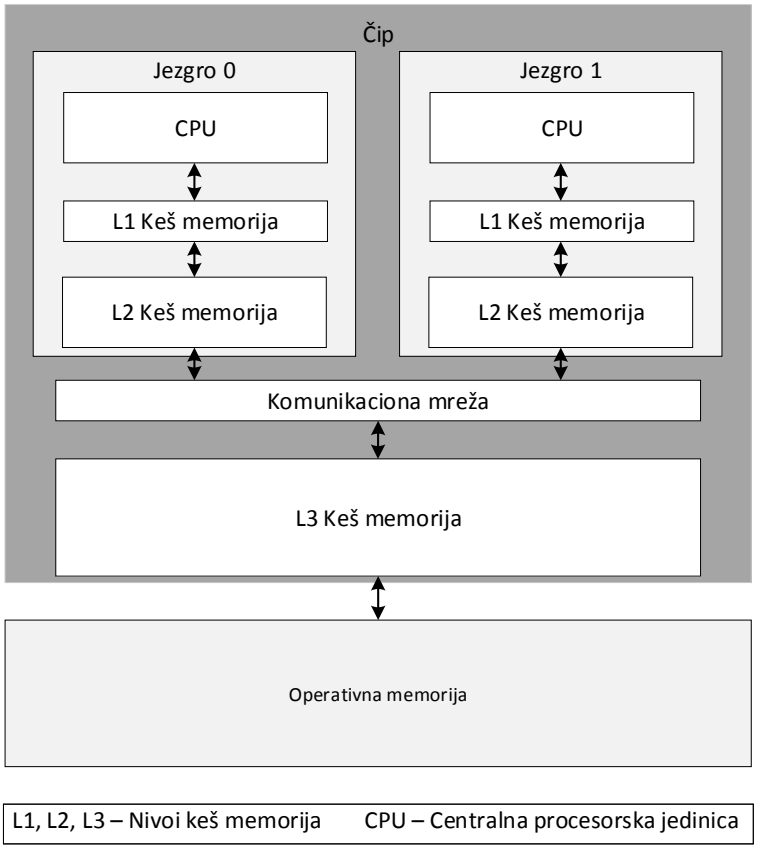
Kada procesor zatraži podatak iz keš memorije, taj podatak može da bude u keš memoriji ili da ne bude. Ako je podatak u keš memoriji to se naziva pogodak (eng. *cache hit*), a ako nije to se naziva promašaj (eng. *cache miss*). Što je veći procenat pogodaka od svih pristupa, to procesor brže izvršava instrukcije.

Kod višejezgarnog procesora više jezgara može da pristupa keš memorijama. Obično keš memorije najnižih nivoa služe za čuvanje podataka samo jednog jezgra i one se nazivaju privatne keš memorije, a nivoi privatni nivoi. Keš memorije višeg nivoa su obično deljenje između više jezgara. Na taj način privatne keš memorije imaju samo jedan port prema jednom jezgru i ne moraju da budu velike. Jedan takav sistem prikazan je na slici 2.1. To olakšava pravljenje keš memorija koje će biti brze.

Postojanje privatnih keš memorija stvara problem koherencije podataka (eng. *data coherence*) [19]. Jedan podatak može da se koristi od strane dva jezgra i da se nalazi u njihovim privatnim keš memorijama. Ako neko jezgro promeni podatak, ta promena nije vidljiva drugom jezgru. To drugo jezgro će da koristi zastarelu verziju koja se nalazi u njegovoj privatnoj keš memoriji. Korišćenje zastarele verzije podatka dovodi do logički neispravnog izvršavanja. Da bi se taj problem prevazišao osmišljeni su protokoli keš koherencije [20]–[22].

Osnovna ideja protokola keš koherencije je da kada se pristupa nekom podatku u keš memoriji, ta keš memorija "javi" drugim keš memorijama da se dogodio pristup i koja je vrsta pristupa. Ta druga keš memorija će odreagovati na odgovarajući način i sprečiti korišćenje nevalidnog podatka. Recimo, ako se menja neki podatak tada keš memorija šalje informaciju ostalim keš memorijama da se podatak menja, kako bi one poništile lokalne kopije tog podatka koje eventualno poseduju. Kada je taj podatak poništen u svim ostalim keš memorijama, prva keš memorija može da promeni podatak. Na taj način najnovija verzija podatka će se naći samo u jednoj privatnoj keš memoriji. Ako taj podatak zatreba nekom drugom jezgru, protokol keš koherencije će obezbediti da se podatak prekopira iz one keš memorije koja ima taj podatak.

Postoji veliki broj protokola keš koherencije. Svi oni pokušavaju da smanje broj poništavanja



Slika 2.1: *Višejezgarni procesor sa dva nivoa privatnih keš memorija i jednim deljenim nivoom*

podataka, broj poruka koje se šalju, prostor za smeštanje informacija koje služe za održavanje keš koherencije, itd. Protokoli se obično dele u dve grupe. Prva je grupa gde jedna keš memorija obaveštava sve ostale keš memorije o pristupu. Takvi protokoli se nazivaju osluškujući (eng. *snooping protocol*). Drugu grupu čine protokoli kod kojih jedna keš memorija obaveštava samo one keš memorije koje je neophodno obavestiti da bi se održala keš koherencija. To se postiže vođenjem evidencije gde se određeni podatak sve nalazi. Takvi protokoli se nazivaju protokoli bazirani na katalogu (eng. *directory-based protocol*). Protokoli prve grupe zahtevaju slanje više poruka, dok protokoli druge grupe zahtevaju više prostora na čipu za smeštanje informacija potrebnih za održavanje keš koherencije.

U višejezgarnom sistemu pored problema koherencije podataka, postoji i problem memorijske konzistencije (eng. *memory consistency*). Dok protokol keš koherencije ima zadatak da redosled menjanja jednog podatka bude vidljiv na isti način kod svih učesnika (onim redosledom kojim su se ti upisi i dogodili), memorijska konzistencija definiše kako će biti vidljiv redosled menjanja podataka na različitim adresama [23]. Striktna memorijska konzistencija (eng. *strict consistency*) definiše da redosled koji je vidljiv svim učesnicima mora biti isti kao onaj koji se i stvarno dogodio [24].

Međutim, postoje i predlozi relaksirane memorijske konzistencije. Postoje modeli slabe, oslobađajuće, lenjo oslobađajuće i ulazne konzistencije [25]. Striktnu konzistenciju je teško implementirati, dok relaksiranom konzistencijom mogu da se dobiju bolje performanse. U slučaju relaksirane konzistencije programer mora da vodi računa o konzistenciji koju želi da postigne primitivama za sinhronizaciju. Različiti modeli garantuju konzistenciju posle nekih sinhronizacionih primitiva. Kod modela slabe konzistencije sve primitive garantuju konzistenciju, kod oslobađajuće samo one primitive koje otključavaju bravu, dok kod ulazne samo primitive koje zaključavaju bravu.

Jedan primer ,kada je potrebno voditi računa o konzistenciji, je kada jezgro izvršava instrukcije van programskog poretka. Može se dogoditi da dva upisa u dve različite adrese budu uočeni u različitom redosledu na dva različita jezgra. Ako to predstavlja problem, programer može da doda instrukciju koja se naziva memorijska barijera (eng. *memory barrier*). Ta instrukcija osigurava da svi

upisi koji se dešavaju u programskom poretku pre nje će biti vidljivi svim ostalim jezgrima, kao da su se dogodili pre upisa koji se nalaze u programskom poretku posle memorijske barijere. To se postiže tako što jezgro sačeka sa izvršavanjem instrukcija koje se nalaze posle memorijske barijere, dok upisi koji su se dogodili pre memorijske barijere ne postanu vidljivi svim ostalim jezgrima. Očigledno često korišćenje memorijske barijere može da degradira performanse izvršavanja.

2.2 Mehanizam transakcione memorije

Programiranje višenitnih (eng. *multithreaded*) programa je znatno teže nego programiranje jednonitnih programa. Problemi koji se teže rešavaju kod višenitnih programa su: 1) održavanje logičke ispravnosti i 2) postizanje zadovoljavajućih performansi programa. Održavanje logičke ispravnosti aplikacije je otežano zbog konkuretnog pristupa deljenim podacima. Konkurentni (eng. *concurrent*) programi su oni programi čiji se delovi izvršavanja međusobno preklapaju u vremenu [26]. Ako jedna nit čita deljene podatke dok ih druga menja, rezultat izvršavanja je nepredvidljiv, jer poredak operacija čitanja i upisa zavisi od mnogo faktora (brzine izvršavanja niti, čekanje niti na ulazno/izlazne operacije, prioriteta niti, itd.) koji se mogu promeniti između dva izvršavanja pa čak i u toku samog izvršavanja programa. Zato o tim faktorima ne sme ništa da se pretpostavlja prilikom pisanja programa [27]. Rezultat izvršavanja tada zavisi od toga koja nit pre počne da koristi deljene podatke i taj problem se naziva utrkivanje (eng. *race condition*).

Jednonitni program se smatra logički ispravnim ako ispunjava sledeće: 1) rezultat izvršavanja ne zavisi od brzine izvršavanja programa i 2) rezultat izvršavanja je uvek isti za isti set ulaznih podataka [26]. Višenitni program se smatra logički ispravnim ako daje isti rezultat kao i jednonitni program koji rešava isti problem. Pošto utrkivanje prouzrokuje više različitih mogućih rezultata, smatra se da višenitni program nije logički ispravan ukoliko postoji utrkivanje. Problem utrkivanja se rešava podelom programa u celine koje moraju atomično (eng. *atomic*) da se izvrše u odnosu na druge celine koje pristupaju istim podacima i koje, ako se izvrše u proizvoljnom redosledu, daju isti rezultat kao i jednonitni program. Atomična celina treba da izvršavanje programa prevede iz jednog konzistentog stanja u drugo konzistentno stanje.

Dodatni teret za programera je što mora, pored rešavanja problema zbog koga se program pravi, da obezbedi da ne dođe do utrkivanja i da obezbedi da atomične celine zadovolje potrebne osobine konzistentnosti. Obezbeđivanje atomičnosti može da utiče na performanse programa. Dok atomična celina pristupa deljenim podacima, ni jedna druga celina koja pristupa tim istim podacima ne sme da im pristupi. Time paralelizam može značajno da se smanji ako je izvršavanje atomičnih celina često i ako su atomične celine velike, što može da prouzrokuje da se program sporije izvršava. U najgorem slučaju to može dovesti do potpune sekvencijalizacije programa kada samo jedna nit u jednom trenutku može da se izvršava i napreduje. Što je atomična celina veća, veća je verovatnoća da druge niti neće moći u paraleli da se izvršavaju i samim tim veća je verovatnoća da će se program sporije izvršiti. Programer mora da smanjuje atomične celine, da bi postigao zadovoljavajuće performanse programa, što predstavlja dodatan teret. Ukoliko programer teži da atomične celine budu što kraće, to se zove programiranje na nivou fine granulacije (eng. *fine-grained level*). U suprotnom je programiranje na nivou grube granulacije (eng. *coarse-grained level*).

Jedna od najranijih tehnika za obezbeđivanje atomičnosti je tehnika zaključavanja (eng. *locking*). Nit, kada želi da izvrši atomičnu celinu, mora da zaključa bravu (eng. *lock*). Ako je brava zaključana od strane druge niti, nit mora da čeka dok se brava ne otključa. Kada nit završi atomičnu celinu treba da otključa bravu da bi druge niti koje žele da zaključaju bravu mogle da nastave svoje izvršavanje. Princip ove tehnike je pesimistički, jer se zaključavanje uvek radi bez obzira da li druge niti izvršavaju celine koje ne smeju da se preklapaju sa celinom za koju se brava zaključava. Zaključavanje je operacija koja se relativno dugo izvršava, jer prouzrokuje poništavanje brave u keš memoriji drugih procesora (vrši se upis) i neku vrstu blokiranja pristupa tom bloku u keš memoriji drugih procesora dok traje

upis. To iziskuje međuprocesorsku komunikaciju koja je spora, a koja će dovesti do promašaja u keš memorijama podataka ostalih procesora kada treba da pristupe bravi. Povećanje frekvencije operacije zaključavanja i otključavanja može značajno da smanji performanse programa [28], [29]. Pošto se zaključavanje uvek radi, bez obzira da li je ono potrebno ili ne, brzina izvršavanja programa je manja nego što bi mogla biti.

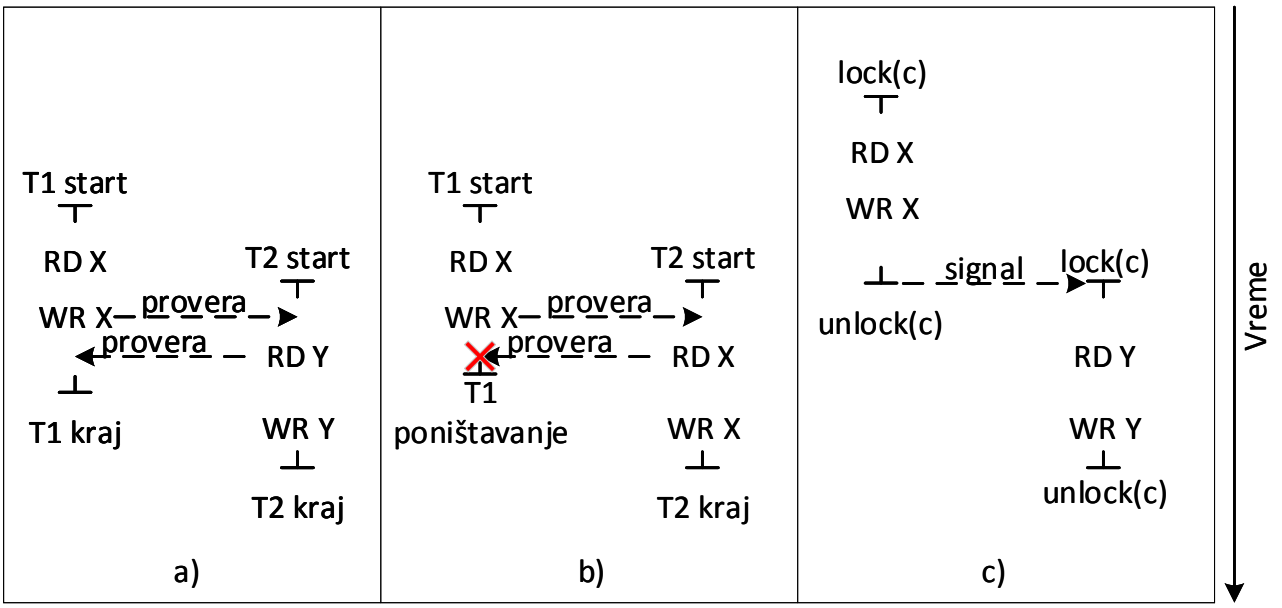
Ukoliko u sistemu postoji više brava, može doći do pojave mrtvog blokiranja (eng. *deadlock*), gde sve niti čekaju na otključavanje brave. U tom slučaju nema niti koja može otključati bravu i program nikad neće završiti sa radom i dati rezultat. Program koji može da dođe u stanje mrtvog blokiranja se smatra logički neispravnim programom. Navedeni problemi dodatno otežavaju programiranje višenitnih aplikacija. Pored zaklučavanja brave, ređe se koriste tehnike barijere (eng. *barrier*) i flegova (eng. *flag*). Sinhronizacija na barijeri blokira nit sve dok sve ostale niti ne dođu do barijere. Na taj način se obezbeđuje da tek kad sve niti završe neki deo, niti mogu da nastave izvršavanje. Sinhronizacija na flegu omogućava da se niti blokiraju sve dok fleg ne dobije neku određenu vrednost. Postavljanjem odgovarajuće vrednosti flega, jedna nit obaveštava sve ostale da se neki događaj desio i da mogu da nastave svoje izvršavanje. Postoje i druge tehnike višeg nivoa koje koriste već opisane tehnike (npr. semafori, monitori, itd.).

Nedostaci tehnike zaključavanja se pokušavaju rešiti tehnikom transakcione memorije (eng. *transactional memory*), prvi put opisanom u [3]. Tehnika se bazira na konceptu transakcija koje su bile već poznate iz oblasti baza podataka [30]. Kod ove tehnike se koristi optimistički pristup. Atomičnoj celini se bezuslovno dozvoli izvršavanje i ako ni jedna druga nit u toku izvršavanja te atomične celine ne pristupi istim podacima izvršavanje je uspešno i logički ispravno. Situacija u kojoj druga nit pristupi istim podacima naziva se konflikt (eng. *conflict*) i dovodi do logički neispravnog rezultata. Postojanje konflikta čini izvršavanje atomične celine neuspešnim.

Problem neuspešnog izvršavanja se rešava tako što se nit vrati u stanje u kojem je bila pre početka atomične celine i ponovo se pokuša izvršavanje atomične celine ispočetka. Time se postiže efekat kao da se transakcija nije ni počela izvršavati. Prema [31] programerima je lakše da rezonuju sa pojmom transakcije i samim tim lakše struktuiraju aplikacije, nego sa tehnikama zaključavanja. Tehnika može biti implementirana u hardveru, softveru ili mešovito. U ovom radu, razmatra se hardverska implementacija ove tehnike, bilo samostalno ili u sinergiji sa softverskom tehnikom. U slučaju hardverske implementacije, programer treba samo da naznači koje celine treba atomično da se izvrše, dok samu logičku ispravnost obezbeđuje hardver. Atomična celina se naziva transakcija (eng. *transction*). Uspešno izvršavanje transakcije se završava potvrđivanjem transakcije (eng. *transaction commit*). Neuspešno izvršavanje transakcije se naziva poništavanje transakcije (eng. *transaction abort*). Posle poništavanja transakcije, transakcija može da se ponovo pokrene (eng. *transaction restart*).

Na slici 2.2 su prikazani primeri izvršavanja kritične sekcije pomoću transakcija i zaključavanja brava kroz vreme. Prikazana kritična sekcija služi za upisivanje jednog podatka u neku strukturu podataka. Upisivanje se radi tako što se pristupi nekom elementu i u njega se izvrši upis adrese podatka koji se ubacuje. Slika 2.2a) prikazuje upisivanje dva različita podatka u paraleli na dva različita mesta. Pošto ne postoji konflikt, jer se pristupa različitim podacima, transakcije se potvrđuju. Prilikom svakog pristupa nekom podatku drugoj transakciji se šalje poruka (na slici označeno sa provera) o pristupu da bi mogao da se detektuje konflikt. U slučaju na slici 2.2b) dva različita podatka se ubacuju na isto mesto, što izaziva konflikt među transakcijama. Transakcija T1, koja je primalac, se poništava, dok se transakcija T2 potvrđuje. Promene ne mogu da se urade paralelno, jer se rade nad istim podatkom.

Izvršavanje kritične sekcije na slici 2.2c) se radi pomoću zaključavanja brave, pa kritične sekcije iz različitih niti ne mogu da se izvršavaju paralelno, bez obzira što se pristupa različitim podacima u nitima. Prilikom otključavanja, na slici je sa signal označeno da kritična sekcija u drugoj niti može da zaključa bravu. Moguće je postići isti efekat kao na slici 2.2a), ali to bi zahtevalo od programera da za svako mesto gde može da se ubaci podatak napravi posebnu bravu. To značajno komplikuje kod i povećava potreban prostor za smeštanje podataka. Takav pristup se ne preporučuje.



Slika 2.2: *Primer izvršavanja transakcija i kritične sekcije*

U otvorenoj literaturi postoji relativno veliki broj načina za implementaciju tehnike transakcione memorije. Transakcija može da bude atomična samo u odnosu na druge transakcije i to se naziva slaba atomičnost (eng. *weak atomicity*) ili transakcija može da bude atomična i u odnosu na kod van transakcija i to se naziva jaka atomičnost (eng. *strong atomicity*) [32]. Ono što važi za implementaciju transakcione memorije u hardveru je da postoje izvesna hardverska ograničenja, npr. dužina transakcija je hardverski ograničena, u transakciji ne sme da se nađe operacija čiji efekti izvršavanje ne mogu da se poništе (npr. upis u fajl), itd. Ukoliko se tokom izvršavanja transakcije dostigne hardversko ograničenje, transakcija se poništava. Razlikuju se četiri razloga za poništavanje transakcije: 1) poništavanje na zahtev programera uz pomoć posebne instrukcije za poništenje, 2) poništavanje usled konflikta (eng. *conflict abort*), 3) poništavanje usled prekoračenja hardverskih ograničenja (eng. *overflow abort*) i 4) poništavanje usled operacije čije efekti ne mogu da se poništе.

Ponovnim pokretanjem transakcije u slučajevima 1), 2) i 3), transakcija može da se uspešno izvrši pod određenim uslovima.

Za slučaj 1) može biti uspeha ako se došlo u deo koda gde ponavljanje može imati uspeha. Zato instrukcija za poništavanje transakcije ima jedan celobrojni parametar koji programer postavlja i na osnovu koga se odlučuje da li može transakcija uspešno da se izvrši prilikom ponovnog pokretanja.

Za slučaj 2) ukoliko postoji konflikt između dve transakcije i implementacija dozvoljava da se poništi bilo koja transakcija, može se doći u stanje živog blokiranja (eng. *live lock*). U tom stanju jedna transakcija se poništava, ponovo se pokrene i ponovo dođe do konflikta koji će poništiti drugu transakciju i tako ciklično. Ni jedna nit neće moći da napreduje i program se nikad neće završiti. Živo blokiranje se takođe smatra logički neispravnim izvršavanjem i tada nema smisla ponovo pokretati transakcije.

U slučaju 3) ponovno pokretanje treba raditi samo u slučaju kad je do prekoračenja došlo zbog nesrećnog spleta okolnosti koji ne mora da se ponovi. Recimo adrese u keš memoriji su se tako preklopile da je odabran podatak za izbacivanje kome je transakcija pristupila. Precizniji razlog u okviru svakog slučaja obično nije dostupan u hardverskoj transakcionoj memoriji. Ponovno pokretanje se radi nekoliko puta, jer sa svakim neuspešnim izvršavanjem veća je verovatnoća da se transakcija ne

može uspešno izvršiti. Broj pokušaja treba da bude određen tako da ponavljanje ne dovodi do usporavanja izvršavanja programa.

Ukoliko transakcija ne može da se izvrši uspešno i nakon nekoliko ponavljanja, programer mora da obezbedi atomičnost celine na drugi način bez upotrebe transakcije, da bi se obezbedilo nesmetano izvršavanje programa. Obično se to radi tehnikom zaključavanja. Zaključavanjem se izvršava kod iz transakcije, ali bez pokretanja transakcije, pa implementacija mora da obezbedi jaku atomičnost transakcija. U slučaju kada konflikti među transakcijama nisu česti, broj zaključavanja zbog sprečavanja žive blokade je redak.

Transakcije se mogu spajati [33]. Time može značajno da se olakša programiranje. Recimo postoji neka struktura podataka za koju su napravljene dve operacije koje su atomične. Ukoliko se sinhronizacija radi pomoću zaključavanja brava i ukoliko želimo da atomično uradimo te dve operacije jednu za drugom, mora da se obezbedi pristup unutrašnjim bravama te strukture, što iziskuje dodatan kod i sa stanovišta dobre softverske prakse nije poželjno. Ako se koriste transakcije, dovoljno je napraviti jednu transakciju u kojoj se te dve operacije izvršavaju.

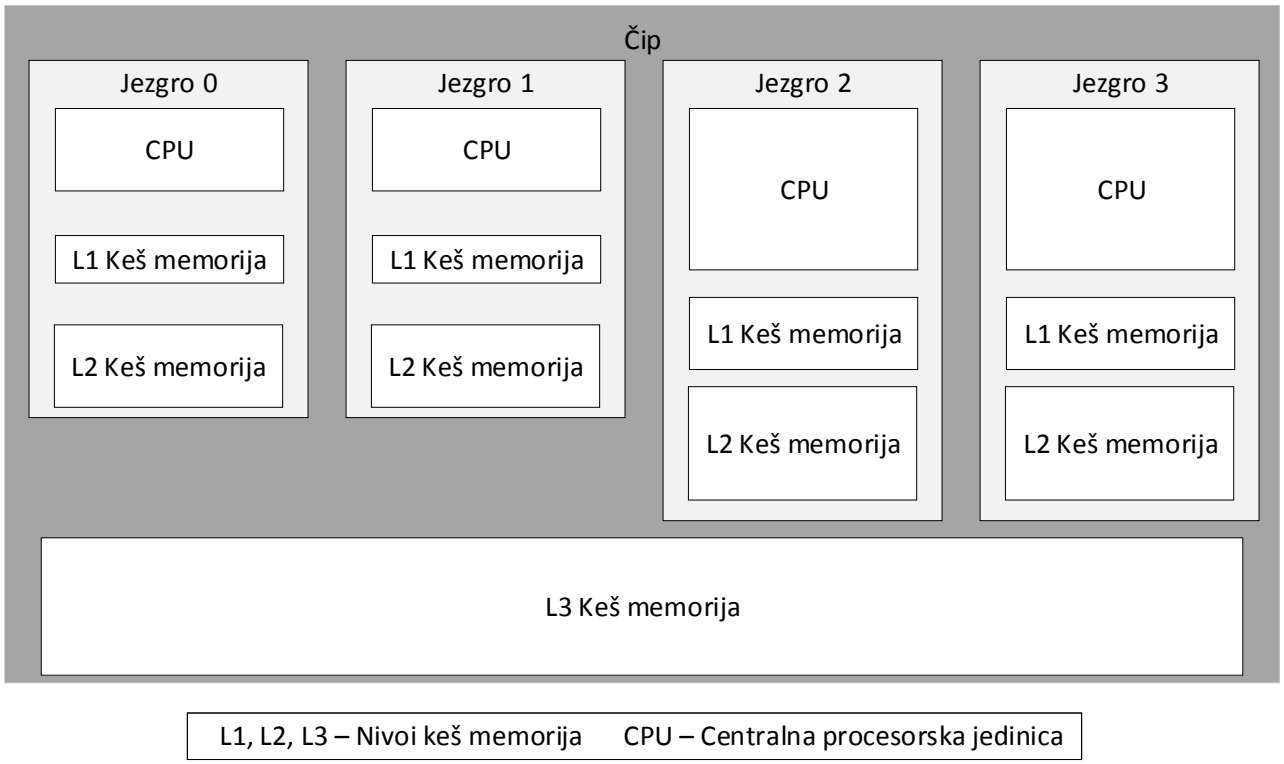
Iako, na prvi pogled deluje da tehnika transakcione memorije otklanja probleme oko pojave mrtvog blokiranja, to nije tako. U [32] su navedeni slučajevi kada se to može desiti. Problemi se javljaju u slučaju pooštravanja domena atomičnosti. Ako se razmatra program u kome je atomičnost obezbeđena pomoću zaključavanja brave, neke celine mogu da se preklapaju, jer nisu zaštićene istom bravom. Kod pokušaja obezbeđivanja atomičnosti pomoću transakcija, ni jedna atomična celina ne može da se preklopi sa drugom atomičnom celinom, što je strožije u odnosu na zaključavanje sa bravom. Program koji je inicijalno bio napisan tako da se osigura da se prvo izvrši jedna celina pa onda druga (recimo uposlenim čekanjem na nekom flegu), nije garantovano da će se program sa transakcijama završiti. To znači da jedna transakcija mora da upiše dozvolu za izvršavanje koju druga transakcija treba da pročita. Kod programa sa zaključavanjem brave je to bilo moguće, jer su celine bile zaštićene različitim bravama, kod programa sa transakcijama ne može, jer transakcije moraju da budu atomične jedna u odnosu na drugu. Da bi se ovakvi problemi prevazišli, programer treba da izbegava situacije u kojima jedna transakcija čeka da se druga završi. Takve situacije se mogu lakše rešiti korišćenjem barijera.

2.3 Razlozi za uvođenje asimetrije u procesore

Mnoštvo aplikacija koje se danas koriste dovelo je do ideje da se multiprocesori učine asimetričnim [4]. Asimetričnost znači da nisu svi procesni elementi (eng. *processing element*) isti. Pošto postoji raznolikost u procesnim elementima, različite aplikacije ili delovi aplikacija mogu da se izvršavaju na onim elementima gde će postići najbolje rezultate. U literaturi postoji više načina da se procesni elementi učine različitim. Procesni elementi mogu biti slični, a mogu biti i u potpunosti različiti. Recimo, multiprocesor može da se sastoji od jezgara opšte namene i procesnih elemenata za obradu slike (eng. *graphics processing unit*). Takva asimetrija prevazilazi okvire ove disertacije i na dalje nije razmatrana. Razmatraju se samo multiprocesori koji se sastoje od različitih jezgara opšte namene. Ti multiprocesori se nazivaju asimetrični višejezgarni procesori.

Asimetrični višejezgarni procesori imaju jezgra koja se razlikuju na mikroarhitekturalnom nivou. Razlike mogu biti u načinu na koji se izvršavaju instrukcije (u programskom poretku ili ne), u broju instrukcija koje se mogu izvršiti istovremeno, u broju instrukcija koji se može istovremeno pokrenuti, u dubini protočne obrade, u veličini i organizaciji memorijskih elemenata (keš memorija, upisnih bafera, ...), u skupu instrukcija koje podržavaju, u broju niti koje podržavaju, itd. Jezgra mogu da se razlikuju po tome na kojoj frekvenciji rade i da li ta frekvencija može da se podešava. U ovoj disertaciji biće razmatrani oni procesori koji imaju identičan instrukcijski skup, a sve ostalo mogu da imaju različito. Oni se nazivaju asimetrični višejezgarni procesori sa jednim instrukcijskim setom (eng. *Single-ISA asymmetric multicore processors*). Skraćeno ćemo ih nazivati asimetrični višejezgarni procesori.

Jedan primer takvog procesora prikazan je na slici 2.3.



Slika 2.3: *Asimetrični višejezgarni procesor sa dva velika i dva mala jezgra*

Asimetrični višejezgarni procesori nastali su modifikacijom simetričnih višejezgarnih procesora. Jezgra u simetričnim procesorima se prave da budu što boljih performansi, što znači da zauzimaju više prostora na čipu i troše više energije. Ta jezgra se nazivaju velika jezgra. Neka od tih jezgara menjaju se slabijim jezgrima, sa stanovišta performansi. Ta jezgra zauzimaju manje mesta na čipu i nazivaju se mala jezgra. Pošto zauzimaju manje mesta može više da ih se doda. Sa stanovišta performansi, korišćenje asimetričnih umesto simetričnih procesora nije opravdan. Prema Amdalovom zakonu, ubrzanje koje može da se dobije za neku paralelnu aplikaciju limitirano je serijalizovanim izvršavanjem. Serijalizovano izvršavanje je izvršavanje gde paralelizam mora da se smanji. Recimo izvršavanje kritične sekcije je serijalizovano izvršavanje [34], jer nijedna druga nit ne sme da izvršava tu kritičnu sekciju. Pošto se serijalizovano izvršavanje usporava izvršavanjem na malom jezgru, prema Amdalovom zakonu cela aplikacija ima manji teorijski limit za ubrzanje [35].

Međutim, sa stanovišta potrošnje energije uvođenje asimetrije ima smisla. Ako neko izvršavanje ograničeno memorijskim pristupima (eng. *memory bound*), malo jezgro će potrošiti manje energije nego veliko, a postići će sličnu brzinu izvršavanja. Asimetrija pruža mogućnosti da se istovremeno optimizuje i brzina izvršavanja i potrošnje energije. Takođe, neke aplikacije ili delovi aplikacije nemaju potrebe za najbržim izvršavanjem, jer se njihovi rezultati ne koriste odmah (recimo aplikacije za logovanje događaja). U tom slučaju asimetrija pruža mogućnosti da se optimizuje samo potrošnja energije. Optimizacija potrošnje energije je važan faktor u dizajnu procesora koji se koriste u mobilnim uređajima. Iz tog razloga najveći broj komercijalno dostupnih asimetričnih višejezgarnih procesora je napravljen za mobilne uređaje. Smanjenje potrošnje energije je postalo važno i u sistemima visokih performansi. Sa svakom novom generacijom procesora više jezgara može da stane na čip zbog više tranzistora koji mogu da se naprave na čipu. Međutim, porast energetske efikasnosti tranzistora ne može da prati brzinu porasta broja tranzistora, pa jezgra moraju da budu slabija ili ne mogu da budu sva aktivna u isto vreme [36].

Drugi razlog za uvođenje asimetrije je taj što pomoću asimetričnog procesora može da poveća propusnost sistema. To povećanje propusnosti obično se dobija na uštrb performansi izvršavanja po-

jedinačnih aplikacija [37]. Asimetrija pruža bolje mogućnosti za istovremeno optimizovanje i brzine izvršavanja pojedinačne aplikacije i propusnosti. Za to je potrebno da procesor ima više malih jezgara i bar jedno veće jezgro. Takva mogućnost je potrebna i za procesore koji se koriste u serverskim sistemima, jer je potrebno povećati propusnost kada je broj korisnika veći, a poboljšati performanse izvršavanja kada je broj korisnika manji.

Asimetrični višejezgarni procesori mogu da budu statički ili dinamički asimetrični. Dinamička asimetrija se postiže promenom frekvencije ili reorganizacijom jezgara tokom rada uređaja. Jedna ideja je da se više malih jezgara spoji u veće jezgro [38]. Iako takva ideja deluje primamljivo, teško je implementirati takav procesor [39]. Da bi se postigle performanse izvršavanja velikog jezgra potrebno je ili spojiti mnogo malih jezgara ili dodati logiku velikog jezgra za kontrolu toka instrukcija pored malih jezgara. Prvi način je nemoguće napraviti na čipu, dok drugi način zahteva logiku čija kompleksnost se može naći u velikom jezgru.

Najbolja konfiguracija asimetričnog višejezgarnog procesora je da postoje dve vrste jezgra, veliko i malo [37]. Veliko treba da bude superskalarno (da izvršava istrukcije van programskog poretka). Da bi mogle da se iskoriste sve prednosti asimetrije potrebno je da deljena keš memorija bude što veća, da se broj velikih i malih jezgara podesi prema potrebama aplikacije i da se koristi raspoređivač. Raspoređivač treba da mapira izvršavanja na onu vrstu jezgra gde će se najbolje optimizovati potrebni parametri (propusnost, performanse ili potrošnja energije). Takvi su i postojeći asimetrični višejezgarni procesori [40]–[44].

Poglavlje 3

Postojeća rešenja

U ovom poglavlju dat je detaljan pregled otvorene literature. Pregled uključuje najznačajnije radove iz oblasti transakcione memorije i asimetričnih višejezgarnih sistema. Takođe, pregled uključuje i radove o podsistemu keš memorija, koji su u bliskoj vezi sa prethodno navedenim oblastima i predloženim rešenjem u ovoj disertaciji.

3.1 Postojeće tehnike za poboljšanje transakcione memorije

U otvorenoj naučnoj literaturi postoji veliki broj radova koji se bave tehnikom transakcione memorije. Postojeća rešenja se mogu svrstati u kategorije koje se tiču dizajnerskih odluka u samoj implementaciji arhitekture koja podržava transakcionu memoriju, tehnike za prevazilaženje ograničenja, algoritama za poboljšanje performansi, interfejsa između hardvera i softvera, modelovanja performansi rešenja, algoritama za optimizaciju radi smanjenja potrošnje energije, itd. U nastavku su opisana neka od najznačajnijih rešenja.

3.1.1 Odluke prilikom dizajna transakcione memorije

Arhitektura koja realizuje tehniku transakcione memorije mora da sadrži rešenja sledećih problema [45] [46]: 1) verzionisanje podataka, 2) vođenje evidencije o radnom skupu transakcije i detekcije konflikata i 3) poništavanje transakcija za koje je detektovan konflikt. Radovi koji opisuju odluke prilikom implementacije i vrše evaluaciju tih odluka dati su u tabeli 3.1.

Tabela 3.1: *Radovi koji opisuju odluke u dizajnu*

Verzionisanje podataka	Detekcija konflikata	Rešavanje konflikata	Radovi
Evidencija promena	Gramzivo	Primalac	
		Algoritam	[47] (teorijsko) [48] [49] [50] [51]
Baferisanje promena	Lenjo	Primalac	[52] [53] [54] [55] [56] [57]
		Algoritam	[58]
	Gramzivo	Primalac	[3] [59] [60] [61] [62]
		Algoritam	[63] [47] (uprošćeno) [64]

Problem verzionisanja podataka se javlja, jer postoji potreba da se efekat transakcije poništi u slučaju konflikta. Efekti transakcije su izvršeni upisi u memorijske lokacije za vreme izvršavanja transakcije. Rešenje mora da obezbedi da konačan rezultat u slučaju konflikta bude kao da se ti upisi nisu ni desili. To se može postići na sledeće načine: 1) vođenjem evidencije šta je pre početka transakcije bilo u lokacijama u kojima je izvršen upis u toku transakcije, upis se vrši normalno, dok

se prilikom poništavanja transakcije vrši povraćaj zapamćenih podataka i 2) baferisanjem upisa tako što se podaci ne bi upisivali u memoriju već na neku privremenu lokaciju, a nakon potvrđivanja transakcije ti se podaci prebacuju u memoriju.

Odluka koja od metoda će biti korišćena nije jednostavna, jer oba načina imaju svoje prednosti i mane. Prva metoda vrši povraćaj podataka ako je transakcija poništena i ako se to nalazi na kritičnom putu može doći do usporenje programa. Druga metoda zahteva potvrđivanje baferisanih podataka prilikom potvrđivanja uspešne transakcije. Opet, ako se ta operacija nalazi na kritičnom putu može doći do usporenja. Obe metode mogu čuvati informacije samo o ograničenom broju promena i ona metoda koja više promena može da sačuva je bolja, jer u tom slučaju transakcija može više podataka da promeni. Broj promena je nepromenljiva granica, što znači da u slučaju potrebe za većim brojem promena transakcija neće moći da se izvrši i na neki drugi način se mora obezbediti atomičnost sekcije.

Drugi problem koji odluke u dizajnu transakcione memorije treba da reše jeste problem detekcije konflikata. Detekcija konflikata se rade proverom radnog skupa jedne transakcije sa radnim skupom druge transakcije. Radni skup je skup svih lokacija kojima je transakcija pristupala bilo zbog čitanja bilo zbog upisa. Konflikt postoji ukoliko je neka transakcija upisala podatak u neku lokaciju koja se nalazi u radnom skupu neke druge transakcije. Pošto nije garantovano kojim redosledom su ti pristupi izvršeni u dvema transakcijama, atomičnost nije garantovana i konflikt mora da se razreši. Provera konflikta može da se radi prilikom svakog pristupa, tzv. gramziva provera (eng. *greedy*), ili ređe, u nekim predefinisanim trenucima (npr. prilikom potvrđivanja transakcije), tzv. lenja provera (eng. *lazy*). Prva metoda zahteva česte provere koji mogu usporiti izvršavanje, dok sprečava nastavak izvršenja transakcije od trenutka kada se konflikt dogodi. Druga metoda radi suprotno, provere su retke, ali transakcije nastavljaju da se izvršavaju iako će njihov efekat biti poništen. Odluka koja metoda da se primeni mora biti bazirana na prirodi transakcija koje će biti izvršavane, npr. ako su konflikti retki i dešavaju se pri kraju transakcije, druga metoda je pogodnija.

Pored samog trenutka provere bitno je i na kom nivou granularnosti se vrši provera. Provera može da se vrši na nivou adresibilne jedinice, na nivou bloka keš memorije, na nivou stranice virtuelne memorije, itd. Što je nivo granularnosti viši potrebno je voditi veću evidenciju o radnim skupovima, dok u slučaju manje granularnosti evidencija je manja, ali postoji veća verovatnoća za pojavu lažnog deljenja (eng. *false sharing*). Lažno deljenje je pojava kada se podatak nalazi u radnom skupu, iako mu nije pristupano (npr. granularnost je na nivou bloka keš memorije, tada kad se pristupi jednom podatku svi podaci koji se nalaze u istom bloku keš memorije se stavljaju u radni skup). Takav podatak može prouzrokovati konflikt koji je nepostojeći. Pojava nepostojećih konflikata može usporiti sistem, jer će se transakcije poništiti bez razloga.

Treći problem koje odluke u dizajnu transakcione memorije treba da reše je kako razrešiti konflikt. Kada se konflikt desi neka od transakcija koja učestvuje u konfliktu treba da se poništi. Detekcija konflikta zahteva da jedno jezgro pošalje drugom informaciju kojim podacima je pristupilo za vreme izvršavanje transakcije. Primalac je taj koji može da detektuje konflikt. Kada su dve transakcije u konfliktu sa stanovišta logičke ispravnosti programa svejedno je koja transakcija će biti poništena. Odabir može uticati na performanse programa, jer se može poništiti transakcija koja se nalazi na kritičnom putu. Takođe, način odabira može uticati na kompleksnost hardvera. Najlakše je implementirati poništavanje one transakcije koja je primalac. Ako primalac detektuje konflikt i poništi se, nikakva druga komunikacija nije potrebna, jer pošiljalac može da se nastavi i ne mora ni da se obavesti o konfliktu. Drugi metod je da se primeni neki algoritam za rešavanje konflikta, pri čemu pošiljalac i primalac moraju razmeniti više poruka i odlučiti koja transakcija je bolja za poništavanje. U slučaju kada u konfliktu učestvuje jedna transakcija i izvršavanje van transakcije, jedino transakcija može da se poništi i nema potrebe za bilo kakvim odlučivanjem.

Prvo rešenje transakcione memorije [3] baferiše podatke pomoću dodatne keš memorije. Ta keš memorija je privatna za dati procesor. Procesor je koristi kada se izvršavaju transakcije. Svi upisi koji se izvrše u toku transakcije se upisuju u nju. Ta keš memorija je povezana na magistralu i učestvuje

u protokolu koherencije keš memorije. Protokol je proširen da poznaje čitanja i upise koji se vrše u transakcijama. Pošto protokol koherencije keš memorije javlja svim ostalim procesorima o izvršenom čitanju ili upisu, taj mehanizam se koristi za detekciju konflikta. Konflikt se detektuje kada se primi informacija o upisu u nekoj drugoj transakciji koji je u konfliktu sa radnim skupom date transakcije i ona se poništava. Prilikom poništavanja sadržaj dodatne keš memorije se poništi, dok prilikom potvrđivanja transakcije sadržaj dodatne keš memorije se prepiše u memoriju i očisti se keš memorija od tih podataka. Autori su pokazali da ovakvo rešenje može dati bolje performanse u odnosu na rešenja sa zaključavanjem.

U cilju olakšanja hardversko softverskog interfejsa i izvršavanja već napisanih programa autori [59] su predložili rešenje koje detektuje ulazak u kritičnu sekciju i izbegava zaključavanje brave. Kritična sekcija kod koje se izbegne zaključavanje brave se izvršava spekulativno, slično kao transakcija, i može biti poništena ako dođe do konflikta sa drugim nitima u pristupu podacima. U slučaju da se kritična sekcija nekoliko puta poništi, pristupa se zaključavanju brave čime se garantuje napredak u izvršavanju. Sve spekulativne promene se čuvaju u baferu za upis podataka u memoriju (eng. *store buffer*), dok se informacija o spekulativno pročitanim podacima čuva u keš memoriji. Prilikom poništavanja samo se ponište upisi u baferu i informacije u keš memoriji, jer to nisu podaci vidljivi ostatku sistema. Konflikt se detektuje protokolom keš koherencije i primalac se poništava. Pošto ne postoje posebne instrukcije za izbegavanje zaključavanja, već se zaključavanje brave implicitno detektuje, samo zaključavanje mora da se uradi na standardizovan način. Autori su pokazali da se veliki broj zaključvanja kritičnih sekcija izbegava u SPLASH kolekciji test programa [65] [66], što doprinosi povećanju paralelizma koji može doprineti pobolšanju performansi programa. Softverske tehnike koje koriste istu ideju postoje [67] [68], ali hardverska implementacija je efikasnija.

Nadogradnja prethodne tehnike urađena je u [63]. Dodat je algoritam za rešavanje konflikata, baziran na vremenskim markama. Algoritam sprečava pojavu živog blokiranja, pa nije potrebno vršiti sinhronizaciju pomoću zaključavanja brave radi sprečavanja živog blokiranja. Transakcija koja se poništava je mlađa transakcija, gde se starost određuje prema vrednosti vremenske marke. Transakcija dobija vremensku marku prvi put kad se pokuša njeno izvršavanje i ne menja se prilikom ponovnog pokretanja transakcije. Ukoliko pošiljalac treba da se poništi, primalac samo ne odgovara na poruke i nastavlja dalji rad. Kada taj primalac izvrši neki upis ili potvrdi transakciju, odgovarajući pošiljalac će poništiti svoju transakciju. U zavisnosti od algoritma za keš koherenciju može da se desi da ovakav scenario nije moguće ostvariti, jer pošiljalac neće čekati na odgovor. U tom slučaju koristi se prediktor instrukcija. Kada se predvidi da će neka instrukcija izvršiti upis, ta informacija se pošalje svima ostalima pre nego što do upisa stvarno dođe. Na taj način se smanjuje verovatnoća da se pojavi gore opisani slučaj. Ukoliko, ipak dođe do tog slučaja, poništiti će se ono što se mora, a to je primalac, čak iako je stariji po vremenskoj marki. U šlučaju da transakcija prekorači hardverska ograničenja, pristupa se sinhronizaciji pomoću zaključavanja brave.

Autori u [60] takođe predlažu izvršavanje kritične sekcije čak iako je brava zaključana. Razlika u odnosu na predhodna rešenja je ta što se pušta da prva nit koja izvršava kritičnu sekciju zaključa bravu. Time se postiže da postoji jedna bezbedna nit koja će sigurno uspešno završiti kritičnu sekciju i time se izbegava živo blokiranje. Sve ostale niti izvršavaju kritičnu sekciju spekulativno. Ukoliko dođe do konflikta, primalac se poništava. Promene u spekulativnim nitima se baferišu u keš memoriji i nisu dostupne drugim nitima. Onog trenutka kada bezbedna nit napusti kritičnu sekciju, sve spekulativne niti koje su završile kritičnu sekciju, a nisu imale konflikte, postaju nespekulativne i njihov rad atomično postaje vidljiv ostalim nitima. Spekulativne niti koje nisu završile kritičnu sekciju pokušavaju da zaključaju bravu i time postanu bezbedna nit. Sve one mogu da nastave dalje izvršavanje bez potrebe za nekim zaustavljanjem. Ovakim načinom implementacije transakcija, spekulacija može da se nastavi posle kraja kritične sekcije. U slučaju poništavanja takve transakcije, veća je količina rada koji se poništava. Pošto je spekulacija duža, lakše može da se dostigne hardversko ograničenje dužine transakcije. Pored spekulativnog izvršavanja koda posle zaključavanja brave, autori predlažu i spekulativno izvršavanje preko barijere i preko provere flega, gde su bezbedne niti one koje nisu još

stigle do barijere i ona koja postavlja fleg, respektivno.

Tehnika prezentovana u [52] zahteva da ceo program bude podeljen u transakcije. Tokom izvršavanja jedne transakcije sve promene se baferišu i promene se ne javljaju ostatku sistema. Kada se transakcija završi, sve promene se šalju atomično ostatku sistema. Atomičnost se obezbeđuje tako što se proglasi da transakcija počinje potvrđivanje i nijedna druga transakcija ne može da pristupi potvrđivanju. U radu nije opisano kakve se poruke razmenjuju između jezgara da bi se postigao dogovor koja transakcija može da pristupi potvrđivanju. Programer može navesti redosled kojim neke transakcije treba da se izvrše. U tom slučaju transakcija može da pristupi potvrđivanju samo ako su sve transakcije koje treba da se završe pre nje stvarno i završile. Prilikom potvrđivanja podaci se šalju svim ostalim jezgrima. Ukoliko jezgro koje primi promenu detektuje da je transakcija koju izvršava u konfliktu sa transakcijom koja se potvrđuje, poništava svoju transakciju. Time se konflikti detektuju tek na kraju jedne transakcije, a ne prilikom svakog pristupa podacima i uvek se poništava primalac. U slučaju da se stigne do hardverskog ograničenja dužine transakcija, izvršavanje se pauzira i traži se dozvola za potvrđivanje. Kada se dozvola dobije, nastavlja se nespekulativno izvršavanje do kraja transakcije. Dok ta transakcija ima dozvolu za potvrđivanje ni jedna druga transakcije ne može da se potvrdi. Ista operacija se primenjuje i kad se naiđe na istrukcije koje nije moguće poništiti. Poželjno je da se ova operacija primenjuje u retkim slučajevima, jer može dovesti do sekvencijalizacije izvršavanja. Ovakva tehnika izbegava potrebe za postojanje hardvera koji obezbeđuje memorijsku koherenciju i konzistenciju, što značajno smanjuje kompleksnost hardvera za međuprocesorsku komunikaciju. Memorijska koherencija je očuvana, jer upisi nisu vidljivi sve do faze potvrđivanja, kada se rade atomično. Memorijska konzistencija je očuvana, jer je definisan redosled vidljivosti upisa samim redosledom potvrđivanja transakcija.

Autori u [58] su primetili da može da se iskoristi činjenica da redosled izvršavanja transakcija u različitim nitima može da bude proizvoljan i da se time ne narušava sekvencijalna konzistencija. U toku izvršavanja jedne transakcije njene promene se baferišu i nisu dostupne ostalim transakcijama. Iz toga proizilazi da kada sve transakcije dođu do završetka, može da se odredi redosled potvrđivanja transakcija koji će smanjiti broj zavisnosti po podacima između transakcija i koji će poništiti samo one transakcije koje su narušile zavisnost po podacima. Tokom izvršavanja transakcije se detektuje konflikt između transakcija i kreira se graf zavisnosti. Ukoliko neka transakcija pročita podatak koji druga transakcija promeni, prva zavisi od druge (dodaje se usmerena grana u grafu od druge do prve transakcije). Ukoliko je graf acikličan postoji redosled potvrđivanja transakcija koji ne narušava zavisnost po podacima. Prvo treba da se potvrdi ona transakcija od koje niko ne zavisi (ni jedna grana u grafu nije usmerena od te transakcije ka nekoj drugoj transakciji). Ukoliko postoji ciklus u grafu, jedna od transakcija mora da se poništi da bi ciklus nestao. Kada se poništi dovoljno transakcija da svi ciklusi nestanu, transakcije koje su ostale u acikličnom grafu mogu u korektnom redosledu da se potvrde. Sve transakcije moraju da se završe, jer je tek onda graf kompletan. Niti mogu da nastave svoje izvršavanje preko granice transakcije, ne moraju da čekaju ostale transakcije da se završe, ali to izvršavanje je spekulativno i može da se nastavi do početka sledeće transakcije, kada se nit pauzira. U radu nije precizirano kako se tačno određuje koji je skup transakcija koji se trenutno izvršava, da bi se znalo na koje transakcije se čeka i nije precizirano šta se dešava ako neka nit počne da izvršava transakciju dok algoritam odlučuje o redosledu transakcija za potvrđivanje.

Autori u [48] uočavaju da je kod nekih aplikacija procenat poništenih transakcija mali i predlažu nekoliko rešenja koja koriste tu činjenicu. Osnovna ideja je da se podaci koji se promene tokom transakcije upišu na za to određeno mesto u virtuelnoj memoriji procesa, a da se spekulativne vrednosti podataka propagiraju kroz celu memorijsku hijerarhiju. U [49] autori detaljno opisuju i obrađuju jednu takvu implementaciju. Nespekulativne vrednosti podataka se pre promene pamte u arhivi koja se nalazi u virtuelnoj memoriji procesa na lokaciji na koju ukazuje jedan registar (operativni sistem mora da inicijalizuje ovaj registar u svakom procesoru na različitu vrednost). Autori tvrde da ovaj upis ne utiče značajno na performanse, jer se upis vrši normalno u keš memoriju i prostor je lokalan za svaku nit. U slučaju da upis treba ubrzati može se dodati i bafer iz koga bi se podaci upisivali u keš memoriju

kad je to moguće bez bilo kakvog usporavanja. Podaci se pamte na nivou jednog bloka u keš memoriji. Pamti se adresa bloka i sami podaci. Podaci se smeštaju u niz, jedan za drugim, redom kojim su pristigli zahtevi za upis. Rešenje predviđa korišćenje protokola keš koherencije zasnovanog na katalogu. Dodata su nova stanja u kojim može da se nađe jedan blok. Kada se izbaci neki blok koji je spekulativno pročitan ili u koji je spekulativno izvršen upis, to se evidentira posebnim stanjem u katalogu. Kada neki drugi procesor pristupi tom bloku, katalog prosleđuje tu informaciju svim procesorima koji su spekulativno izbacili taj blok iz keš memorije. Na taj način procesori koji su izbacili taj blok mogu da detektuju konflikt. Rešavanje konflikta se radi pomoću algoritma zasnovanog na principu vremenskih marki. Prilikom uspešnog izvršavanja transakcije poništava se arhiva i započinje se lenja promena stanja blokova koji su bili izbačeni iz keš memorije. Na taj način potvrđivanje transakcije se obavlja relativno brzo. U slučaju da je transakcija poništena poziva se softverska rutina koja restaurira podatke iz arhive, što može da traje poprilično dugo. Rezultati testiranja pokazuju da predloženo rešenje ubrzava izvršavanje aplikacija u odnosu na prethodne tehnike, ukoliko je broj abortiranih transakcija mali.

Dosad navedene tehnike nisu podržavale transakcije u simultanom višenitnom procesoru (eng. *simultaneous multithreading processor*). Takav procesor ima više hardverskih konteksta za niti gde se neki delovi procesora dele. Najznačajnija stvar za implementaciju transakcione memorije koja se deli je keš memorija. U [62] se predlaže jedno rešenje koje implementira transakcionu memoriju na simultanom višentnom procesoru. Predlaže se čuvanje podataka kojima se spekulativno pristupilo u posebnom transakcionom baferu. U njemu se čuva nespekulativna vrednost, spekulativna vrednost, vrsta pristupa i bit vektor koji sadrži informaciju koja hardverska nit je pristupila tom podatku. Transakcioni bafer je visoko asocijativan da bi se smanjila mogućnost konflikta. Autori tvrde da ovakav pristup brže izvršava operaciju potvrđivanja ili poništavanja, jer se u transakcionom baferu lako pamti koja hardverska nit je izvršila operaciju. Pošto se u transakcionom baferu čuva i nespekulativna vrednost, protokol keš koherencije ne treba da se menja, jer ne mora da se obezbedi da nespekulativna vrednost bude u onom nivou keš memorije koji je deljen između više procesora. Ovakva implementacija zahteva dodatan prostor i unosi dodatnu složenost u proizvodnju procesora u odnosu na rešenja koja čuvaju spekulativne podatke u keš memoriji. Dodavanjem transakcionog bafera praktično se povećava veličina prvog nivoa keš memorije i u radu nije ispitano koliko praktično duplirana veličina keš memorije prvog nivoa doprinosi ubrzanju aplikacija u odnosu na ostale tehnike.

Nekoliko radova predlaže korišćenje skupova za čuvanje radnog skupa transakcije. Svaki put kada se pristupi podatku, njegova adresa se ubacuje u odgovarajući radni skup (za čitanje ili za upis). Kada se primi informacija o upisu od strane drugog procesora, proverava se da li se adresa tog podatka nalazi u radnom skupu. Ako se nalazi, došlo je do konflikta. Time se izbegava ubacivanje dodatnih bita u keš memoriju za vođenje evidencije o radnom skupu i izbegavaju se problemi koji mogu nastati tim ubacivanjem. Da bi implementacija pomoću skupova bila opravdana, skupovi moraju podržavati efikasne operacija ubacivanja u skup i proveru da li se neki podatak nalazi u skupu. Takođe, implementacija ne sme biti hardverski kompleksna. Način za implementaciju koji ispunjava te uslove je prikazan u [69] i naziva se Blumov filtar (eng. *Bloom's Filter*). Blumov filtar je aproksimativni skup. Prilikom provere da li se podatak nalazi u Blumovom filtru, može da se dogodi da se podatak nalazi u skupu iako nije ubačen u skup. To se naziva lažni pogodak (eng. *false positive*). Lažni promašaj (eng. *false negative*) nije moguć. Suštinski Blumov filtar je nadskup onih podataka koji su u njega ubačeni. Lažni pogoci prouzrokuju samo lažne konflikte. Time se može samo uticati na performanse izvršavanje programa, ali ne i na ispravnost izvršavanja.

Autori u [53] tvrde da povećanje kompleksnosti protokola keš koherencije i prvog nivoa keš memorije može prouzrokovati probleme prilikom implementacije procesora i predlažu implementaciju transakcione memorije koja to izbegava. Osnovna ideja je da za vreme izvršavanja transakcije procesor beleži radni skup u hardverski implementiranom skupu. Koristi se jedan skup za adrese pročitanih podataka i jedan skup za adrese upisanih podataka. Adrese podataka kojima je pristupano se ne šalju ostalim procesorim. Za implementaciju skupova se koristi uprošćeni Blumov filtar. Kada neka trans-

akcija završi izvršavanje, ona šalje svoj skup adresa upisa svim ostalim procesorima, koji tada proveravaju da li je došlo do konflikta. Ako jeste, primalac se poništava. Provera se radi efikasnom operacijom preseka skupova, koja se lako implementira u hardveru. Pored provere da li je došlo do konflikta, kada neki procesor primi skup upisanih adresa, on poništava te adrese u svojoj keš memoriji, jer to nisu više validni podaci (validni su oni koje je proizvela transakcija koja se trenutno potvrđuje). Rezultati testiranja pokazuju da je prosečan procenat lažnih konflikata od svih konflikata oko 14%. Uračunati su samo lažni konflikti zbog načina implementacije skupova. I pored lažnih konflikata performanse izvršavanja aplikacija su neznatno lošije u odnosu na ranije implementacije.

Prethodno rešenje iskorišćeno je za obezbeđivanje sekvencijalne konzistencije u [56]. Ideja je da se celokupno izvršavanje programa podeli u transakcije. Instrukcije u transakcijama mogu da se izvršavaju u proizvoljnom poretku. Takođe i instrukcije iz različitih transakcije mogu da se izvršavaju u proizvoljnom poretku. Sekvencijalna konzistencija se obezbeđuje time što upisi u memorijske lokacije postaju vidljivi ostatku sistema tek u trenutku potvrđivanja transakcije. Postoji arbitrator koji određuje koja transakcija može da se potvrdi. U jednom trenutku može biti više transakcija koje se potvrđuju. Arbitrator ima hardverski implementiran skup u kome se nalaze sve adrese u koje su upisale transakcije koje se trenutno potvrđuju. Ako transakcija pokuša da se potvrdi, šalje svoj radni skup arbitratoru. Ako arbitrator ustanovi da nema konflikta, daje dozvolu transakciji da se potvrdi. Početak i kraj transakcije u jednoj niti određuje sam hardver. Hardver započinje transakciju kada se prethodna završi. Tokom izvršavanja transakcije hardver broji instrukcije. Prvo pokuša da izvrši transakciju određene dužine. Ako transakcija ne uspe, pokuša opet, ali pritom skrati dužinu transakcije. Transakcije kreirane na ovakav način služe samo za postizanje sekvencijalne konzistencije, ali ne i za sinhronizaciju među nitima, jer kraj transakcije može da se nađe u kritičnoj sekciji. Sinhronizacija među nitima, mora da se uradi na neki drugi način. Autori pokazuju da ovakvo rešenje ima slične performanse izvršavanja aplikacija kao rešenja koja realizuju relaksirane konzinstencije, ali olakšava posao programeru, jer obezbeđuje semantiku sekvencijalne konzinstencije.

Kombinacija rešenja sa čuvanjem arhive podataka koji su promenjeni i rešenja sa skupovima prikazana je u [50] sa ciljem da se u keš memoriju ne dodaju dodatni biti. Razlika je u odnosu na prethodno rešenje sa skupovima je što se konflikti detektuju gramzivo, a ne na kraju transakcije. Pošiljaocu se prilikom konflikta pošalje poruka da je došlo do konflikta i pošiljalac odlučuje kako da razreši konflikt (može da se poništi ili da sačeka i pokuša operaciju ponovo). Pošto nema tačnih informacija koji podaci su promenjeni (hardverski skupovi čuvaju samo nadskup svih adresa podataka koji su promenjeni), u arhivi se uvek moraju beležiti podaci koji se menjaju (čak iako se jedan podatak menja više puta u okviru jedne transakcije). Radi smanjenja redudantnih upisa, dodaje se jedan mali asocijativni niz koji čuva adrese nekoliko poslednje promenjenih podataka. Evaluacija predloženog rešenja pokazuje da je rešenje bolje od sinhronizacije pomoću zaključavanja brave.

Autori u [57] prikazuju implementaciju transakcione memorije na procesoru koji podržava mehanizam za višestruko pamćenje stanja (eng. *multi-checkpointing*). Procesor zapamti svoje stanje kada naiđe na odgovarajuću instrukciju (memorijski pristup podatku koji nije u keš memoriji, instrukciju skoka, itd.). Nastavlja spekulativno da izvršava instrukcije posle toga. Ako se dogodi neki izuzetak ili mispredikcija skoka, stanje procesora se vraća u stanje koje je prethodno zapamćeno. Na taj način se poništava pogrešan spekulativan rad. Višeprocesorski sistem je napravljen tako da se između dva pamćenja stanja ne propagiraju adrese na kojima je vršen upis. Prilikom potvrđivanja se pošalju sve adrese upisa drugim procesorima i ukoliko primalac detektuje konflikt, on poništava svoju transakciju. Dužinu transakcije određuje hardver, prema broju poništavanja. Ako poništavanja jedne transakcije postanu česta, transakcija se skraćuje. Sinhronizacija se vrši pomoću zaključavanja brava. Ukoliko se operacija potvrđivanja izvrši unutar kritične sekcije, mora da se brava zaključa. U suprotnom zaključavanje može da se izbegne. Hardver detektuje početak kritične sekcije i pokušava da izbegne operaciju potvrđivanja u kritičnoj sekciji. To nije moguće uraditi ako je kritična sekcija dugačka. U radu nije prikazana evaluacija predloženog rešenja.

Autori u [51] predlažu rešavanje konflikta uzimajući u obzir prioritete niti i politiku raspoređivanja

operativnog sistema. Prilikom konflikta poništava se ona transakcija koja se izvršava u niti kojoj raspoređivač operativnog sistema pridaje manju važnost (npr. veću važnost pridaje nitima koje moraju da se izvršavaju u realnom vremenu). Ukoliko su niti iste važnosti, upoređivanje se radi na osnovu prioriteta niti. Ideja je da se obe vrednosti koduju kao celi brojevi i da se za vreme izvršavanja niti pamte u registru posebne namene. Taj registar je dostupan samo operativnom sistemu i njegova vrednost se postavlja prilikom raspoređivanja niti na procesor. Ukoliko niti imaju isti prioritet onda se primenjuje algoritam sa vremenskim markama. Ideja ovakvog rešavanja konflikta je da se spreči inverzija prioriteta ili politike raspoređivanja prilikom poništavanja transakcija. Autori su testirali implementaciju na operativnom sistemu Linux u kome su kritične sekcije operativnog sistema zamenili transakcijama. Pokazali su da se u takvom sistemu inverzije ne dešavaju, uz vrlo malo usporenje izvršavanja.

3.1.2 Neograničene transakcione memorije

Problem koji je dugo okupirao naučnu zajednicu je kako izbeći poništavanje transakcije koje nije vezano za konflikte. Iako se predlažu brojna rešenja, većina tih rešenja nisu uključena u moderne komercijalno dostupne procesore, zbog svoje kompleksnosti. Uzroci za takva poništavanja su prekoračenje kapaciteta keš memorija (ili nekih drugih struktura gde se vodi evidencija o podacima kojima je pristupano u transakciji), promene konteksta, generisanje stranične greške, generisanje prekida i izvršavanje ulazno-izlaznih operacija u toku izvršavanja transakcije.

Prekoračenje kapaciteta se pokušava rešiti nekom vrstom virtualizacije gde bi se podaci kojima je pristupano u transakciji, a za koje nema mesta u keš sistemu, premestili u operativnu memoriju. Takva rešenja mogu da se implementiraju u hardveru ili softveru. Mogu da budu izuzetno kompleksna za implementaciju i vreme potrebno za njihovu primenu može da bude nekoliko redova veličine veće od izvršavanja kratkih transakcija.

Promena konteksta i generisanje stranične greške predstavljaju problem za transakciono izvršavanje, jer dok se proces ne izvršava ne može da se radi provera konflikta ili njegovi podaci moraju da se izbace u operativnu memoriju. Takođe, ubacivanje i izbacivanje stranica može da promeni fizičke adrese podataka i ako se fizičke adrese podataka pamte u strukturama za transakcionu memoriju može doći do neispravnog izvršavanja. Slična virtualizacija kao i kod prekoračenja kapaciteta može da se iskoristi za rešavanje ovog problema.

Ulazno-izlazne operacije i prekidi ne mogu da se izvršavaju u transakcijama jer su njihovi efekti neponištivi (npr. kada se podatak pošalje na disk, ne može se poništiti efekat te operacije). Predlažu se rešenja sa pauziranjem transakcije gde bi se te operacije uradile na stardardni način. To naravno stvara druge probleme. Kada se transakcija poništi i ponovo pokrene, ako je ulazno-ulazna operacija urađena pre nego što se transakcija poništila, operacija ne sme da se izvrši ponovo.

U tabeli 3.2 su grupisana rešenje koja pokušavaju prevazići prethodno navedene probleme. Za svako rešenje je navedeno koji problem se razmatra, ko učestvuje u rešavanju problema hardver (H) ili softver (S), da li je rešenje kompleksno za implementaciju (C) i da li je vremenski zahtevno (T).

Radi olakšanja posla programeru u situacijama kada transakcija dosegne hardversko ograničenje (npr. spekulativno upisan podatak u keš memoriji mora da se zameni, dogodila se stranična greška, proces se suspenduje zbog dodeljenog vremenskog intervala) u [61] se predlaže hardversko rešenje koje omogućava da transakcija izvrši prekoračenje i da se uspešno izvrši. To se postiže pamćenjem spekulativnog stanja na posebno mesto u memoriji rezervisano samo za te slučajeve. Može da se zapamti samo deo promena ili celokupan kontekst transakcije. Hardver obezbeđuje da se provera konflikta radi i nad podacima koji su zapamćeni na taj način. U posebnom registru se pamti stanje transakcije. Ukoliko nije došlo do prekoračenja, transakcija se izvršava na uobičajen način. Ukoliko jeste, prilikom pristizanja informacije o pristupu nekom podatku od strane druge transakcije, mora se vršiti provera da li se taj podatak nalazi zapamćen u memoriji. U radu se predlaže prvo jednostavna provera pomoću Blumovog filtra, pa tek onda pristup memoriji. Ta provera da li se podatak nalazi

u memoriji može dugo da traje jer se vrši sekvencijalno i pristup memoriji traje mnogo duže nego pristup lokalnom baferu. Kako autori tvrde, na ovakav način se izbegava uticaj na performanse kada se transakcija izvršava bez prekoračenja. Prilikom potvrđivanja transakcije koja je prekoračila ograničenja, mora da se sačeka da svi zapamćeni podaci postanu nespekulativni. Ta operacija traje i za vreme nje, druge transakcije se pauziraju ukoliko pristupaju tim podacima ili mora da se obezbedi da mogu da pristupe najnovijoj verziji. Iako ovakva tehnika značajno olakšava programiranje aplikacija koje koriste transakcionu memoriju, ne postoje informacije u radu kako ova tehnika utiče na broj poništavanja transakcija i kako utiče na performanse aplikacija.

Problem hardverskog ograničenja transakcija razmatran je i u [47]. Autori prvo predlažu teorijsko rešenje koje omogućava da transakcija bude velika koliko god to virtuelna memorija dozvoljava. Zatim predlažu pojednostavljeno rešenje koje je moguće implementirati u savremenim arhitekturama. Teorijsko rešenje dozvoljava da se vrše promene podataka u memoriji za vreme transakcije. Promena u memoriji se vrši kad promenjen podatak mora da se izbaci iz keš memorije zbog ograničenog kapaciteta keš memorije. Tada se stara vrednost podatka pamti u za to posebno određeno mesto u memoriji. Najbolje mesto za to je neki vid globalne virtuelne memorije za koju je odgovoran operativni sistem. Prilikom poništavanja transakcije, svaki promenjen podatak mora da se restaurira. Autori tvrde da bi se to dešavalo u malom broju slučajeva i da ne bi znatno uticalo na performanse aplikacije. Za svaki podatak tada mora postojati i dodatna informacija da li je taj podatak spekulativno promenjen i gde mu se nalazi stara vrednost. Predlaže se da te informacije budu u virtuelnoj memoriji samog procesa. Prema tome, ukoliko nema prostora u fizičkoj memoriji da se sve to zapamti, neki delovi koji trenutno nisu potrebni, mogu se premestiti na disk. Takođe, pošto je sve virtuelizovano, proces se može suspendovati u toku izvršavanja transakcija zbog isteka vremenskog kvanta izvršavanja ili zbog stranične greške i transakcije ne moraju da se poništavaju.

Autori kažu da je prethodno rešenje isuviše kompleksno i predlažu uprošćenje. Uprošćena tehnika pamti promenjen podatak koji je izbačen iz keš memorije u za to posebno određeno mesto u memoriji. U keš memoriji se mesto sa koga je blok izbačen obeleži bitom izbacivanja. Kada jezgro dobije informaciju da je pristupljeno nekom podatku koji bi mogao da bude u bloku u keš memoriji koji je obeležen bitom izbacivanja, vrši se provera da li se taj podatak stvarno i izbačen, proverom mesta gde bi trebao da se nađe taj podatak u memoriji. Podaci se smeštaju u memoriju na mesto koje je određeno heširanjem dela adrese podatka. Autori su za testiranje svoje ideje simulirali programe kod kojih je sinhronizacija urađena pomoću zaključavanja brave, a gde su oni svaku kritičnu sekciju pretvorili u transakciju. Rezultati simulacije pokazuju da su transakcije u 99% slučajeva kratke i da ne dolazi do prekoračenja i da u takvim slučajevima njihovo rešenje ne utiče na performanse izvršavanja programa. Nije uzeto u obzir da su postojeće aplikacije optimizovane, da kritične sekcije budu što kraće i kako bi se ponašale nove aplikacije koje su isključivo pisane za tehniku transakcione memorije.

Tabela 3.2: *Radovi koji predlažu neograničenu transakcionu memoriju*

Rešenje	Stranična greška	Prekoračenje	Promena konteksta	I/O i prekidi
[61]	-	HCT	-	-
[47] oba	HCT	HCT	HCT	-
[48] [49]	H	H	-	-
[53]	H	H	HC	-
[50]	ST	H	ST	-
[62] [70] [54] [55] [71]	ST	ST	ST	-
[64]	HC	HC	HC	-
[72] [51]	-	-	-	HC
[73]	-	HST	-	HST

U već opisanim rešenjima [48], [49] transakcije mogu da budu proizvoljne dužine, jer se arhiva nalazi u virtuelnoj memoriji i omogućeno je izbacivanje spekulativnih podataka iz keš memorije. Rešenje je dodatno uprošćeno u odnosu na prethodna, jer ne predviđa proveru arhive da li je došlo do konflikta kod velikih transakcija. To se obezbeđuje izvršenom promenom protokola keš koherencije. Ovakvo rešenje nema dugu operaciju provere arhive, ali može prouzrokovati mnogo lažnih konflikata. Predloženo rešenje ne podržava promenu konteksta ili sistemske pozive unutar transakcija.

Slična tehnika sa korišćenjem arhive se predlaže i u već opisanom rešenju [53]. Nadogradnja je omogućavanje izvršavanja transakcije čak iako dođe do preuzimanja konteksta unutar transakcije. Ideja je da za svaku nit, koja se izvršava na jednom procesoru, postoji postoje posebni skupovi za čuvanje adresa. Svi oni se proveravaju u trenutku kad se vrši provera konflikta. Broj niti koje na takav način mogu da se prekinu je ograničen brojem dostupnih skupova za čuvanje adresa. Kombinacija prethodna dva rešenja [50] predlaže postojanje skupa svih podataka kojima su pristupale niti koje su pauzirane zbog promene konteksta. Pošto se pošiljalac poništava, dovoljno je samo proveriti taj skup i obavestiti pošiljaoca o konfliktu. Taj skup se ažurira prilikom promene konteksta i prilikom potvrđivanja neke niti. Ažuriranje vrši operativni sistem. Straničenje je podržano, jer je arhiva u virtuelnoj memoriji, ali problem može nastati kada se stranica premesti sa jedne fizičke adrese na drugu. Fizičke adrese podataka se čuvaju u skupovima. Time promena lokacije stranice dovodi do nekorektnog sadržaja skupova. Autori predlažu da operativni sistem ažurira skupove prilikom obrade stranične greške.

Rešenje predloženo u [64] koristi mehanizam virtuelne memorije za podržavanje velikih transakcija. Dok se ne prekorači veličina transakcije, može se koristiti bilo koja implementacija transakcione memorije. Kada se prekorači hardversko ograničenje transakcije, neke virtuelne stranice dobijaju dodatnu fizičku stranicu. Ideja je da se spekulativan podatak pamti u jednoj fizičkoj stranici, a nespekulativan podatak u drugoj fizičkoj stranici. Trenutno validan podatak može da se nađe tačno u jednoj fizičkoj stranici. Gde se nalazi, pamti se u jednom bit vektoru, koji postoji za svaku virtuelnu stranicu koja se preslikava u dve fizičke stranice. Prilikom potvrđivanja transakcije, spekulativni podaci postaju nespekulativni. To se radi promenom bit vektora, tako što se za promenjene podatke u transakciji odgovarajući bit komplementira. Prilikom poništavanja transakcije, nije potrebno ništa raditi, jer bit vektor ispravno pokazuje koji su podaci nespekulatvni. Nakon operacije potvrđivanja ili poništavanja, hardver za preslikavanje mora da obezbedi da se uvek pristupa onom podatku za koji bit vektor pokazuje da je nespekulativan. Nakon transakcije, validni podaci iz jedne fizičke stranice mogu lenjo da se kopiraju u drugu fizičku stranicu. Kad jedna fizička stranica sadrži sve validne podatke, druga stranica može da se dealocira. Prilikom detekcije konflikta, postoje dodatna dva bit vektora za svaku virtuelnu stranicu koja ima dve fizičke stranice. Jedan bit vektor sadrži informacije sa kojih adresa su pročitani podaci u transakcijama koje se trenutno izvršavaju, dok drugi sadrži informacije na koje adrese je izvršen upis podataka u transakcijama koje se trenutno izvršavaju. Na osnovu tih bit vektora se određuje da li je došlo do konflikta ili ne kada pristigne informacija o upisu od strane nekog procesora preko protokola keš koherencije. Hardver koji vrši sve navede poslove je centralizovan i nalazi se u memorijskom kontroleru, dok se podaci koji su mu potrebni za rad čuvaju u posebnom delu virtuelne memorije procesa. Radi bržeg pristupa tim podacima dodata je keš memorija koja služi samo za ubrzavanja pristupa tim podacima. Pošto se radi o centralizovanom sistemu mora postojati način da se odredi šta je koja transakcija radila sa podacima (npr. da se zna koji spekulativni podaci treba da postanu nespekulativni). Za to se koristi ulančana lista koja za svaku transakciju čuva informacije kojim podacima je pristupano. Tu listu kontroliše hardver koji se takođe nalazi u memorijskom kontroleru. Lista se takođe nalazi u posebnom delu virtuelne memorije procesa i postoji posebna keš memorija za ubrzavanje pristupa samoj listi. Prilikom konflikta taj centralizovani sistem određuje koja transakcija treba da se poništi. Koristi se algoritam vremenskih marki. Pošto se svi podaci nalaze u virtuelnoj memoriji moguća je promena konteksta u samoj transakciji. Autori su pokazali da ovakav sistem brže izvršava aplikacije od ranije opisanih sistema i od aplikacija koje koriste bravu za sinhronizaciju.

Nekoliko radova predlaže korišćenje softverske implementacije transakcione memorije za

rešavanje hardverskog ograničenja transakcione memorije. Transakcije koje mogu da se izvrše hardverski se izvršavaju hardverski, a kad se utvrdi da se transakcija ne može izvršiti hardverski pokreće se softverska transakcija. Kod softverskih implementacija transakcija, obično se evidencija o podacima vodi na nivou objekta [74], za razliku od hardverske implementacije gde se evidencija obično vodi na nivou bloka keš memorije ili procesorske reči. Objekat obično ima polje za stanje, referencu u kom stanju se nalazi transakcija i same podatke. Kad se vrši čitanje podataka iz objekta proverava se u kom je stanju objekat i ako niko drugi ne upisuje onda nema konflikta, u suprotnom ima. Kad se vrši upis, prvo mora da se napravi kopija podataka (uključuje i dinamičku alokaciju prostora za kopiranje), pa se tek onda vrši upis u kopiju. Prilikom potvrđivanja transakcije original se uklanja, a kopija se proglašava originalom. Prilikom poništavanja, kopija se uništava. Softverska implementacija je znatno sporija od hardverske implementacije, jer se velika količina podataka kopira, dok dozvoljava da transakcije budu proizvoljne veličine.

Autori u [62] predlažu da hardverske i softverske transakcije koriste iste objekte u kodu. Kada se izvršava hardverska transakcija, standardno se čuvaju spekulativni podaci i standardno se detektuju konflikti. Kada neka transakcija ne uspe nekoliko puta pokreće se softverska verzija te transakcije. U softverskoj verziji, pomoću specijalne instrukcije, stanje objekta kojem se pristupa, ubacuje se u transakcioni bafer. Na taj način podaci kojima je pristupila softverska transakcija su vidljivi i hardverskim transakcijama i konflikt između softverske i hardverske transakcije može da se detektuje. Detekcija u tom slučaju je na nivou objekta, isto kao i u slučaju kad su dve softverske transakcije u pitanju. Rezultati testiranja pokazuju da je ovakvo rešenje značajno bolje nego samo softversko rešenje, dok je lošije nego samo hardversko rešenje. Testiranje je izvršeno na jednostavnim i optimizovanim aplikacijama koje imaju male transakcije i kao takve više pogoduju hardverskoj implementaciji.

Autori u [70] predlažu softverske transakcije koje vode evidenciju o podacima na nivou procesorske reči i na taj način se smanjuje broj lažnih konflikata. Evidencija se vodi u heš tabeli. U evidenciji se između ostalog nalazi i stanje u kome se ta reč nalazi (nespekulativno, spekulativno čitanje ili spekulativan upis). Hardverska implementacija transakcija može da bude bilo kakva, ali mora da podržava instrukciju eksplicitnog poništavanja transakcije. Prevodilac treba da pre čitanja ili upisa neke lokacije ubaci kod koji čita iz heš tabele kakvo je stanje reči u softverskim transakcijama i ako ne odgovara željenoj operaciji, transakcija se eksplicitno poništava. Na taj način se u skupu pročitanih podataka nalaze stanja svih reči kojima je pristupano u hardverskoj transakciji i ako softverska transakcija promeni to stanje, hardverska transakcija se poništava zbog konflikta. Rezultati testiranja pokazuju da je ovakvo rešenje bolje u odnosu na čisto softverska rešenja, ali da nije bolje u odnosu na čisto hardversko rešenje. Predlažu se neke optimizacije da se jaz u performansama smanji, ali se zaključuje da jaz ne može da se ukloni. Limitirajući faktor su čitanja stanja koja moraju da se obave pre svakog čitanja ili upisa podatka u hardverskoj transakciji, a koji usporavaju izvršavanje programa.

Obrnut pristup, gde se transakcije izvršavaju softverski, a neki delovi se ubrzavaju hardverski predložen je u nekoliko radova. Ideja u [54] je dodavanje instrukcija koje omogućavaju da se podaci kojima pristupa softverska transakcija obeleže u keš memoriji. Ta instrukcija se poziva svaki put kad se podacima prvi put pristupa u transakciji. Hardver vodi evidenciju koliko je podataka koji su obeleženi izbačeno iz keša zbog invalidacije (bilo zbog protokola keš koherencije bilo zbog zamene bloka). Ta informacija se čuva u jednom brojaču, koji se resetuje svaki put kad se počne sa izvršavanjem transakcije. Softverska transakcija prilikom potvrđivanja proverava da li je taj brojač i dalje na nuli. Ako je na nuli, sigurno nije došlo do konflikta, jer je spekulativan podatak i dalje u keš memoriji i transakcija se potvrđuje. Na taj način se izbegava softverska provera da li je došlo do konflikta. U slučaju da je brojač veći od nule, prelazi se na softversku proveru da bi se izbegli lažni konflikti. Rezultati testiranja pokazuju da je ovakav pristup bolji od čisto softverskih transakcija i da daje slične rezultate u poređenju sa hardverskom implementacijom, ali samo za jednonitne programe. Nije izvršeno poređenje sa hardverskom implementacijom za višenitni program, već se samo tvrdi da neće dati gore rezultate u slučaju da su transakcije velike, jer bi i hardverska implementacija morala da poništi transakciju, a sinhronizacija bi morala da se uradi na drugačiji način.

Komplikovaniji predlog, koji pored izbegavanja softverske provere da li je došlo do konflikta, izbegava i kopiranje objekata, dat je u [55]. Za implementaciju tog predloga je potrebno obezbediti dodatne instrukcije i promeniti protokol keš koherencije, dodavanjem novih transakcionih stanja u kojima mogu da budu blokovi u keš memoriji. Promene u keš koherenciji prouzrokuju samo dodavanje novih vrsta poruka između procesora i keš memorije i jednu novu poruku između keš memorija, pa autori tvrde da promene ne otežavaju značajno implementaciju samog protokola. Softverske transakcije vode evidenciju na nivou objekta. U hardveru se različito tretiraju zaglavlje objekta i ostali podaci u objektu. Kada se prvi put pristupi objektu njegovo zaglavlje se učitava u keš memoriju. Podaci iz zaglavlja učestvuju na standardan način u protokolu keš koherencije i mogu izazvati prekid transakcije usled konflikta. Pored zaglavlja, svi ostali podaci kojima se pristupa u okviru objekta se učitavaju u keš memoriju u posebnom transakcionom stanju, ali za njih je protokol keš koherencije izmenjen i ne mogu odmah izazvati prekid transakcije prilikom konflikta, već se transakcija sa konfliktom nastavlja. Ako izazovu konflikt, samo im se dodeljuje odgovarajuće transakciono stanje. Samo jedna transakcija, koja ima konflikt, a nije poništena, može da se potvrdi. O tome odlučuje softver proverom da li se narušila atomičnost. Provera se radi analizom podataka koji su transakcionim stanjima u svim aktivnim transakcijama. U slučaju da neki podatak u transakcionom stanju mora da se izbaci iz keš memorije zbog kapaciteta, transakcija se poništava i pristupa se izvršavanju softverske transakcije, bez pomoći hardvera. Nije prikazana evaluacija predloženog rešenja.

Ubrzanje softverskih transakcija korišćenjem hardverski implementiranih skupova dat je u [71]. Softverske transakcije vode evidenciju o podacima na nivou jedne reči i upisi se pre potvrđivanja pamte u poseban bafer. Tokom potvrđivanja, podaci se iz bafera upisuju na odgovarajuće lokacije. Kada se pristupi nekom podatku, njegova adresa se stavlja u odgovarajući hardverski implementiran skup. Ubacivanje u skup se vrši pomoću posebnih instrukcija. Procesor, kada primi poruku o upisu od strane drugog procesora, proverava da li se ta adresa nalazi u njegovom radnom skupu i ako se nalazi detektuje konflikt. U tom slučaju generiše prekid. Prekid pokreće softversku obradu konflikta. Ovakva implementacija može da reaguje na poruku koja je prouzrokovana i nespekulativnim upisom. Ovo je prvo predloženo rešenje sa softverskim transakcijama koje realizuje jaku atomičnost. Prilikom potvrđivanja svaki upisani podatak se posebnom instrukcijom dovlači u keš memoriju. Time se upisi javljaju ostalim procesorima. Kada su svi podaci dovučeni u keš, počinje se njihovim upisom. Dok se vrši upis, zahtevi za dohvatanje tih podataka od strane drugih procesora se odbijaju. Evaluacija rešenja pokazuje da za neke aplikacije, performanse izvršavanja su slične sa hardverskim implementacijama, dok za neke aplikacije performanse mogu da budu lošije do dva puta.

Autori u [75] predlažu poboljšanje hibridne implementacije transakcione memorije. Ideja je da se detektuju različite faze u izvršavanju programa i da u zavisnosti od faze programa se pokrene odgovarajuća verzija transakcije. Za svaku transakciju prevodilac pravi tri verzije transakcije: 1) hardverska transakcija bez proveravanja konflikta sa softverskim transakcijama, 2) hardverska transakcija sa proveravanjem konflikta sa softverskim transakcijama i 3) softverska transakcija. Za svaku fazu se definiše koja varijanta transkacije se pokreće. Recimo, ako je faza takva da sve transakcije mogu da se izvrše u hardveru, pokreće se prva varijanta transakcije. Ovakav način izvršavanja omogućava smanjenja uticaja na performanse koje imaju česte provere da li su u konfliktu hardverska i softverska transakcija. Postoje varijante transakcija koje ne smeju istovremeno da se izvršavaju. Prelazak iz faze u fazu, može promeniti varijantu transakcija koje se pokreću. U nekim prelazima mora se sačekati da se transakcije neodgovarajuće vrste završe pre nego sto se startuju varijante transakcija koje odgovaraju tekućoj fazi.

Studija mogućih načina za implementaciju hardverskih skupova je prikazana u [76]. Polazi se od činjenice da skupovi implementirani kao Blumovi filtri mogu zauzeti značajan prostor na čipu. Uzrok su višeportne memorije koje su potrebne za implementaciju skupova. Memorija mora da ima onoliko portova koliko ima i heš funkcija. Verovatnoća da se dogodi lažni pogodak u malom skupu je manja za one implementacije koje imaju veći broj heš funkcija. Za veće skupove bolje je koristiti samo jednu heš funkciju. Autori predlažu da se memorija podeli na jednake delove. Broj delova je jednak

broju heš funkcija. Svaka heš funkcija mapira upise u samo jedan deo. Time se postiže da postoji više jednoportnih memorija. Prostor na čipu za implementaciju takvih skupova je nekoliko puta manja od običnih Blumovih filtara. Ispostavlja se da je verovatnoća da se dogodi lažni pogodak jednaka kao i kod običnih Blumovih filtara, ako je broj heš funkcija značajno manji od veličine memorije u bitovima. Veličina u bitovima je uvek bar red veličine veća od broja heš funkcija. Pored ove implementacije autori predlažu još jednu implementaciju skupova. Implementacija koristi algoritam za heširanje iz [77]. Implementacija skupa pomoću tog algoritma ima manju verovatnoću da se dogodi lažni pogodak, ali operacija umetanja u skup može da traje dugo. Princip ubacivanja podatka u skup je sledeći. Pokuša se ubacivanje. Ako dođe do kolizije sa već ubačenim podatkom, taj podatak se izvadi i ponovo se ubaci, samo na drugo mesto. Ta operacija se rekurzivno ponavlja dok ubacivanje ne uspe bez kolizije. Autori predlažu da se ta implementacija koristi dok je skup mali i dok je broj kolizija neznatan. Kada skup poraste, predlaže se prelazak na implementaciju skupa pomoću Blumovog filtra. Implementacija takvog skupa zauzima sličan prostor na čipu kao i u prethodnom predlogu. Simulacija pokazuje da je ovakvo rešenje bolje od Blumovog filtra sa dve heš funkcije, dok je slična Blumovom filtru sa četiri heš funkcije.

Autori u [72] predlažu način za obradu prekida unutar transakcija. Njihova tehnika se zasniva na instrukcijama koje mogu da zapamte i restauriraju stanje transakcije. Prilikom obrade stanje transakcije se zapamti i sama transakcija se pauzira. Nakon toga se izvršava prekid van transakcije i kada se prekid završi transakcija se nastavlja kao da se prekid nije dogodio. Podrška za višestruke transakcije je potrebna da bi se pauziranje moglo implementirati, u slučaju da se moraju detektovati i konflikti sa pauziranom transakcijom. Autori razmatraju problem živog blokiranja kad transakcija može da se pokrene u prekidnoj rutini. Ako dođe do konflikta između transakcije u prekidnoj rutini i pauzirane transakcije, mora da se poništi pauzirana transakcija, jer bi u suprotnom transakcija u prekidnoj rutini mogla biti restartovana beskonačno puta. Autori u [51] predlažu nadogradnju koja podržava i ulazno-izlazne (eng. *input/output*, skraćeno I/O) operacije unutar transakcija. Ideja je da se dodaju komplikovanije instrukcije za početak transakcije koje mogu da izvrše i zaključavanje brave ako je potrebno. Kada se detektuje I/O operacija, transakcija se završava, brava se zaključava i nastavlja se netransakciono izvršavanje kritične sekcije.

Slično rešenje za I/O operacije i rešavanje prekoračenja se predlaže u [73]. Programer treba eksplicitno da ubaci instrukcije za pauziranje izvršavanja transakcije kada treba da se izvrši neka neponištiva operacija ili kada se vrši upis u neku memorijsku lokaciju koja ne treba da se restaurira prilikom prekida. Svi podaci, kojima se pristupilo za vreme pauzirane transakcije, treba da se koriste za detekciju konflikta. Stara vrednost podataka koji treba da se restauriraju, a promenjeni su za vreme pauze, treba da se ubace u listu. Prilikom poništenja transakcije, treba da se pozove funkcija koja će restaurirati podatke redom iz liste. Ovakav pristup stavlja dodatan teret na programera koji bi morao o svakom upisu da odlučuje da li da se uradi u transakciji ili za vreme pauze, kao i da vodi računa o restauriranju promenjenih podataka u toku pauze.

3.1.3 Optimizacije

Povećanje broja transakcija koje se izvršavaju paralelno dovodi do povećane šanse za konflikt. Kada taj broj dostigne određenu granicu, napredovanje transakcija je otežano i dolazi do zagušenja, koje se obično završava izvršavanjem transakcija pomoću brava. Rešavanje tog problema zahteva raspoređivanje transakcija, tj. odlaganje transakcija, da bi se zagušenje smanjilo ili izbeglo. Postojeća rešenja predlažu uspavljivanje niti, korišćenje dodatnih brava ili korišćenja redova za čekanje. Procena da li transakciju treba odložiti radi se pomoću istorije izvršavanja ili analize koda za vreme prevođenja. Ulazni podaci za algoritme su broj potvrđivanja i poništavanja niti, radni skupovi transakcije ili informacija koje su se transakcije istovremeno izvršavale.

Prvo su nastajala rešenja koja su predlagala algoritme koji rade na osnovu preciznih informacija. Očekivalo se da će hardver moći da dostavi te podatke. Kada su napravljeni prvi komercijalni

procesori koji nisu imali te sofisticirane mogućnosti, pojavila su se rešenja koja rade sa nepreciznim informacijama. Poslednja su se pojavila rešenja koja koriste mašinsko učenje za odlučivanje o parametrima raspoređivanja [78].

Raspoređivanje, koje uključuje rad raspoređivača operativnog sistema, je prikazano u [51]. Za svaku transakciju vodi se evidencija koliko često se transakcija ponovo pokrenula usled konflikta u prethodnom periodu i koliko je čekala da ponovo pokuša izvršavanje. Ako se transakcija često ponovo pokreće i čekanje je bilo duže nego promena konteksta niti, onda se niti oduzima procesor i neka druga nit sličnog prioriteta se odabira za izvršavanje. Eksperimenti su sprovedeni na implementaciji Linux operativnog sistema gde su transakcije korišćenje samo za kritične sekcije sistemskog koda. Nisu dobijena značajnija poboljšanja, jer u takvom sistemu nema mnogo zagušenja. Autori su osmislili mikro testove koji pokazuju da u slučaju kad postoji zagušenje, ovakvo rešenje poboljšava performanse izvršavanja i smanjuje broj ponovnog pokretanja transakcija.

Autori u [73] predlažu korišćenje rešenja za virtualizaciju iz [61] radi omogućavanja softverskom raspoređivaću da pauzira nit radi izbegavanja poništavanja usled konflikta. Autori tvrde da kada se konflikt dogodi, jedna transakcija je vlasnik konfliktnog podatka dok druga transakcija traži taj podatak. Predlažu da se druga transakcija pauzira, sve dok se prva ne završi. To se može postići generisanjem prekida za nit koja izvršava drugu transakciju. U obradi prekida nit treba da se izbaci iz raspoređivača sve dok se prva transakcija ne završi. Da bi to bilo moguće prva transakcija treba da ima listu niti koje čekaju da se ona završi. Kad se ta transakcija završi, sve te niti treba da se aktiviraju. Za čuvanje te liste koristi se tehnika virtuelizacije. Autori nisu prikazali nikakvu eksperimentalnu analizu kakve bi benefite pružilo ovakvo rešenje.

Sprečavanje zagušenja predloženo je u [79]. Opisano rešenje limitira broj transakcija koje mogu da se izvršavaju paralelno. Limit se određuje dinamički u toku izvršavanja. Svaka nit za sebe određuje koliko je zagušenje u sistemu. Zagušenje se određuje na osnovu istorije izvršavanja transakcija. Što je bilo više poništavanja transakcija, to se određuje da je zagušenje veće. Kada nit naiđe na početak transakcije, ako je zagušenje veće od nekog limita, nit traži od raspoređivača operativnog sistema da je pauzira. U suprotnom, započinje transakciju. Niti, koje su pauzirane, smeštaju se u jedan red i beleži se koju transakciju žele da izvrše. Kada se ni jedna transakcija ne izvršava zbog koje je transakcija bila poništavana, uzima se jedna nit iz reda i pušta se da nastavi rad. Pokazano je da ovakvo rešenje radi bolje nego kad se transakcije izvršavaju zaključavanjem globalne brave.

Autori u [80] predlažu tehniku za smanjene poništavanja zbog konflikata. Osnovna ideja rešenja je da se uspostavi određeni redosled transakcija. Podatak koji promeni neka transakcija može da se prosledi sledećoj transakciji koja se nalazi u redosledu. Iako konflikt postoji, transakcije se ne poništavaju, jer konačan efekat će biti kao da se prvo izvršila prva transakcija pa onda druga. Na taj način se formira zavisnost druge transakcije od prve. Ukoliko se prva poništi zbog nekog razloga mora i druga da se poništi. Nije dozvoljeno da se uspostavi ciklična zavisnost, jer ona narušava logičku ispravnost izvršavanja. Rešenje je implementirano pomoću novog protokola keš koherencije koji omogućava praćenje prosleđivanja podataka i zavisnosti koje se formiraju između transakcija. Rešenje donosi ubrzanje u odnosu na konvencionalne implementacije transakcione memorije. Mana je što iziskuje potpuno novi protokol keš koherencije.

Rešenje koje predlaže odlaganje izvršavanja transakcije dato je u [81]. Kada se detektuje konflikt između dve transakcije, jedna od njih se prekida. Prekinuta transakcija se ne pokreće ponovo dok se prva transakcija ne završi. Rešenje se realizuje u hardveru. Kada se neka transakcija poništi, ona putem keš koherencije obavesti jezgro koje je izvršavalo transakciju koja je prouzrokovala poništavanje, da poništena transakcija čeka ponovno izvršavanje. Jezgro ima niz bitova za čuvanju informacije o čekanju za izvršenje transakcije, po jedan bit za sva ostala jezgra. Kada bit ima vrednost jedan, to znači da neko drugo jezgro čeka na izvršavanje transakcije. Kada se potvrdi transakcija, jezgro šalje svim drugim jezgrima koja čekaju na njega da pokrenu transakciju ponovo. Na ovaj način jezgro može dugo da ne izvršava ništa, ako se transakcija na koju čeka izvršava dugo, jer raspoređivač operativnog sistema nema mogućnost da odreaguje na takvo čekanje.

Autori u [82] predlažu praćenje radnog skupa transakcija i predviđanje kakvi će biti radni skupovi budućih transakcija. Radni skup adresa sa kojih je vršeno čitanje podataka se predviđa da će biti sličan kao kod poslednjih nekoliko izvršavanja transakcije, dok se predviđa da radni skup adresa na koje se upisuju podaci je isti kao radni skup poslednjeg izvršavanja transakcije. Radni skup za upise je dobro procenjen samo kada je poslednje izvršavanje transakcije bilo neuspešno. Kada neka transakcija, koja se u bliskoj prošlosti više poništavala nego potvrđivala, treba da se pokrene, njen procenjen radni skup se upoređuje sa radnim skupovima svih ostalih transakcija koje se izvršavaju. Ako se utvrdi da postoji preklapanje radnih skupova koji će izazvati konflikt, transakcija se izvršava zaključavanjem globalne brave. Iako su autori napisali da se predloženo rešenje može koristiti u hardverskoj transakcionoj memoriji, nisu naveli predlog implementacije.

Autori u [83] predlažu unapređenje prethodnog rešenja. Za svaku transakciju formira se radni skup i on se pamti u programski dostupnom registru. Na sličan način se pamti jedinstveni identifikator transakcije koja je prouzrokovala konflikt i veličina radnog skupa. Radni skup se pamti u Blumovom filtru. Za svaku transakciju se pamti brojač kolika je šansa da se konflikt dogodi sa nekom drugom transakcijom. To se pamti u jednoj matrici. Za svako jezgro se vodi evidencija koja transakcija se izvršava na njemu. Kada neka transakcija treba da se izvrši, softver proverava matricu i da li se konflikt može dogoditi sa bilo kojom transakcijom koja se trenutno izvršava na ostalim jezgrima. Ako ima takve transakcije i radni skup transakcije je velik, tada se transakcija odlaže jednostavnim prepuštanjem jezgra nekoj drugoj niti. Pamti se koja transakcija prouzrokuje čekanje. Kada se transakcije prekine, softver pročita koja je transakcija prouzrokovala konflikt iz programski dostupnog registra i ažurira matricu, tako što poveća vrednost u brojaču. Takođe, softver za svaku transakciju pamti radni skup i njegovu veličinu kada se transakcija potvrdi. U tom trenutku, ako je čekala na neku drugu transakciju, proverava se da li je uzaludno čekala (proverom radnih skupa transakcija). Ako je uzaludno čekala, vrednost brojača za verovatnoću konflikta se smanjuje.

Jedan koncept sličnosti transakcija je predložen u [84]. Što je veći odnos veličine preseka radnih skupova transakcije i prosečne veličine radnog skupa transakcije, to je sličnost veća. Taj koncept treba da predvidi da li će se ponovljenim izvršavanjem određene transakcije uglavnom pristupati istim podacima ili ne. Za računanje sličnosti koriste se modifikovani Blumovi filtri koji omogućavaju procenu veličine skupa. Vođenje evidencije o konfliktima između transakcija radi se na isti način kao u prethodnom rešenju iz [83], uz modifikacije kao što su: dodavanje malih memorija za ubrzavanje pristupa podacima kada se odlučuje da li transakciju treba izvršiti ili komprimovanje struktura podataka radi zauzimanja manje prostora u memoriji. Sličnost se koristi da se odredi koliko će brojač za procenu šanse da se konflikt dogodi, promeniti.

Pored standardnih transakcija u komercijalno dostupnim procesorima postoji i varijanta sa kreiranjem transakcije oko već napisane kritične sekcije koja zaključava brave [85]. Ta varijanta izbegava zaključavanje brave i izvršava kritičnu sekciju kao transakciju, a bravu stavlja samo u radni skup pročitanih podataka transakcije (jer implementacija transakcione memorije ne garantuje napredak). Autori u [86] predlažu grupisanje transakcija koje mogu da se nađu u konfliktu. Svakoj toj grupi se dodeli jedna pomoćna brava. Samo se brave te vrste zaključavaju van transakcija, kada transakcija ne uspe nekoliko puta. Brave koje se inicijalno nalaze u kodu se uvek izbegavaju na već pomenuti način. U radu nije objašnjeno kako se grupišu transakcije i kako se dodeljuju pomoćne brave.

Pored izvršavanja pomoću transakcija i pomoću zaključavanja brava autori u [87] predlažu da postoji i mogućnost da se kritična sekcija izvrši optimistički u softveru. Optimistično izvršavanje u softveru podrazumeva podelu kritične sekcije u male celine. Na kraju svake celine se proverava da se slučajno verzija podatka nije promenila (neka druga nit upiše u podatak). Ako se verzija promenila, kritična sekcija se ponavlja. Rešenje skuplja statističke podatke o izvršavanju kao što su broj pokušaja da se izvrši kritična sekcija, koliko dugo traje, itd. Pamti se koja brava se koristi za tu kritičnu sekciju i gde se izvršava ta kritična sekcija (programer imenuje delove koda pomoću makroa). Svaki put kada treba da se izvrši kritična sekcija, određuje su u kojoj varijanti treba da se izvrši na osnovu skupljenih podataka. Problem sa ovakvim rešenjem je što kod aplikacije mora da se menja, radi implementacije

varijante sa optimističkim izvršavanjem u softveru.

Autori u [88] razmatraju pitanje šta treba uraditi kada se transakcija poništi, pod kojim uslovima treba ponovo pokrenuti transakciju i kada. Zaključak do koga su došli je da ne postoji odgovor koji odgovara svim aplikacijama. Neka tehnika će postići odlične performanse za neke aplikacije, dok će za druge postići loše. Oni predlažu da se parametri dinamički menjaju prema karakteristikama aplikacije korišćenjem mašinskog učenja, preciznije učenja sa podrškom (eng. *reinforced learning*). Koliko puta treba da se ponovi transakcija, koja je prekidana zbog prekoračenja, se procenjuje metodom procene gornje granice poverenja (eng. *upper confidence bounds*) [89], dok broj ponavljanja svih vrsta poništavanja se optimizuje metodom istraživanje gradijentnog spuštanja (eng. *gradient descent exploration*).

Slična metoda je predložena u [90]. Razlikuju se u načinima izvršavanja transakcija, a za procenu koji način izvršavanja treba koristiti se koristi samo tehnika gornje granice poverenja [89]. Mogući načini izvršavanja transakcije su: 1) ne detektuju se konflikti, ali samo su dozvoljena čitanja iz memorije, 2) za transakciju se čuva samo radni skup sa adresama upisa, ali ne i čitanja i 3) normalne transakcije. Prve dve vrste transakcije ne garantuju logičko ispravno izvršavanje. Zato autori predlažu da se pamti njihov skup čitanja softverski i da se prilikom potvrđivanja proverava da li je došlo do konflikta. Prilikom potvrđivanja uključuje se hardverski mehanizam praćenja konflikata i svi podaci koji su se čitali se ponovo čitaju. Prvo se pokuša sa izvršavanjem normalnih transakcija, ako to dovodi do poništavanja, prelazi se na relaksiranije varijante izvršavanja transakcija i na kraju kad se se iscrpe sve mogućnosti prelazi se na zaključavanje sa globalnom bravom.

Procena koliko puta treba da se ponavlja transakcija data je u [10]. Procena se vrši na osnovu prethodne procene i na osnovu razlike u dužini trajanja prethodnih izvršavanja transakcija. Dužina jednog izvršavanja uključuje sva poništena izvršavanja i poslednje uspešno izvršavanje. Traži se minimalna dužina tog trajanja pomoću metode istraživanje gradijentnog spuštanja. Pošto taj algoritam može da pronađe lokalni minimum i ne može više da se koristi za pretragu, autori predlažu korišćenje algoritma impulsa (eng. *momentum algorithm*) [91].

Autori u [92] predlažu da se transakcije rasporede u redove. Svaki red sadrži transakcije koje mogu međusobno da se poništavaju. Pušta se po jedna transakcija iz svakog reda. Kada se završi transakcija iz jednog reda pušta se transakcija koja čeka u tom redu. Broj redova se menja dinamički u toku izvršavanja na osnovu broja poništavanja i potvrđivanja transakcija. Koristi se algoritam planinarenja. Raspoređivanje u redove se radi na osnovu nekog objekta koji se pridružuje transakciji. To može biti nešto što programer podešava ili adresa nekog podatka kome transakcija pristupa.

Još jedno rešenje sa pomoćnim bravama je prikazano u [93]. Predlaže se korišćenje tehnike analize struktura podataka [94] da se pronađu oni podaci u transakcijama za koje postoji velika verovatnoća da će biti deljeni. Programski prevodilac pronalazi takve podatke i pre njih ubacuje zaključavanje pomoćne brave. Zaključavanje je uslovno. U toku izvršavanja se prati koji podaci uzrokuju konflikte i adrese instrukcija koje su pristupale tim podacima (hardver mora da ima mogućnost da dostavi te podatke prilikom poništavanja transakcije). U zavisnosti od istorije, odlučuje se da li će se pomoćna brava aktivirati. Ako se ponavlja samo adresa podatka, a ne i adresa instrukcije, onda se zaključavanje radi uvek kada se naiđe na tu adresu pristupa, bez obzira na instrukciju koja se izvršava. Postoji i obrnuta varijanta. Ako se obe stvari ponavljaju, onda se zaključava brava samo na određenoj instrukciji kada se pristupa određenoj adresi. Ako nema ponavljanja adresa, onda se brava ne zaključava. Brava se zaključava korišćenjem ne transakcionih instrukcija.

Primećeno je da se javljaju ciklična poništavanja kada transakcije pristupaju više različitih podataka, kojima pristupa mnogo niti [95]. Autori predlažu predviđanje koji podaci to mogu da izazovu. Predviđanje se radi u hardveru. Prediktor označi pristup tim podacima kada im neka transakcija pristupi i onda sve druge transakcije koje hoće da pristupe istim podacima moraju da čekaju dok prva transakcija ne završi pristup. Hardver prilikom izvršavanja transakcije za adrese koje su najskorije izazvale konflikte pamti gde je prva instrukcija u transakciji koja joj pristupa i gde poslednja. Brojač se koristi da se prati koliko ta adresa je puta izazvala konflikt. Kada se naiđe na instrukciju koja prva

pristupa adresi koja izaziva konflikte, transakcija izaziva prekid i sve ostale transakcije koje hoće da pristupe istoj adresi se odlažu. Prilikom konflikta koriste se i vremenske marke početka transakcije da se utvrdi da li je ciklično poništavanje u pitanju. Ako jeste, povećava se šansa da će doći do odlaganja transakcija za adresu koja je izazvala konflikt.

Autori u [96] predlažu kreiranje posebnih redova za svaki objekat kome se pristupa u transakciji. Kada transakcija treba da se izvrši, ubacuje se u sve redove koji su kreirani za podatke kojima pristupa. Kada se nađe na prvom mestu u svim redovima može da počne svoje izvršavanje. Na taj način se želi postići da se izvršavaju samo one transakcije koje međusobno neće ući u konflikt. Procena kojim podacima se pristupa radi se na osnovu istorije, tako što se čitaju radni skupovi (Blumovi filtri) prilikom svakog završetka transakcije. Dodatno se predlaže da se transakcije, koje ne mogu da se izvrše pomoću hardverske transakcione memorije, izvršavaju pomoću softverske. I softverske transakcije podležu istom mehanizmu raspoređivanja.

Osnovna ideja predloženog rešenja u [97], [98] je da se napravi raspoređivač koji će raditi na postojećim komercijalno dostupnim procesorima koji prilikom prekida transakcije ne dostavljaju mnogo informacija o razlogu prekida (samo koja transakcija je prekinuta i da li zbog konflikta ili prekoračenja). Raspoređivač koristi te informacije da napravi statistički model šablona konflikata. Predloženo rešenje pamti dve matrice. Jedna matrica služi za pamćenje koje transakcije su bile aktivne kada se neka transakcija potvrdila, dok druga služi za pamćenje koje transakcije su bile aktivne kad se neka poništila. Pamti se broj slučajeva koji je uočen. Na osnovu tih matrica konstruišu se verovatnoće (npr. deljenjem broja poništavanja sa ukupnim brojem izvršavanja transakcija) za sledeće događaje. Prvi događaj je da se jedna transakcija poništava dok je druga transakcija aktivna. Drugi događaj je kada se druga transakcija izvršava sa prvom, a prva se poništi. Kada su verovatnoće tih događaja preko određenih granica, transakcija zaključava pomoćnu bravu. Granice su promenljive i za traženje dobre granice koristi se stohastički algoritam planinarenja. Pored sprečavanja konflikata u rešenju je predloženo da se sprečava i promena niti na jezgru kada se dešavaju poništavanja transakcija, jer autori tvrde da keš memorije koje imaju moderni procesori ne mogu da pruže dovoljno prostora za izvršavanje transakcija.

Rešenje prikazano u [99], slično kao prethodno, koristi matricu za predviđanje konflikata, ali samo onu gde se čuva informacija koje su se transakcije izvršavale kada se neka transakcija poništila. Rešenje je implementirano u hardveru. Kada se proceni da transakcija koja se startuje ima veliku šansu da se poništi jer se neka druga transakcija izvršava, može ili da se odloži ili da se toj niti oduzme procesor. Ako se transakcija koja se izvršava, prethodnih nekoliko puta dugo izvršavala, onda se niti oduzima procesor. Pored toga za svaku transakciju se pamti kojim adresama je pristupala. Ako se neka transakcija odložila zbog velike šanse za konflikt, kada se izvršava, proverava se da li se adrese kojima pristupa poklapaju sa adresama kojima je pristupala transakcija zbog koje je prva bila odložena. Ako se poklapaju, znači da šansa za konflikt se ne smanjuje. U suprotnom procena šanse za konflikt se smanjuje.

3.1.4 Komercijalni procesori

Iako je istraživanje transakcione memorije generisalo mnogo zanimljivih ideja, komercijalni procesori koji su napravljeni podržavaju samo najosnovnije mehanizme transakcione memorije [85], [100], [101]. Transakcija može da se izvršava dok postoje resursi za čuvanje njenog radnog skupa i neponištive operacije nisu dozvoljene unutar transakcije. Razlozi za poništavanje transakcije su dostupni. Može se utvrditi da li je bilo konflikta ili prekoračenja. Varijanta odlučivanja, koja će transakcija biti poništena, je uvek ona u kojoj će biti poništen primalac. Radni skup se čuva u keš memoriji prvog nivoa. Protokol keš koherencije se koristi za detektovanje konflikta. Ne postoje garancije da će se transakcija izvršiti uspešno, nema zaštite od izgladnjivanja i živog blokiranja. Programer mora dodati programski kod za odluku šta raditi kada se transakcija poništi. Postoji podrška za već napisane programe sa zaključavanjem brava, da se brave ne zaključaju i da se kritična sekcija izvrši kao

transakcija.

Procesor prikazan u [102] ima naprednije mogućnosti. Moguće je pauzirati i nastaviti transakciju. Postoje transakcije koje ne čuvaju radni skup za podatke koji se samo čitaju. Dodata je podrška za izvršavanje instrukcija korak po korak (eng. *debugging*).

Jedan od prvih procesora koji je trebao da podrži osnovnu varijantu transakcione memorije prikazan je u [103]. Međutim, razvoj procesora je obustavljen i nikada nije bio komercijalno dostupan.

3.2 Postojeće tehnike za ubrzavanje aplikacija na asimetričnim višejezgarnim procesorima

U otvorenoj literaturi postoji veliki broj radova na temu asimetričnih procesora. Veliki deo tih radova pokriva procesore koji se razmatraju u ovoj tezi, asimetrične višejezgarne procesore sa jednim instrukcijskim skupom. Organizacija takvih procesora je istražena i nema velikog prostora za poboljšanje. Ono što je otvorena oblast je raspoređivanje aplikacija ili dela aplikacije na različite tipove jezgra. Pošto se to razmatra u ovoj disertaciji, prikazana su postojeća rešenja iz te oblasti. Detaljan pregled svih rešenja za asimetrične višejezgarne procesore može se naći u [4].

Raspoređivanje niti se radi zbog optimizacije nekog ili nekih parametara. Mogući parametri su brzina izvršavanja pojedinih aplikacija, propusnost sistema i smanjenje potrošnje energije. Smanjene potrošnje energije izlazi van okvira ove disertacije i rešenja koja to razmatraju nisu navedena (osim ako imaju mogućnost i za optimizaciju drugih parametara). Raspoređivanje može da bude fer ili da favorizuje neke procese/niti. Migracijom niti ili dela niti na veliko jezgro se postiže optimizacija parametara, jer izvršavanje na određenom tipu jezgra pogoduje poboljšanju parametara. Ona se može raditi na nivou fine granulacije ili na nivou grube granulacije. Prva opcija omogućava migriranje dela niti i obično je implementirana u hardveru, dok druga opcija premešta nit i obično je implementirana u softveru. Rešenja su uglavnom predložena za konvencionalne aplikacije, ali ima nekih rešenja koja su predložena za aplikacije koje se sastoje od paralelnih poslova (eng. *task parallel*). U tabeli 3.3 su grupisana rešenja po navedenim opcijama. Navedena su rešenja (kolona R), oznaka da li se optimizuju performanse aplikacije (kolona A) ili propusnost (kolona P) ili se raspoređivač ponaša fer (kolona F). Navedeno je da li je raspoređivač realizovan u hardveru ili softveru i da li je namenjen aplikacijama sa nitima ili poslovima.

Tabela 3.3: *Pregled raspoređivača za asimetrične višejezgarne procesore*

	Softver				Hardver			
	R	A	P	F	R	A	P	F
Niti	[15]	•			[5] [13]	•		
	[104]	•	•		[14]	•	•	
	[7] [8] [9] [10]		•	•	[11]		•	•
	[6]	•	•	•	[12]	•		•
Poslovi	[105] [106] [107]	•			[108]	•	•	

Autori u [5] primećuju da kritične sekcije predstavljaju serijalizovani deo aplikacije koji može da smanji broj niti koje se uporedo izvršavaju i na taj način štete performansama izvršavanja (što je i kasnije pokazano u [34]). Predlaže se da se kritične sekcije migriraju hardverski na veliko jezgro. Na taj način se ubrzavaju, druge niti koje čekaju na bravama tih kritičnih sekcija, čekaju kraće i aplikacija se brže izvrši. Ne migriraju se sve transakcije, već samo one koje se nalaze na kritičnom putu (one čije ubrzanje stvarno ubrzava aplikaciju). Kada kritična sekcija počne da se izvršava na jezgru, proverava se da li neke druge kritične sekcije čekaju da se izvršavaju na jezgru (imaju drugu

bravu). Ako se takav slučaj događa, znači da ta transakcija nije na kritičnom putu, jer druge niti ne pokušavaju da je izvrše. To se prati pomoću jednog brojača koji se inkrementira kada se naiđe na takav slučaj, u suprotnom se dekrementira. Kada brojač dostigne limit, blokira se dalje izvršavanje kritične sekcije na velikom jezgru i kritična sekcija se na dalje izvršava na malom jezgru. Posle nekoliko miliona instrukcija dozvoljava se ponovo migriranje kritične sekcije da bi se proverilo da se nije promenila faza programa i tada je možda kritična sekcija počela da izaziva usporenje drugih niti. U jezgra se dodaje mehanizam za migraciju transakcija. Takođe, potrebno je dodati u instrukcijski skup instrukcije za početak i završetak kritične sekcije.

Pored kritičnih sekcija autori u [13] razmatraju i druge faktore koji mogu da uspore niti. Zaključuju da niti mogu biti usporene događajima kao što su promašaji u keš memoriji. To može dovesti da neka nit zaostaje za drugim. U aplikacijama se često javlja slučaj da postoji sinhronizacija niti na barijeri i onda sve druge moraju da čekaju najsporiju nit. Takođe, ako postoji protočna, paralelna obrada, niti čekaju na najsporiju nit da bi započele sledeću fazu. Pored promena hardvera, koja su preuzete iz prethodnog rešenja, potrebna je i nova instrukcija za čekanje da se završi neki deo niti gde se radi sinhronizacija (kritična sekcija, barijera, ...). Hardver meri koliko dugo se čeka na nekom mestu. Oni delovi niti na koje se najviše čeka se ubrzavaju.

Matematički model koji obuhvata prethodna dva rešenja je predložen u [14] i dodatno razmatra povećanje propusnosti sistema sa dve aplikacije. Daje se model za procenu koristi od ubrzavanja nekog dela niti. To je odnos koliko se kraće izvršavaju aplikacije sa ubrzanjem i dužina trajanja aplikacija kada nema ubrzavanja. Taj odnos je proizvod tri komponente ubrzanje samog dela niti koji se migrira, ubrzanje cele aplikacije zbog tog dela niti i ubrzanje svih aplikacija zbog tog dela niti. Procena prve komponente se radi na osnovu procene performansi velikog jezgra. Druga se procenjuje na osnovu odnosa dužine izvršavanja bez čekanja u nekom prethodnom periodu i ukupne dužine tog perioda. Treća komponenta se određuju posebno za svaki tip dela niti (kritična sekcija ili spora nit). Za spore niti to je recipročna vrednost broja svih sporih niti, a za kritične sekcije je odnos broja niti koje su čekale na kritičnu sekciju u nekom prethodnom periodu i ukupnog broja niti. Kada se odrede delovi niti sa najvećom vrednosti koristi, ti delovi se migriraju.

Rešenje koje izbegava promenu instrukcijskog skupa u odnosu na prethodna rešenja dato je u [12]. Hardverski raspoređivač prati u kom modu se izvršava program na jezgru i detektuje promene iz korisničkog u sistemski mod i obrnuto. Na osnovu toga se zaključuje gde počinje i gde se završava kritična sekcija. Pored ubrzavanja same aplikacije, raspoređivač se trudi da bude fer tako što ne dozvoljava ni jednoj niti da se izvršava na velikom jezgru više od unapred određenog vremena.

Autori u [15] predlažu migriranje na veliko jezgro one niti koja je najdalje od barijere (najsporija nit). Prilikom startovanja niti ili prolaska kroz barijeru, svim nitima se dodeljuje ista procena trajanja naredne faze (do sledeće barijere). Za procenu se uzima veliki broj (da nikad ne bi bio smanjen ispod nule). Kako se nit izvršava vrednost te procene se smanjuje. Smanjuje se na osnovu informacija koje procesor daje u toku izvršavanja (npr. broj izvršenih instrukcija). Raspoređivanje se radi u softveru, tj. raspoređivač operativnog sistema odlučuje koja nit će biti migrirana na veliko jezgro.

Softverski raspoređivač predložen je u [104]. Osnovna ideja je da niti, koje najviše vremena koriste za komunikaciju sa memorijom, se rasporede na mala jezgra, dok niti koje najviše vremena koriste resurse jezgra za računanje se izvršavaju na velikom jezgru. Procena koliko se pristupa memoriji se radi na osnovu izvršavanja na nekom jezgru. Dok se nit izvršava na malom jezgru računa se prosek promašaja u keš memoriji najvišeg nivoa po instrukciji. Taj broj se pomnoži sa veličinom bafera koji kontroliše da se instrukcije potvrde u programskom redosledu kod velikog jezgra. Kad se nit izvršava na velikom jezgru, procena se radi opet na osnovu proseka promašaja, ali se taj broj množi sa prosečnim brojem instrukcija koje se izvrše između promašaja i trenutka kada je instrukcija koja je izazvala promašaj dobila traženi podatak. Procena koliko vremena provede koristeći resurse jezgra za računanje radi se tako što se procenjuje koliko instrukcija može paralelno da se izvršava. Procene se rade različito na velikom i malom jezgru. Na velikom jezgru se računa verovatnoća da ne postoje zavisnosti po podacima između dve, tri i četiri uzastopne instrukcije. Na malom jezgru se pretpostavlja

da će moći da se izvršava paralelno onoliko instrukcija koliko veliko jezgro može da pokrene u jednom ciklusu takta. Na početku se niti rasporede na standardni način. Računaju se procene korišćenja resursa za računanje. Kada dođe vreme za sledeće raspoređivanje, proverava se da li su niti optimalno raspoređene na osnovu procene. Ako nisu, vrši se migracija niti.

Fer raspoređivač je prikazan u [109]. Predložen je matematički model za procenu koliko je fer bilo raspoređivanje. Procenjuje se na osnovu koeficijenta varijacije usporenja koje postiže nit koja se izvršava na asimetričnom višejezgarnom procesoru u odnosu kad se izvršava na velikom jezgru. Težnja da se poveća taj parametar prouzrokuje je da se raspoređivanje vrši na osnovu podjednakog napredovanja niti. Izračunavanje koliko vremena joj treba da se izvrši na asimetričnom procesoru je trivijalno, samo se meri vreme dok se nit izvršava. Međutim, teže je izračunati vreme izvršavanja kad se nit izvršava na velikom jezgru. Izračunavanje se radi na osnovu ukupnog vremena izvršavanja na asimetričnom procesoru i vremena izvršavanja na malom jezgru u odnosu na veliko. Da se ne bi ugrozila propusnost sistema, raspoređivanje se radi bez uzimanje u razmatranje procene koliko je fer raspoređivanje, sve dok ta procena ne padne ispod granice.

Softverski raspoređivač je prikazan u [7] čiji je rad baziran na tehnikama mašinskog učenja. Na osnovu tragova izvršavanja, koji uključuju i monitore performansi procesora, se generiše model. Model na osnovu monitora performansi procenjuje koliko će biti vreme izvršavanja na velikom i malom jezgru. Raspoređivač, dok je broj niti manji ili jednak broju velikih jezgara, raspoređuje niti samo na velika jezgra. Kada se taj broj poveća, na velika jezgra se raspoređuju niti koje su se najbrže izvršavale na velikom jezgru. Ako se nit izvršavala samo na malom jezgru, onda se za nju uzima procena brzine izvršavanja na velikom jezgru.

Autori u [8] predlažu da se raspoređivanje radi na osnovu procene koliko je nit napredovala. Niti koje su manje napredovale treba premestiti na veliko jezgro. Na taj način se povećava propusnost sistema. Napredovanje na malom jezgru je manje. Da bi moglo da se uporedi napredovanje na malom i velikom jezgru, poveća se procena napredovanja na malom jezgru za razliku u brzina velikog i malog jezgra. Prilikom povećanja procene napredovanja dodatno se koristi prioritet niti i aplikacije. Na taj način se radi fer raspoređivanje uz poštovanje prioriteta. Procena koliko je nit napredovala na velikom jezgru se radi na sličan način kao u prethodnom rešenju.

Unapređenje prethodnog rešenja je dato u [9]. U prethodnom rešenju usporenje napredovanja na velikom jezgru je smatrano konstantno jedan (normalizacija se radila samo za niti na malom jezgru). Ovde autori predlažu da se u računicu uključi i usporenje na velikom jezgru zbog promašaja u keš memorijama, zbog čekanja na bravama, itd. Da bi se to izračunalo pamti se broj instrukcija koje se izvrše u jednom ciklusu na velikom jezgru u trenucima kada nema zagušenja u sistemu. Usporenje se računa kao odnos trenutnog broja instrukcija koje se izvrše u jednom ciklusu i zapamćene vrednosti. Trenutak kada nema zagušenja, procenjuje se na osnovu stope kojom se prenose podaci između procesora i keš memorije.

Nekoliko algoritama za raspoređivanje se analizira u [10]. Prvi algoritam je raspoređivanje niti po fer principu dok se ne postigne minimalna granica fer pristupa, nakon toga se raspoređivanje radi da se poveća propusnost sistema. Drugi algoritam je raspoređivanje na fer principu neko vreme, nakon toga se raspoređivanje radi da se poveća propusnost sistema i tako u krug. Treći algoritam raspoređuje niti tako da se poveća propusnost sistema ali kao da ima više velikih jezgara nego što ih stvarno ima. Četvrti algoritam kombinuje sva tri prethodna. Raspoređivanje po fer principu i radi povećanja propusnosti sistema se radi na sličan način kao u prethodnim rešenjima. Autori su zaključili da se može postići minimalna granica fer pristupa uz neznatno smanjenje propusnosti.

Autori u [6] predlažu softverski raspoređivač koji radi na osnovu sva tri faktora: performansi pojedinačnih aplikacija, propusnosti i fer principa (uzima u obzir i potrošnju energije). Za procenu performansi koriste se tehnike mašinskog učenja, slično kao u [7]. Za fer pristup koristi se postojeći mehanizam Linux kernela, samo što se vreme izvršavanja normalizuje za izvršavanje na malom jezgru. Raspoređivanje se radi na sledeći način. Niti se podele u tri grupe: jedna grupa su one niti koje će se izvršavati na velikom jezgru, druga se koje će se izvršavati na malom, a treća su koje će

se izvršavati na bilo kom jezgru. Ta podela se radi na osnovu procenjenih performansi izvršavanja. Niti koje će se najviše ubrzati na velikom jezgru ili koje prouzrokuju najviše drugih niti da čekaju idu u prvu grupu, one koje imaju suprotne odlike idu u drugu grupu, a koje ostanu idu u treću grupu. Raspoređivanje unutar grupe se radi po fer principu. Ukoliko je red za čekanje nekog velikog jezgra prazan, može se migrirati nit sa manjeg jezgra po fer principu.

Raspoređivanje za aplikacije sa paralelnim poslovima predloženo je u [105]. Pokušava se skoro optimalna raspodela poslova na jezgra. Predlaže se da se na osnovu dužine posla odredi gde će se koji posao izvršavati. Dužina posla se određuje na osnovu broja ciklusa koji su potrebni da se izvrši. Procena se radi na osnovu prethodnih izvršavanja i normalizuje se kao da se izvršavao na velikom jezgru. Poslovi se raspoređuju u redove i svaki tip jezgara ima svoje redove. Kada neki tip jezgara isprazni svoje redove, on pokušava da ukrade poslove od drugog tipa jezgra.

Autori u [106] predlažu kreiranje grafa poslova. Prilikom pokretanja poslova zadaje se koji poslovi moraju da se urade pre njega. Raspoređivač pronalazi kritičnu putanju u grafu (najdužu putanju). One poslove koji se nalaze na kritičnoj putanji raspoređuje na velika jezgra, dok ostale raspoređuje na mala jezgra. Graf se rekonstruiše svaki put kada se završi neki posao ili kada se napravi nov. Algoritam ne uzima u razmatranje dužinu posla. Ukoliko veliko jezgro nema spremnog posla, preotima poslove malom jezgru. Sličnu tehniku koriste i autori u [107], ali uzimaju u razmatranje i dužinu posla (i potrošenu energije), pa se dužina putanje ne računa na osnovu broja poslova na njoj, već na osnovu dužine tih poslova.

Korišćenje dinamičke i statičke tehnike za postizanje asimetrije predloženo je u [108]. Dinamička tehnika je povećanje i snižavanje napona na pojedinim jezgrima, dok je statička tehnika ista kao i u prethodnim rešenjima (fiksan broj malih i velikih jezgara). Raspoređivač je implementiran u hardveru dok se preko posebnih instrukcija daje informacija o trenutno spremnim poslovima. Detektuju se dve faze izvršavanja. Prva faza je faza visokog paralelizma kada ima više poslova nego jezgara. U toj fazi se povećava napon malih jezgara, a smanjuje napon velikih. Na taj način se prva ubrzavaju, a druga usporavaju. Time se postiže veća propusnost sistema. Druga faza je faza niskog paralelizma kada ima više jezgara nego poslova. Tada se napon povećava na velikim jezgrima, a smanjuje na malim, da bi se ubrzalo izvršavanje poslova. Takođe, raspoređivač podržava i mogućnost da veliko jezgro otima poslove malim jezgrima, kad je nivo paralelizma nizak.

3.3 Tehnike za podelu keš memorije i korišćenje keš memorija sa drugim tehnikama

Podsistem keš memorija, kao podsistem koji najviše utiče na performanse procesora, izuzetno je prisutan u otvorenoj literaturi (ima preko milion radova gde se spominju keš memorije[1]). Pregled tolike literature nije moguće obraditi u okviru jedne disertacije. Iz tog razloga ovde su navedeni samo neki radovi, koji su osnov i koji su najsličniji predloženom rešenju. Ostali radovi koji su deo te grupe su već detaljno opisani u [1], [2], [110]. Takođe, dat je pregled radova koji istovremeno analiziraju podsistem keš memorija i asimetrične procesore, kao i oni koji istovremeno analiziraju podsistem keš memorija i transakcionu memoriju.

Podeljenje keš memorije predložene su u [111], [112]. Ideja je da se podaci, kojima se pristupa sa različitom lokalnošću, smeštaju u različite delove keš memorije. Na taj način, svaki deo keš memorije može da se optimizuje za određeni način pristupa. Tako može da se smanji vreme pristupa keš memoriji i da se poveća procenat pogodaka. Prvi rad ([111]) predlaže da delovi budu iste veličine. Podaci će biti smešteni u deo keš memorije koja se određuje na osnovu niza adresa kojima se pristupalo u poslednje vreme. Ako se uzastopne adrese iz tog niza razlikuju za slične vrednosti, onda ti podaci idu u isti deo keš memorije, a ostali u drugi deo keš memorije. Ako je ta razlika ogromna predlaže se

[1] Izvor Google Scholar `https://scholar.google.com/` (Pristupljeno 14.06.2021. godine)

da se podaci uopšte ne smeštaju u keš memoriju, jer je mala verovatnoća da će im se opet pristupiti i nema potrebe da zauzimaju mesto u keš memoriji. Drugi rad ([112]) predlaže da jedan deo bude manje veličine i da se na taj način uštedi i prostor na čipu. Postupak koji se koristi u tom radu je da se odredi kako se podacima pristupa u jednom bloku. Ako se pristupa samo nekim delovima bloka onda se ti delovi bloka smeštaju u jedan deo keš memorije, a ako se pristupa skoro svim delovima bloka onda se taj blok smešta u drugi deo keš memorije. Rešenje je namenjeno za poboljšanje performansi jednoprocesorskog sistema.

Poboljšanje performansi simetričnog višejezgarnog procesora pokušano je u [113], [114] korišćenjem podeljenih keš memorija. Podeljen je samo prvi nivo keš memorija. Ostali nivoi sadrže konvencionalne keš memorije. Konvencionalne keš memorije učestvuju u protokolu keš koherencije. Primećeno je da postoje blokovi koji imaju visoku učestanost pristupa, nezavisno od lokalnosti pristupa. Odlučeno je da se ti blokovi smeštaju u mali deo keš memorije. Na taj način oni blokovi kojima se najčešće pristupa biće u keš memoriji sa kratkim vremenom pristupa, što će ubrzati izvršavanje. Da bi se odredila frekvencija pristupa za svaki blok u keš memoriji se koristi po jedan brojač. Pristup tom bloku ujedno i povećava vrednost tog brojača. Oni koji imaju najveći broj pristupa, smeštaju se mali u deo keš memorije. Autori predlažu smanjenje vrednosti brojača periodično da bi se uočile promene u frekvenciji pristupa blokovima.

Prva grupa rešenja predlaže logičku podelu keš memorija na osnovu benefita koji će imati aplikacija koja ga koristi. U [115] se predlaže podela deljenih keš memorija tako što svaka aplikacija dobije onoliko više prostora u keš memoriji kolika je verovatnoća da će taj dodatni prostor smanjiti broj promašaja. Podela nije fizička (nema odvojenih delova keš memorije), već logička. To se postiže tako što se menja algoritam zamene podataka u keš memoriji. Za svaki ulaz u keš memoriju se pamti kom jezgru pripada. Svakom jezgru se dodeljuje određeni broj ulaza. Kada se dogodi promašaj za zamenu se odabira blok koji pripada nekom drugom jezgru, jer je jezgru dodeljeno manje ulaza nego što mu je potrebno. U suprotnom, za zamenu su bira neki blok koji pripada tom jezgru. Sa druge strane u [116] se predlaže fizička podela, gde bi jedna grupa jezgara imala jednu veličinu keš memorije poslednjeg nivoa, dok bi druga grupa jezgara imala drugu veličinu. Svaka keš memorija ima dodatni prostor za tagove, koji nisu povezani sa podacima. Ti tagovi su organizovani kao da pripadaju keš memoriji druge veličine. Pomoću njih se vrši procena kakve bi performanse postigla aplikacija na drugom jezgru. Te informacije su dostupne operativnom sistemu preko posebnih instrukcija i raspoređivač operativnog sistema donosi odluku gde da rasporedi aplikacije.

Druga grupa rešenja razmatra podelu keš memorija tako da jedan deo keš memorije bude napravljen konvencionalnom tehnologijom, dok se drugi pravi pomoću perzistentih memorija (eng. *non-volatile memory*). Perzistente memorije zauzimaju manje mesta, troše manje energije za čitanje, više za upis, imaju malo lošije vreme čitanja, ali red veličine duže vreme upisa [117]. U [117] se predlaže da se keš memorija najvišeg nivoa sastoji od malog dela napravljenog na konvencionalan način i drugog velikog dela, napravljenog pomoću perzistentne memorije. Ako se blok traži za čitanje, on se smešta u veliki deo. U suprotnom, smešta se u mali. Ako se promeni razlog pristupa i taj novi razlog se ponovi nekoliko puta, blok se premešta u drugi deo. Autori u [118] primećuju da u višejezgarnom procesoru keš memorije drugog nivoa čuvaju privatne podatke niti i da se oni ne menjaju često. Razlog za to je što se upisuju u keš memoriju drugog nivoa samo kad se zamenjuje u keš memoriji drugog nivoa, a nema nikad poništavanja zato što im ne pristupa druga nit. Zato predlažu da se keš memorije drugog nivoa podele na veći deo napravljen od perzistentne memorije i mali deo napravljen na konvencionalan način.

Autori u [119] su razmatrali uticaj veličine keš memorije u asimetričnim višejezgarnim procesorima. U radu asimetrija se postiže samo različitom veličinom keš memorija. Zaključili su da postoje konfiguracije asimetričnih procesora koje imaju manje keš memorije, a postižu bolje performanse izvršavanja nego simetrični procesori sa više keš memorije.

Autori u [120] su razmatrali implementaciju podsistema keš memorija i zaključili su da je potrebno više truda da se implementira na asimetričnom procesoru nego na simetričnom. Asimetrični

procesori imaju različite keš memorije za različita jezgra, pa je potrebno za svaki tip keš memorije napisati različiti kod u jeziku za dizajn hardvera, svaki tip treba posebno verifikovati i međusobne interakcije između različitih tipova keš memorija treba verifikovati. Zato su predložili alat koji generiše kod za ceo podsistem keš memorija napisan u jeziku za dizajn hardvera. Ulazni podaci su svi parametri keš memorija (veličina, tip mapiranja, širina bloka, itd.).

Postoji nekoliko predloga za smanjenje lažnih konflikata korišćenjem prilagođenog podsistema keš memorija. Jedan predlog je opisan u [121]. Osnovna ideja je da se konflikti proveravaju na finom nivou granularnosti. Svaki blok u keš memoriji deli se na delove (logička podela). Svakom delu dodaju se dva bita koja označavaju kako je tom delu keš memorije pristupano. Deo bloka može biti u četiri stanja: 1) transakcija mu nije pristupala i nije menjan, 2) transakcija mu nije pristupala, ali je menjan, 3) pročitan je u transakciji, i 4) promenjen je u transakciji. Na nivou tih delova se detektuju konflikti i zbog postojanja stanja 2) može da se očuva jaka atomičnost transakcije.

Autori u [122] istražuju koliko su česti lažni konflikti i zaključuju da je, u prosečnom slučaju, oko 27% konflikata lažno. Takođe, predlažu da se za neke blokove u keš memoriji koriste dva Blumova filtra, jedan za čuvanje adresa podataka koji su pročitani u transakciji, drugi za čuvanje adrese podataka koji su upisani tokom transakcije. Adrese se čuvaju na finom nivou granularnosti. Blokovi koji imaju filtre ne izazivaju lažne konflikte (osim ako su izazvani zbog prirode Blumovog filtra).

3.4 Razlike između predloženih i postojećih rešenja

Predloženo rešenje, u ovoj disertaciji, za migraciju transakcija je prvo rešenje koje analizira izvršavanje aplikacija na asimetričnom višejezgarnom procesoru sa transakcionom memorijom. Pokušaj pronalaska, u otvorenoj literaturi, da se pronađe rad sa istim ili sličnim rešenjem je završen bezuspešno, tj. rad nije pronađen. Predloženom rešenje se smatra da spada u domen raspoređivanja, za koji su prethodno navedena postojeća rešenja i za asimetrične višejezgrarne procesore i za transakcionu memoriju. Raspoređivanje u predloženom rešenju se radi na finom nivou granulacije. Osmišljeno je isključivo za implementaciju u hardver. Rešenje ima za cilj da pobljša performanse izvršavanja jedne aplikacije. Nije razmatrana propusnost sistema i fer politika raspoređivanja. Neka od postojećih rešenja, pomenutih u ovom radu, koja optimizuju i to, mogu da se koriste sa predloženim rešenjem, skoro bez bilo kakvih promena. Bilo koji način za implementaciju transakcione memorije, može da se koristi sa predloženim rešenjem. Za opis i evaulaciju rešenja korišćena je verzija transakcione memorije koja je nalik verzijama dostupnim u komercijalnim procesorima. Isto važi i za implementaciju asimetričnog višejezgarnog procesora. Jedina razlika u odnosu na komercijalna rešenja je to što je korišćen asimetrični višejezgarni procesor sa jednim velikim jezgrom.

Predloženo rešenje, u ovoj disertaciji, za prilagođenje podsistema keš memomorija se oslanja na već poznata jednoprocesorska rešenja. Razlika je što se ta ideja koristi u višejezgarnom procesoru sa transakcionom memorijom. Pored ubrzavanja rada pojedinačnih jezgara i smanjenja veličina keš memorije, cilj predloženog rešenja je i smanjenje lažnih konflikata, koje dovodi do nepotrebnog poništavanja transakcije, a time i degradacije performansi rada procesora. U odnosu na rešenja za višejezgarne procesore za podelu keš memorija, predloženo rešenje se razlikuje u načinu podele, broju nivoa koji sadrže podeljene keš mempmorije i protokolu keš koherencije koji se koristi. Razmotrena je i asimetrija jezgara u procesoru i transakciona memorija, što nije razmatrano u dosadašnjim rešenjima za višejezgarne procesore. Dosadašnja rešenja za rešavanje lažnih konflikata u sistemima sa transakcionom memorijom su slična predloženom rešenju, ali za njihovu implementaciju je potrebno povećanje keš memorije, dok za predloženo rešenje to nije potrebno. Ostala rešenja, koja razmatraju organizaciju podsistema keš memorija u sistemu sa asimetričnim jezgrima, nemaju dodirnih tačaka sa predloženim rešenjem u ovoj disertaciji.

Poglavlje 4

Postavka problema

Cilj ove disertacije je poboljšanje performansi procesora kombinacijom postojećih tehnika za dizajn procesora uz potrebne modifikacije. U ovom poglavlju predstavljena su dva problema koji se javljaju prilikom dizajna višejezgarnih procesora. Svaki od njih je opisan u posebnoj sekciji. Takođe, za svaki problem razmatrano je na koji način mogu da se reše i koje su odluke u dizajnu koje treba razmatrati prilikom rešavanja problema.

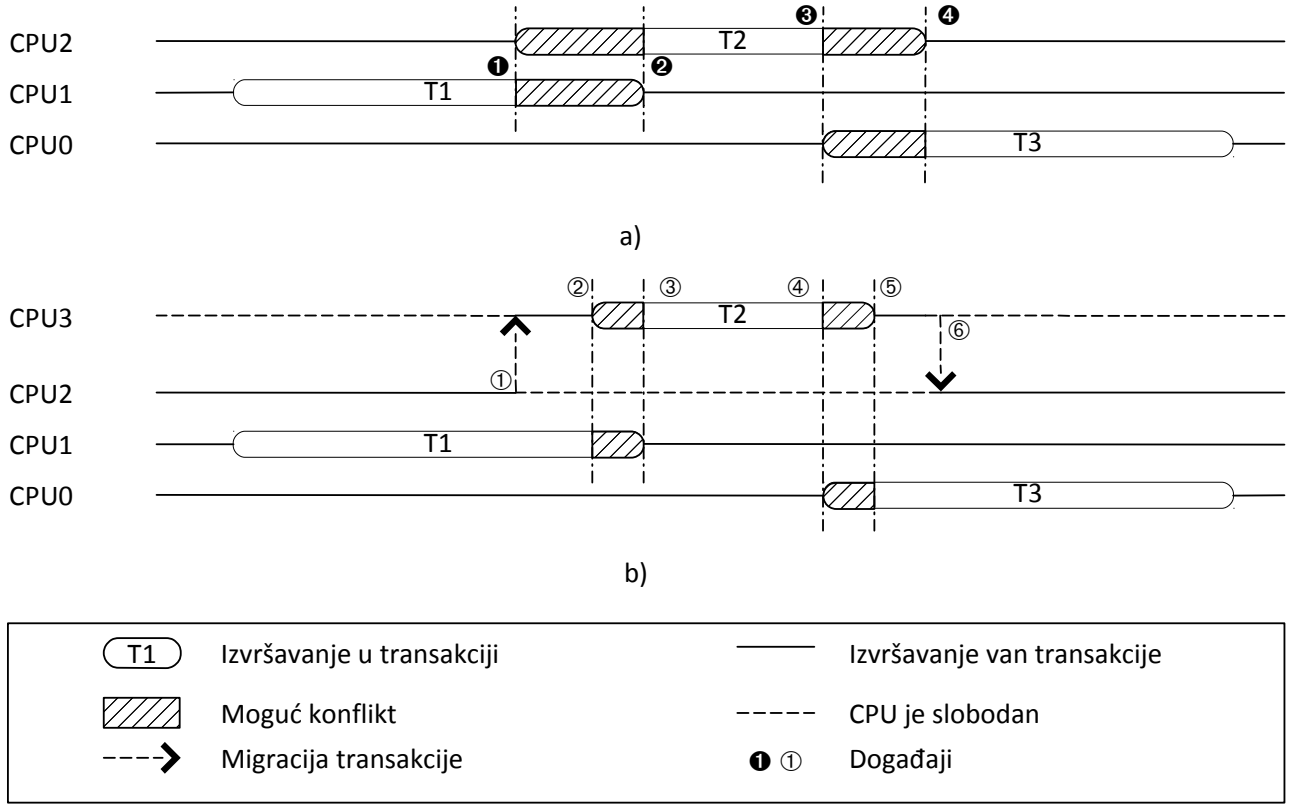
4.1 Migracija transakcija

Konflikt može da se desi samo u toku izvršavanja transakcije. Pošto na asimetričnom višejezgarnom procesoru postoji veliko jezgro, možemo ga koristiti za izvršavanje transakcija. Veliko jezgro može skratiti vreme izvršavanja transakcije. Hipoteza je da program koji koristi transakcionu memoriju može da se ubrza na asimetričnom višejezgarnom procesoru ubrzavanjem transakcija. Na slici 4.1a) prikazano je jedno moguće izvršavanje transakcija na simetričnom višejezgarnom procesoru. U trenutku ❶ transakcija T2 počinje svoje izvršavanje. Od tog trenutka aktivne su transakcije T1 i T2, sve dok se ne završi transakcija T1 u trenutku ❷. U tom vremenskom periodu, ako transakcije T1 i T2 pristupaju istim podacima, može doći do konflikta koji će prouzrokovati poništavanje jedne od transakcija. Rad poništene transakcije je bio uzaludan i mora da se ponovi, izvršavanje niti se produžava i to može smanjiti brzinu izvršavanja programa. Ista situacija je i od trenutka ❸ do trenutka ❹ kada su istovremeno aktivne transakcije T2 i T3.

Što su vremenski periodi u kojima su aktivne bar dve transakcije duži, veća je verovatnoća da se pojavi konflikt. Pretpostavka je da se dužina takvih vremenskih perioda može skratiti, ako se isti program izvršava na asimetričnom višejezgarnom procesoru. Osnovna ideja je da se jedna od transakcija izvrši na velikom jezgru. Na taj način dužina vremena izvršavanja transakcije se skraćuju pa se skraćuju i vremenski periodi u kojima su aktivne bar dve transakcije. Smanjuje se i verovatnoća da dođe do konflikta između dve transakcije. Postoji više načina da se jedna od transakcija izvrši na velikom jezgru. Mogući načini će biti razmatrani u nastavku poglavlja. U ovom primeru jedna od transakcija je premeštena da se izvršava na velikom jezgru, neposredno pre njenog početka. Na slici 4.1b) je dat primer izvršavanja istih transakcija kao na slici 4.1a), samo na asimetričnom višejezgarnom procesoru. Dati primer je idealizovan, tj. prikazuje najbolji mogući slučaj. U trenutku ①, kada jezgro CPU2 treba da krene sa izvršavanjem transakcije T2, jezgro CPU2 migrira izvršavanje transakcije na veliko jezgro CPU3. Do trenutka ②, vrši se migriranje niti, koje se sastoji od zaustavljanja izvršavanja na jezgru CPU2, prenosa konteksta na jezgro CPU3 i startovanje izvršavanja na tom jezgru. U tom vremenskom periodu ne može doći do konflikta, jer transakcija T2 nije aktivna. Ona se aktivira tek u trenutku ②. Od tog trenutka do trenutka ③, aktivne su dve transakcije T1 i T2.

Taj period je kraći u odnosu na izvršavanje prikazano na slici 4.1a), jer je transakcija T2 počela kasnije, i samim tim verovatnoća za konflikt je manja. Od trenutka ④ do trenutka ⑤ su aktivne transakcije T2 i T3. Taj period je takođe kraći nego period od ❸ do ❹, jer se transakcija T2 izvršava na

bržem jezgru CPU3 i završava se ranije. Od trenutka kada T2 završi svoje izvršavanje transakcija se vraća na CPU2 do trenutka ⑥. Razlika u brzini izvršavanja u oba slučaja nema ako ne dođe do konflikta i poništavanja transakcija. Kada do konflikta dođe, izvršavanje će se usporiti. Ako se scenario prikazan na slici 4.1b) dešava često, verovatnoća za usporenje usled poništavanja transakcija će biti manja što može dovesti do boljih performansi izvršavanja programa na asimetričnom višejezgarnom procesoru.



Slika 4.1: *Izvršavanje transakcija na: a) simetričnom i b) asismetričnom višejezgarnom procesoru*

Na ovom primeru je prikazano da nit koja se izvršava na jezgru CPU2, uradi svoj posao u vremenskom intervalu slične dužine u oba slučaja. Situacija je uprošćena i zanemareni su neki detalji, da bi se jednostavnije objasnila osnovna ideja. Prilikom stvarnog izvršavanja ta vremena će se razlikovati. Kako će se razlikovati zavisi od vremena koje je potrebno da se transakcija migrira i koliko brže će se izvršiti transakcija T2 na drugom jezgru. Migriranje transakcija je čist režijski trošak, koji može dovesti do usporavanja izvršavanja niti. Da bi se migriranje transakcija isplatilo, režijski trošak mora biti manji nego uzaludni rad koji je izvršila poništena transakcija.

Prikazani primer razmatra slučaj kada se pristupi podacima koji izazivaju konflikt dešavaju u transakcijama. Situacija koja može da nastane u sistemima sa jakom atomičnošću je slična. Umesto u drugoj transakciji, pristup koji izaziva konflikt bi se dogodio u kritičnoj sekciji koja je zaštićena bravom. Isto važi kao i u slučaju sa preklapanjem dveju transakcija. Što je preklapanje transakcije i kritične sekcije kraće manja je verovatnoća da dođe do konflikta. Naravno, pristupi koji izazivaju konflikte mogu da se dogode i van kritične sekcije, ali takvi slučajevi su retki i u ovoj disertaciji su tretirani kao plod programerske greške. Razlog za to je što moderni višejezgarni procesori u najvećem broju slučajeva podržavaju samo relaksiranu memorijsku konzistenciju i problemi koji bi tu mogli da nastanu prevazilaze okvire ovog rada.

Pored upravo opisane potencijalne koristi, izvršavanje na velikom jezgru može doprineti da neke transakcije uopšte mogu da se izvrše. Ako veliko jezgro ima veće bafere i veću keš memoriju, onda to jezgro ima veći kapacitet za smeštanje spekulativnih podataka jedne transakcije. Veći kapacitet omogućava da na tom jezgru mogu da se izvrše neke transakcije koje se ne mogu izvršiti na malom

jezgru zbog poništavanja transakcije usled prekoračenja. Ova vrsta benefita mogla bi da postoji samo kod tačno onih aplikacija čiji su skupovi spekulativnih podataka u transakcijama takvih veličina da na malom jezgru ne mogu da se izvrše, a na velikom mogu.

Postoji više načina na koji može da se obezbedi izvršavanje transakcija na velikom jezgru. Trivijalan način je da se jedna nit programa izvršava na velikom jezgru i da se sve njene transakcije izvršavaju brže u odnosu na slučaj kad bi se ta nit izvršavala na malom jezgru. Takav način, iako jednostavan za implementaciju, ubrzava sve transakcije jedne niti. Time je vrlo mala verovatnoća da se dogodi sličan scenario onom opisanom na slici 4.1. Neke transakcije možda već imaju malu verovatnoću konflikta (kratke transakcije ili one koji imaju mali skup podataka) pa ubrzavanjem takvih transakcija se neće dobiti nikakvi benefiti. Ovaj način je statičan, jer ne omogućava smanjenje konflikta između transakcija koje se izvršavaju na malim jezgrima.

Drugi način je da operativni sistem kontroliše kada će se neka transakcija izvršavati na velikom jezgru. Jednostavniji pristup je da operativni sistem raspoređuje niti tako da se nit za koju se predviđa da će imati najviše konflikta kod izvršavanja transakcija prebaci na brže jezgro. Za takav pristup potrebno je osmisliti algoritam za predviđanje koja nit će imati najviše transakcija sa konfliktima i ugraditi ga u raspoređivač operativnog sistema. Da bi softver mogao da vrši tu procenu, potrebno je obezbediti pristup informacijama o izvršenim transakcijama (i o potvrđenim i o poništenim). U većini komercijalno dostupnih implementacija transakcione memorije, npr. [85], moguće je samo dobiti informaciju o uspehu izvršavanja transakcije i razlog poništavanja u slučaju neuspeha. Za neke druge informacije potrebno je proširiti instrukcijski set procesora (npr. koliko instrukcija se izvršilo u transakciji, koja transakcija je izazvala konflikt, itd.). Potencijalni problem sa ovakvim pristupom je što je premeštanje transakcije moguće raditi u relativno dalekim vremenskim trenucima, između kojih se može izvršiti veliki broj transakcija (i one koje su poželjne da se izvrše na velikom jezgru i one koje nisu poželjne). Tako da bi ovakvo rešenje bilo pogodno za one programe koji imaju faze izvršavanja u kojima transakcije jedne niti imaju konflikte sa transakcijama iz drugih niti. Drugi problem, koji može da se javi, je relativno često menjanje jezgra na kom se nit izvršava. To može dovesti do problema sa "hladnim startom", gde nit koja počne da se izvršava na drugom jezgru nema u keš memoriji mnogo podataka koji su joj potrebni. To će izazvati mnogo promašaja u keš memoriji i može značajno usporiti izvršavanje niti. Treći problem je što se ne smanjuje konflikt između transakcija koje se izvršavaju na malim jezgrima. Složeniji pristup bi bio da prilikom startovanja svake transakcije softver odluči da li transakcija treba da se izvrši na bržem jezgru. Ako treba, transakcija se hardverski migrira na brže jezgro. Ovakav pristup zahteva dodatnu promenu instrukcijskog skupa procesora, da bi se dodale odgovarajuće instrukcije za startovanje transakcije. Izvršavanje algoritma predviđanja od strane operativnog sistema može ugroziti performanse izvršavanja, ako se nalazi na kritičnom putu.

Treći način je da se u hardveru realizuje algoritam za predviđanje da li transakciju treba prebaciti na veliko jezgro. S obzirom na to da je transakcija jasno definisana celina koja počinje instrukcijom za startovanje transakcije i završava se ili instrukcijom za potvrđivanje ili poništavanje transakcije, hardver može na lak način da detektuje u kom trenutku treba migrirati izvršavanje sa jezgra na jezgro i nije potrebno menjati instrukcijski set i softver. Potrebno je voditi evidenciju o transakcijama da bi algoritam mogao da odredi da li neku transakciju treba prebaciti na veliko jezgro. Sve te informacije mogu da budu dostupne algoritmu bez potrebe menjanja instrukcijskog seta. S obzirom da je algoritam implementiran u hardveru, njegovo izvršavanje može da se izvršava paralelno sa izvršavanjem programa. To izvršavanje algoritma neće uticati na performanse izvršavanja programa. Pošto algoritam treba da se implementira u hardver, taj algoritam mora da bude dovoljno jednostavan da ga je moguće implementirati. Njegova implementacija mora da zauzima malo prostora na čipu, jer će ona najviše koristiti memorijski prostor koji je uvek potreban za povećanje kapaciteta keš memorija. Softverska rešenja su tu u prednosti, jer mogu značajno složenije algoritme da implementiraju. Druga prednost softverskih rešenja je to što algoritmi mogu da budu značajno konfigurabilni i da se fino podešavaju za svaki tip programa, dok konfigurabilnost hardverskog rešenja može da se svede na promenu nekih celobrojnih parametara. Najveći problemi prilikom primene novog rešenja su: promena

instrukcijskog seta i promena već postojećih programa. Ti problemi su krivi za spori prodor novije tehnike u komercijalne proizvode. Iz tog razloga u ovoj disertaciji se razmatra hardversko rešenje koje je moguće minimalno konfigurisati.

I hardversko i softversko rešenje sa migriranjem pojedinačnih transakcija ima jedan problem, koji može učiniti takva rešenja praktično neupotrebljivim. Prilikom migracije niti potrebno je zaustaviti protočnu obradu jezgra, zapakovati kontekst niti, prekopirati kontekst na drugo jezgro i pokrenuti protočnu obradu na tom jezgru. Ukoliko se na tom drugom jezgru izvršavala neka nit u drugom adresnom prostoru, moraju se poništiti neki delovi keš memorije, baferi za preslikavanje virtualne memorije, itd. Cela ta promena može trajati nekoliko miliona taktova u najgorem slučaju [123]. To vreme je isuviše dugačko da bi česta migracija bila isplativa, pa bilo koje rešenje sa migriranjem pojedinačnih transakcija ne bi poboljšalo performanse izvršavanja. Međutim, transakcija je takva celina da u njoj ne smeju da se koriste neke stvari zbog kojih to vreme migracije može da bude veliko. Na primer, ne mora ceo kontekst niti da se prekopira već samo onaj deo koji se čuva prilikom startovanja transakcije. Migracija će biti vršena samo ako se prethodno na velikom jezgru izvršavala nit u istom adresnom prostoru kao i nit čiju transakciju hoćemo da migriramo. Na taj način može se postići migracija u nekoliko redova veličine manjem vremenu, što omogućuje implementaciju nekog rešenja koja će poboljšati performanse izvršavanja.

Prenos konteksta prilikom migracije može da se izvrši na dva načina. Prvi način je da se na steku niti zapamti celokupan kontekst i da se prilikom migriranja drugom jezgru proslede samo adresa naredne instrukcije i adresa vrha steka. Tada na drugom jezgru treba da se sa steka skine celokupan kontekst niti i tada se započinje sa izvršavanjem transakcije. Isti postupak se primenjuje i prilikom vraćanja izvršavanja na malo jezgro ukoliko je transakcija bila uspešna. Na ovaj način, prenos konteksta se suštinski obavlja preko protokola keš koherencije. Drugi način je da se prebaci celokupan kontekst zajedno sa porukom za migriranje transakcije. Za to se može iskoristiti mehanizam za pamćenje trenutnog sadržaja arhitekturalnih registara [124], koji postoji i zbog same implementacije transakcione memorije. Taj sadržaj je potrebno proslediti odgovarajućim porukama drugom jezgru koje treba da nastavi izvršavanje. Sa stanovišta performansi i kompleksnosti implementacije ne postoji neka očigledna prednost jednog ili drugog načina. Potrebno je istražiti koji način nekoj mikroarhitekturi više odgovara. Prilikom neuspešnog migriranja transakcije oba načina mogu iskoristiti već postojeći mehanizam transakcione memorije čuvanja konteksta i nije potrebno vraćati kontekst malom jezgru. Potrebno je samo obavestiti porukom malo jezgro da ponovo pokrene transakciju.

Važna odluka u dizajnu je šta će raditi veliko jezgro dok ne izvršava migriranu transakciju. Najjednostavnije je da ne radi ništa. Na taj način, čim se pojavi zahtev za izvršavanje migrirane transakcije može da ga prihvati. Drugi pristup je da veliko jezgro izvršava nit programa kao i druga jezgra. Kada pristigne zahtev za izvršavanje migrirane transakcije treba da zaustavi izvršavanje svoje niti i izvrši migriranu transakciju. Korektnost izvršavanja mora da se očuva, pa se zaustavljanje niti mora uraditi izvan bilo koje transakcije. Ovaj pristup povećava mogući paralelni rad, ali usporava samu migraciju i može doprineti usporavanju migracija. Ne može se unapred tvrditi koji je pristup bolji. U nastavku disertacije će biti analiziran i jedan i drugi pristup.

4.2 Prilagođenje podsistema keš memorija podelom keš memorija

Podsistem keš memorija zauzima veliki prostor na čipu. Broj keš memorija zavisi od broja jezgara, pa sa porastom tog broja raste i zauzeće prostora na čipu. Konvencionalne keš memorije su dizajnirane tako da najviše iskorišćavaju prostornu lokalnost u pristupu podacima. Zbog takvog dizajna keš memorije moraju da budu velike da bi mogle da čuvaju i susedne podatke onog podatka kojem se trenutno pristupa. Pristup podacima se radi na nivou od nekoliko reči (četiri ili osam). Zbog veličine keš memorija vreme pristupa je dugo, od nekoliko do nekoliko desetina ciklusa takta. Međutim, kad

Algoritam 1 Pseudo kod za pronalaženje maksimuma

```
1:  function FIND_MAX(array, size)
2:      if size < 1 then
3:          return 0
4:      end if
5:      max ← array[0]
6:      while i < size do
7:          if max > array[i] then
8:              max ← array[i]
9:          end if
10:         i ← i + 1
11:     end while
12:     return max
13: end function
```

se pogleda neki kod može se uočiti da ne ispoljavaju svi pristupi podacima prostornu lokalnost. Na primer, listing algoritma 1 prikazaju pseudo kod koji pronalazi maksimum u nekom nizu. Namerno je uzet jednostavan algoritam da bi se uočilo da čak i u najjednostavnijem primeru postoje podaci kojima se ne pristupa sa prostornom lokalnošću. Skalarnim podacima *size*, *max*, *array* i *i* se pristupa sa vremenskom lokalnošću.

Podaci kojima se pristupa sa vremenskom lokalnošću se smeštaju u keš memoriju zajedno sa ostatkom bloka. Taj ostatak bloka možda nije potreban i uzalud je tu smešten i zauzima prostor u keš memoriji. Zbog toga se predlaže podela keš memorije na dva dela koja će biti prilagođena lokalnosti sa kojom se pristupa podacima [1], [111], [112]. Jedan deo bi bio isto implementiran kao i obična keš memorija i služio bi za podatke kojima se pristupa sa prostornom lokalnošću, dok bi drugi deo služio za smeštanje pojedinačnih reči, tj. za one podatke kojima se pristupa sa vremenskom lokalnošću. Pretpostavka je da se na taj način može poboljšati iskorišćenje keš memorije, tako što će ti delovi zajedno zauzimati manje prostora nego konvencionalna keš memorija. Deo koji služi za čuvanje reči može da se implementira znatno jednostavnije, pa vreme pristupa može da bude značajno kraće nego kod konvencionalnih keš memorija. Pretpostavka je da kraćim pristupom pojedinačnim delovima može da se dobije ubrzanje koje će nadoknaditi smanjenu veličinu keš memorije, a možda i dovesti do ubrzanja izvršavanja celokupne aplikacije.

Druga pretpostavka može da se obrazloži na sledeći način. Poznato je da je u konvencionalnim keš memorijama prosečno vreme pristupa t, uključujući i one situacije kad podatak nije u keš memoriji, jednako:

$$t = p \times t_c + (1 - p) \times t_m \tag{4.1}$$

gde je p verovatnoća da se podatka nalazi u keš memoriji, dok je t_c vreme pristupa kad je podatak u keš memoriji, a t_m vreme pristupa kad nije u keš memoriji. U podeljenoj keš memoriji prosečno vreme pristupa t je:

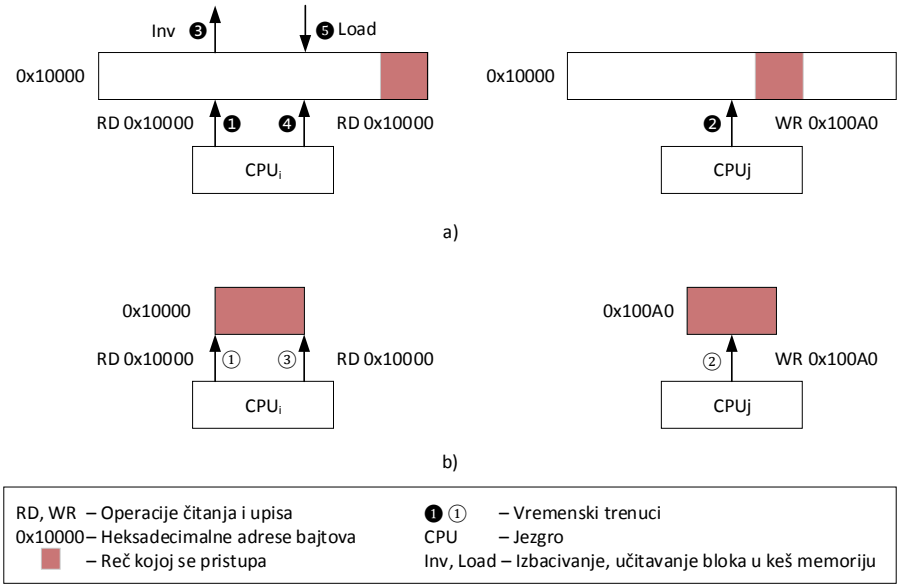
$$t = p_p \times t_p + p_v \times t_v + (1 - (p_p + p_v)) \times t_m \tag{4.2}$$

gde su p_p i p_v verovatnoće da se podatak nalazi u jednom od delova keš memorije, a t_p i t_v su vremena pristupa kad je podatak prisutan u keš memoriji, dok je t_m vreme pristupa kad podatak nije u keš memoriji. Ako se pretpostavi da je verovatnoća da se podatak nalazi u keš memoriji ista i za slučaj konvencionalne keš memorije i za slučaj podeljene keš memorije, lako se pokazuje da je prosečno vreme pristupa manje za podeljenje keš memorije. Tada važi:

$$p_p \times t_p + p_v \times t_v < p_p \times t_c + p_v \times t_c = (p_p + p_v) \times t_c = p \times t_c \tag{4.3}$$

Pored pomenutih benefita koje može da ostvari jedno jezgro od podeljenje keš memorije, moguće je da se ostvare benefiti i u višejezgarnom procesoru. Ukoliko se podaci u keš memorijama čuvaju

na nivou jedne reči, može da se izbegne lažno deljenje podataka (eng. *false sharing*). Lažno deljenje je pojava da jedno jezgro pristupa jednoj reči u bloku, dok neko drugo jezgro pristupa drugoj reči u istom bloku. Ako je jedan od tih pristupa upis, poništava se ceo blok u keš memoriji jednog jezgra i to jezgro prilikom sledećeg pristupa tom podatku će imati dugačko vreme pristupa, jer podatak nije u keš memoriji. Na slici 4.2a) prikazan je sled događaja koji izazivaju lažno deljenje. Prvo jezgro *i* u trenutku ❶ pročita reč na adresi 0x10000. Nakon toga jezgro *j* u trenutku ❷ upiše reč na adresi 0x100A0. To prouzrokuje poništavanje bloka u keš memoriji jezgra *i* u trenutku ❸. Kada jezgro *i* u trenutku ❹ ponovo pokuša da pročita reč na adresi 0x10000, moraće da se dovuče blok u keš memoriju u trenutku ❺. Ta poslednja operacija traje dugo. Operacije u trenucima ❸ i ❺ za održavanje memorijske koherencije nisu potrebne.



Slika 4.2: *Pristup jednom bloku u: a) konvencionalnoj keš memoriji i b) podeljenoj keš memoriji*

Kada je keš memorija podeljena na opisani način, lažno deljenje može da se izbegne, jer će biti poništavane samo one reči kojima jezgra stvarno pristupaju paralelno. Na slici 4.2b) je prikazan taj slučaj. Jezgro *i* u trenutku ① čita reč sa adrese 0x10000, koja je jedina reč iz tog ulaza u keš memoriji. Nakon toga u trenutku ② jezgro *j* upisuje u reč na adresi 0x100A0. Taj upis ne poništava ni jedan ulaz u keš memoriji jezgra *i*, jer ti ulazi ne sadrže reči na istoj adresi. U trenutku ③, jezgro *i* čita reč na adresi 0x10000 i to čitanje je kratka operacija, jer je reč prisutna u keš memoriji.

Lažno deljenje može da prouzrokuje i veće probleme od jednog dodatnog izbacivanja bloka iz keš memorije pa njegovog ponovnog dohvatanja. Ako bi se lažno deljenje pojavilo u toku izvršavanja transakcije, transakcija bi morala da bude poništena. U zavisnosti koliko je izvršavanje transakcije poodmaklo, može uzaludno biti poništeno mnogo ispravno izvršenih instrukcija. Kada je keš memorija podeljena, postoji šansa da se lažno deljene izbegne pa samim ti i da se smanji broj poništavanja transakcija.

U koji deo keš memorije će podatak bi ti smešten mora da se proceni na neki način. Tu procenu može da radi prevodilac, statički za vreme prevođenja. Procena može da ugradi u objektni kod, koja će biti dostupna podsistemu keš memorija. To se može uraditi posebnom instrukcijom. Pored statičke analize, procena može da se uradi i profilisanjem koda (eng. *profiling*), gde bi se program pokrenuo u testnom okruženju. Prilikom izvršavanja bi se vodila evidencija kako se pristupa podacima i ta procena bi mogla na sličan način kao i kod prethodne varijante da se ugradi u objektni kod. Treća mogućnost je da se u hardveru doda mehanizam za procenu kako se podacima pristupa. Implementacija u hardveru bi bila transparentna za programera i već postojeće aplikacije bi mogle da se izvršavaju u tom sistemu bez bilo kakvih promena.

U programski prevodilac može da se ugradi mehanizam koji bi koristio na efikasan način podeljenu keš memoriju. To bi se postiglo tako što bi prevodilac grupisao podatke u blokove prema procenjenoj lokalnosti pristupa. Na taj način bi bilo lakše i preciznije utvrditi sa kojim tipom lokalnosti se pristupa podacima i može se postići bolje iskorišćenje keš memorije. Promenljive za koje može da se utvrdi da im se pristupa sa vremenskom lokalnošću, a deljenje su između više niti mogu da se stave u odgovarajuće blokove, tako da im se lako odredi da imaju vremensku lokalnost pristupa i da se izbegne lažno deljenje. Takve promenljive su na primer promenljive koje služe za zaključavanje brava, implementaciju barijera, itd. U ovoj disertaciji rad prevodioca nije analiziran i svi eksperimenti su rađeni pomoću komercijalno dostupnih i neizmenjenih prevodilaca.

Važna odluka u dizajnu ovakvog rešenja je: kako organizovati nivoe keš memorija i koje memorije treba da budu podeljenje. Ako su podeljene keš memorije samo prvog privatnog nivoa, onda protokol koherencije keš memorije ne mora da se menja značajno, jer u protokolu koherencije učestvuju keš memorije poslednjeg privatnog nivoa i keš memorije deljenog nivoa. Međutim, u tom slučaju lažno deljenje ne može da se izbegne, jer su keš memorije obično inkluzivne. Takođe, niži nivoi keš memorija su i manji, pa su uštede prostora na čipu manje. Ukoliko bi sve keš memorije privatnih nivoa bile podeljenje, pomenuti benefiti mogu više da dođu do izražaja po cenu promene protokola keš koherencije. U sledećem poglavlju pokazano je kako se može implementirati podsistem keš memorija sa podeljenim keš memorijama svih privatnih nivoa i da su promene protokola keš koherencije male, ali svakako moraju biti pažljivo verifikovane prilikom implementacije.

Poglavlje 5

Predloženo rešenje

U ovom poglavlju predstavljena su dva rešenja za ubrzanje programa koji koriste transakcionu memoriju. Jedno je rešenje za migraciju transakcija (M-HTM). Drugo rešenje je prilagođenje podsistema keš memorija. Svako od predloženih rešenja biće prikazano pojedinačno. Prikaz se sastoji od dva dela. Jedan deo opisuje algoritam rada rešenja, dok drugi deo opisuje jedan mogući način implementacije. Opis implementacije sadrži i diskusiju o ceni te implementacije.

5.1 Sistem za migraciju transakcija M-HTM

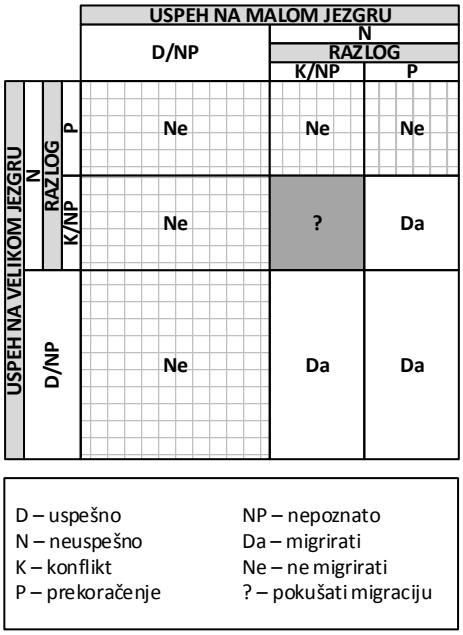
Algoritam za migraciju transakcija, koji je nazvan M-HTM, se u potpunosti implementira u hardver. Za njegov rad nije potrebno menjati kod aplikacije ni na koji način. Podrazumeva se da procesor podržava transakcionu memoriju i da ima eksplicitne instrukcije za startovanje i za završavanje transakcije. Za sve ostale detalje konkretna implementacija transakcione memorije nije važna i algoritam se može prilagoditi implementacionim detaljima transakcione memorije (u većini slučajeva nema potrebe za bilo kakvom promenom). Višejegarni procesor mora biti asimetričan. Sva jezgra imaju isti instrukcijski set, a asimetična su u pogledu mikroarhitekturalnih svojstava.

Predloženi algoritam je osmišljen za procesor koji ima N jezgara, gde su N - 1 jezgara mala jezgra, dok je N-to jezgro veliko i drugačijih mikroarhitekturalnih svojstava. Veliko jezgro treba da ima bolje performanse od svih ostalih i treba da ima veće kapacitete svojih jedinica. Instrukcijski set svih jezgara je identičan. Postojanje više od jednog velikog jezgra je moguće. Predloženi algoritam može da se modifikuje da podržava rad i na procesorima koji imaju više velikih jezgara, ali to prevazilazi okvire ovog istraživanja i može da bude tema budućeg rada.

5.1.1 Opis algoritma za migraciju transakcija

Algoritam donosi odluku da li transakciju treba migrirati na veliko jezgro na osnovu istorije izvršavanja te transakcije. Radi preciznosti potrebno je razlikovati pojam transakcije i jedno izvršavanje te transakcije. Pod pojmom transakcije podrazumevamo deo koda koji počinje i završava se instrukcijama za početak i kraj transakcije. Pod pojmom jedno izvršavanje transakcije podrazumevamo izvršavanje dela koda koji predstavlja transakciju u okviru bilo koje niti na bilo kom jezgru. Dakle, istorija se pamti na nivou transakcije. Istorija jedne transakcije uključuje sva pojedinačna izvršavanja te transakcije.

Za svaku transakciju vodi se evidencija o uspešnosti, o tipu neuspeha i o veličini transakcije. Evidencija o uspešnosti i o tipu neuspeha se vodi odvojeno za izvršavanja na malom i velikom jezgru. Evidencija o uspešnosti čuva informaciju da li je transakcija u prethodnih nekoliko izvršavanja potvrđena ili poništena. Evidencija o tipu neuspeha čuva informaciju da li je transakcija u prethodnih nekoliko neuspešnih izvršavanja poništena jer je došlo do konflikta sa drugom transakcijom ili je transakcija poništena zbog prekoračenja. U evidenciji o veličini transakcije čuva se informacija da



Slika 5.1: *Akcije u tipičnim slučajevima*

li je transakcija dovoljno dugačka da bi migracija bila opravdana. Algoritmu se mora dostaviti dinamička dužina transakcije, izražena u broju instrukcija, da bi vodio tu evidenciju. Svako jezgro dok izvršava transakciju broji svaku izvršenu instrukciju. Brojanje se vrši pomoću brojača koji se resetuje prilikom započinjanja transakcije. Vrednost brojača se prosleđuje algoritmu nakon završetka transakcije. Dinamička dužina transakcije može značajno da varira, zbog toka izvršavanja programa ili zato što je transakcija poništena u proizvoljnom trenutku. Algoritam zbog toga određuje da li je transakcija dovoljno dugačka za migraciju na osnovu nekoliko prethodnih izvršavanja (u eksperimentima u ovoj disertaciji korišćena je najveća dotad zabeležena dužina transakcije).

Trenutno stanje istorije određuje se na osnovu podataka u evidenciji. Na osnovu stanja istorije određuje se da li transakciju treba migrirati ili ne. Slika 5.1 prikazuje koje akcije treba preduzeti u zavisnosti od stanja istorije za pojedinačnu transakciju. Ukoliko se transakcija uspešno izvršava na malom jezgru, nema potrebe za migracijom, jer se ne događaju konflikti pa nema potrebe smanjivati verovatnoću njihovog nastanka. Takođe, nema potrebe migrirati transakciju ni u slučaju da transakcija ne uspeva da se izvrši na velikom jezgru zbog prekoračenja. U tom slučaju najbolje je što pre preći na izvršavanje sa zaključavanjem, jer transakcija ne može da se izvrši ni na jednom jezgru datog procesora. Transakciju treba migrirati u slučaju da se uspešno izvršava na velikom jezgru i ne uspeva da se izvrši na malom jezgru, iz bilo kog razloga. U slučaju da transakcija ne može da se izvrši na malom jezgru zbog prekoračenja, poželjno je migrirati na veliko jezgro, ako postoji šansa da se na njemu uspešno izvrši.

Ako je transakcija poništavana dok se izvršavala i na malom i na velikom jezgru pretežno usled konflikata, stanje istorije ne pomaže u određivanju da li transakciju treba migrirati. U tom slučaju je paralelizam isuviše veliki i izaziva česte konflikte među transakcijama. Takva faza izvršavanja na kraju mora da se završi zaključavanjem koje će obezbediti potrebnu serijalizaciju i smanjenje paralelizma. U tom slučaju najbolje je ne gubiti vreme na migraciju transakcije i obezbediti da na malim jezgrima što pre dođe do izvršavanja pomoću zaključavanja brave. U slučaju da se paralelizam smanji toliko da transakcije i dalje ne mogu da se izvršavaju na malim jezgrima, a mogle bi na velikom, algoritam ne može da promeni stanje istorije, jer migracije nisu dozvoljene. Takvu promenu nije lako otkriti, pa mogućnost za proveru ne postoji u algoritmu. Da sistem ne bi ostao inertan, dok traje takvo stanje periodično se dozvoljava migracija transakcije. Ako migracija bude uspešna prelazi se u stanje u kome se transakcija uspešno izvršava na velikom jezgru i transakcija se na dalje migrira. U suprotnom, ostaje se u istom stanju i migracije se opet ne dozvoljavaju. Pokušavanje migracije ne bi trebalo

da ugrozi performanse značajno, jer se transakcije ionako neuspešno izvršavaju.

U slučaju kada ne postoji istorija izvršavanja transakcije (recimo prvi put se izvršava transakcija), uzima se da se transakcija uspešno izvršava na oba tipa jezgra i da nije bilo poništavanja transakcije. To je stanje istorije u kome nema migracije i na taj način se daje prednost izvršavanju transakcija na malim jezgrima. Ako transakcija, počne da se izvršava neuspešno na malim jezgrima, migracija će biti omogućena, jer početna pretpostavka je da se i na velikom jezgru transakcija uspešno izvršava. Za početan razlog za poništavanje transakcije uzima se poništavanje usled konflikta za obe vrste jezgra. To je izabrano, jer se stanje transakcije lakše menja u stanje gde je dozvoljena migracija iz onih stanja gde je razlog za neuspeh konflikt nego prekoračenje.

Svaki put kada transakcija počne uspešno da se izvršava na malom jezgru, evidencija za izvršavanje na velikom jezgru se poništava. Drugim rečima evidencija za veliko jezgro pokazuje da se transakcija uspešno izvršavala i da su razlozi za neuspešno izvršavanje pretežno konflikti. Na taj način se izbegava da daleka istorija sprečava migraciju. Jedna primer je da transakcija u nekim izvršavanjima bila dugačka pa je prouzrokovala poništavanja usled prekoračenja. Nakon nekog vremena dužina transakcije se smanji, pa počne uspešno da se izvršava na malim jezgrima. Ako transakcija počne neuspešno da se izvršava zbog konflikta, može da se desi da stanje istorije bude takvo da se transakcija ne migrira, jer se davno neuspešno izvršavala na velikom jezgru zbog prekoračenja.

Pseudo kod za algoritam je dat na listingu algoritma 2. Prikazani kod oslikava odlučivanje da li transakciju treba migrirati na osnovu tekućeg stanja kada se neka transakcija završi. Kod ne prikazuje periodično ažuriranje stanja migracije. Vremenska složenost algoritma je konstantna, pošto algoritam ne sadrži petlje i ne zavisi od broja jezgara. Prostorna složenost algoritma je takođe konstantna, jer ne zavisi od broja jezgara, već samo zahteva pet brojača i četiri flip flopa za vođenje evidencije o trenutnom stanju. Takođe, nekoliko sabirača i komparatora su potrebni za implementaciju. Potrebna širina elemenata je mala (u sprovedenim eksperimentima je četiri). Procenjeno je da vreme izvršavanja za ovaj algoritam može biti jedan ili najviše dva ciklusa takta na modernim procesorima koji rade na visokoj frekvenciji.

Broj transakcija koje je u jednom trenutku moguće migrirati zavisi od konfiguracije velikog jezgra. Taj broj je jednak broju niti koje jezgro može paralelno da izvršava. Ako je broj malih jezgara koji treba da migriraju svoja izvršavanja transakcija veći od maksimalnog broja niti koje mogu da se izvršavaju na velikom jezgru, moguće je odustati od migracije ili sačekati da se neka od migriranih niti završi. Poželjno je sačekati ako je očekivano vreme čekanja kraće od vremena koje bi bilo utrošeno na jalovo poništavanje i ponovno izvršavanje transakcije. Ukupan broj transakcija koje su migrirane i koje čekaju na migraciju, treba da bude manji od broja malih jezgara, jer u suprotnom mogla bi da se dobija potpuna serijalizacija transakcija što značajno može da smanji performanse izvršavanja programa. U predloženom algoritmu obezbeđeno je da mali broj transakcija može da bude na čekanju, tako da se da šansa migraciji, a da se paralelizam ne smanji značajno. Ako malo jezgro pokuša da migrira transakciju, a nema mesta za čekanje, malo jezgro odustaje od migracije i pokušava izvršavanje transakcije. Obratiti pažnju da zbog ovog ograničenja, stanje u kome se transakcija uspešno izvršava na velikom jezgru nije trajno, jer na nekim malim jezgrima može da se pokuša sa izvršavanjem transakcije. Ako ti pokušaji budu često uspešni, može da se pređe u stanje u kome se transakcije na malom jezgru uspešno izvršavaju i algoritam može da blokira migraciju te transakcije.

Obratiti pažnju da se stanje istorije određuje na osnovu globalnih informacija o transakciji, tj. koriste se informacije o transakciji koja se izvršavala na bilo kojim jezgrima. Pored globalnih informacija malo jezgro uzima u obzir i lokalnu informaciju. Ako je prethodno izvršavanje transakcije na tom jezgru bilo uspešno, onda se sledeći put pokušava izvršavanje opet bez migriranja. Takođe, ako je malo jezgro migriralo transakciju na veliko jezgro i tamo je poništena, malo jezgro sledeći put izvršava tu transakciju bez migriranja. Na taj način neposredna lokalna istorija ima prvenstvo u odnosu na globalnu istoriju. Time se izbegava veliki broj migracija, čije su verovatnoće da budu korisne male.

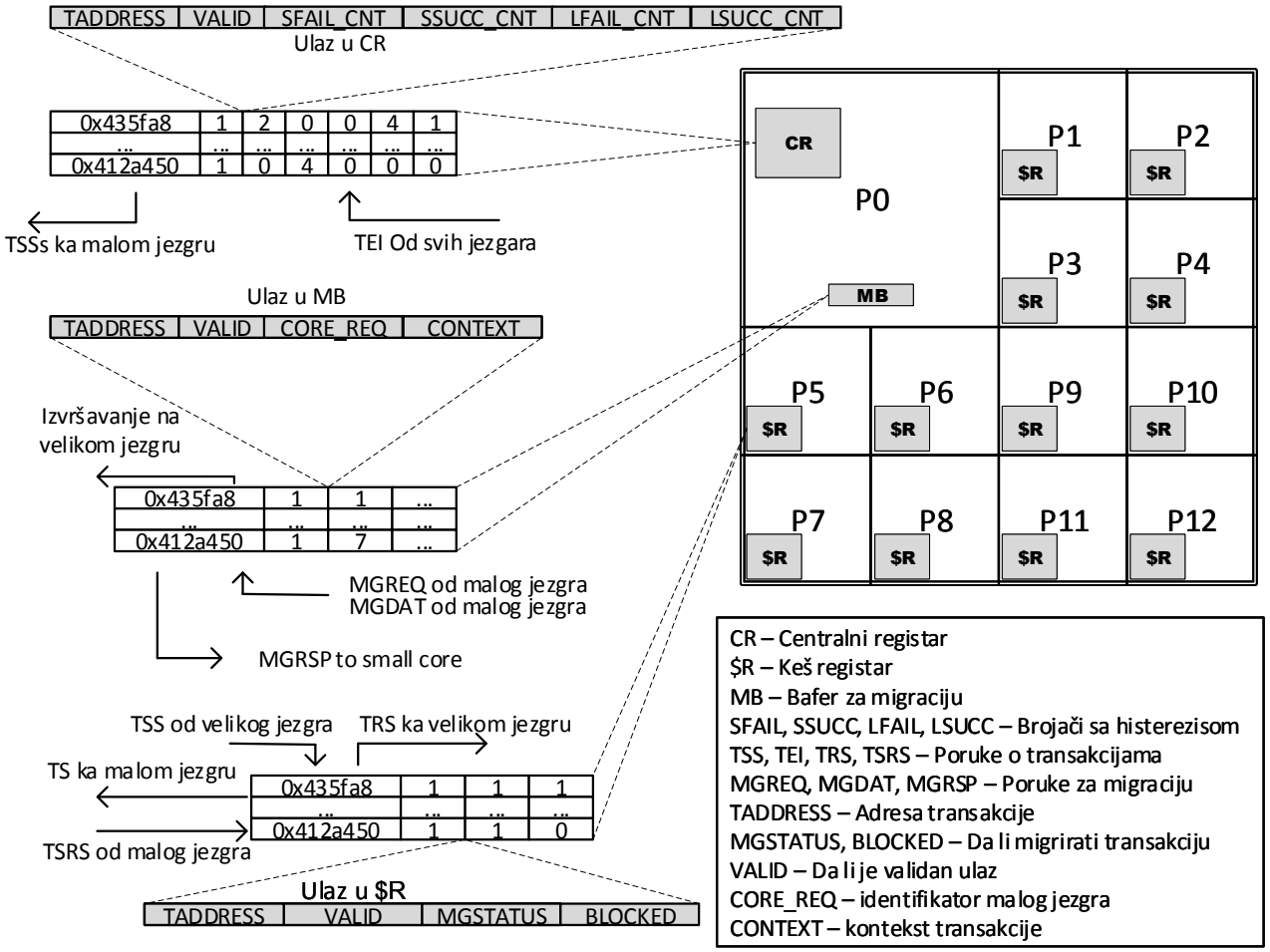
Algoritam 2 Pseudo kod algoritma koji određuje da li transakciju treba migrirati, na osnovu istorije izvršavanja

```
1: procedure HYSTERESIS(counter, state, value)
2:     counter ← counter + value
3:     if state = NO and counter >= UPPER_LIMIT then
4:         state ← YES
5:     else if state = YES and counter <= LOWER_LIMIT then
6:         state ← NO
7:     end if
8: end procedure
9: small_scnt ← MIDDLE
10: big_scnt ← MIDDLE
11: small_ccnt ← MIDDLE
12: big_ccnt ← MIDDLE
13: small_success ← YES
14: big_success ← YES
15: big_conflict ← YES
16: small_conflict ← YES
17: max_length ← 0
18: function TO_MIGRATE(core, result, failure_type, length)
19:     success_value ← 1
20:     if result = FAIL then
21:         success_value ← −1
22:     end if
23:     failure_value ← 1
24:     if core = BIG then
25:         hysteresis(big_scnt, big_success, success_value)
26:         hysteresis(big_ccnt, big_conflict, failure_value)
27:     else
28:         previous_state ← small_success
29:         hysteresis(small_scnt, small_success, success_value)
30:         hysteresis(small_ccnt, small_conflict, failure_value)
31:         if previous_state! = small_success and previous_state = NO then
32:             big_scnt ← MIDDLE
33:             big_ccnt ← MIDDLE
34:             big_success ← YES
35:             big_conflict ← YES
36:         end if
37:     end if
38:     if max_length < length then
39:         max_length ← length
40:     end if
41:     if max_length < LENGTH_LIMIT or small_success = YES then
42:         return NO
43:     end if
44:     if big_success = YES or (small_conflict = NO and big_conflict = YES) then
45:         return YES
46:     end if
47:     return NO
48: end function
```

5.1.2 Predlog implementacije sistema M-HTM

Za rad algoritma je potrebno skupljati informacije o izvršenim transakcijama i treba ih čuvati u hardveru. Na slici 5.2 je prikazan asimetrični višejezgarni procesor sa dodatim jedinicama za implementaciju predloženog algoritma. Takođe, prikazane su i poruke koje šalju i primaju dodate jedinice. Informacije o transakcijama se čuvaju centralizovano i to u jednoj maloj memoriji u velikom jezgru. Ta mala memorija biće nazivana centralni registar CR. Prilikom svakog započinjanja transakcije malo jezgro treba da proveri da li transakciju treba migrirati na osnovu njene istorije. Dohvatanje tog podatka iz centralnog registra može da potraje, u zavisnosti od brzine komunikacije. To usporenje može značajno da ugrozi performanse izvršavanja svake transakcije, čak i one koju ne treba migrirati. Iz tog razloga ima smisla u malo jezgro dodati malu memoriju koja će čuvati najsvežiju informaciju o transakcijama koje su se skoro izvršavale na malom jezgru. Tu malu memoriju nazivamo keš registar ($R). Keš registar čuva samo jedinstveni identifikator transakcije, da li tu transakciju treba migrirati i jedan bit koji označava da li migriranje treba lokalno blokirati (videti sliku 5.2). Jedinstveni identifikator transakcije može biti adresa prve instrukcije te transakcije. Pošto je keš registar mala memorija, čitanje iz njega može da se izvrši u jednom ciklusu takta i pokretanje transakcije neće biti usporeno. Pored centralnog registra u veliko jezgro treba dodati i bafer za migraciju (MB). On služi za čuvanje konteksta niti koje se migriraju i koje trebaju da čekaju dok veliko jezgro ne postane slobodno. Kada se veliko jezgro oslobodi, uzima se jedna nit iz bafera za migraciju i pokreće se transakcija na velikom jezgru.

Malo jezgro, kada naiđe na instrukciju za početak transakcije, treba da proveri da li transakcija treba da se migrira ili ne. Pseudo kod koji opisuje šta se dešava u tom trenutku dat je na listingu



Slika 5.2: *Asimetrični višejezgarni procesor sa dodatim jedinicama*

Algoritam 3 Pseudo kod za pokretanje transakcije na malom jezgru

```
1:  procedure LITTLE_BEGIN(transaction_id)
2:      to_migrate ← get_status(transaction_id, cache_registry)
3:      if to_migrate = YES then
4:          status ← migrate()
5:          if status = ACCEPT then
6:              return
7:          end if
8:      else if to_migrate = NOT_VALID then
9:          send_TRS_to_big(transaction_id)
10:     end if
11:     reset_length_counter()
12:     start_transaction()
13: end procedure
```

algoritma 3. Malo jezgro prvo zahteva status (TSRS) iz keš registra. Ukoliko je informacija o transakciji prisutna vrši se čitanje statusa (TSS). Ako je transakcija označena za migriranje i migriranje transakcije nije lokalno blokirano, šalje zahtev za migriranje (MGREQ) velikom jezgru, tačnije baferu za migraciju. Bafer za migraciju odgovara sa porukom (MGRSP) u kojoj se nalazi informacija da li može da prihvati migraciju (ima slobodnog mesta) ili ne. Ako je zahtev prihvaćen, malo jezgro šalje kontekst i čeka da se migrirana transakcija završi. Slanje posebne poruke za zahtev za migraciju, pa nakon toga poruke sa kontekstom niti očigledno usporava izvršavanje, ali se ne može izbeći zbog ograničene veličine bafera za migraciju. Kada je došlo do migracije, instrukcija za startovanje transakcije se ne izvršava. U svim ostalim slučajevima (kada transakcija nije za migraciju ili je migracija lokalno blokiranja, kada nema informacije o transakciji u keš registru i kada nema mesta u baferu za migraciju) transakcija se izvršava na malom jezgru. Tom prilikom se poništava brojač instrukcija, da bi se za tekuće izvršavanje odredila dužina transakcije.

U slučaju da keš registar ne sadrži informaciju o transakciji, keš registar zahteva status (TRS) od centralnog registra. Poruka se šalje asinhrono i ne čeka se odgovor, već malo jezgro izvršava transakciju. Iako transakcija može da bude obeležena za migriranje u centralnom registru, ovakav način izvršavanja ne utiče na korektnost programa, jer izvršavanje transakcije na bilo kom jezgru ima isti efekat i samo izvršavanje će biti isto kao da algoritma za migraciju i nema. Ovakav način izvršavanja može da utiče samo na performanse. Kada odgovor (TS) od centralnog registra stigne, informacija o transakciji se pamti u keš registru. Sledeći put kada malo jezgro bude izvršavalo istu transakciju na raspolaganju će imati podatke o transakciji.

Malo jezgro čeka i ne radi ništa dok čeka da veliko jezgro izvrši migriranu transakciju. Kada veliko jezgro završi izvršavanje transakcije, šalje malom jezgru poruku ili da je transakcija poništena (MGRSP) ili da je transakcija potvrđena sa kontekstom niti (MGDAT). Ukoliko je transakcija potvrđena, malo jezgro učitava kontekst koji je primilo i nastavlja dalje izvršavanje niti. U suprotnom, malo jezgro izvršava iste instrukcije kao da je pokrenulo transakciju i da je ona bila poništena. Na taj način se izvršava kod koji treba da se izvrši kad se transakcija poništi (restaurira se stanje pre početka transakcije i moguće je ponovno pokretanje transakcije). Efekat je isti kao da se transakcija izvršavala na malom jezgru. Obratiti pažnju da se kontekst restaurira iz lokalnih podataka jezgra i da nije potrebno da veliko jezgro šalje kontekst. Na taj način se smanjuje količina podataka koja se razmenjuje radi implementacije algoritma migriranja.

Sva komunikacija između jezgara se vrši preko linija za razmenu podataka iz keš memorija. Postojeći višejezgarni procesori imaju mrežu za komunikaciju između jezgara koja omogućava razmenu potrebnih poruka. Jedan od prvih primera takve arhitekture je Sun Niagara-1 procesor [125], gde jezgra šalju i primaju podatke sa deljenog koprocesora za aritmetiku realnih brojeva. Prilikom razmene

Algoritam 4 Završetak transakcije na velikom jezgru

```
1: procedure BIG_END(transaction_id, status, reason, length)
2:     if status = SUCCESS then
3:         data ← pack_data(transaction_id, length, YES, BIG)
4:     else
5:         data ← pack_data(transaction_id, length, NO, reason, BIG)
6:     end if
7:     save_info(data, central_registry)
8: end procedure
```

Algoritam 5 Završetak transakcije na malom jezgru

```
1: procedure LITTLE_END(transaction_id, status, reason, length)
2:     if status = SUCCESS then
3:         data ← pack_data(transaction_id, length, YES, SMALL)
4:         block_migration(transaction_id, cache_registry)
5:     else
6:         data ← pack_data(transaction_id, length, NO, reason, SMALL)
7:         unblock_migration(transaction_id, cache_registry)
8:     end if
9:     send_info_to_big(data)
10: end procedure
```

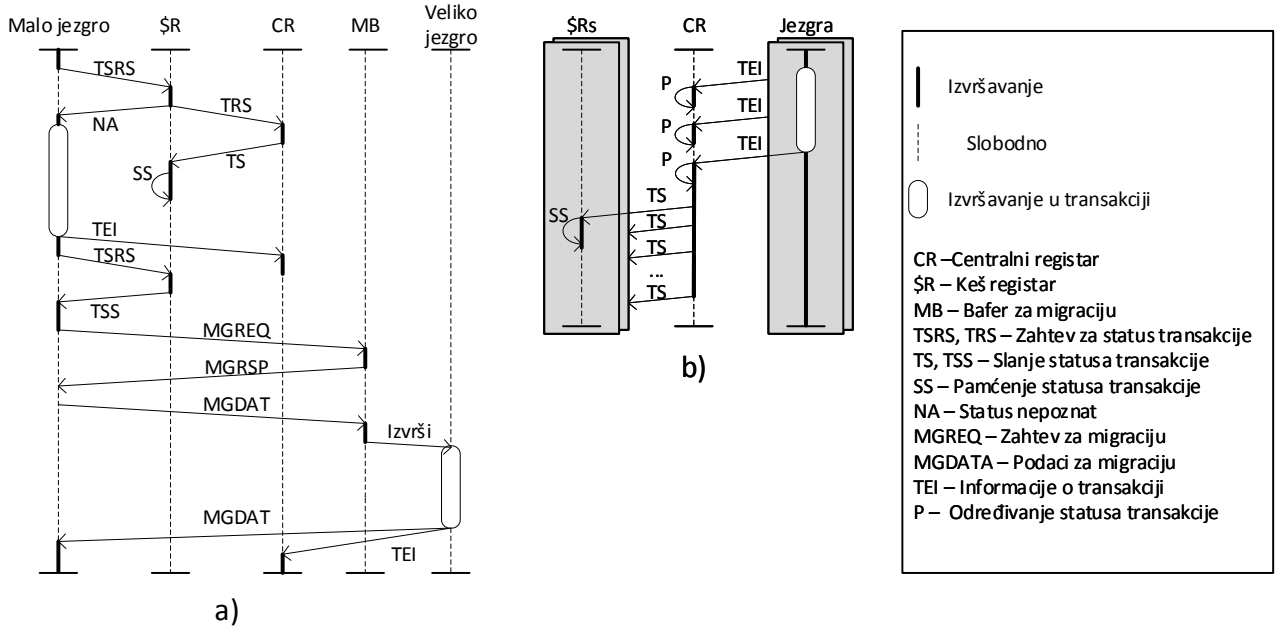
poruka pored svih podataka malo jezgro mora da pošalje i svoj identifikator da bi veliko jezgro znalo kome da odgovori. Taj identifikator mora da se čuva i u baferu za migraciju, jer se odgovor malom jezgru šalje kada se transakcija izvrši.

U toku izvršavanja transakcije sva jezgra broje koliko je dugačka transakcija u instrukcijama. Pseudo kod koji oslikava šta se radi prilikom završetka transakcije dat je na listinzima algoritama 4 i 5, za veliko i malo jezgro. Kada jezgro treba da izvrši instrukciju za kraj transakcije (bilo potvrđivanja ili poništavanja), jezgro šalje poruku o završetku (TEI) centralnom registru. Poruka sadrži status transakcije, razlog za neuspeh (ako je poništena), dužini transakcije, identifikator transakcije i informacije o tipu jezgra (veliko ili malo). Odgovor na ovu poruku nije potreban, pa jezgro odmah može da nastavi dalje izvršavanje. Stoga, poruka o završetku ne usporava rad jezgra. Prilikom završetka malo jezgro dodatno postavlja bit za lokalno blokiranje migracije. Ako je transakcija potvrđena, bit se postavlja na aktivnu vrednost, u suprotnom na neaktivnu. Kao što je već rečeno, ako je bit za blokiranje postavljen na aktivnu vrednost, transakcija se ne migrira bez obzira na stanje istorije izvršavanja transakcije.

U okviru algoritma 4 veliko jezgro može da pošalje malom jezgru neku poruku. Da li će veliko jezgro poslati neku poruku zavisi od toga koju transakciju izvršava. Ako izvršava svoju transakciju, ne šalje nikakvu poruku. Ako izvršava migriranu transakciju šalje poruku malom jezgru (ili MGRSP ili MGDAT).

Slika 5.3a) prikazuje koje se poruke razmenjuju i akcije koje se vrše da bi se migrirale transakcije. Dva slučaja su prikazana. Prvi prikazuje slučaj kada u keš registru nije prisutna informacija o transakciji. Tada keš registar zahteva informaciju od centralnog registra, a malo jezgro u paraleli izvršava transakciju. Drugi slučaj je kad je ta informacija prisutna u keš registru i transakcija treba da se migrira. Migriranje transakcije je uspešno i transakcija se uspešno izvršava.

Odluka da li transakcija treba da se migrira se donosi u centralnom registru i algoritam 2 je tu implementiran. Kada primi poruku o završetku transakcije, centralni registar ažurira stanje istorije transakcije i odlučuje da li transakciju treba migrirati. Slika 5.3b) prikazuje poruke koje se razmenjuju i radnje koje se vrše u tom trenutku. Pošto se informacije o transakcijama čuvaju i u keš registrima malih jezgara, moguće je da mala jezgra nemaju najsvežiju informaciju. Bajata informacija ne utiče na



Slika 5.3: *Poruke koje razmenjuju jezgra i pridružene jedinice*

ispravnost izvršavanja programa, već samo može uticati na performanse izvršavanja. Kada se promeni informacija o tome da li treba migrirati transakciju, centralni registar šalje poruku o statusu transakcije (TS) svim keš registrima, da bi ih obavestio o najnovijoj promeni. Ona mala jezgra koja imaju informaciju o toj transakciji u svom keš registru, upisaće novu informaciju. Druga će samo da ignorišu poruku. Ovaj način dostave najnovije informacije može zahtevati veliki protok preko komunikacione mreže, koji odavno postoji u dostupnim komercijalnim procesorima [126]–[128].

Za čuvanje evidencije o uspešnosti, razlozima poništavanja i dužini transakcije potrebno je pamtiti najskorije događaje. Vođenje evidencije je implementirano na isti način za sve podatke, osim za dužinu transakcije. Pošto promena istorije stanja dovodi do slanja poruka svim jezgrima, treba imlementirati vođenje evidencije tako da ne dolazi do promene stanja istorije na svaki događaj, da bi se smanjio broj poruka. U suprotnom komunikaciona mreža bi mogla biti zatrpana porukama i performanse izvršavanja bi mogle biti degradirane. Jedan način da se to uradi je da se obezbedi histerezis prilikom određivanja stanja istorije (pogledati algoritam 2). To se može postići sa po jednim brojačem za svaki podatak. Brojači su ograničeni i mogu da broje od minimalne i do maksimalne vrednosti. Kada je vrednost brojača maksimalna, brojač više ne reaguje na događaje koji mu povećavaju vrednost, a kada je vrednost minimalna brojač više ne reaguje na događaje koji mu smanjuju vrednost.

Stanje istorije se menja samo kad vrednost brojača dostigne minimalnu ili maksimalnu vrednost. Recimo brojač za uspešnost transakcije ima opseg 4, trenutna vrednost je minimalna i stanje istorije određuje da se transakcija ne izvršava uspešno. Jedan mogući sled događaja koji bi doveo do promene stanja istorije bi bio: transakcija se dvaput potvrdi, jedanput poništi i onda triput potvrdi.

Opseg brojača može da bude konfigurabilan, pa da se pre izvršavanja svake aplikacije postavi odgovarajuća vrednost. U predloženom rešenju uzeta je ista i fiksna vrednost za sve brojače, jer promenljivi opseg zahteva analizu aplikacija koja prevazilazi okvire ovog istraživanja.

Predložena implementacija ne zahteva preterano prostora na čipu. Centralni registar i keš registar mogu da sadrže informacije o ograničenom broju transakcija. Ako u registru nema mesta, vrši se zamena po aproksimativnom algoritmu najdavnije korišćen (eng. *least recently used*). Neki od ulaza u registru može biti prazan, pa je potrebno za svaki ulaz voditi evidenciju da li je validan pomoću jednog bita. Ako centralni registar ima 32 ulaza, a keš registar za svako malo jezgro ima 8 ulaza, potreban prostor je manji od 3KB, za višejezgarni procesor sa 32 jezgra. Jedan ulaz u baferu za migraciju MB sadrži kontekst niti koji se sastoji od 32 registra širine 8 bajtova, adrese transakcije

i broja jezgra, što je oko 0.25KB. U predloženoj implementaciji je postavljeno da je broj ulaza u MB bafer 4, pa je potreban prostor oko 1KB. Sveukupno za procesor sa 32 jezgra potrebno je 4KB memorije.

5.2 Prilagođeni podsistem keš memorija

Predloženo rešenje zahteva izmenu samo podsistema keš memorija, dok se implementacija procesora ne menja i može biti proizvoljna. Implementacija transakcione memorije može biti proizvoljna. Isto rešenje se može primeniti i na asimetrični i na simetrični višejezgarni procesor. Algoritam i implementacija bazirani su na već predloženim rešenjima [1], [111]–[113]. Predloženo rešenje modifikuje postojeća rešenja tako da podržava rad na višejezgarnom procesoru, koji ima podršku za transakcionu memoriju i podaci mogu da promene keš memoriju prilikom promašaja u najnižem nivou podsistema keš memorija.

Podsistem keš memorija mora biti organizovan tako da postoje privatne keš memorije za pojedinačna jezgra i deljenje keš memorije najvišeg nivoa. Broj nivoa privatnih keš memorije nije važan, ali u predloženom rešenju, a i u evaluaciji predloženog rešenja je uzeto da je broj nivoa dva. Moderni višejezgarni procesori koji su trenutno dostupni najčešće imaju takvu organizaciju, sa dva privatna nivoa i jednim deljenim nivoom.

Predloženo rešenje se u potpunosti implementira u hardver. Za njegov rad nije potrebno menjati kod aplikacije ni na koji način. Moguće je primeniti isti algoritam i za vreme prevođenja i prilikom profilisanja aplikacije [1], [112]. Time bi se odmah, na početku izvršavanja, imala predistorija pristupa podacima. Pomoću predistorije može se izbeći problem "hladnog starta". Čak i ta softverska realizacija ne zahteva promenu izvornog koda.

5.2.1 Opis algoritma za rad podeljenje keš memorije i algoritma za keš koherenciju

Organizacija privatnih keš memorija mora biti takva da se sastoje iz dva dela. Prvi deo je organizovan na konvencionalni način, tj. širina jednog ulaza je nekoliko reči (tipično četiri ili osam). Drugi deo je organizovan tako da je širina jednog ulaza samo jedna reč. Za prvi deo korišćen je izraz prostorni deo, dok je za drugi deo korišćen izraz vremenski deo. Prostorni deo, služi za smeštanje podataka kojima se pristupa pretežno sa prostornom lokalnošću, dok vremenski deo služi za čuvanje podataka kojima se pristupa pretežno sa vremenskom lokalnošću. Organizacija deljenih keš memorija treba da bude konvencionalna, tj. sastoji se samo iz jednog dela u kome se čuvaju obe vrste podataka.

Lokalnost pristupa podacima se procenjuje na osnovu istorije pristupa. Evidencija o istoriji se vodi za svaki pojedinačan blok u keš memoriji najnižeg nivoa. Pod pojmom blok ovde se podrazumeva blok koji bi se koristio u konvencionalnim keš memorijama. Drugim rečima, smatra se da je memorija podeljena u blokove fiksne veličine (obično četiri ili osam reči). Za sve podatke u jednom bloku se vodi jedna evidencija i za sve njih se generiše ista procena koja lokalnost se ispoljava prilikom pristupa. Ovakav pristup je odabran da cena implementacije (u vidu prostora na čipu) ne bi bila previsoka.

Prilikom svakog pristupa beleži se kojem delu bloka se pristupilo. Uzeto je da se pristupi prate na nivou polovine bloka, da cena implementacije ne bi bila previsoka. Broji se koliko puta se pristupilo svakom delu bloka. Nakon nekog vremena proverava se kako se pristupalo delovima bloka. Ako su pristupi pretežno bili u okviru samo jednog dela, smatra se da podaci u tom bloku ispoljavaju vremensku lokalnost. U suprotnom, smatra se da podaci u tom bloku ispoljavaju prostornu lokalnost. Kada treba za neki blok izvršiti procenu o lokalnosti, ona se radi pomoću brojača. Taj brojač beleži koliko puta se pristupilo tom bloku od prethodne procene. Na taj način procena se ne radi često, pa samim tim podaci ne menjaju često deo keš memorije u kojem treba da se pamte.

Algoritam 6 Pseudo kod za procenu lokalnosti

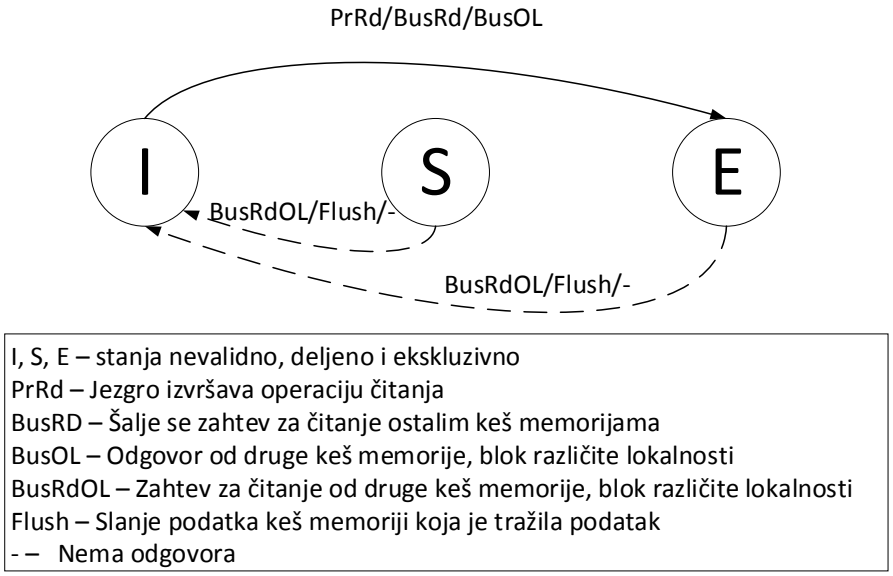
```
1:  procedure LOCALITY_TAGGING(block, block_part)
2:      if −X < block.x < +X then
3:          if block_part = HIGH then
4:              block.x ← block.x + 1
5:          else
6:              block.x ← block.x − 1
7:          end if
8:      end if
9:      if block.y < Y then
10:         block.y ← block.y + 1
11:     else
12:         if −X < block.x < +X then
13:             block.tag ← SPATIAL
14:         else
15:             block.tag ← TEMPORAL
16:         end if
17:         block.x ← 0
18:         block.y ← 0
19:     end if
20: end procedure
```

Pseudo kod za dati algoritam prikazan je na listingu algoritma 6. U pseudo kodu brojač za evidenciju kom delu bloka se pristupalo je označen imenom **x**, dok je brojač za određivanja trenutka kada se vrši procena lokalnosti označen sa **y**. Procena kom delu bloka se pristupilo radi se tako što se u brojaču **x** povećava vrednost kada se pristupa višem delu bloka, dok se u suprotnom smanjuje. Ako brojač **x** dostigne pozitivnu ili negativnu granicu on se tu zaustavlja, sve do sledeće procene lokalnosti. Dostizanje granice vrednosti brojača **x** znači da se uglavnom pristupa samo jednom delu bloka i da podaci pretežno ispoljavaju vremensku lokalnost. Vrednost u brojaču **y** se povećava prilikom svakog pristupa. Kada vrednost brojača **y** dostigne granicu, vrši se procena lokalnosti na osnovu vrednosti brojača **x**. Tom prilikom se i brojači **x** i **y** postavljaju na početnu vrednost. Početna vrednost za brojače je nula.

Granica za brojač **x** određuje koji uslov mora da ispuni jedan blok da bi se označio da ima vremensku lokalnost. Što je vrednost granice niža, to će više blokova biti označeno da imaju vremensku lokalnost. Ako je vrednost granice visoka, malo ili nijedan blok neće biti označen da ima vremensku lokalnost, što bi dovelo da se vremenski deo keš memorije ne koristi dovoljno. Ako je vrednost granice niska, biće previše blokova označenih da imaju vremensku lokalnost, što bi dovelo do prepunjenosti vremenskog dela keš memorije. Ta popunjenost bi dovela do velikog broja promašaja usled ograničenog kapaciteta i mogla bi da dovede do smanjena performansi izvršavanja. Vrednost granice mora biti veća od polovine broja reči u bloku. U suprotnom, prilikom sekvencijalnog pristupa, koji ispoljava prostornu lokalnost, blok bi bio označen da ima vremensku lokalnost, jer bi brojač sigurno dostigao granicu i ostao nepromenjen do sledeće procene lokalnosti.

Treba primetiti da optimalna vrednost granice za brojač **x**, zavisi od šeme pristupa podacima koje ima aplikacija i od veličine vremenskog dela keš memorije. Najbolje bi bilo da parametar bude konfigurabilan i da se postavi na početku izvršavanja aplikacije. U tom slučaju parametar bi morao da bude deo konteksta niti koji se čuva prilikom promene konteksta. Ta promena bi zahtevala promenu postojećih aplikacija, operativnog sistema i instrukcijskog seta, pa je u ovom istraživanju uzeto da je vrednost granica fiksna.

Procesor prilikom pristupa nekom podatku šalje zahtev za čitanje i vremenskom i prostornom delu



Slika 5.4: *Izmene koje treba dodati u protokol koherencije*

keš memorije najnižeg nivoa. Samo jedan od njih treba da odgovori, jer jedan blok može biti obeležen da ima ili vremensku ili prostornu lokalnost. Drugim rečima blok ne može biti u oba dela istovremeno. Pošto se ovaj algoritam primenjuje u višejezgarnom procesoru, blok može biti samo u jednom delu keš memorije u okviru svih jezgara. Ukoliko jedan deo ima podatak, šalje poruku drugom delu iste keš memorije da je zahtev opslužen, da taj deo ne bi poslao zahtev za dohvatanje podataka višem nivou keš memorije.

Blok može da promeni tip lokalnosti u toku izvršavanja programa (npr. u različitim fazama podacima se pristupa na različiti način). Promena lokalnosti može da se dogodi samo u trenutku kada se bloku pristupa, što znači da blok (ili deo bloka, ako je blok označen da ima vremensku lokalnost) mora biti prisutan u keš memoriji najnižeg nivoa. Premeštanje bloka odmah nakon promene tipa lokalnosti iz prostorne u vremensku nema smisla, jer se svakako događa pogodak u keš memoriji i bilo kakva dodatna akcija bi mogla samo da uspori izvršavanje. Promena lokalnosti iz vremenske u prostornu znači da se procenjuje da će uskoro biti pristupljeno nekom drugom delu tog bloka. Sa stanovišta tog budućeg pristupa, najbolje je blok premestiti u prostorni deo keš memorije. Međutim, to bi zahtevalo da se svi delovi bloka izbace iz vremenskog dela keš memorije prvog i drugog nivoa, pa da se tek onda dovuče ceo blok. S obzirom na to da je u pitanju višejezgarni procesor, delovi tog bloka mogu da budu u drugim keš memorijama što dodatno usporava dovlačenje celog bloka. Iz tih razloga algoritam ne radi premeštanje iz jednog dela u drugi deo keš memorije kada se detektuje promena lokalnosti, već onda kada ni jedan deo bloka nije prisutan u keš memoriji prvog nivoa.

Dozvoljeno je da delovi jednog bloka, koji su označeni da imaju vremensku lokalnost, nađu u više vremenskih delova različitih jezgara. Time se postiže da deo bloka bude u nekom stanju nezavisno od ostalih delova bloka. Postiže se mogućnost da dva jezgra upisuju paralelno u dva različita dela istog bloka, čime se povećava mogućnost paralelnog rada i poboljšanja performansi. Dodatno, zbog ove mogućnosti, u slučaju da deo bloka prelazi iz jednog jezgra u drugo, prenosi se samo taj deo bloka. Prilikom takvog prenosa, podatak se čita samo iz vremenskih delova keš memorije (koji imaju kraće vreme pristupa), pa je prenos podataka brži. Takođe, pošto se prenosi samo deo bloka, a ne ceo blok, manje je zagušenje na komunikacionoj mreži koja spaja keš memorije. Ovakav pristup zahteva da se prilikom dovlačenja bloka, kojem je prethodno promenjena lokalnost iz vremenske u prostornu, sakupe svi delovi bloka koji su modifikovani u vremenskim delovima keš memorija. To usporava dovlačenje tog podatka, jer je kod konvencionalnih keš memorija potrebno da samo jedna keš memorija pošalje blok.

Ovaj predlog podsistema keš memorije nema velike zahteve prilikom odabira protokola keš ko-

herencije. Važno da protokol ima deljeno i modifikovano stanje bloka (kao protokoli MESI [129] i MOESI[130]), jer se time dobijaju najveći benefiti sa predloženim rešenjem u višejezgarnim procesorima. Većina protokola koji se danas koriste imaju ta stanja. Stanja i mašina stanja protokola keš koherencije ne treba da se značajnije menjaju. I delovi blokova koji su označeni da imaju vremensku lokalnost, mogu da budu u istim stanjima kao i celi blokovi koji su označeni da imaju prostornu lokalnost. Njihova promena stanja zavisi na isti način od vrste pristupa (čitanje ili upis) kao i kod celih blokova. Promena jedino mora da se uvede u slučaju kada se menja deo keš memorije u kome treba da se nađe blok (ili deo bloka). Na primer, ako se blok nalazi u prostornom delu keš memorije jednog jezgra, drugo jezgro nema taj podatak u svojoj keš memoriji i pokušava čitanje, a došlo je do promene lokalnosti tog bloka. Tada po standardnom protokolu keš koherencije deo bloka treba da se dovuče u deljenom stanju, jer taj blok postoji u istom stanju u drugoj keš memoriji. Međutim, predloženo rešenje ne dozvoljava postojanje istog bloka u dva različita dela keš memorije, odnosno može biti ili u prostornom ili u vremenskom delu. Iz tog razloga, ona keš memorija koja ima blok u neodgovarajućem delu, treba da ga poništi. Ona keš memorija koja dobije blok treba da mu dodeli ekskluzivno stanje (ako protokol ne podržava to stanje, onda odgovarajuće, recimo deljeno), jer je u svim ostalim keš memorijama blok poništen.

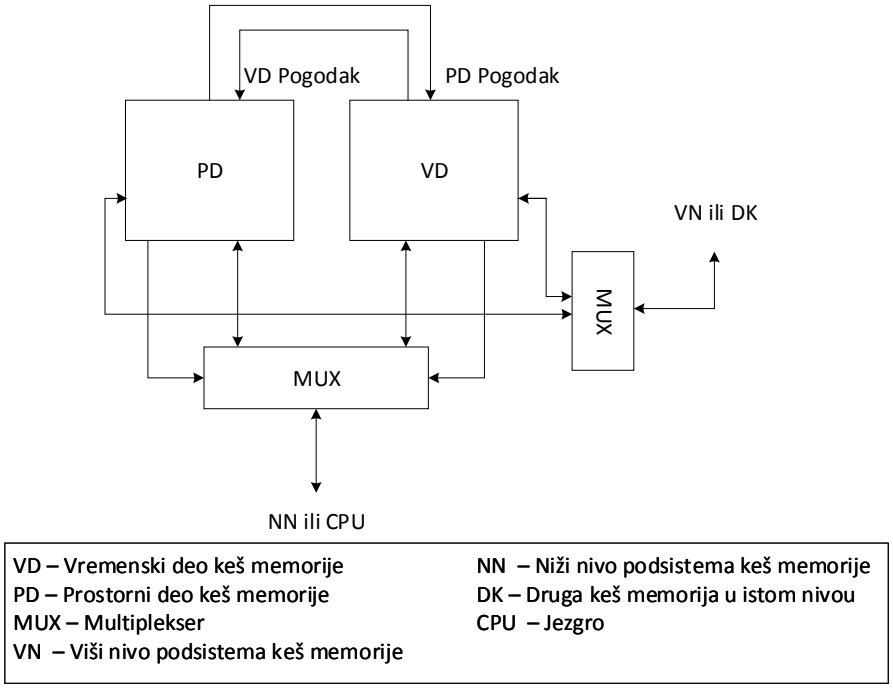
Na slici 5.4 su prikazane promene koje je potrebno ugraditi u mašinu stanja protokola keš koherencije. Prikazan je samo deo mašine stanja gde treba dodati promene. Promene su samo novi prelazi između stanja. Stanja su na slici prikazani kao krugovi, dok strelice između njih predstavljaju promenu stanja u zavisnosti od operacija. Pored strelica su prikazane operacije kao i poruke koje razmenjuju keš memorije. Prva poruka je zahtev ostatku podsistema keš memorija, dok je druga poruka odgovor koji je stigao zbog prve poruke.

5.2.2 Implementacija podeljenih keš memorija u višejezgarnom sistemu

Predloženo rešenje zahteva da keš memorije imaju dva dela. Svaki od ta dva dela implementiran je kao obična memorija na čipu. Razlika je samo u veličini memorija tih delova i načinu organizacije keširanja. Prostorni deo organizovan je kao konvencionalna keš memorija. Ulaz je veličine jednog bloka, dok mapiranje set-asocijativno. Veličina skupa za set-asocijativno mapiranje može biti podešena tako da se dobije željena brzina pristupa. Vremenski deo organizovan je tako da veličina ulaza bude jedna reč iz bloka, dok je mapiranje direktno. Na taj način dobija se veća brzina pristupa. Broj ulaza može biti podesiv. U okviru evaluacije su razmatrane implementacije sa različitim veličinama vremenskog dela i različitim veličinama skupova prostornog dela.

Dva dela keš memorije treba da se ponašaju prema ostatku sistema kao da se radi o jednoj jedinstvenoj keš memorije. Implementacija podeljenje keš memorije je prikazana na slici 5.5. Organizacija je ista za sve privatne keš memorije. Svaki od dva dela keš memorije prima zahteve od jezgra ili druge keš memorije. Onaj deo u kome se nalazi podatak ili u njemu treba da se nalazi podatak generiše signal da njemu pripada podatak. U slučaju da se dogodio pogodak, deo šalje podatak na izlaz. Podatak samo od jednog dela se propušta. Za propuštanje su zaduženi multiplekseri koji za signal selekcije imaju signal koji generiše deo keš memorije kome podatak pripada. U slučaju da se dogodio promašaj, deo kome pripada podatak generiše zahtev ostatku podsistema keš memorije za dohvatanje. Zahtev takođe propuštaju multiplekseri sa istim signalom za selekciju kao i za propuštanje podataka.

Vremenski deo keš memorije može da odredi da podatak treba da se nalazi u njemu, čak iako sam podatak nije prisutan. To može da se odredi na osnovu ostalih podataka u vremenskom delu keš memorije. Ako se neki drugi deo bloka nalazi u vremenskom delu, to znači da je pretpostavljena vremenska lokalnost za taj blok, pa bilo koji podatak iz bloka treba da bude u vremenskom delu. Pošto je vremenski deo sa direktnim mapiranjem, ostali podaci iz bloka mogu da se nađu samo u nekih sedam lokacija (pretpostavka je da blok ima osam reči). Provera da li se u tih sedam reči nalazi neka reč iz istog bloka radi se pomoću istog mehanizma kao i provera da li sam podatak nalazi u memoriji (poređenjem tražene adrese i taga). Razlika je samo što se u ovom slučaju ne porede svi biti

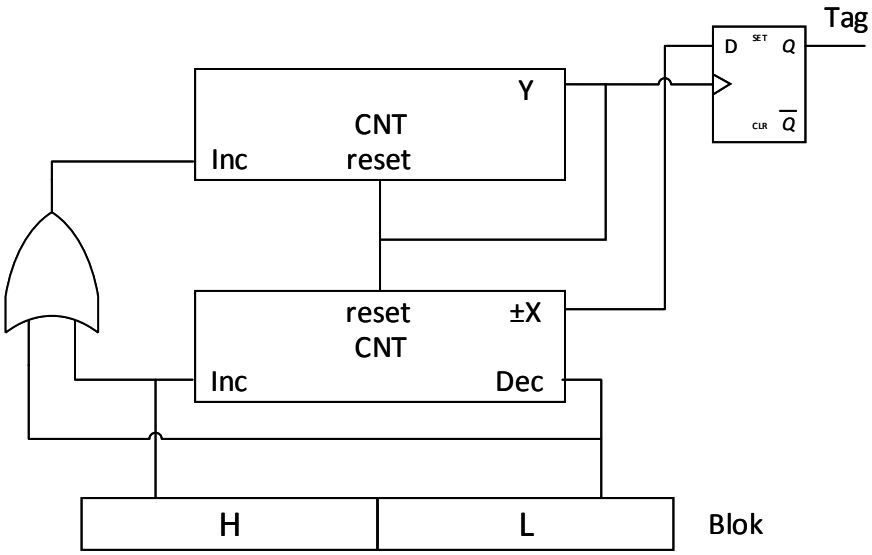


Slika 5.5: *Implementacija keš memorije sa dva dela*

od značaja. Ne porede se poslednja tri bita, jer se ne traži određena reč iz bloka, već bilo koja reč. To se može uraditi tako što se jedan komparator za pronalaženje određene reči podeli na dva. Jedan poredi najnižih tri bita, dok drugi poredi ostale bite. Za pronalaženje određenje reči bloka koriste se oba komparatora. Određivanje rezultata radi se pomoću samo jednog kola logičkog I. Za pronalazak bilo koje reči bloka koristi se samo drugi komparator. Izmena koja je potrebna je mala, pa ne bi trebalo da poveća vreme pristupa keš memoriji.

Može se dogoditi da nijedan deo keš memorije nema informaciju da u njemu treba da se nađe podatak. To se dešava kada nijedan deo tog bloka nije u vremenskom delu niti ceo blok nije u prostornom delu. Tada oba dela generišu zahtev ostatku podsistema keš memorija i propušta se bilo koji zahtev. Odgovor koji pristigne sadrži i koje lokalnosti je poslati podatak. Ta informacija se koristi kao signal selekcije multipleksera da bi se podatak smestio u odgovarajući deo keš memorije. Keš memorija koja odgovaranja na zahtev dodaje informaciju o lokalnosti za podatak koji šalje. Ako odgovor šalje privatna keš memorija, ona ima informaciju o lokalnosti. Deljena keš memorija nema dva dela pa nema tu informaciju. U deljenoj memoriji se nalazi jedan dodatan bit za svaki blok, koji opisuje koja je lokalnost određena za taj blok. Ta informacija se dobija iz poslednje poruke koju je poslala neka keš memorija nižeg nivoa. Poruka može biti informacija o zameni nekog podatka ili obaveštenje o upisu u neki podatak. To možda nije najsvežija informacija pošto svako jezgro nezavisno od ostalih procenjuje lokalnost jednog bloka, ali to ne utiče na logičku ispravnost izvršavanja, već samo može uticati na performanse izvršavanja.

Implementacija algoritma čiji je pseudo kod dat na listingu 6 zahteva dva brojača za svaki blok (ili deo bloka) koji je prisutan u keš memoriji prvog nivoa. Na slici 5.6 je prikazana detaljna implementacija evidencije za jedan blok. Brojač **x** ima mogućnost inkrementiranja i dekrementiranja, dok brojač **y** ima samo mogućnost inkrementiranja. Oba brojača imaju samo jedan jednobitni izlazni signal, koji ima aktivnu vrednost kada brojač dosegne granicu. Brojač ne broji preko granice. Koja operacija sa brojačima se vrši zavisi od adrese kojoj je pristupano. Ako se adresa nalazi u gornjoj polovini bloka, aktivira se inkrementiranje, u suprotnom dekrementiranje, brojača **x**. Uvek se vrši inkrementiranje brojača **y**. Kada brojač **y** dostigne granicu, aktivira D flip-flop koji pamti procenu lokalnosti. D ulaz je rezultat brojača **x**, koji je na aktivnoj vrednosti samo ako je brojač **x** dostigao granicu. Aktivan vrednost brojača **y** poništava oba brojača na nulu.



CNT – Brojač koji geneše aktivnu vrednost kad dosegne limit
X, Y – Limiti za brojače
H, L – Pristup gornjoj polovini ili donjoj polovini bloka
Tag – Procena lokalnosti sa kojom se pristupa bloku

Slika 5.6: *Implementacija algoritma za određivanje lokalnosti*

Pošto se blok ne prebacuje iz jednog dela keš memorije u drugi, D flip flop se ne čita prilikom pristupa bloku. Da bi se prosledila informacija o lokalnosti koja je određena za jedan blok, prilikom izbacivanja bloka iz keš memorije šalje se i najnovija informacija o lokalnosti ostatku podsistema keš memorija. Viši nivoi podsistema keš memorije moraju imati jedan bit koji će služiti za čuvanje informacije o određenoj lokalnosti za svaki blok koji se nalazi u keš memorijama. Da bi prenos te informacije bio moguć između različitih nivoa podsistema keš memorija, taj podsistem mora biti inkluzivan. U suprotnom, u višem nivou keš memorije neće biti prostora za smeštanje te informacije.

Prilagođenje podsistema keš memorije zahteva da se informacija o lokalnosti čuva u svim nivoima memorijskog sistema. Problematično može biti čuvanje te informacije u operativnoj memoriji koja je obično sinhrona dinamička memorija sa proizvoljnim pristupom dvostruke brzine prenosa podataka (eng. *Double Data Rate Synchronous Dynamic Random-Access Memory*). Način za promenu takve memorije prevazilazi okvire ovog rada, a potrebna je, jer uz svaku memorijsku reč mora da se čuva i jedan dodatan bit za informaciju o lokalnosti. Iz tog razloga u predloženom rešenju se ta informacija čuva u podsistemu keš memorije. Ukoliko operativna memorija ima i prostor za kodove za korekciju greške (eng. *error-correcting codes*) kao u [131], ti dodatni bitovi se mogu iskoristi za čuvanje informacije o lokalnosti bez potrebe za menjanjem organizacije same operativne memorije. Time bi se žrtvovala otpornost na otkaze zbog poboljšanja performansi izvršavanja.

Implementacija protokola keš koherencije zahteva dve promene u odnosu na konvencionalan protokol keš koherencije. Prva je izmena poruka tako da može da se implementira mašina stanja koja je opisana u prethodnoj sekciji 5.2.1. Druga je izmena načina slanja i primanja podataka, koji u predloženom rešenju mogu biti na dve širine (ili na širini od jedne reči ili jednog bloka). Prva promena dodaje samo tri nova prelaza između postojećih stanja, pa može da stvori probleme u najkritičnijem delu implementacije protokola keš koherencije, a to je verifikacija ispravnosti implementacije [132]. Druga promena zahteva promenu implementacije samih keš memorija, koje se ionako menjaju i koje mogu da se verifikuju nezavisno od drugih keš memorija. Prilikom opisa potrebnih promena pretpostavka je da je osnovni protokol keš koherencije baziran na katalogu, pošto se taj tip protokola češće koristi u višejezgarnim procesorima sa većim brojem jezgara [19].

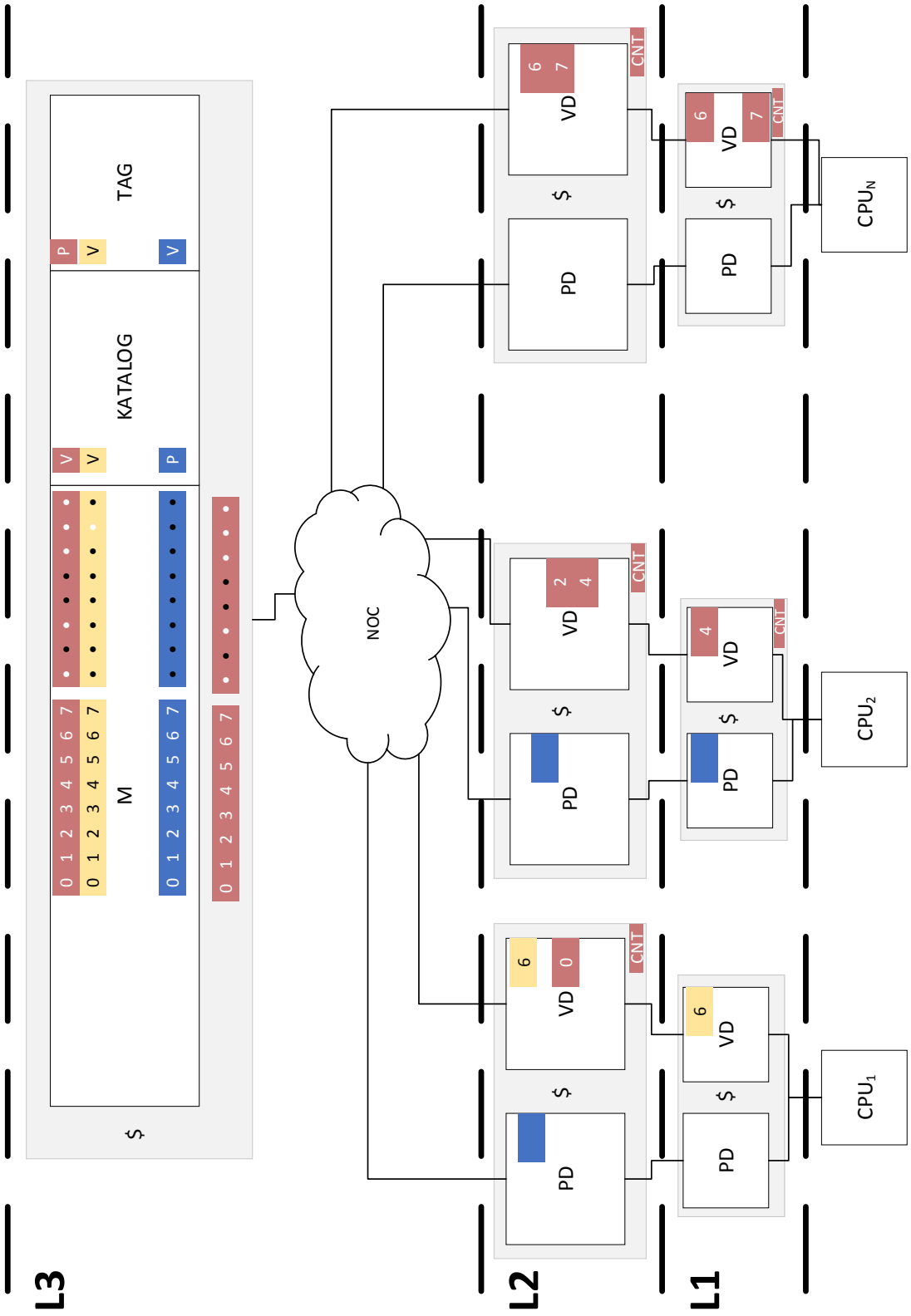
Potrebno je da svaka poruka koju keš memorije razmenuju u okviru protokola keš koherencije ima informaciju da li se radi o pristupu prostornom ili vremenskom delu. Na taj način deo keš memorije može da detektuje da li je zahtev došao od druge vrste dela keš memorije. Ako je došao od druge vrste, onda treba da promeni stanje na način prikazan na slici 5.4. Takođe, može da pošalje odgovarajući odgovor sa informacijom o tipu lokalnosti, da bi i keš memorija koja šalje zahtev mogla da promeni stanje podatka na ispravan način. Ima slučajeva kada keš memorija šalje zahtev o bloku za koji lokalnost nije poznata, potrebno je dodati i informaciju da li je lokalnost poznata. Takođe, potrebno je da svaka poruka ima i potpunu adresu reči kojoj se pristupa. Adresa reči je potrebna da bi podatku u vremenskom delu keš memorije moglo da se pristupi i da mu se promeni stanje na odgovarajući način. Prostorni deo treba da odseče deo adrese, tj. da koristi samo one bite koji predstavljaju adresu bloka. Tu adresu bloka koristi na konvencionalan način. Proširena adresa i informacija o tipu lokalnosti zahtevaju dodatna pet bitova (tri za proširenu adresu, jedan za informaciju o lokalnosti i jedan za informaciju da li je lokalnost poznata).

Događa se da keš memorija prosleđuje zahtev koji je dobila od keš memorije nižeg nivoa. Može da se dogodi da u zahtevu nema informacije o lokalnosti, a da keš memorija koja prosleđuje ima tu informaciju. U tom slučaju prilikom prosleđivanja zahteva postoje dva slučaja. Jedan slučaj je da se informacija o lokalnosti ne doda, a drugi je da se doda. Prvi slučaj je kada zahtev dobije keš memorija drugog nivoa. To je slučaj da keš memorija prvog nivoa nema nijedan deo bloka (jer nema informaciju o lokalnosti). Moguće je da se procena lokalnosti promenila. Zbog toga keš memorija drugog nivoa treba da pošalje keš memoriji trećeg nivoa zahtev bez informacije o lokalnosti i da na osnovu odgovora koji tu informaciju sadrži nastavi dalji rad. Drugi slučaj je kada takvu poruku primi keš memorija trećeg nivoa. Ona treba da doda informaciju o lokalnosti ako prosleđuje zahtev nekoj drugoj keš memoriji (to je slučaj kada je najsvežija verzija podatka u nekoj drugoj keš memoriji).

Način na koji keš memorije primaju i šalju podatke u većini slučajeva implementira se na isti način kao i kod konvencionalnog pristupa. Razlikuje se samo u tri slučaja, koja ni ne postoje kod konvencionalnog pristupa. Prvi slučaj je kad keš memorija menja deo keš memorije u kojoj se podatak nalazi i to tako što blok učitava u prostorni deo, a prethodno je bio u vremenskom delu. Drugi slučaj je kada zajednička keš memorija čuva informaciju o delovima bloka koji su izbačeni iz vremenskih delova zbog ograničenog kapaciteta keš memorija. Treći slučaj je kad keš memorija menja deo keš memorije u kojoj se podatak nalazi i to tako da se blok učitava u vremenski deo, a prethodno je bio u prostornom delu. Na slici 5.7 je prikazana organizacija celokupnog sistema keš memorija. Za svaki od slučajeva prikazano je stanje jednog bloka i potrebne jedinice za realizaciju tih promena. Svaki od navedenih slučajeva detaljno je opisan u nastavku.

Prvi slučaj na slici 5.7 je označen crvenom bojom. Za svaki deo bloka napisan je redni broj te reči u bloku. Delovi jednog bloka mogu da se nađu u svim privatnim keš memorijama, osim u onoj koja traži podatak. Pošto se promenila procena lokalnosti, ceo blok treba da se učita u prostorni deo. Sve druge privatne memorije moraju da izbace iz svojih vremenskih delova podatke koji pripadaju tom bloku, čak i privatna keš memorija drugog nivoa koja pripada istom jezgru. To se radi pomoću brojača koji broji od nula do sedam i generiše sve adrese reči koje pripadaju tom bloku. Za svaku adresu se vrši izbacivanje te reči iz vremenskog dela. Ovo izbacivanje može da se radi i paralelno, ukoliko memorija ima više portova za upis. Pretpostavka je da memorija ima najmanje moguće vreme pristupa, ako se koristi samo jedan port. Ukoliko tehnologija dozvoljava da memorija ima više portova, bez većeg uticaja na vreme pristupa, rezultati eksperimenata su dali lošije rezultate nego što će oni biti u takvom slučaju.

Prilikom promene lokalnosti u prvom slučaju potrebno je prikupiti sve reči iz jednog bloka i ceo blok poslati keš memoriji koja je traži. Svaka reč može da bude u vlasništvu bilo koje keš memorije. Za skupljanje svih reči zadužena je keš memorija najvišeg nivoa. Ukoliko je reč deljena sa drugim memorijama, znači da je keš memorija najvišeg nivoa vlasnik i da ima najsvežiju vrednost. Prilikom poništavanja te reči u keš memorijama nižih nivoa, nju nije potrebno slati keš memoriji višeg nivoa. Ukoliko je keš memorija nižeg nivoa vlasnik te reči, mora se poslati keš memoriji višeg nivoa da



CPU_i – i-to jezgro	TAG – Informacije o procenjenoj lokalnosti
$ - Keš memorija	• - Bit validnosti za jednu reč
Li – i-ti nivo keš memorija	V – Vremenska lokalnost
PD – Prostorni deo	P – Prostorna lokalnost
VD – Vremenski deo	NOC – Komunikaciona mreža
M – Memorijski deo keš memorije	CNT – Trobitni brojač reči u bloku

Slika 5.7: *Organizacija podeljenog keš podsistema u višejezgarnom procesoru*

bi se sačuvala najsvežija vrednost. Tim postupkom se reči iz jednog bloka prenose do keš memorije najvišeg nivoa, koja ih skuplja i stavlja u jedan blok. To radi tako što ima jedan registar širine jednog bloka, gde na odgovarajuće mesto smešta primljenu reč. Pored tog registra ima i osam bitova gde pamti koje su reči primljene. U prikazanom primeru, keš memorija trećeg nivoa je već pripremila te registre i bite za prihvatanje reči iz ostalih keš memorija, dok keš memorije prva dva nivoa još nisu počela da izbacuju svoje reči. Keš memorija najvišeg nivoa treba da doda reči, koje su u njenom vlasništvu, u blok za slanje. Kada primi sve reči šalje blok onoj keš memoriji drugog nivoa koja je tražila taj blok. U katalogu keš memorija najvišeg nivoa beleži da je sada taj blok u vlasništvu one keš memorije kojoj je poslala taj blok.

Drugi slučaj na slici 5.7 je označen žutom bojom. Za svaki deo bloka napisan je redni broj te reči u bloku. Dok se delovi bloka nalaze u vremenskim delovima keš memorija, svaka keš memorija može da bude vlasnik jednog dela. O tome mora da se vodi evidencija u keš memoriji najvišeg nivoa, tj. u katalogu celog podsistema keš memorija. Da bi se smanjila potrebni kapaciteti za smeštanje informacija o blokovima, gde se koji blok nalazi čuva se na konvencionalni način. Samo se pamti u kojim keš memorijama se nalazi blok, a ne pamti se pojedinačna reč. Na taj način prilikom pristupa jednoj reči može da se generiše više poruka nego što je stvarno potrebno, ali to samo utiče na performanse, a ne i na logičku ispravnost izvršavanja. Ono što mora dodatno da se čuva je informacija o rečima za koje je vlasnik keš memorija najvišeg nivoa, pošto je ona konvencionalna i nema posebno vremenski deo. Za podatke se koristi ista ona memorija kao i slučaju da se pamti ceo blok. Dodatno se za svaku lokaciju čuva još osam bitova koji označavaju da li je zapamćena reč validna. Ako jeste, onda je keš memorija vlasnik te reči i kod nje je najsvežija informacija. Ako nije, najsvežija reč se nalazi u nekoj keš memoriji nižeg nivoa. Prilikom čitanja iz keš memorije nema potrebe za nikakvom promenom. Ukoliko se upisuje, ceo blok prvo mora da se pročita, da se izmeni reč koja se upisuje, pa da se upiše ceo blok.

Drugi slučaj se ne nalazi na kritičnom putu, jer se dešava kada se podatak izbacuje iz keš memorije nižeg nivoa prilikom zamene, pa se operacija upisa može izvršiti tek nakon čitanja podatka koji je prouzrokovao zamenu. Ukoliko se blok nalazi u prostornom delu keš memorija nižih nivoa, onda su biti validnosti ili svi na nuli ili svi na jedinici, jer je ceo blok ili nije ili jeste u vlasništvu keš memorije najvišeg nivoa. Na prikazanom primeru blok je označen kao vremenski i samo jedna reč nije u vlasništvu keš memorije najvišeg nivo. Ta reč se nalazi u keš memorijama nižeg nivoa, gde je u modifikovanom stanju. Prilikom realizacije premeštanja bloka iz vremenskog dela u prostorni, treba da se pročita parcijalni blok iz keš memorija najvišeg nivoa zajedno sa bitovima o validnosti reči i da se ti podaci ubace u registre za sakupljanje svih reči iz bloka pre nego što se pošalje zahtev ostalim keš memorijama za sakupljanje reči. Na taj način pristigle reči mogu da se dodaju u parcijalni blok. Na pokazanom primeru za premeštanje već je pročitan blok i inicijalizovani su biti validnosti za reč. Biti validnosti za taj blok treba da se anuliraju, jer će vlasnik tog bloka postati neka druga keš memorija.

Treći slučaj je na slici 5.7 označen plavom bojom. Kada se pošalje zahtev za dohvatanje dela bloka u vremenski deo, a u ostalim keš memorijama taj blok je u prostornim delovima, sve te keš memorije treba da ponište svoje blokove. Ako je neko od njih vlasnik bloka treba da pošalje keš memoriji najvišeg nivoa ceo blok. Keš memorija najvišeg nivoa treba da inicijalizuje bite validnosti za pojedinačne reči na jedan i da pošalje reč koju je tražila keš memorija koja je pokrenula promenu. Kada pošalje tu reč, njen bit validnosti treba da se anulira, jer je sad vlasnik te reči keš memorija kojoj je ta reč poslata. Keš memorija koja primi tu reč samo treba da joj postavi stanje koje označava da je ona vlasnik te reči.

Prilikom dešavanja slučajeva jedan i tri, potrebno je da se promene u keš memorija sinhronizuju. To se radi tako što se blokiraju zahtevi za tim blokom dok se promena ne izvrši. Mehanizam za taj postupak već postoji, jer se slična stvar radi kada postoji više deljenih blokova u različitim keš memorijama, a neka keš memorija izda zahtev za upis. Dok sve keš memorije ne potvrde da su poništile blok, ne sme tom bloku da se menja stanje. To se vrši tako što katalog odlaže obradu poruka za taj blok. Keš memorija najvišeg nivoa treba da primeni isti mehanizam dok traje promena. Ono što keš

memorija mora da uradi je da detektuje promenu. Da bi detektovala promenu mora da ima informaciju u kom delu privatnih keš memorija se nalazi blok. To ne može da se odredi na osnovu bita validnosti reči, jer sve jedinice ili sve nule mogu da budu aktivne kad je blok u bilo kom delu privatnih keš memorija. Iz tog razloga potrebno je dodati još jedan bit za svaki blok u kojem bi se vodila evidencija u kom je delu privatnih keš memorija taj blok. Prilikom obrade zahteva koji nema informaciju o tipu lokalnosti, ako se bit za procenu lokalnosti ne poklapa sa bitom koji kazuje u kojem delu privatnih keš memorija se blok nalazi, dešava se promena lokalnosti. Keš memorija tada treba da blokira zahteve za promenu stanja tog bloka i da pošalje odgovarajuće poruke za poništavanje tog bloka svim keš memorijama.

Prilikom izbacivanja bloka iz keš memorije najvišeg nivoa, postupak je isti kao kod promene dela keš memorije gde se čuva blok. Razlika je samo to što se rezultujući blok šalje operativnoj memoriji, a ne keš memoriji nižeg nivoa i što se ulaz u keš memoriji najvišeg nivoa poništi za taj blok.

Potrebni resursi za implementaciju predloženog rešenja su mali. Potrošnja resursa prikazana je na primeru koji ima 32 jezgra, gde keš memorije prvog nivoa imaju 1024 ulaza (64KB), gde keš memorije drugog nivoa imaju 2048 ulaza (128KB), a keš memorija trećeg nivoa 256K ulaza (16MB). Potreban je jedan bit za procenu lokalnosti za svaki ulaz, što je 44KB. U keš memoriji trećeg nivoa potrebno je devet bitova za svaki ulaz da bi se vodila evidencija o validnosti reči u bloku, što je 288KB. U keš memorijama prvog nivoa potrebna su dva brojača širine šest bitova za procenu lokalnosti, što je 4KB. Svaka keš memorija prva dva nivoa ima po jedan trobitni brojač za izbacivanje reči iz vremenskog dela kada se poništava blok, što je 24B. Ukupno je potrebno oko 336KB dodatnog prostora. Ušteda koja može da se dobije zbog podeljenog podsistema keš memorije je oko 2MB, u zavisnosti od konfiguracije vremenskog dela. Pošto je, najveći deo dodatnog prostora, potreban za keš memoriju trećeg nivoa, koja bi se povećala za oko 5% i koja nije kritična što se tiče vremena pristupa, dodatni troškovi su opravdani.

Poglavlje 6

Simulaciona metodologija

Evaluacija predloženog rešenja je rađena pomoću simulatora. Simulacija je rađena pomoću simulatora Gem5 [133] (simulator je nastao spajanjem simulatora M5 i GEMS). Simulator podržava simulaciju procesora (asimetričnih i simetričnih višejezgarnih procesora), podsistema keš memorija, operativne memorije i komunikacione mreže između njih. Simulacija svih delova se vrši na nivou procesorskih ciklusa uz detaljno modelovanje komunikacija i kašnjenja usled ograničenih kapaciteta komunikacionih bafera. Ovaj simulator je odabran, jer omogućava evaluaciju performansi sistema sa dobrom preciznošću. Preciznost je razmatrana u [134]. Autori tvrde da je najgora moguća razlika između realnog sistema i simulatora Gem5 oko 17% i da toj grešci najviše doprinosi simulacija DRAM memorije. S obzirom na to da predložena rešenja imaju uticaja na događanja između podsistema keš memorija i procesora, simulacija DRAM memorije je ista i za predloženo rešenje i za postojeća rešenja, pa se međusobni odnos performansi koji se može izmeriti pomoću simulatora može smatrati kao dobra procena.

Drugi važan faktor pri odabiru simulatora je bio taj što je programski kod simulatora otvoren i javno dostupan. Treći važan faktor je bilo to što je razvoj simulatora podržan od strane Univerziteta u Mičigenu i Univerziteta u Viskonsinu i što se koristi u istraživačkim centrima kompanija ARM, AMD, Google, Micron, Metempsy, HP i Samsung [135]. Imajući u vidu brojne primene u industrijskom okruženju i istraživačkoj zajednici može se smatrati da je tačnost korišćenog simulatora zadovoljavajuća. Veća preciznost u simulaciji se može postići korišćenjem simulatora na RTL (eng. *register transfer level*) nivou, za model koji je napisan na HDL-u (eng. *hardver design language*). Problem sa tim pristupom bi bio što su razvoj modela i trajanje simulacije nekoliko redova veličine veći nego za simulator kao sto je Gem5.

Simulator je značajno nadograđen da bi se mogla simulirati predložena rešenja. Te nadogradnje obuhvataju podršku za transakcionu memoriju, hardversku migraciju niti, prilagođen podsistem keš memorija i sve ostale delove predloženih rešenja. Osnovni princip prilikom nadogradnje simulatora je bio da se simulirani deo može implementirati u hardver tako da vremenske karakteristike tog dela ne budu značajno promenjene, a ako se koristi neko uprošćenje da to uprošćenje bude konzervativno, drugim rečima da kašnjenja ne budu veća nego što bi bila u realnom sistemu, a mogu biti manja.

6.1 Gem5

Svi modelovani objekti u simulatoru Gem5 su napisani u jeziku C++ radi postizanja što boljih performansi simuliranja. Neki od objekata su ručno napisani, dok su neki objekti generisani automatski, prevođenjem specifikacije objekta napisane u jeziku posebne namene (eng. *domain specific language*).

Simulator podržava simulaciju različitih vrsta jezgara. Postoje dva tipa jezgra koji ne modeluju detaljno izvršavanje jezgra, a to su Atomic i Timing. Prvi modeluje samo funkcionalnost jezgra bez vođenja računa o protoku vremena, dok drugi modeluje isto funkcionalnost jezgra ali i vođenje računa

```
${SIM_DIR}/gem5/build/X86_MESI_Three_Level_TSX/gem5.opt --remote-gdb-port=0 --debug-
flags=Exec,ExecMacro --outdir=detailed_n2_STAMP_KmeansHigh_7_AMP_3 ${SIM_DIR}/gem5/configs/
example/he.py --num-cpu=2 --cpu-type=detailed --ruby --mem-size=4096MB '--hl1i_assoc=4;8' '--hnum-
cpus=1;1' --cluster-cpus=1:1 --hnum-l3caches=1 '--hl2_assoc=8;12' '--hl1d_latency=1;6' '--hcpu-
type=minor;detailed' --hetero --hl3_size=16MB '--hcpu-voltage=1.2V;1.2V' '--hl1d_size=64kB;64kB' '--
hl2_size=128kB;256kB' --hl3_assoc=32 '--hcpu-clock=2GHz;2GHz' '--hl1d_assoc=4;8' '--hl2_latency=2;10' '-
-hl1i_size=16kB;64kB' -c ${SIM_DIR}/stamp-mp/tsx/kmeans -o '-m15 -n15 -t0.05 -i ${SIM_DIR}/stamp-
mp/kmeans/inputs/random-n2048-d16-c16.txt -p2'
```

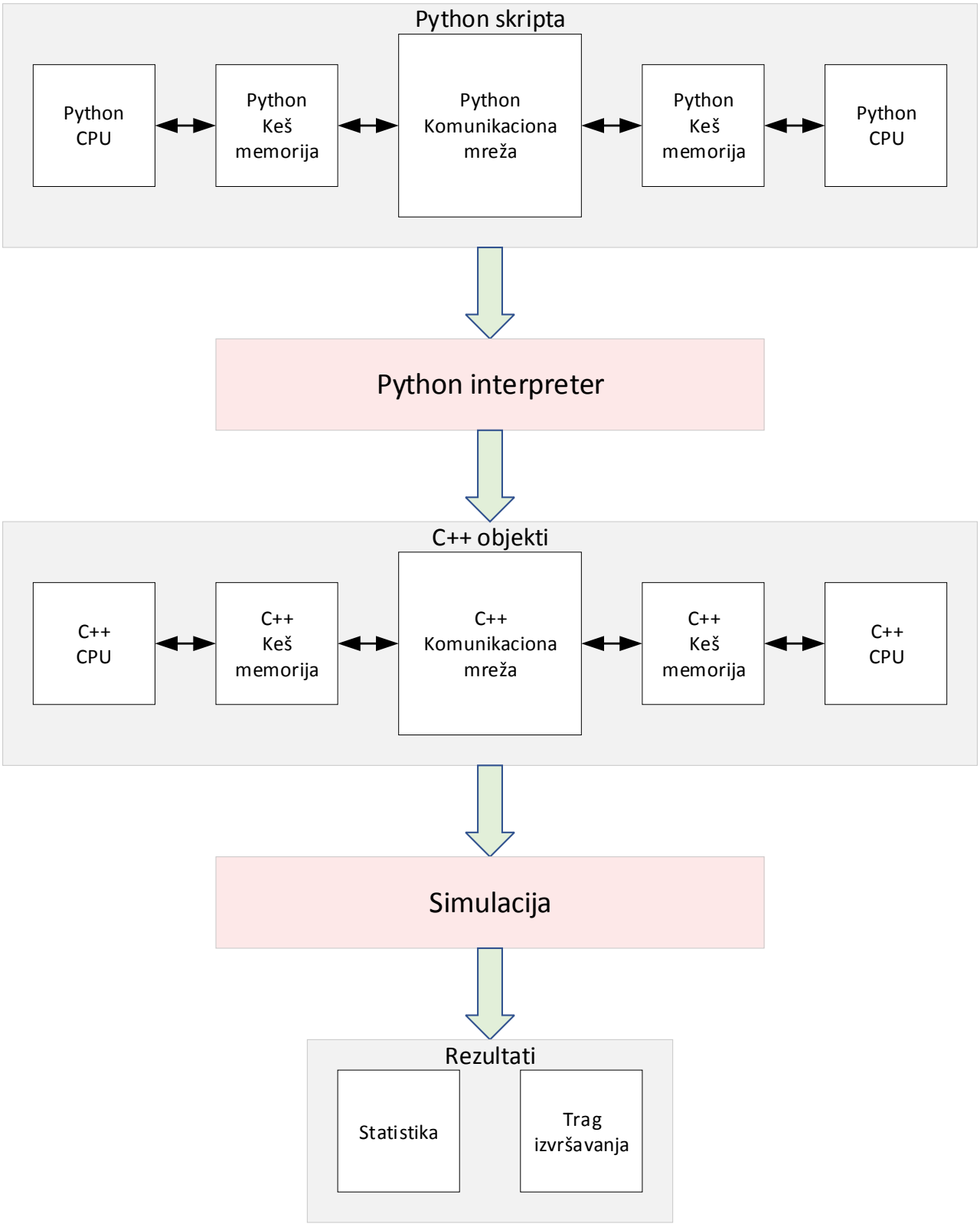
Slika 6.1: *Komanda za pokretanje jedne simulacije*

```
l2_cache = L2Cache(size = l2_size[j], assoc = l2_assoc[j],
                   start_index_bit = block_size_bits,
                   is_icache = False,
                   latency = l2_latency[j], is_dual = is_dual,
                   dualLatency=get_dual_latency(l2_size[j]),
                   dual_size_lines=l2_dual_size_lines[i],
                   dual_assoc=l2_dual_assoc[i],
                   dual_count_x = dual_count_x,
                   dual_count_y = dual_count_y)

class_l2controller = L2CacheDual_Controller if is_dual else
L2Cache_Controller

l2_cntrl = class_l1controller(version = cpu_id,
                              cache = l2_cache,
                              l2_select_num_bits = l2_bits,
                              cluster_id = i,
                              ruby_system = ruby_system,
                              l1_spatial_latency=l2_latency[j],
                              l2_temporal_latency=2)

exec("ruby_system.l2_cntrl%d = l0_cntrl" % (cpu_id))
exec("ruby_system.l2_cntrl%d = l1_cntrl" % (cpu_id))

#
# Add controllers and sequencers to the appropriate lists
#
cpu_sequencers.append(cpu_seq)
l1_cntrl_nodes.append(l1_cntrl)
l2_cntrl_nodes.append(l2_cntrl)

# Connect the L1 and L2 controllers
l1_cntrl.requestFromCache = l2_cntrl.bufferToL2FromL1
l1_cntrl.bufferToCache = l2_cntrl.bufferFromL2ToL1
```

Slika 6.2: *Konfiguracija simulacije u Python programskom jeziku*

o protoku vremena prilikom pristupa podsistemu keš memorija. Tip jezgara koji modeliraju rad jezgara na nivou procesorskih ciklusa sa svim značajnim detaljima protočne obrade su O3 (Out-of-order) i Minor (in order). O3 model simulira superskalarno jezgro, koje može da izvršava instrukcije van programskog poretka, dok Minor model izvršava instrukcije u poretku. Oba tipa jezgra dozvoljavaju izvršavanje više instrukcija paralelno. Jezgra su parametrizovana i mogu se menjati broj instrukcija koji mogu da se istovremeno učitaju, izvrše i završe, može se menjati instrukcijski set koji procesor podržava, broj kanala i širina za komunikaciju sa podsistemom keš memorija, frekvencija takta, itd. Svi parametri jezgra se mogu podesiti u toku startovanja simulacije osim instrukcijskog seta. Instrukcijski set se bira prilikom prevođenja koda simulatora.

Instrukcije se definišu pomoću tri različita jezika posebne namene. Prvi jezik služi za definisanje instrukcijskog seta. Za svaku instrukciju se definiše koji ima kod, koje i koliko operanada ima. Postoje definicije za nekoliko različitih instrukcijskih setova (x86, ARM, MIPS, ...). Drugi jezik služi za definisanje šta instrukcije rade. Pošto neki instrukcijski setovi sadrže izrazito složene instrukcije, one



Slika 6.3: *Proces simulacije u Gem5 simulatoru*

se izvršavaju kao niz jednostavnih RISC (eng. *reduced instruction set computer*) mikro instrukcija koje jezgra zapravo izvršavaju. Prevođenje instrukcije u niz mikro instrukcija se vrši u toku rada posle dohvatanja instrukcije. Skup mikro instrukcija je isti za sve instrukcijske setove. Ako neki tip instrukcijskog seta već sadrži RISC instrukcije, instrukcija se sastoji od jedne mikro instrukcije. Šta radi pojedinačna mikro instrukcija, tj. šta čita i upisuje u memoriju i registre i koju obradu podataka

```
system.ruby.l1_cntrl1.Dcache.demand_hits      5590021          # Number of cache demand hits
system.ruby.l1_cntrl1.Dcache.demand_misses    16646            # Number of cache demand misses
system.ruby.l1_cntrl1.Dcache.demand_accesses  5606667          # Number of cache demand accesses
system.ruby.l1_cntrl1.Icache.demand_hits      1822782          # Number of cache demand hits
system.ruby.l1_cntrl1.Icache.demand_misses    396              # Number of cache demand misses
system.ruby.l1_cntrl1.Icache.demand_accesses  1823178          # Number of cache demand accesses
```

Slika 6.4: *Deo statističkih informacija za jednu simulaciju*

vrši, definiše se pomoću trećeg jezika posebne namene. Postoje nekoliko varijanti mikro instrukcija, koje se koriste u različitim modelima jezgara. Različite varijante postoje da bi se protok vremena simulirao na različite načine (npr. nema protoka vremena kod mikro instrukcija za Atomic jezgro).

Simulator omogućava korišćenje konvencionalnog podsistema keš memorija, gde se broj nivoa keš memorije može podešavati. Za svaku keš memoriju se može podešavati veličina, veličina skupa za set-asocijativno mapiranje, vreme pristupa, algoritam zamene, itd. Moguće je menjati na koji način su povezane keš memorije međusobno, a i sa procesorom, koja je konfiguracija komunikacione mreže između njih, koje keš memorije su deljene, a koje su privatne. Sve navedene parametre je moguće podešavati prilikom startovanja simulacije. Primer komande za startovanje jedne aplikacije dat je na slici 6.1.

Implementiran je veći broj protokola keš koherencije koji se može izabrati prilikom startovanja simulacije. Protokol keš koherencije se definiše pomoću jezika posebne namene. Za svaki nivo keš memorije se definiše u kojim stanjima može da se nađe ulaz i koja tranzicija može da se izvrši iz jednog stanja u drugo kada pristigne neka poruka od druge keš memorije, procesora ili operativne memorije. Prevodilac, na osnovu specifikacije, generiše kontroler keš memorije (programski kod koji se dobije je C++ kod) koji vrši tranzicije iz stanja u stanje i komunicira sa ostalim kontrolerima. Komunikacija između kontrolera je bazirana na imenovanim porukama i nije moguće osluškivanje (eng. *snooping*) poruka. Preko poruka keš koherencije se šalju odgovarajući zahtevi, ali i sami podaci. Dostupni protokoli keš koherencije su: jednostavni protokol sa dva stanja (modifikovan i nevalidan) i složeniji protokoli MESI [129] i MOESI [130].

Konfiguracija sistema se vrši u programskom jeziku Python. Svaki objekat koji se simulira ima klasu napisanu i u jeziku Python. Atributi klase služe kao parametri modelovanog objekta ili služe kao portovi za povezivanje sa drugim objektima. Pre startovanja simulacije prave se objekti u programskom jeziku Python, podešavaju odgovarajućim upisom u atribute i povezuju se. Primer kreiranja keš memorija drugog nivoa i povezivanje sa keš memorijama prvog nivoa dat je na slici 6.2. Nakon toga se izvršavanjem Python koda pokreće simulator kome se nalaže da napravi iste objekte, čije su klase napisane u jeziku C++, i da ih podesi i poveže kako su podešeni i povezani objekti u jeziku Python. Simulacija se nakon toga izvršava pomoću koda napisanog na jeziku C++. Prikaz procesa simulacije dat je na slici 6.3.

Simulator podržava dva režima izvršavanja aplikacija. Prvi je izvršavanje samo jedne aplikacije, bez izvršavanja bilo čega drugog. Drugi je izvršavanje aplikacije u okviru operativnog sistema zajedno sa drugim aplikacijama. Sve aplikacije za simulaciju se mogu prevesti pomoću komercijalno dostupnih prevodilaca iz viših programskih jezika. Prilikom završetka simulacije, simulator u tekstualnom obliku generiše rezultate simulacije. Rezultati simulacije su sve bitne statističke informacije koje opisuju izvršavanje simuliranog programa. Na primer to je simulirano vreme, broj pogodaka u keš memoriji, broj mikro instrukcija koje su izvršene, itd. Primer dela rezultata za jednu simulaciju dat je na slici 6.4.

Za potrebe testiranja simulatora omogućeno je izvršavanja instrukcija korak po korak (eng. *single-stepping*), ali i testiranje pomoću logovanja. Testiranje pomoću logovanja uključuje ispisivanje proizvoljnih podataka u nekom trenutku simulacije. Moguće je uključiti ispisivanje poruke samo za neke delove sistema, pošto broj ispisanih linija može biti prevelik (ispis može zauzeti prostor od nekoliko

GB pa naviše) i za vrlo kratke simulirane intervale. Prilikom ispisivanja poruke simulator ispisuje koji objekat je generisao poruku i u kom trenutku simulacije. Ispisivanje poruka može početi u naznačenom trenutku u simulaciji. Izvršavanje u ovim režimima je znatno sporije nego izvršavanje u normalnom režimu, zbog ispisa poruka.

6.2 Simulacija transakcione memorije

Simulacija transakcione memorije implementirana je nadogradnjom dva dela simulatora. Prvi deo je procesor, zajedno sa dodavanjem novih instrukcija u instrukcijski skup. Drugi deo je promena protokola keš koherencije i samim tim simulacije keš kontrolera.

Transakciona memorija je implementirana tako da se evidencija o podacima kojima se pristupa u transakciji vodi u keš memoriji prvog nivoa. Detekcija konflikata između transakcija vrši se pomoću modifikovanog MESI protokola keš koherencije [129]. Modifikacija uključuje nove poruke koje sadrže informacije o spekulativnim čitanjima i upisima za vreme transakcije. Da li se konflikt desio utvrđuje se poređenjem adrese podatka kome je spekulativno pristupila druga transakcija sa skupom podataka kojima je trenutna transakcija spekulativno pristupila. Skup pročitanih podataka se pravi pomoću Blumovog filtra [69]. On se koristi da bi se smanjila količina hardvera za njegovu implementaciju i da bi operacije nad skupovima bile efikasne. Negativan efekat je pojava lažnih konflikata koji će prouzrokovati bespotrebno poništavanje transakcije. Čim se detektuje konflikt jedna od transakcija se poništava. Poništava se transakcija na onom jezgru koje primi poruku.

Za izvršene eksperimente se koristio instrukcijski skup x86 (64-bitna varijanta). Taj instrukcijski skup sadrži instrukcije za transakcionu memoriju i one su date u [85]. Ovaj instrukcijski skup je izabran zato što postojeći prevodioci podržavaju sve te instrukcije. Samim tim, prevođenje programa za simulaciju može da se uradi na jednostavan način.

Instrukcije, koje su dodate, mogu se klasifikovati kao CISC (eng. *complex instruction set computer*). Dodate su instrukcije za startovanje transakcije, za potvrđivanje transakcije, za proveru da li se trenutno izvršava transakcija i za poništavanje transakcije. Karakteristike tih instrukcija su iste kao i instrukcije opisane u [85]. Sve instrukcije mogu biti programski pozvane, a instrukcija za poništavanje može da se pozove i prilikom detekcije situacije koja zahteva prekid transakcije (npr. konflikt sa drugim jezgrom). Te instrukcije su razložene na niz jednostavnih RISC instrukcija. Za to su iskorišćene postojeće RISC instrukcije, ali su dodate i nove. Te nove instrukcije služe za prosleđivanje informacije podsistemu keš memorija. Prosleđuju se informacije o početku, potvrđivanju ili poništavanju transakcije. Podsistem keš memorija može da odgovori na svaku od tih instrukcija sa informacijom da li je došlo do neregularne situacije (npr. konflikt sa drugim jezgrom). Tom prilikom šalje i informaciju šta je uzrok, tj. kod uzroka. Isti odgovor može da pošalje i u slučaju bilo koje druge instrukcije koja pristupa memoriji i na taj način, najranije moguće, obaveštava jezgro o problemu.

Instrukcija za početak transakcije čuva kontekst registarskog fajla i javlja podsistemu keš memorije da je transakcija aktivna. Instrukcija prihvata jedan operand koji predstavlja relativni pomeraj do instrukcije na koju treba skočiti posle poništavanja transakcije. Prilikom izvršenja instrukcije u registar EAX (najnižih 32-bita registra RAX) se upisuje status da je počelo izvršavanje transakcije. Instrukcija za potvrđivanje transakcije javlja podsistemu keš memorija da se transakcija uspešno završila i da može spekulativno promenjene podatke u keš memoriji proglasiti nespekulativnim. U registar EAX upisuje status da se transakcija uspešno izvršila. Instrukcija za poništavanje transakcije restaurira kontekst koji je sačuvala instrukcija za početak transakcije, javlja podsistemu keš memorija da je transakcija poništena i da može da poništi sve podatke koji su spekulativno promenjeni, prelazi na instrukciju čiju je adresu zapamtila instrukcija za početak transakcije i u registar EAX upisuje status da je transakcija poništena i razlog poništavanja.

Instrukcije, koje su dodate, treba da sačuvaju ili restauriraju kontekst registarskog fajla jezgra. To se radi pomoću RISC instrukcija koje vrše transfer iz jednog registra u drugi. Mikro instrukcija se

izvršava za svaki registar koji je potrebno sačuvati ili restaurirati. Broj tih transfera u jednom ciklusu je ograničen brojem instrukcija koje jezgro može da izvrši u jednom ciklusu. Time je taj pristup konzervativan, jer svi registri mogu i paralelno da se prekopiraju u jednom ciklusu. Za potrebe čuvanja konteksta uveden je duplikat dela registarskog fajla. Deo registarskog fajla koji se čuva je registar naredne instrukcije (PC), registarske statusne reči i šesnaest programski dostupnih registra za x86 instrukcijski set (RAX, ..., RSI, R9, ..., R16). Ako transakcija promeni neki drugi registar, dolazi do poništavanja transakcije i taj deo koda se ne može izvršiti u transakciji. Pored duplikata registara, dodat je i statusni registar transakcije. Taj registar sadrži jedan bit koji govori da li se transakcija trenutno izvršava, brojač koji govori do kog nivoa je transakcija ugnežđena i razlog završetka transakcije. Ugnežđavanje transakcija se radi odmotavanjem, tj. transakcijom se smatra sve od početka transakcije prvog nivoa do njenog završetka. Ako u bilo kom trenutku nastane problem poništavaju se sve unutrašnje transakcije. Na taj način, unutrašnje transakcije ne čuvaju i ne restauriraju posebno svoj kontekst prilikom započinjanja i završetka.

Keš memorija prvog nivoa sadrži podršku za transakciono izvršavanje. Dodata su dva bita za svaki blok u keš memoriji koji označavaju da li se tom bloku pristupilo za čitanje i/ili upis u toku transakcije. Instrukcije koje započinju ili završavaju transakciju sadrže memorijsku barijeru da bi se odgovarajući pristupi memoriji završili pre početka ili pre završetka transakcije. Na taj način se izbegava pristup memoriji u toku transakcije od strane one instrukcije koja se nalazi van transakcije zbog instrukcijskog paralelizma koji podržava jezgro. Prilikom poništavanja transakcije svi blokovi, koji su promenjeni tokom transakcije, se poništavaju. Poništavanje se radi u jednom ciklusu. Prilikom potvrđivanja transakcije poništavaju se svi bitovi koji označavaju da se podatku pristupilo u transakciji. Poništavanje se radi u jednom ciklusu.

Kontroler keš memorije prvog nivoa prima poruke od jezgra i od ostalih kontrolera prvog nivoa koje su deo standardnog protokola keš koherencije. Pored tih poruka od jezgra prima i informaciju o početku ili završetku transakcije. Na osnovu poruka od strane jezgra prilikom čitanja i upisa, kontroler ažurira bitove za blok u keš memoriji kojem se pristupilo. Ukoliko se ukaže potreba za zamenu bloka koji se pročitao u transakciji, kontroler tu zamenu vrši, ali beleži adresu bloka u Blumov filter [69]. Na taj način Blumov filter sadrži adrese onih blokova koji su pročitani u toku izvršavanja transakcije. Na početku svake transakcije Blumov filter se isprazni. Ukoliko se pojavi potreba da se zameni blok u koji se upisalo tokom transakcije, kontroler odgovara procesoru da je došlo do problema u toku izvršavanja transakcije zbog prekoračenja. Da bi validna verzija podatka bila očuvana, kontroler prilikom upisa u blok, tokom izvršavanja transakcije, taj blok prosledi višem nivou podsistema keš memorije i tek tada izvrši upis u blok. Za to je potrebno dodati novu poruku koja se šalje u okviru održavanja keš koherencije.

Poruke koje prima od drugih jezgara, kontroler koristi za proveru da li je došlo do konflikta. Ta provera se radi samo kada se izvršava transakcija. Ukoliko je primljena poruka, koja opisuje čitanje, kontroler provera da li je u taj blok bilo upisa tokom izvršavanja transakcije i ako jeste, transakcija se poništava. Ukoliko se radi o upisu, proverava se da li se tom bloku pristupilo zbog čitanja ili zbog upisa, pri čemu se proverava i sadržaj Blumovog filtra, jer se tu nalaze adrese sa kojih je vršeno čitanje, a koje su zamenjene u keš memoriji. Kontroler od tog trenutka na dalje ne šalje poruke ostalim jezgrima i čeka da odgovori na neki memorijski zahtev svog jezgra. U odgovoru mu javlja da transakcija mora da se prekine. To dovodi do ispravnog izvršavanja programa, jer se podaci koji su modifikovani nalaze u višem nivou podsistema keš memorije koji će odgovarati na zahteve ostalih jezgara.

U slučaju da keš memorija višeg nivoa vrši zamenu bloka, kontroler takođe poništava transakciju ako je tom bloku pristupljeno u toku transakcije. To se mora uraditi, jer poruke keš koherencije više neće stizati za taj blok. Protokol keš koherencije je takav da se poruke šalju samo onim keš memorijama za koje se zna da imaju taj blok. Kontroler sve poruke, koje mogu da prekinu transakciju, šalje i svom jezgru, da bi jezgro poništilo transakciju u slučaju da ima neku instrukciju koja pristupa tom podatku, ali ta instrukcija još nije stigla na red da uputi poruku keš kontroleru. Sve poruke koje se šalju za ove potrebe već postoje za održavanje keš koherencije sistema bez transakcione memorije,

tako da sam protokol keš koherencije nije potrebno menjati.

6.3 Simulacija migracije

Simulacija migracije je implementiratirana što je moguće sličnije onome kako bi bila implementirana fizički na čipu. Poruke koje razmenjuju jezgra se prenose preko linija koje služe za prenos poruka protokola keš koherencije. Transfer poruka se obavlja preko bafera poruka koji mogu da propuste konačan broj poruka u jednom ciklusu simulacije. Baferi su ograničenog kapaciteta. Time se dobija blokiranje pošiljalaca, a to je jezgro procesora za poruke migracije. Poruka može biti proizvoljne veličine, ali se transfer vrši u delovima koji su jednaki veličini podatka koji može da se pošalje prvom nivou keš podsistema. Simulacija transfera na ovakav način omogućava da poruke mogu da budu odložene i da svaka poruka ima različito vreme prenosa u zavisnosti od količine saobraćaja na linijama. Na taj način rezultati simulacije bi trebalo da uključe i režijsko vreme slanja poruke.

Kontekst koji se prenosi sa jednog jezgra na drugo je minimalan koji je potreban za izvršavanje transakcije. Ograničenje x86 instrukcijskog skupa ne dozvoljava da se u transakcijama koriste instrukcije koje pristupaju kontrolnim, segmentnim, X87 i MMX registrima [136]. Iz tog razloga, prilikom migracije prenose se samo osnovni arhitekturalni registri koji su dostupni korisničkoj aplikaciji. Prilikom migracije jezgro se u potpunosti zaustavlja, protočna obrada se u potpunosti isprazni pa se onda kontekst pamti i šalje drugom jezgru. Migracija je zbog toga spora operacija koja u slučaju čestog korišćenja može značajno da ugrozi performanse sistema. Sve poruke koje se šalju za potrebe predloženog rešenja implementirane su isto pomoću protokola za keš koherenciju. Centralni registar i keš registri za čuvanje podataka o potrebama za migraciju transakcija implementirani su sa vremenom pristupa od jedne periode takta, jer su to asocijativne strukture male veličine (maksimalno 32 ulaza).

Algoritam za odlučivanje, da li transakciju treba migrirati, napravljen je tako da se aktivira svaki put kada se neka transakcija završi i kada u direktorijum stigne informacija o njenom uspehu zajedno sa ostalim informacijama, koje su potrebne za rad sistema M-HTM. Sve te informacije prenose se putem poruka za keš koherenciju u onoliko paketa koliko je potrebno da stanu sve informacije. Pre početka izvršavanja algoritma jedan ciklus takta se koristi za čitanje prethodnih informacija o migraciji. Pošto je algoritam jednostavan, implementiran je tako da se njegovo izvršavanje odradi u jednom taktu, dok se upis rezultata radi u sledećem taktu. Ukoliko se završetak dva izvršavanja iste transakcije poklopi da bude u susednim ciklusima takta, rezultat od prvog izvršavanja može da se koristi u istom taktu kad traje i upis rezultata, što znači da algoritam za drugo izvršavanje mora da se odloži za jedan takt. Vreme izvršavanja algoritma je simulirano tako da traje od tri do četiri periode takta. Izvršavanje algoritma se radi paralelno sa ostalim aktivnostima velikog jezgra.

6.4 Simulacija prilagođenog podsistema keš memorija

Simulacija prilagođenog podsistema keš memorija zahteva izmenu tri stvari. Prva je izmena koda za samu simulaciju memorije koja se koristi. Druga je izmena keš kontrolera koji pristupa memoriji. Treća je izmena protokola keš koherencije.

Memorija se simulira pomoću koda koji je napisan na jeziku C++. On podržava čuvanje podataka na nivou jednog bloka. Sve funkcionalnosti zavise od veličine bloka. Pod funkcionalnostima se podrazumevaju tip mapiranja, algoritam zamene, implementacija upisa, čitanja i pretrage. Kod je modifikovan, da se omogući promenljiva veličina bloka i da bi se simulirao i prostorni i vremenski deo. Dodat je kod za pretragu vremenskog dela keš memorije, koji omogućava proveru da li se blok nalazi u keš memoriji. Provera se vrši na osnovu adrese jedne reči tog bloka. Ta reč ne mora biti prisutna u keš memoriji, a pretraga može da vrati da se blok nalazi u keš memoriji, jer se neka druga reč tog bloka nalazi u keš memoriji. Pretraga vremenskog dela je iste dužine trajanja kao i običan pristup

vremenskom delu (za upis ili čitanje). Pomoću te provere detektuje se u kom delu podatak treba da se nađe u keš memoriji ili se detektuje da treba izbaciti blok iz keš memorije prilikom premeštanja tog bloka u prostorni deo keš memorije. Memoriji je dodat podatak o tipu dela koji predstavlja (prostorni ili vremenski), da bi mogla da se pretraga vrši na odgovarajući način.

Keš kontroler je napisan u jeziku posebne namene. Taj jezik omogućuje pisanje funkcija koje treba da se izvrše kada se primi zahtev preko nekog porta, omogućava definisanje tih portova, definisanje memorija kojima upravlja keš kontroler, načina vođenja evidencije za svaki blok (bit validnosti, stanje, ...), pomoćne bafere za smeštanje podataka koji su u tranziciji iz jednog stanja u drugo, kao i druge pomoćne stvari (npr. Blumov filter [69] za transakcionu memoriju).

U keš kontroler privatnih keš memorija je dodat vremenski deo keš memorije. Svaka funkcija koja izvršava neku operaciju prilikom prijema zahteva preko nekog porta je promenjena da uključi pristup i vremenskom delu. Recimo, ako stigne zahtev od jezgra za čitanje podatka, funkcija za taj port proverava da li se taj podatak nalazi u bilo kom delu keš memorije. Keš memorije prvog nivoa imaju i deo keš memorije za instrukcije. Taj deo je implementiran sa blokom iste veličine kao za prostorni deo. Moguće je da se i neki blok nađe u instrukcijskom delu, ako sadrži i instrukciju i podatke. U tom slučaju keš kontroler prvo izbaci taj deo iz keš memorije, pa onda od keš memorije drugog nivoa traži taj podatak za deo za podatke. Dalje dohvatanje podatka se nastavlja kao da blok nikad nije bio u instrukcijskom delu (može da se dohvati ili u prostorni ili u vremenski deo u zavisnosti od bita procene lokalnosti). Postoji mogućnost i za obrnutu situaciju, u tom slučaju se blok izbaci iz prostornog dela pa se onda pošalje zahtev za dohvatanje u instrukcijski deo. Dok je blok u instrukcijskom delu ne vrši se procena tipa lokalnosti za taj blok.

Keš kontroler prvog nivoa sadrži i kod za simulaciju transakcione memorije. Da bi keš kontroler podržavao podeljenu keš memoriji i transakcije mora se obezbediti detekcija konflikata i za podatke koji se nalaze u vremenskom delu. To je urađeno tako što je dodat Blumov filter i za vremenski deo. Taj Blumov filter sadrži sve adrese reči kojima je pristupano u toku transakcije. Blumov filter za prostorni deo sadrži adrese blokova. Radi ispravnog simuliranja transakcione memorije kada pristigne neka informacija o čitanju neke adrese, provera te adrese se vrši u oba Blumova filtra, jer je moguće da se promena lokalnosti dogodi u toku trajanja transakcije. Ako pristigne zahtev sa oznakom prostorne lokalnosti, Blumov filter za vremenski deo se proverava za svaku adresu iz bloka. Ta provera traje onoliko ciklusa takta koliko ima reči u bloku (osam u sprovedenim eksperimentima). Ako pristigne zahtev sa oznakom vremenske lokalnosti, Blumov filter za prostorni deo se proverava sa adresom bloka, koja se dobija od adrese reči poništavanjem najnižih bitova koji označavaju broj reči u bloku. Ta operacija traje jedan ciklus takta.

Keš kontroler trećeg nivoa je izmenjen da sadrži bite validnosti reči za svaki blok. Oni se modifikuju u funkciji koja obrađuje zahteve od nižih nivoa keš memorije. Njihova promena se obavlja onoliko koliko traje operacija upisa u memoriju, čak i u slučaju kada se menjaju svi bitovi, jer se pristupa celom bloku. Omogućena je promena samo jedne reči iz bloka kada se vrši upis i ta operacija traje koliko dva pristupa memoriji plus jedan ciklus takta (čita se blok, menja se, pa se promenjen upisuje). Dodat je jedan bit za svaki blok koji čuva informaciju o procenjenom tipu lokalnosti. Keš memorija prvog nivoa kada napravi novu procenu lokalnosti u za to namenjenoj poruci šalje zahtev za izmenu tipa lokalnosti (poruka ide preko keš memorije drugog nivoa). Dodat je jedan bit u katalog koji označava lokalnost koju trenutno ima blok u podsistemu. Svaki put kada kontroler detektuje promenu lokalnosti taj bit se menja. Promena lokalnosti se detektuje tako što je procena lokalnosti različita od trenutne lokalnosti, a pristigla je poruka od nižeg nivoa keš memorije koja nema informaciju o lokalnosti. Ta operacija traje onoliko koliko traju dva pristupa memoriji.

Protokol keš koherencije je implementiran u istom jeziku kao i keš kontroler. Za svaki nivo keš memorije definišu se stanja u kojima mogu da se nađu podaci. Koristi se MESI [129] protokol. Pošto ima više nivoa keš memorije podaci u keš memoriji mogu da budu u među stanjima. Ta među stanja se isto deklarišu kao obična stanja. Jezik dozvoljava da se pišu funkcije koje reaguju na neki zahtev kad je podatak u odgovarajućem stanju. Sve te funkcije za keš memorije privatnih nivoa su promenjene,

tako da mogu da rade različitu obradu za podatke koji se smeštaju u različite delove keš memorije. Recimo ako stigne zahtev za čitanje podatka kojeg nema u keš memoriji, a taj podatak treba da se smesti u vremenski deo keš memorije, pre dohvatanja iz keš memorije višeg nivoa, mora da se u vremenskom delu obezbedi ulaz (ako nema slobodnog, mora da se izbaci jedan podatak). Takođe, te funkcije su izmenjene da rade sa adresama reči, a pre toga su radile samo sa adresama blokova.

Dodato je među stanje za čekanje na obradu celog bloka u vremenskom delu. Recimo ako se izbacuju svi delovi bloka iz vremenskog dela (na primer prilikom promene lokalnosti), ulazi se u stanje čekanja sve dok se ne izbace sve reči tog bloka iz keš memorije. Pošto ta operacija traje osam ciklusa takta, prilikom ulaska u to stanje blokira se pristup tom bloku i svi zahtevi koji stignu za taj blok se odlažu sve dok sve reči ne budu izbačene. Na taj način se sprečava problem koji može da nastane kad se izbace neke reči bloka, a za jednu od njih stigne drugi zahtev za pristup. U tom slučaju moglo bi da se krene sa dovlačenjem reči koja je izbačena i ako bi se vratila, ne bi bio ceo blok izbačen, što bi uvelo sistem u neregularno stanje. Ovakva simulacija tog slučaja je konzervativna, jer bi moglo da se pristupa rečima koje još nisu izbačene. U keš memoriji drugog nivoa uvedeno je među stanje za čekanje da keš memorije prvog nivoa izbaci sve reči jednog bloka iz vremenskog dela. U svim drugim slučajevima, za jedan zahtev treba da stigne jedan odgovor. Međutim, kod ovog zahteva stiže osam odgovora (sve reči jednog bloka), pa se u tom stanju čekaju svi ti odgovori.

U istom jeziku definišu se i poruke koje se koriste za komunikaciju između keš memorija. Poruke su promenjene tako da omoguće rad predloženog rešenja. Imaju dodate potrebne informacije o lokalnosti i adresa je proširena da mogu da stanu svi biti adrese jedne reči. Postoje dve vrste poruke jedna za slanje zahteva druga za slanje odgovora.

6.5 Pronalaženje grešaka u simulaciji

Izmene koje su urađene u simulatoru su značajne. Prilikom dodavanja tolikog broja naredbi, velika je verovatnoća da se pojave greške u simulaciji. Većinu grešaka je lako otkriti i ispraviti, dok se manji broj grešaka vrlo teško ispravlja, jer ne može lako da se otkrije uzrok greške. Na primer pošto simulacije traju dugo (nekoliko dana, a za jednu aplikaciju nekoliko nedelja), dešavalo se da se greška pojavi posle nekoliko dana izvršavanja simulacije. U nastavku će biti ukratko prikazana metodologija korišćena za pronalazak i otkrivanje uzroka greške. Pod pojmom uspešne simulacije izvršavanja aplikacije uzima se ono izvršavanje koje generiše logički ispravan rezultat. Za logički ispravan rezultat uzima se rezultat koji generiše izvršavanje aplikacije na postojećem računaru. Za prikazane rezultate svako simulirano izvršavanje aplikacije je generisalo logički ispravan rezultat.

Tokom razvoja simulatora uočene su dve vrste grešaka. Prvoj vrsti pripadaju one greške koje dovedu do prekida simulacije. Prekid može da se dogodi zbog greške koja prekida izvršavanja procesa simulatora (npr. kod simulatora dereferencira pokazivač koji ima vrednost nula). Takve greške se obično javljaju rano u simulaciji i najlakše ih je pronaći izvršavanjem simulatora u test režimu (eng. *debug run*). Kada se otkrije mesto u kodu gde se ta greška javlja i kada se pogledaju vrednosti okolnih promenljivih obično se lako zaključi šta je uzrok greške. Prekid simulacije može da se dogodi i kad je izvršavanje simuliranih instrukcija generisalo izuzetak u simulaciji. Tada simulator ispiše simulirano vreme u kome se dogodila greška. Najbolje je pokrenuti simulaciju sa uključenim ispisom svih instrukcija koje se izvršavaju. Primer dela jednog ispisa dat je na slici 6.5. Moguće je podesiti trenutak početka ispisa informacija o radu. To vreme treba podesiti malo pre nego što se simulacija prekinula. Na taj način se zabeleže samo bitne informacije za prekid rada, pa rezultujući ispis sadrži samo potrebne informacije, te brzina izvršavanja simulacije nije značajno ugrožena. Za svaku izvršenu instrukciju ispisuje se koja je to instrukcija sa kojim podacima je radila, na kojoj adresi su ti podaci, koje jezgro ju je izvršilo, u kom trenutku simulacije, itd. Treba ispratiti koja instrukcija je doprinela da se dobijaju pogrešni rezultati. Kada se utvrdi koja je to instrukcija, malo pre njenog izvršavanja treba uključiti sve značajne ispise za deo koda koji je modifikovan (šta radi svaka faza protočne obrade, šta

```
25230500: system.cpu0 T0 : @spin_lock+31.1  :  0x40480a   POP_R : addi   rsp, rsp, 0x8 : IntAlu :  D=0x00007fffffffec50
25231000: system.cpu0 T0 : @spin_lock+31.2  :  0x40480a   POP_R : mov   rbp, rbp, t1 : IntAlu :  D=0x00007fffffffec60
25232000: system.cpu0 T0 : @spin_lock+32    :  0x40480b ret
25232000: system.cpu0 T0 : @spin_lock+32.0  :  0x40480b   RET_NEAR : ld   t1, SS:[rsp] : MemRead :  D=0x0000000000404db4 A=0x7fffffffec50
25232000: system.cpu0 T0 : @spin_lock+32.1  :  0x40480b   RET_NEAR : addi   rsp, rsp, 0x8 : IntAlu :  D=0x00007fffffffec58
25232500: system.cpu0 T0 : @spin_lock+32.2  :  0x40480b   RET_NEAR : wripi   , t1, 0 : IntAlu :
25237500: system.cpu0 T0 : @pthread_mutex_lock+28    :  0x404db4 mov           eax, 0
25237500: system.cpu0 T0 : @pthread_mutex_lock+28.0  :  0x404db4   MOV_R_I : limm   eax, 0 : IntAlu :  D=0x0000000000000000
25238000: system.cpu0 T0 : @pthread_mutex_lock+33    :  0x404db9 leave
25238000: system.cpu0 T0 : @pthread_mutex_lock+33.0  :  0x404db9   LEAVE : mov   t1, t1, rbp : IntAlu :  D=0x00007fffffffec60
```

Slika 6.5: *Trag jednog izvršavanja sa ispisom mikro instrukcija*

radi keš memorija sa promenama stanja podataka, poruke koje se šalju između keš memorija, itd.). Ispis koji se dobije na taj način je izuzetno velik, zato se i samo lokalizovano uključuje. Dobijene podatke treba prekontrolisati i kada se uoči neka greška treba otkriti zašto se dešava.

Drugoj vrsti pripadaju greške koje ne dovode do prekida simulacije, već simulacija nastavlja da se izvršava beskonačno. Takve greške obično prouzrokuju da neka simulirana nit uđe u beskonačnu petlju. Kada se posumnja na taj slučaj, potrebno je prekinuti simulaciju nakon nekog vremena. To se može uraditi slanjem signala za regularno gašenje programa od strane operativnog sistema. Tada će simulator prekinuti izvršavanje i ispisati simulirano vreme. Nakon toga pokrenuti simulaciju od početka i treba uključiti ispisivanje instrukcija značajno pre tog trenutka. Ukoliko se primeti da se instrukcije stalno ponavljaju, treba ponoviti simulaciju sa ranijim vremenom početka ispisa instrukcija. To treba ponavljati sve dok se ne pronađe prvo izvršavanje te beskonačne petlje. Tada treba ispratiti koje instrukcije doprinose da se uslov za izvršavanje petlje ne menja. Potom treba uraditi isti postupak kao i za prvu vrstu grešaka.

Nekada se jave greške koje imaju izuzetno veliku razliku između vremena kada greška počne da prouzrokuje probleme koji se mogu uočiti i trenutka kada je ona nastala. U tom slučaju nije lako ispratiti trag instrukcija da bi se pronašla ona koja prouzrokuju grešku. Metoda koja je primenjena u tim slučajevima je pronalaženje instrukcija koje prouzrokuju probleme u kodu višeg programskog jezika. U višem programskom jeziku je lakše ispratiti koje prethodne instrukcije su mogle da utiču na podatke koji stvaraju probleme u izvršavanju i u kojim funkcijama se te instrukcije izvršavaju. U tom slučaju ispis simulatora je moguće filtrirati da ispisuje samo one instrukcije koje izvršavaju u okviru funkcija za koje se pretpostavlja da prouzrokuju grešku.

Poglavlje 7

Evaluacija predloženog rešenja

Evaluacija predloženog rešenja vršena je pomoću simulacije izvršavanja postojećih aplikacija koje služe za testiranje transakcione memorije. U ovom poglavlju su opisana podešavanja simulatora, test aplikacije koje su korišćene, prezentovani su dobijeni rezultati i data je diskusija o dobijenim rezultatima. Pojedinačno su razmatrani eksperimenti sa migracijom transakcija i sa prilagođenim podsistemom keš memorija.

Simulacija je vršena pomoću značajno nadograđenog simulatora Gem5[133]. Detalji nadogradnje su opisani u prethodnom poglavlju. Simulacija jedne test aplikacije vrši se u izolaciji, tj. samo se ta aplikacija izvršava u sistemu, od početka do kraja. Sve niti test aplikacije imaju svoje jezgro za izvršavanje, ne prekidaju se i ne vrši se raspoređivanje.

Oba jezgra rade na istoj frekvenciji, iako to u realnim implementacijama nije slučaj. Obično veliko jezgro radi na većoj frekvenciji pa i na taj način ubrzava izvršavanje niti. Međutim, u [137] utvrđeno je da simulacije sa jezgrima koji rade na različitim frekvencijama daju nerealne rezultate. Rezultati pokazuju da veliko jezgro radi mnogo brže od malog jezgra nego što je to u realnim implementacijama. Ovde je uzet restriktivniji pristup za predloženo rešenje i simulirano je da obe vrste jezgra rade na istoj frekvenciji. Keš memorije prvog i drugog nivoa su privatne za svako jezgro, dok je keš memorija trećeg nivoa zajednička za sva jezgra.

Malo jezgro je modelovano po Intel Pentium procesoru [138] koje zauzima oko 3.3 miliona tranzistora. Veliko jezgro je modelovano po Intel Pentium-M jezgru koje zauzima oko 14 miliona tranzistora [139]. U ovoj analizi veliko jezgro ne podržava višenitno izvršavanje, tako da bi njegova implementacija zahtevala manji broj tranzistora. Pošto je ta pretpostavka restriktivnija po simulirano rešenje, usvojeno je da je veliko jezgro četiri puta veće od malog jezgra. Zauzeće prostora na čipu se na dalje određuje u broju malih jezgara.

Komunikaciona mreža je podrazumevana topologija simulatora Gem5. Podrazumevana topologija je Crossbar. Svaki kontroler keš memorije je povezan na mrežni prekidač (eng. *switch*), a svaki mrežni prekidač je povezan na centralni mrežni prekidač. Centralni mrežni prekidač može da prosledi poruku između bilo koja dva mrežna prekidača. Komunikacija preko bilo koje veze traje dve periode takta.

Odabrani skup aplikacija je skup STAMP [140], jer je skup aplikacija koji se koristio u mnogim istraživanjima transakcione memorije. Nivo sinhronizacije u pojedinačnim testovima nije menjan, već se koriste testovi u izvornom obliku. Ono što je specifično je način na koji se ponovo pokreću transakcije, jer to nije definisano u izvornom obliku. Izabrano je da se svaka transakcija ponavlja ograničen broj puta (ograničenje je isto za sve aplikacije). Ukoliko se dostigne maksimalan broj ponavljanja pristupa se sinhronizaciji pomoću globalne brave [141] i na taj način se garantuje napredak u izvršavanju. U slučaju da se transakcija poništava zbog prekoračenja, maksimalan broj ponavljanja se smanjuje na pola. Korišćeni algoritam, je jedan od algoritama, koji je opisan u [142] i koji daje relativno dobre rezultate za sve aplikacije. Pre ponovnog započinjanja transakcije svaka nit sačeka neko vreme. To vreme je pseudoslučajno i bira se iz opsega koji eksponencijalno raste sa brojem ponavljanja te konkretne transakcije. Na taj način pokušano je izbegavanje ponavljanja scenarija koji

dovodi do konflikta i poništavanja transakcija.

Svaka od aplikacija može obrađivati nekoliko različitih skupova podataka. Skupovi podataka se razlikuju u veličini i korišćeni su skupovi podataka koji su predloženi za simulaciju u [140]. Za Bayes aplikaciju simuliranje traje nekoliko nedelja, dok za ostale aplikacije traje dan ili dva. Parametri koji su korišćeni za svaku od aplikacija dati su u tabeli 7.1.

Tabela 7.1: *Parametri sa kojima se pokreću aplikacije u simulaciji*

Aplikacija	Parametri
Bayes	-v32 -r1024 -n2 -p20 -i2 -e2
Genome	-g256 -s16 -n16384
Intruder	-a10 -l4 -n2048 -s1
Kmeans High	-m15 -n15 -t0.05 -i random-n2048-d16-c16
Labyrinth	-i random-x32-y32-z3-n96
Vacation High	-n4 -q60 -u90 -r16384 -t4096
Vacation Low	-n2 -q90 -u98 -r1048576 -t4096
Yada	-a20 -i 633.2

7.1 M-HTM

U ovoj sekciji su opisani eksperimenti koji su se sprovedeni za sistem u kome se transakcije migriraju pomoću predloženog rešenja M-HTM. Opisane su konfiguracije višejezgarnih procesora koje su simulirane i prikazani su dobijeni rezultati. Prikazani rezultati su bazirani na rezultatima objavljenim u radu [143].

7.1.1 Postavka

Analizirano je pet različitih konfiguracija višejezgarnih procesora: simetrični višejezgarni procesor (SMP) sa malim jezgrima, asimetrični višejezgarni procesor (AMP) sa jednim velikim i ostalim malim jezgrima, AMP sa algoritmom za migriranje i bez izvršavanja niti na velikom jezgru (MAMP), AMP sa algoritmom za migriranje i izvršavanjem niti na velikom jezgru (MTAMP) i simetrični višejezgarni procesor sa algoritmom za migriranje (FAMP). Karakteristike svake konfiguracije date su u tabeli 7.2. Veličine svih konfiguracija su varirane od 5 do 36 malih jezgara sa korakom 4. Analiza polazi od 5, da bi svaka konfiguracija imala bar dva jezgra i završava sa 36 da bi svaka konfiguracija mogla da izvršava bar 32 niti.

U tabeli 7.2 date su karakteristike velikog i malog jezgra, kao i mreže koja postoji između njih.

Od postojećih aplikacija nije korišćena aplikacija ssca2, jer verovatnoća za konflikt među transakcijama je mala i ne očekuje se da predloženo rešenje može da poboljša performanse te aplikacije, pošto ni jedna transakcija neće biti izabrana za migraciju. Slične karakteristike ima aplikacija Kmeans, pa se na njenom primeru može videti kako se ponaša predloženo rešenje.

Aplikacije Kmeans i Vacation imaju dve verzije, jedna je sa ređim konfliktima druga sa češćim konfliktima među transakcijama. Za potrebe sagledavanja predloženog rešenja uzeta je samo varijanta sa češćim konfliktima, jer se za tu varijantu transakcije teže izvršavaju paralelno.

7.1.2 Rezultati simulacije

Na slici 7.1 je prikazano ubrzanje koje postižu sve konfiguracije u odnosu na izvršavanje na jednom malom jezgru za sve aplikacije. Merenja su vršena samo za paralelne delove aplikacija, jer je

Tabela 7.2: *Konfiguracija simuliranih višejezgarnih procesora*

Parametar	Veliko jezgro	Malo jezgro
Tip	Superskalarno, 4 instrukcije, 2Ghz	Protočna obrada, 2 instrukcije, 2GHz
L1 Keš[a]	64KB, 4-set asocijativan, 2-ciklusa	16KB, 4-set asocijativan, 1-ciklus
L2 Keš	256KB, 8-set asocijativan, 6-ciklusa	64KB, 8-set asocijativan, 2-ciklusa
L3 Keš	16MB, 32-set asocijativan, 24-ciklusa	
Mreža	16B široka, Crossbar topologija, 2-ciklusa svaka veza	

Konfiguracija[b]	Velika jezgra	Mala jezgra	Maks. broj niti
SMP	0	N	N
AMP	1	N - 4	N - 3
MAMP	1	N - 4	N - 4
MTAMP	1	N - 4	N - 3
FAMP	0	N - 3	N - 4

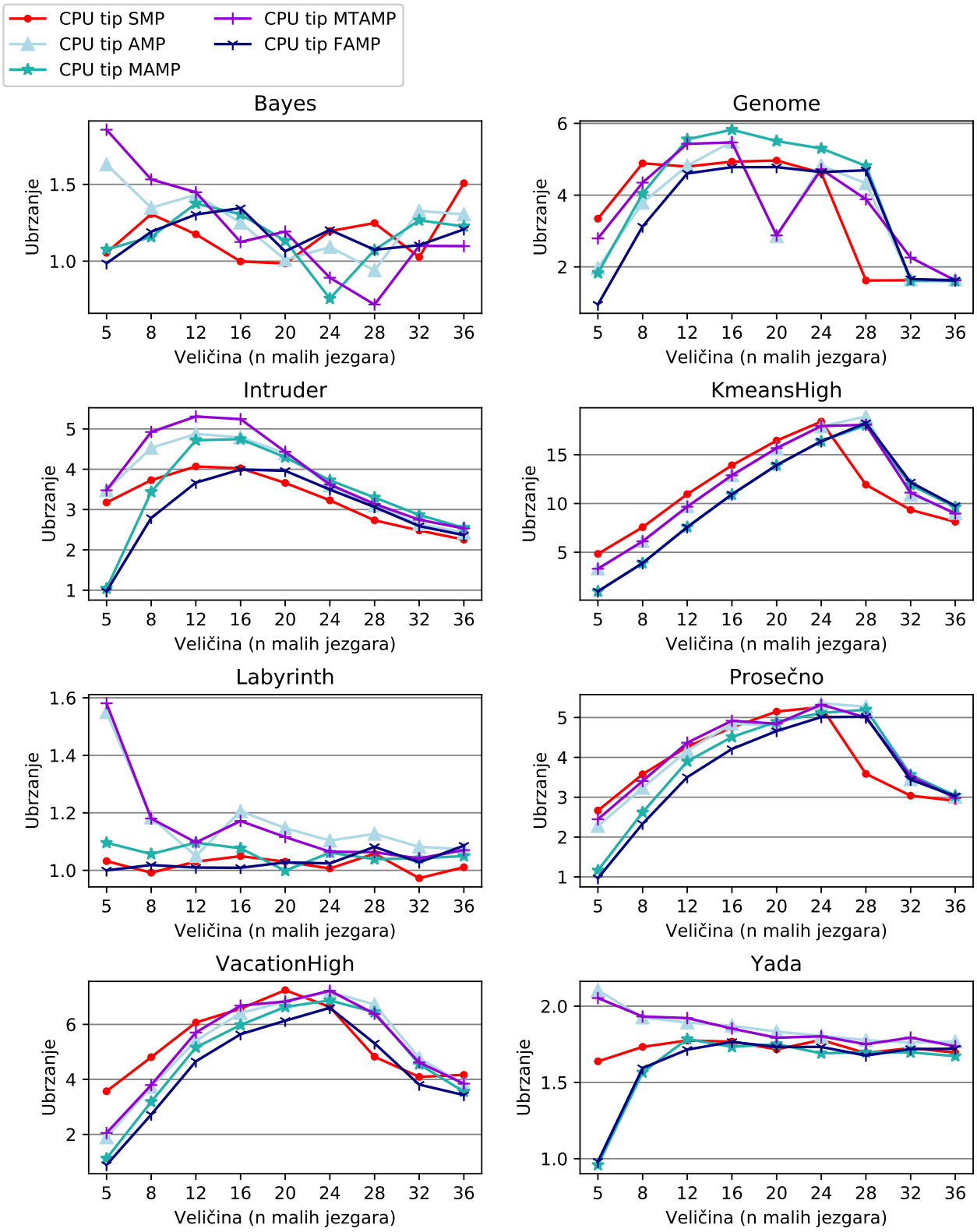
[a] Prikazana je samo keš memorija za podatke.
[b] Konfiguracija je površine N.

ubrzanje serijskog dela aplikacije na asimetričnom višejezgarnom procesoru već razmatrano u [35], [144], [145]. Isti postupci se mogu i ovde primeniti. Na slici 7.2 je prikazan procenat uspešnosti transakcija koji postižu sve konfiguracije za sve aplikacije. Aplikacije su izvršavane sa maksimalnim brojem niti koje dozvoljava određena konfiguracija višejezgarnog procesora i njegova veličina (pogledati tabelu 7.2). Pored grafika za pojedinačne aplikacije, prikazani su i grafici za prosečno ubrzanje i procenat uspešnosti transakcije na nivou svih simuliranih aplikacija.

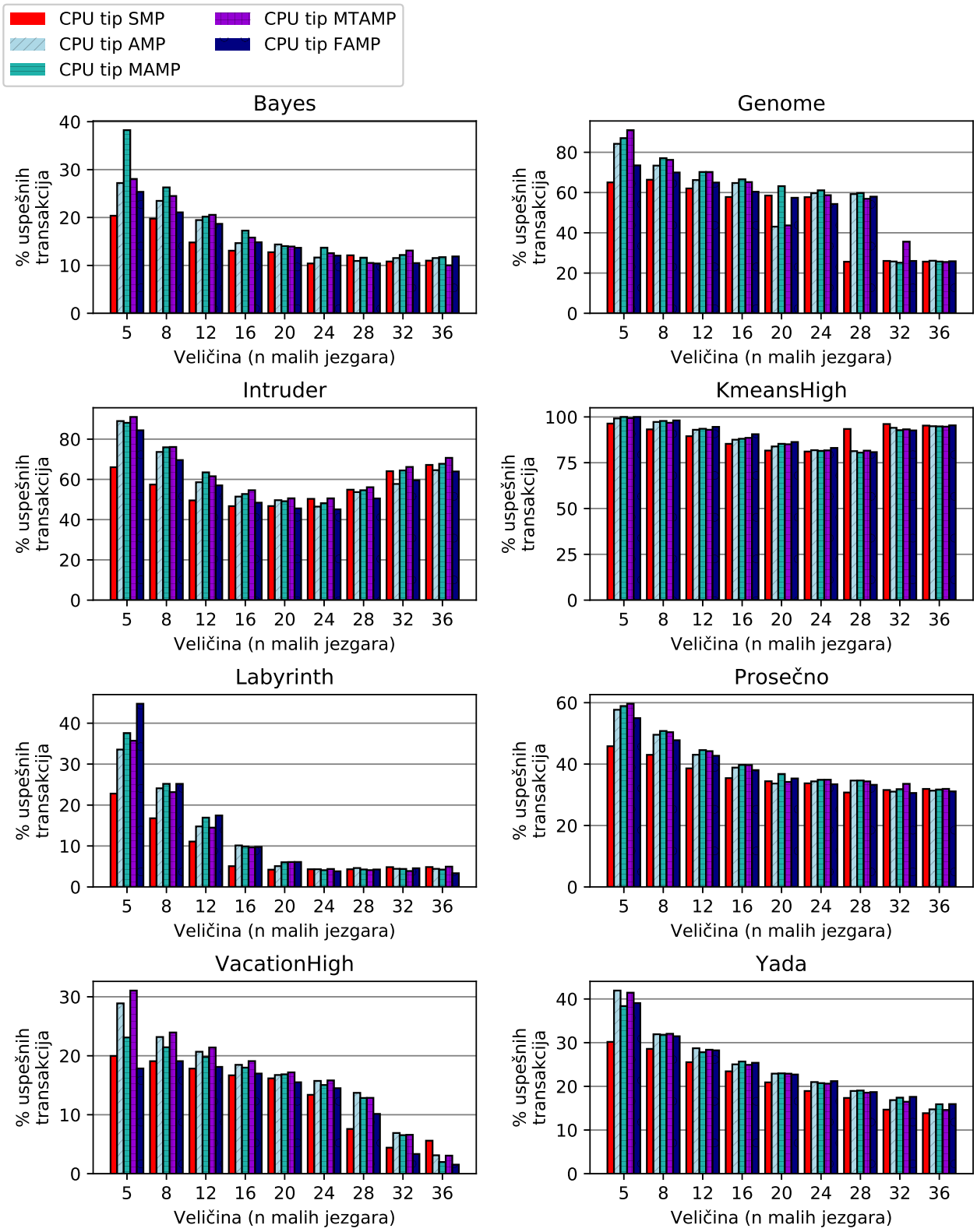
Grafici za konfiguraciju pokazuju da sa porastom broja niti performanse izvršavanja programa Kmeans i Vacation rastu linearno, programa Genome i Intruder sub linearno, dok performanse izvršavanja programa Bayes, Labyrinth i Yada ne rastu ili opadaju. Dobijeni rezultati su u skladu sa rezultatima koje su ostvarili drugi istraživači [88], [142], [146]. Prethodno pomenuta merenja su vršena na postojećim procesorima sa veličinama keš memorija i organizacijom transakcione memorije sličnim procesorima koje smo simulirali. Takođe, rezultati su slični i sa simuliranim sistemima [140]. Dakle, rezultati koje postiže naš simulator ne odstupaju previše od rezultata koji se očekuju na postojećim procesorima.

Neke aplikacija sa povećanjem broja niti imaju jasan trend poboljšanja performansi do neke granice, a nakon toga naglo pogoršanje performansi sa daljim povećanjem. Može se očekivati da procenat uspešnosti izvršavanja transakcija ima blag trend opadanja do iste granice, a onda nagli trend opadanja za te aplikacije. Dok je to slučaj za neke, može se primetiti da to nije slučaj za aplikacije Intruder i Kmeans. To je kontra intuitivno i potrebno je razmotriti te slučajeve pažljivije. Sa povećanjem broja niti, niti imaju sve manje posla, pa sve više čekaju da dobiju posao i transakcije izvršavaju ređe. Pošto su transakcije retke, retko se dešavaju konflikti, pa je procenat uspešnosti transakcije veliki. Stoga se smatra da je dobijeni rezultat u redu i da simulator daje rezultate bliske rezultatima realnog procesora.

Ubrzanja za aplikaciju Genome na AMP i MTAMP konfiguraciji veličine 20 jezgara su izrazito mala i neočekivana. To se dešava zbog prirode same simulacije, jer se teži ponovljivim eksperimentima, pa ne postoji nikakva slučajnost. Ono što se desilo je da se izvršavanje transakcija slučajno učešljalo tako da je broj poništavanja ogroman, što je prouzrokovalo ubrzanje manje od očekivanog. Malom promenom bilo kog parametra može se izbeći nezgodno učešljavanje, a to bi dalo očekivan rezultat. To nije prikazano, jer se žele prikazati rezultati za sva izvršavanja pod istim uslovima. U realnom sistemu i najmanja slučajnost koja bi bila uneta, recimo zbog prekida ili slično, prouzrokovala bi da do nezgodnog učešljavanja ne dođe. Iz tog razloga ovaj rezultat je zanemaren.



Slika 7.1: *Ubrzanje u odnosu na jedno malo jezgro*



Slika 7.2: *Procenat uspešnih transakcija*

Radi boljeg razmatranja šta se dešava tokom izvršavanja aplikacija date su tabele 7.3 i 7.4, koje prikazuju koliko se prosečno, po aplikaciji transakcija, migrira i kolika je uspešnost tih migracija. Prva tabela prikazuje informacije za konfiguraciju MAMP, dok druga prikazuje tabelu za konfiguraciju MTAMP, respektivno.

Kmeans je aplikacija koja se dobro paralelizuje i njene performanse izvršavanja se linearno poboljšavaju sa povećanjem broja jezgara. Takođe, aplikacija ima visok procenat uspešnosti transakcija. Broj transakcija koje se migriraju je neznatan, jer su transakcije kratke i nema mnogo konflikta. Zbog toga je uticaj predloženog rešenja minoran, čak iako se više od pola migriranih transakcija izvrši uspešno. Najbolje performanse izvršavanja daje AMP konfiguracija veličine 28 jezgara. Predloženo rešenje daje neznatno lošije performanse (oko 2% u proseku) u odnosu na AMP konfiguraciju. Razlike se uočavaju zbog poruka koje se šalju na konfiguracijama MTAMP i MAMP, a ne šalju se na konfiguraciji AMP. Pošto se te poruke šalju preko istih kanala kao i poruke koje se koriste u keš koherenciji, dolazi do usporenja izvršavanja pogotovu u konfiguraciji sa većim brojem jezgara. Iz tog razloga bolje je isključiti migraciju za aplikacije ovog tipa, jer predloženo rešenje može samo da smanji performanse izvršavanja.

Aplikacija Vacation se slično dobro paralelizuje kao i Kmeans, ali aplikacija Vacation ima značajno manje procente uspešnosti transakcija. Iz toga se može zaključiti da smanjene broja poništavanja transakcija neće imati značajnijeg efekta na povećanje performansi izvršavanja. Prikazani rezultati pokazuju da konfiguracija MTAMP ima isto ubrzanje kao i konfiguracija SMP, za optimalan broj niti (u proseku MTAMP konfiguracija je neznatno bolja). Migracije na konfiguracijama MTAMP i MAMP povećavaju procenat uspešnosti transakcija, i u proseku i za optimalan broj niti. Procenat transakcija koje se migriraju ispod 10%, dok procenat uspešnih migracija ispod 30%. Iako, migracija pomaže da se smanji broj poništenih transakcija, zbog osobine aplikacije da se dobro paralelizuje, a da je procenat uspešnosti transakcija relativno mali, za ovakve aplikacije bolje isključiti migraciju.

Labyrinth je aplikacija koja ima izuzetno dugačke transakcije koje ne mogu da se izvršavaju zbog prekoračenja. Većina transakcija se izvršava pomoću zaključavanja brave. Postoji mali broj manjih transakcije koje mogu da se izvrše, ali sa porastom broja niti procenat uspešnosti transakcija naglo opada. Pad broja uspešnih transakcija guši ubrzanje koje je samo značajnije za izvršavanja sa dve niti i to je optimalan broj za ovu aplikaciju. Konfiguracija MTAMP daje neznatno bolje ubrzanje nego AMP konfiguracija, jer neznatno povećava procenat uspešnosti transakcija. Procenat migriranih transakcija je oko 27% od ukupnog broja, dok je uspešnost migracija mala (malo više od 1%). Iako, može da čudi sto je procenat uspešnosti transakcija najbolji za konfiguracije MAMP i FAMP na veličini od 5 jezgara, treba obratiti pažnju da ta izvršavanja imaju samo jednu nit, pa do prekida transakcija usled konflikta ni ne dolazi. Međutim, pošto nema ni malo paralelnog izvršavanja ubrzanje je znatno lošije nego za konfiguracije MTAMP i AMP. Ovakvu aplikaciju nema smisla izvršavati na sistemu sa hardverskom transakcionom memorijom, pa nije važno da li treba uključiti ili isključiti migraciju.

Tabela 7.3: *MAMP uspešnost migracije*

Aplikacija	Broj uspešnih migracija (% svih migracija)	Broj migracija	Broj svih transakcija
Bayes	412 (18,0%)	2291 (10,5%)	21748
Genome	3478 (78,7%)	4418 (4,6%)	96079
Intruder	6161 (55,1%)	11179 (6,9%)	162284
Kmeans High	19 (63,3%)	30 (0,0%)	82388
Labyrinth	36 (1,5%)	2430 (27,0%)	8977
Vacation High	4577 (24,6%)	18634 (9,6%)	193499
Yada	245 (1,2%)	20788 (13,0%)	160127

Slično kao i Labyrinth, aplikacija Yada je neskalabilna sa povećanjem broja niti. Razlikuju se u tome što kod aplikacije Yada procenat uspešnosti linearno opada sa povećanjem broja niti, pa ubrzanje sporije opada. Kod ove aplikacije transakcije su kraće, pa se stvara manje konflikata. Ali ova aplikacija

Tabela 7.4: *MTAMP uspešnost migracije*

Aplikacija	Broj uspešnih migracija (% svih migracija)	Broj migracija	Broj svih transakcija
Bayes	181 (8,6%)	2113 (9,1%)	23318
Genome	1901 (50,6%)	3760 (3,9%)	96679
Intruder	11097 (68,4%)	16218 (10,2%)	158742
Kmeans High	17 (74,0%)	23 (0,0%)	82382
Labyrinth	25 (1,0%)	2437 (27,0%)	9035
Vacation High	4299 (28,0%)	15358 (8,3%)	185267
Yada	869 (4,7%)	18556 (11,6%)	160314

takođe ima veliki broj transakcije koje se poništavaju zbog prekoračenja. Opet je optimalan broj niti dva, pa se najbolje performanse postižu na konfiguracijama AMP i MTAMP. Konfiguracija AMP daje neznatno bolji rezultat za optimalan broj niti. Procenat migracija je nizak, malo iznad 10%, dok je procenat uspešnih migracija ispod 5%. Ovakva aplikacija nije pogodna za izvršavanje na sistemima sa hardverskom transakcionom memorijom i treba isključiti migracije, jer migracije ne povećavaju ukupan procenat uspešnosti transakcija.

Aplikacija Bayes se nalazi u sredini između aplikacija Labyrinth i Yada po veličini transakcija i po veličini radnog skupa transakcija. Trend opadanja uspešnosti transakcija sa povećanjem broja niti je linearan, kao kod aplikacije Yada, dok je početna tačka niža. Ubrzanje koje se postiže je, takođe, između aplikacija Labyrinth i Yada. Optimalan broj niti je dva i najbolje performanse se postižu na konfiguraciji MTAMP, koje su za 14% bolje nego na drugim konfiguracijama. Procenat migriranih transakcija je nizak, oko 10% (slično kao kod aplikacije Yada), međutim procenat uspešnosti migracija je značajno veći što doprinosi boljim performansama. Zbog uspešnih migracija i procenat uspešnosti je veći za konfiguracije koje podržavaju migracije. Najveći procenat uspešnosti postiže konfiguracija MAMP. Međutim, ubrzanja koja postižu nisu dobra, jer nema benefita korišćenja velikog jezgra koje imaju konfiguracije AMP i MTAMP. Jedino što značajnije razlikuje Bayes aplikaciju od aplikacija Yada i Labyrinth je to što nije ceo tok izvršavanja u transakcijama [140]. To otvara mogućnost za uspešno migriranje transakcija, pa se predlaže da se za aplikacije koje nisu skalabilne, ali im nije celo izvršavanje u transakcijama, uključi migracija transakcija.

Intruder je aplikacija koja je skalabilna sa povećanjem broja niti, ali ne ekstremno kao aplikacije Kmeans i Vacation. Ima relativno visoke procente uspešnosti transakcija. Konfiguracija MTAMP postiže najbolje performanse i za optimalan broj niti i u proseku. Optimalan broj niti je 9. Procenat migriranih transakcija nije visok (oko 10%), ali je uspešnost migracije visoka (oko 60%) što doprinosi značajnom broju smanjena poništavanja transakcija. To je ujedno i uzrok većeg ubrzanja za konfiguraciju MTAMP. Aplikacije ovog tipa su one za koje predloženo rešenje daje najbolje rezultate i za koje ga treba koristiti.

Slična aplikacija je i aplikacija Genome, koja za razliku od aplikacije Intruder ima manji broj konflikata. To doprinosi da je optimalan broj niti veći (12), da je ubrzanje malo veće i da je procenat uspešnosti transakcija veći. Kao najbolja konfiguracija se pokazala konfiguracija MAMP, koja je u proseku, a i za optimalan broj niti, najbolja. Konfiguracije MAMP i MTAMP povećavaju uspešnost transakcija. Procenat transakcija koje se migriraju je manji od aplikacije Intruder, ali je uspešnost migracije veća. Predloženo rešenje i za ovaj tip aplikacija daje povoljne rezultate, pa se predlaže da se migracija uključi za ovakve aplikacije.

Konfiguracija FAMP je kreirana sa ciljem da se opovrgne početna pretpostavka, a to je da migracijom transakcija na veliko jezgro mogu da se poboljšaju performanse aplikacija i da se smanji broj poništavanja transakcija. Pomenuta konfiguracija ima isti broj jezgara kao i konfiguracija MAMP, vrši se migracija, ali sva jezgra su mala. Na taj način migracija se vrši na malo jezgro. Ideja je da se proveri da li samo migracija utiče na rezultate ili i to što veliko jezgro ubrzava izvršavanje transakcija. Ako samo migracija utiče na poboljšanja, to znači da samo različito učešljavanje transakcija (jer migracija

odlaže početak transakcije za neko nasumično vreme) daje bolje rezultate. U tom slučaju, predloženo rešenje nije potrebno, jer isti efekat može da se postigne softverskim putem, tako što bi se transakcije malo bolje raspoređivale. Rezultati pokazuju da ova konfiguracija daje najgora ubrzanja ili bar jednaka konfiguraciji SMP, kada se upoređivanje radi sa istim brojem niti. Procenat uspešnosti transakcija je malo veći od konfiguracije SMP, ali je niži od ostalih konfiguracija. Iz svega se zaključuje da sama migracija doprinosi smanjenju poništavanja transakcija, ali ne i poboljšanju performansi, što znači da eksperiment nije opovrgao početnu pretpostavku.

7.1.3 Diskusija

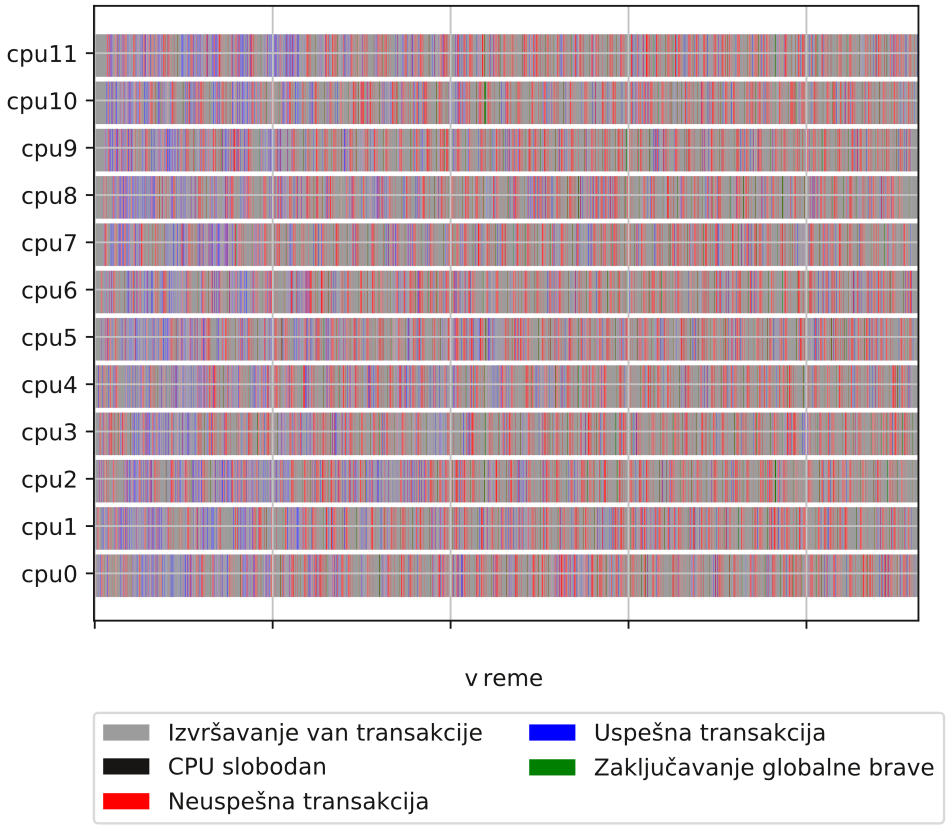
Iz eksperimenata može se zaključiti da predloženi algoritam na konfiguraciji MTAMP (gde veliko jezgro izvršava jednu nit pored migriranih transakcija) može da poboljša performanse nekih aplikacija. To su aplikacije koje ispoljavaju srednji nivo paralelizma i one koje ispoljavaju nizak nivo paralelizma, ali svo izvršavanje nije u transakcijama. Takođe, za drugi tip aplikacije, važno je da transakcije nisu izrazito dugačke, tj. da postoje neke koje se mogu izvršiti bez prekoračenja. Ovaj zaključak se odnosi na izvršavanja u sistemu koji ima komercijalno dostupnu hardversku transakcionu memoriju i asimetrične višejezgarne procesore sa jednim instrukcijskim setom. Predstavljeni algoritam postiže ubrzanja do 14%, u proseku 10%, za sve aplikacije koje su korišćene u poređenju sa sistemom koji ne uzima u obzir postojanje asimetrije.

U eksperimentima su isprobane dve konfiguracije koje koriste asimetriju, MAMP i MTAMP. Prva ne koristi veliko jezgro za izvršavanje niti već samo za izvršavanje transakcija, dok ga druga koristi. To znači da konfiguracija MTAMP izvršava jednu nit više nego konfiguracija MAMP (pogledati tabelu 7.2). Za sve aplikacije osim za aplikaciju Genome, konfiguracija MTAMP je postigla bolje performanse izvršavanja. Posebno je razmotren slučaj aplikacije Genome u nastavku da bi se objasnilo što se taj izuzetak događa. Aplikacija Genome je upoređena sa aplikacijom Intruder, jer su one slične po karakteristikama. Upoređivanje je urađeno za izvršavanje aplikacija sa optimalnim brojem niti.

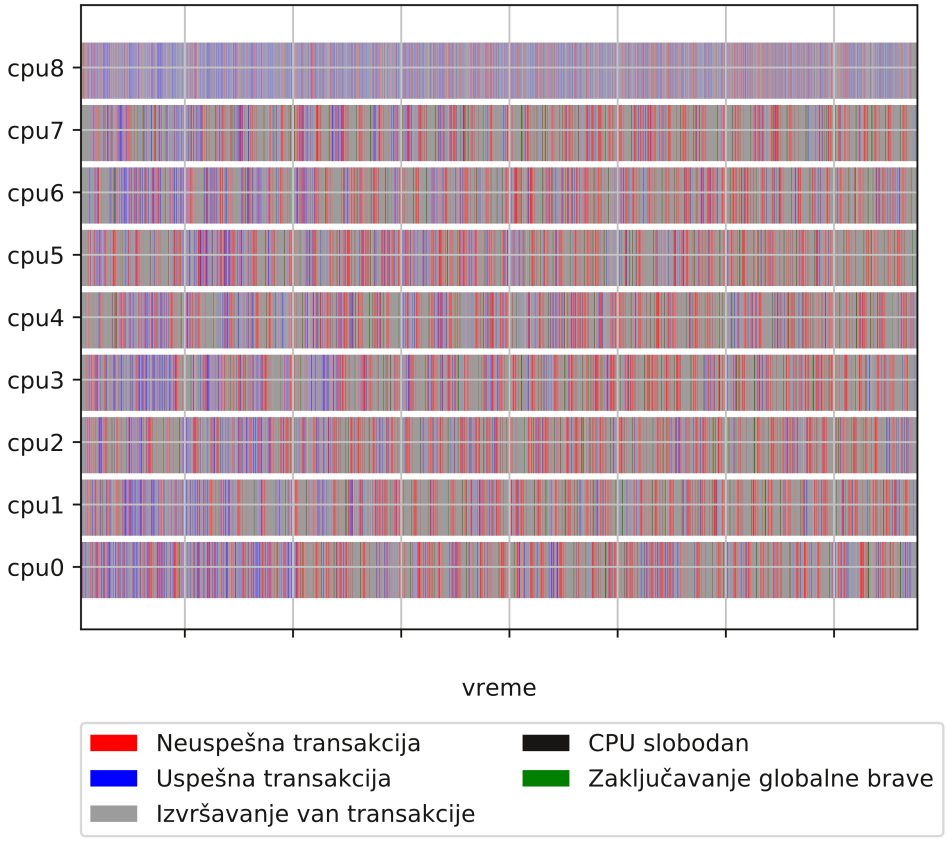
Na slikama od 7.3 do 7.10 su prikazani tragovi izvršavanja za aplikacije Intruder i Genome za sve konfiguracije. Aplikacija Intruder je prikazana za procesor veličine 12 malih jezgara, dok je aplikacija Genome prikazana za procesor veličine 16 malih jezgara. Te veličine procesora su izabrane jer za te procesore obe konfiguracije postižu najbolja ubrzanja. Na svakoj slici je prikazano šta svako jezgro izvršava u vremenu. Vreme je prikazano na horizontalnoj osi. Slike prikazuju samo izvršavanje paralelnih delova aplikacija. Na konfiguracijama koje imaju veliko jezgro, veliko jezgro je jezgro sa najvećim rednim brojem. Trag izvršavanja sadrži prikaz izvršavanje instrukcija u transakcijama, van transakcija i pauziranje u izvršavanju instrukcija (eng. *idle time*). Izvršavanje u transakciji prikazuje se kao uspešno ili neuspešno (bez obzira na razlog). Izvršavanje van transakcija može biti u kritičnoj sekciji sa zaključavanjem bravom ili van kritične sekcije. Dužina izvršavanja svakog dela je srazmerna dužini dela na slikama.

Aplikacija Intruder ima dve faze, što se može najbolje videti na slici 7.3. Prva faza nema mnogo konflikata i transakcije se uspešno izvršavaju. U drugoj fazi se povećava broj konflikata i transakcije se uglavnom neuspešno izvršavaju. Na slici 7.4 može da se primeti da veliko jezgro sve vreme podjednako učestvuje u izvršavanju transakcija, ali ima bolju uspešnost izvršavanja transakcija. Iz toga se može zaključiti da ubrzavanje transakcija značajno povećava šansu da se konflikt izbegne. Ostala jezgra i dalje imaju veliki broj neuspešnih transakcija. Izvršavanje na konfiguraciji MAMP, prikazano na slici 7.5, ima veliki broj migracija u drugoj fazi, koja su uglavnom uspešna, što se poklapa sa prosečnom uspešnošću migracija (pogledati tabelu 7.3). Može se primetiti da mala jezgra takođe imaju manje poništenih transakcija u drugoj fazi, i da zapravo ubrzavanje migriranih transakcija doprinosi smanjenju konflikata. Na slici 7.6 se može videti da migriranje transakcija na veliko jezgro, dok jezgro izvršava neku, ne ugrožava uspešnost transakcija. U drugoj fazi izvršavanja, konflikti su takođe smanjeni.

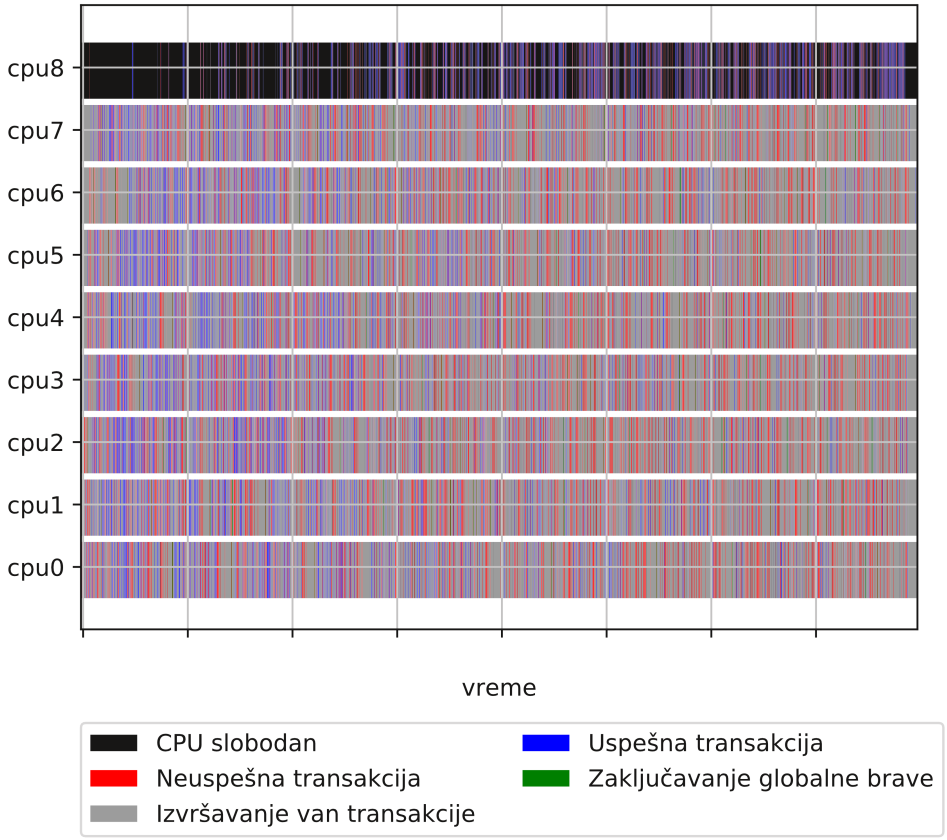
Aplikacija Genome ima četiri faze izvršavanja, što se može videti na slici 7.7. Prva faza ima



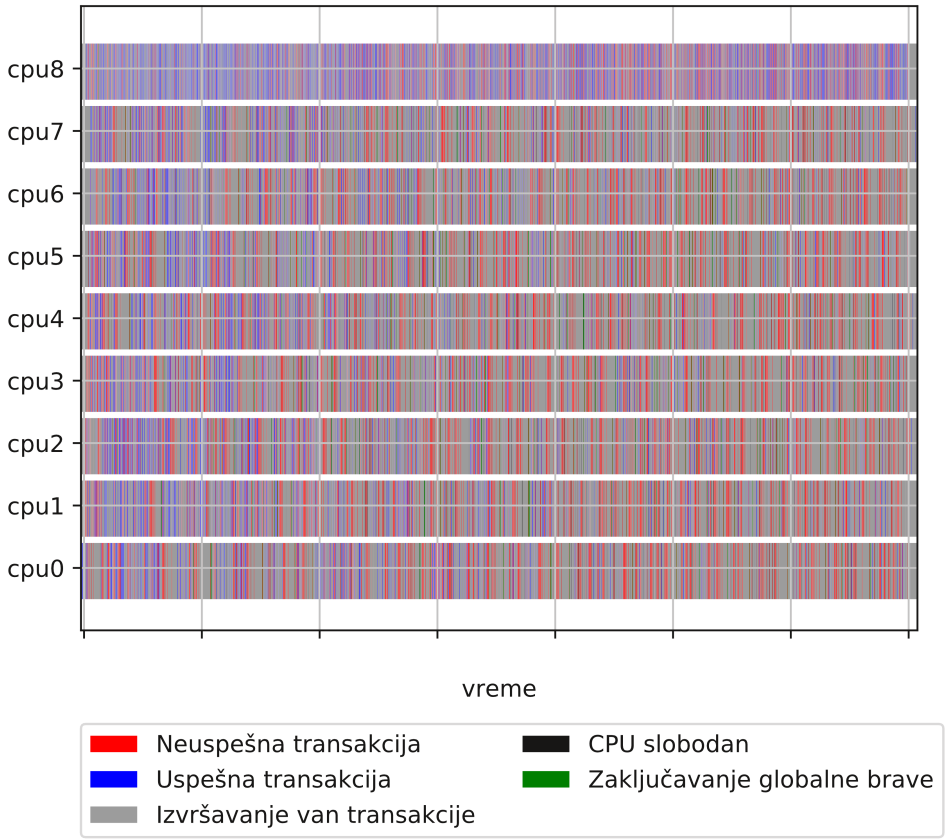
Slika 7.3: *Izvršavanja STAMP Intruder aplikacije na SMP procesoru veličine 12 malih jezgara*



Slika 7.4: *Izvršavanja STAMP Intruder aplikacije na AMP procesoru veličine 12 malih jezgara*



Slika 7.5: *Izvršavanja STAMP Intruder aplikacije na MAMP procesoru veličine 12 malih jezgara*



Slika 7.6: *Izvršavanja STAMP Intruder aplikacije na MTAMP procesoru veličine 12 malih jezgara*

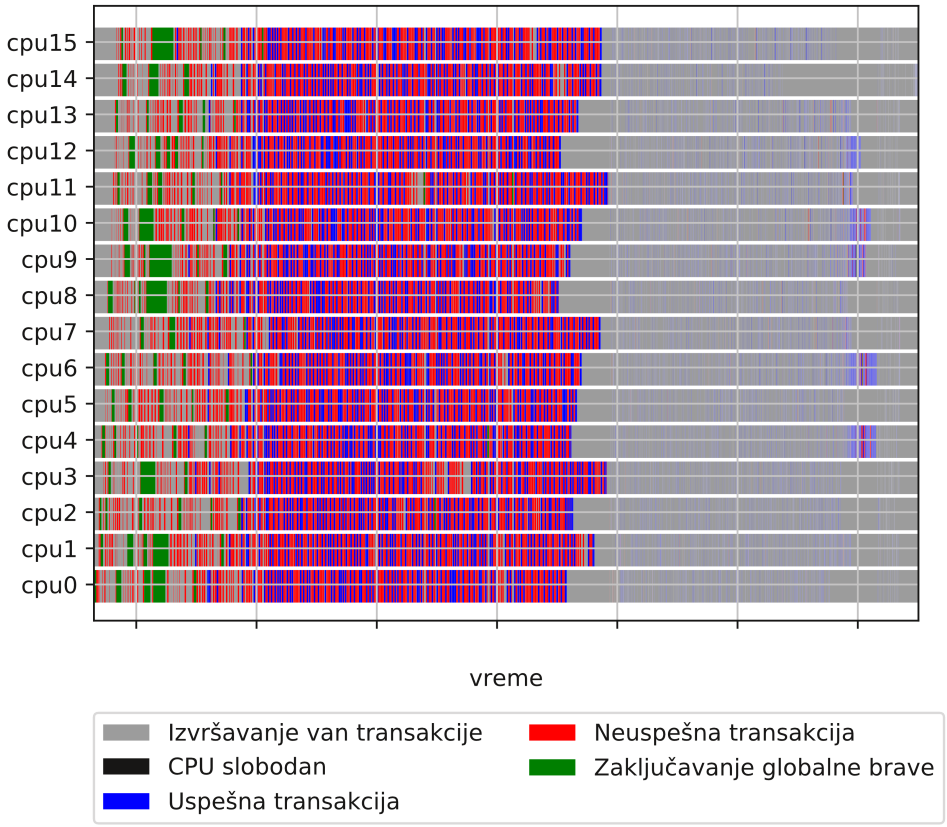
izuzetno mnogo poništavanja transakcija, i na kraju se uglavnom kritična sekcija izvrši pomoću zaključavanja. Druga faza, ujedno i najduža, ima polovičan uspeh transakcija zbog konflikata. Treća i četvrta faza su slične i u njima ima malo konflikta, dok su transakcije većinom uspešne. Takođe, primećuje se da se prelasci između faza rade pomoću barijere, što znači da sve niti moraju završiti prethodnu fazu pre nego što se pređe na sledeću. Ako neka nit završi fazu ranije od ostalih, ona uposleno čeka na barijeri. Uposleno čekanje svakako ne utiče na performanse, jer se u simulaciji izvršava jedna aplikacija, a svaka nit ima svoje jezgro. Na slici 7.8 može se videti da veliko jezgro brzo izvrši svoj posao i da dugo čeka na ostale niti. Veliko jezgro u drugoj fazi ima značajno veliki procenat uspešnih transakcija, pa se može zaključiti da brže izvršavanje transakcija može smanjiti konflikte. Konfiguracija MAMP, čije izvršavanje prikazano na slici 7.9, značajno poboljšava performanse jer se u drugoj fazi uspešno migrira veliki broj niti. Može se primetiti da veliko jezgro maltene sve vreme tokom druge faze izvršava transakcije, skoro bez pauze. Uspešno migriranje doprinosi da se konflikti smanje i da procenat uspešnosti svih transakcija bude povećan u odnosu na druge konfiguracije. Prva, treća i četvrta faza su nepromenjene u odnosu na ostale slučajeve, ali prva faza u ovom slučaju ima manje neuspešnih migracija. Na slici 7.10 može da se primeti da veliko jezgro pored svoje niti izvršava i veliki broj migriranih transakcija. Velika navala na veliko jezgro prouzrokuje da se ta nit uspori toliko da produži čekanje ostalih niti na barijeri. Ovaj problem nastaje samo u ovom simuliranom okruženje gde su niti fiksirane na jedno i samo jedno jezgro. U realnom sistemu, niti će biti raspoređivane na različita jezgra. Tada neće samo jedna nit biti usporavana, pa do ovog problema neće doći. Možemo se zaključiti da će konfiguracija MTAMP u realnom sistemu postizati bolje performanse izvršavanja nego što je to u simuliranom okruženju.

Ne treba izgubiti iz vida ranije navedenu činjenicu da simulirano veliko jezgro ne podržava višenitno izvršavanje. Trenutni razvoj tehnologije dozvoljava implementacije u kojima veliko jezgro ima podršku za višenitno izvršavanje, gde pod višenitno podrazumevamo najčešću implementaciju sa podrškom za dve niti. U sistemima sa takvom konfiguracijom velikog jezgra došlo bi do povećanja paralelnog rada, čije je smanjenje smetalo nekim aplikacijama, i postojala bi mogućnost za ubrzanjem većeg broja transakcija u fazama aplikacije koje imaju veći broj konflikata među transakcijama.

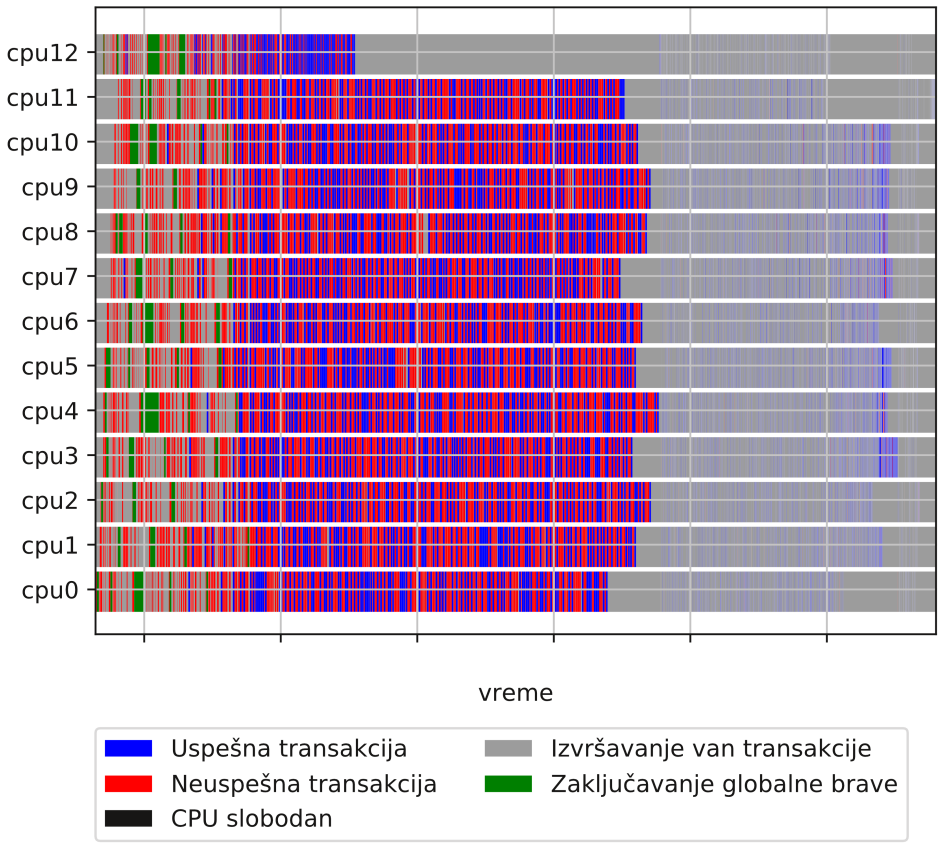
Primećeno je i da postoje neke transakcije koje su uzaludno migrirane i koje su prekinute zbog konflikta. Da bi se sprečio gubitak vremena na migriranje moguće je napraviti veliko jezgro takvo da njegove transakcije imaju prioritet u odnosu na transakcije na malim jezgrima, pa da se konflikt razrešava poništavanjem transakcija na malom jezgru. Generalno, postoji bojazan da je rešenje sa kompleksnim razrešavanjem konflikata značajno teže za verifikaciju [147], ali u ovom slučaju razrešavanje konflikata nije kompleksno, jer samo treba odložiti odgovor na poruke koje bi izazvale poništenje transakcije. To je nešto što treba ispitati u budućnosti, jer bi moglo da doprinese poboljšanju predloženog rešenja.

U eksperimentima su analizirane aplikacije koje se izvršavaju bez drugih aplikacija u sistemu. Predloženo rešenje može se da radi i u slučaju kada se više aplikacija izvršava istovremeno na procesoru sa malim modifikacijama samog rešenja. Najvažnija promena je da malo jezgro ima informaciju koji proces se izvršava na velikom jezgru. Malo jezgro bi preskočilo migraciju ukoliko se na velikom jezgru izvršava različiti proces u odnosu na njegov. Raspoređivač operativnog sistema mogao bi da odlučuje koji proces ima veći prioritet i da njega rasporedi na veliko jezgro. Transakcije tog procesa će biti ubrzavane migracijom. Druge manje promenu su da centralni i keš registar predloženog rešenja imaju informaciju kom procesu transakcija pripada (npr. da se doda identifikator procesa za svaku transakciju), da se prema potrebi podese veličine centralnog i keš registara, itd.

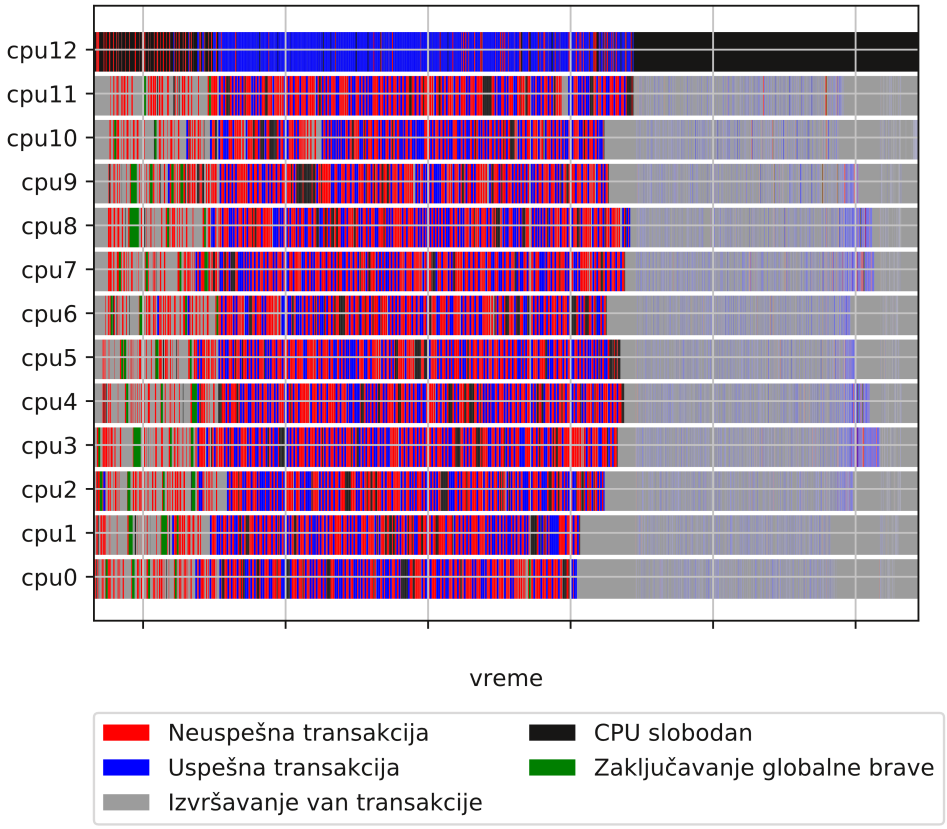
Simulacije su koristile samo jednu topologiju komunikacione mreže između jezgara. Ta topologija se ne sreće često u realnim sistemima. U budućem radu treba se ispitati kakav uticaj ima topologija mreže na migraciju transakcija i kako promena kašnjenja utiče na korist od migracija.



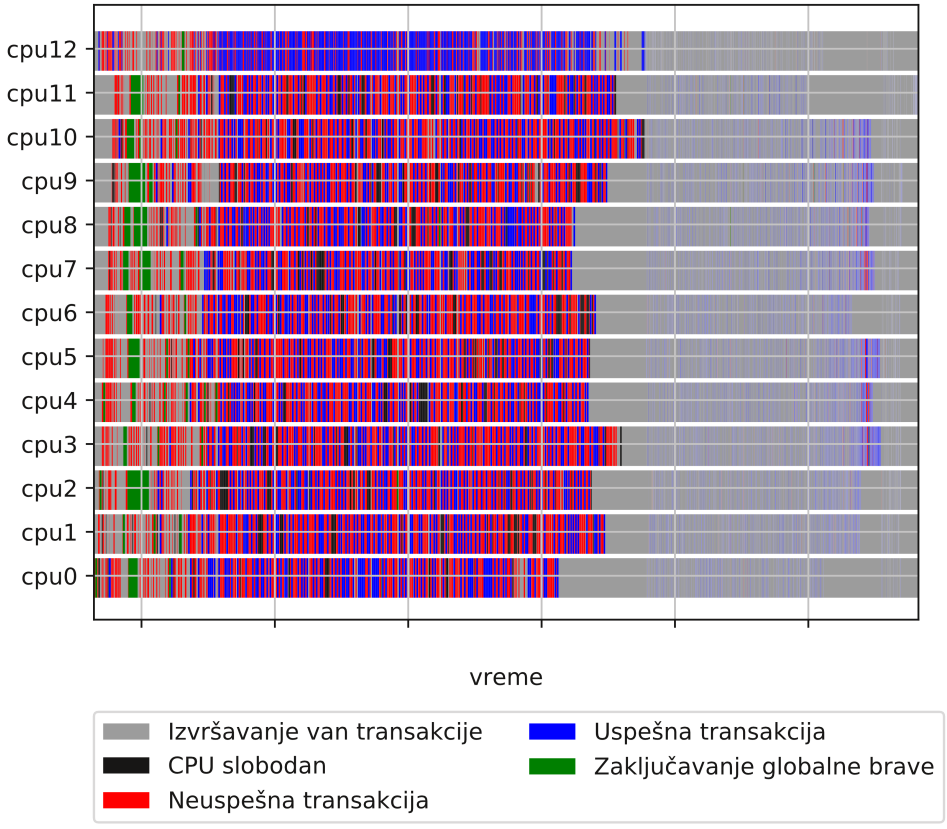
Slika 7.7: *Izvršavanja STAMP Genome aplikacije na SMP procesoru veličine 16 malih jezgara*



Slika 7.8: *Izvršavanja STAMP Genome aplikacije na AMP procesoru veličine 16 malih jezgara*



Slika 7.9: *Izvršavanja STAMP Genome aplikacije na MAMP procesoru veličine 16 malih jezgara*



Slika 7.10: *Izvršavanja STAMP Genome aplikacije na MTAMP procesoru veličine 16 malih jezgara*

7.2 Prilagođeni podsistem keš memorija

U ovoj sekciji su opisani eksperimenti koji su se sprovedeni za sistem u kome se nalazi prilagođeni podsistem keš memorija. Opisane su konfiguracije višejezgarnih procesora koje su simulirane i prikazani su dobijeni rezultati. Na kraju su dati zaključci koji su izvedeni na osnovu rezultata.

7.2.1 Postavka

Analizirano je 18 konfiguracija. Zajedničke karakteristike svih konfiguracija date su tabeli 7.5. Veličine svih konfiguracije procesora varirane su od 5 do 36 malih jezgara sa korakom 4. Kreće se od 5 da bi svaka konfiguracija imala bar dva jezgra i završava na 36 da bi svaka konfiguracija mogla da izvršava bar 32 niti. Konfiguracije se razlikuju u karakteristikama i vrsti keš memorija prvog i drugog nivoa. Konfiguracije su podeljenje u dve grupe. Prva grupa sadrži konfiguracije (SMP) koje imaju simetrični višejezgarni procesor, dok druga grupa sadrži konfiguracije (AMP) koje imaju asimetrični višejezgarni procesor. Svaka od dve grupe ima po dve podgrupe. Jedna podgrupa sadrži konfiguracije sa konvecionalnim podsistemom keš memorija, dok druga podgrupa sadrži konfiguracije sa prilagođenim podsistemom keš memorija. Skraćenice druge podgrupe imaju dodato slovo P uz osnovnu skraćenicu (SMPP i AMPP).

Tabela 7.5: *Konfiguracija simuliranih višejezgarnih procesora sa zajedničkim podešavanjima jezgara za sve konfiguracije*

Parametar	Veliko jezgro	Malo jezgro
Tip	Superskalarno, 4 instrukcije, 2Ghz	Protočna obrada, 2 instrukcije, 2GHz
L3 Keš	16MB, 32-set asocijativan, 24-ciklusa	
Mreža	16B široka, Crossbar topologija, 2-ciklusa svaka veza	

Konfiguracija[a]	Velika jezgra	Mala jezgra	Maks. broj niti
SMP i SMPP	0	N	N
AMP i AMPP	1	N - 4	N - 3

[a] Konfiguracija je površine N.

Karakeristike keš memorija za prvu podgrupu (konfiguracije SMP i AMP) su date u tabeli 7.6. Karakteristike keš memorija malih jezgara za drugu podgrupu (konfiguracije SMPP i AMPP) date su

Tabela 7.6: *Podešavanja keš memorija za SMP i AMP konfiguracije*

Keš memorija	Konfiguracija	Malo jezgro			Veliko jezgro		
		LL	LM	LH	LL	LM	LH
L1	Latencija	1	2	3	2	4	6
	Skup	4			8		
	Veličina	64 KB			128 KB		
L2	Latencija	2	4	6	3	5	7
	Skup	8			12		
	Veličina	128 KB			256 KB		

Tabela 7.7: *Podešavanja podeljenih keš memorija malog jezgra*

Keš memorija	Konfiguracija	SL AL	SL AM	SL AH	SH AL	SH AM	SH AH
L1 Prostorni deo	Latencija	2					
	Skup	1	2	4	1	2	4
	Veličina	32 KB					
L1 Vremenski deo	Latencija	1					
	Skup	1					
	Veličina	8 KB			16 KB		
L2 Prostorni deo	Latencija	4					
	Skup	1	4	8	1	4	8
	Veličina	64 KB					
L2 Vremenski deo	Latencija	1			2		
	Skup	1					
	Veličina	16 KB			32 KB		

u tabeli 7.7, dok su konfiguracije keš memorija velikih jezgara data u tabeli 7.8. Sve keš memorije imaju set-asocijativno mapiranje. Za svaki deo keš memorija, prikazano je koliko je vreme pristupa u periodama ciklusa takta (red sa nazivom latencija), veličina skupa prilikom set-asocijativnog mapiranja (red sa nazivom skup) i veličina u bajtovima (red sa nazivom veličina). Obratiti pažnju da set-asocijativno mapiranje sa veličinom skupa od jednog elementa je zapravo direktno mapiranje. Veličina ulaza u vremenskom delu keš memorija je jedna reč, dok je veličina u prostornim delovima osam reči. Reč je širine 64 bita. Veličina keš memorija je odabrana da bude malo veća nego što je u trenutno komercijalno dostupnim procesorima, jer je trend da se sa svakom novom generacijom procesora poveća veličina. Veličina delova podeljenih keš memorija je uzeta tako da se, u zavisnosti od konfiguracije, između $\frac{3}{8}$ i $\frac{1}{4}$ prostora uštedi.

Tabela 7.8: *Podešavanja podeljenih keš memorija velikog jezgra*

Keš memorija	Konfiguracija	SL AL	SL AM	SL AH	SH AL	SH AM	SH AH
L1 Prostorni deo	Latencija	4					
	Skup	1	4	8	1	4	8
	Veličina	64 KB					
L1 Vremenski deo	Latencija	1			2		
	Skup	1					
	Veličina	16 KB			32 KB		
L2 Prostorni deo	Latencija	5					
	Skup	1	6	12	1	6	12
	Veličina	128 KB					
L2 Vremenski deo	Latencija	2			2		
	Skup	1					
	Veličina	32 KB			64 KB		

Konfiguracije sa konvencionalnim podsistemom keš memorija se razlikuju u vremenu pristupa keš memorija. Simulirana su tri različita vremena. Najmanje vreme pristupa imaju konfiguracije sa ozna-

kom LL, srednje sa oznakom LM i najviše sa oznakom LH. Vremena pristupa su tako odabrana da su u prvom slučaju za konvencionalu keš memoriju kraća nego prostorni deo podeljene keš memorije, u drugom je vreme pristupa isto, dok u trećem je vreme pristupa duže. U zavisnosti od tehnologije implementacije, na realnom čipu će verovatno biti drugi ili treći slučaj, dok je prvi slučaj simuliran samo zbog teorijskog razmatranja gornje granice.

Konfiguracije sa prilagođenim podsistemom keš memorija se razlikuju u veličini vremenskog dela keš memorija i u veličini skupova prilikom set-asocijativnog mapiranja kod prostornog dela keš memorija. Oznaka SL u konfiguraciji znači da vremenski deo manji, dok SH označava da je veći. Prvi slučaj je da vremenski deo ima isti broj ulaza kao i prostorni deo, dok u drugom slučaju vremenski deo ima duplo više ulaza nego prostorni deo. Oznake AL, AM i AH u konfiguracijama znače da je veličina skupa mala, srednja ili velika kod set-asocijativnog mapiranja.

Kod svih konfiguracija sa prilagođenim keš podsistemom za brojač **x** je korišćena granica 32, a za brojač **y** 64. Vrednosti za granice su uzete iz [1], zato što je pokazano da su to najbolje vrednosti za jezgro.

Za razliku od eksperimenata za M-HTM rešenje nije simulirana aplikacija Bayes, već aplikacija Vacation Low. Simulacija Bayes aplikacije bi trajala nekoliko meseci. S obzirom na to da se ona ne skalira sa povećanjem broja niti, nedostatak te aplikacije ne utiče značajno na konačan sud o rešenju.

7.2.2 Rezultati simulacije

Na slikama 7.11 i 7.13 su prikazana ubrzanja koje postižu sve konfiguracije u odnosu na izvršavanje na jednom malom jezgru za sve aplikacije. Prva slika je za konfiguracije sa simetričnim višejezgarnim procesorom, dok je druga slika za konfiguracije sa asimetričnim višejezgarnim procesorom. Malo jezgro čije je izvršavanje uzeto kao osnova za poređenje ima konvencionalan podsistem keš memorija. Na slikama 7.12 i 7.14 je prikazan procenat uspešnosti transakcija koji postižu sve konfiguracije za sve aplikacije. Prva slika je za konfiguracije sa simetričnim višejezgarnim procesorom, dok je druga slika za konfiguracije sa asimetričnim višejezgarnim procesorom. Aplikacije su izvršavane sa maksimalnim brojem niti koje dozvoljava određena konfiguracija višejezgarnog procesora i njegova veličina (pogledati tabelu 7.5). Pored grafika za pojedinačne aplikacije, prikazani su i grafici za prosečno ubrzanje i procenat uspešnosti transakcije na nivou svih simuliranih aplikacija za konfiguracije sa jednim tipom višejezgarnog procesora.

Grafici pokazuju da je trend rasta i opadanja linija sličan opisanim eksperimentima za M-HTM rešenje, pa je zaključeno da ove simulacije ne odstupaju previše od rezultata koji se očekuju na postojećim procesorim. Apsolutna ubrzanja koja se prikazuju ovde se razlikuju od već opisanih eksperimenata, jer je konfiguracija keš memorija drugačija (nisu iste veličine) i zbog toga što je ovde prikazano ubrzanje za celokupne aplikacije, a ne samo za paralelni deo.

Konfiguracije sa konvencionalnim keš memorijama imaju slične krive, samo se razlikuju u apsolutnim vrednostima. Konfiguracije LL, one sa najkraćim vremenom pristupa keš memorijama, imaju očekivano najveće ubrzanje. LL konfiguracije su neprikoslovene i nijedna druga konfiguracija ne može da se poredi sa njima. To je i očekivano jer one imaju najmanje vreme pristupa keš memorija u odnosu na sve ostale konfiguracije. Ta vremena pristupa su nerealna i prikazane performanse su teorijski limit, koji pokazuje koliko keš memorije mogu da ubrzaju aplikacije kada bi se implementirale skoro idealno. Konfiguracije LM i LH, kod kojih je povećano vremena pristupa, imaju srazmerno smanjeno ubrzanje.

Konfiguracije SL AL, SH AL i SL AM, simetričnih višejezgarnih procesora sa podeljenom keš memorijom koje imaju mali vremenski deo ili malu veličinu skupova u set-asocijativnom mapiranju se ne skaliraju dobro sa povećanjem broja niti. Procenat uspešnih transakcija je značajno niži u odnosu na ostale konfiguracije. Taj loš procenat je prouzrokovan čestim poništavanjem transakcija usled prekoračenja, jer keš memorije ne mogu da čuvaju sve podatke koje su potrebne transakcijama. Tome najviše doprinosi mala veličina skupa koja prouzrokuje više zamena podataka iz keš memorije.

Zamene prouzrokuju poništavanje transakcije zbog izbacivanja podataka u koje se upisalo za vreme transakcije. Aplikacije koje se ne skaliraju dobro sa povećanjem broja niti, Yada i Labyrinth, su manje osetljive u odnosu na ostale i kod njih je ubrzanje koje postižu ove konfiguracije uporedivo sa ostalim konfiguracijama.

Konfiguracije SL AH i SH AM simetričnih višejezgarnih procesora sa podeljenom keš memorijom, koje imaju veće skupove kod set-asocijativnog mapiranja, se bolje skaliraju sa povećanjem broja niti. Procenat uspešnih transakcija je povećan, ali i dalje ne može da se poredi sa konfiguracijama koje imaju konvencionalnu keš memoriju. Povećanje veličine skupa doprinosi da se manje podataka zamenjuje u keš memorijama, pa se manje transakcija poništava. Slično kao i prethodna grupa konfiguracija, aplikacije Yada i Labyrinth imaju slično ubrzanje kao i ostale konfiguracije. Aplikacije koje imaju manje skupove podataka u transakcijama, Kmeans, Intruder i Genome se dobro skaliraju na ovim konfiguracijama i ubrzanje koje se postiže nadmašuje konfiguraciju LH sa konvencionalnim keš memorijama. Aplikacije Vacation Low i Vacation High imaju malo veće skupove podataka u transakcijama pa je i ubrzanje koje daju ove transakcije manje i lošije od konfiguracija sa konvencionalnim keš memorijama.

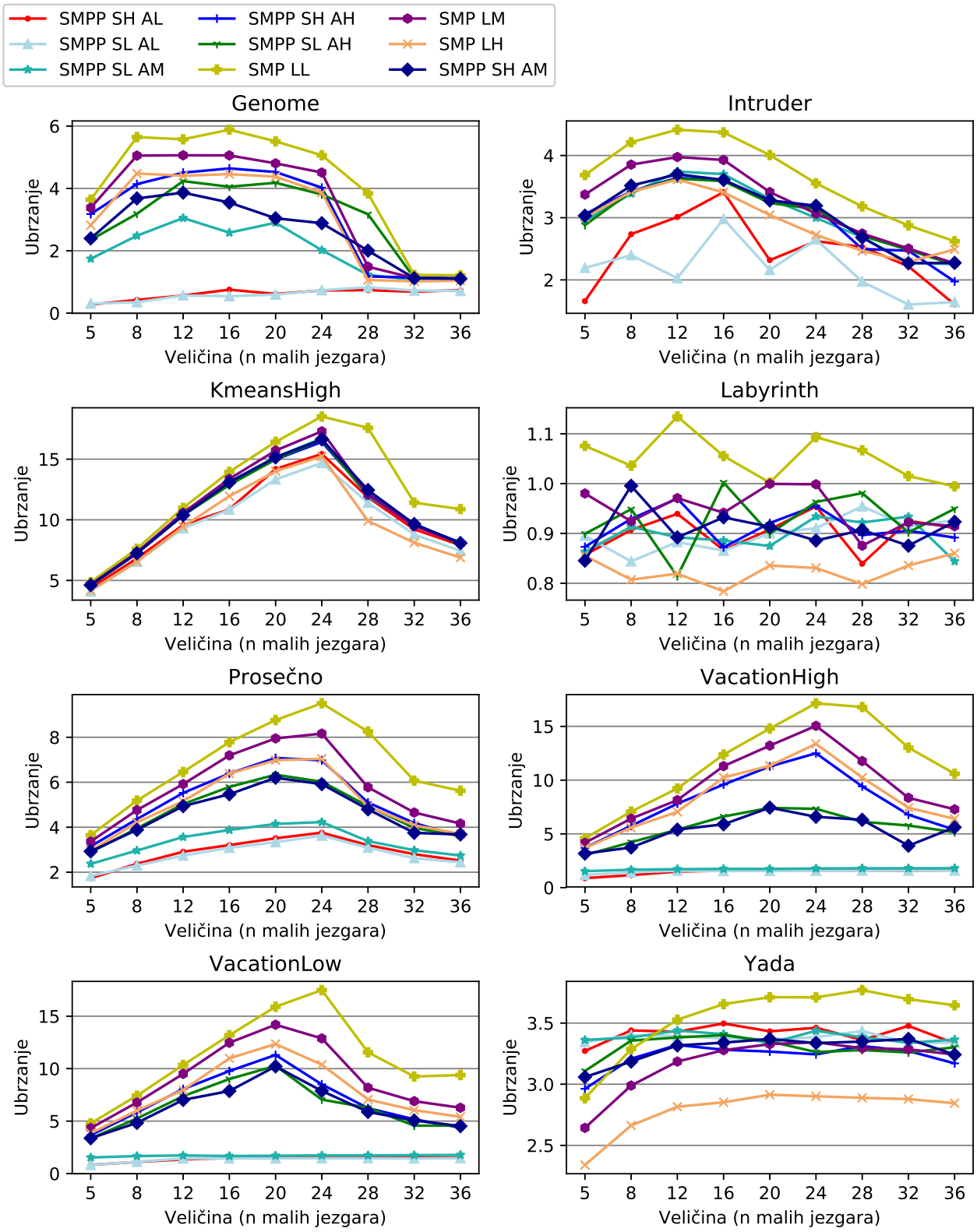
Konfiguracija SH AH simetričnog višejezgarnog procesora sa podeljenom keš memorijom, ima najveću veličinu vremenskog dela keš memorija i skupova kod set-asocijativnog mapiranja. Procenat uspešnih transakcija je najveći u odnosu na sve konfiguracije sa podeljenim keš memorija, ali je manji od svih konfiguracija sa konvencionalnom keš memorijom. U proseku postiže bolje ubrzanje nego LH konfiguracija, kao i za sve aplikacije osim za aplikacije Vacation High i Vacation Low. Ubrzanje koje je kod njih postignuto neznatno je lošije nego kod konfiguracije LH.

Konfiguracije SL AL, SH AL i SL AM, asimetričnih višejezgarnih procesora sa podeljenom keš memorijom koje imaju mali vremenski deo ili malu veličinu skupova u set asocijativnom mapiranju se ne skaliraju dobro sa povećanjem broja niti. Pokazuju iste karakteristike kao i te konfiguracije za simetrični višejezgarni procesor. Međutim, postoji jedna značajna razlika. Aplikacije koje se ne skaliraju dobro, Yada i Labyrinth, imaju najveće ubrzanje na tim konfiguracijama. Procenat uspešnosti transakcija je viši u odnosu na ostale konfiguracije za manji broj niti. To se dešava, jer veliko jezgro brže počne da izvršava kritične sekcije sa zaključavanjem. Ranije se počinje sa zaključavanjem, jer se transakcije ranije prekidaju zbog prekoračenja koje je prouzrokovano konfiguracijama keš memorija. Ove aplikacije se svakako najbolje izvršavaju na velikom jezgru, bez ostalih niti, jer u suprotnom veliko jezgro samo mora još dodatno uzaludno da čeka malo jezgro koje sa vremena na vreme zaključava bravu.

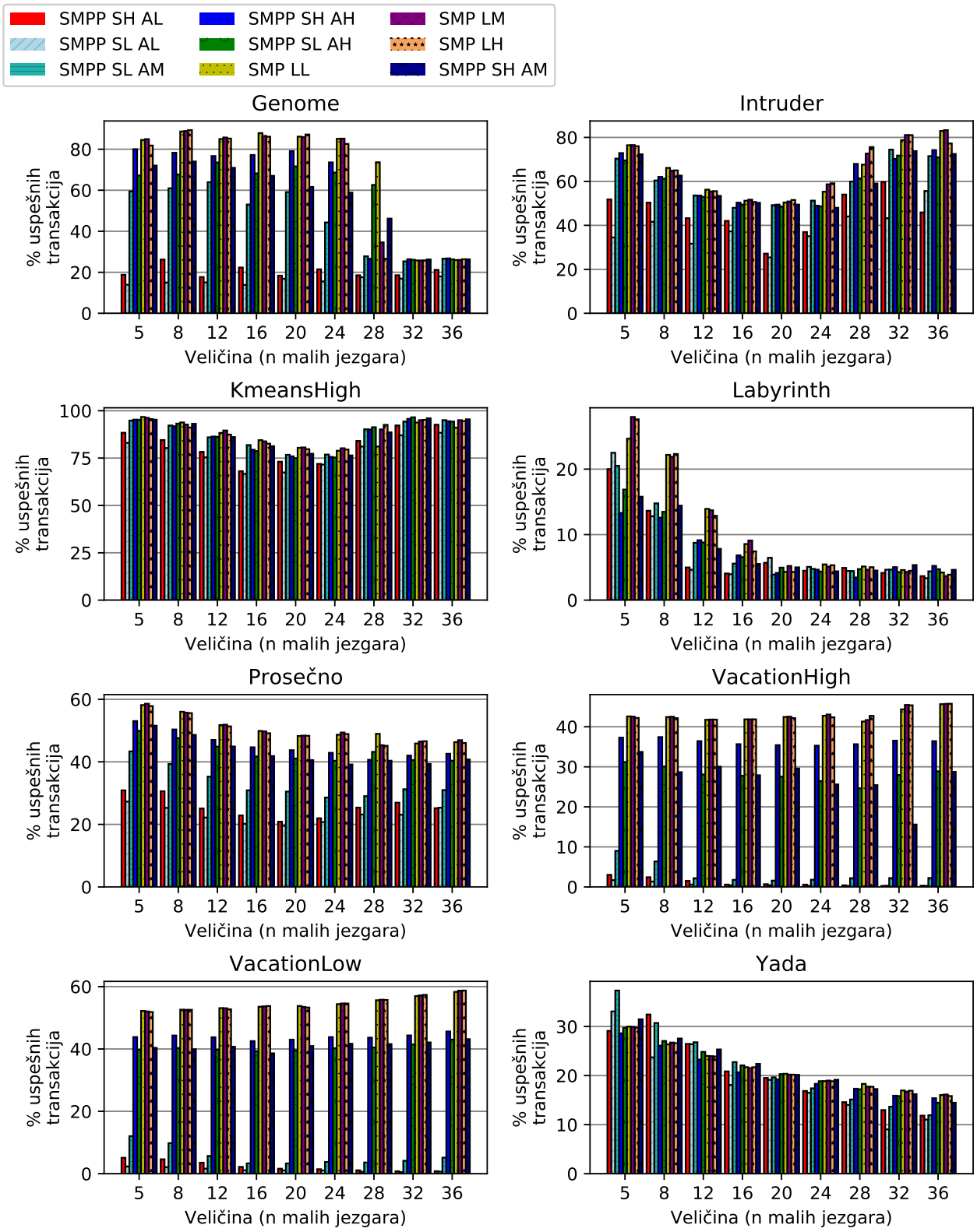
Konfiguracije SL AH i SH AM asimetričnih višejezgarnih procesora sa podeljenom keš memorijom, koje imaju veće skupove kod set-asocijativnog mapiranja, se bolje skaliraju sa povećanjem broja niti. Pokazuju iste karakteristike kao i te konfiguracije za simetrični višejezgarni procesor.

Konfiguracija SH AH asimetričnog višejezgarnog procesora sa podeljenom keš memorijom, ima najveću veličinu vremenskog dela i skupova kod set-asocijativnog mapiranja. Procenat uspešnih transakcija je najveći u odnosu na sve konfiguracije sa podeljenim keš memorijama, ali je manji od svih konfiguracija sa konvencionalnom keš memorijom. U proseku postiže značajno bolje ubrzanje nego LH konfiguracija, kao i za sve aplikacije osim za aplikacije Vacation High i Vacation Low. Ubrzanje koje je kod njih postignuto neznatno je lošije nego kod konfiguracije LH. Konfiguracija za aplikaciju Genome postiže iste rezultate kao i konfiguracija LL. Ubrzanje koje postiže prilikom izvršavanja aplikacije Kmeans je isto kao konfiguracija LM.

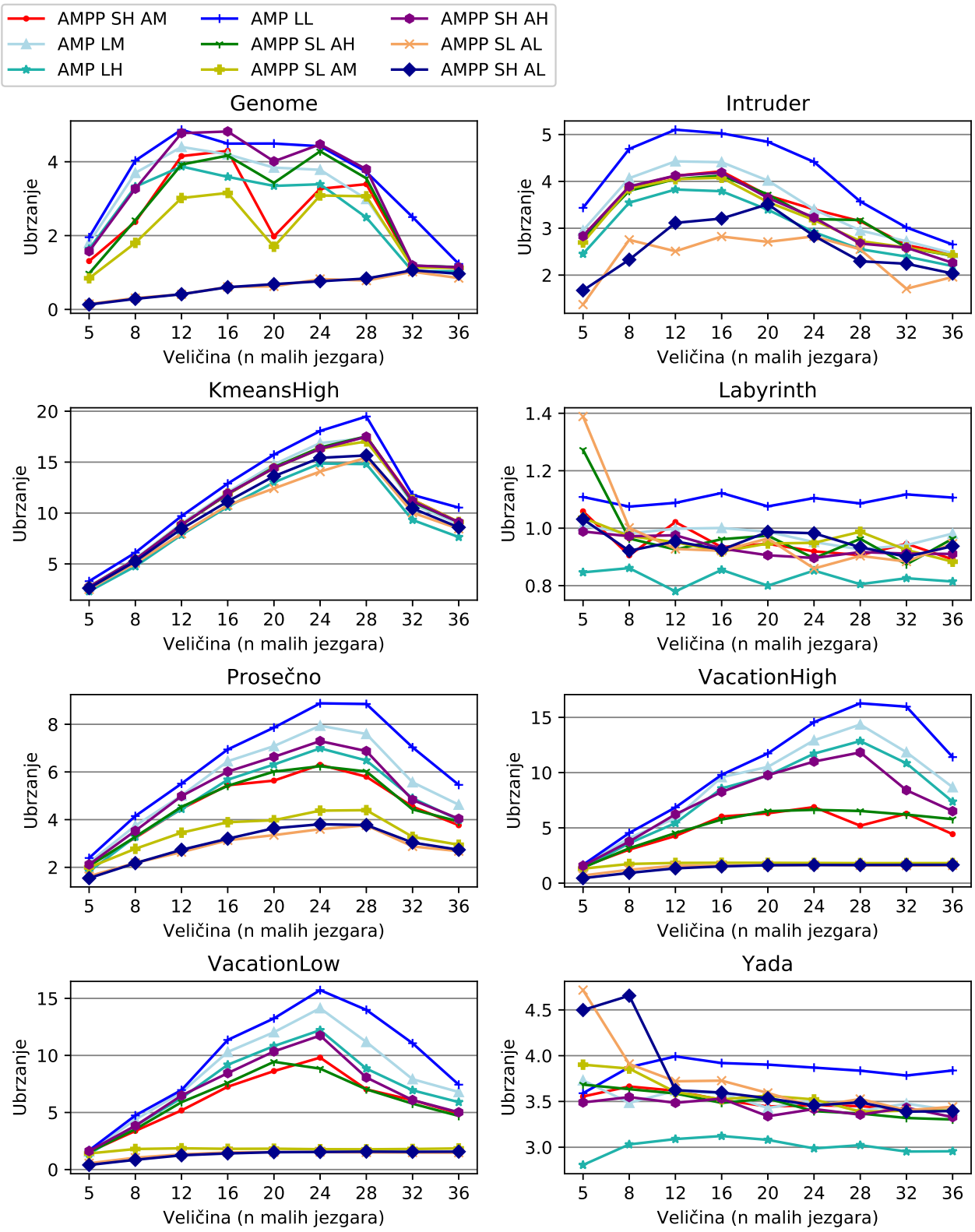
U tabelama 7.9 i 7.10 su prikazani prosečni procenti pogodaka u keš memorijama prvog i drugog nivoa za četiri konfiguracije. Prva tabela prikazuje podatke za konfiguracije SMP LM i SMP SH AH. Druga tabela prikazuje podatke za konfiguracije AMP LM i AMPP SH AH. LM konfiguracije su izabrane za prikaz, jer LL konfiguracija ima lošije vrednosti procenata od LM konfiguracije. Zbog brzine keš memorije instrukcije se brzo izvršavaju, pa ne stižu da se dovuku dovoljno brzo podaci iz viših nivoa keš memorije. SH AH konfiguracije su izabrane za prikaz, jer postižu najbolje rezultate od svih konfiguracija sa prilagođenim podsistemom keš memorija.



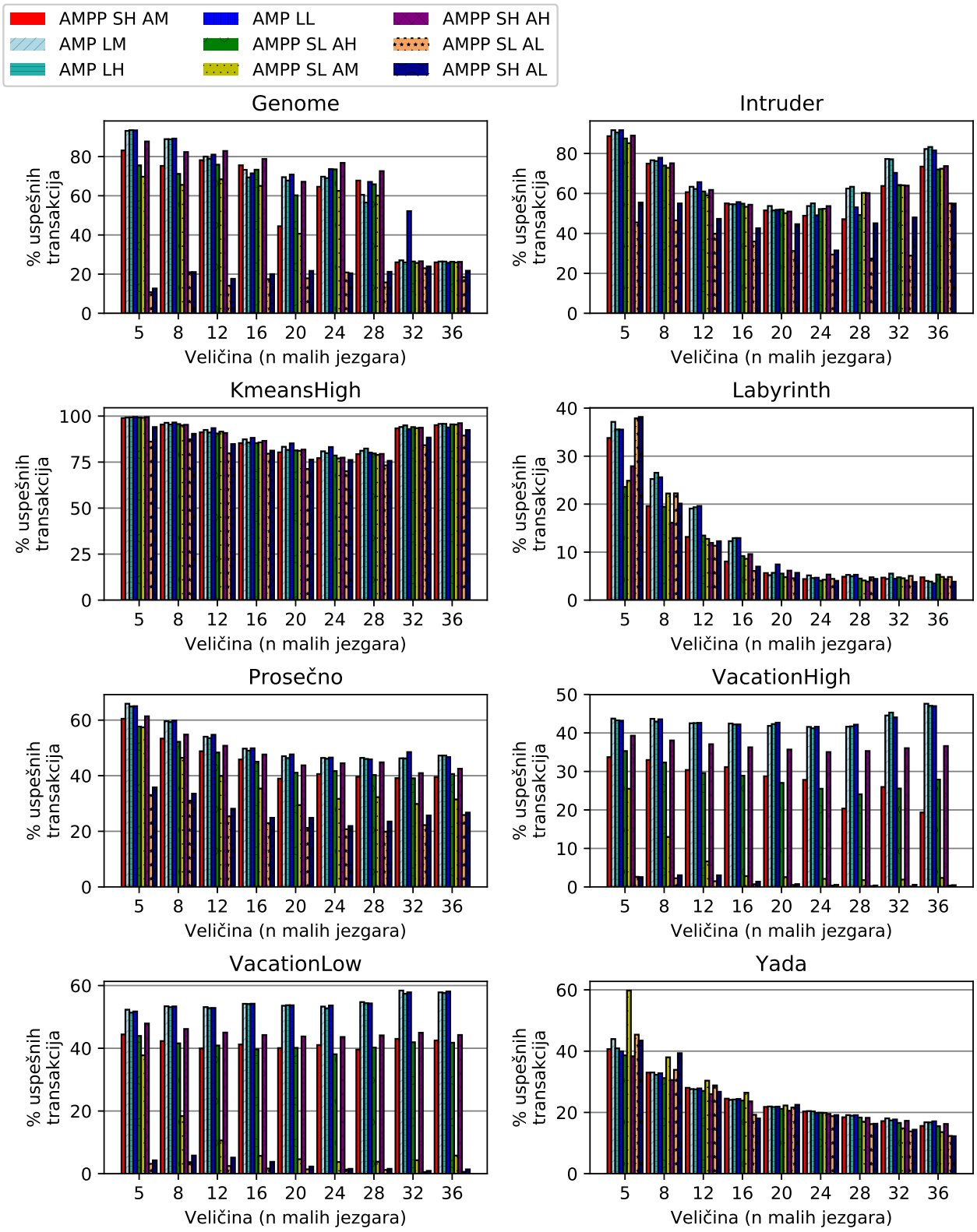
Slika 7.11: *Ubrzanje u odnosu na jedno malo jezgro za SMP konfiguracije*



Slika 7.12: *Procenat uspešnih transakcija za SMP konfiguracije*



Slika 7.13: *Ubrzanje u odnosu na jedno malo jezgro za AMP konfiguracije*



Slika 7.14: *Procenat uspešnih transakcija za AMP konfiguracije*

Tabela 7.9: *Upoređivanje prosečnih procenata pogodaka u keš memorijama po nivoima za SMP konfiguracije*

Konfiguracija	SMP		SMPP SH AH	
Aplikacija \ Nivo	L1	L2	L1	L2
Genome	99.20%	12.63%	99.14%	17.20%
Intruder	97.27%	29.19%	96.60%	38.07%
Kmeans High	99.38%	4.69%	99.30%	4.13%
Labyrinth	99.97%	44.62%	99.92%	47.39%
Vacation High	98.65%	22.09%	97.86%	44.22%
Vacation Low	98.24%	18.33%	97.37%	39.07%
Yada	99.81%	69.71%	99.65%	67.06%

Tabela 7.10: *Upoređivanje prosečnih procenata pogodaka u keš memorijama po nivoima za AMP konfiguracije*

Konfiguracija	AMP		AMPP SH AH	
Aplikacija \ Nivo	L1	L2	L1	L2
Genome	99.20%	13.60%	99.10%	16.09%
Intruder	97.37%	28.51%	96.61%	35.33%
Kmeans High	99.45%	5.39%	99.33%	4.45%
Labyrinth	99.96%	60.39%	99.90%	40.96%
Vacation High	98.98%	24.11%	98.19%	46.38%
Vacation Low	98.65%	19.61%	97.78%	40.45%
Yada	99.81%	68.16%	99.65%	65.55%

Konfiguracija SMPP SH AH ima neznatno lošije procenta pogodaka u keš memorijama prvog nivoa nego konfiguracija SMP. Što je ta razlika veća, pogoci u keš memorijama drugog nivoa se povećavaju u odnosu na konfiguraciju SMP. Jedini izuzeci od tog trenda su aplikacije Kmeans i Yada. Za njih je procenat pogodaka u keš memorijama drugog nivoa neznatno smanjen. Konfiguracije za asimetrični višejezgarni procesor pokazuju sličan trend, osim što i kod aplikacije Labyrinth konfiguracija sa podeljenim podsistemom keš memorija ima značajno niži procenat pogodaka u keš memorijama drugog nivoa.

7.2.3 Diskusija

Na osnovu rezultata simulacije zaključeno je da predloženo rešenje, pod određenim uslovima, može da poboljša performanse izvršavanja aplikacija na asimetričnom višejezgarnom procesoru sa transakcionom memorijom u odnosu na sistem sa konvencionalnom keš memorijom. S obzirom na to da konfiguracija AMPP SH AH daje bolje rezultate nego konfiguracija AMP LH do 24% (u proseku oko 9%). Da bi se dobilo poboljšanje performansi izvršavanja, prostorni deo keš memorije treba da bude napravljen tako da ima kraće vreme pristupa od konvencionalne keš memorije i mora imati istu veličinu skupa u set-asocijativnom mapiranju. Ako je tako nešto moguće onda predloženo rešenje ima bolje performanse izvršavanja i za četvrtinu manje keš memorije prva dva nivoa. Procenu kašnjenja moguće je uraditi pomoću odgovarajućih alata, kao što je CACTI [148], ali to izlazi iz okvira i ciljeva

ovog istraživanja. Procena se može uraditi u budućem istraživanju.

Ako prostorni deo i konvencionalna keš memorija imaju približno ista vremena pristupa, korišćenje predloženog rešenja je isplativo ako će na procesoru da se izvršavaju aplikacije koje imaju manje skupove podataka i koje su skalabilne sa povećanjem broja niti. Na taj način performanse izvršavanja će ostati približno iste, a uštedeće se prostor na čipu za keš memorije. U slučaju kada se planira da se na procesoru izvršavaju i aplikacije drugog tipa od navedenih, moguće je napraviti podeljenu keš memoriju promenljivom, tako da vremenski deo memorije može da se isključi, a da ostatak radi kao konvencionalna keš memorija. Na taj način se može učiniti da taj deo memorije bude isključen poluprovodnik (eng. *dark silicon*) umesto nečega drugog na čipu, jer u modernim procesorima uvek ima isključenih delova zbog ograničenja u pogledu energije i disipacije [36]. Na koji način je to moguće implementirati izlazi iz okvira ovog istraživanja.

Na osnovu rezultata, jasno se uočava da su performanse izvršavanja aplikacija sa transakcionom memorijom izuzetno osetljive na konfiguraciju keš memorija. Čim keš memorije ne mogu da čuvaju u sebi skupove podataka koji su potrebni transakcijama za izvršavanje, performanse izvršavanja značajno opadaju. Zato prostorni deo keš memorije mora da ima istu asocijativnost kao i konvencionalne keš memorije, dok vremenski deo mora da bude što je moguće veći, a da se ne ugrozi vreme pristupa.

Razlike u ubrzanjima između simetrične i asimetrične konfiguracije sa prilagođenim podsistemom keš memorija proizilaze iz te potrebe da skup podataka transakcije može da se smesti u keš memorije. Kod asimetrične konfiguracije, veliko jezgro ima veće delove keš memorija, pa više podatak može da stane i transakcije se ređe poništavaju. To doprinosi većem ubrzanju koje to jezgro postiže. Ovako prilagođeni podsistem keš memorija više koristi može da da na asimetričnim višejezgarnim procesorima nego na simetričnim.

Prilagođeni podsistem keš memorija ne utiče značajno na procente pogodaka u keš memorijama prvog nivoa, što znači da promašaji ne doprinose lošijim performansama izvršavanja. Procenat pogodaka u keš memoriji drugog nivo se značajno menja. Osnovni razlog je što je broj pristupa keš memoriji drugog nivoa nekoliko redova veličine manji nego broj pristupa kod keš memorija prvog nivo, pa svaka promena u broju pogodaka značajnije utiče na procenat. To znači da keš memorija prvog nivoa ne može da drži sve podatke koji su potrebni za izvršavanje niti pa se izbacuju u keš memoriju drugog nivoa, a ubrzo im se opet pristupa. Tome najviše doprinosu vremenski deo, koji je simuliran konzervativno. Veličina skupa kod set-asocijativnog mapiranja je uvek jedan i ako bi mogla da se poveća to bi doprinelo da se ovaj efekat smanji i na taj način poboljšaju performanse izvršavanja.

Poboljšanja koje pruža pomenuto rešenje pre implementacije rešenja treba uporediti sa komplikacijama koje implementacija može da donese. S obzirom na to da se menja protokol keš koherencije, verifikacija tih izmena može da bude kompleksna. Takođe, verifikacija može da propusti greške koje će tek biti vidljive kada se procesor proizvede i kada počne da se koristi. Pošto se keš memorije sastoje od dva dela, a jedan deo je napravljen na isti način kao i konvencionalne keš memorije, predlaže se da se prilikom implementacije doda logika za isključivanje vremenskog dela. Isključivanjem treba da se postigne da podsistem keš memorija radi na način kao da prilagođenja nije ni bilo, razume se uz duplo manje kapacitete keš memorija, jer su prostorni delovi upola manji. Na taj način može da se spreči proizvodnja neispravnog procesora. Slična stvar je urađena za jedan od prvih procesora sa podrškom za transakcionu memoriju kada je otkriveno da usled problema sa tajmingom može da dođe do nepredvidljivih rezultata [149].

Poglavlje 8

Zaključak

U ovoj disertaciji razmatrani su asimetrični višejezgarni procesori sa transakcionom memorijom. Oni su ušli u upotrebu, jer omogućavaju iskorišćenje tranzistora na čipu tako da se energetski budžet čipa ne prekorači. Transakciona memorija olakšava programiranje takvih procesora gde se teret pravilnog deljenja podatka među nitima prebacuje sa programera na hardver. Za takve procesore predložene su dve tehnike za poboljšanje performansi procesora.

Primećeno je da postoji mogućnost za ubrzanje transakcija izvršavanjem na velikom jezgru. U ovoj disertaciji se predlaže detekcija kritičnih transakcija (one koje najviše ograničavaju performanse izvršavanja). Kritične transakcije se detektuju na osnovu stope njihovog poništavanja ili potvrđivanja. Predložena je tehnika za migraciju kritičnih transakcija sa malog na veliko jezgro. Pošto tako kritična transakcija traje kraće, manje su šanse da uđe u konflikt sa drugim transakcijama. Implementacija i resursi potrebni za implementaciju tehnike su detaljno opisani. Napravljen je prototip te tehnike u simulatoru Gem5, koji vrši detaljnu simulaciju na nivou procesorskih ciklusa. Izvršena je evaluacija predloženog rešenja poređenjem sa rešenjima koja ne primenjuju tu tehniku. Više konfiguracija asimetričnog procesora je korišćeno za simulaciju. Aplikacije iz skupa STAMP su korišćene za evaluaciju predloženog rešenja. Uočeno je da predloženo rešenje poboljšava performanse izvršavanja za one aplikacije koje ispoljavaju srednji nivo paralelizma, dok za aplikacije sa visokim nivoom paralelizma ne smanjuje performanse. Za aplikacije sa niskim nivoom paralelizma u nekim slučajevima poboljšava performanse izvršavanja, ali takve aplikacije je najbolje izvršavati na drugim vrstama procesora.

Svako izvršavanje pa i izvršavanje transakcija ograničeno je brzinom pristupa podacima. U ovoj disertaciji predloženo je da se prilagodi podsistem keš memorija tako da se poveća brzina pristupa. To se radi podelom keš memorija privatnih nivoa na dva dela. Jedan deo je optimizovan za iskorišćenje prostorne lokalnosti, drugi za iskorišćenje vremenske lokalnosti. Delovi su manji od konvencionalnih keš memorija i drugi deo je jednostavniji, tako da se vreme pristupa skraćuje. S obzirom na to da se u jednom delu podaci čuvaju na nivou jedne reči, predloženo je da se keš koherencija održava na nivou reči. Dat je detaljan algoritam kako to može da se izvede. Na taj način je moguće otkrivati konflikte na finijem nivou granularnosti što doprinosi smanjenu broja lažnih konflikata. Prototip predložene tehnike je napravljen detaljno u simulatoru Gem5. Evaluacija je rađena za veliki broj konfiguracija podsistema keš memorija. Uočeno je da tehnika postiže bolje performanse izvršavanja u odnosu na slučajeve kad je nema, ako je vreme pristupa kraće do pojedinačnih delova podeljene keš memorije od konvencionalne keš memorije i ako oni delovi koji pamte ulaze na nivou blokova zadrže istu veličinu skupa kod set-asocijativnog mapiranja.

Ova disertacija ima sledeće elemente:

- Prikazuje i klasifikuje postojeća rešenja u oblasti transakcione memorije, asimetričnih procesora i podeljenog podsistema keš memorije.
- Predlaže algoritam i arhitekturu za hardversku migraciju transakcija. Algoritam koristi heuri-

stiku, koja uzima u obzir dužinu transakcije, procenat uspešnosti i razloge poništavanja transakcije.

- Predlaže arhitekturu podsistema keš memorije koji će omogućiti da se aplikacija sa transakcionom memorijom izvršava jednako brzo ili brže od izvršavanja na konvencionalnom podsistemu keš memorije. Predložena arhitektura koristi manje prostora na čipu i time se štedi prostor.

- Predstavlja implementaciju sistema za migraciju M-HTM u simulatoru Gem5. Transakciona memorija je u osnovi modelovana po rešenjima trenutno dostupnim u komercijalnim procesorima, dok je asimetrični višejezgarni procesor fiksne arhitekture.

- Predstavlja implementaciju predloženog podsistema keš memorije u simulatoru Gem5.

- Prikazuje evaluaciju predloženog rešenja za migraciju na skupu test aplikacija za transakcionu memoriju STAMP. Poboljšanja koja algoritam postiže su do 14% (u proseku 10%) ubrzanja u odnosu na rešenja koja ne uključuju postojanje asimetrije u procesoru prilikom izvršavanja transakcija

- Prikazuje evaluaciju predloženog rešenja za podsistem keš memorija na skupu test aplikacija za transakcionu memoriju STAMP. Poboljšanja koja predložena arhitektura postiže su do 24% (u proseku 9%) u odnosu na konfiguracije bez predloženog rešenja i sa smanjenom brzinama pristupa keš memorijama.

Pravci daljeg istraživanja za migraciju transakcija uključuju dalje unapređenje algoritma za određivanje kritičnih transakcija i arhitekture asimetričnih procesora. Ovde je predložen jednostavan algoritam namenjen za implementaciju u hardver. Moguće je dalje proširiti model kritičnih transakcija uključivanjem u razmatranje i ubrzanja koje bi transakcija mogla da ima izvršavanjem na velikom jezgru. Takođe, moguće je razmatrati međusobnu zavisnost transakcija u izazivanju konflikata, pa ubrzavati transakcije koje najviše utiču na poništavanje. U ovoj disertaciji razmatran je jednostavan asimetrični procesor sa jednim velikim jezgrom koji ima podršku za izvršavanje jedne niti. U komercijalno dostupnim procesorima ima više velikih jezgara sa podrškom za izvršavanje do dve niti. Potrebno je ispitati kakve performanse će dati predloženo rešenje u takvim sistemima.

Takođe, potrebno je u daljem radu proveriti kako komunikaciona mreža između keš memorija utiče na performanse sistema M-HTM. Treba razmotriti i da li prebacivanjem jednog dela algoritma u softver može da se smanji kompleksnost hardverskog dela bez značajnije degradacije performansi. Time bi se mogle uključiti dodatne heuristike, pa i metode mašinskog učenja za bolju predikciju kritičnih transakcija.

Predloženo prilagođenje podsistema keš memorija može poboljšati performanse. Budući rad treba da uključi detaljnu analizu veličine keš memorija i kolika će vremena pristupa biti za takve keš memorije. Takođe, potrebno je razmotriti kako predloženo rešenje utiče na potrošnju energije celog sistema. Manje keš memorije mogu trošiti manje energije, ali unapređen protokol keš koherencije i strukture potrebne za njegov rad mogu da povećaju potrošnju.

Literatura

[1] Z. Sustran, G. Rakocevic, and V. Milutinovic, „Dual Data Cache Systems", in *Advances in Computers*, vol. 96, 2015, pp. 187–233. DOI: 10.1016/bs.adcom.2014.11.001.

[2] Z. Sustran, S. Stojanovic, G. Rakocevic, V. Milutinovic, and M. Valero, „A survey of dual data cache systems", in *2012 IEEE International Conference on Industrial Technology, ICIT 2012, Proceedings*, 2012, ISBN: 9781467303422. DOI: 10.1109/ICIT.2012.6209979.

[3] M. Herlihy and J. E. B. Moss, „Transactional memory", *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 289–300, 1993, ISSN: 01635964. DOI: 10.1145/173682.165164.

[4] S. Mittal, „A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors", *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–38, 2016, ISSN: 03600300. DOI: 10.1145/2856125.

[5] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, „Accelerating critical section execution with asymmetric multi-core architectures", in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09*, New York, New York, USA: ACM Press, 2009, p. 253, ISBN: 9781605584065. DOI: 10.1145/1508244.1508274.

[6] T. Yu, R. Zhong, V. Janjic, *et al.*, „Collaborative Heterogeneity-Aware OS Scheduler for Asymmetric Multicore Processors", *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1224–1237, 2021, ISSN: 1045-9219. DOI: 10.1109/TPDS.2020.3045279.

[7] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley, „Portable performance on asymmetric multicore processors", in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, New York, NY, USA: ACM, 2016, pp. 24–35, ISBN: 9781450337786. DOI: 10.1145/2854038.2854047.

[8] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, „Towards completely fair scheduling on asymmetric single-ISA multicore processors", *Journal of Parallel and Distributed Computing*, vol. 102, pp. 115–131, 2017, ISSN: 07437315. DOI: 10.1016/j.jpdc.2016.12.011.

[9] A. Garcia-Garcia, J. C. Saez, and M. Prieto-Matias, „Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems", *IEEE Transactions on Computers*, vol. 67, no. 12, pp. 1703–1719, 2018, ISSN: 0018-9340. DOI: 10.1109/TC.2018.2836418.

[10] C. Kim and J. Huh, „Exploring the Design Space of Fair Scheduling Supports for Asymmetric Multicore Systems", *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1–1, 2018, ISSN: 0018-9340. DOI: 10.1109/TC.2018.2796077.

[11] K. V. Craeynest, „Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE) type I type II type III", vol. 00, no. c, pp. 213–224, 2012.

[12] N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Crista, „Thread Lock Section-Aware Scheduling on Asymmetric Single-ISA Multi-Core", *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 160–163, 2015, ISSN: 1556-6056. DOI: 10.1109/LCA.2014.2357805.

[13] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, „Bottleneck identification and scheduling in multithreaded applications", in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, vol. 40, New York, New York, USA: ACM Press, 2012, p. 223, ISBN: 9781450307598. DOI: 10.1145/2150976.2151001.

[14] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, „Utility-based acceleration of multithreaded applications on asymmetric CMPs", *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, p. 154, 2013. DOI: 10.1145/2485922.2485936.

[15] N. B. Lakshminarayana, J. Lee, and H. Kim, „Age based scheduling for asymmetric multiprocessors", in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, New York, New York, USA: ACM Press, 2009, p. 1, ISBN: 9781605587448. DOI: 10.1145/1654059.1654085.

[16] G. Scarrott, „The efficient use of multilevel storage", in *Proc. IFIPS Congress Labs*, Spartan Books, 1965, pp. 137–141.

[17] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*, 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 0123747503.

[18] A. J. Smith, „Cache Memories", *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982, ISSN: 0360-0300. DOI: 10.1145/356887.356892.

[19] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, ISBN: 9780080573076.

[20] M. Tomasevic and V. Milutinovic, „Hardware approaches to cache coherence in shared-memory multiprocessors, Part 1", *IEEE Micro*, vol. 14, no. 5, p. 52, 1994, ISSN: 0272-1732. DOI: 10.1109/MM.1994.363067.

[21] M. Tomasevic and V. Milutinovic, „Hardware approaches to cache coherence in shared-memory multiprocessors, Part 2", *IEEE Micro*, vol. 14, no. 6, pp. 61–66, 1994, ISSN: 0272-1732. DOI: 10.1109/40.331392.

[22] I. Tartalja and V. Milutinovic, „Classifying software-based cache coherence solutions", *IEEE Software*, vol. 14, no. 3, pp. 90–101, 1997, ISSN: 0740-7459. DOI: 10.1109/52.589244.

[23] B. Jacob, S. W. Ng, and D. T. Wang, „Management of Cache Consistency", in *Memory Systems*, Cdc, vol. 07, Elsevier, 2008, pp. 217–256, ISBN: 2634791631. DOI: 10.1016/B978-012379751-3.50006-0.

[24] S. Adve and K. Gharachorloo, „Shared memory consistency models: a tutorial", *Computer*, vol. 29, no. 12, pp. 66–76, 1996, ISSN: 00189162. DOI: 10.1109/2.546611.

[25] J. Protic, I. Tartalja, and M. Tomasevic, „Memory consistency models for shared memory multiprocessors and DSM systems", in *Proceedings of 8th Mediterranean Electrotechnical Conference on Industrial Applications in Power Systems, Computer Science and Telecommunications (MELECON 96)*, vol. 2, IEEE, 1996, pp. 1112–1115, ISBN: 0-7803-3109-5. DOI: 10.1109/MELCON.1996.551403.

[26] P. B. Hansen, „Concurrent Programming Concepts", *ACM Computing Surveys*, vol. 5, no. 4, pp. 223–245, 1973, ISSN: 03600300. DOI: 10.1145/356622.356624.

[27] E. W. DIJKSTRA, „Co-operating sequential processes", *Programming Languages*, pp. 43–112, 1968.

[28] J. Stone, H. Stone, P. Heidelberger, and J. Turek, „Multiple reservations and the Oklahoma update", *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 4, pp. 58–71, 1993, ISSN: 1063-6552. DOI: 10.1109/88.260295.

[29] A. Kägi, D. Burger, and J. R. Goodman, „Efficient synchronization", in *Proceedings of the 24th annual international symposium on Computer architecture - ISCA '97*, New York, New York, USA: ACM Press, 1997, pp. 170–180, ISBN: 0897919017. DOI: 10.1145/264107.264166.

[30] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, „The notions of consistency and predicate locks in a database system", *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976, ISSN: 00010782. DOI: 10.1145/360363.360369.

[31] J. Gray, „The transaction concept: Virtues and limitations", in *Seventh international conference on Very Large Data*, vol. 81, 1981, pp. 144–154.

[32] C. Blundell, E. Lewis, and M. Martin, „Subtleties of transactional memory atomicity semantics", *IEEE Computer Architecture Letters*, vol. 5, no. 2, pp. 17–17, 2006, ISSN: 1556-6056. DOI: 10.1109/L-CA.2006.18.

[33] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, „Composable memory transactions", in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '05*, vol. 51, New York, New York, USA: ACM Press, 2005, p. 48, ISBN: 1595930809. DOI: 10.1145/1065944.1065952.

[34] S. Eyerman and L. Eeckhout, „Modeling critical sections in Amdahl's law and its implications for multicore design", in *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, vol. 38, New York, New York, USA: ACM Press, 2010, p. 362, ISBN: 9781450300537. DOI: 10.1145/1815961.1816011.

[35] M. D. Hill and M. R. Marty, „Amdahl's Law in the Multicore Era", *Computer*, vol. 41, no. 7, pp. 33–38, 2008, ISSN: 0018-9162. DOI: 10.1109/MC.2008.209.

[36] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, „Dark silicon and the end of multicore scaling", *IEEE Micro*, vol. 32, no. 3, pp. 122–134, 2012, ISSN: 02721732. DOI: 10.1109/MM.2012.17.

[37] K. Van Craeynest and L. Eeckhout, „Understanding fundamental design choices in single-ISA heterogeneous multicore architectures", *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–23, 2013, ISSN: 1544-3566. DOI: 10.1145/2400682.2400691.

[38] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, „Core fusion", in *Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07*, vol. 35, New York, New York, USA: ACM Press, 2007, p. 186, ISBN: 9781595937063. DOI: 10.1145/1250662.1250686.

[39] P. Salverda and C. Zilles, „Fundamental performance constraints in horizontal fusion of in-order cores", in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, IEEE, 2008, pp. 252–263, ISBN: 978-1-4244-2070-4. DOI: 10.1109/HPCA.2008.4658644.

[40] B. Jeff, „Big. little system architecture from arm: Saving power through heterogeneous multiprocessing and task context migration.", in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1143–1146.

[41] P. Greenhalgh, *Big.LITTLE processing with ARM® Cortex-A15 & Cortex-A7*, https://www.eetimes.com/big-little-processing-with-arm-cortex-a15-cortex-a7/#, 2011 (Pristupljeno 15.10.2019.)

[42] H. Chung, M. Kang, and H.-D. Cho, *Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big.LITTLE™ Technology*, https://s3.ap-northeast-2.amazonaws.com/global.semi.static/Heterogeneous_Multi-Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf, 2013 (Pristuplenjo 03.11.2019.)

[43] R. Whitwam, *Qualcomm unveils 64-bit Snapdragon 808 and 810 SoCs: The Apple A7 stop-gap measures continue*, https://www.extremetech.com/computing/179987-qualcomm-unveils-64-bit-snapdragon-808-and-810-socs-the-apple-a7-stop-gap-measures-continue, 2014 (Pristupljeno 09.10.2019.)

[44] NVIDIA Corporation, *NVIDIA Jetson TX2 Datasheet*, https://www.assured-systems.com/uploads/media/products/axiomtek/eboxs/jetson%20tx2/data%20sheet%20-%20nvidia%20jetson%20tx2%20system-on-module.pdf, 2017 (Pristupljeno 04.05.2020.)

[45] T. Harris, A. Cristal, O. S. Unsal, *et al.*, „Transactional memory: An overview", *IEEE Micro*, vol. 27, no. 3, pp. 8–20, 2007, ISSN: 02721732. DOI: 10.1109/MM.2007.63.

[46] T. Harris, J. Larus, and R. Rajwar, „Transactional Memory, 2nd edition", *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010, ISSN: 1935-3235. DOI: 10.2200/S00272ED1V01Y201006CAC011.

[47] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie, „Unbounded Transactional Memory", in *11th International Symposium on High-Performance Computer Architecture*, IEEE, 2005, pp. 316–327, ISBN: 0-7695-2275-0. DOI: 10.1109/HPCA.2005.41.

[48] K. E. Moore, M. D. Hill, and D. A. Wood, „Thread-Level Transactional Memory", University of Wisconsin, Tech. Rep., 2005, pp. 1–11.

[49] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, „LogTM: Log-based Transactional Memory", in *The Twelfth International Symposium on High-Performance Computer Architecture*, IEEE, 2006, pp. 258–269, ISBN: 0-7803-9368-6. DOI: 10.1109/HPCA.2006.1598134.

[50] L. Yen, J. Bobba, M. R. Marty, *et al.*, „LogTM-SE: Decoupling Hardware Transactional Memory from Caches", in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, IEEE, 2007, pp. 261–272, ISBN: 1-4244-0804-0. DOI: 10.1109/HPCA.2007.346204.

[51] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel, „TxLinux: Using and Managing Hardware Transactional Memory in an Operating System", in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP '07*, New York, New York, USA: ACM Press, 2007, p. 87, ISBN: 9781595935915. DOI: 10.1145/1294261.1294271.

[52] L. Hammond, V. Wong, M. Chen, *et al.*, „Transactional memory coherence and consistency", in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, vol. 32, IEEE, 2004, pp. 102–113, ISBN: 0-7695-2143-6. DOI: 10.1109/ISCA.2004.1310767.

[53] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, „Bulk Disambiguation of Speculative Threads in Multiprocessors", in *33rd International Symposium on Computer Architecture (ISCA'06)*, vol. 2006, IEEE, 2006, pp. 227–238, ISBN: 0-7695-2608-X. DOI: 10.1109/ISCA.2006.13.

[54] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, „Architectural Support for Software Transactional Memory", in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, IEEE, 2006, pp. 185–196, ISBN: 0-7695-2732-9. DOI: 10.1109/MICRO.2006.9.

[55] A. Shriraman, V. J. Marathe, S. Dwarkadas, *et al.*, „Hardware acceleration of software transactional memory", Computer Science Department, University of Rochester, Tech. Rep., 2006.

[56] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, „BulkSC", in *Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07*, New York, New York, USA: ACM Press, 2007, pp. 278–290, ISBN: 9781595937063. DOI: 10.1145/1250662.1250697.

[57] E. Vallejo, M. GalIuzzi, A. CristaI, *et al.*, „Implementing kilo-instruction multiprocessors", in *ICPS '05. Proceedings. International Conference on Pervasive Services, 2005.*, vol. 2005, IEEE, 2005, pp. 325–336, ISBN: 0-7803-9032-6. DOI: 10.1109/PERSER.2005.1506430.

[58] P. Rundberg and P. Stenstrom, „Speculative lock reordering: optimistic out-of-order execution of critical sections", in *Proceedings International Parallel and Distributed Processing Symposium*, IEEE Computer Society, 2003, p. 8, ISBN: 0-7695-1926-1. DOI: 10.1109/IPDPS.2003.1213086.

[59] Ravi Rajwar, „Speculative Lock Elision:: Enabling Highly Concurrent Multithreaded Execution", *Sciences-New York*, pp. 294–305, 2001. DOI: 10.1145/570000.

[60] J. F. Martínez and J. Torrellas, „Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications", *10th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, pp. 18–29, 2002, ISSN: 0362-1340. DOI: 10.1145/605397.605400.

[61] R. Rajwar, M. Herlihy, and Konrad Lai, „Virtualizing Transactional Memory", in *32nd International Symposium on Computer Architecture (ISCA'05)*, IEEE, 2005, pp. 494–505, ISBN: 0-7695-2270-X. DOI: 10.1109/ISCA.2005.54.

[62] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, „Hybrid transactional memory", in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '06*, New York, New York, USA: ACM Press, 2006, pp. 209–220, ISBN: 1595931899. DOI: 10.1145/1122971.1123003.

[63] R. Rajwar and J. R. Goodman, „Transactional lock-free execution of lock-based programs", in *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X) - ASPLOS '02*, New York, New York, USA: ACM Press, 2002, p. 5, ISBN: 1581135742. DOI: 10.1145/605397.605399.

[64] W. Chuang, S. Narayanasamy, G. Venkatesh, *et al.*, „Unbounded page-based transactional memory", in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII*, New York, New York, USA: ACM Press, 2006, p. 347, ISBN: 1595934510. DOI: 10.1145/1168857.1168901.

[65] J. P. Singh, W.-D. Weber, and A. Gupta, „SPLASH", *ACM SIGARCH Computer Architecture News*, vol. 20, no. 1, pp. 5–44, 1992, ISSN: 01635964. DOI: 10.1145/130823.130824.

[66] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, „The SPLASH-2 programs: characterization and methodological considerations", in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ACM, 1995, pp. 24–36, ISBN: 0-89791-698-0. DOI: 10.1109/ISCA.1995.524546.

[67] B. Bershad, „Practical considerations for non-blocking concurrent objects", in *Proceedings of the 13th International Conference on Distributed Computing Systems*, IEEE Comput. Soc. Press, 1991, pp. 264–273, ISBN: 0-8186-3770-6. DOI: 10.1109/ICDCS.1993.287700.

[68] M. Herlihy, „A methodology for implementing highly concurrent data objects", *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, 1993, ISSN: 01640925. DOI: 10.1145/161468.161469.

[69] B. H. Bloom, „Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970, ISSN: 00010782. DOI: 10.1145/362686.362692.

[70] P. Damron, A. Fedorova, and Y. Lev, „Hybrid transactional memory", in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII*, New York, New York, USA: ACM Press, 2006, p. 336, ISBN: 1595934510. DOI: 10.1145/1168857.1168900.

[71] C. C. Minh, M. Trautmann, J. Chung, *et al.*, „An effective hybrid transactional memory system with strong isolation guarantees", in *Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07*, vol. 35, New York, New York, USA: ACM Press, 2007, pp. 69–81, ISBN: 9781595937063. DOI: 10.1145/1250662.1250673.

[72] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, „MetaTM/TxLinux", in *Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07*, vol. 28, New York, New York, USA: ACM Press, 2007, p. 92, ISBN: 9781595937063. DOI: 10.1145/1250662.1250675.

[73] C. Zilles and L. Baugh, „Extending hardware transactional memory to support non-busy waiting and non-transactional actions", in *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Canada, 2006.

[74] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, „Software transactional memory for dynamic-sized data structures", in *Proceedings of the twenty-second annual symposium on Principles of distributed computing - PODC '03*, New York, New York, USA: ACM Press, 2003, pp. 92–101, ISBN: 1581137087. DOI: 10.1145/872035.872048.

[75] Y. Lev, M. Moir, and D. Nussbaum, „PhTM: Phased transactional memory", in *In Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[76] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, „Implementing signatures for transactional memory", *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 123–133, 2007, ISSN: 10724451. DOI: 10.1109/MICRO.2007.24.

[77] R. Pagh and F. F. Rodler, „Cuckoo Hashing", in *IEEE Transactions on Knowledge and Data Engineering*, 4, vol. 14, 2001, pp. 121–133, ISBN: 978-0-387-30162-4. DOI: 10.1007/3-540-44676-1_10.

[78] I. Vurdelja, Z. Sustran, J. Protic, and D. Draskovic, „Survey of machine learning application in transactional memory", in *2020 28th Telecommunications Forum, TELFOR 2020 - Proceedings*, 2020, ISBN: 9780738142425. DOI: 10.1109/TELFOR51502.2020.9306547.

[79] R. M. Yoo and H.-H. S. Lee, „Adaptive transaction scheduling for transactional memory systems", in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures - SPAA '08*, New York, New York, USA: ACM Press, 2008, p. 169, ISBN: 9781595939739. DOI: 10.1145/1378533.1378564.

[80] H. E. Ramadan, C. J. Rossbach, and E. Witchel, „Dependence-aware transactional memory for increased concurrency", in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2008, pp. 246–257, ISBN: 978-1-4244-2836-6. DOI: 10.1109/MICRO.2008.4771795.

[81] M. Ansari, B. Khan, M. Luján, C. Kotselidis, C. Kirkham, and I. Watson, „Improving Performance by Reducing Aborts in Hardware Transactional Memory", in *High Performance Embedded Architectures and Compilers*, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–49, ISBN: 978-3-642-11515-8.

[82] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, „Preventing versus curing", in *Proceedings of the 28th ACM symposium on Principles of distributed computing - PODC '09*, New York, New York, USA: ACM Press, 2009, p. 7, ISBN: 9781605583969. DOI: 10.1145/1582716.1582725.

[83] G. Blake, R. G. Dreslinski, and T. Mudge, „Proactive transaction scheduling for contention management", in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*, New York, New York, USA: ACM Press, 2009, p. 156, ISBN: 9781605587981. DOI: 10.1145/1669112.1669133.

[84] G. Blake, R. G. Dreslinski, and T. Mudge, „Bloom Filter Guided Transaction Scheduling", in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, IEEE, 2011, pp. 75–86, ISBN: 978-1-4244-9432-3. DOI: 10.1109/HPCA.2011.5749718.

[85] I. Corporation, „Intel® 64 and IA-32 Architectures Software Developer's Manual", in vol. 1, Santa Clara, CA, USA: Intel Corporation, 2018, ch. Programming with Intel® Transactional Synchronization Extensions, pp. 385–392.

[86] Y. Afek, A. Levy, and A. Morrison, „Software-improved hardware lock elision", in *Proceedings of the 2014 ACM symposium on Principles of distributed computing - PODC '14*, New York, New York, USA: ACM Press, 2014, pp. 212–221, ISBN: 9781450329446. DOI: 10.1145/2611462.2611482.

[87] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir, „Adaptive integration of hardware and software lock elision techniques", in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, New York, NY, USA: ACM, 2014, pp. 188–197, ISBN: 9781450328210. DOI: 10.1145/2612669.2612696.

[88] N. Diegues and P. Romano, „Self-tuning Intel Restricted Transactional Memory", *Parallel Computing*, vol. 50, pp. 25–52, 2015, ISSN: 01678191. DOI: 10.1016/j.parco.2015.10.001.

[89] T. L. Lai and H. Robbins, „Asymptotically efficient adaptive allocation rules", *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[90] S. Issa, P. Felber, A. Matveev, and P. Romano, „Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume", *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 91, no. 28, pp. 1–16, 2017, ISSN: 18688969. DOI: 10.4230/LIPIcs.DISC.2017.28.

[91] B. T. Polyak, „Some methods of speeding up the convergence of iteration methods", *USSR computational mathematics and mathematical physics*, vol. 4, no. 5, pp. 1–17, 1964.

[92] Z. Chen, A. Hassan, M. J. Kishi, J. Nelson, and R. Palmieri, „HATS: Hardware-assisted Transaction Scheduler", *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 153, no. 10, pp. 1–10, 2020, ISSN: 18688969. DOI: 10.4230/LIPIcs.OPODIS.2019.10.

[93] L. Xiang and M. L. Scott, „Conflict Reduction in Hardware Transactions Using Advisory Locks", in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures - SPAA '15*, New York, New York, USA: ACM Press, 2015, pp. 234–243, ISBN: 9781450335881. DOI: 10.1145/2755573.2755577.

[94] C. LATTNER, „Macroscopic data structure analysis and optimization", Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[95] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar, „Wait-n-GoTM", in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, New York, New York, USA: ACM Press, 2013, p. 521, ISBN: 9781450318709. DOI: 10.1145/2451116.2451173.

[96] M. Mohamedin, R. Palmieri, and B. Ravindran, „On Scheduling Best-Effort HTM Transactions", in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA: ACM, 2015, pp. 74–76, ISBN: 9781450335881. DOI: 10.1145/2755573.2755612.

[97] N. Diegues, P. Romano, and S. Garbatov, „Seer: Probabilistic scheduling for Hardware Transactional Memory", *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, vol. 2015-June, no. June, pp. 224–233, 2015. DOI: 10.1145/2755573.2755578.

[98] N. Diegues, P. Romano, and S. Garbatov, „Seer", *ACM Transactions on Computer Systems*, vol. 35, no. 3, pp. 1–41, 2017, ISSN: 07342071. DOI: 10.1145/3132036.

[99] A. Armejach, A. Negi, A. Cristal, O. Unsal, P. Stenstrom, and T. Harris, „HARP: Adaptive abort recurrence prediction for Hardware Transactional Memory", in *20th Annual International Conference on High Performance Computing*, IEEE, 2013, pp. 196–205, ISBN: 978-1-4799-0730-4. DOI: 10.1109/HiPC.2013.6799100.

[100] J. Chung, L. Yen, S. Diestelhorst, *et al.*, „ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory", *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 39–50, 2010. DOI: 10.1109/MICRO.2010.40.

[101] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, „Performance evaluation of Intel® transactional synchronization extensions for high-performance computing", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, New York, New York, USA: ACM Press, 2013, pp. 1–11, ISBN: 9781450323789. DOI: 10.1145/2503210.2503232.

[102] C. Jacobi, T. Slegel, and D. Greiner, „Transactional Memory Architecture and Implementation for IBM System Z", *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 25–36, 2012. DOI: 10.1109/MICRO.2012.12.

[103] S. Chaudhry, R. Cypher, M. Ekman, *et al.*, „Rock: A High-Performance Sparc CMT Processor", *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009, ISSN: 0272-1732. DOI: 10.1109/MM.2009.34.

[104] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, „Scheduling heterogeneous multi-cores through performance impact estimation (PIE)", in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2012, pp. 213–224, ISBN: 978-1-4673-0476-4. DOI: 10.1109/ISCA.2012.6237019.

[105] Q. Chen, Y. Chen, Z. Huang, and M. Guo, „WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures", in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IEEE, 2012, pp. 249–260, ISBN: 978-1-4673-0975-2. DOI: 10.1109/IPDPS.2012.32.

[106] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, „Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures", in *Proceedings of the 29th ACM on International Conference on Supercomputing*, New York, NY, USA: ACM, 2015, pp. 329–338, ISBN: 9781450335591. DOI: 10.1145/2751205.2751235.

[107] M. Han, J. Park, and W. Baek, „Design and Implementation of a Criticality- and Heterogeneity-Aware Runtime System for Task-Parallel Applications", *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1117–1132, 2021, ISSN: 1045-9219. DOI: 10.1109/TPDS.2020.3031911.

[108] C. Torng, M. Wang, and C. Batten, „Asymmetry-Aware Work-Stealing Runtimes", in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 40–52, ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.14.

[109] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, „Fairness-aware scheduling on single-ISA heterogeneous multi-cores", in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2013, pp. 177–187, ISBN: 978-1-4799-1018-2. DOI: 10.1109/PACT.2013.6618815.

[110] Ž. Šuštran, „Sistemi sa podeljenom keš memorijom", Master teza, Univerzitet u Beogradu, Elektrotehnički fakultet, 2012.

[111] A. González, C. Aliagas, and M. Valero, „A data cache with multiple caching strategies tuned to different types of locality", in *Proceedings of the 9th international conference on Supercomputing - ICS '95*, New York, New York, USA: ACM Press, 1995, pp. 338–347, ISBN: 0897917286. DOI: 10.1145/224538.224622.

[112] V Milutinovic, M Tomasevic, B. Markovic, and M Tremblay, „The split temporal/spatial cache: initial performance analysis", in *Proceedings of the SCIzzL-5 Workshop*, 1996, pp. 63–69.

[113] J. Sahuquillo and A. Pont, „The split data cache in multiprocessor systems: an initial hit ratio analysis", in *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing. PDP'99*, IEEE, 1999, pp. 27–34, ISBN: 0-7695-0059-5. DOI: 10.1109/EMPDP.1999.746641.

[114] J. Sahuquillo, S. Petit, A. Pont, and V. Milutinović, „Exploring the performance of split data cache schemes on superscalar processors and symmetric multiprocessors", *Journal of Systems Architecture*, vol. 51, no. 8, pp. 451–469, 2005, ISSN: 13837621. DOI: 10.1016/j.sysarc.2004.12.002.

[115] M. Qureshi and Y. Patt, „Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, IEEE, 2006, pp. 423–432, ISBN: 0-7695-2732-9. DOI: 10.1109/MICRO.2006.49.

[116] X. Jiang, A. Mishra, L. Zhao, *et al.*, „ACCESS: Smart scheduling for asymmetric cache CMPs", in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, IEEE, 2011, pp. 527–538, ISBN: 978-1-4244-9432-3. DOI: 10.1109/HPCA.2011.5749757.

[117] Xiaoxia Wu, Jian Li, Lixin Zhang, E. Speight, and Yuan Xie, „Power and performance of read-write aware Hybrid Caches with non-volatile memories", in *2009 Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2009, pp. 737–742, ISBN: 978-1-4244-3781-8. DOI: 10.1109/DATE.2009.5090762.

[118] S. Agarwal and H. K. Kapoor, „Reuse-Distance-Aware Write-Intensity Prediction of Dataless Entries for Energy-Efficient Hybrid Caches", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 1881–1894, 2018, ISSN: 1063-8210. DOI: 10.1109/TVLSI.2018.2836156.

[119] B. de Abreu Silva, L. A. Cuminato, and V. Bonato, „Reducing the overall cache miss rate using different cache sizes for Heterogeneous Multi-core Processors", in *2012 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2012, pp. 1–6, ISBN: 978-1-4673-2921-7. DOI: 10.1109/ReConFig.2012.6416783.

[120] T. Okamoto, T. Nakabayashi, T. Sasaki, and T. Kondo, „FabCache: Cache Design Automation for Heterogeneous Multi-core Processors", in *2013 First International Symposium on Computing and Networking*, IEEE, 2013, pp. 602–606, ISBN: 978-1-4799-2796-8. DOI: 10.1109/CANDAR.2013.108.

[121] L. Nai and H.-H. S. Lee, „Reducing False Transactional Conflicts with Speculative Sub-Blocking State – An Empirical Study for ASF Transactional Memory System", in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, IEEE, 2013, pp. 1879–1888, ISBN: 978-0-7695-4979-8. DOI: 10.1109/IPDPSW.2013.113.

[122] Y. Futamase, M. Hayashi, T. Tajimi, R. Shioya, M. Goshima, and T. Tsumura, „An Analysis and a Solution of False Conflicts for Hardware Transactional Memory", in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, IEEE, 2018, pp. 529–532, ISBN: 978-1-5386-9562-3. DOI: 10.1109/ICECS.2018.8617977.

[123] M. Becchi and P. Crowley, „Dynamic thread assignment on heterogeneous multiprocessor architectures", in *Proceedings of the 3rd conference on Computing frontiers - CF '06*, vol. 10, New York, New York, USA: ACM Press, 2006, p. 29, ISBN: 1595933026. DOI: 10.1145/1128022.1128029.

[124] K. Yeager, „The Mips R10000 superscalar microprocessor", *IEEE Micro*, vol. 16, no. 2, pp. 28–41, 1996, ISSN: 02721732. DOI: 10.1109/40.491460.

[125] P. Kongetira, K. Aingaran, and K. Olukotun, „Niagara: A 32-Way Multithreaded Sparc Processor", *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005, ISSN: 0272-1732. DOI: 10.1109/MM.2005.35.

[126] A. Ahmed, P. Conway, B. Hughes, and F. Weber, „Amd opteron shared memory mp systems", in *Proceedings of the 14th HotChips Symposium*, 2002.

[127] D. Bossen, J. Tendler, and K. Reick, „Power4 system design for high reliability", *IEEE Micro*, vol. 22, no. 2, pp. 16–24, 2002, ISSN: 0272-1732. DOI: 10.1109/MM.2002.997876.

[128] R. Kalla, B. Sinharoy, and J. Tendler, „Simultaneous multi-threading implementation in power5", in *Conference Record of Hot Chips 15 Symposium*, 2003.

[129] M. S. Papamarcos and J. H. Patel, „A low-overhead coherence solution for multiprocessors with private cache memories", in *Proceedings of the 11th annual international symposium on Computer architecture - ISCA '84*, New York, New York, USA: ACM Press, 1984, pp. 348–354, ISBN: 0818605383. DOI: 10.1145/800015.808204.

[130] P. Sweazey and A. J. Smith, „A class of compatible cache consistency protocols and their support by the ieee futurebus", in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86, Tokyo, Japan: IEEE Computer Society Press, 1986, 414–423, ISBN: 081860719X.

[131] S. Inc., „Opensparc t2 system-on-chip (soc) microarchitecture specification", in Santa Clara, CA, USA: SunMicrosystems Inc., 2008.

[132] A. Adir, D. Goodman, D. Hershcovich, *et al.*, „Verification of transactional memory in power8", in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2014, pp. 1–6.

[133] N. Binkert, S. Sardashti, R. Sen, *et al.*, „The gem5 simulator", *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, 2011, ISSN: 01635964. DOI: 10.1145/2024716.2024718.

[134] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, „Accuracy evaluation of GEM5 simulator system", in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, IEEE, 2012, pp. 1–7, ISBN: 978-1-4673-2572-1. DOI: 10.1109/ReCoSoC.2012.6322869.

[135] N. Binkert, *About gem5 simulator*, https://www.gem5.org/about/, 2021 (Pristupljeno 10.02.2021.)

[136] I. Corporation, „Intel® 64 and IA-32 Architectures Optimization Reference Manual", in Santa Clara, CA, USA: Intel Corporation, 2016, ch. INTEL® TSX RECOMMENDATIONS, pp. 465–494.

[137] S. Balakrishnan, R. Rajwar, M. Upton, and Konrad Lai, „The Impact of Performance Asymmetry in Emerging Multicore Architectures", in *32nd International Symposium on Computer Architecture (ISCA'05)*, vol. 33, IEEE, 2005, pp. 506–517, ISBN: 0-7695-2270-X. DOI: 10.1109/ISCA.2005.51.

[138] I. Corporation, „Pentium® Processor User's Manual Volume 1: Pentium® Processor Data Book", in 1993.

[139] S. Gochman, „The Intel® Pentium® M processor: Microarchitecture and performance", *Intel Technology Journal*, vol. 7, no. 2, pp. 21–36, 2003.

[140] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun, „STAMP: Stanford Transactional Applications for Multi-Processing", in *2008 IEEE International Symposium on Workload Characterization*, IEEE, 2008, pp. 35–46, ISBN: 978-1-4244-2777-2. DOI: 10.1109/IISWC.2008.4636089.

[141] I. Corporation, „Intel® 64 and IA-32 Architectures Optimization Reference Manual", in vol. 1, 2016, ch. Programming with Intel® Transactional Synchronization Extensions, pp. 385–392.

[142] N. Diegues and P. Romano, „Self-Tuning Intel Transactional Synchronization Extensions", in *11th International Conference on Autonomic Computing*, USENIX, 2014, pp. 1–11, ISBN: 978-1-931971-11-9.

[143] Z. Sustran and J. Protic, „Migration in Hardware Transactional Memory on Asymmetric Multiprocessor", *IEEE Access*, vol. 9, pp. 69 346–69 364, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3077539.

[144] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, „Heterogeneous chip multiprocessors", *Computer*, vol. 38, no. 11, pp. 32–38, 2005, ISSN: 00189162. DOI: 10.1109/MC.2005.379.

[145] T. Morad, U. Weiser, A. Kolodny, M. Valero, and E. Ayguade, „Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors", *IEEE Computer Architecture Letters*, vol. 5, no. 1, pp. 4–4, 2006, ISSN: 1556-6056. DOI: 10.1109/L-CA.2006.6.

[146] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, „Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory", *9th Workshop on Transactional Computing (TRANSACT'14)*, 2014.

[147] R. Quislant, E. Gutierrez, E. L. Zapata, and O. Plata, „Conflict Detection in Hardware Transactional Memory", in *"Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*, 2015, pp. 127–149.

[148] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, „Cacti 6.0: A tool to model large caches", *HP laboratories*, vol. 1, pp. 1–24, 2009.

[149] I. Corporation, *E3-1200 v3 Product Family Specification Update August 2020 Revision 025*, August. Intel Corporation, 2020.

Biografija

Živojin Šuštran je rođen 05. novembra 1987. godine u Beogradu, gde je završio osnovnu školu i Trinaestu beogradsku gimnaziju. Tokom pohađanja osnovne i srednje škole učestvovao je i osvajao nagrade na mnogobrojnim takmičenjima iz hemije, matematike i fizike. Bio je prvi i treći na republičkom takmičenju iz hemije 2004. i 2005. godine. Osvojio je srebrnu medalju na balkanijadi iz hemije 2005. godine u Bukureštu. Nakon završetka osnovnog i srednjeg obrazovanja nagrađen je Vukovim diplomama.

Elektrotehnički fakultet u Beogradu upisao je 2006. godine, sa maksimalnim brojem bodova na prijemnom ispitu. Diplomirao je 14. oktobra 2010. godine na Odseku za računarsku tehniku i informatiku, sa prosečnom ocenom 9,71. Diplomski rad pod naslovom "Parser OQL upita" izradio je pod mentorstvom prof. dr Dragana Milićeva i odbranio ga sa ocenom 10.

Master akademske studije je upisao 2010. godine na Elektrotehničkom fakultetu u Beogradu, na modulu Računarska tehnika i informatika. Studije je završio 25. septembra 2012. godine sa prosečnom ocenom 10,00, odbranom master rada "Sistemi sa podeljenom keš memorijom" kod mentora prof. dr Veljka Milutinovića.

Doktorske akademske studije je upisao u novembru 2012. godine na Elektrotehničkom fakultetu u Beogradu. Do sada je ostvario 120 ESPB poena, odnosno položio je sve ispite predviđene studijskim programom sa ocenom 10.

Kao koautor je objavio tri rada u međunarodnim časopisima sa impakt faktorom, jedan rad u domaćem časopisu i više radova na konferencijama. Učestvovao je na projektu Ministarstva nauke pod nazivom "Razvoj novih informaciono-komunikacionih tehnologija, korišćenjem naprednih matematičkih metoda, sa primenama u medicini, telekomunikacijama, energetici, zaštititi nacionalne baštine i obrazovanju". Tokom doktorskih studuja 2013. godine održao je kurs na Fakultetu za računarstvo, Univerzitet u Ljubljani pod nazivom "Programiranje superračunara", kao i tutorijal pod nazivom "Control-Flow versus Data-Flow Supercomputers, and the Maxeler Programming Paradigm" na ISCA konferenciji u Tel Avivu 2013. godine. Recenzirao je radove na konferencijama ETRAN i TELFOR.

Usavršavao se na stručnoj praksi u Maxeler Technologies London u periodu od 5. jula 2016. godine do 6. oktobra 2016. godine, gde je radio na razvoju procesora specijalne namene na FPGA čipovima.

Bio je stipendista grada Beograda 2009. godine.

Od maja 2011. godine angažovan je pri Katedri za računarsku tehniku i informatiku, gde je prvi put izabran u zvanje saradnika u nastavi. U zvanje asistenta prethodni put je izabran 1. februara 2017. godine. Bio je angažovan u nastavi na 9 različitih predmeta, koji se izvode na oba studijska programa, Elektrotehnika i računarstvo i Softversko inženjerstvo. Izvodio vežbe na tabli i laboratorijske vežbe za izuzetno veliki broj studenata na sve četiri godine studija. Kao asistent na Elektrotehničkom fakultetu bio je angažovan na sledećim predmetima: Programiranje 1, Programiranje 2, Praktikum iz programiranja 1, Praktikum iz programiranja 2, Operativni sistemi 1, Operativni sistemi 2, Praktikum iz operativnih sistema, Računarski VLSI sistemi i Projektovanje softvera. Aktivno je učestvovao u osmišljavanju i postavljanju novih vežbi iz predmeta Praktikum iz operativnih sistema.

Izjava o autorstvu

Ime i prezime autora Živojin Šuštran

Broj indeksa 2012/5032

Izjavljujem

da je doktorska disertacija pod naslovom

Poboljšanje performansi asimetričnih višejezgarnih procesora kroz migraciju transakcija i prilagođenje podsistema keš memorija

- rezultat sopstvenog istraživačkog rada;
- da disertacija u celini ni u delovima nije bila predložena za sticanje druge diplome prema studijskim programima drugih visokoškolskih ustanova;
- da su rezultati korektno navedeni i
- da nisam kršio/la autorska prava i koristio/la intelektualnu svojinu drugih lica.

Potpis autora

U Beogradu, 21.06.2021. godine

Ž. Šuštran

Izjava o istovetnosti štampane i elektronske verzije doktorskog rada

Ime i prezime autora Živojin Šuštran

Broj indeksa 2012/5032

Studijski program Elektrotehnika i računarstvo

Naslov rada Poboljšanje performansi asimetričnih višejezgarnih procesora kroz migraciju transakcija i prilagođenje podsistema keš memorija

Mentor prof. dr Jelica Protić i prof. dr Milo Tomašević

Izjavljujem da je štampana verzija mog doktorskog rada istovetna elektronskoj verziji koju sam predao/la radi pohranjena u **Digitalnom repozitorijumu Univerziteta u Beogradu.**

Dozvoljavam da se objave moji lični podaci vezani za dobijanje akademskog naziva doktora nauka, kao što su ime i prezime, godina i mesto rođenja i datum odbrane rada.

Ovi lični podaci mogu se objaviti na mrežnim stranicama digitalne biblioteke, u elektronskom katalogu i u publikacijama Univerziteta u Beogradu.

Potpis autora

U Beogradu, 21.06.2021. godine

Ž. Šuštran

Izjava o korišćenju

Ovlašćujem Univerzitetsku biblioteku „Svetozar Marković“ da u Digitalni repozitorijum Univerziteta u Beogradu unese moju doktorsku disertaciju pod naslovom:

Poboljšanje performansi asimetričnih višejezgarnih procesora kroz migraciju transakcija i prilagođenje podsistema keš memorija

koja je moje autorsko delo.

Disertaciju sa svim prilozima predao/la sam u elektronskom formatu pogodnom za trajno arhiviranje.

Moju doktorsku disertaciju pohranjenu u Digitalnom repozitorijumu Univerziteta u Beogradu i dostupnu u otvorenom pristupu mogu da koriste svi koji poštuju odredbe sadržane u odabranom tipu licence Kreativne zajednice (Creative Commons) za koju sam se odlučio/la.

1. Autorstvo (CC BY)

2. Autorstvo – nekomercijalno (CC BY-NC)

3. Autorstvo – nekomercijalno – bez prerada (CC BY-NC-ND)

4. Autorstvo – nekomercijalno – deliti pod istim uslovima (CC BY-NC-SA)

5. Autorstvo – bez prerada (CC BY-ND)

6. Autorstvo – deliti pod istim uslovima (CC BY-SA)

(Molimo da zaokružite samo jednu od šest ponuđenih licenci.
Kratak opis licenci je sastavni deo ove izjave).

Potpis autora

U Beogradu, 21.06.2021. godine

Н. Шуштран

1. **Autorstvo**. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence, čak i u komercijalne svrhe. Ovo je najslobodnija od svih licenci.

2. **Autorstvo – nekomercijalno**. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca ne dozvoljava komercijalnu upotrebu dela.

3. **Autorstvo – nekomercijalno – bez prerada**. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, bez promena, preoblikovanja ili upotrebe dela u svom delu, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca ne dozvoljava komercijalnu upotrebu dela. U odnosu na sve ostale licence, ovom licencom se ograničava najveći obim prava korišćenja dela.

4. **Autorstvo – nekomercijalno – deliti pod istim uslovima**. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence i ako se prerada distribuira pod istom ili sličnom licencom. Ova licenca ne dozvoljava komercijalnu upotrebu dela i prerada.

5. **Autorstvo – bez prerada**. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, bez promena, preoblikovanja ili upotrebe dela u svom delu, ako se navede ime autora na način određen od strane autora ili davaoca licence. Ova licenca dozvoljava komercijalnu upotrebu dela.

6. **Autorstvo – deliti pod istim uslovima**. Dozvoljavate umnožavanje, distribuciju i javno saopštavanje dela, i prerade, ako se navede ime autora na način određen od strane autora ili davaoca licence i ako se prerada distribuira pod istom ili sličnom licencom. Ova licenca dozvoljava komercijalnu upotrebu dela i prerada. Slična je softverskim licencama, odnosno licencama otvorenog koda.